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

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

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

Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh PA 15123

CHI 2003 Tutorial

Table of Contents

Agenda

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

Instructor Biographies

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

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

Objectives of the course

Participants in this tutorial will

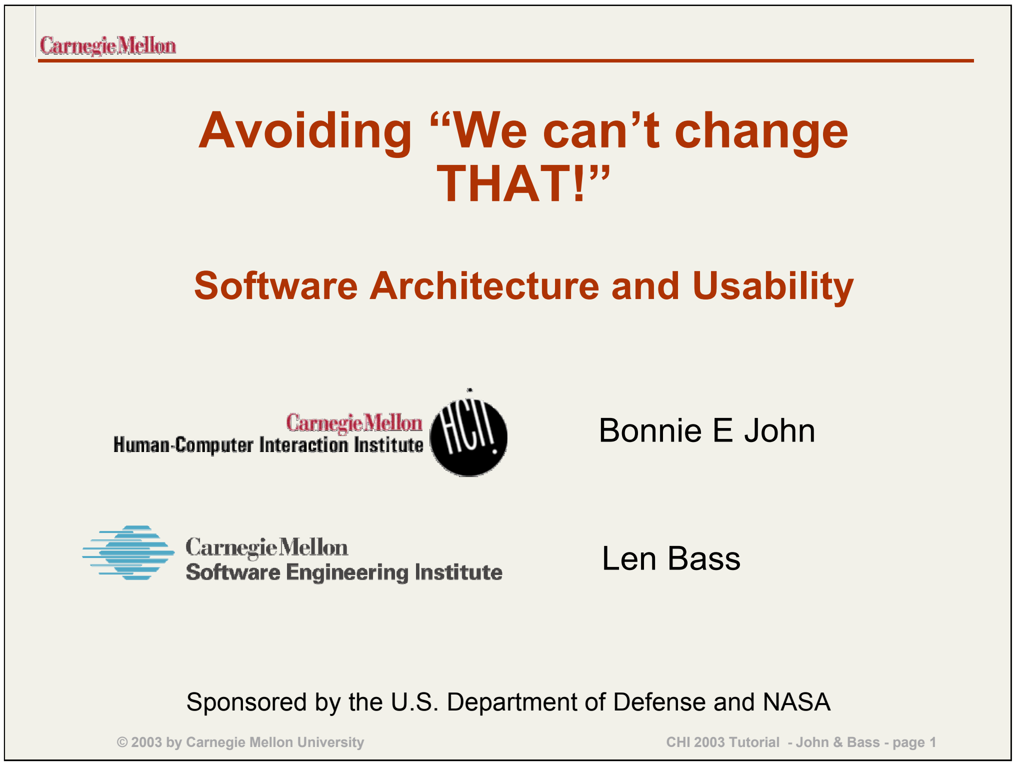
- Understand basic principles of software architecture for interactive systems and its relationship to the usability of that system
- Be able to evaluate whether common usability scenarios will arise in the systems they are developing and what implications these usability scenarios have for software architecture design
- Understand patterns of software architecture that facilitate usability, and recognize architectural decisions that preclude usability of the end-product, so that they can effectively bring usability considerations into early architectural design.

Abstract

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

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

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



Carnegie Mellon

Avoiding "We can't change THAT!"

Software Architecture and Usability

Carnegie Mellon Human-Computer Interaction Institute — Bonnie E John

Carnegie Mellon Software Engineering Institute — Len Bass

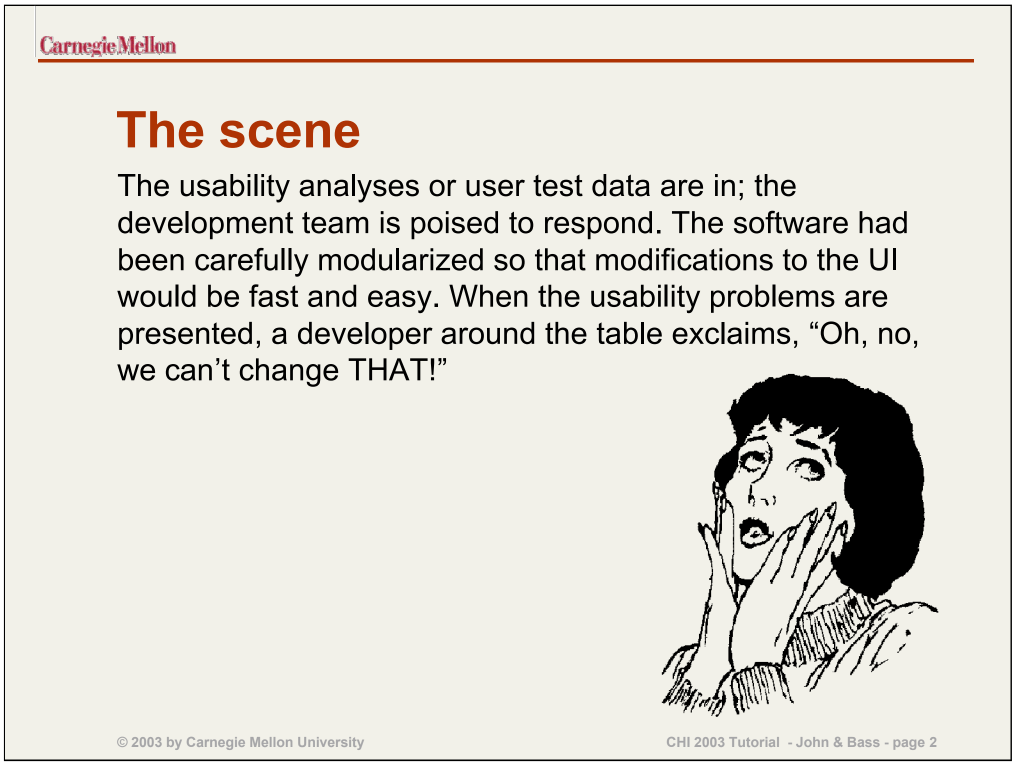
Sponsored by the U.S. Department of Defense and NASA

CHI 2003 Tutorial - John & Bass - page 1

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

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

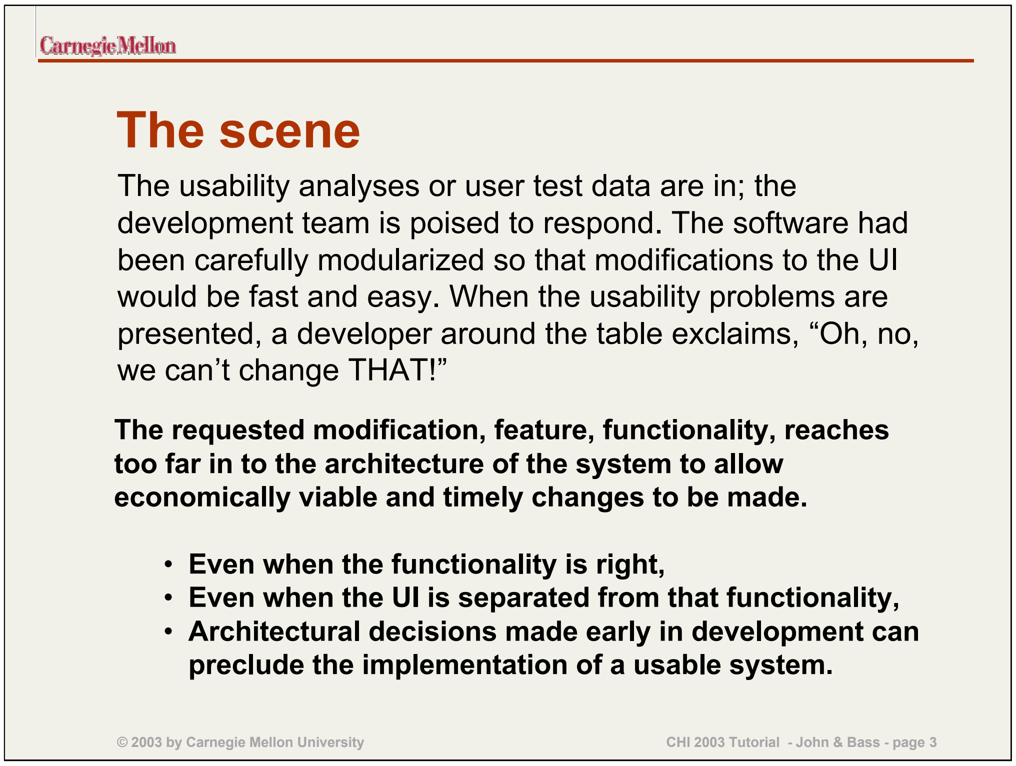
We appreciate the assistance from
Rob J. Adams
Human-Computer Interaction Institute
Carnegie Mellon University
Pittsburgh PA 15213
1-412-268-1612
rjadams@cs.cmu.edu



Carnegie Mellon

The scene

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

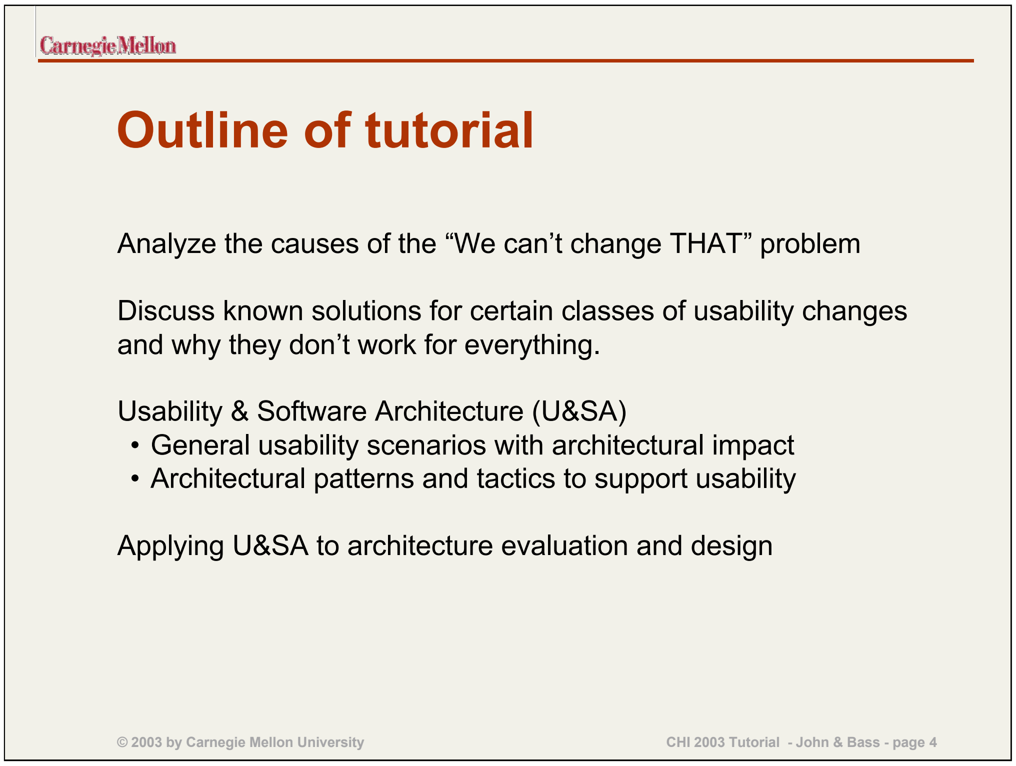


The scene

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

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

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


Carnegie Mellon

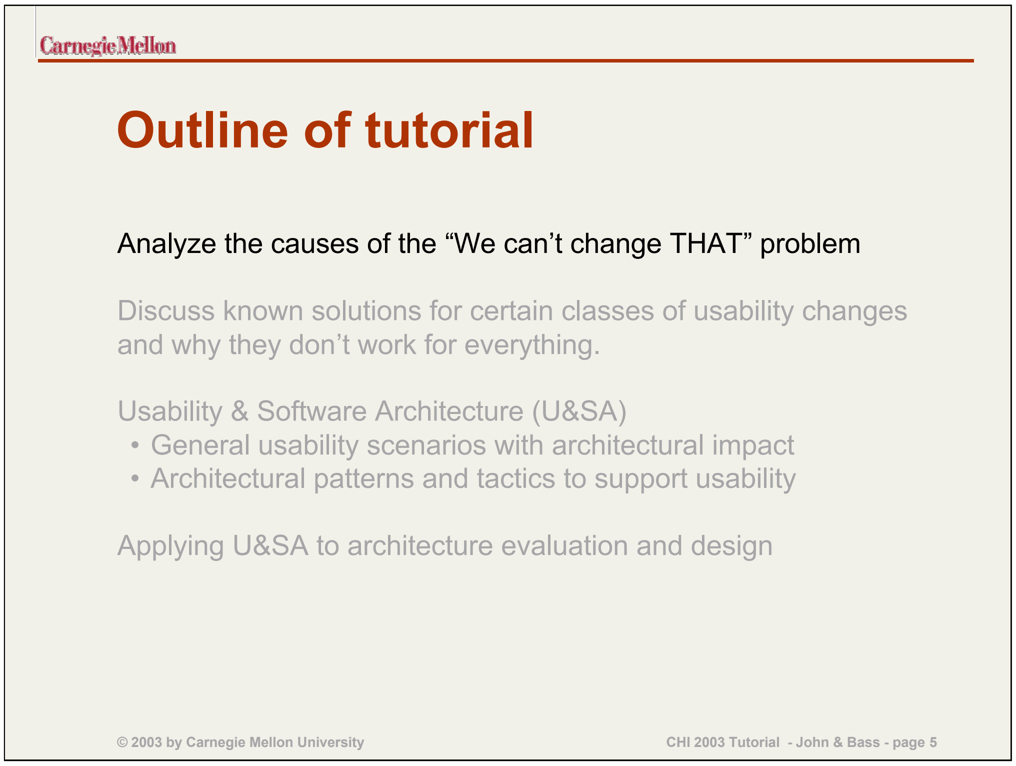
Outline of tutorial

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

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

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

Applying U&SA to architecture evaluation and design



Carnegie Mellon

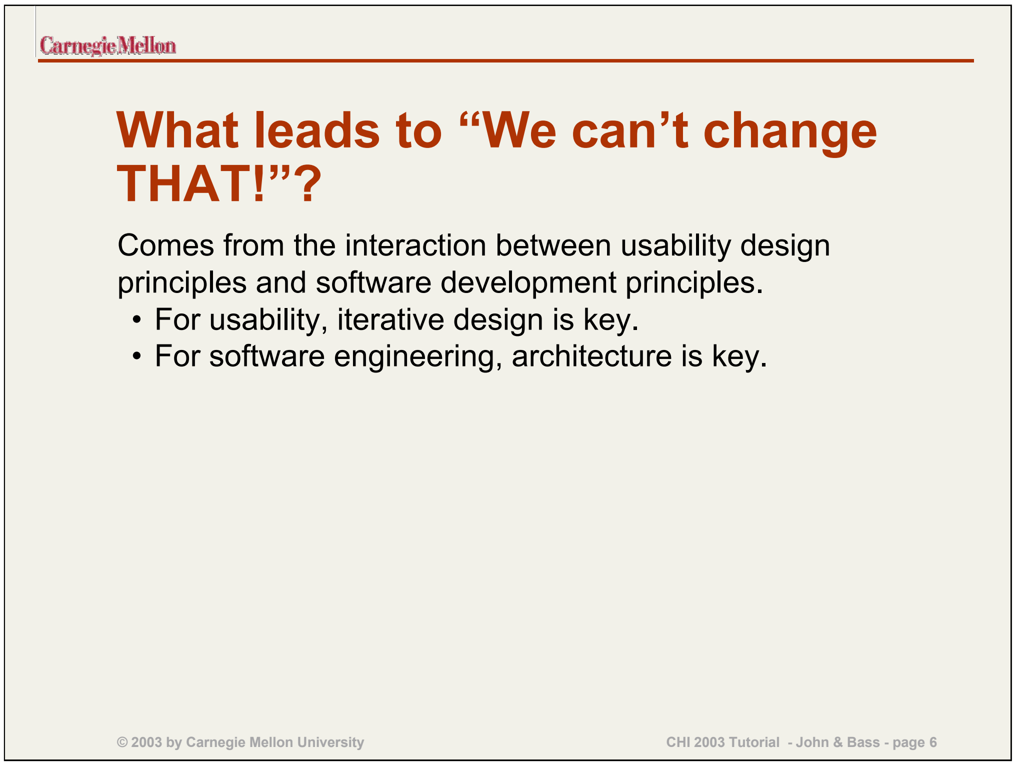
Outline of tutorial

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

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

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

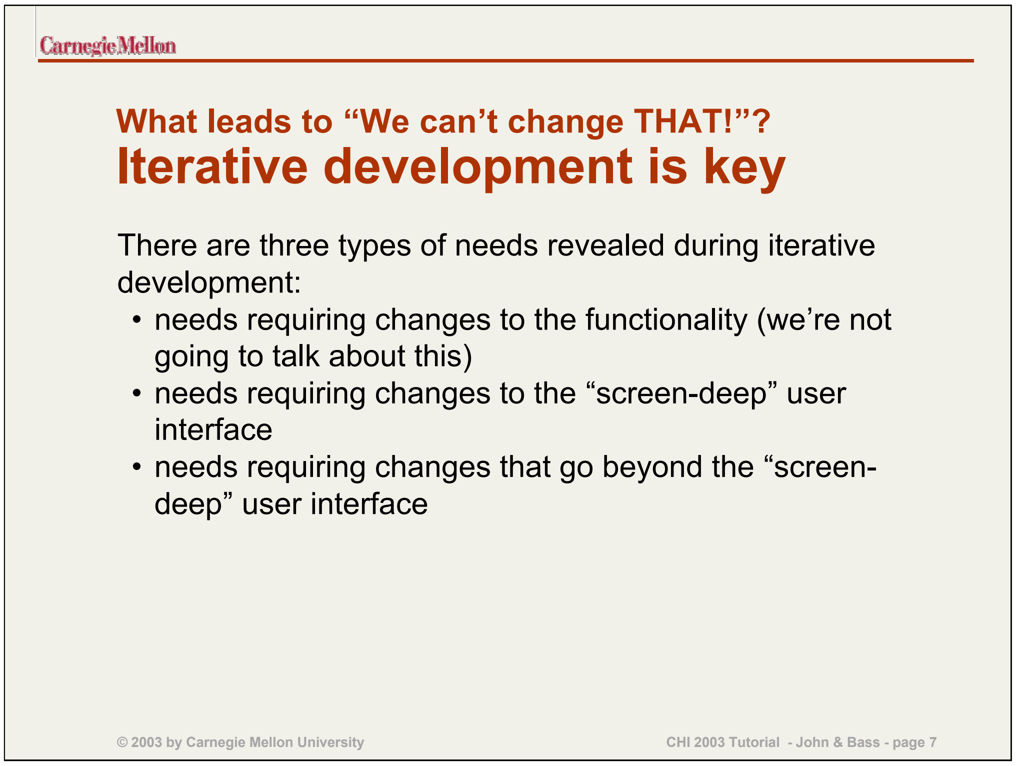
Applying U&SA to architecture evaluation and design


Carnegie Mellon

What leads to "We can't change THAT!"?

Comes from the interaction between usability design principles and software development principles.

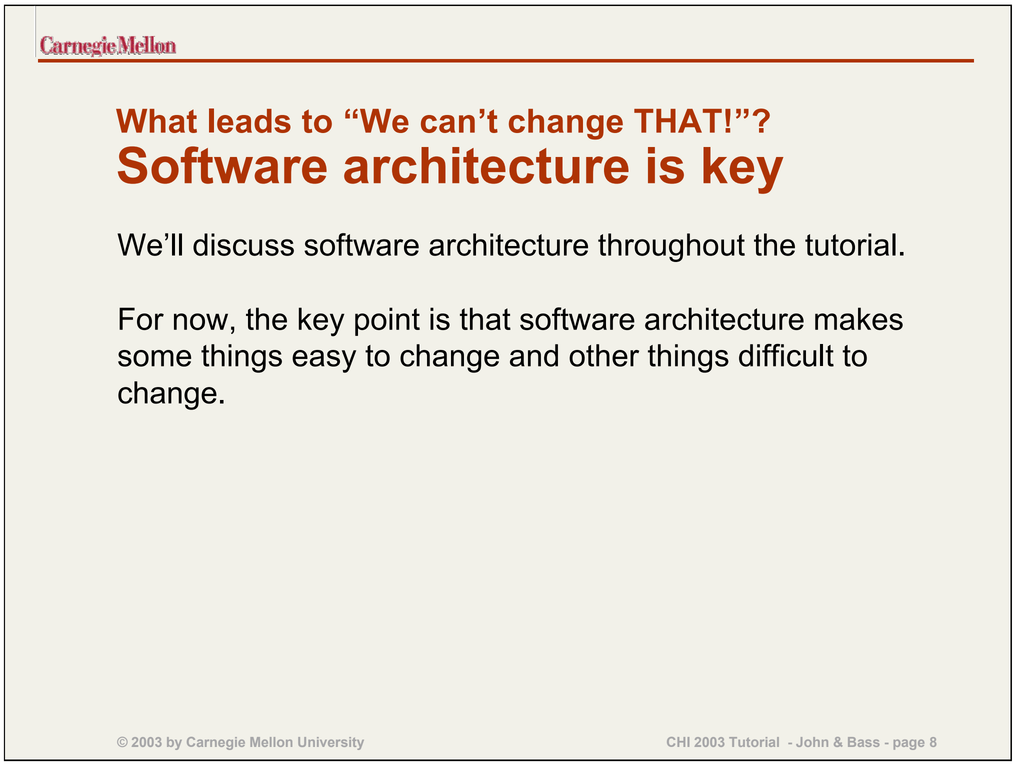
- For usability, iterative design is key.
- For software engineering, architecture is key.



What leads to “We can’t change THAT!”? Iterative development is key

There are three types of needs revealed during iterative development:

- needs requiring changes to the functionality (we’re not going to talk about this)
- needs requiring changes to the “screen-deep” user interface
- needs requiring changes that go beyond the “screen-deep” user interface

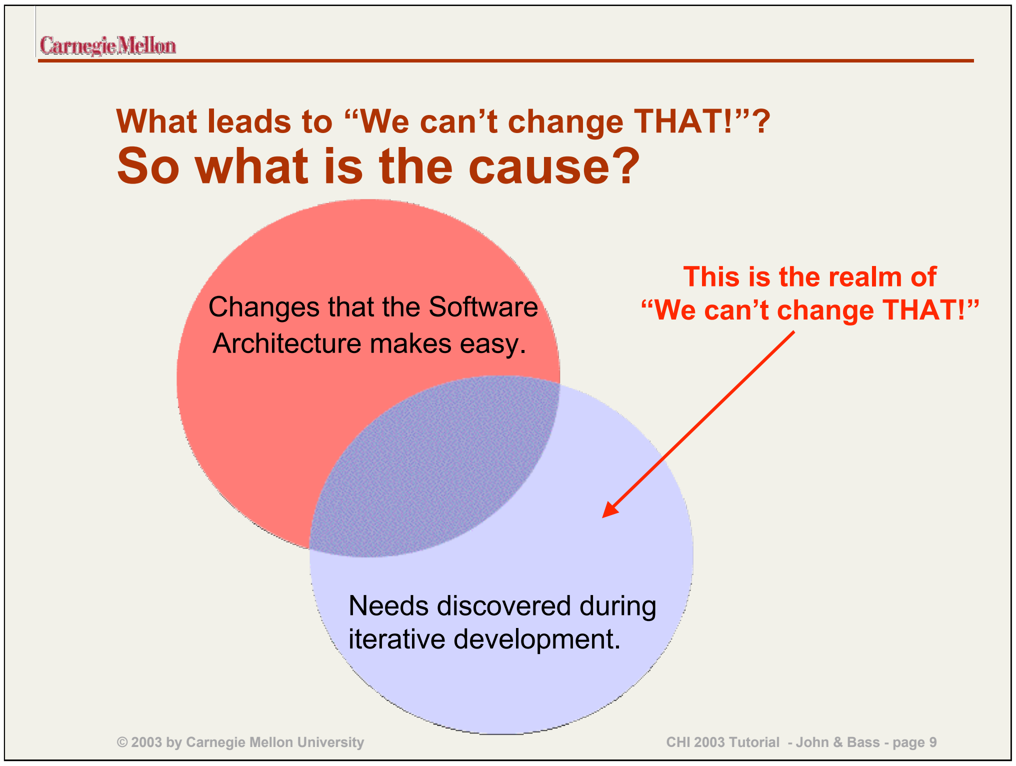


Carnegie Mellon

What leads to “We can’t change THAT!”?
Software architecture is key

We’ll discuss software architecture throughout the tutorial.

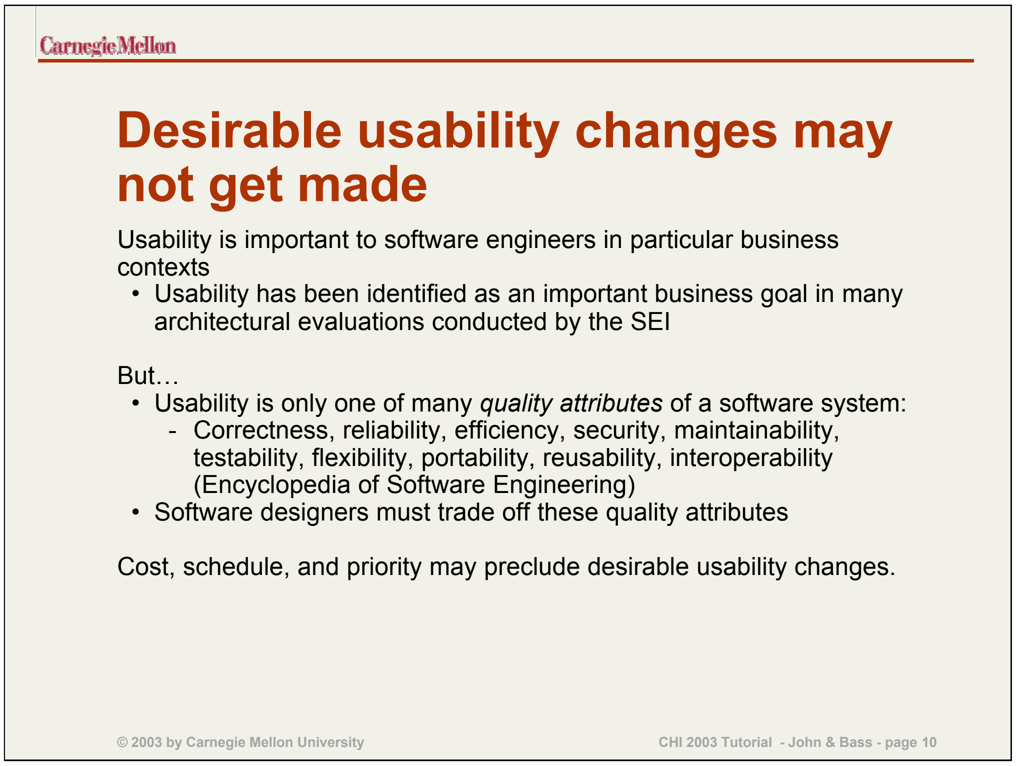
For now, the key point is that software architecture makes some things easy to change and other things difficult to change.



Carnegie Mellon

What leads to "We can't change THAT!"? So what is the cause?

Changes that the Software Architecture makes easy.
Needs discovered during iterative development.
This is the realm of "We can't change THAT!"


Carnegie Mellon

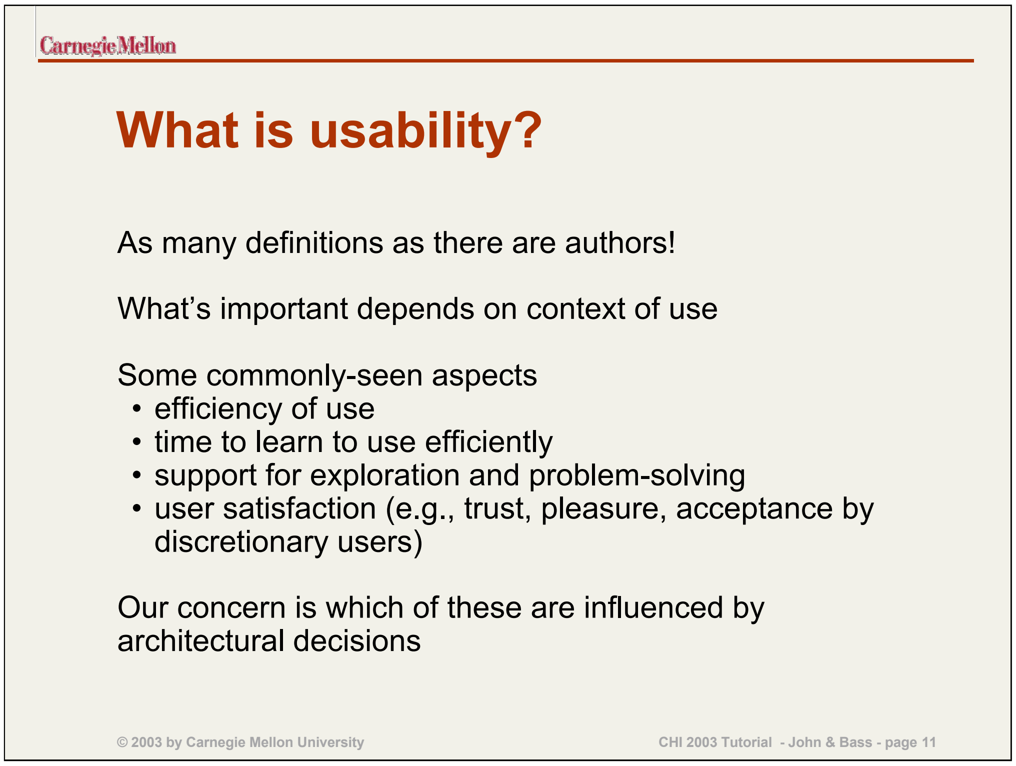
Desirable usability changes may not get made

Usability is important to software engineers in particular business contexts
- Usability has been identified as an important business goal in many architectural evaluations conducted by the SEI

But…
- Usability is only one of many *quality attributes* of a software system:
    - Correctness, reliability, efficiency, security, maintainability, testability, flexibility, portability, reusability, interoperability (Encyclopedia of Software Engineering)
- Software designers must trade off these quality attributes

Cost, schedule, and priority may preclude desirable usability changes.

McCall, J. (2001) Quality Factors. In *Encyclopedia of Software Engineering* (2nd edition) John Marciniak, ed., John Wiley, New York, pp 1083-1093



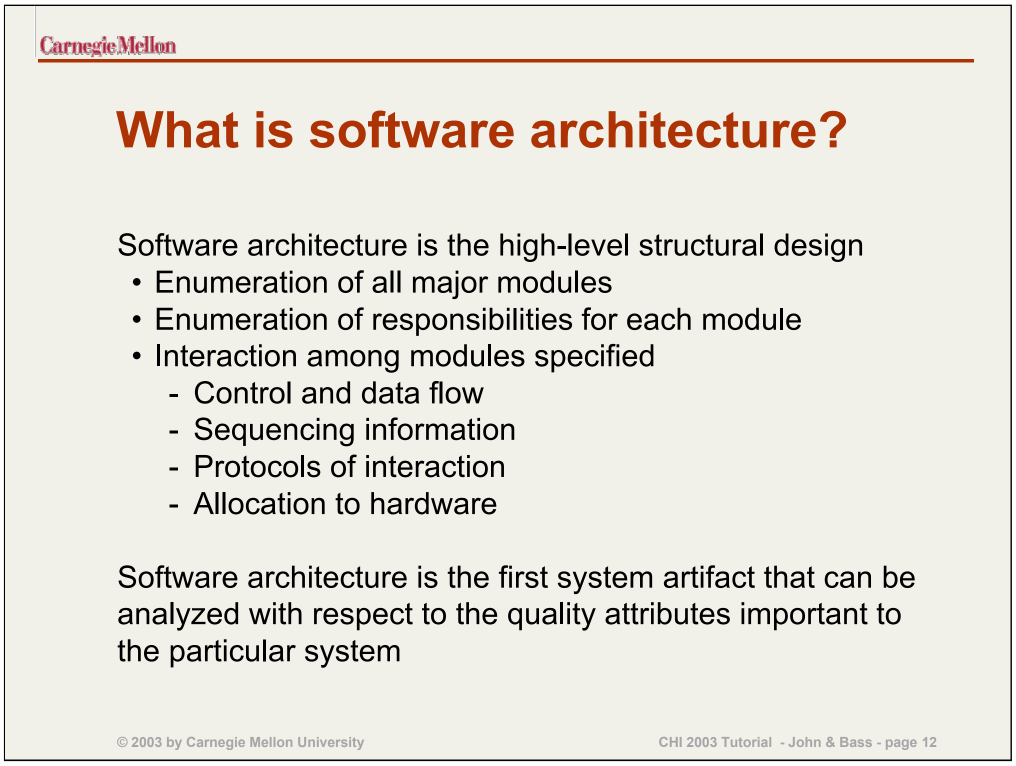
What is usability?

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)

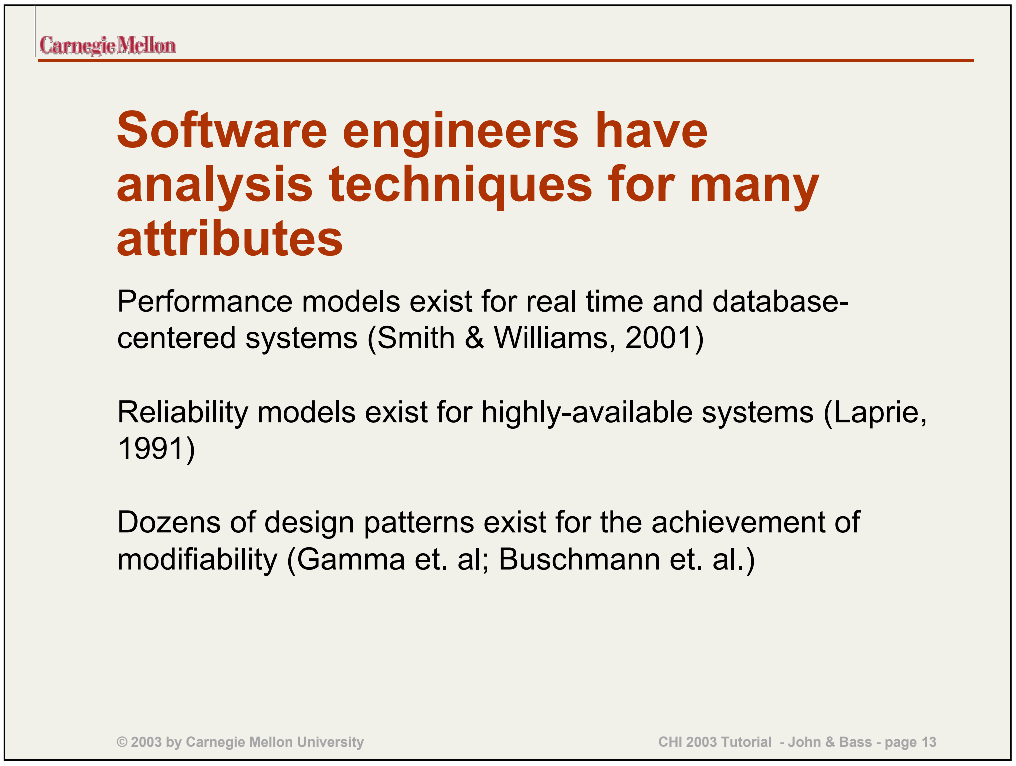
Our concern is which of these are influenced by architectural decisions


Carnegie Mellon

What is software architecture?

Software architecture is the high-level structural design
- Enumeration of all major modules
- Enumeration of responsibilities for each module
- Interaction among modules specified
  - Control and data flow
  - Sequencing information
  - Protocols of interaction
  - Allocation to hardware

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


Carnegie Mellon

Software engineers have analysis techniques for many attributes

Performance models exist for real time and database-centered systems (Smith & Williams, 2001)

Reliability models exist for highly-available systems (Laprie, 1991)

Dozens of design patterns exist for the achievement of modifiability (Gamma et. al; Buschmann et. al.)

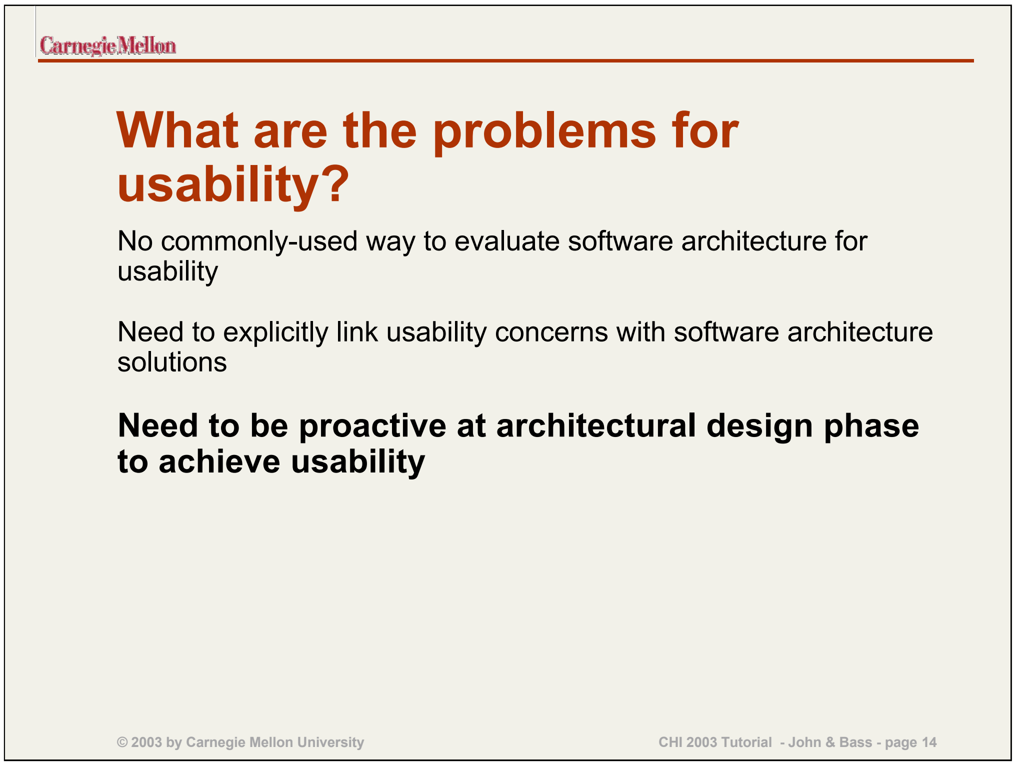
© 2003 by Carnegie Mellon University CHI 2003 Tutorial - John & Bass - page 13

Smith, C. & Williams, L., (2001) *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Reading, Ma.:Addison Wesley Longman.

Laprie, J.-C. (1992) *Dependability: Basic Concepts and Terminology*. Springer-Verlag: Vienna.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., (1995) *Design Patterns, Elements of Reusable Object-Oriented Software*, 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.

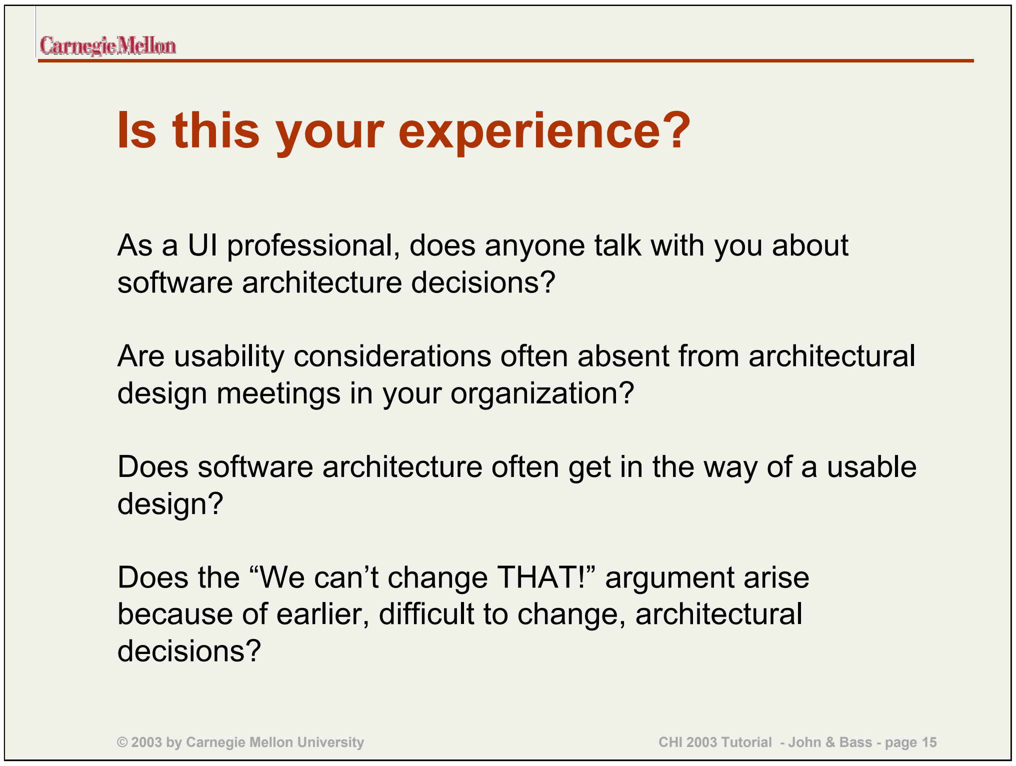

Carnegie Mellon

What are the problems for usability?

No commonly-used way to evaluate software architecture for usability

Need to explicitly link usability concerns with software architecture solutions

Need to be proactive at architectural design phase to achieve usability


Carnegie Mellon

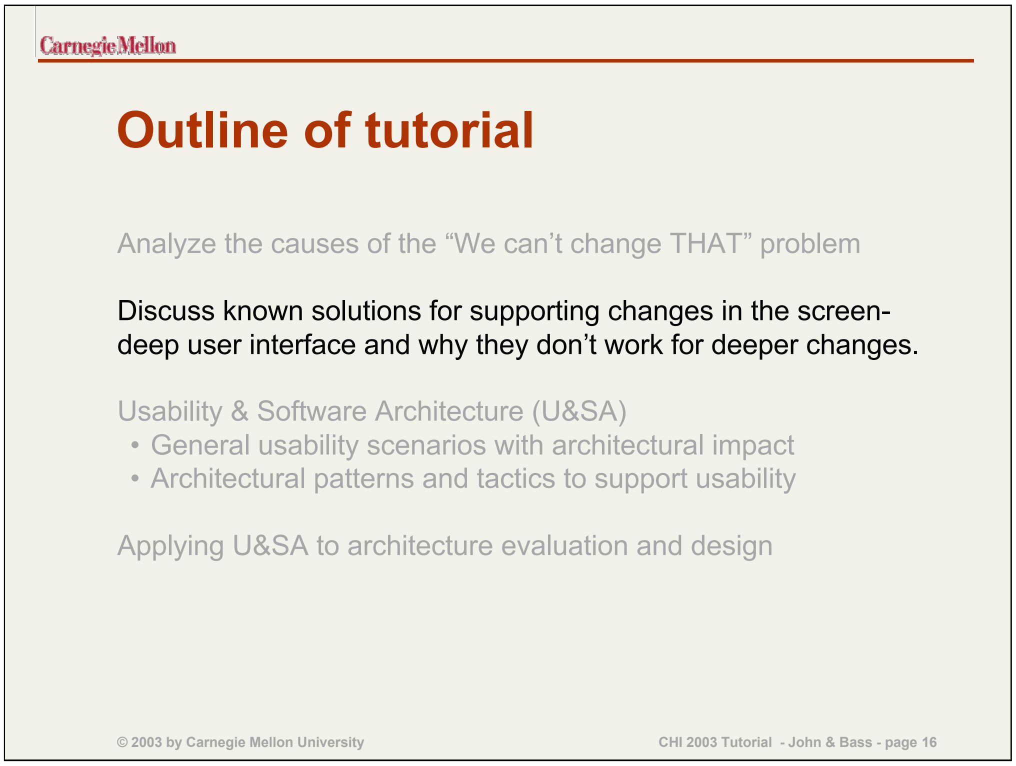
Is this your experience?

As a UI professional, does anyone talk with you about software architecture decisions?

Are usability considerations often absent from architectural design meetings in your organization?

Does software architecture often get in the way of a usable design?

Does the “We can’t change THAT!” argument arise because of earlier, difficult to change, architectural decisions?



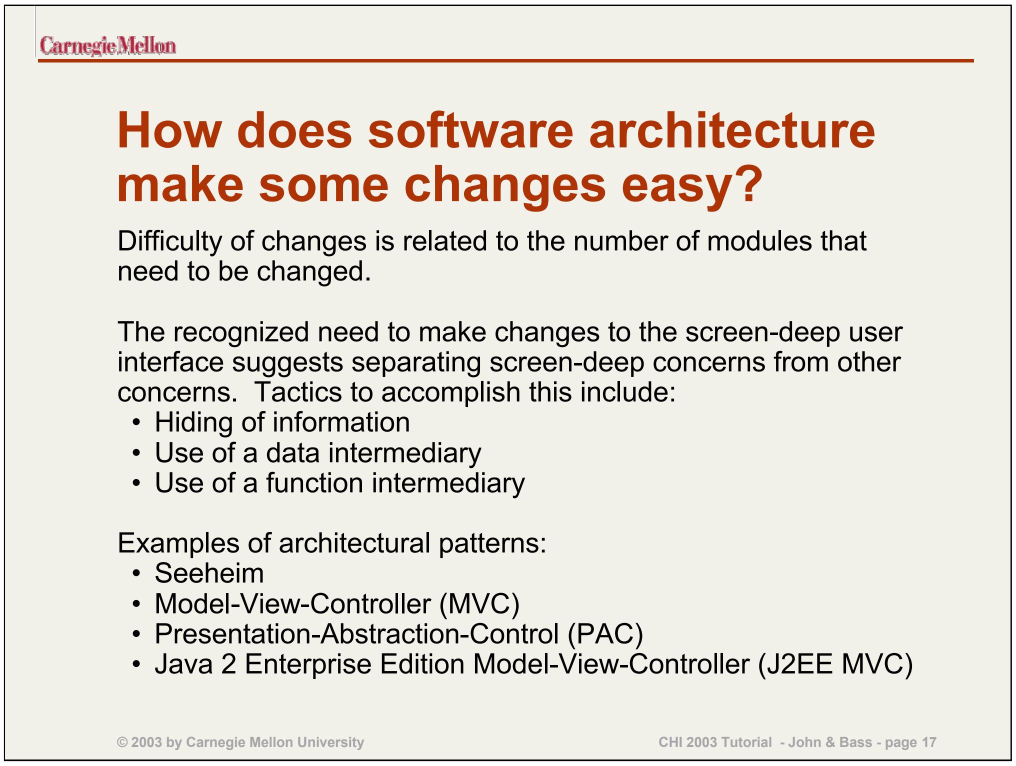
Outline of tutorial

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

Discuss known solutions for supporting changes in the screen-deep user interface and why they don’t work for deeper changes.

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

Applying U&SA to architecture evaluation and design



Carnegie Mellon

How does software architecture make some changes easy?

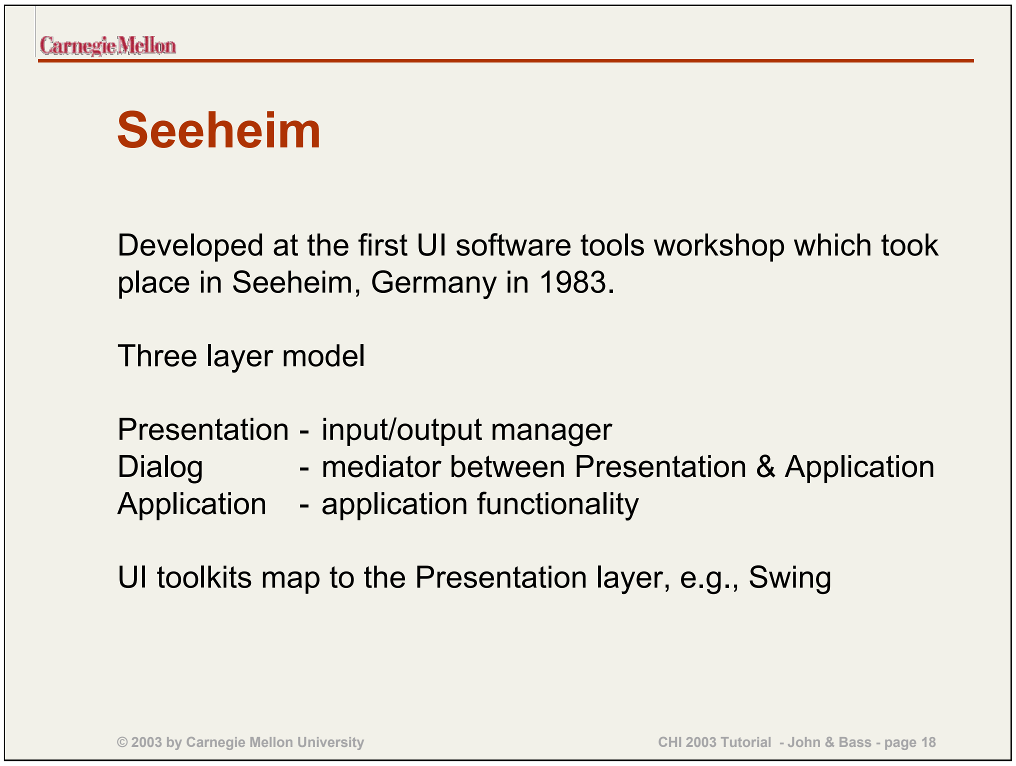
Difficulty of changes is related to the number of modules that need to be changed.

The recognized need to make changes to the screen-deep user interface suggests separating screen-deep concerns from other concerns. Tactics to accomplish this include:

- Hiding of information
- Use of a data intermediary
- Use of a function intermediary

Examples of architectural patterns:

- Seeheim
- Model-View-Controller (MVC)
- Presentation-Abstraction-Control (PAC)
- Java 2 Enterprise Edition Model-View-Controller (J2EE MVC)



Carnegie Mellon

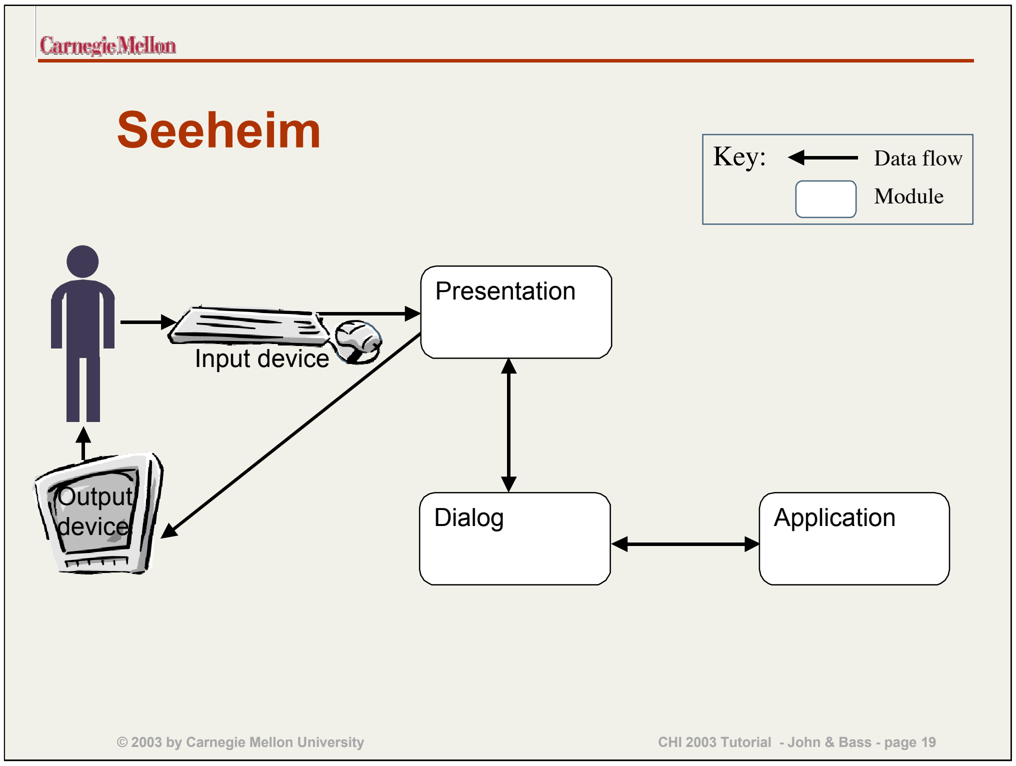
Seeheim

Developed at the first UI software tools workshop which took place in Seeheim, Germany in 1983.

Three layer model

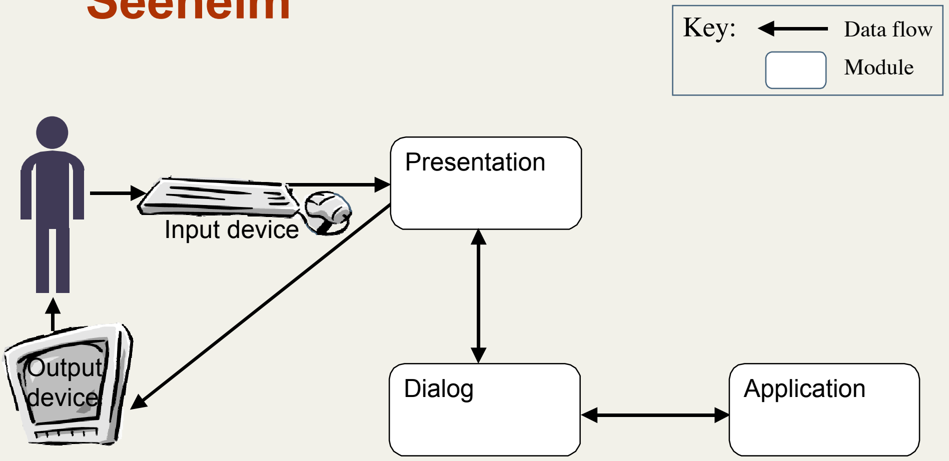
Presentation	- input/output manager
Dialog	- mediator between Presentation & Application
Application	- application functionality

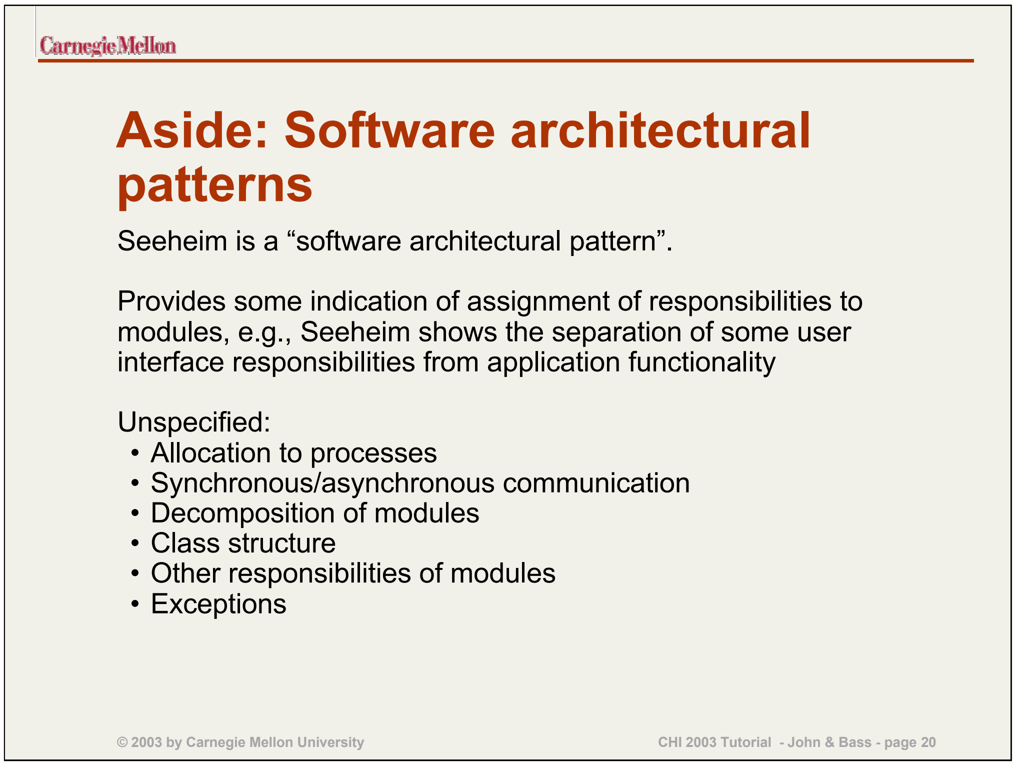
UI toolkits map to the Presentation layer, e.g., Swing



Carnegie Mellon

Seeheim




Carnegie Mellon

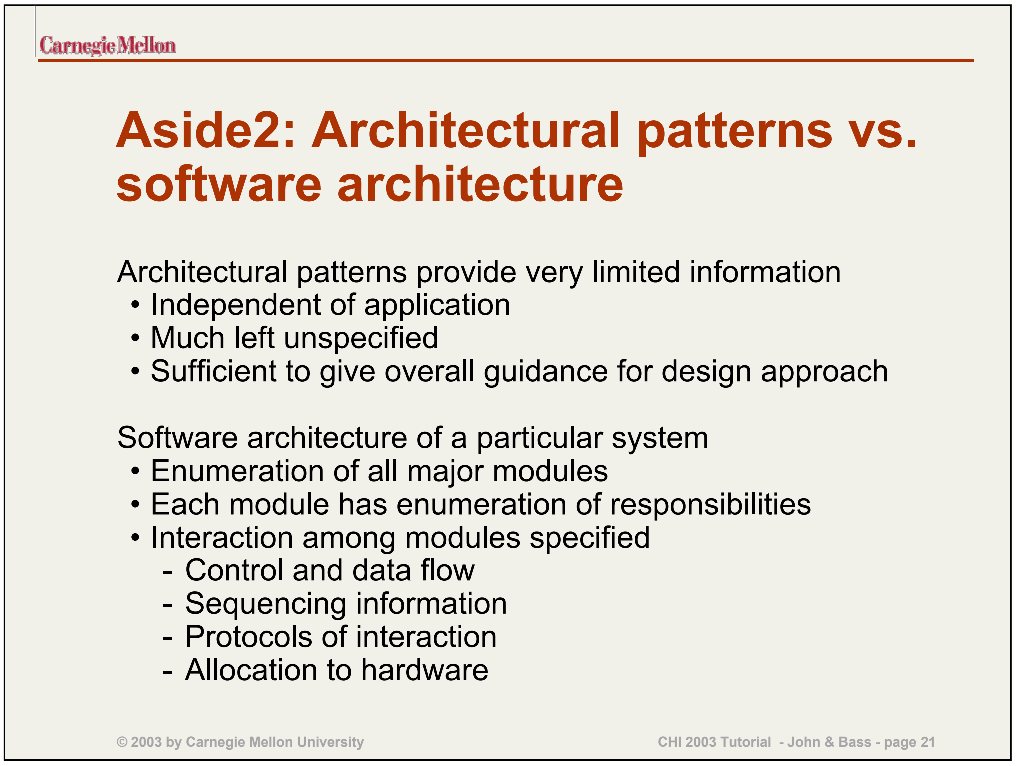
Aside: Software architectural patterns

Seeheim is a “software architectural pattern”.

Provides some indication of assignment of responsibilities to modules, e.g., Seeheim shows the separation of some user interface responsibilities from application functionality

Unspecified:

- Allocation to processes
- Synchronous/asynchronous communication
- Decomposition of modules
- Class structure
- Other responsibilities of modules
- Exceptions



Carnegie Mellon

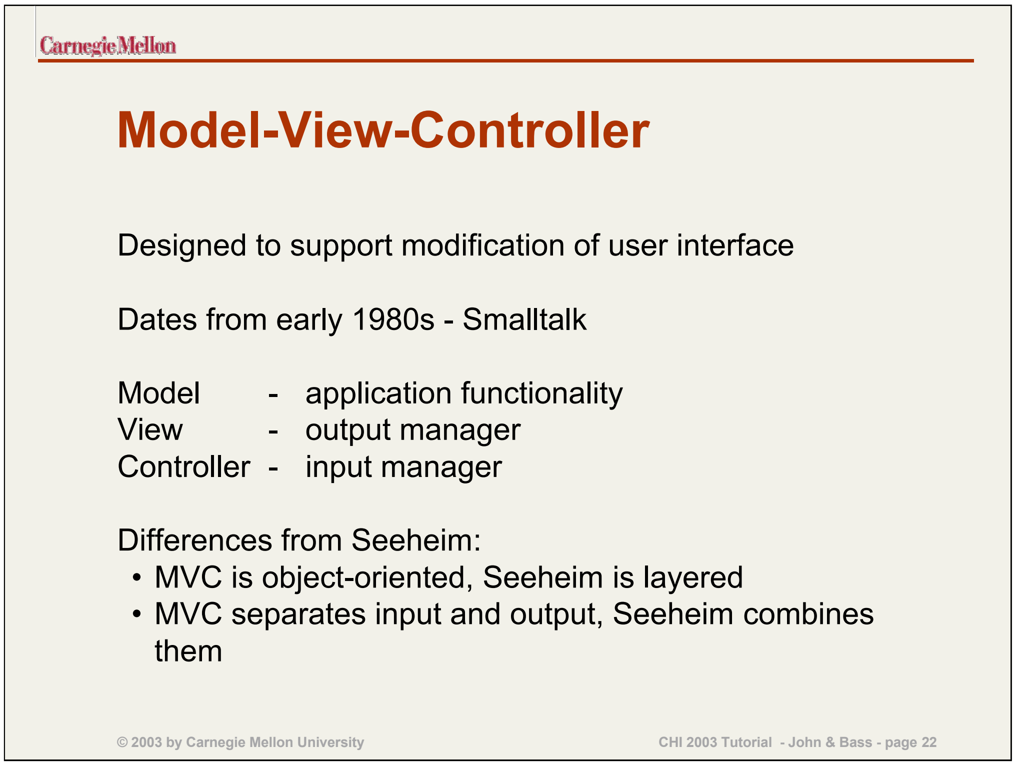
Aside2: Architectural patterns vs. software architecture

Architectural patterns provide very limited information

- Independent of application
- Much left unspecified
- Sufficient to give overall guidance for design approach

Software architecture of a particular system

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



Carnegie Mellon

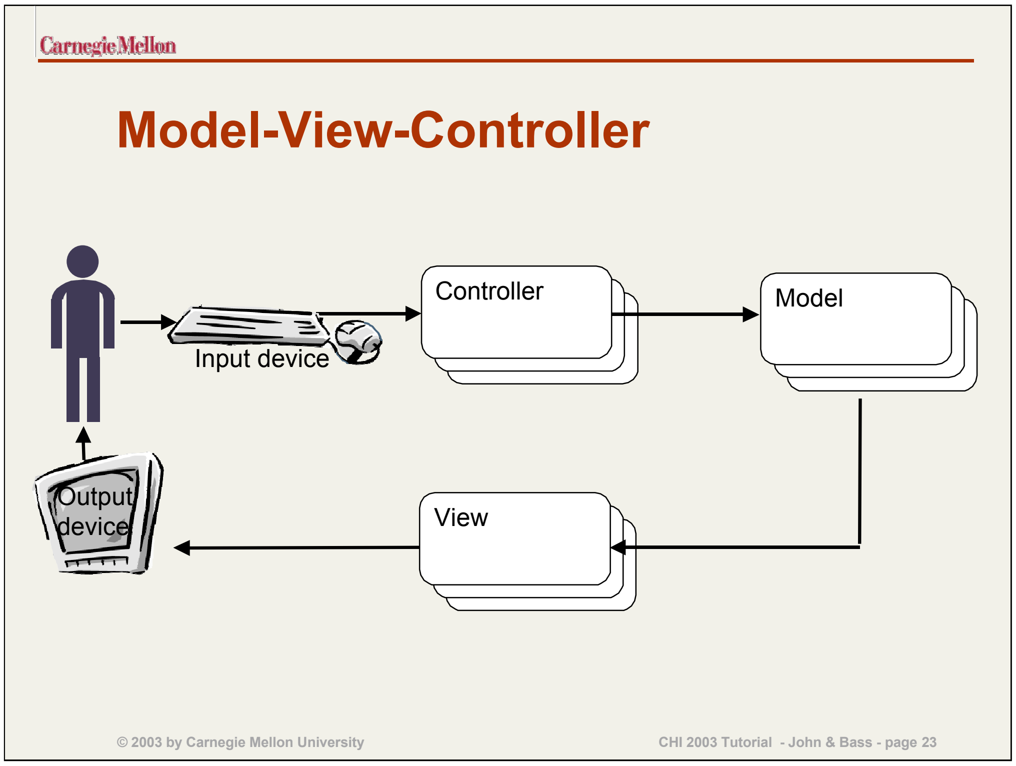
Model-View-Controller

Designed to support modification of user interface

Dates from early 1980s - Smalltalk

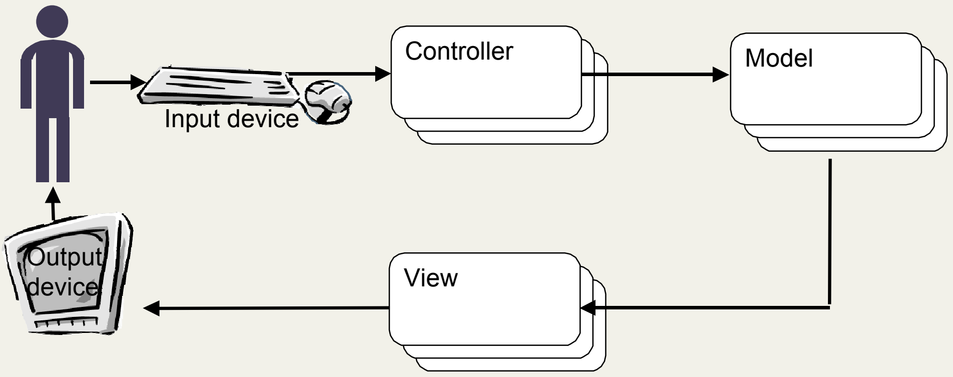
Model	-	application functionality
View	-	output manager
Controller	-	input manager

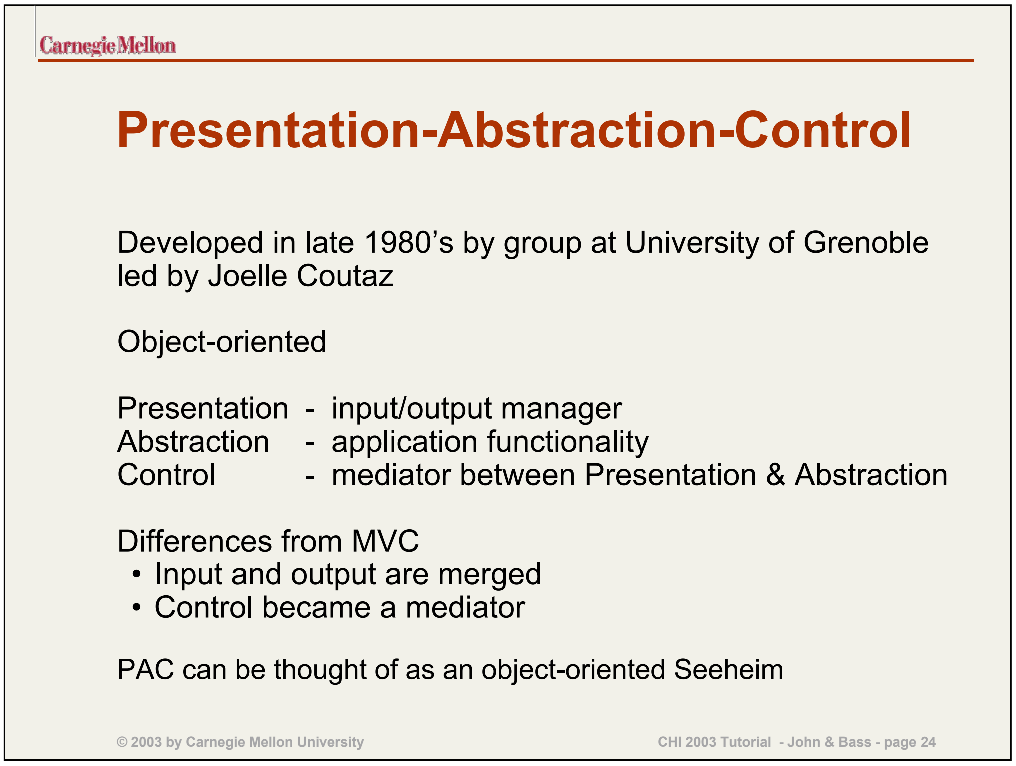
Differences from Seeheim:
- MVC is object-oriented, Seeheim is layered
- MVC separates input and output, Seeheim combines them



Carnegie Mellon

Model-View-Controller





Carnegie Mellon

Presentation-Abstraction-Control

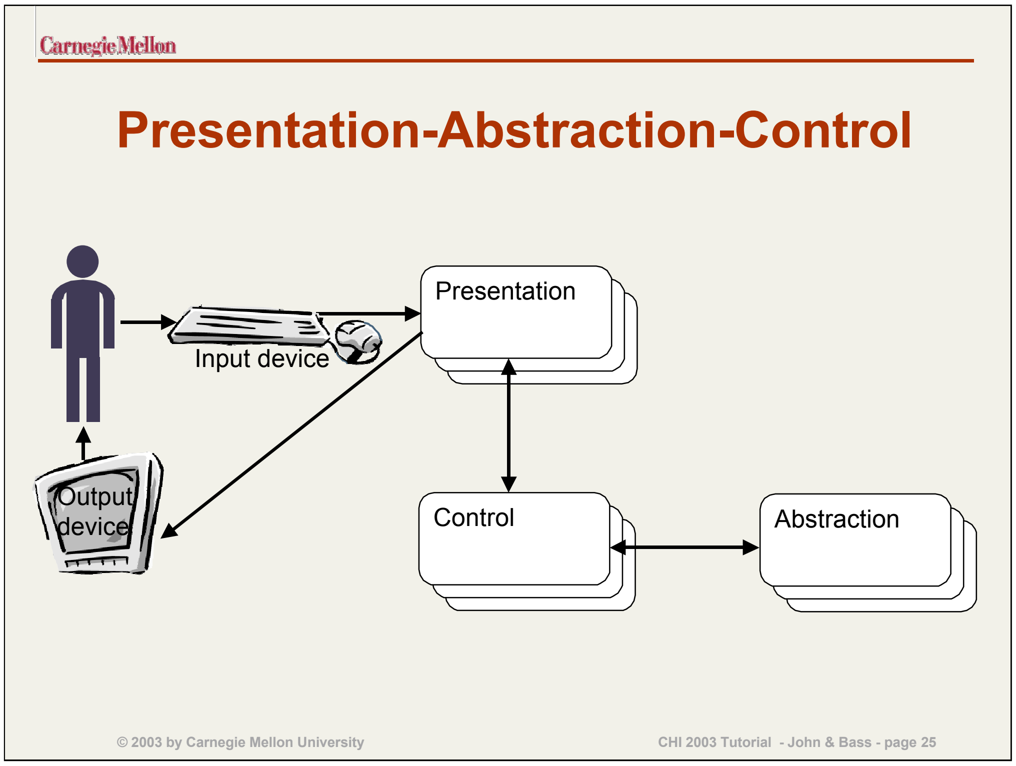
Developed in late 1980’s by group at University of Grenoble led by Joelle Coutaz

Object-oriented

Presentation	-	input/output manager
Abstraction	-	application functionality
Control	-	mediator between Presentation & Abstraction

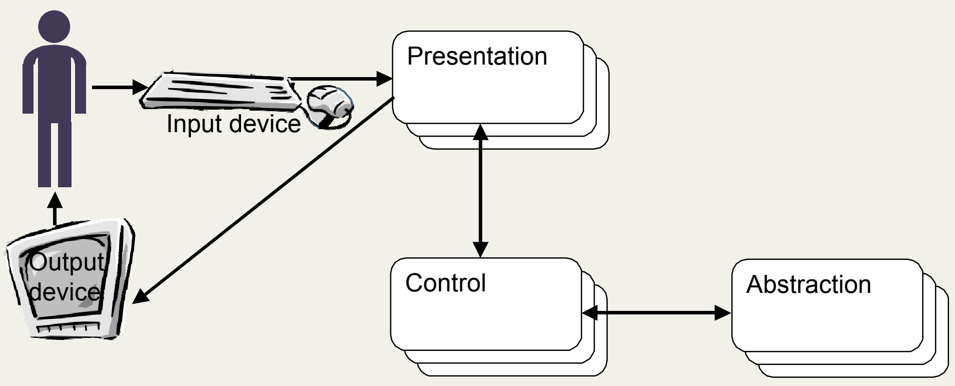
Differences from MVC
- Input and output are merged
- Control became a mediator

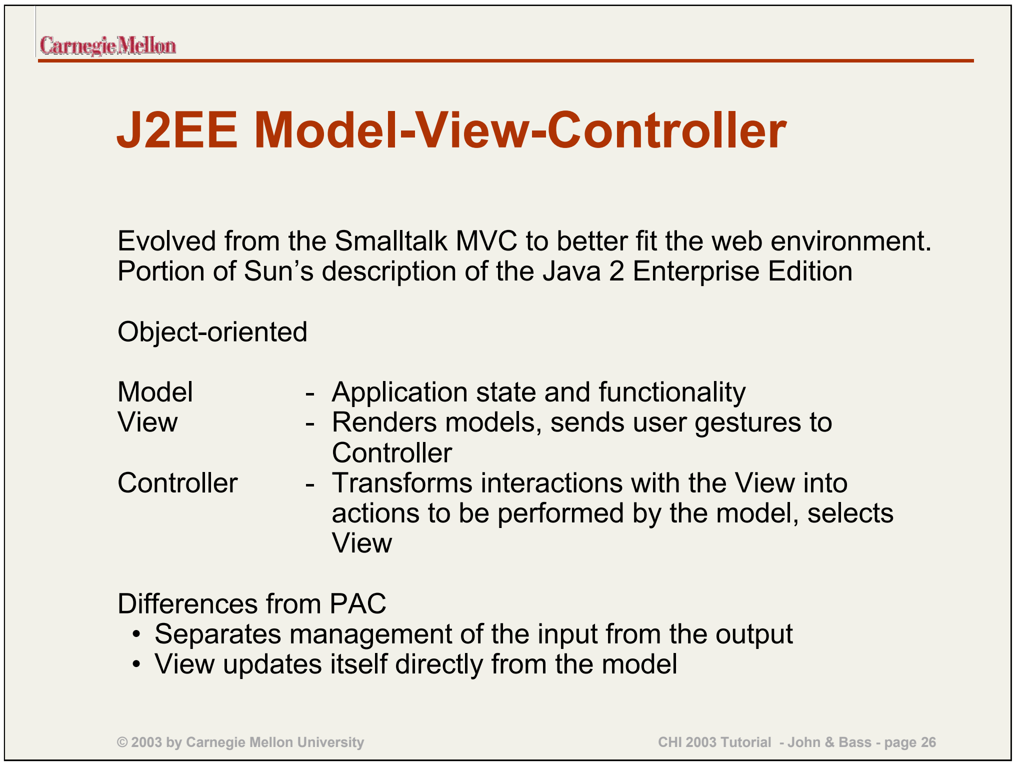
PAC can be thought of as an object-oriented Seeheim



Carnegie Mellon

Presentation-Abstraction-Control





Carnegie Mellon

J2EE Model-View-Controller

Evolved from the Smalltalk MVC to better fit the web environment.
Portion of Sun's description of the Java 2 Enterprise Edition

Object-oriented

Model	- Application state and functionality
View	- Renders models, sends user gestures to Controller
Controller	- Transforms interactions with the View into actions to be performed by the model, selects View

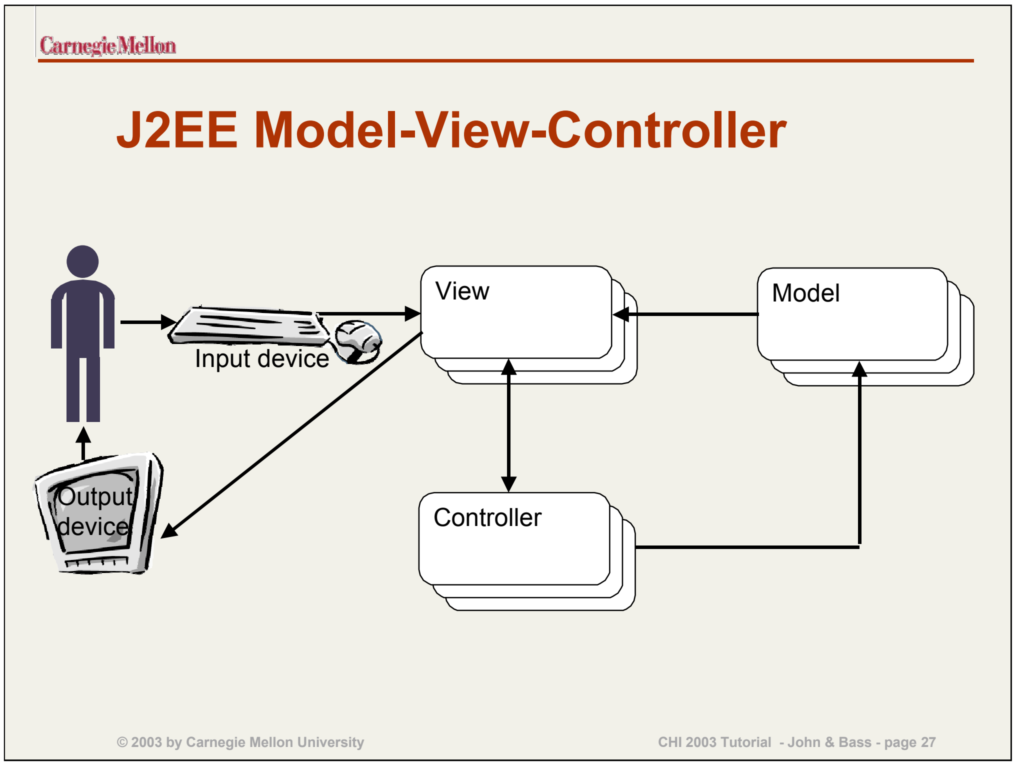
Differences from PAC
- Separates management of the input from the output
- View updates itself directly from the model

© 2003 by Carnegie Mellon University CHI 2003 Tutorial - John & Bass - page 26

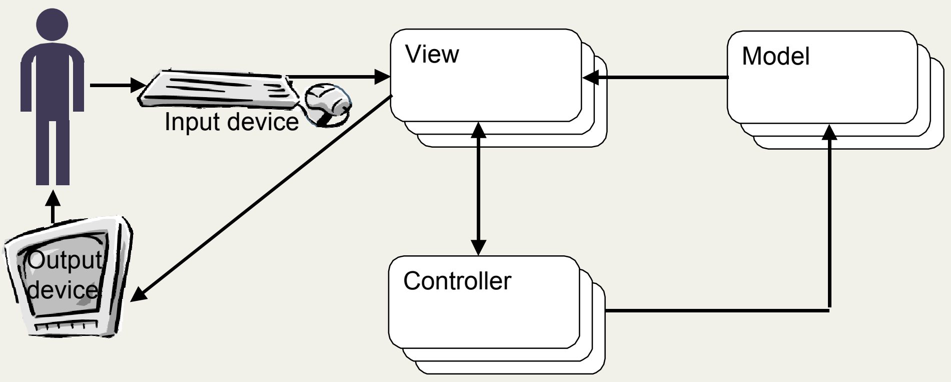
The Model is independent of both the View and the Controller. In the J2EE framework, the Model is usually implemented using EJBs and a database (RDBMS).

In the J2EE framework, the Controller is typically implemented with Java Servlets.

The View is implemented using JSPs and HTML plus the client (e.g., the user's web browser).


Carnegie Mellon

J2EE Model-View-Controller

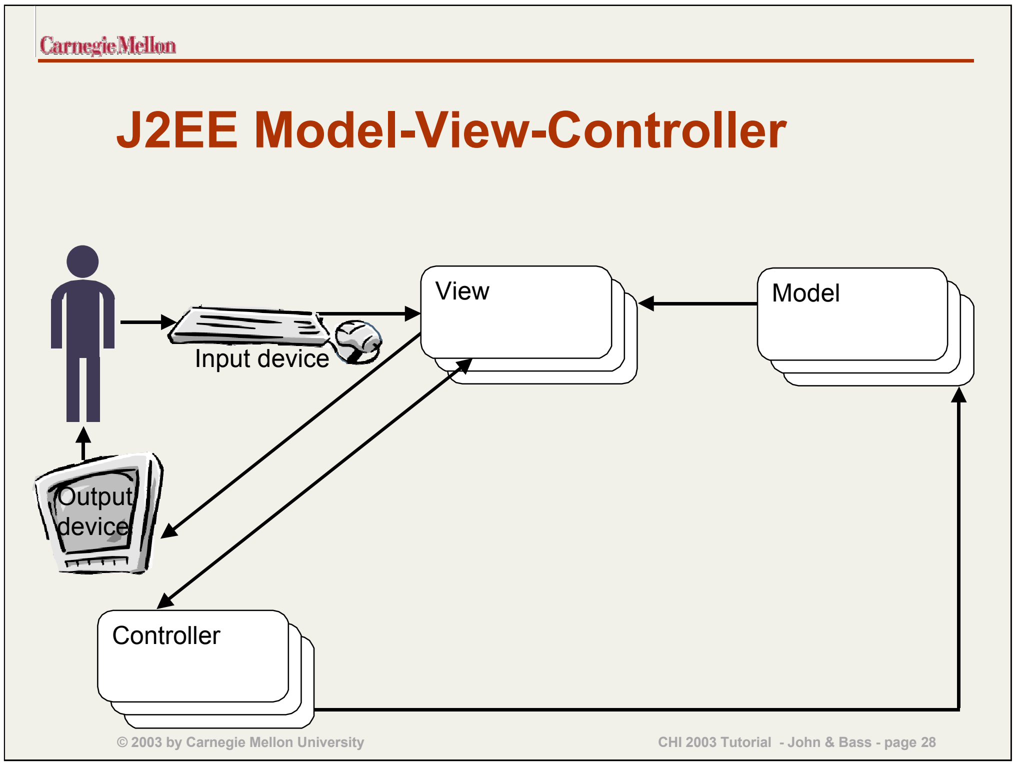


© 2003 by Carnegie Mellon University

CHI 2003 Tutorial - John & Bass - page 27

The View is intended to run on the client computer, the Controller is intended to run on the server computer, consequentially input is passed through the View to the Controller.

The arrow from the Controller to the View represents the fact that the Controller determines the active View.



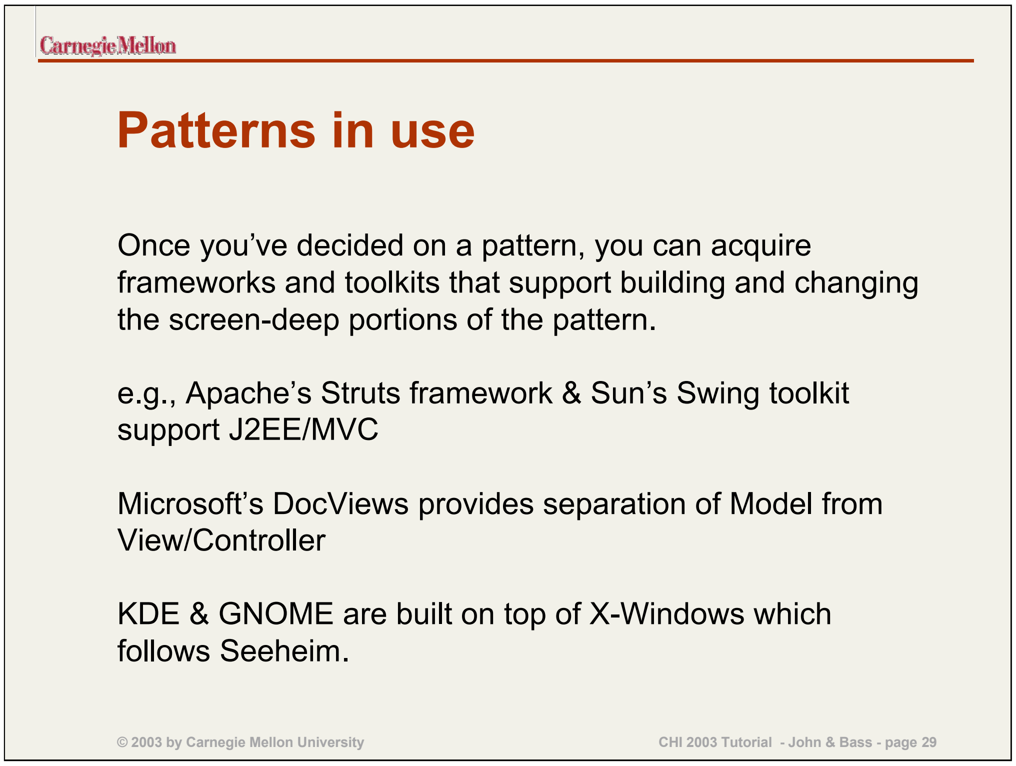
Carnegie Mellon

J2EE Model-View-Controller

© 2003 by Carnegie Mellon University

CHI 2003 Tutorial - John & Bass - page 28

All subsequent examples will be based on this architecture.



Carnegie Mellon

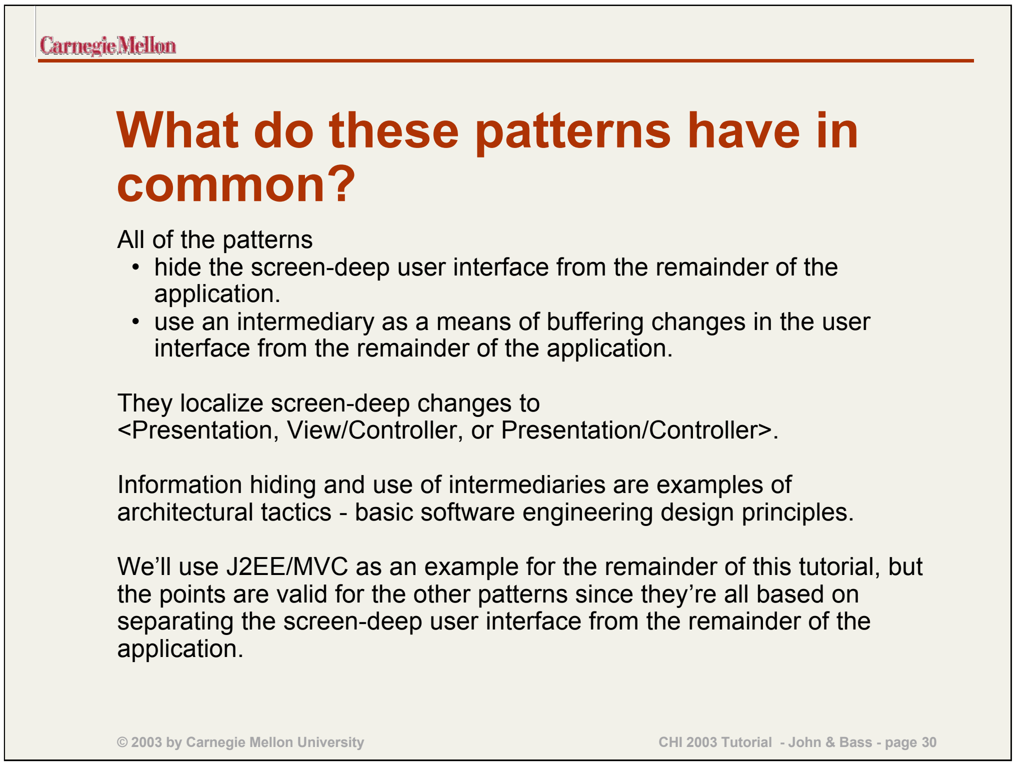
Patterns in use

Once you've decided on a pattern, you can acquire frameworks and toolkits that support building and changing the screen-deep portions of the pattern.

e.g., Apache's Struts framework & Sun's Swing toolkit support J2EE/MVC

Microsoft's DocViews provides separation of Model from View/Controller

KDE & GNOME are built on top of X-Windows which follows Seeheim.


Carnegie Mellon

What do these patterns have in common?

All of the patterns
- hide the screen-deep user interface from the remainder of the application.
- use an intermediary as a means of buffering changes in the user interface from the remainder of the application.

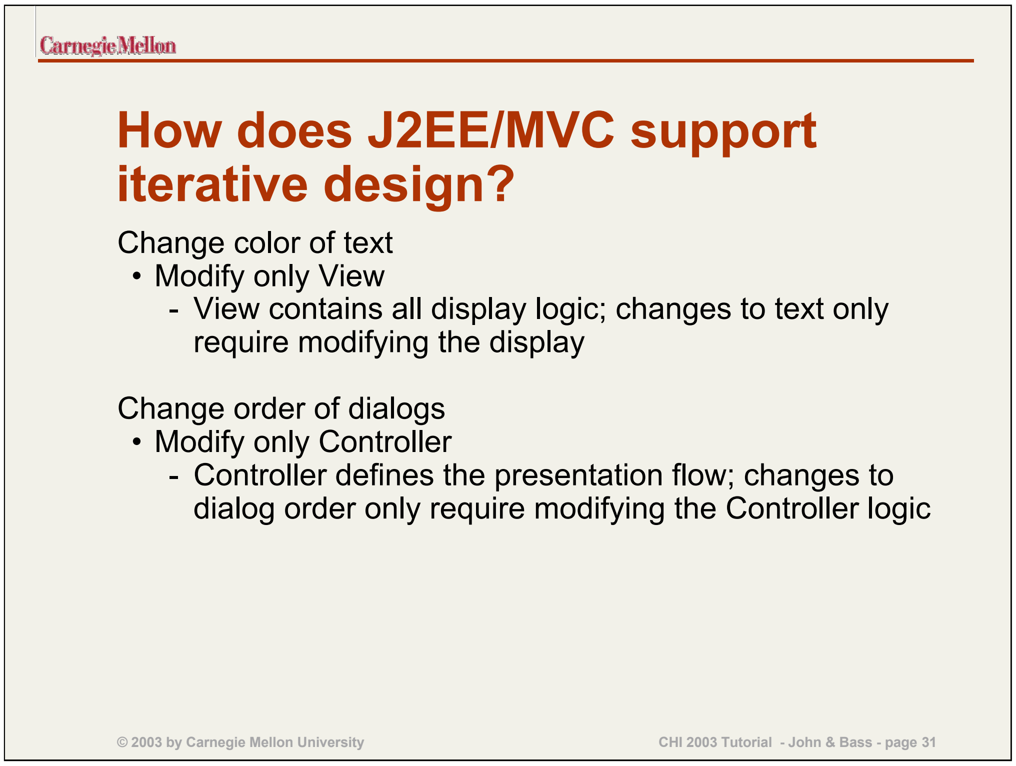
They localize screen-deep changes to
<Presentation, View/Controller, or Presentation/Controller>.

Information hiding and use of intermediaries are examples of architectural tactics - basic software engineering design principles.

We'll use J2EE/MVC as an example for the remainder of this tutorial, but the points are valid for the other patterns since they're all based on separating the screen-deep user interface from the remainder of the application.

© 2003 by Carnegie Mellon University CHI 2003 Tutorial - John & Bass - page 30

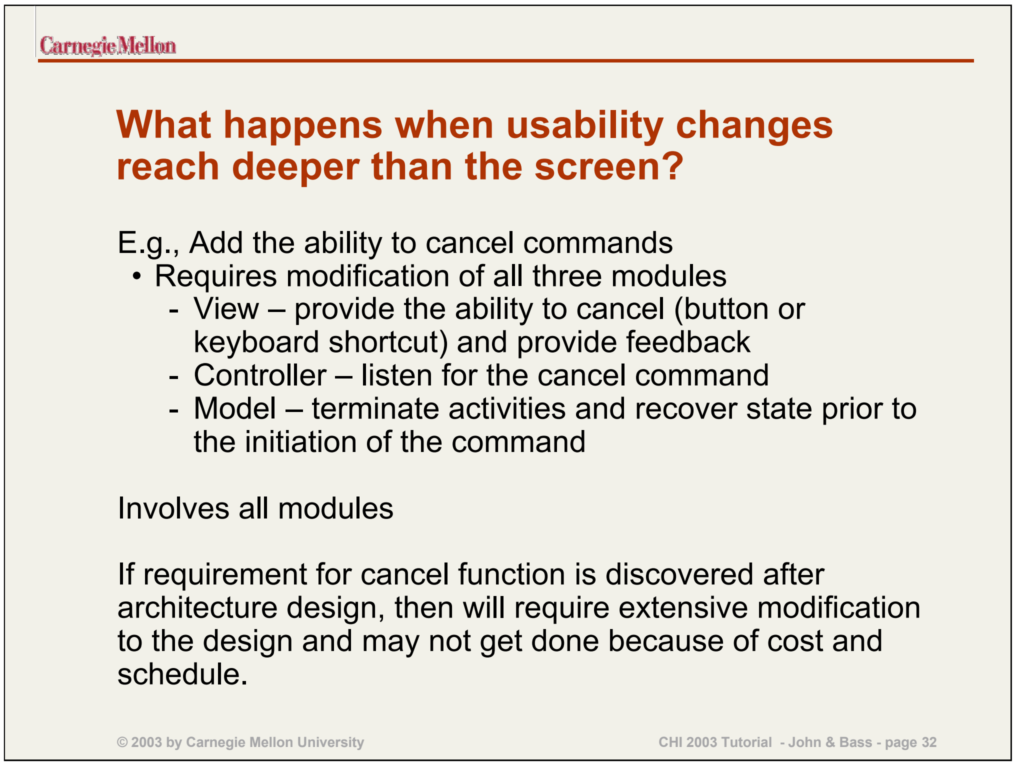
A full list of the tactics used in this tutorial is provided in Appendix III.


Carnegie Mellon

How does J2EE/MVC support iterative design?

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

Change order of dialogs
- Modify only Controller
  - Controller defines the presentation flow; changes to dialog order only require modifying the Controller logic

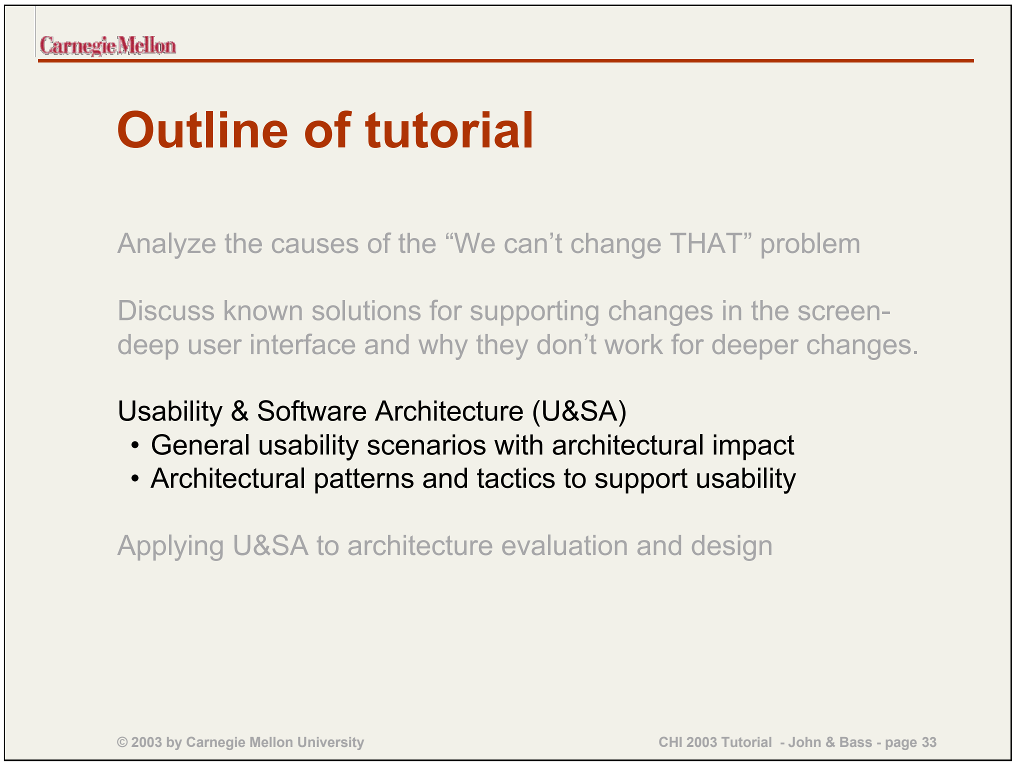

Carnegie Mellon

What happens when usability changes reach deeper than the screen?

E.g., Add the ability to cancel commands
- Requires modification of all three modules
  - View – provide the ability to cancel (button or keyboard shortcut) and provide feedback
  - Controller – listen for the cancel command
  - Model – terminate activities and recover state prior to the initiation of the command

Involves all modules

If requirement for cancel function is discovered after architecture design, then will require extensive modification to the design and may not get done because of cost and schedule.



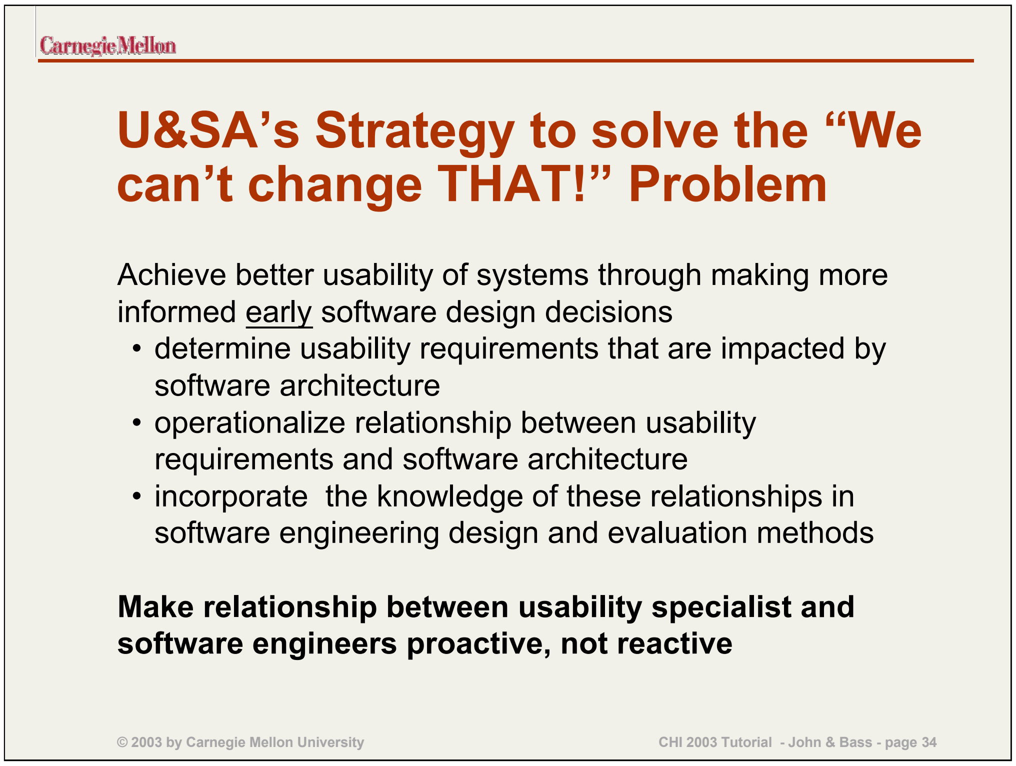
Outline of tutorial

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

Discuss known solutions for supporting changes in the screen-deep user interface and why they don’t work for deeper changes.

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

Applying U&SA to architecture evaluation and design



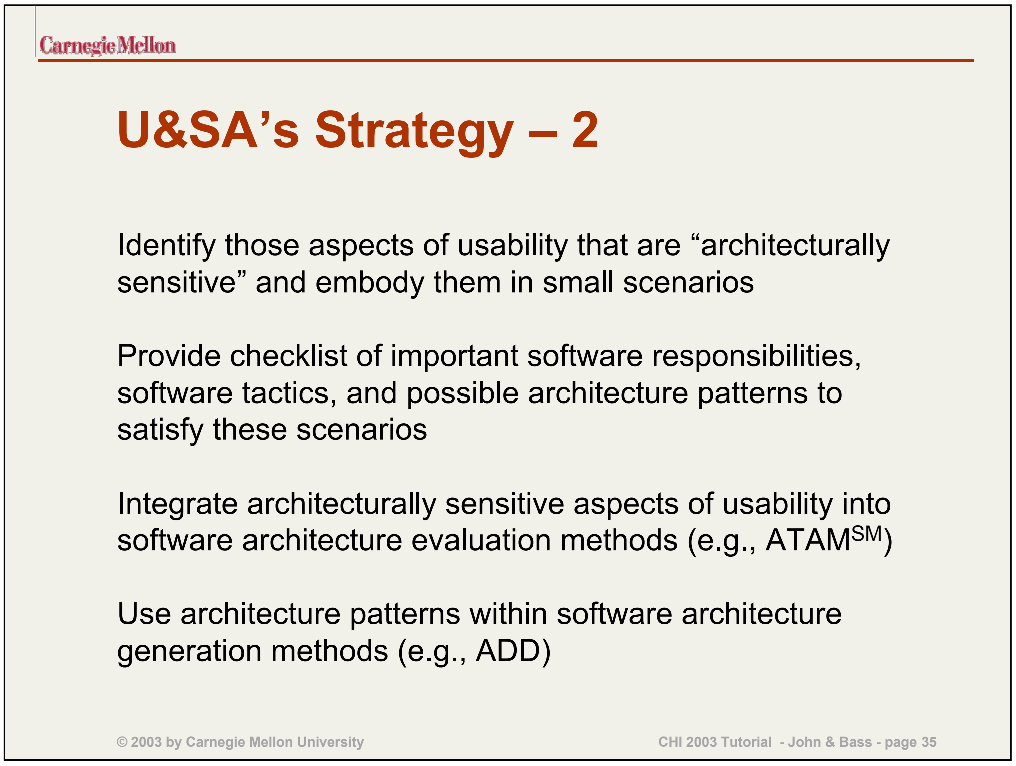
Carnegie Mellon

U&SA’s Strategy to solve the “We can’t change THAT!” Problem

Achieve better usability of systems through making more informed <u>early</u> software design decisions

- determine usability requirements that are impacted by software architecture
- operationalize relationship between usability requirements and software architecture
- incorporate the knowledge of these relationships in software engineering design and evaluation methods

Make relationship between usability specialist and software engineers proactive, not reactive


Carnegie Mellon

U&SA's Strategy – 2

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

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

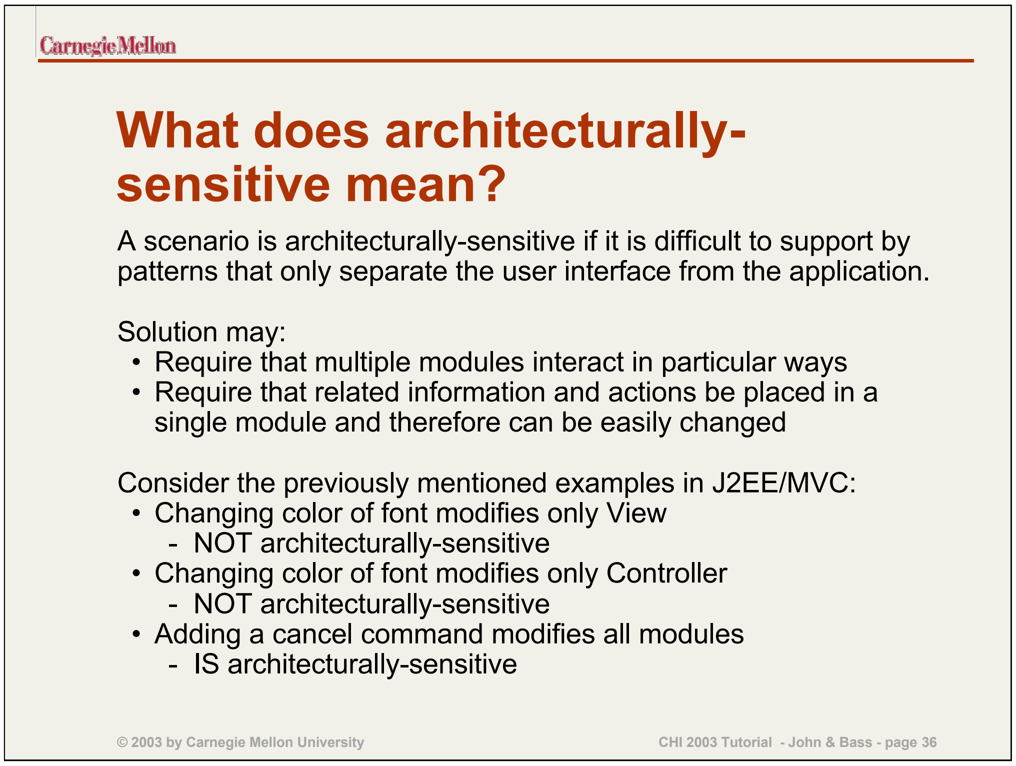
Integrate architecturally sensitive aspects of usability into software architecture evaluation methods (e.g., ATAM^{SM})

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

© 2003 by Carnegie Mellon University

CHI 2003 Tutorial - John & Bass - page 35

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

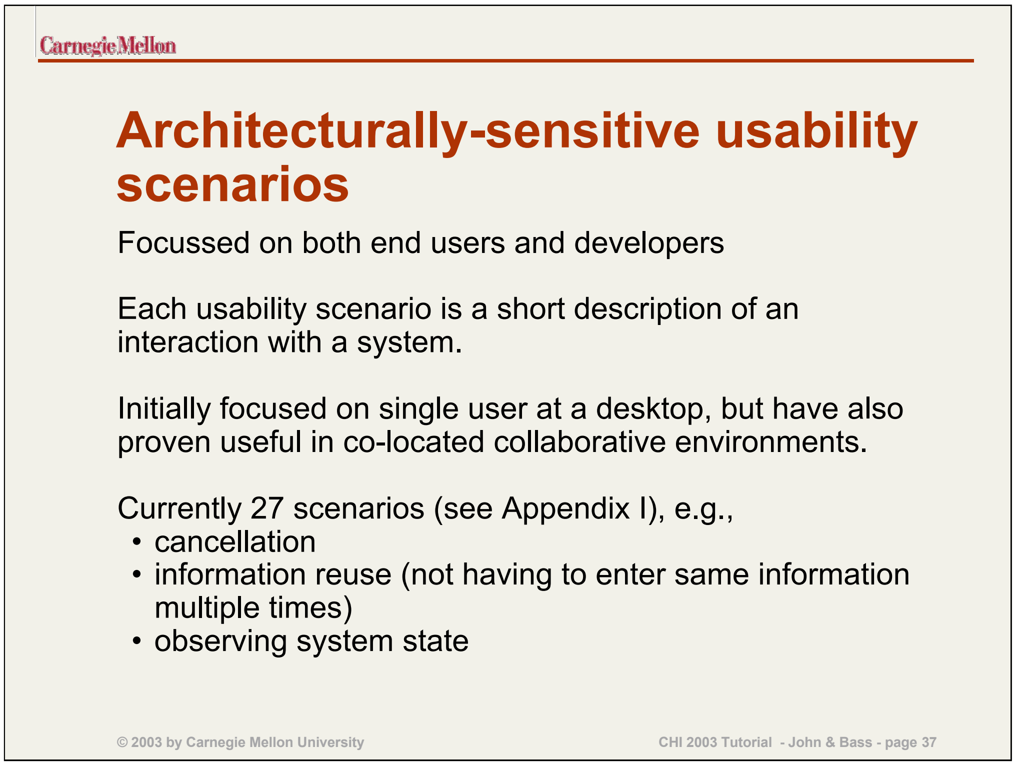

Carnegie Mellon

What does architecturally-sensitive mean?

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

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

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



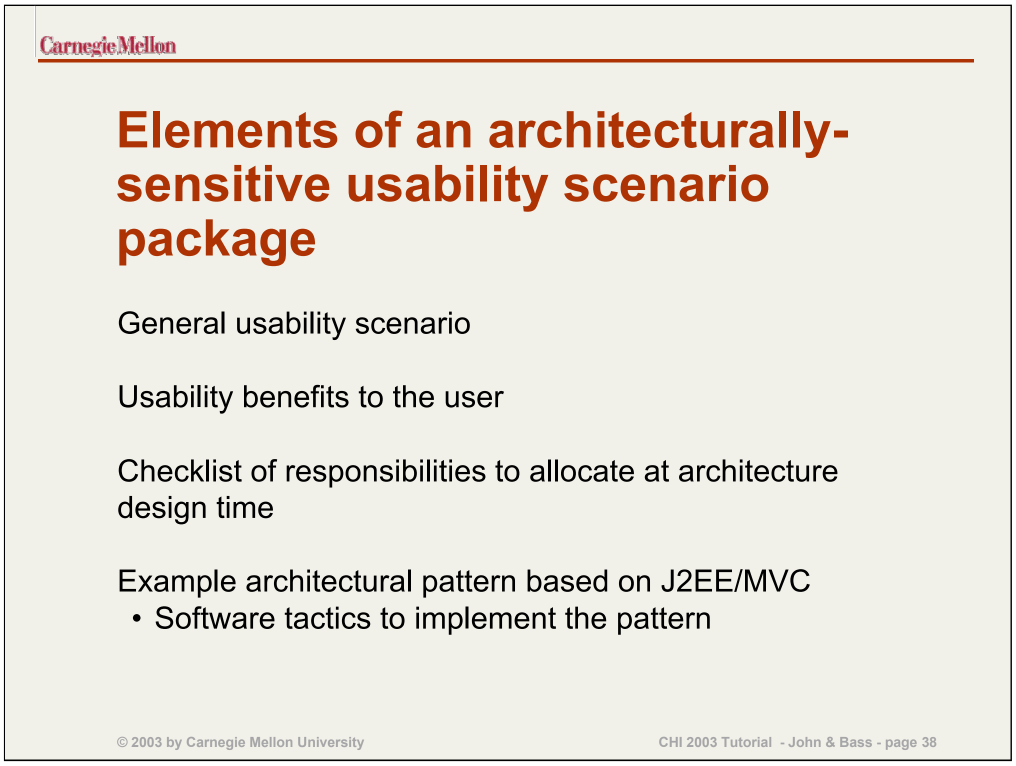
Architecturally-sensitive usability scenarios

Focussed on both end users and developers

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

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

Currently 27 scenarios (see Appendix I), e.g.,
- cancellation
- information reuse (not having to enter same information multiple times)
- observing system state



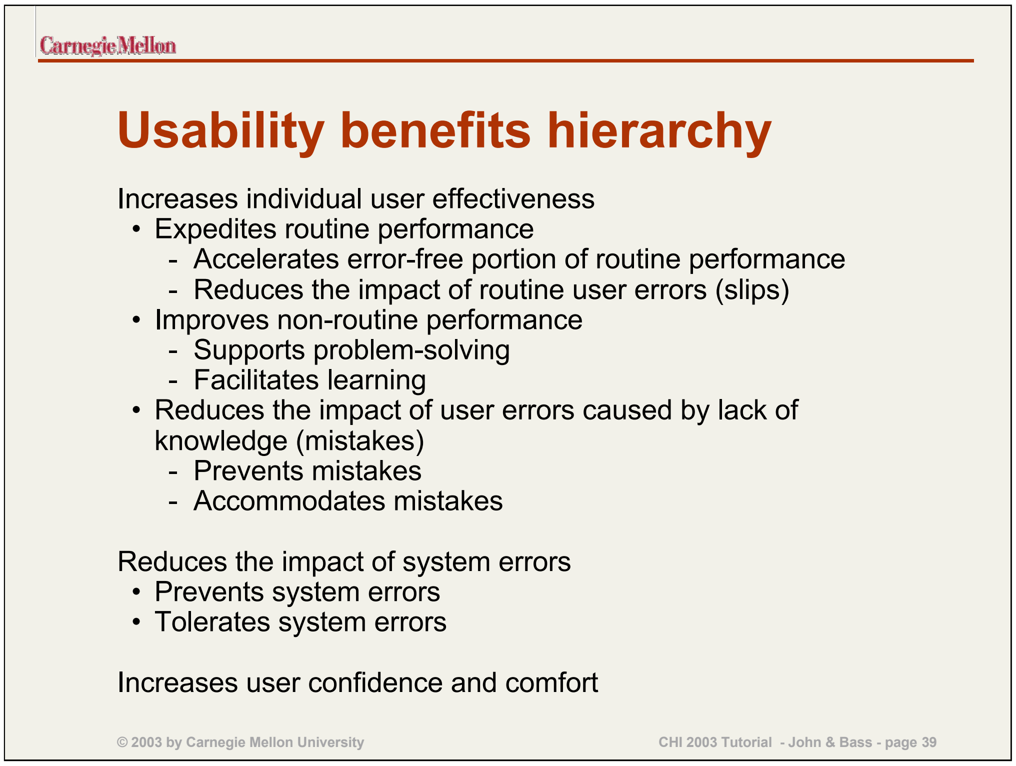
Elements of an architecturally-sensitive usability scenario package

General usability scenario

Usability benefits to the user

Checklist of responsibilities to allocate at architecture design time

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



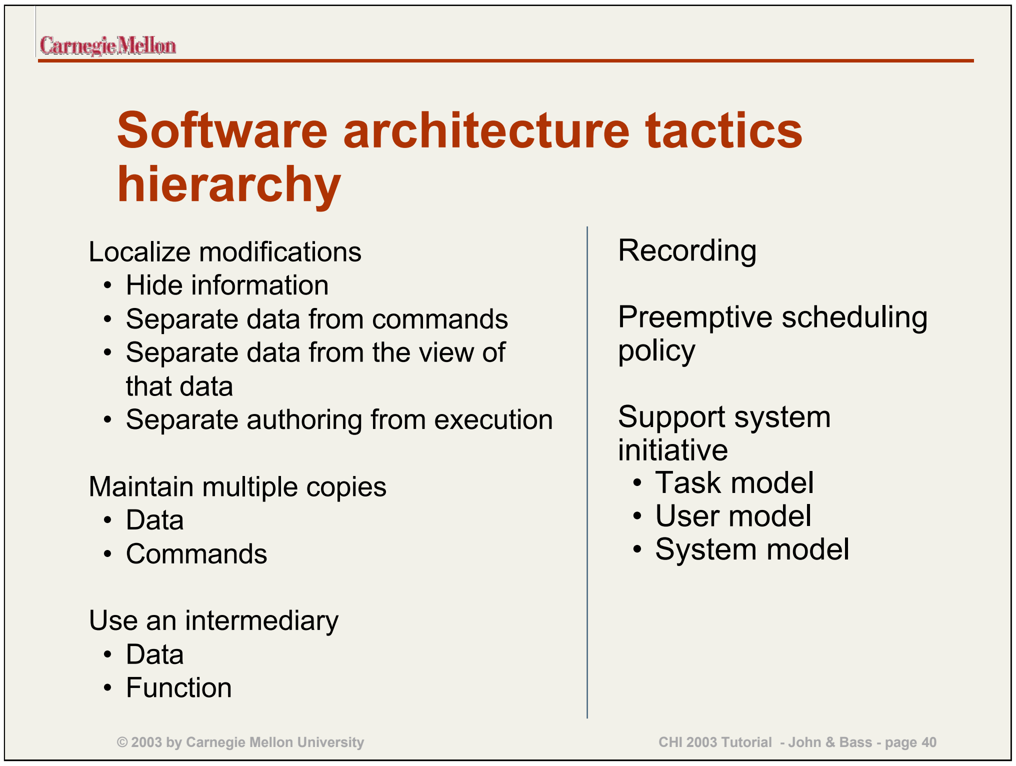
Carnegie Mellon

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



Software architecture tactics hierarchy

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

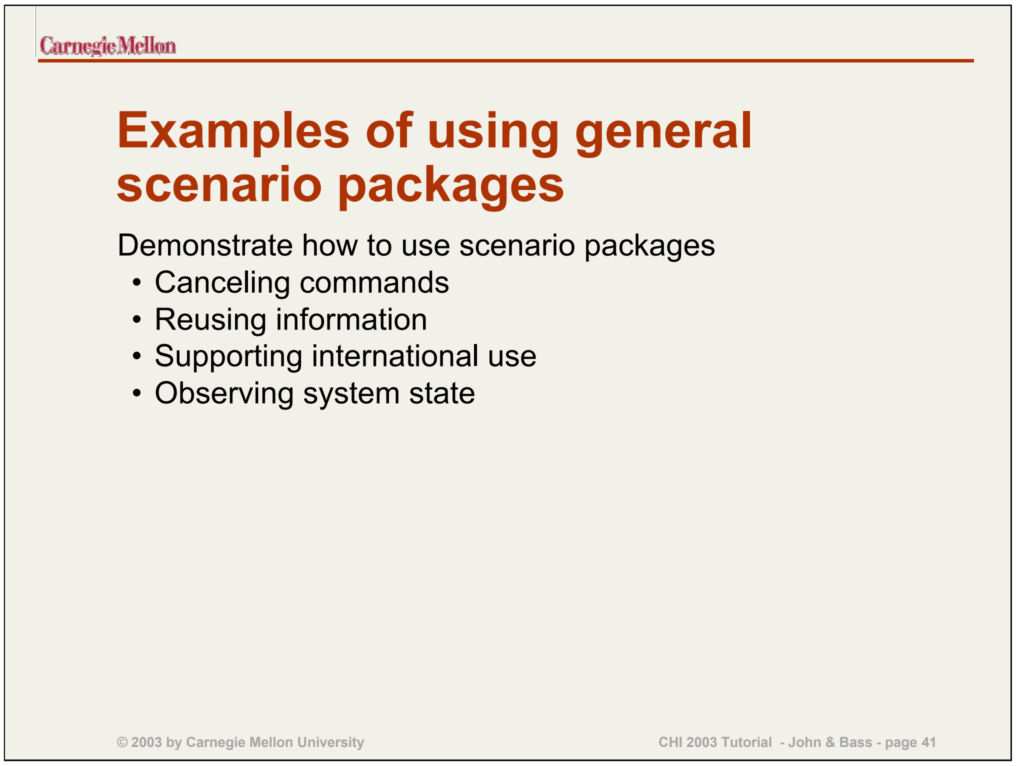
Maintain multiple copies
- Data
- Commands

Use an intermediary
- Data
- Function

Recording

Preemptive scheduling policy

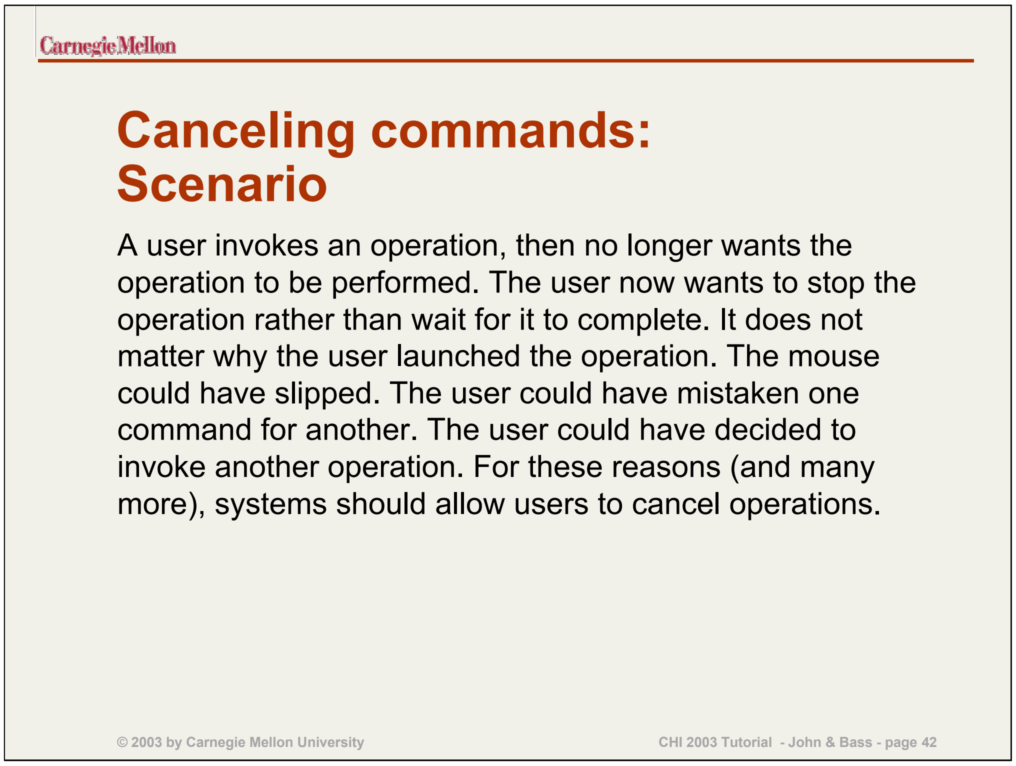
Support system initiative
- Task model
- User model
- System model



Examples of using general scenario packages

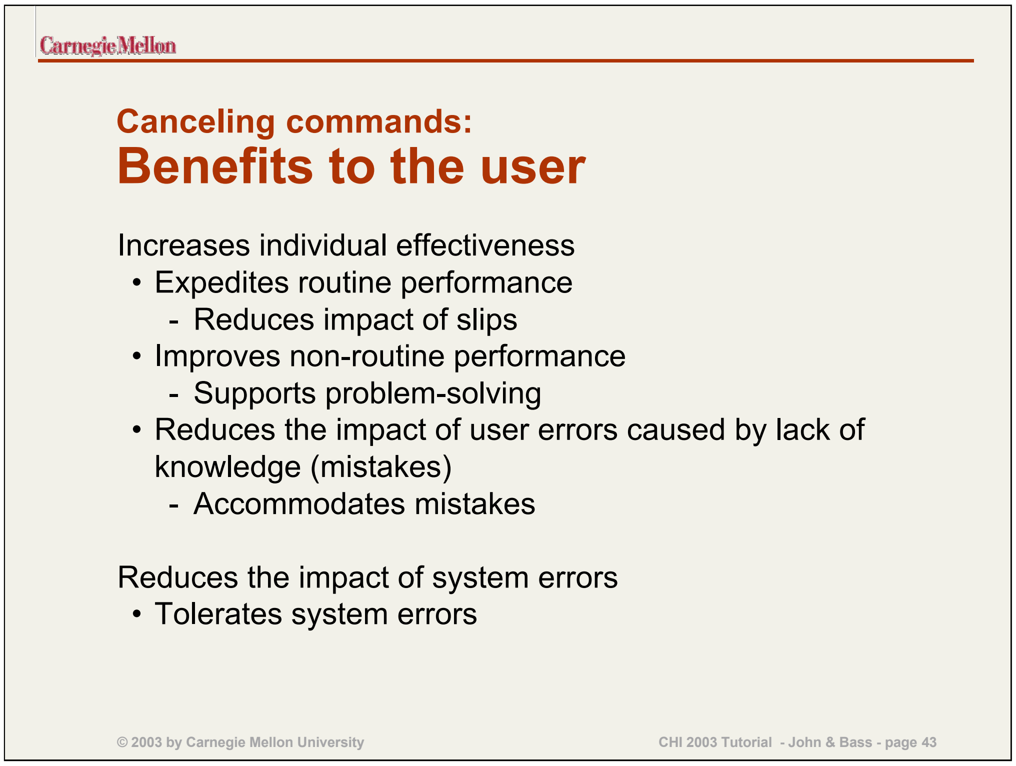
Demonstrate how to use scenario packages

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


Carnegie Mellon

Canceling commands: Scenario

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.


Carnegie Mellon

Canceling commands: Benefits to the user

Increases individual effectiveness
- Expedites routine performance
  - Reduces impact of slips
- Improves non-routine performance
  - Supports problem-solving
- Reduces the impact of user errors caused by lack of knowledge (mistakes)
  - Accommodates mistakes

Reduces the impact of system errors
- Tolerates system errors

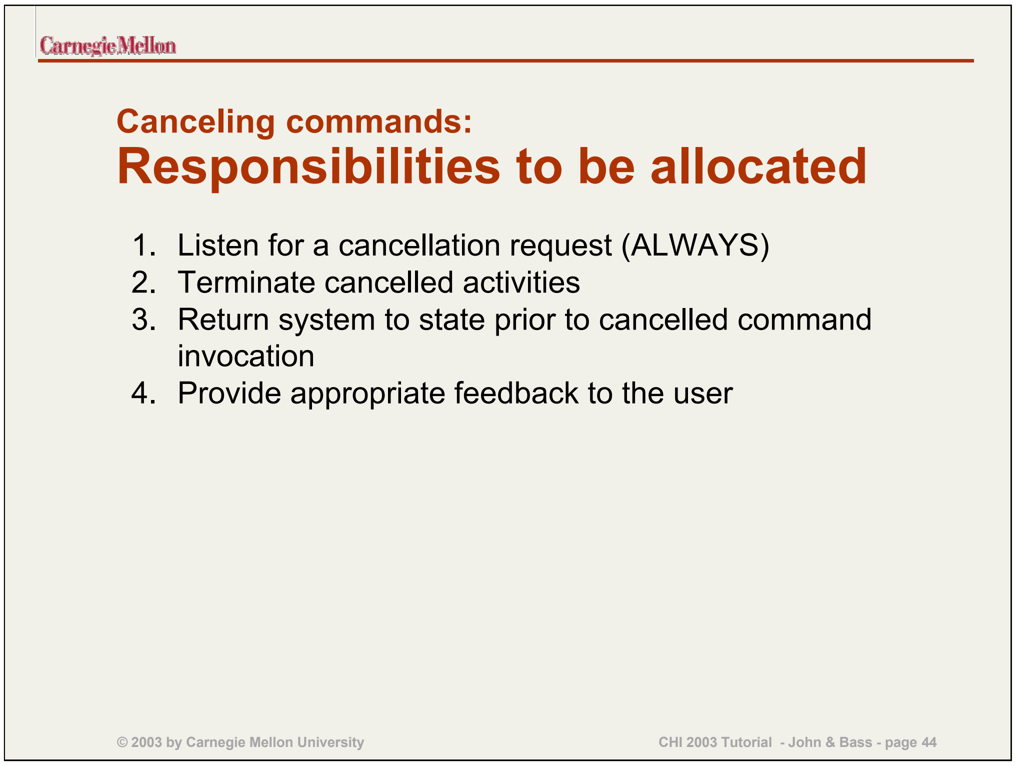
© 2003 by Carnegie Mellon University CHI 2003 Tutorial - John & Bass - page 43

Increases individual effectiveness
Expedites routine performance
Reduces impact of slips
Cancellation reduces the impact of slips by allowing users to revoke accidental commands.

Increases individual effectiveness
Improves non-routine performance
Supports problem-solving
Cancellation facilitates problem-solving by allowing users to apply commands and explore without fear, because they can always abort their actions.

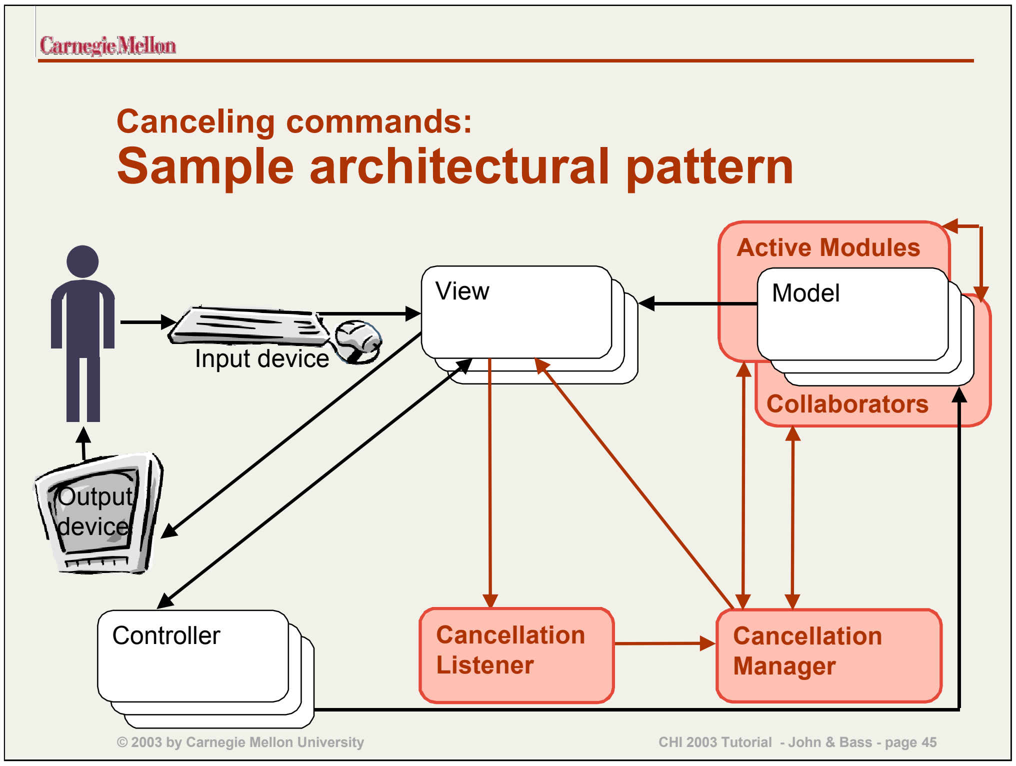
Increases individual effectiveness
Reduces the impact of user errors caused by lack of knowledge (mistakes)
Accommodates mistakes
Cancellation accommodates user mistakes by allowing users to abort commands they invoke through lack of knowledge.

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


Carnegie Mellon

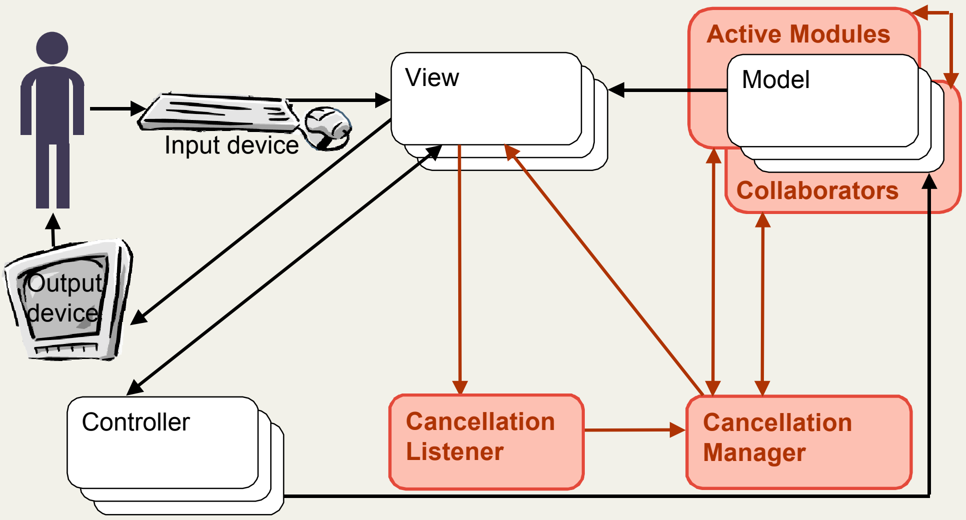
Canceling commands: Responsibilities to be allocated

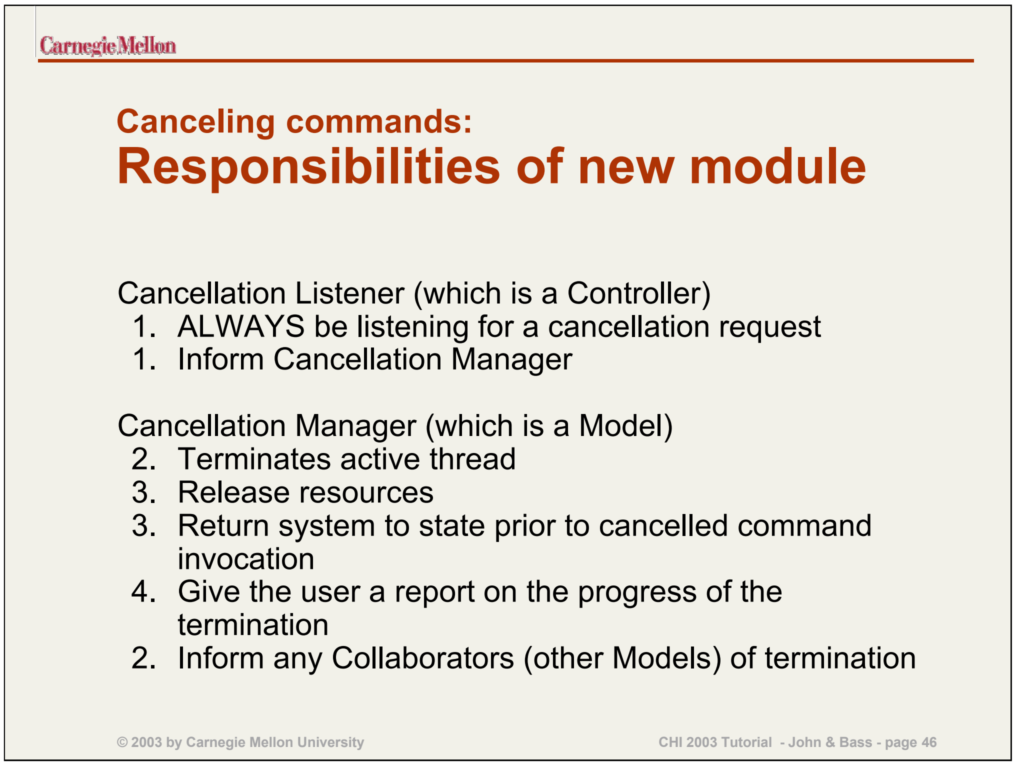
1. Listen for a cancellation request (ALWAYS)
2. Terminate cancelled activities
3. Return system to state prior to cancelled command invocation
4. Provide appropriate feedback to the user



Carnegie Mellon

Canceling commands:
Sample architectural pattern




Carnegie Mellon

Canceling commands: Responsibilities of new module

Cancellation Listener (which is a Controller)

1. ALWAYS be listening for a cancellation request
1. Inform Cancellation Manager

Cancellation Manager (which is a Model)

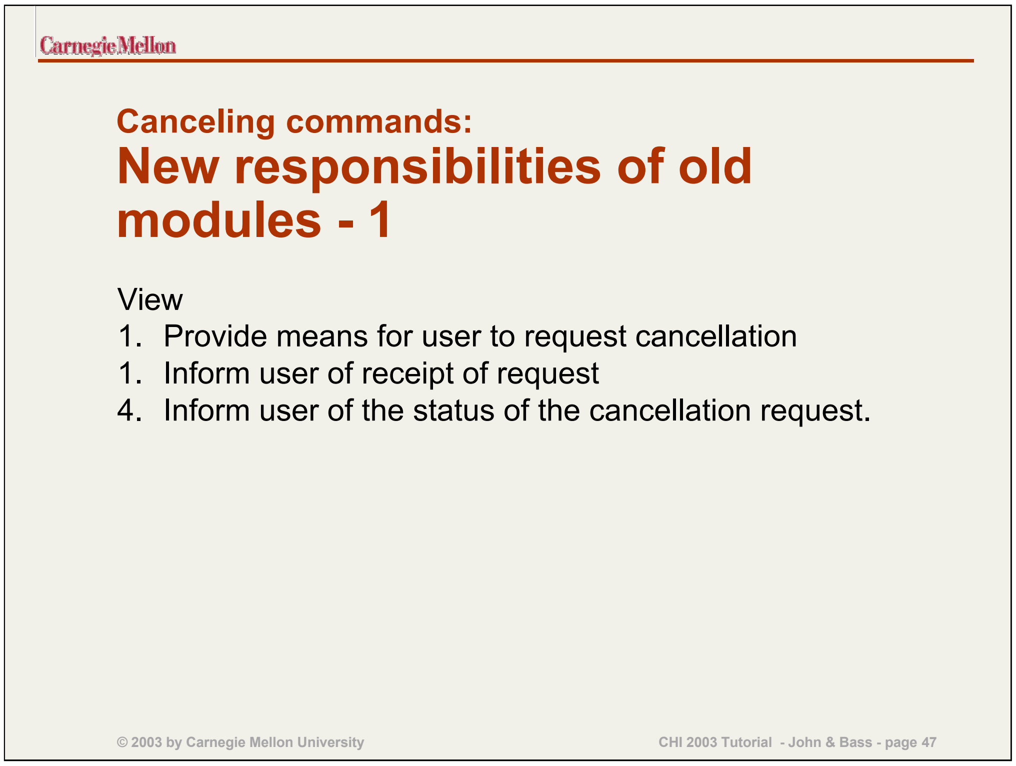
2. Terminates active thread
3. Release resources
3. Return system to state prior to cancelled command invocation
4. Give the user a report on the progress of the termination
2. Inform any Collaborators (other Models) of termination

© 2003 by Carnegie Mellon University

CHI 2003 Tutorial - John & Bass - page 46

Responsibilities to be allocated:

1. Listen for a cancellation request (ALWAYS)
2. Terminate cancelled activities
3. Return system to state prior to cancelled command invocation
4. Provide appropriate feedback to the user

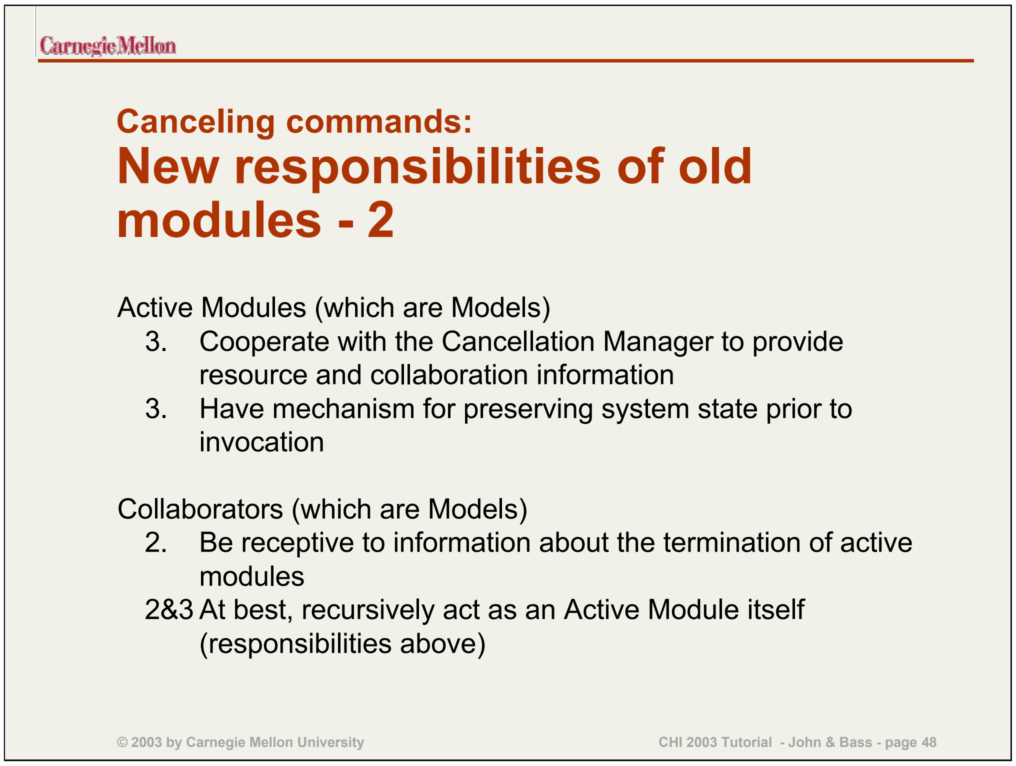

Carnegie Mellon

Canceling commands: New responsibilities of old modules - 1

View
1. Provide means for user to request cancellation
1. Inform user of receipt of request
4. Inform user of the status of the cancellation request.

Responsibilities to be allocated:

1. Listen for a cancellation request (ALWAYS)
2. Terminate cancelled activities
3. Return system to state prior to cancelled command invocation
4. Provide appropriate feedback to the user



Carnegie Mellon

Canceling commands: New responsibilities of old modules - 2

Active Modules (which are Models)

3. Cooperate with the Cancellation Manager to provide resource and collaboration information
3. Have mechanism for preserving system state prior to invocation

Collaborators (which are Models)

2. Be receptive to information about the termination of active modules

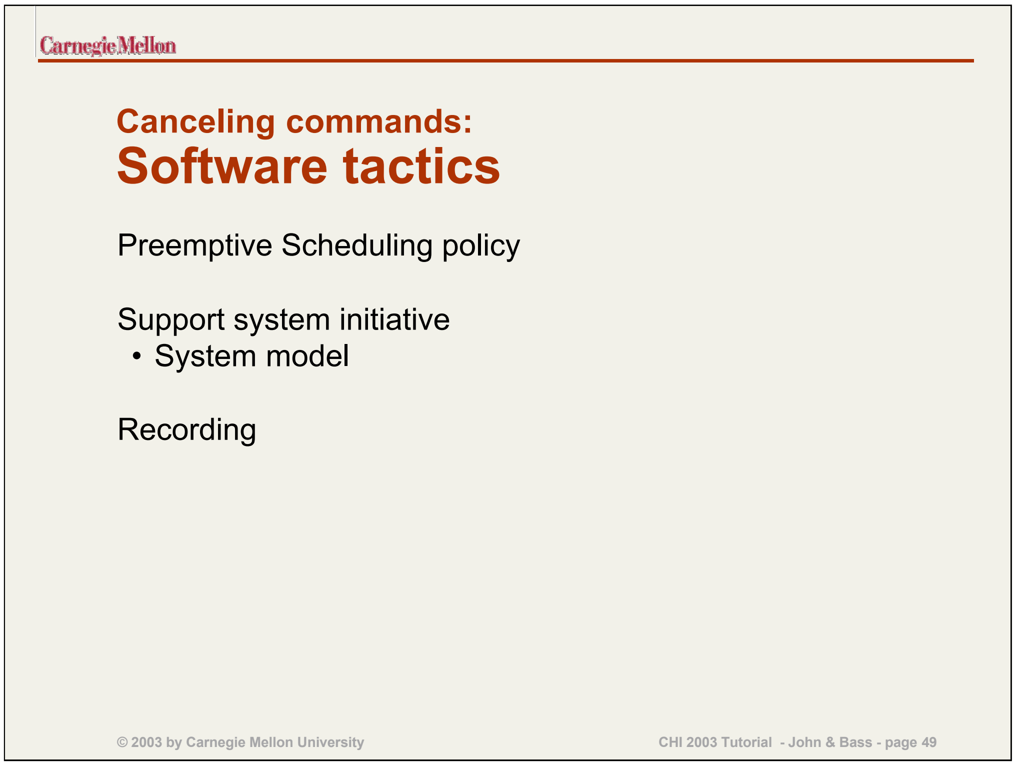
2&3 At best, recursively act as an Active Module itself (responsibilities above)

© 2003 by Carnegie Mellon University

CHI 2003 Tutorial - John & Bass - page 48

Responsibilities to be allocated:

1. Listen for a cancellation request (ALWAYS)
2. Terminate cancelled activities
3. Return system to state prior to cancelled command invocation
4. Provide appropriate feedback to the user


Carnegie Mellon

Canceling commands: Software tactics

Preemptive Scheduling policy

Support system initiative
- System model

Recording

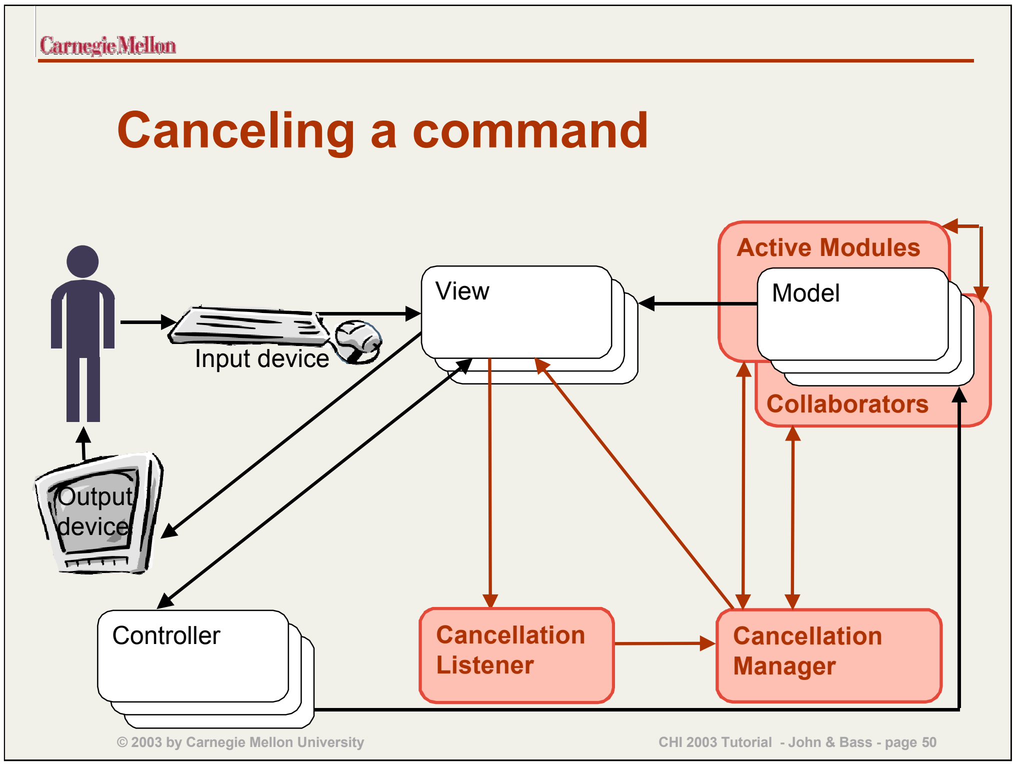
© 2003 by Carnegie Mellon University

CHI 2003 Tutorial - John & Bass - page 49

Preemptive Scheduling
To adequately implement cancellation, the Cancellation Listener and Cancellation Controller must occupy independent threads.

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

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


Carnegie Mellon

Canceling a command

© 2003 by Carnegie Mellon University

CHI 2003 Tutorial - John & Bass - page 50

Step 1. User selects the option to cancel a running command.

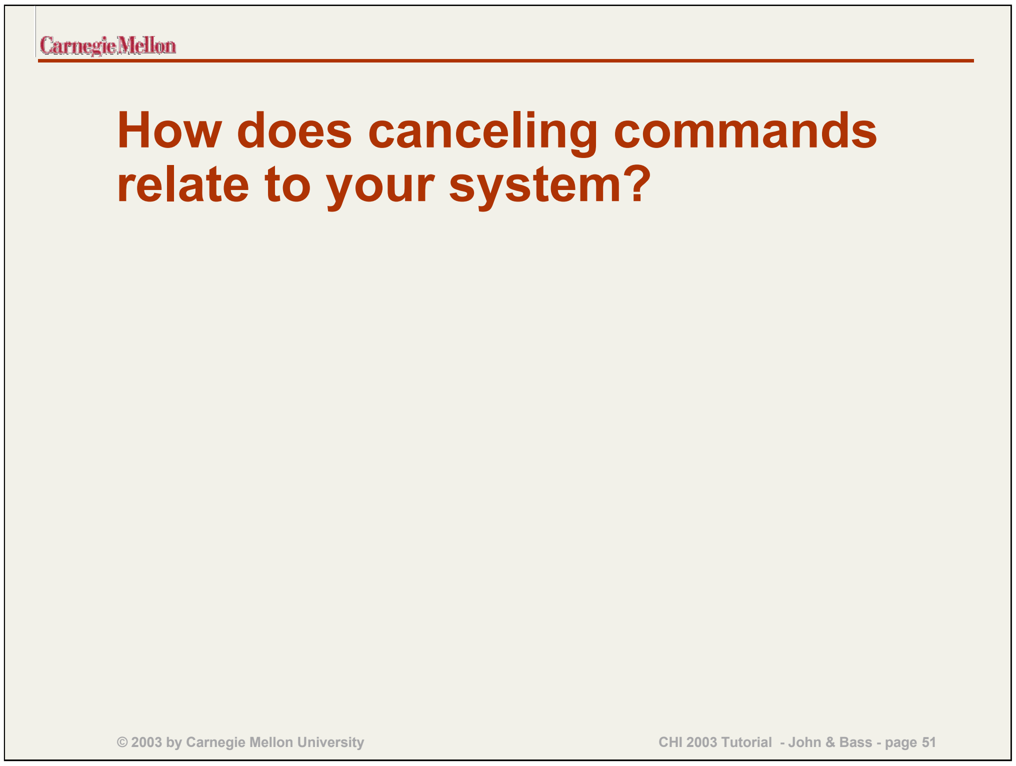
Step 2. The View receives this command and passes it to the Cancellation listener, which is waiting on a separate thread, then notifies the user that the cancellation command was received.

Step 3. The Cancellation Listener notifies the Cancellation Manager that the selected command must be terminated.

Step 4. The Cancellation Manager notifies the Active Module and its Collaborators that their current activity must be terminated. There are two cases. Case 1: The active module can respond to a cancel command. In this case it cleans up and terminates itself. Case 2: The active module can't respond and the Cancellation Manager tells the operating system to kill the Active Module and the Cancellation Manager cleans up after the Active Module.

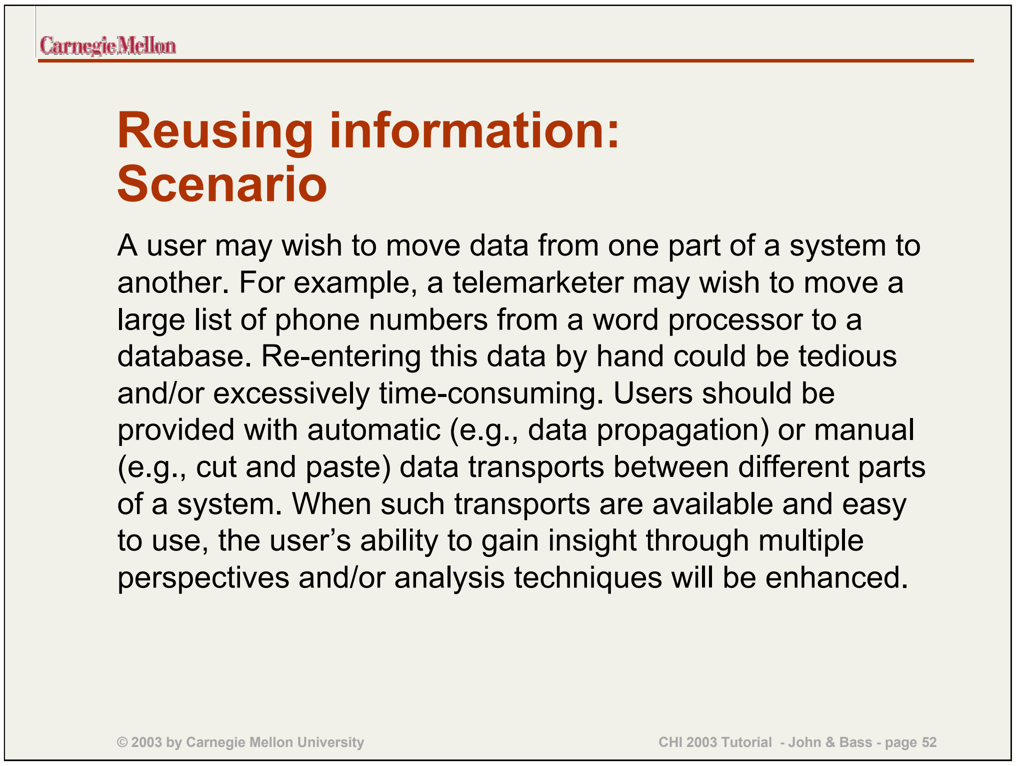
Step 5. The Cancellation Manager notifies the View about the progress of the command's cancellation.

Step 6. The View notifies the user about the progress of the command's cancellation.



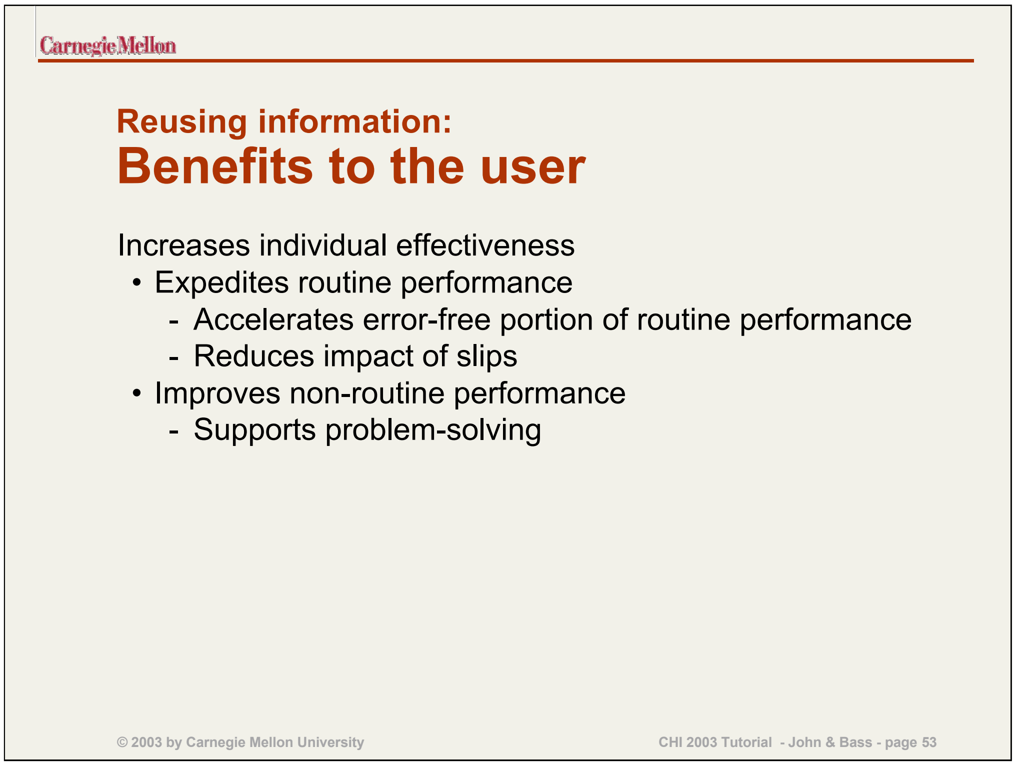
Carnegie Mellon

How does canceling commands relate to your system?



Reusing information: Scenario

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.


Carnegie Mellon

Reusing information:
Benefits to the user

Increases individual effectiveness
- Expedites routine performance
  - Accelerates error-free portion of routine performance
  - Reduces impact of slips
- Improves non-routine performance
  - Supports problem-solving

Increases individual effectiveness
Expedites routine performance
Accelerates error-free portion of routine performance

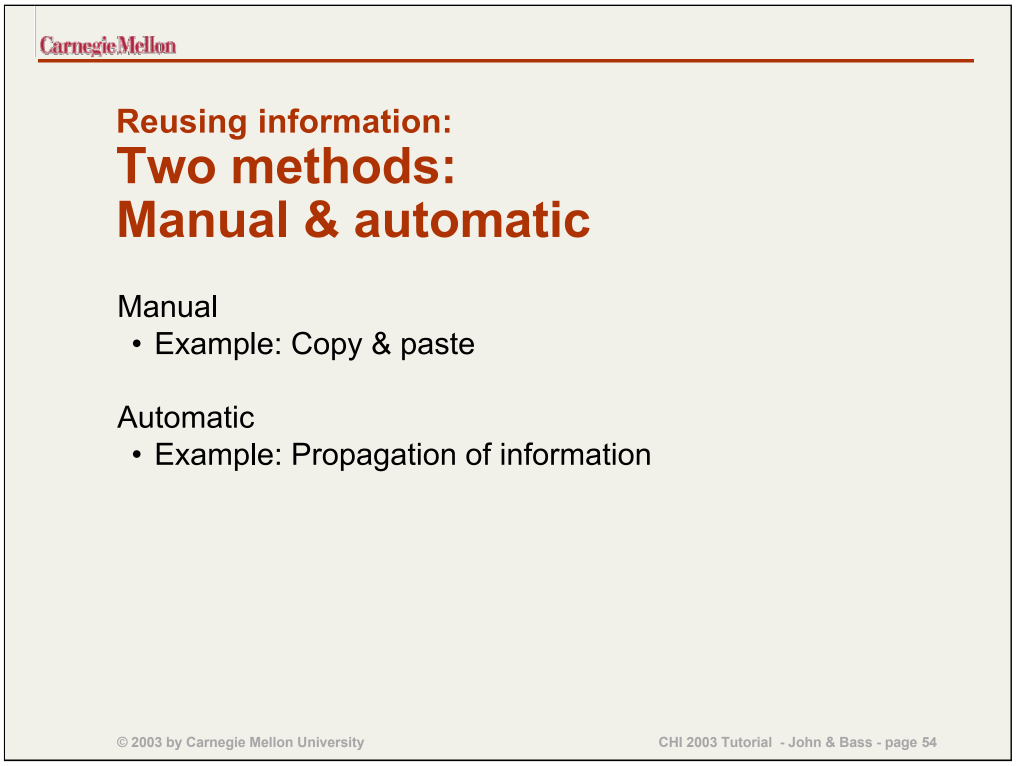
In most cases, it is more efficient for systems to transport information from place to place than it is for users to re-enter this information by hand. Thus, systems that support information reuse accelerate routine performance.

Increases individual effectiveness
Expedites routine performance
Reduces impact of slips

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

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

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



Carnegie Mellon

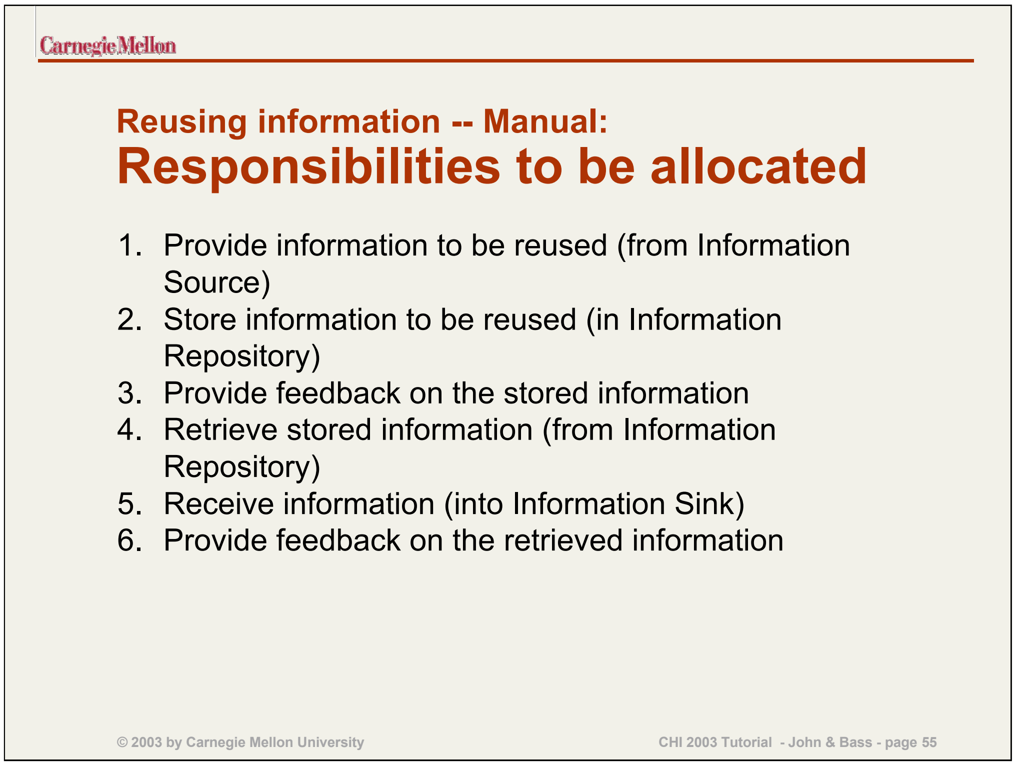
Reusing information: Two methods: Manual & automatic

Manual

- Example: Copy & paste

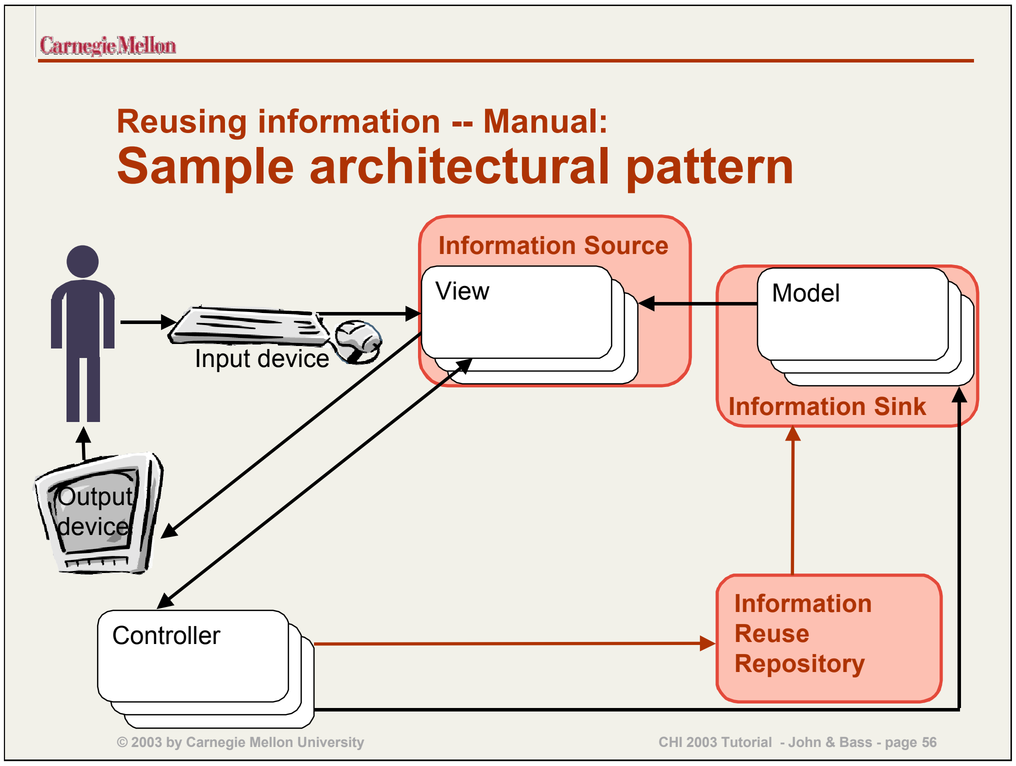
Automatic

- Example: Propagation of information

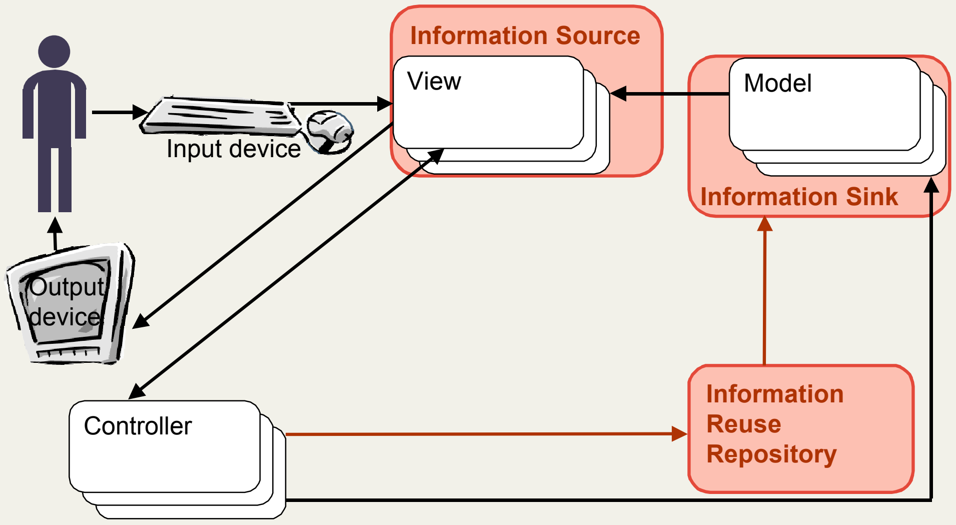

Carnegie Mellon

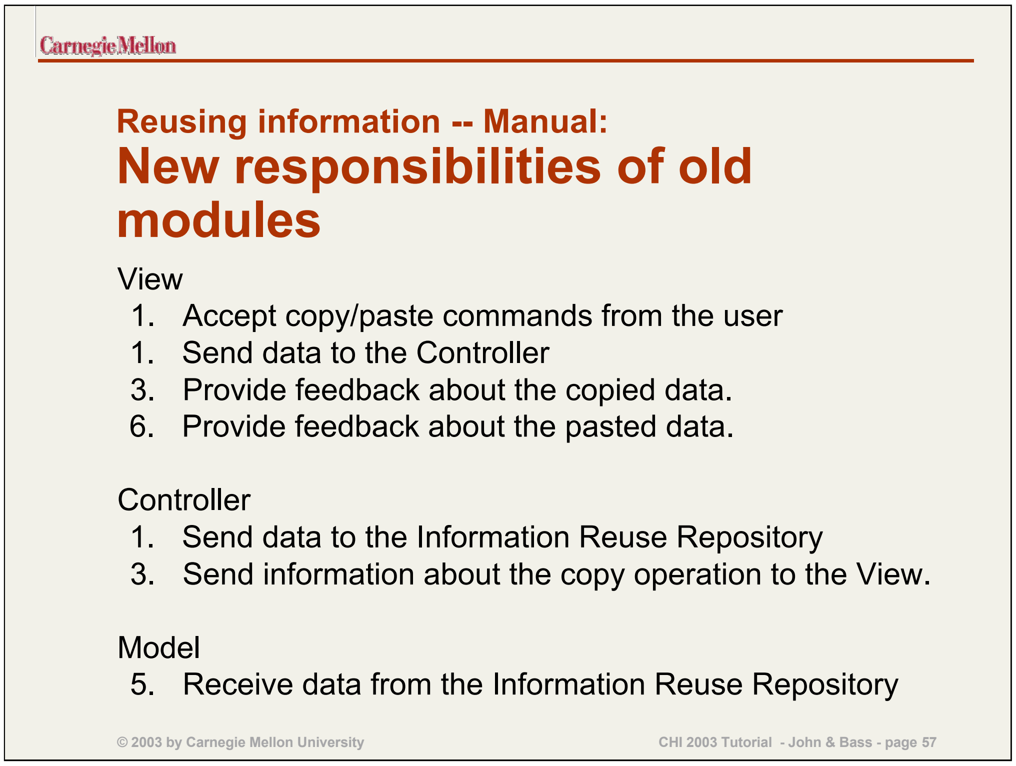
Reusing information -- Manual: Responsibilities to be allocated

1. Provide information to be reused (from Information Source)
2. Store information to be reused (in Information Repository)
3. Provide feedback on the stored information
4. Retrieve stored information (from Information Repository)
5. Receive information (into Information Sink)
6. Provide feedback on the retrieved information


Carnegie Mellon

Reusing information -- Manual:
Sample architectural pattern




Carnegie Mellon

Reusing information -- Manual: New responsibilities of old modules

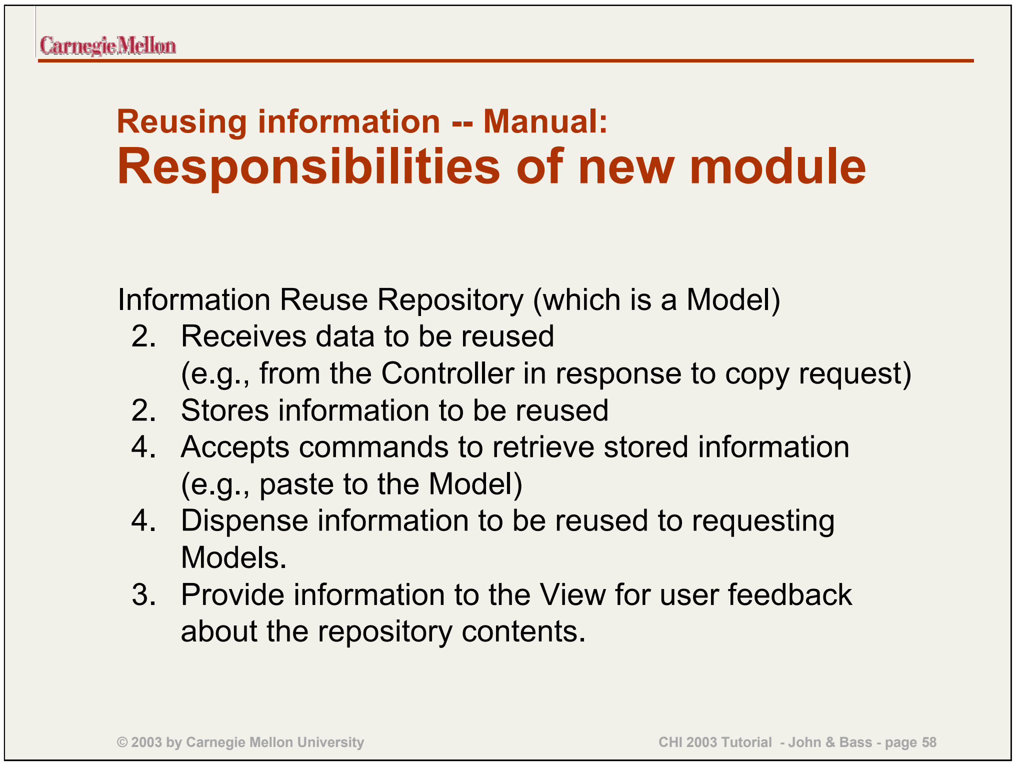
View
1. Accept copy/paste commands from the user
1. Send data to the Controller
3. Provide feedback about the copied data.
6. Provide feedback about the pasted data.

Controller
1. Send data to the Information Reuse Repository
3. Send information about the copy operation to the View.

Model
5. Receive data from the Information Reuse Repository

Responsibilities to be allocated:

1. Provide information to be reused (from Information Source)
2. Store information to be reused (in Information Repository)
3. Provide feedback on the stored information
4. Retrieve stored information (from Information Repository)
5. Receive information (into Information Sink)
6. Provide feedback on the retrieved information



Carnegie Mellon

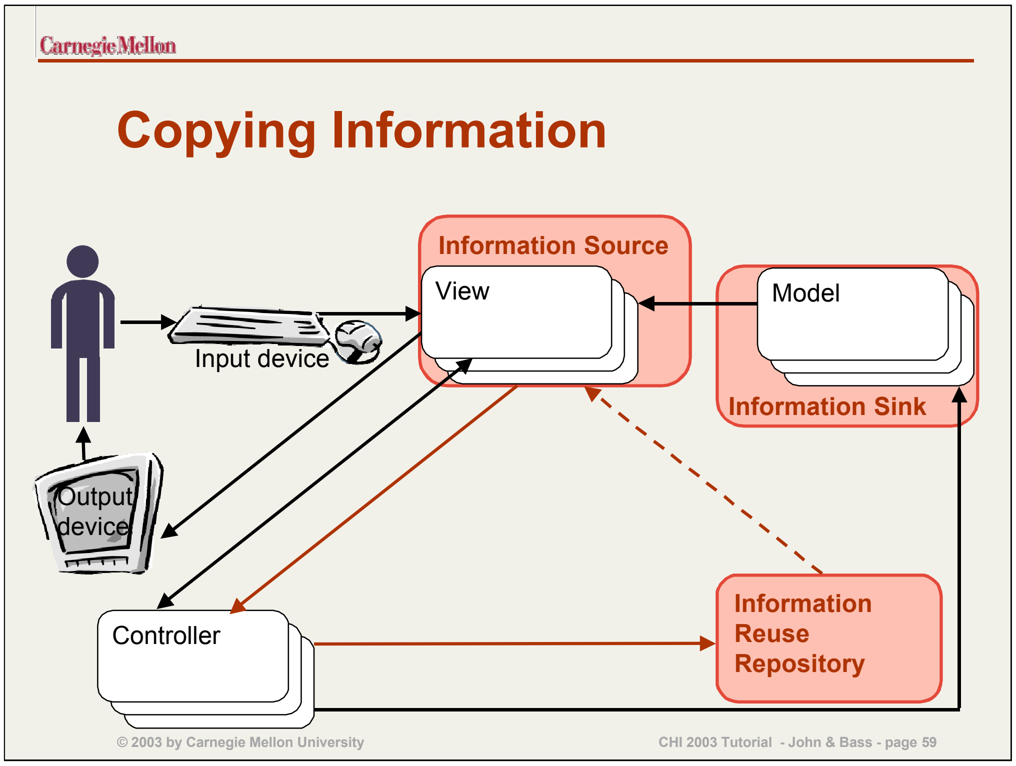
Reusing information -- Manual: Responsibilities of new module

Information Reuse Repository (which is a Model)

2. Receives data to be reused
   (e.g., from the Controller in response to copy request)
2. Stores information to be reused
4. Accepts commands to retrieve stored information
   (e.g., paste to the Model)
4. Dispense information to be reused to requesting Models.
3. Provide information to the View for user feedback about the repository contents.

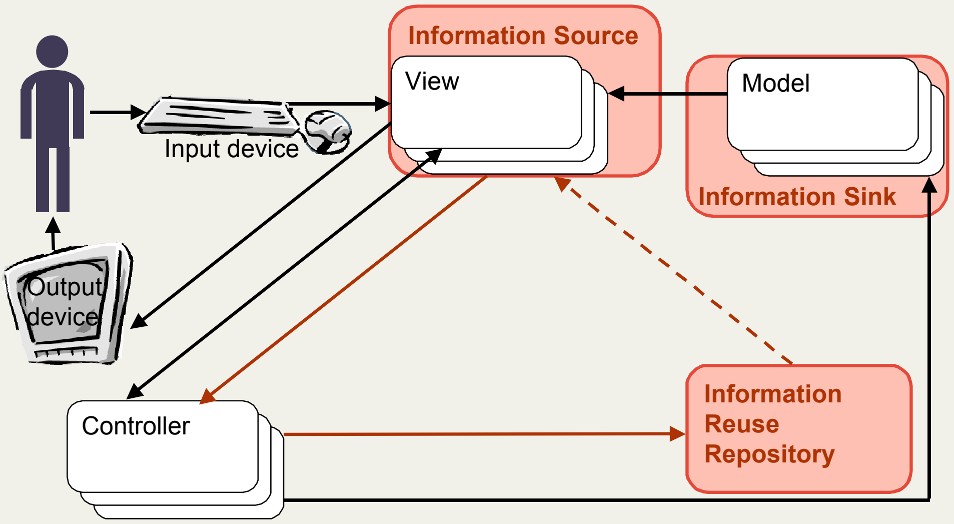
Responsibilities to be allocated:

1. Provide information to be reused (from Information Source)
2. Store information to be reused (in Information Repository)
3. Provide feedback on the stored information
4. Retrieve stored information (from Information Repository)
5. Receive information (into Information Sink)
6. Provide feedback on the retrieved information



Carnegie Mellon

Copying Information



© 2003 by Carnegie Mellon University

CHI 2003 Tutorial - John & Bass - page 59

Step 1. The user issues the copy request to the View.

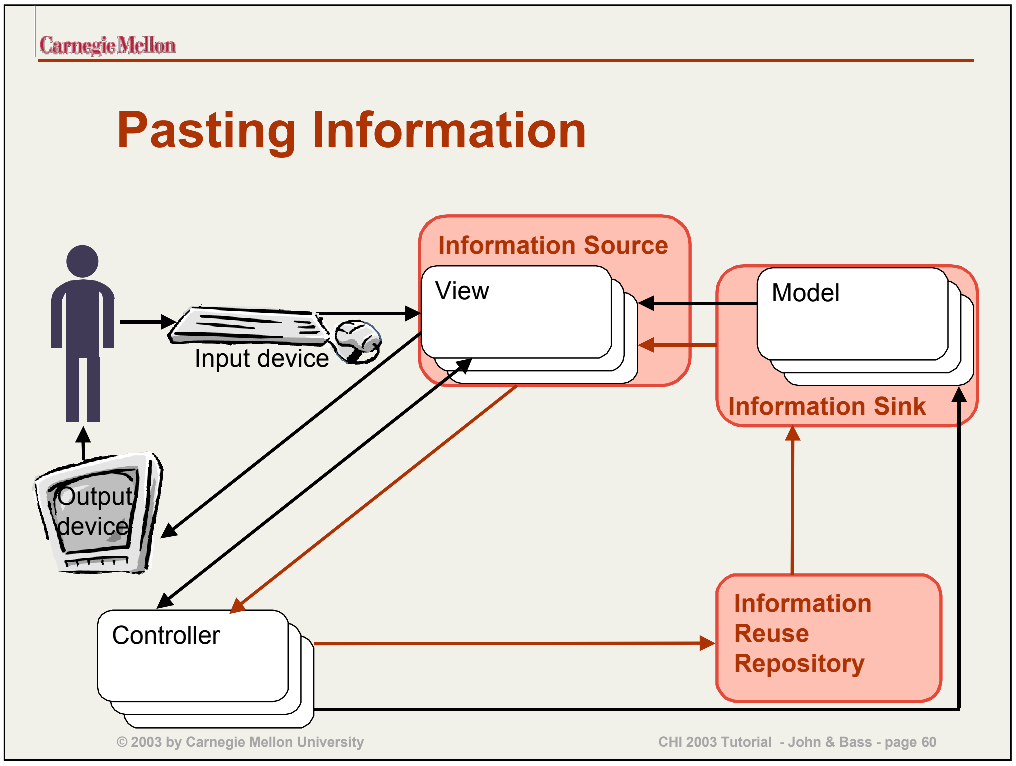
Step 2. The View forwards the copy request to the Controller for processing.

Step 3. The Controller obtains the data to copy from the Information Source, which is part of the View (we are assuming the data to copy is on the screen).

Step 4. The Controller sends the copied data to the Information Reuse Repository for temporary storage.

Step 5. The Controller messages the View to provide feedback to the user that the copy operation was successful.

Step 6. The View provides feedback to the user that the copy operation was successful.



Pasting Information

Step 1. The user issues the paste request to the View.

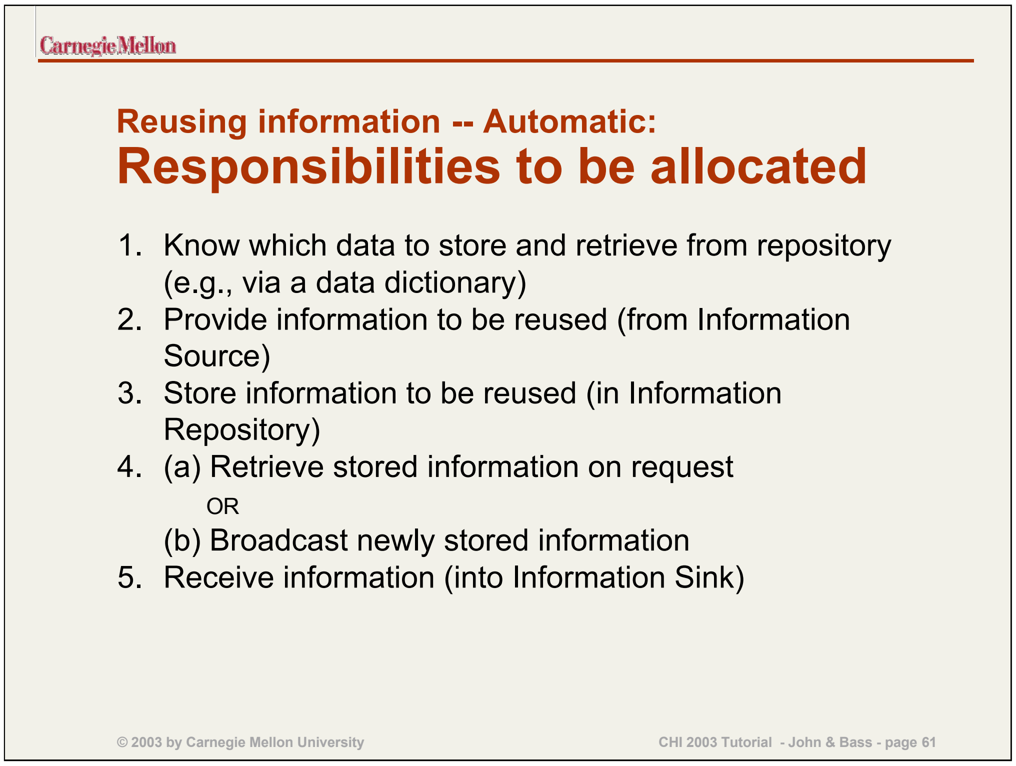
Step 2. The View forwards the paste request to the Controller for processing.

Step 3. The Controller sends the paste request and destination (Information Sink) information to the Information Reuse Repository.

Step 4. The Information Reuse Repository sends its current contents to the Information Sink.

Step 5. The Information Sink messages the View to provide feedback to the user that the paste operation was successful.

Step 6. The View provides feedback to the user that the paste operation was successful.

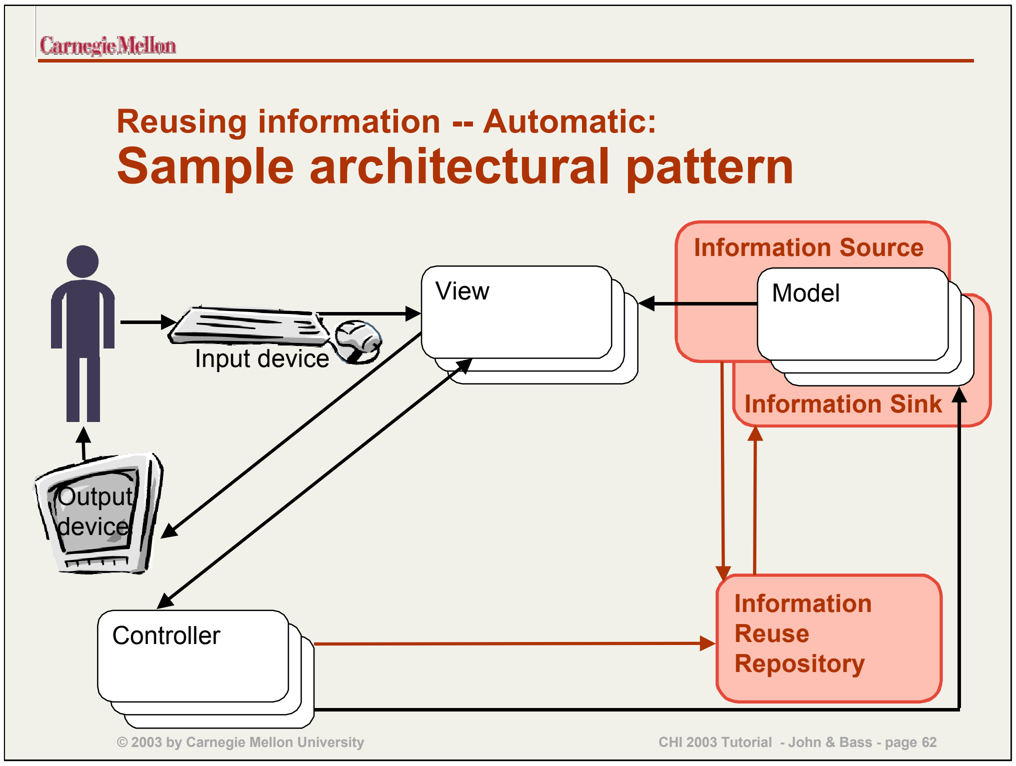

Carnegie Mellon

Reusing information -- Automatic: Responsibilities to be allocated

1. Know which data to store and retrieve from repository (e.g., via a data dictionary)
2. Provide information to be reused (from Information Source)
3. Store information to be reused (in Information Repository)
4. (a) Retrieve stored information on request

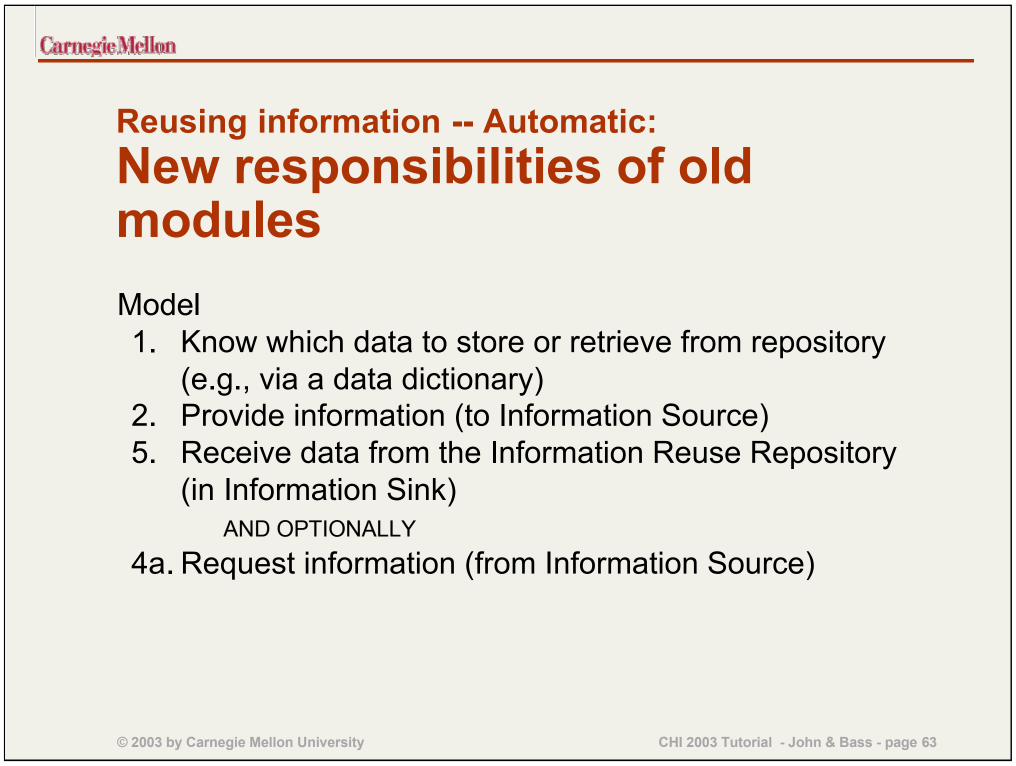
   OR

   (b) Broadcast newly stored information
5. Receive information (into Information Sink)



Carnegie Mellon

Reusing information -- Automatic: Sample architectural pattern


Carnegie Mellon

Reusing information -- Automatic: New responsibilities of old modules

Model

1. Know which data to store or retrieve from repository (e.g., via a data dictionary)
2. Provide information (to Information Source)
5. Receive data from the Information Reuse Repository (in Information Sink)

AND OPTIONALLY

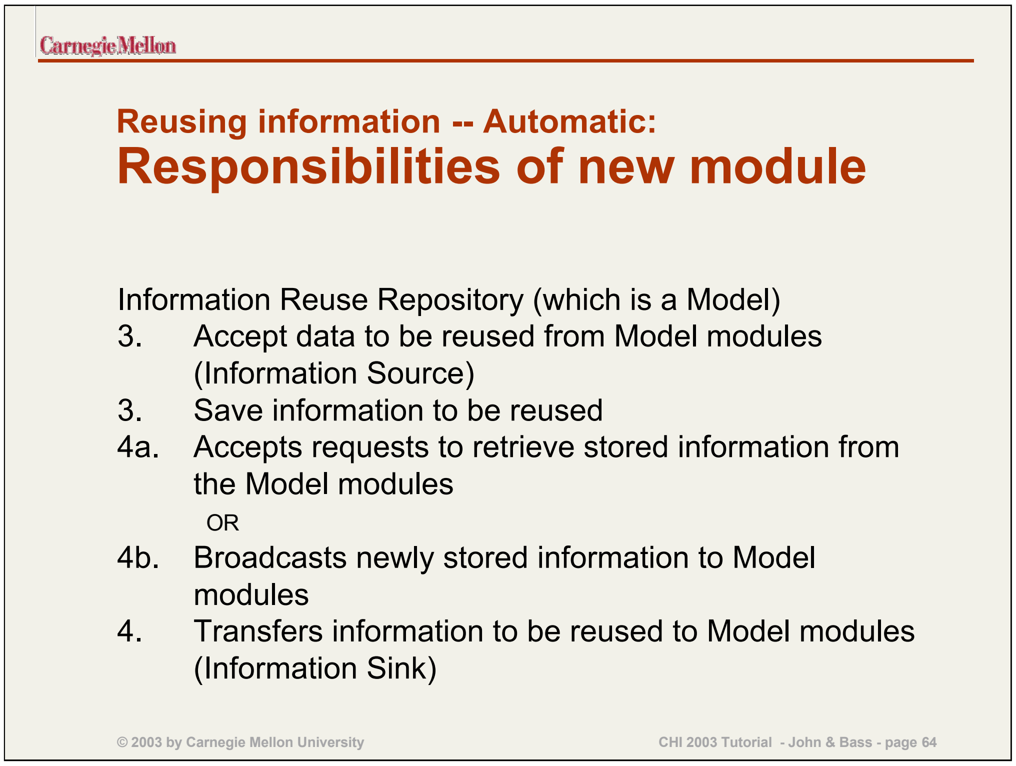
4a. Request information (from Information Source)

Responsibilities to be allocated:

1. Know which data to store and retrieve from repository (e.g., via a data dictionary)
2. Provide information to be reused (from Information Source)
3. Store information to be reused (in Information Repository)
4. (a) Retrieve stored information on request

   OR

   (b) Broadcasts newly stored information
5. Receive information (into Information Sink)


Carnegie Mellon

Reusing information -- Automatic:
Responsibilities of new module

Information Reuse Repository (which is a Model)

3. Accept data to be reused from Model modules (Information Source)
3. Save information to be reused
4a. Accepts requests to retrieve stored information from the Model modules

OR

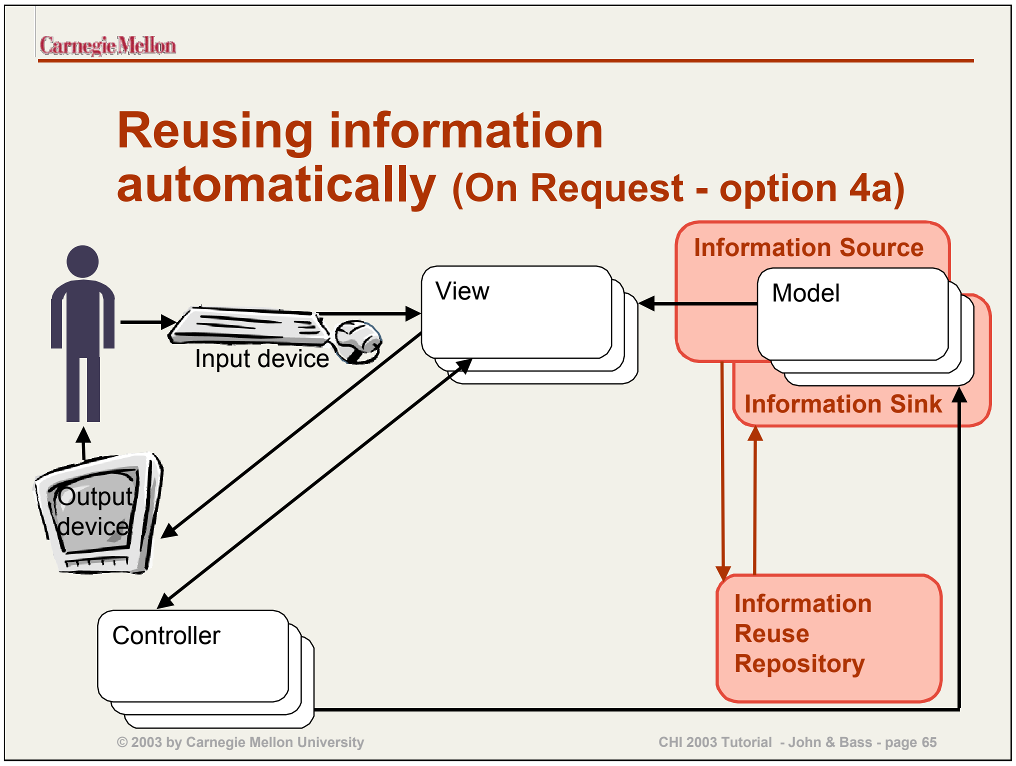
4b. Broadcasts newly stored information to Model modules
4. Transfers information to be reused to Model modules (Information Sink)

Responsibilities to be allocated:

1. Know which data to store and retrieve from repository (e.g., via a data dictionary)
2. Provide information to be reused (from Information Source)
3. Store information to be reused (in Information Repository)
4. (a) Retrieve stored information on request

   OR

   (b) Broadcasts newly stored information
5. Receive information (into Information Sink)


Carnegie Mellon

Reusing information automatically (On Request - option 4a)

Step 1. The user performs an action that requires information to be reused.

Step 2. The View sends this action to the Controller for interpretation.

Step 3. The Controller sends the action to the appropriate Model (Information Sink) for processing.

Step 4. The Information Sink recognizes that it requires certain reusable information to complete the request, perhaps by querying a data dictionary.

Step 5. The Information Sink requests this reusable information from the Information Reuse Repository.

Step 6. The Information Reuse Repository locates the appropriate Model (Information Source) that contains the reusable information and retrieves this information from it.

Step 7. The Information Reuse Repository returns this information to the Information Sink.

Step 8. The Information Sink uses this information to complete the processing of the user’s request and notifies the View of its changes.

Step 9. The View provides feedback to the user of the results of the action.

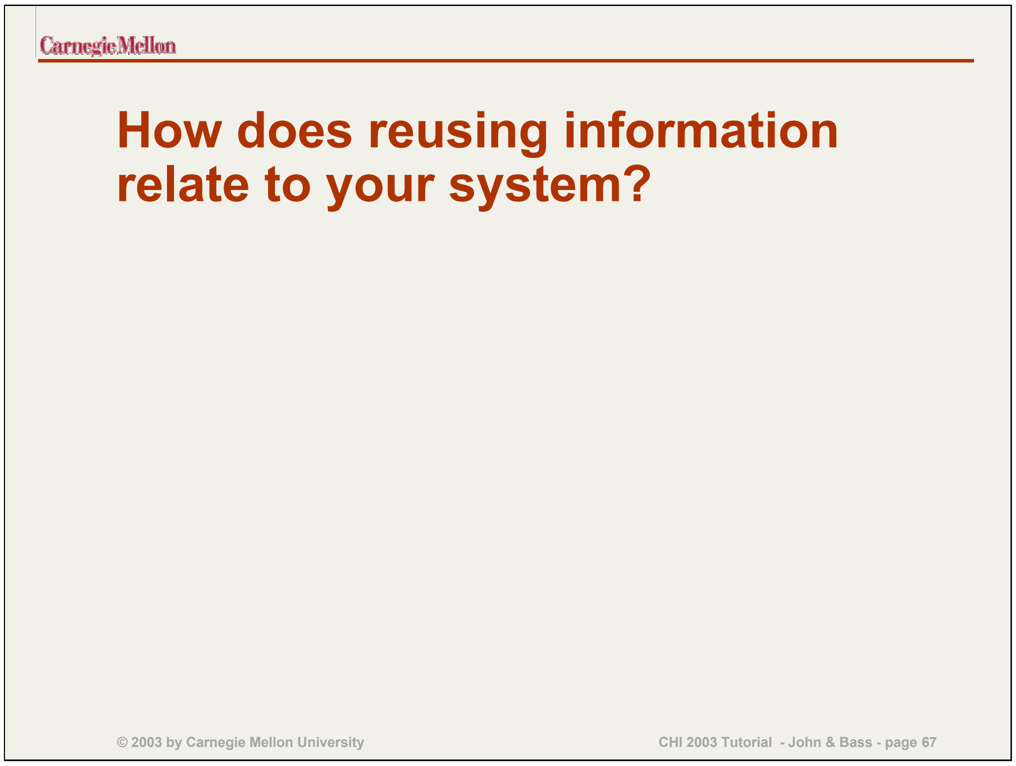
Carnegie Mellon

Reusing information: Software tactics

Use an intermediary
- Data

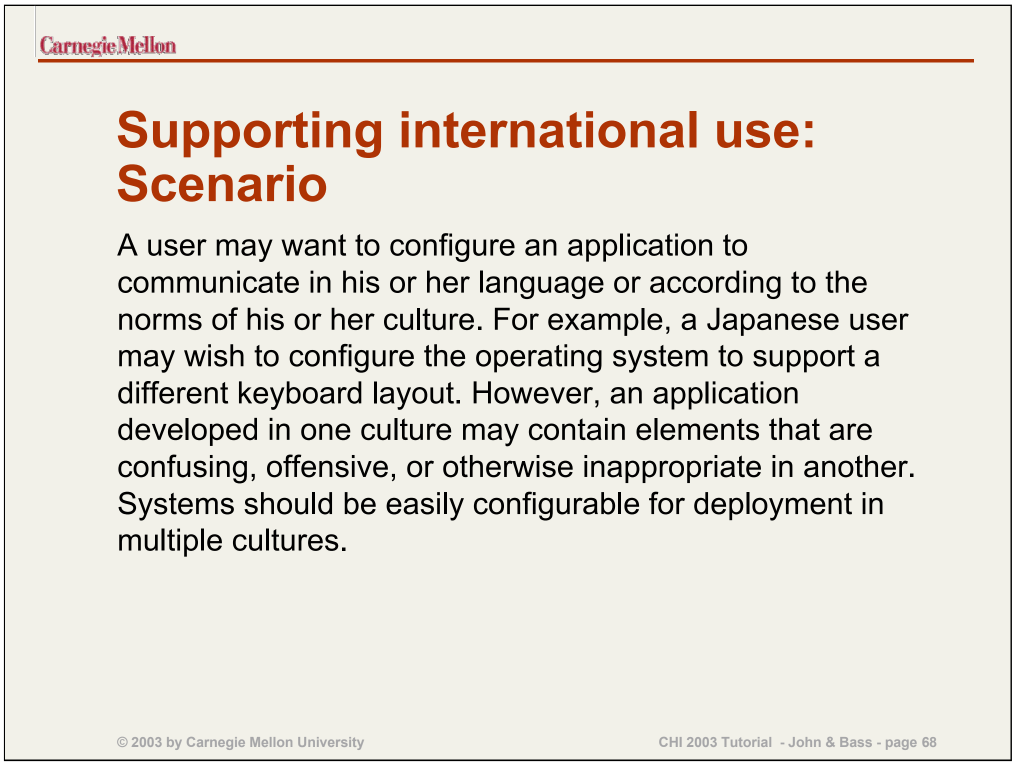
Use an intermediary
Data

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



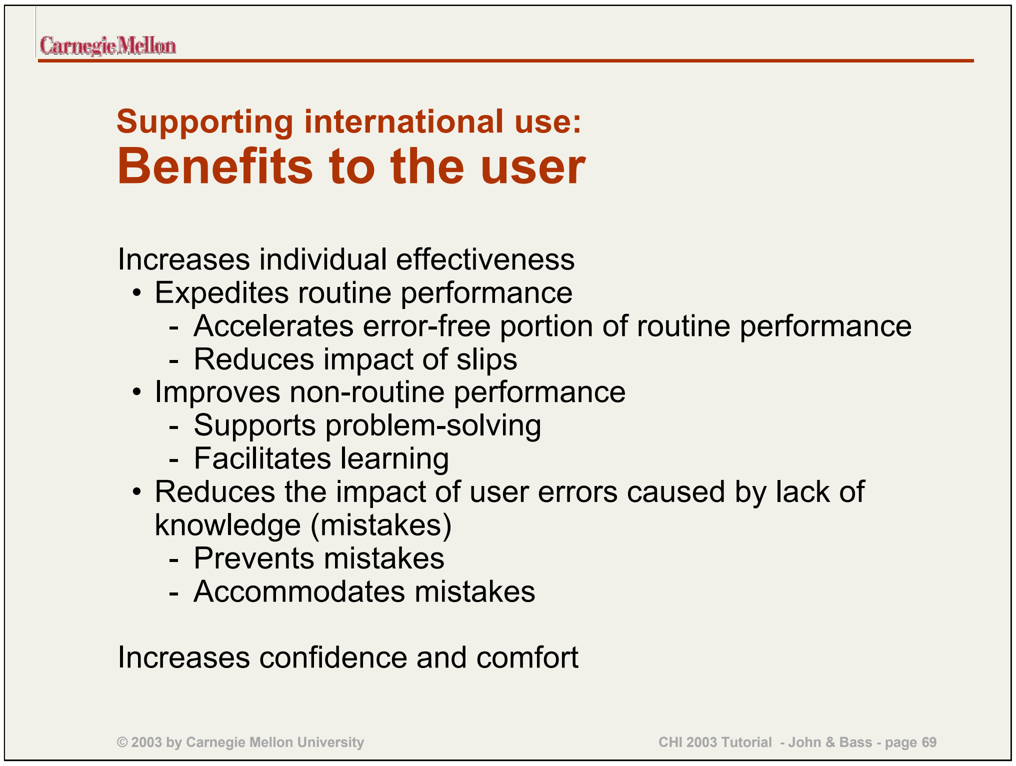
Carnegie Mellon

How does reusing information relate to your system?


Carnegie Mellon

Supporting international use: Scenario

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.



Carnegie Mellon

Supporting international use: Benefits to the user

Increases individual effectiveness
- Expedites routine performance
  - Accelerates error-free portion of routine performance
  - Reduces impact of 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

Increases confidence and comfort

© 2003 by Carnegie Mellon University   CHI 2003 Tutorial - John & Bass - page 69

Increases individual effectiveness
Expedites routine performance
Accelerates error-free portion of routine performance
Systems that support internationalization accelerate users' performance by allowing them to communicate with the system in the language that they know best.

Increases individual effectiveness
Expedites routine performance
Reduces impact of slips
Systems that support internationalization help accommodate users' slips by presenting error messages in the language that they know best.

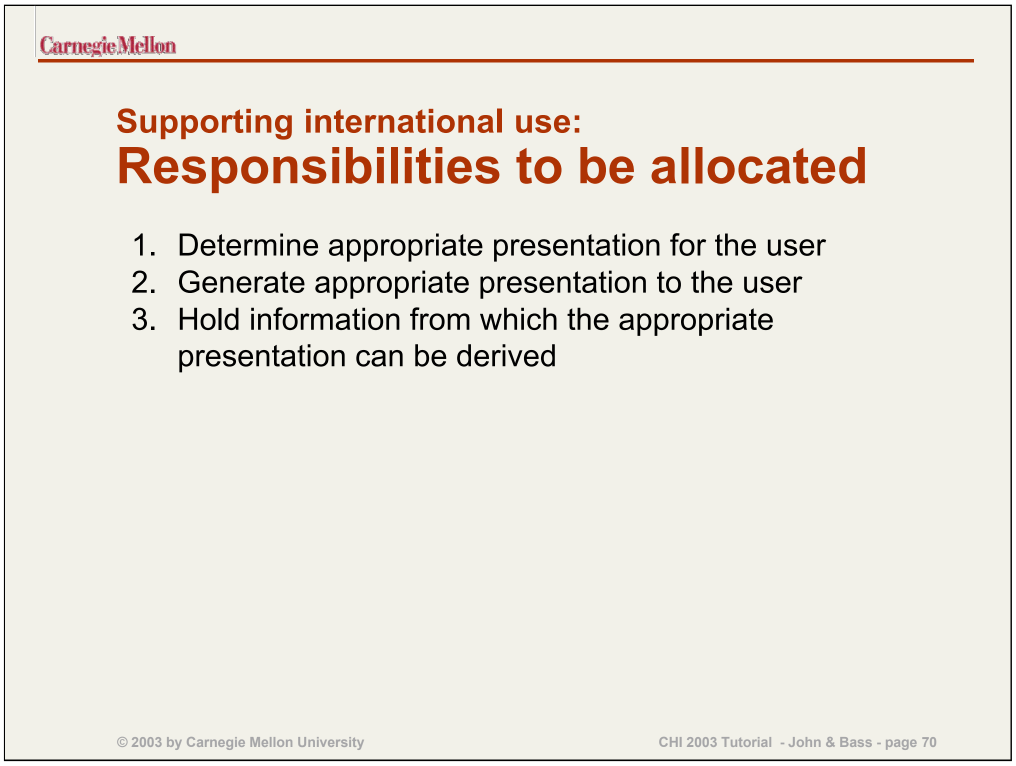
Increases individual effectiveness
Improves non-routine performance
Supports problem-solving
Systems that support internationalization facilitate problem-solving by allowing users to receive feedback from the system in the language that they know best.

Increases individual effectiveness
Improves non-routine performance
Facilitates learning
Systems that support internationalization facilitate learning by allowing users to receive feedback from the system in the language that they know best.

Increases individual effectiveness
Reduces the impact of user errors caused by lack of knowledge
Prevents mistakes
Systems that support internationalization help users avoid linguistic mistakes by allowing them to communicate with the system in the language that they know best.

Increases individual effectiveness
Reduces the impact of user errors caused by lack of knowledge
Accommodates mistakes
Systems that support internationalization help accommodate user mistakes by presenting error messages in the language that they know best. Incomprehensible error messages can compound existing misunderstanding.

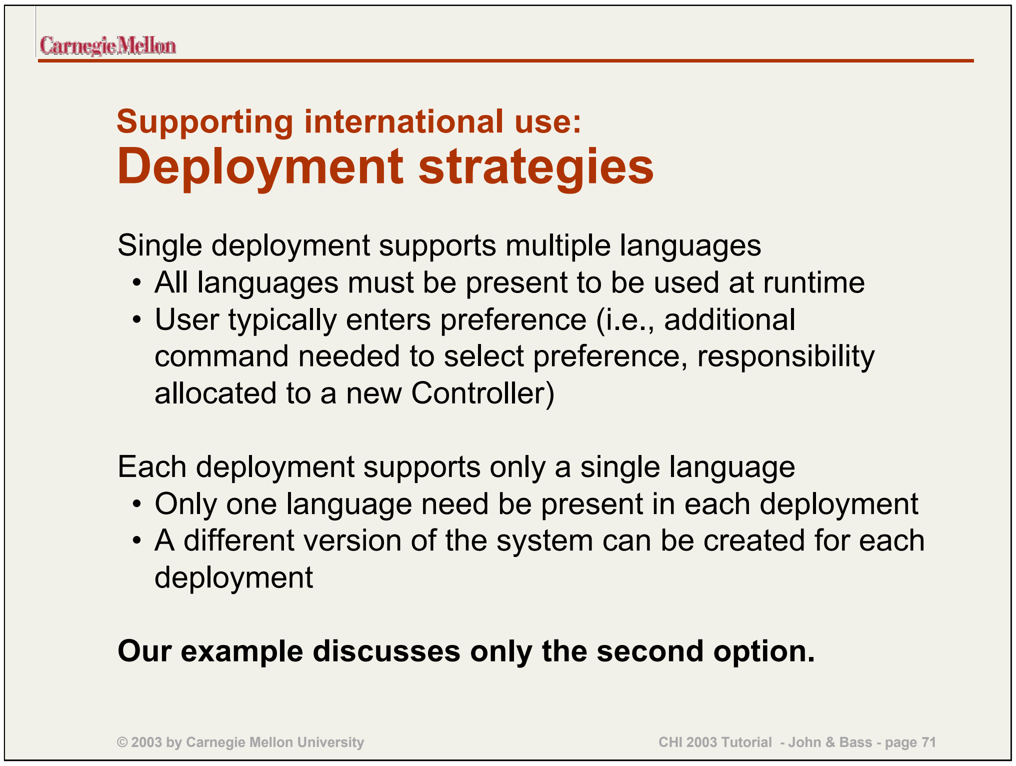
Increases confidence and comfort
Being able to communicate with a system in the language that a user knows best reduces frustration and increases user satisfaction by affirming the importance of the user's national or cultural identify.



Carnegie Mellon

Supporting international use: Responsibilities to be allocated

1. Determine appropriate presentation for the user
2. Generate appropriate presentation to the user
3. Hold information from which the appropriate presentation can be derived


Carnegie Mellon

Supporting international use: Deployment strategies

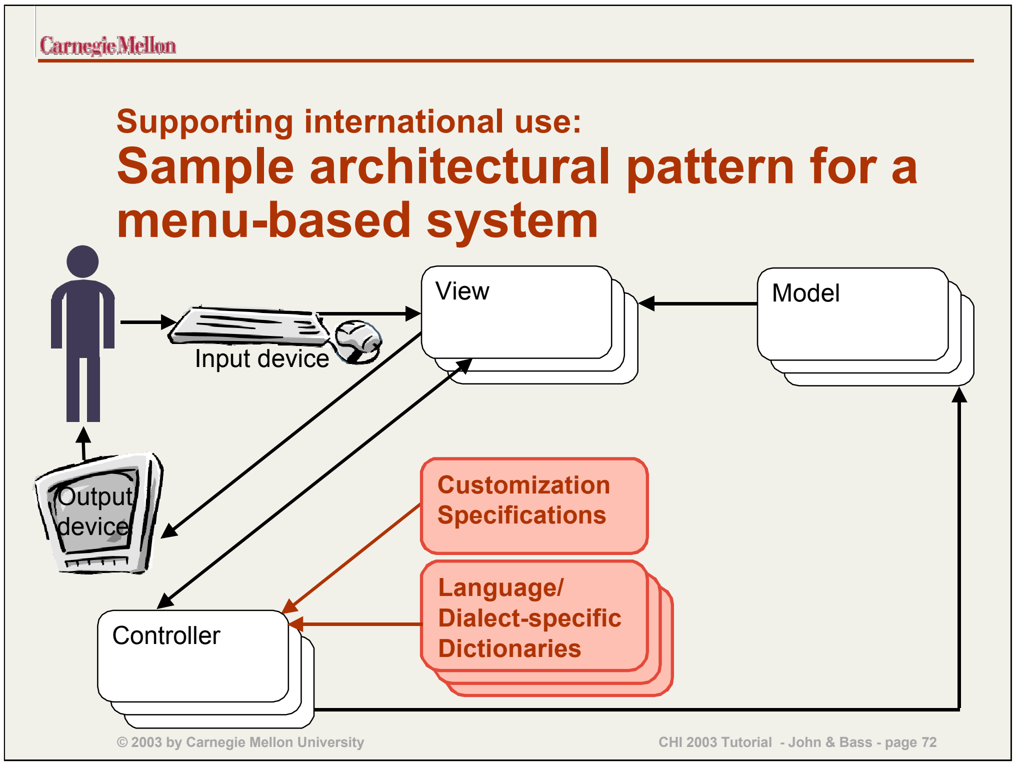
Single deployment supports multiple languages
- All languages must be present to be used at runtime
- User typically enters preference (i.e., additional command needed to select preference, responsibility allocated to a new Controller)

Each deployment supports only a single language
- Only one language need be present in each deployment
- A different version of the system can be created for each deployment

Our example discusses only the second option.

© 2003 by Carnegie Mellon University CHI 2003 Tutorial - John & Bass - page 71

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



Carnegie Mellon

Supporting international use: Sample architectural pattern for a menu-based system

View
Model
Input device
Output device
Customization Specifications
Language/ Dialect-specific Dictionaries
Controller

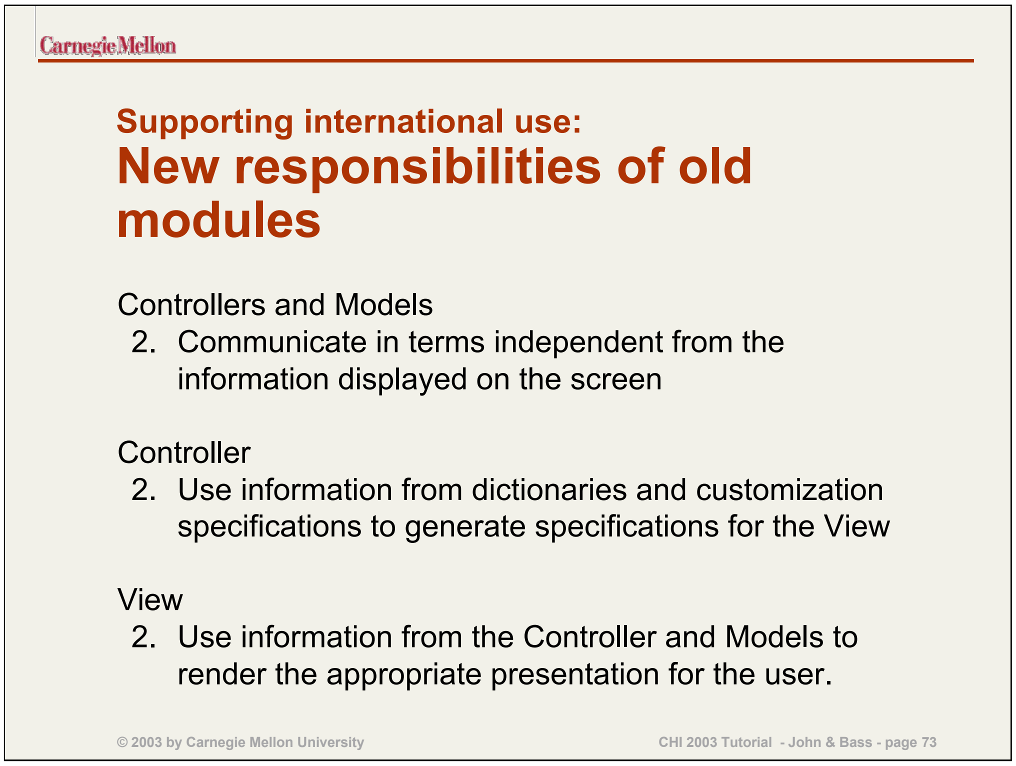
CHI 2003 Tutorial - John & Bass - page 72

This sample pattern assumes a menu-based system, not a command-based where the commands have to be recognized in different languages.

This pattern does not support end-user customization.

In this pattern, customization specifications and the dictionaries are put in by the developers. Thus, the first responsibility, determine appropriate presentation for this user, is allocated to the developer, not to any software module.

The dictionaries are often not of single words from which prose is generated, but of all the text that has to be presented. For instance, an entire error message may be an entry in the dictionary.


Carnegie Mellon

Supporting international use: New responsibilities of old modules

Controllers and Models

2. Communicate in terms independent from the information displayed on the screen

Controller

2. Use information from dictionaries and customization specifications to generate specifications for the View

View

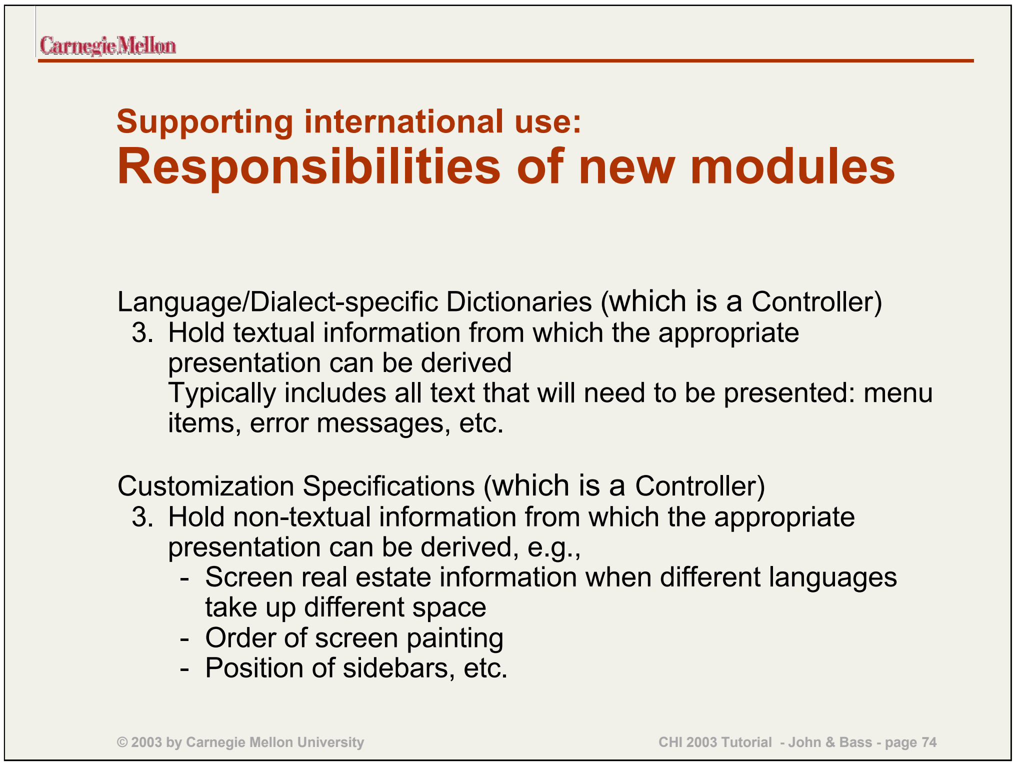
2. Use information from the Controller and Models to render the appropriate presentation for the user.

© 2003 by Carnegie Mellon University

CHI 2003 Tutorial - John & Bass - page 73

Responsibilities to be allocated:

1. Determine appropriate presentation for the user (allocated to the developer in this example pattern)
2. Generate appropriate presentation to the user
3. Hold information from which the appropriate presentation can be derived


Carnegie Mellon

Supporting international use: Responsibilities of new modules

Language/Dialect-specific Dictionaries (which is a Controller)

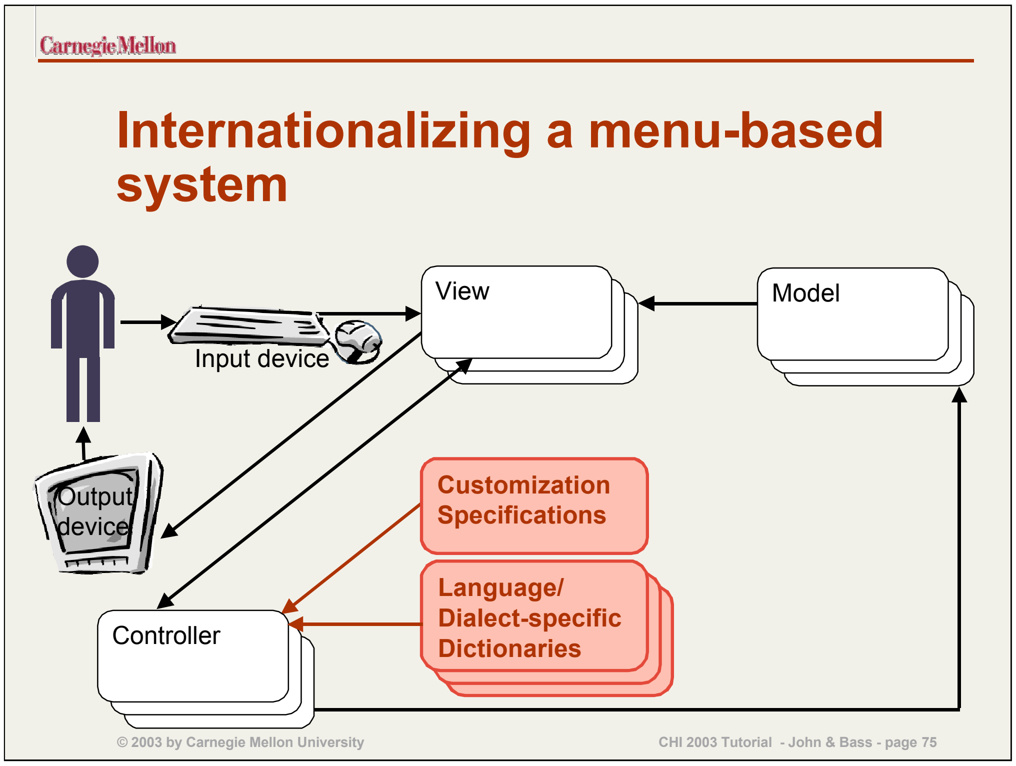
3. Hold textual information from which the appropriate presentation can be derived
   Typically includes all text that will need to be presented: menu items, error messages, etc.

Customization Specifications (which is a Controller)

3. Hold non-textual information from which the appropriate presentation can be derived, e.g.,
   - Screen real estate information when different languages take up different space
   - Order of screen painting
   - Position of sidebars, etc.

Responsibilities to be allocated:

1. Determine appropriate presentation for the user (allocated to the developer in this example pattern)
2. Generate appropriate presentation to the user
3. Hold information from which the appropriate presentation can be derived



Carnegie Mellon

Internationalizing a menu-based system

View
Model
Input device
Output device
Customization Specifications
Language/ Dialect-specific Dictionaries
Controller

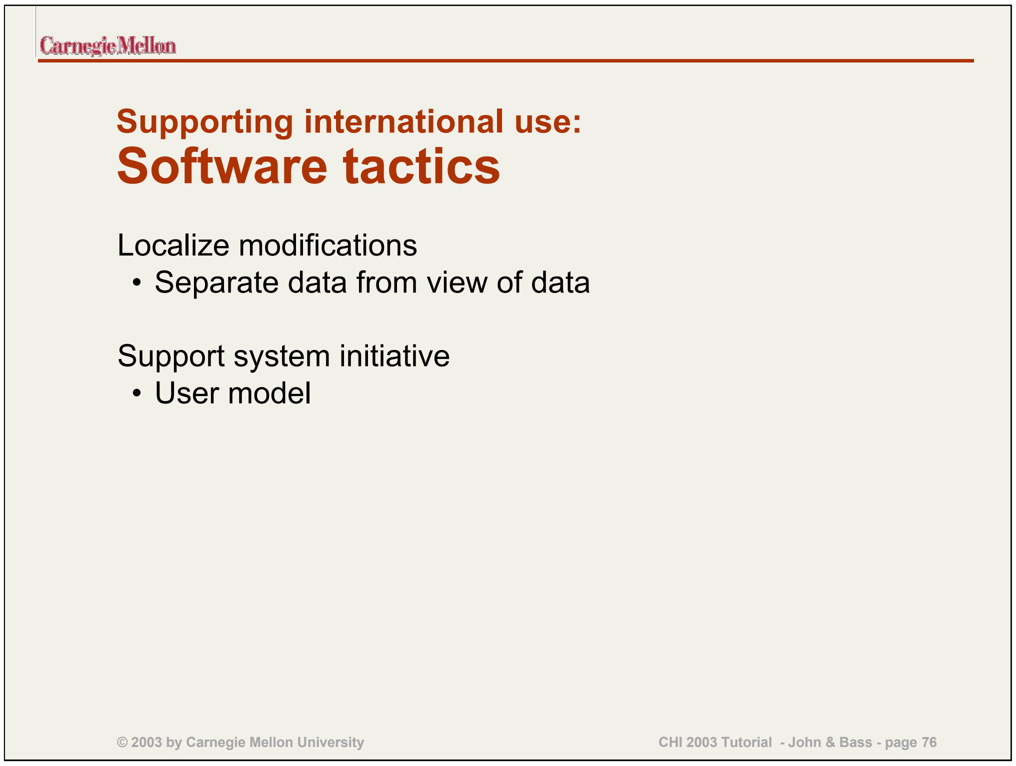
Step 1. The user requests a screen that contains internationalized information from the View.

Step 2. The View passes the request to the Controller to determine the appropriate presentation.

Step 3. The Controller consults the Customization Specifications and Language / Dialect-specific Dictionaries to obtain the necessary information to produce the internationalized screen.

Step 4. The Controller uses this information to generate an internationalized screen specification and passes this screen specification to the View.

Step 5. The View renders the internationalized screen and sends the output to the user.



Carnegie Mellon

Supporting international use: Software tactics

Localize modifications
- Separate data from view of data

Support system initiative
- User model

Localize modifications

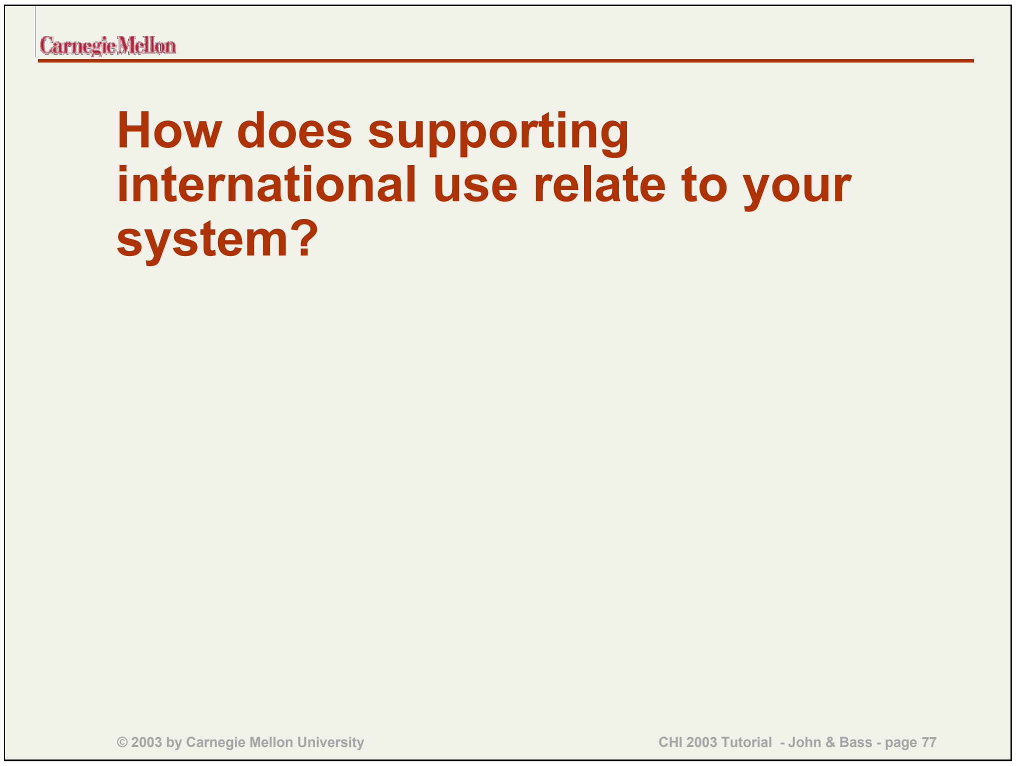
Separate data from view of data

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

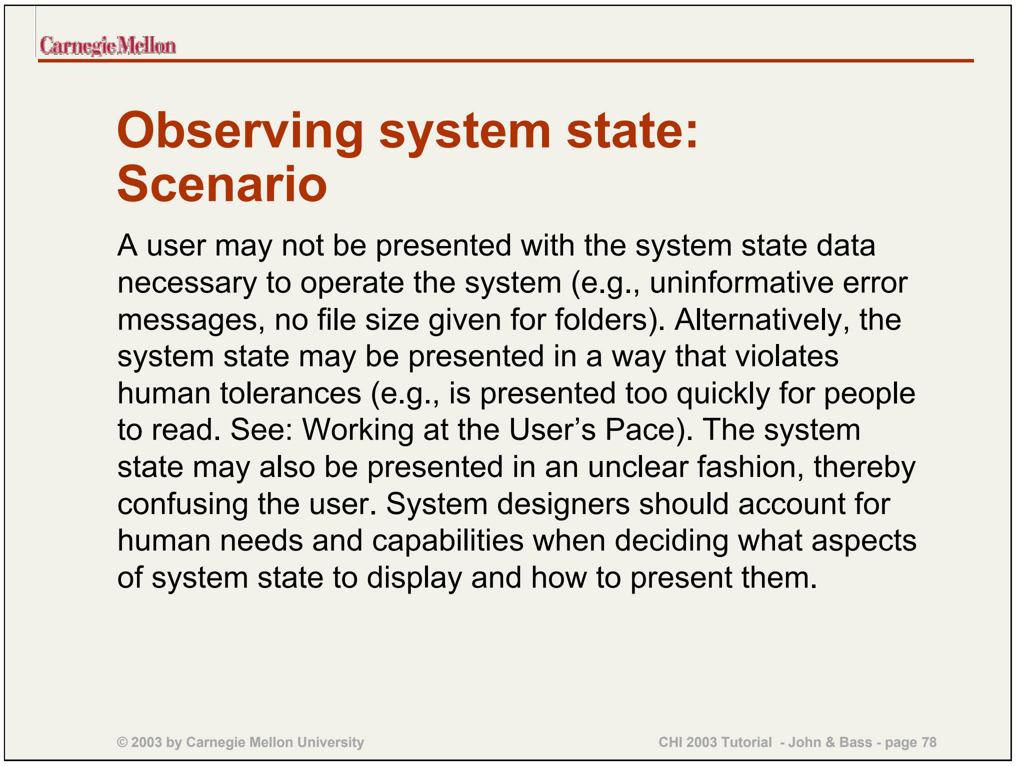
Support system initiative

User model

The View module accesses user-supplied customization specifications to configure itself appropriately. These specifications model the user.


Carnegie Mellon

How does supporting international use relate to your system?

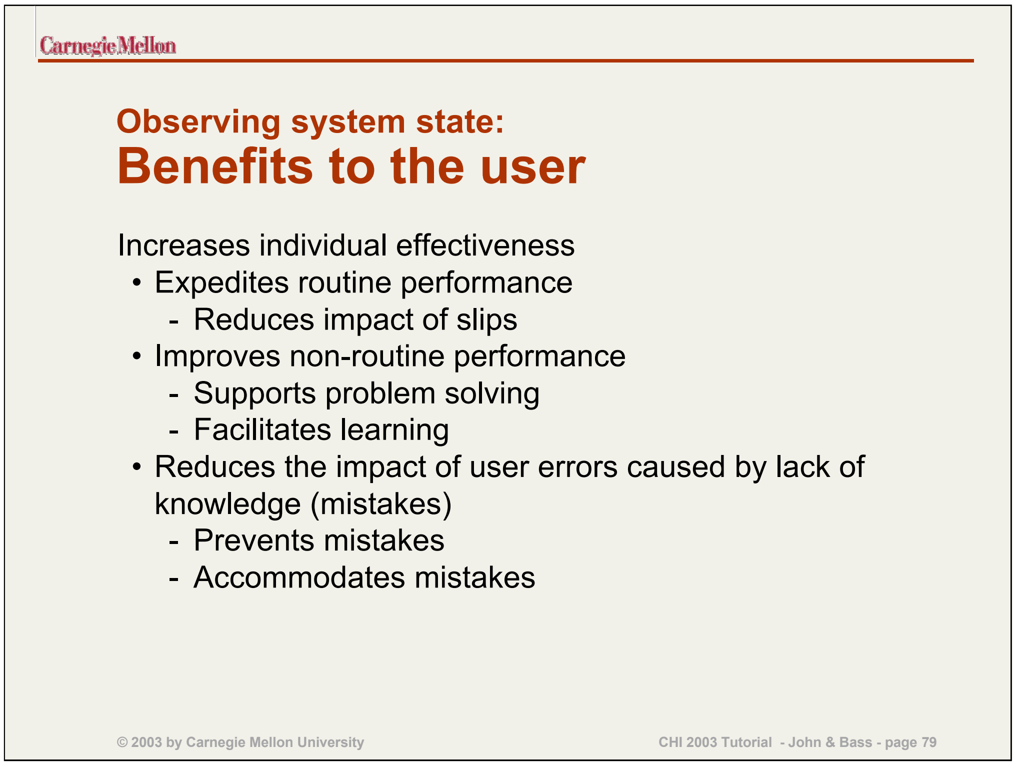


Carnegie Mellon

Observing system state: Scenario

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.


Carnegie Mellon

Observing system state: Benefits to the user

Increases individual effectiveness
- Expedites routine performance
  - Reduces impact of 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

CHI 2003 Tutorial - John & Bass - page 79

Increases individual effectiveness
Expedites routine performance
Reduces impact of slips
When the inevitable slip happens, if the system state is readily and easily observed, the user will know how to correct the slip before continuing further down an incorrect path.

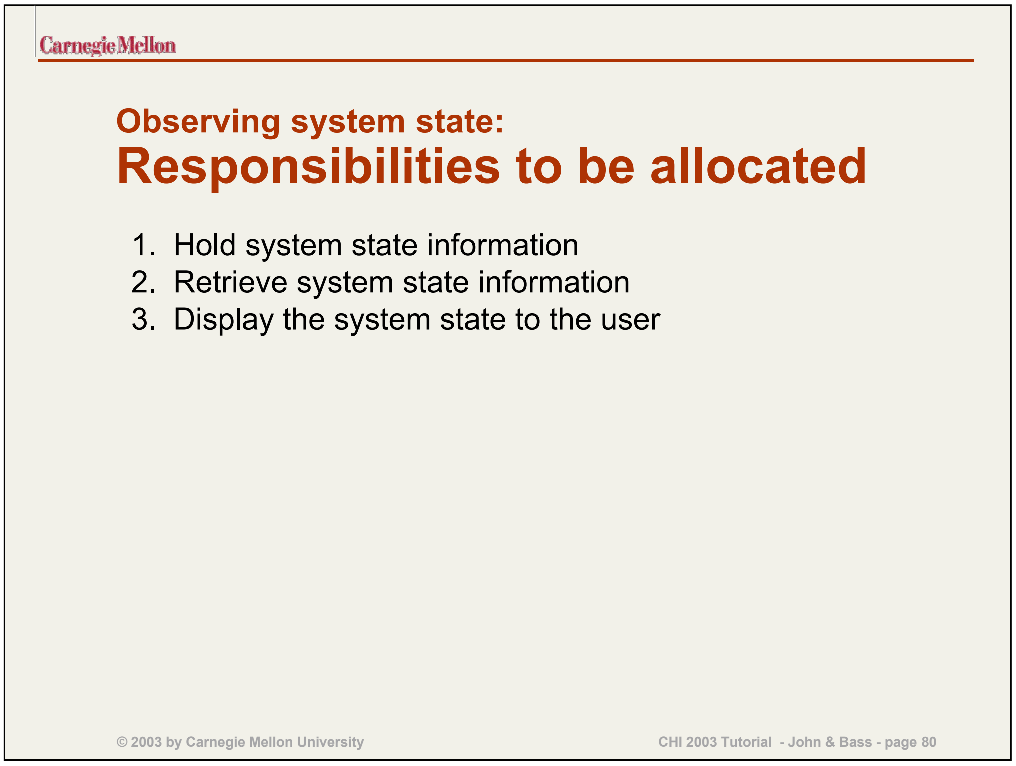
Increases individual effectiveness
Improves non-routine performance
Supports problem solving
Human problem-solving depends on knowledge of current state (where you are), the goal state (where you want to be), and awareness of the range of available actions. Thus, being able to observe the current system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

Increases individual effectiveness
Improves non-routine performance
Facilitates learning
Learning correct actions depends on knowing the system state when the action produced a desired response. Thus, if the system state is obscured or unobservable, the user's ability to learn will be inhibited.

Increases individual effectiveness
Reduces the impact of user errors caused by lack of knowledge
Prevents mistakes
A common type of mistake occurs when a user applies knowledge and procedures appropriate to one system state to a different, inappropriate, system state. Making the system state easily available to users reduces the likelihood of this type of mistake.

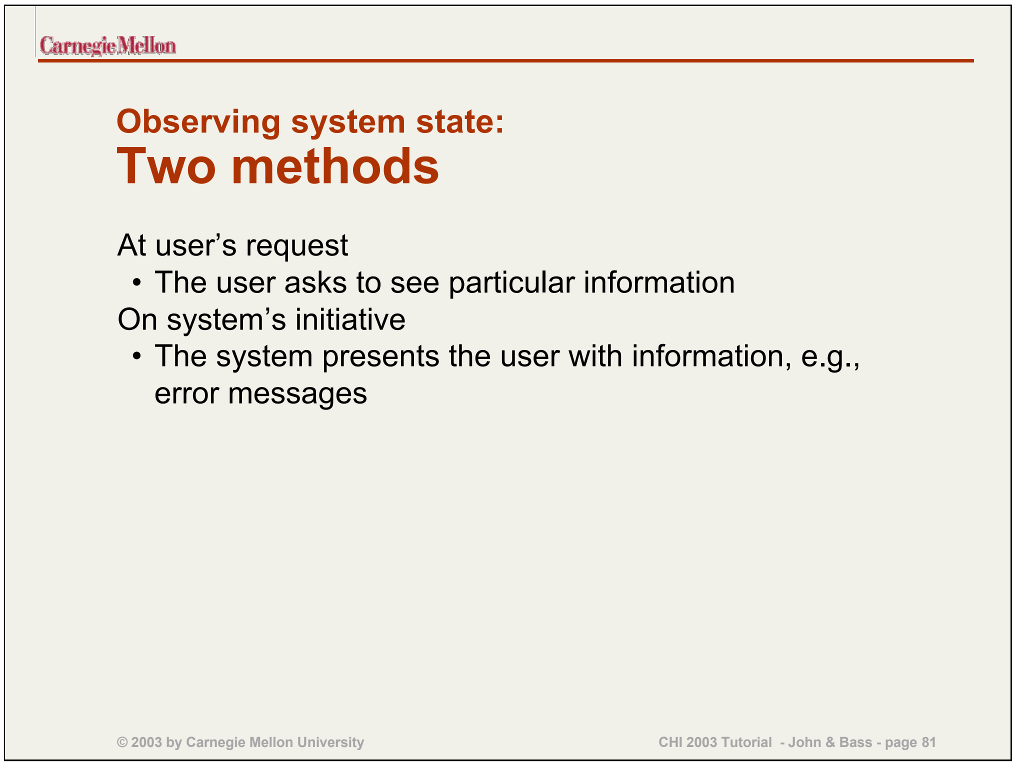
Increases individual effectiveness
Reduces the impact of user errors caused by lack of knowledge
Accommodates mistakes
If the system state is readily and easily observed, the user will know how to correct the mistake before continuing further down an incorrect path.

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


Carnegie Mellon

Observing system state: Responsibilities to be allocated

1. Hold system state information
2. Retrieve system state information
3. Display the system state to the user


Carnegie Mellon

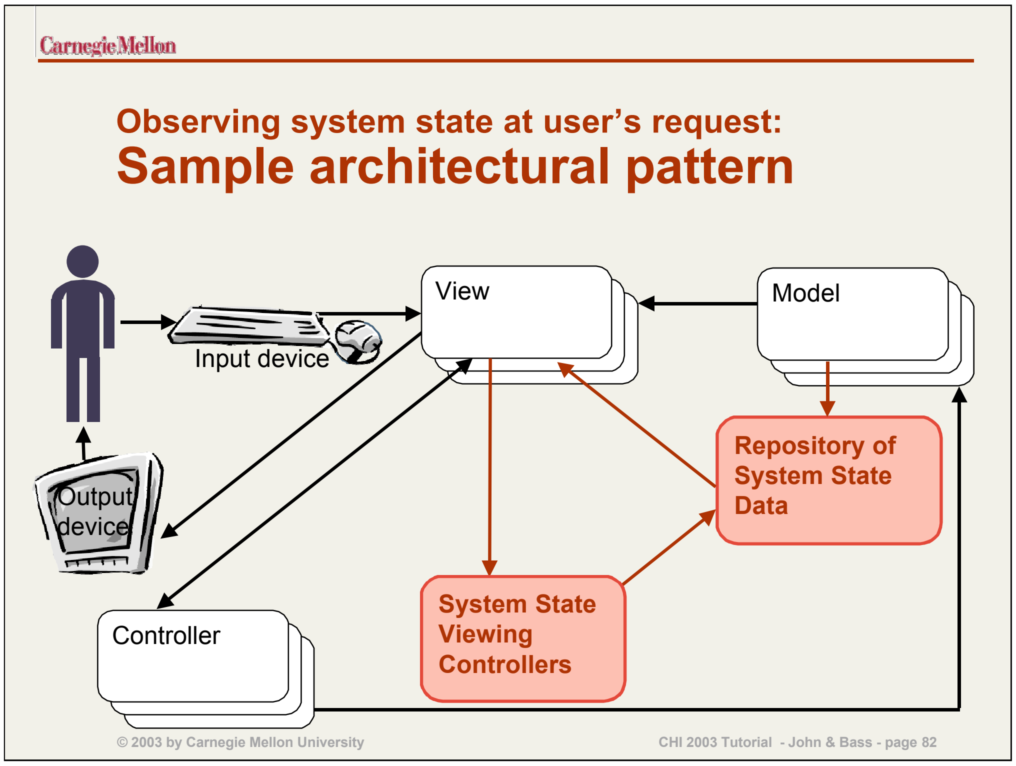
Observing system state:
Two methods

At user's request

- The user asks to see particular information

On system's initiative

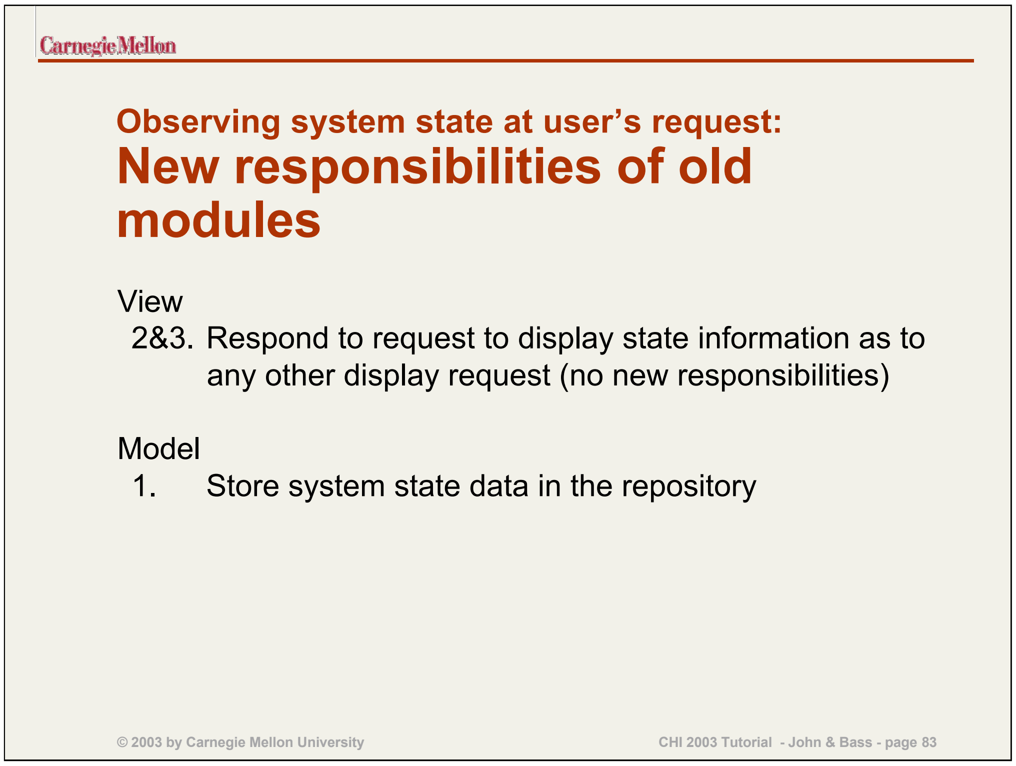
- The system presents the user with information, e.g., error messages



Carnegie Mellon

Observing system state at user's request: Sample architectural pattern

Input device
View
Model
Output device
Repository of System State Data
System State Viewing Controllers
Controller


Carnegie Mellon

Observing system state at user's request: New responsibilities of old modules

View

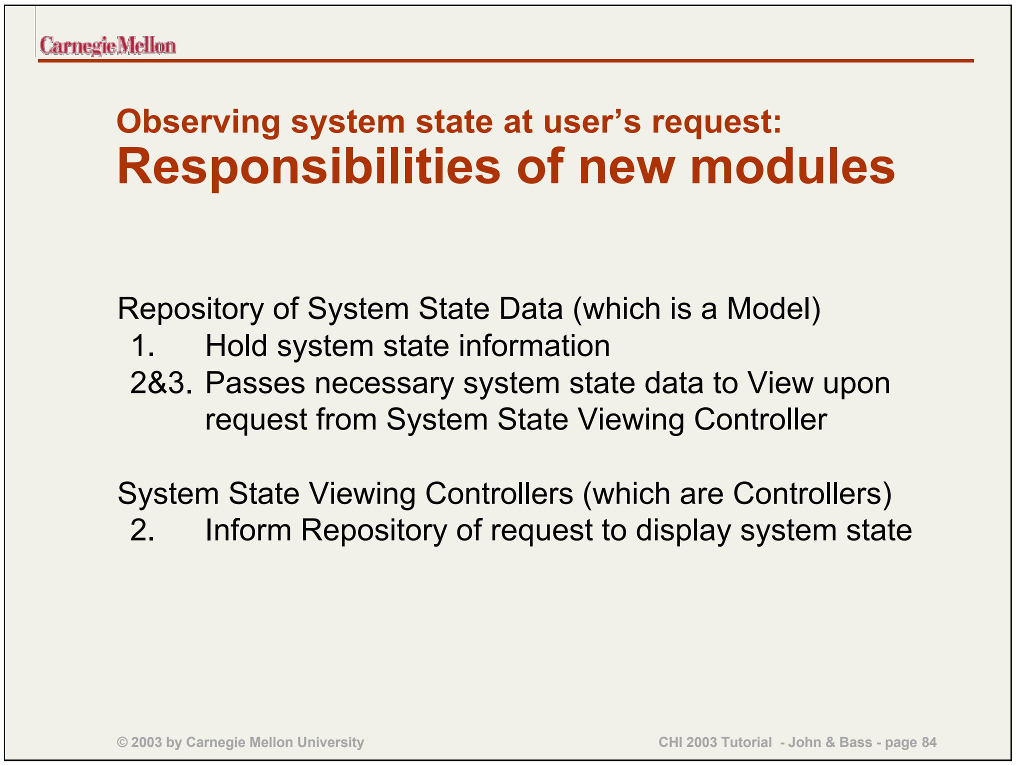
2&3. Respond to request to display state information as to any other display request (no new responsibilities)

Model

1. Store system state data in the repository

Responsibilities to be allocated:

1. Hold system state information
2. Retrieve system state information
3. Display the system state to the user


Carnegie Mellon

Observing system state at user's request: Responsibilities of new modules

Repository of System State Data (which is a Model)

1. Hold system state information

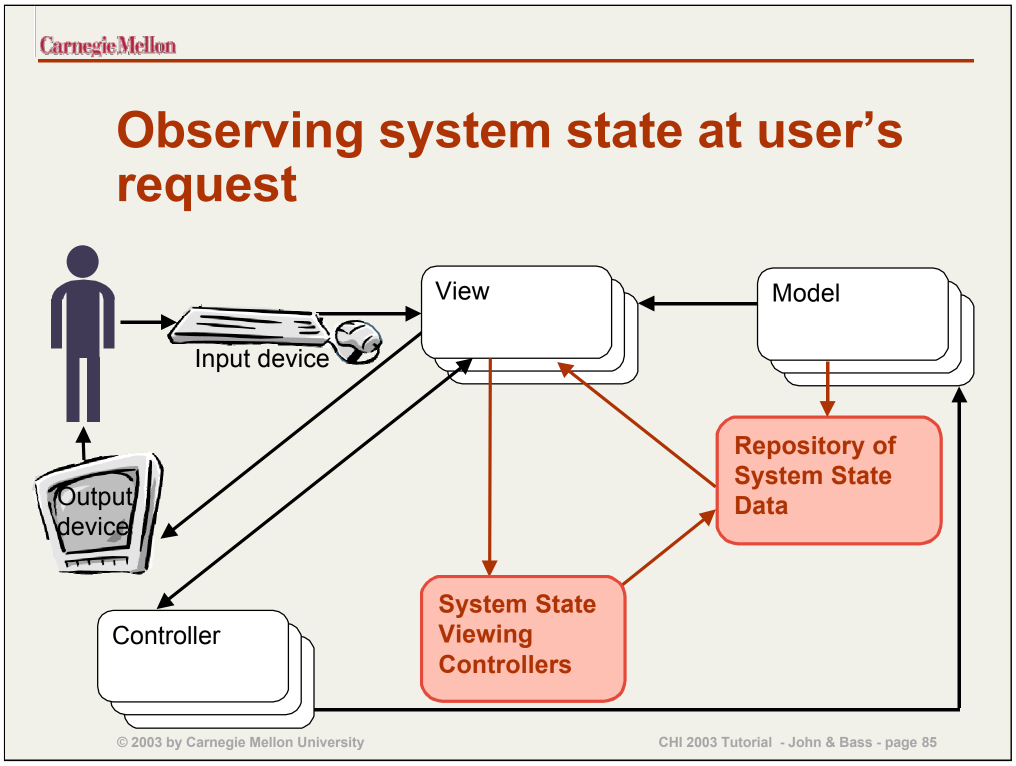
2&3. Passes necessary system state data to View upon request from System State Viewing Controller

System State Viewing Controllers (which are Controllers)

2. Inform Repository of request to display system state

Responsibilities to be allocated:

1. Hold system state information
2. Retrieve system state information
3. Display the system state to the user



Carnegie Mellon

Observing system state at user's request

Input device
Output device
View
Model
Repository of System State Data
System State Viewing Controllers
Controller

© 2003 by Carnegie Mellon University

CHI 2003 Tutorial - John & Bass - page 85

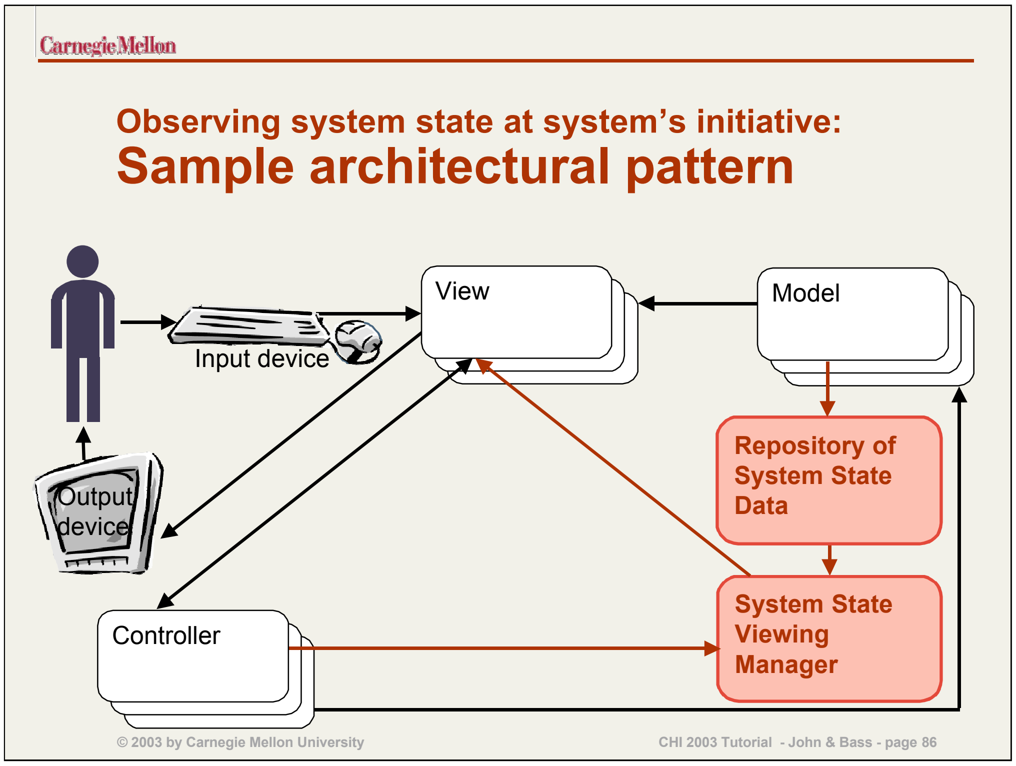
Step 1. The user issues a request to observe a part of the system's current state to the View.

Step 2. The View passes the request to a System State Viewing Controller for processing.

Step 3. The System State Viewing Controller informs the Repository of System State Data of a request. (The Repository was updated by the relevant Model when its system state changed.)

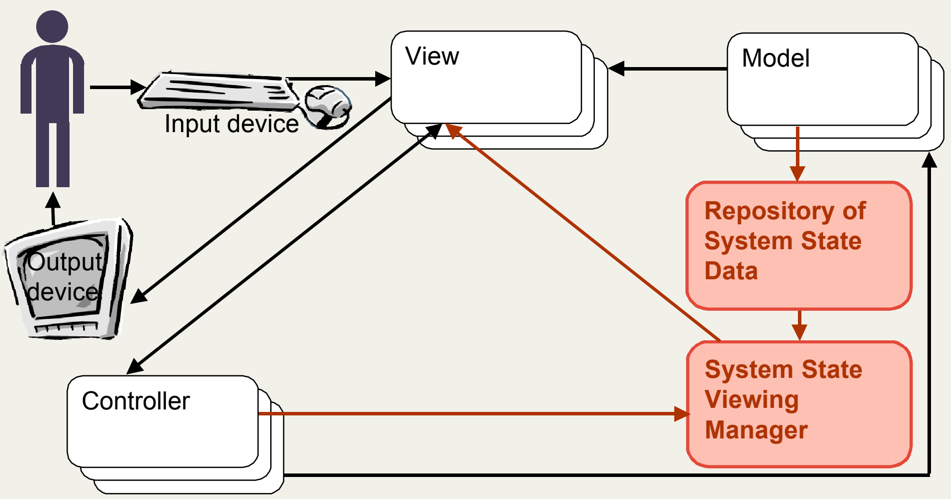
Step 4. The Repository of System State Data sends the requested system state data to the View for presenting to the user.

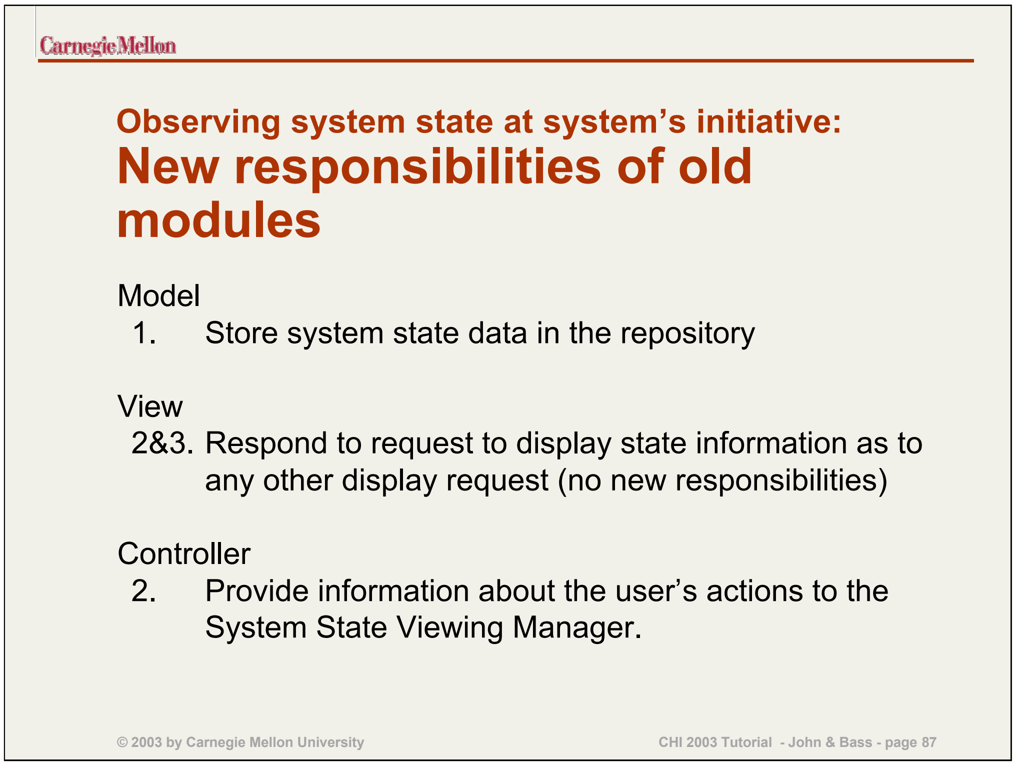
Step 5. The View displays the requested system state data to the user.



Carnegie Mellon

Observing system state at system's initiative: Sample architectural pattern




Carnegie Mellon

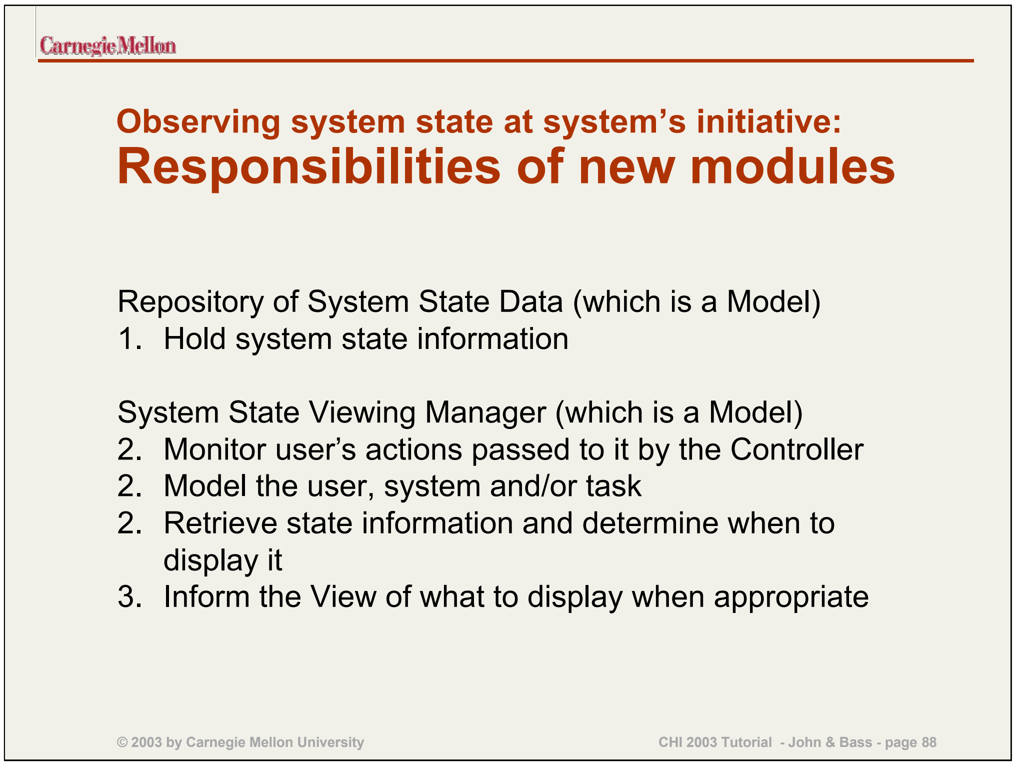
Observing system state at system's initiative: New responsibilities of old modules

Model
1. Store system state data in the repository

View
2&3. Respond to request to display state information as to any other display request (no new responsibilities)

Controller
2. Provide information about the user's actions to the System State Viewing Manager.

Responsibilities to be allocated:
1. Hold system state information
2. Retrieve system state information
3. Display the system state to the user

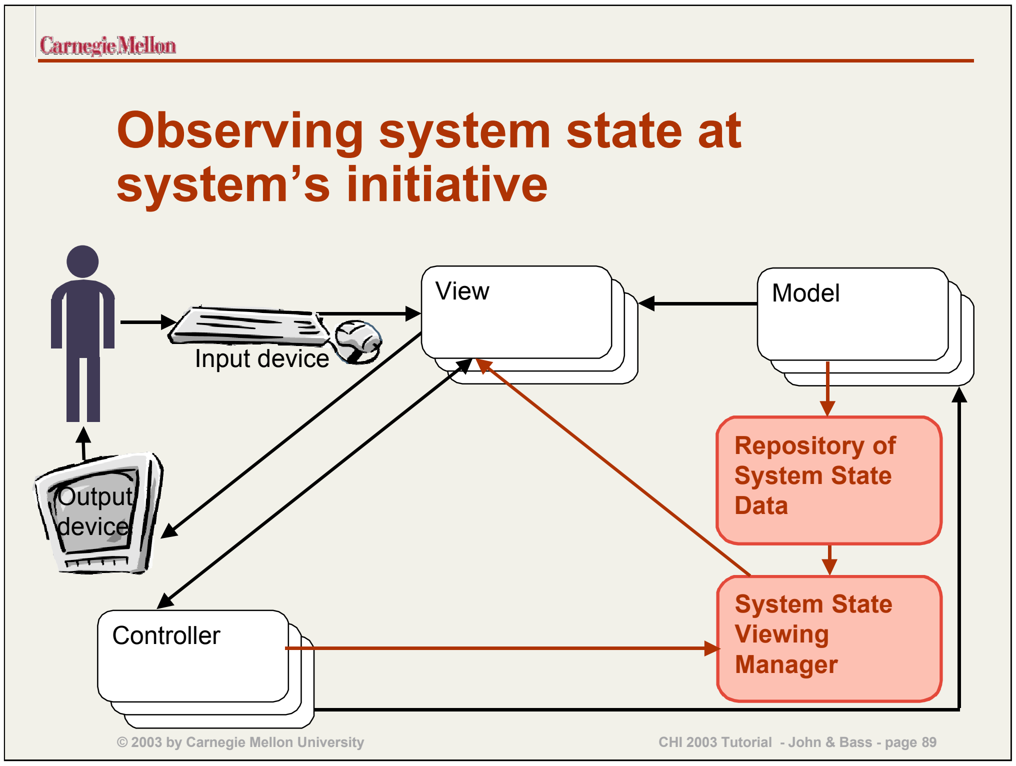

Carnegie Mellon

Observing system state at system's initiative: Responsibilities of new modules

Repository of System State Data (which is a Model)
1. Hold system state information

System State Viewing Manager (which is a Model)
2. Monitor user's actions passed to it by the Controller
2. Model the user, system and/or task
2. Retrieve state information and determine when to display it
3. Inform the View of what to display when appropriate

Responsibilities to be allocated:
1. Hold system state information
2. Retrieve system state information
3. Display the system state to the user


Carnegie Mellon

Observing system state at system's initiative

© 2003 by Carnegie Mellon University

CHI 2003 Tutorial - John & Bass - page 89

Step 1. The View receives an action from the user that might effect the display of system state data.

Step 2. The View sends this action to the appropriate Controller.

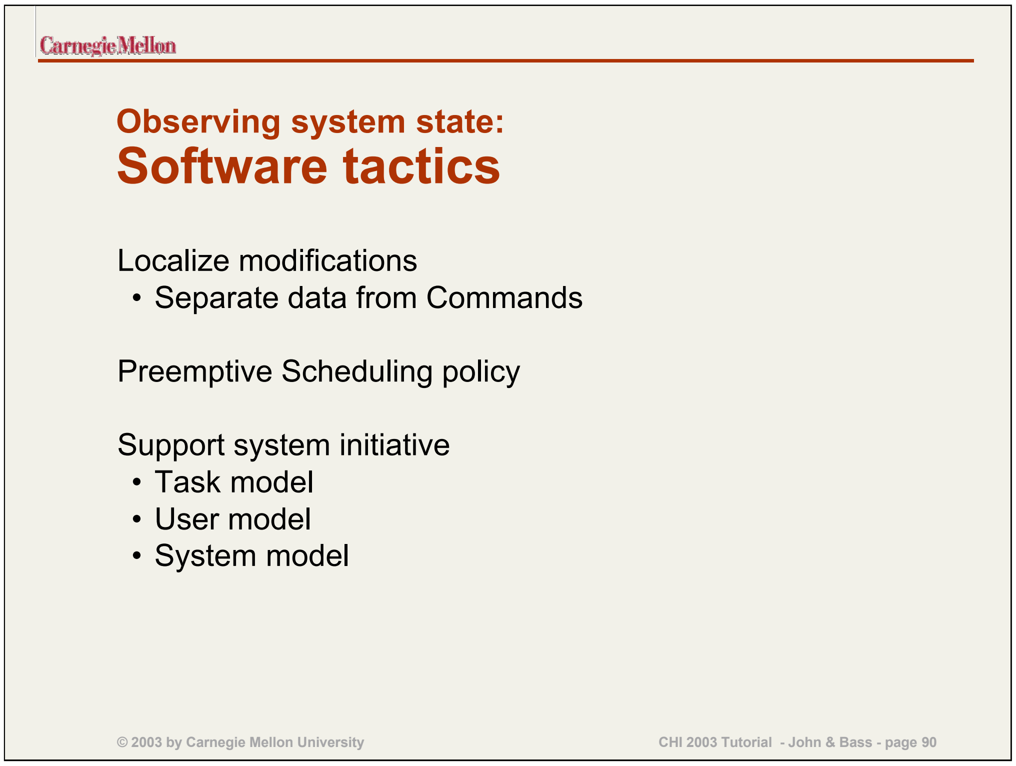
Step 3. The Controller notifies the System State Viewing Manager that the user performed an action that might effect the display of system state.

Step 4. The System State Viewing Manager, which runs on a separate thread from the main control thread, determines whether it is appropriate to display system state data.

Step 5. If system state data should be displayed, the System State Viewing Manager obtains it from the Repository of System State Data.

Step 6. The System State Viewing Manager updates the View to present the system state data to the user.

Step 7. The View displays the system state data to the user.



Carnegie Mellon

Observing system state: Software tactics

Localize modifications
- Separate data from Commands

Preemptive Scheduling policy

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

Localize modifications

Separate data from Commands

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

Preemptive Scheduling policy

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

Support system initiative

Task model

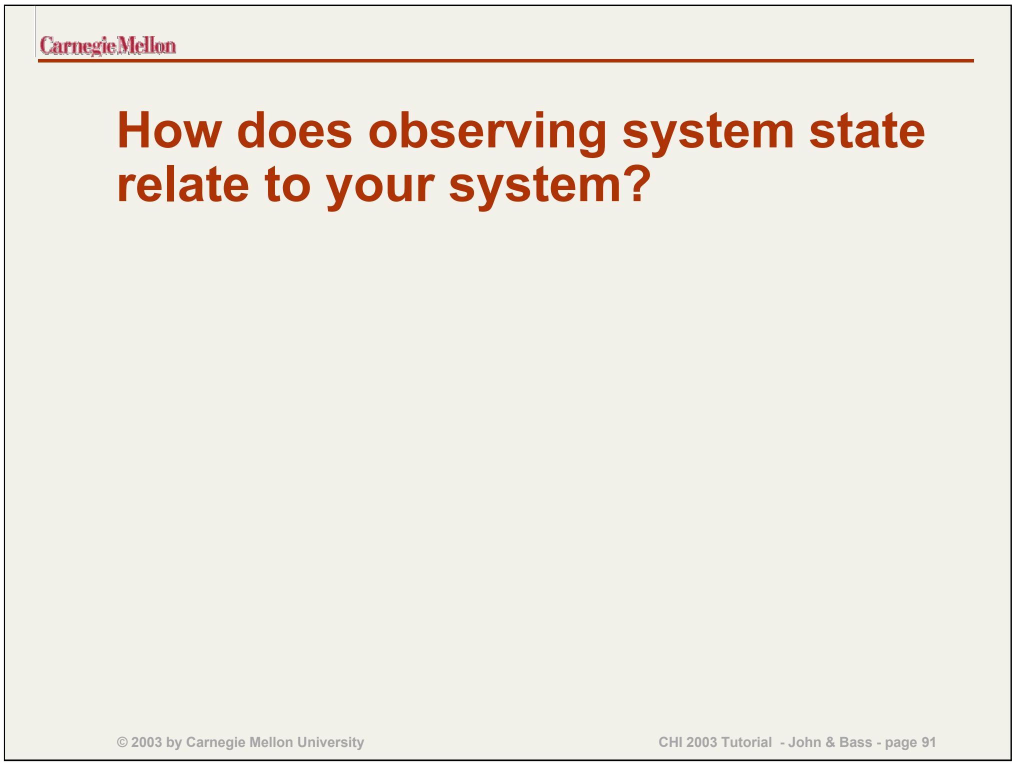
If data is to be presented on the system's initiative, the system can consult a model of the task to determine what information to present to the user.

User model

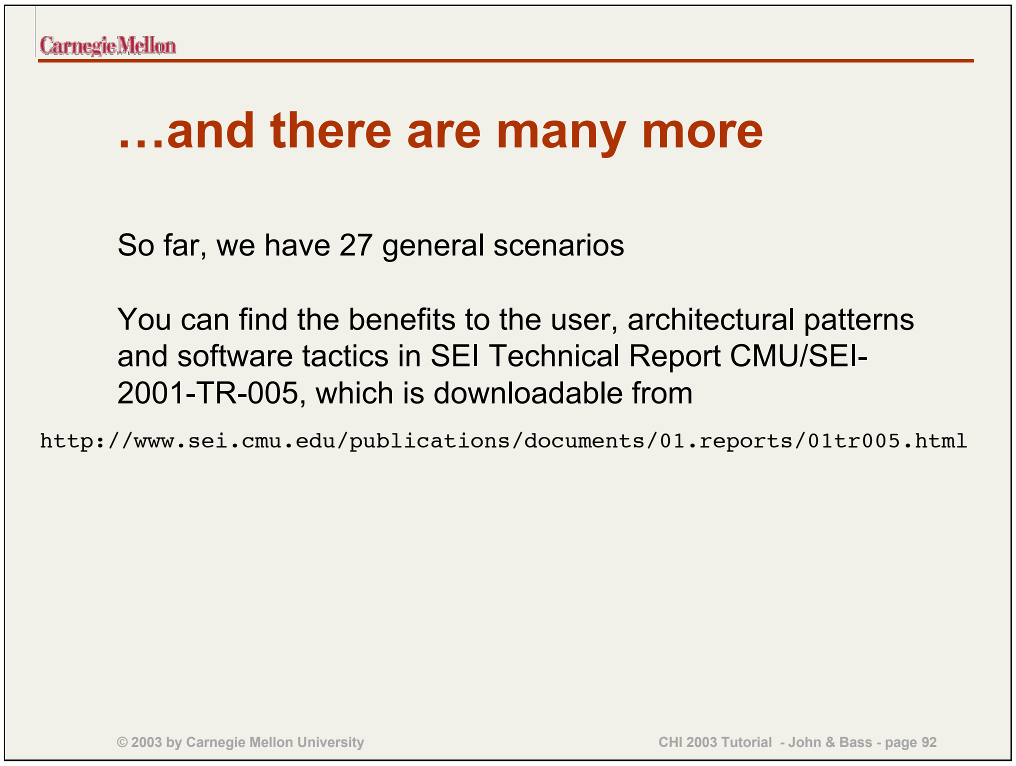
If data is to be presented on the system's initiative, the system can consult a model of the user to determine what information to present.

System model

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


Carnegie Mellon

How does observing system state relate to your system?


Carnegie Mellon

…and there are many more

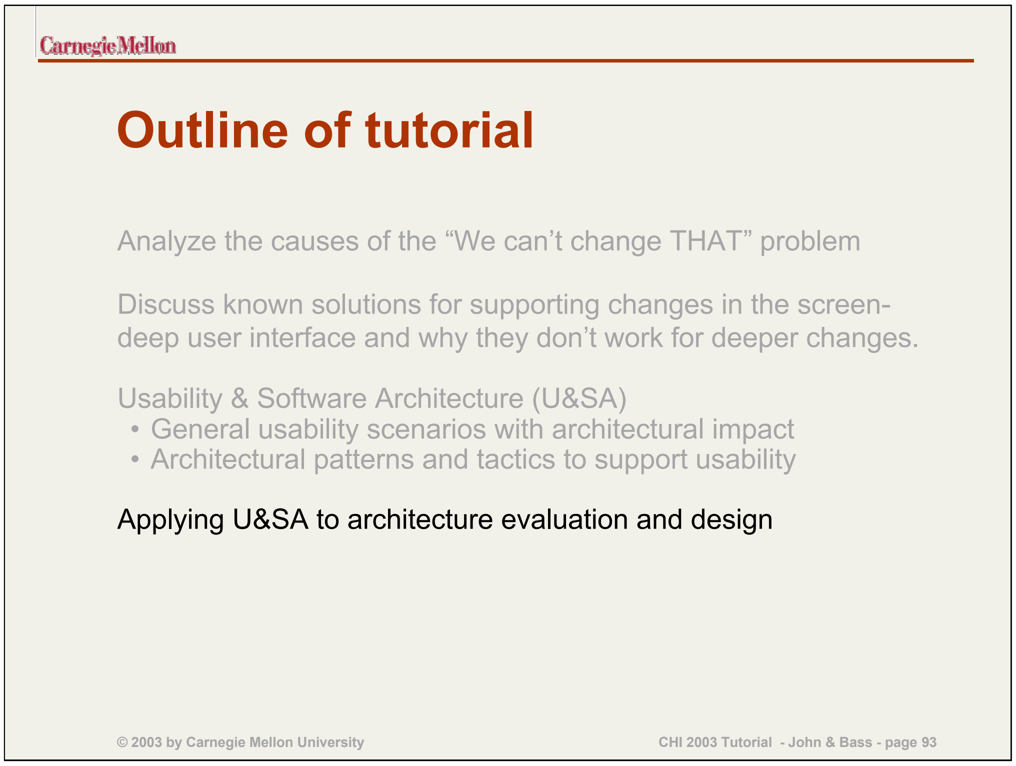
So far, we have 27 general scenarios

You can find the benefits to the user, architectural patterns and software tactics in SEI Technical Report CMU/SEI-2001-TR-005, which is downloadable from

http://www.sei.cmu.edu/publications/documents/01.reports/01tr005.html

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.

Note: in previous publications, including this technical report, we used the term "software engineering mechanisms" where we now use "software architecture tactics".



Outline of tutorial

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

Discuss known solutions for supporting changes in the screen-deep user interface and why they don’t work for deeper changes.

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

Applying U&SA to architecture evaluation and design

Applying the scenarios

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

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

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

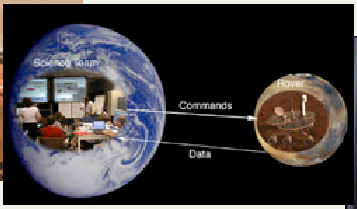
Scenarios should be re-checked during any modification effort

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

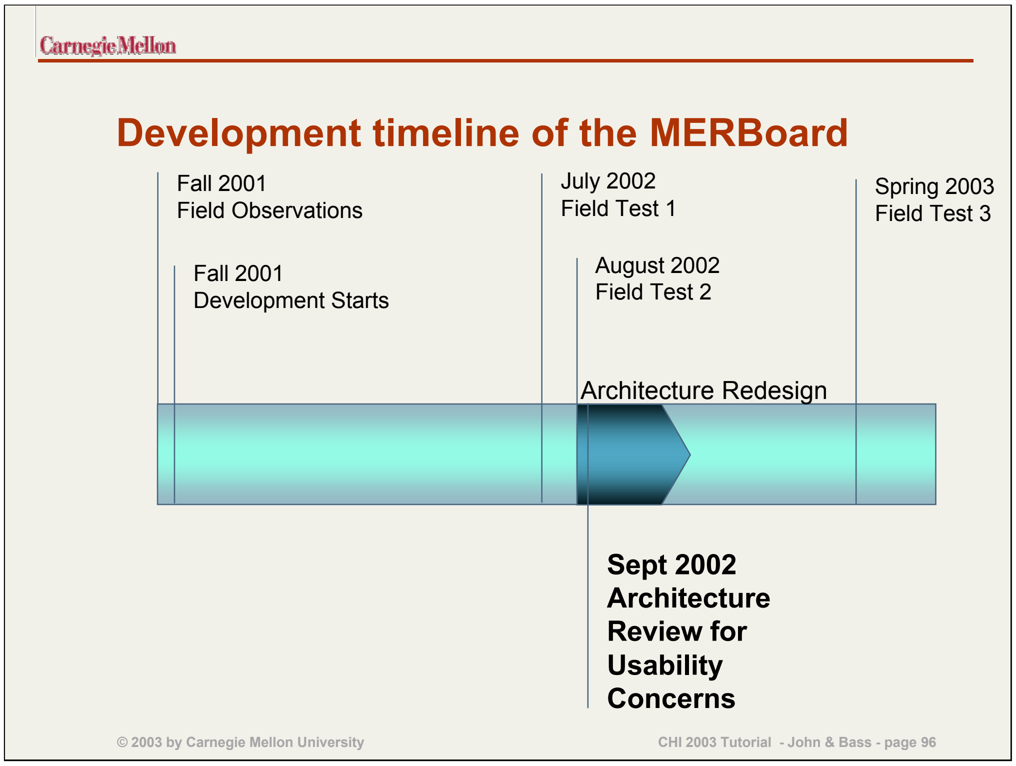
U&SA applied to the NASA MERBoard

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

Rover acts as a remote field geologist on Mars
Remote Sensing Instruments



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



Carnegie Mellon

Development timeline of the MERBoard

Developers' stated goals of the architecture redesign

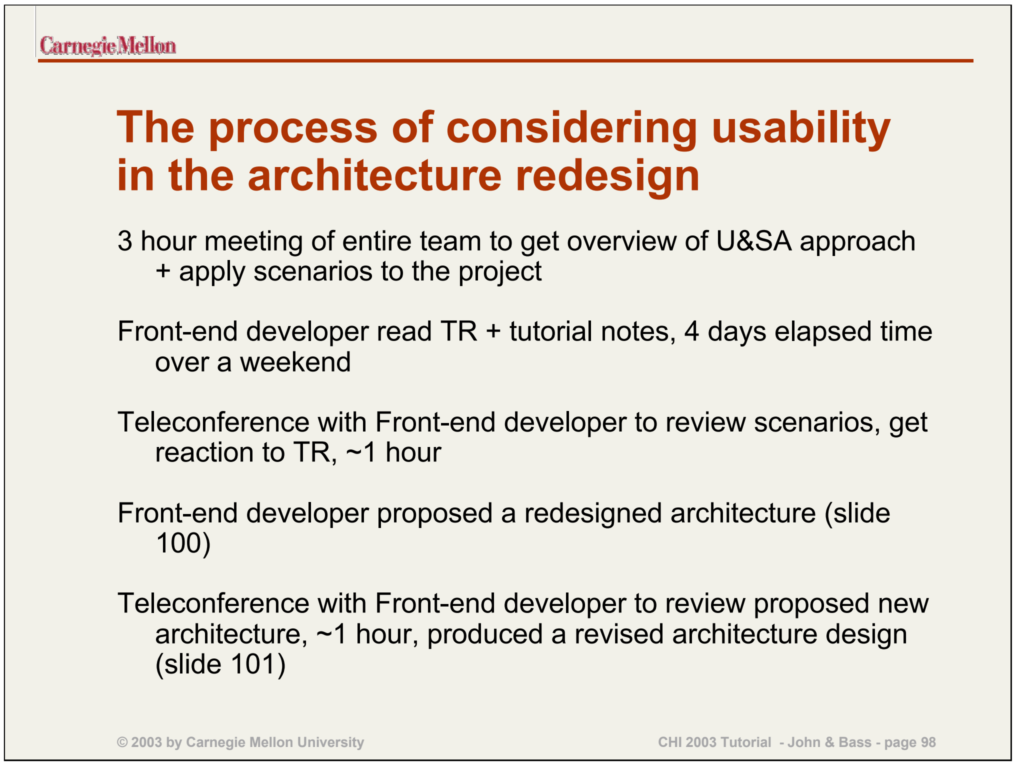
Initial design's priority was to deliver a working system to the two field trials, produced a "monolithic" system

Initial design worked, and received excellent response at the two field trials

For future trials and deployment, ***extendibility***, ***performance***, and ***reliability*** were driving changes to the architecture

Usability had always been stated as a goal for the MERBoard project as a whole, but not at the architecture level

Through NASA's High Dependability Computing Program, John & Bass proposed ***usability*** as a quality aspect to be considered during architecture redesign



The process of considering usability in the architecture redesign

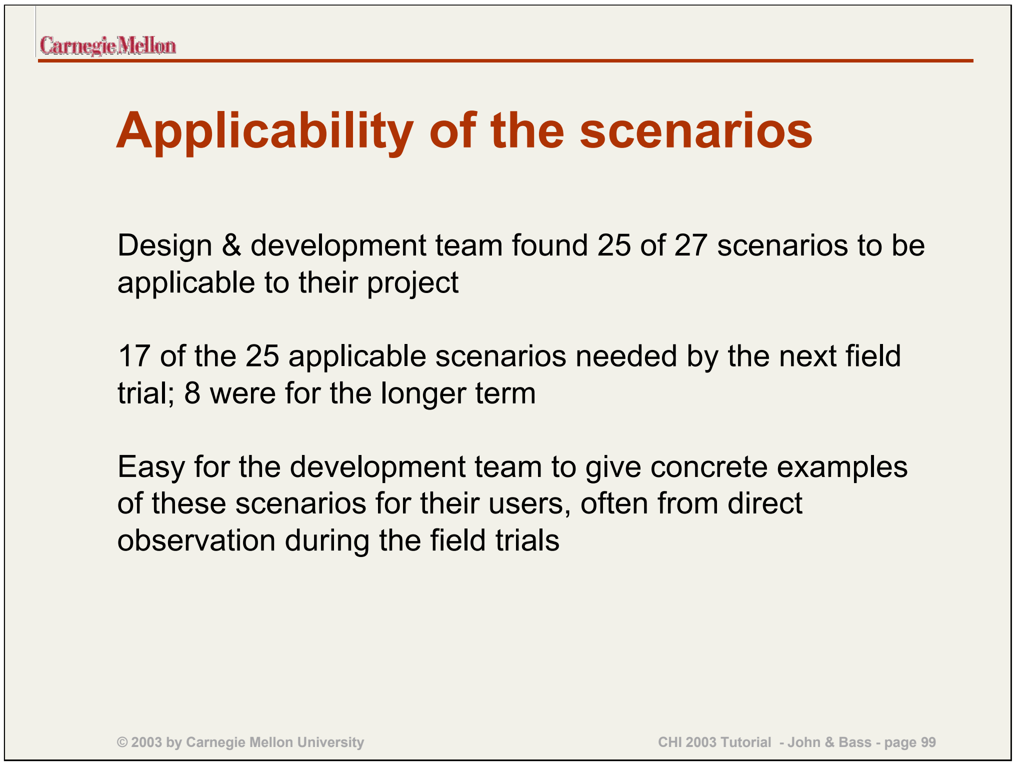
3 hour meeting of entire team to get overview of U&SA approach + apply scenarios to the project

Front-end developer read TR + tutorial notes, 4 days elapsed time over a weekend

Teleconference with Front-end developer to review scenarios, get reaction to TR, ~1 hour

Front-end developer proposed a redesigned architecture (slide 100)

Teleconference with Front-end developer to review proposed new architecture, ~1 hour, produced a revised architecture design (slide 101)

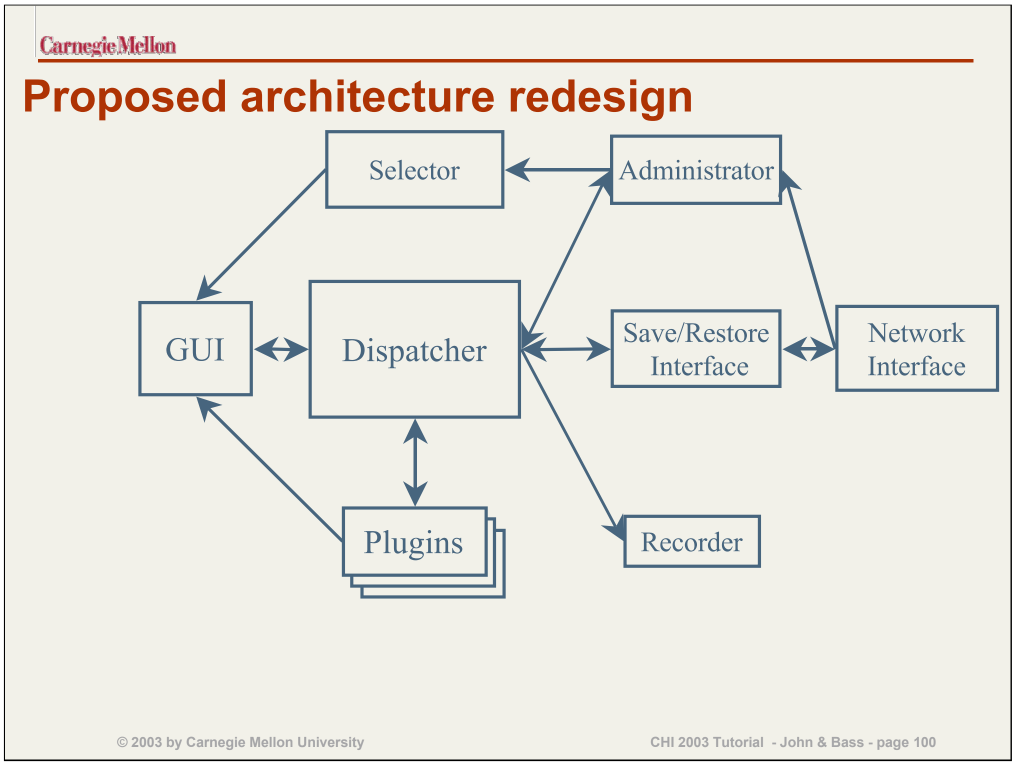

Carnegie Mellon

Applicability of the scenarios

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

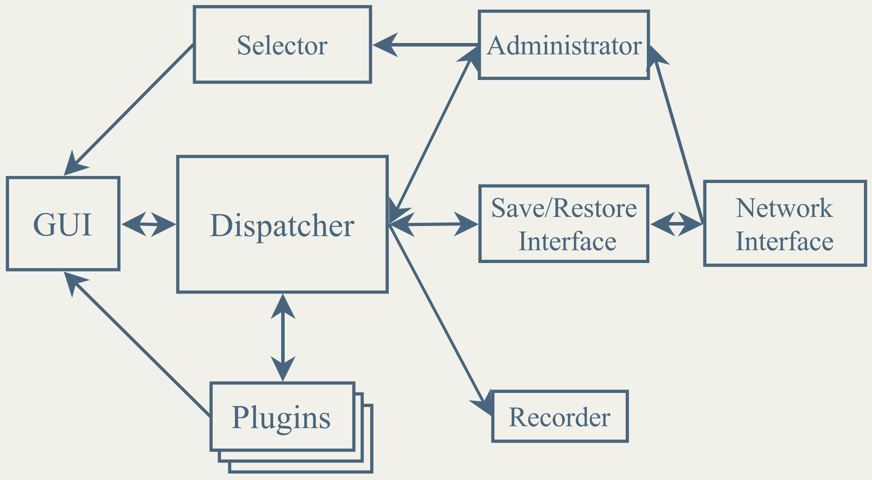
17 of the 25 applicable scenarios needed by the next field trial; 8 were for the longer term

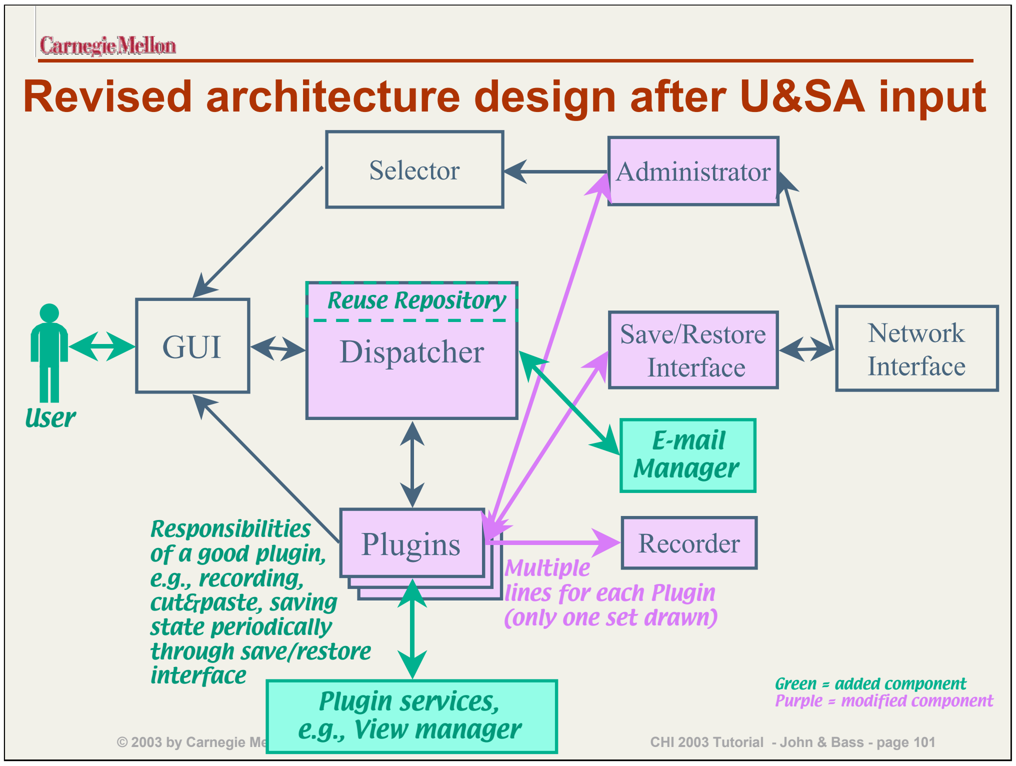
Easy for the development team to give concrete examples of these scenarios for their users, often from direct observation during the field trials



Carnegie Mellon

Proposed architecture redesign





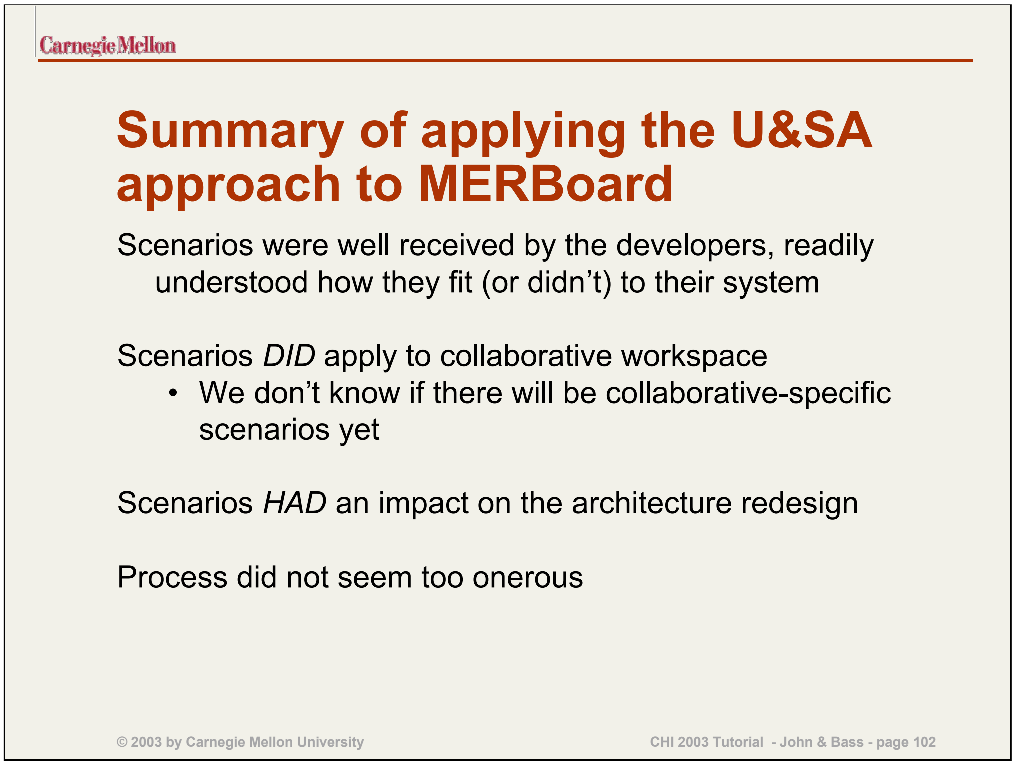
Carnegie Mellon

Revised architecture design after U&SA input

Selector
Administrator
Reuse Repository
Dispatcher
GUI
User
Save/Restore Interface
Network Interface
E-mail Manager
Plugins
Recorder
Responsibilities of a good plugin, e.g., recording, cut&paste, saving state periodically through save/restore interface
Multiple lines for each Plugin (only one set drawn)
Plugin services, e.g., View manager
Green = added component
Purple = modified component

© 2003 by Carnegie Me

CHI 2003 Tutorial - John & Bass - page 101


Carnegie Mellon

Summary of applying the U&SA approach to MERBoard

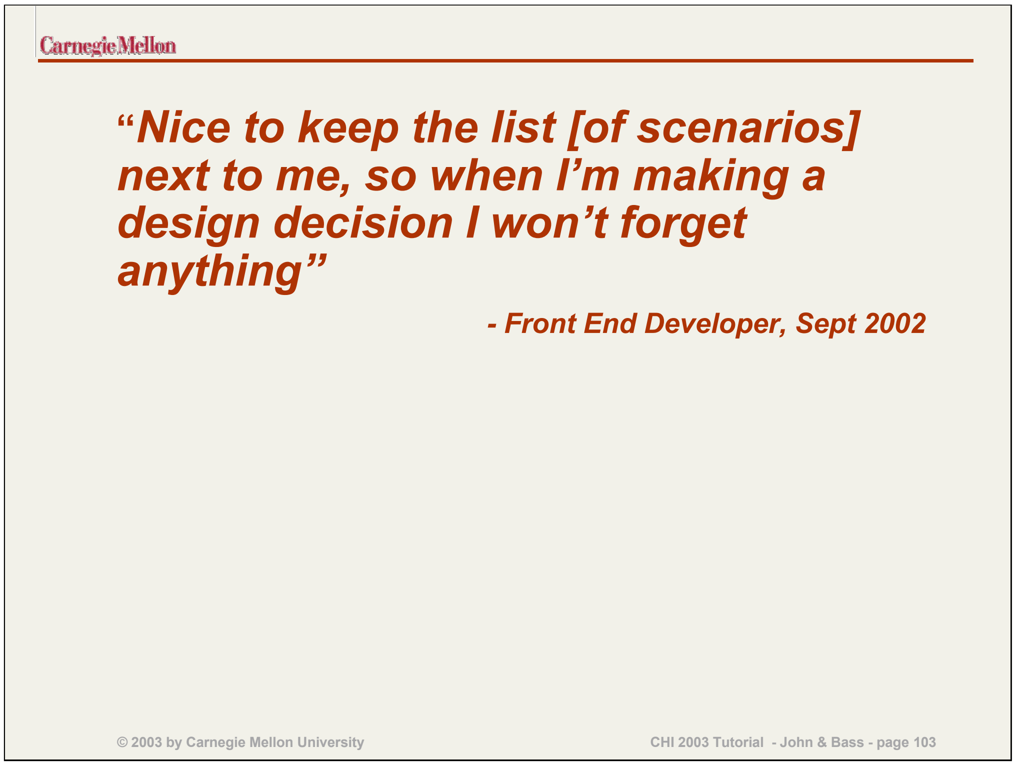
Scenarios were well received by the developers, readily understood how they fit (or didn't) to their system

Scenarios *DID* apply to collaborative workspace

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

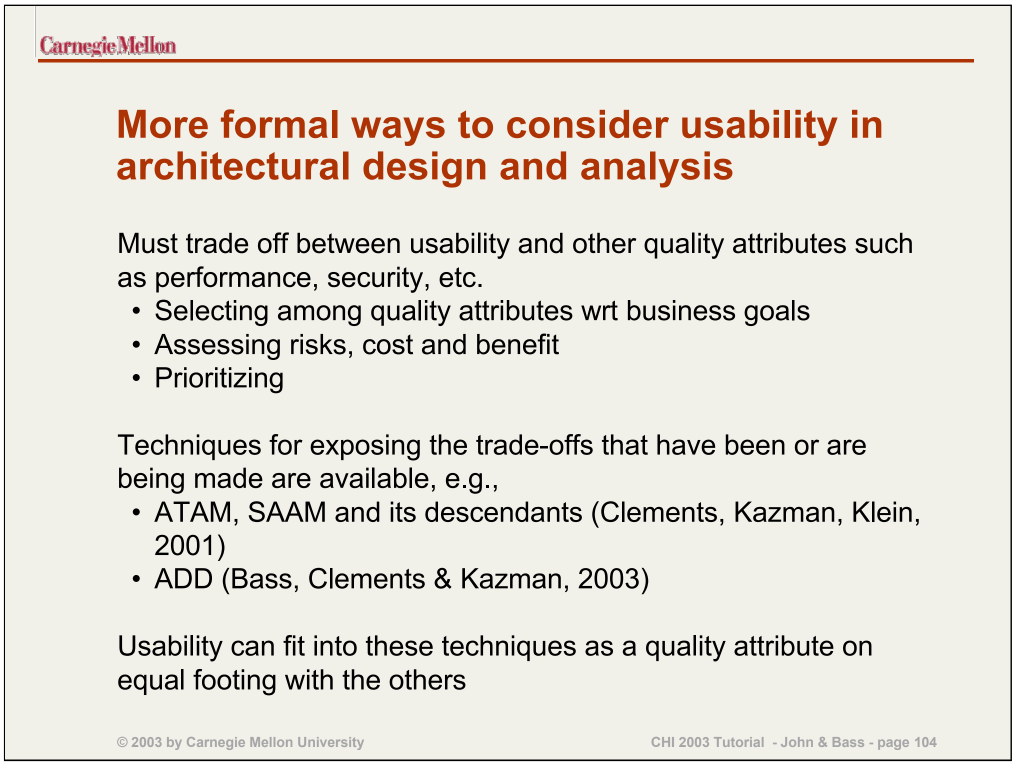
Scenarios *HAD* an impact on the architecture redesign

Process did not seem too onerous



"Nice to keep the list [of scenarios] next to me, so when I'm making a design decision I won't forget anything"

- Front End Developer, Sept 2002



Carnegie Mellon

More formal ways to consider usability in architectural design and analysis

Must trade off between usability and other quality attributes such as performance, security, etc.

- Selecting among quality attributes wrt business goals
- Assessing risks, cost and benefit
- Prioritizing

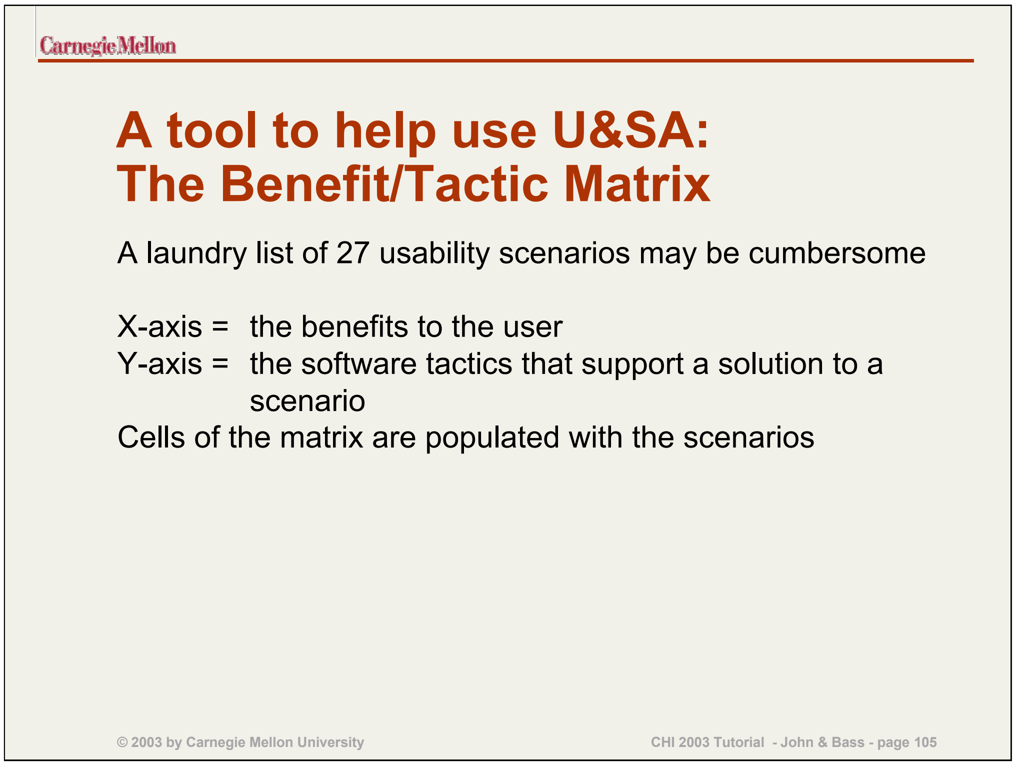
Techniques for exposing the trade-offs that have been or are being made are available, e.g.,

- ATAM, SAAM and its descendants (Clements, Kazman, Klein, 2001)
- ADD (Bass, Clements & Kazman, 2003)

Usability can fit into these techniques as a quality attribute on equal footing with the others

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

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

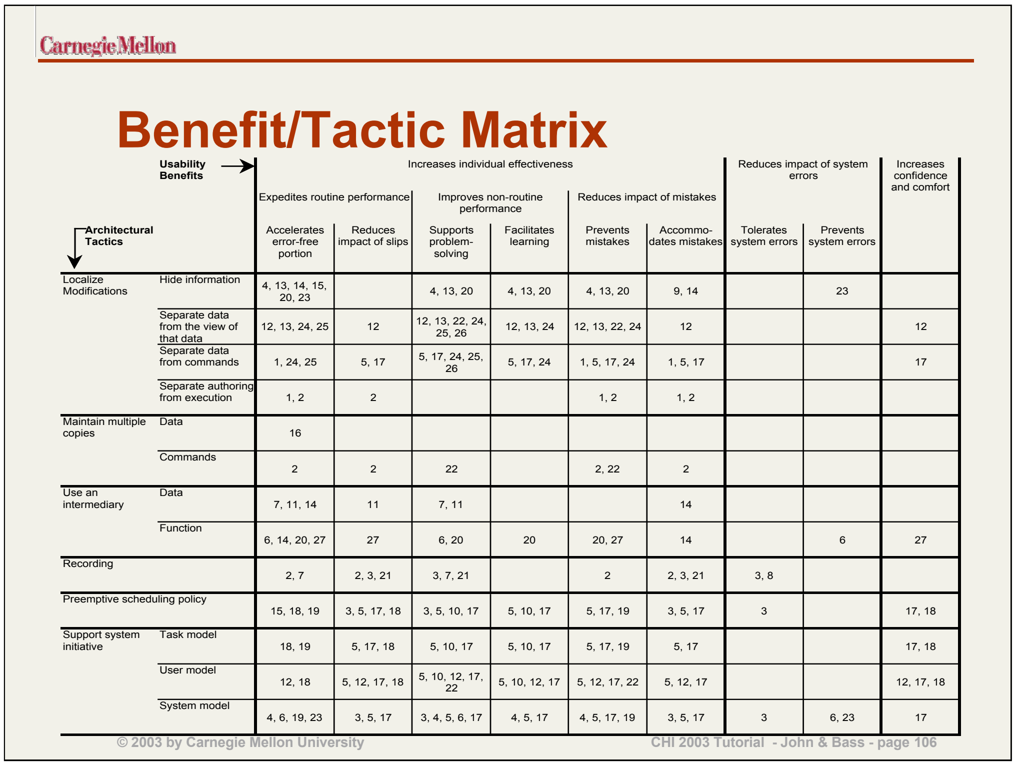


Carnegie Mellon

A tool to help use U&SA: The Benefit/Tactic Matrix

A laundry list of 27 usability scenarios may be cumbersome

X-axis = the benefits to the user
Y-axis = the software tactics that support a solution to a scenario
Cells of the matrix are populated with the scenarios

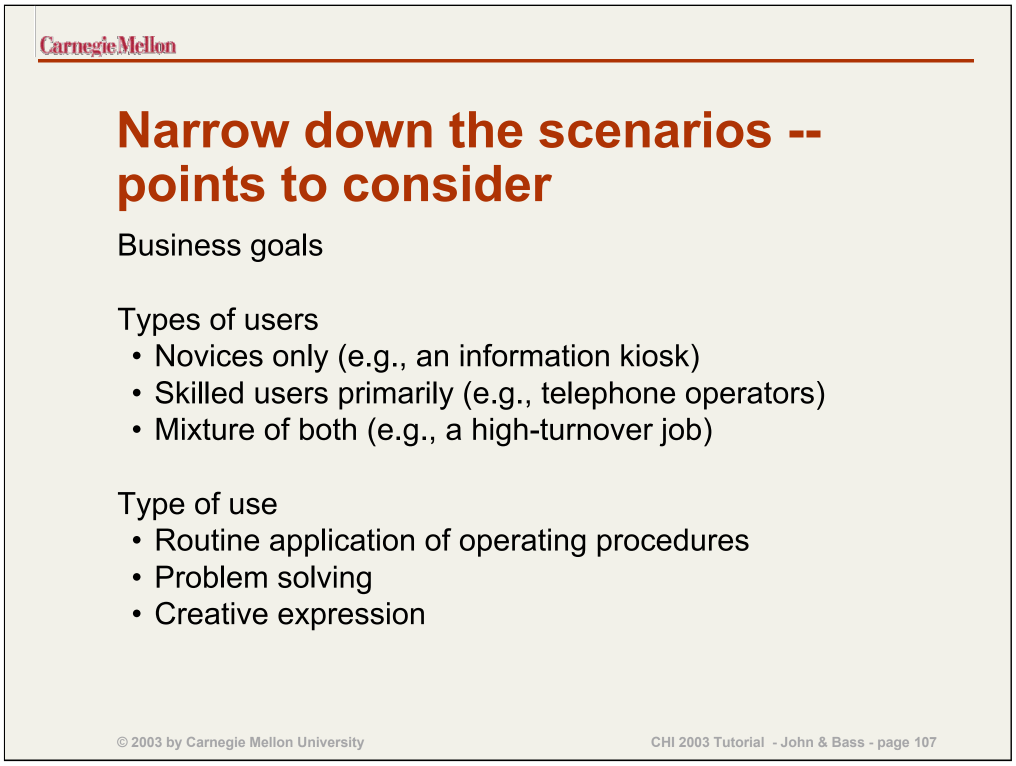

Carnegie Mellon

Benefit/Tactic Matrix

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

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

A larger version of this matrix appears in Appendix IV.

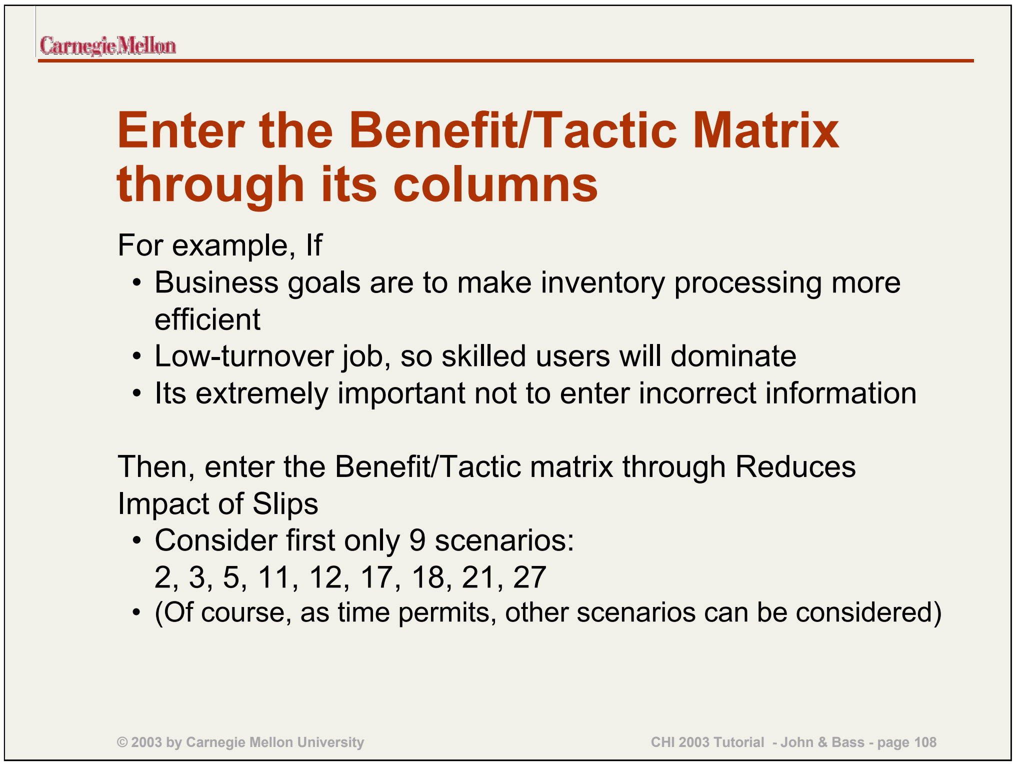


Narrow down the scenarios -- points to consider

Business goals

Types of users
- Novices only (e.g., an information kiosk)
- Skilled users primarily (e.g., telephone operators)
- Mixture of both (e.g., a high-turnover job)

Type of use
- Routine application of operating procedures
- Problem solving
- Creative expression



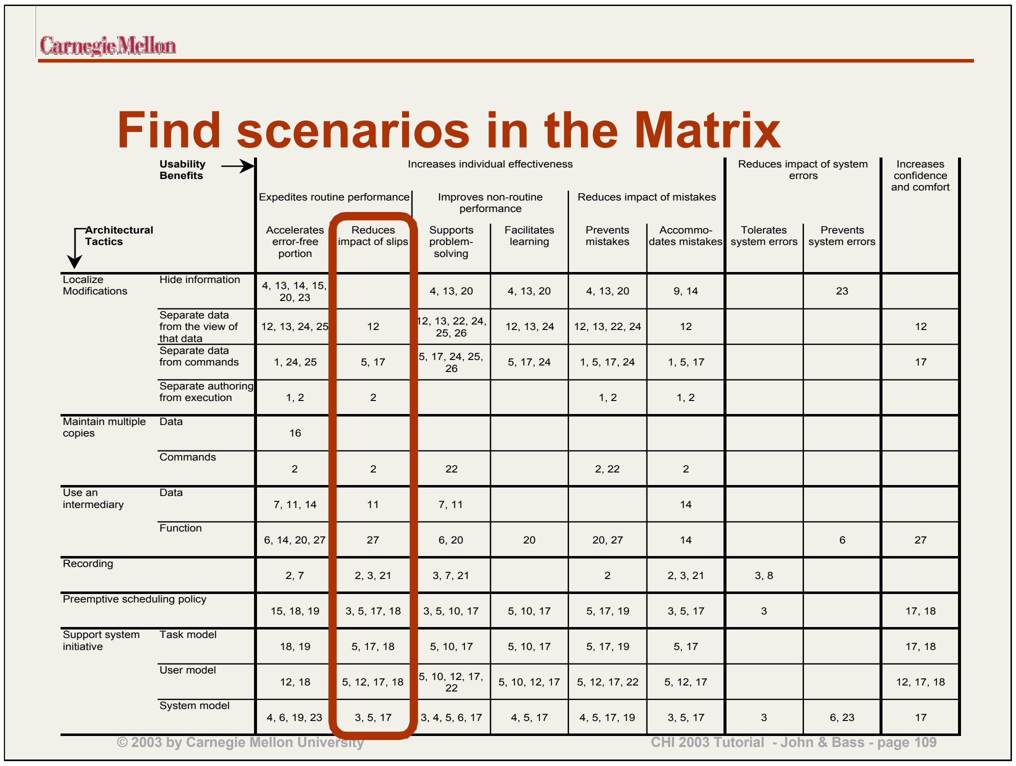
Enter the Benefit/Tactic Matrix through its columns

For example, If

- Business goals are to make inventory processing more efficient
- Low-turnover job, so skilled users will dominate
- Its extremely important not to enter incorrect information

Then, enter the Benefit/Tactic matrix through Reduces Impact of Slips

- Consider first only 9 scenarios:
  2, 3, 5, 11, 12, 17, 18, 21, 27
- (Of course, as time permits, other scenarios can be considered)

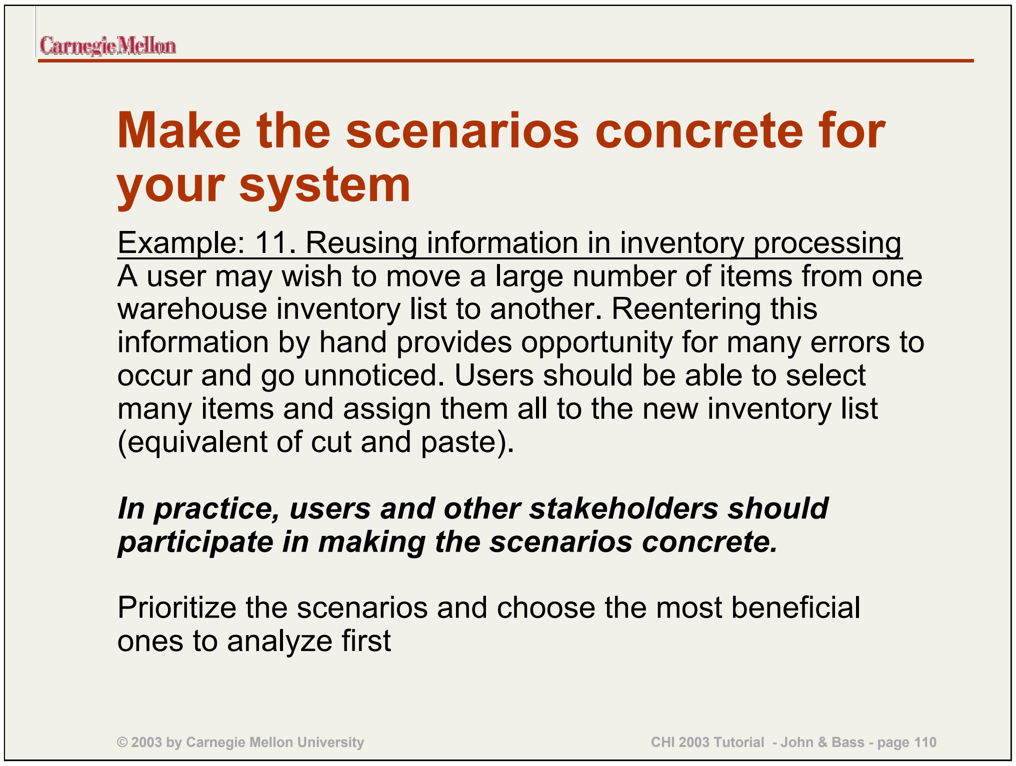


Carnegie Mellon

Find scenarios in the Matrix

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

2. Aggregating commands
3. Canceling commands
5. Checking for correctness
11. Reusing information
12. Supporting international use
17. Observing system state
18. Working at the user’s pace
21. Supporting undo
27. Supporting personalization



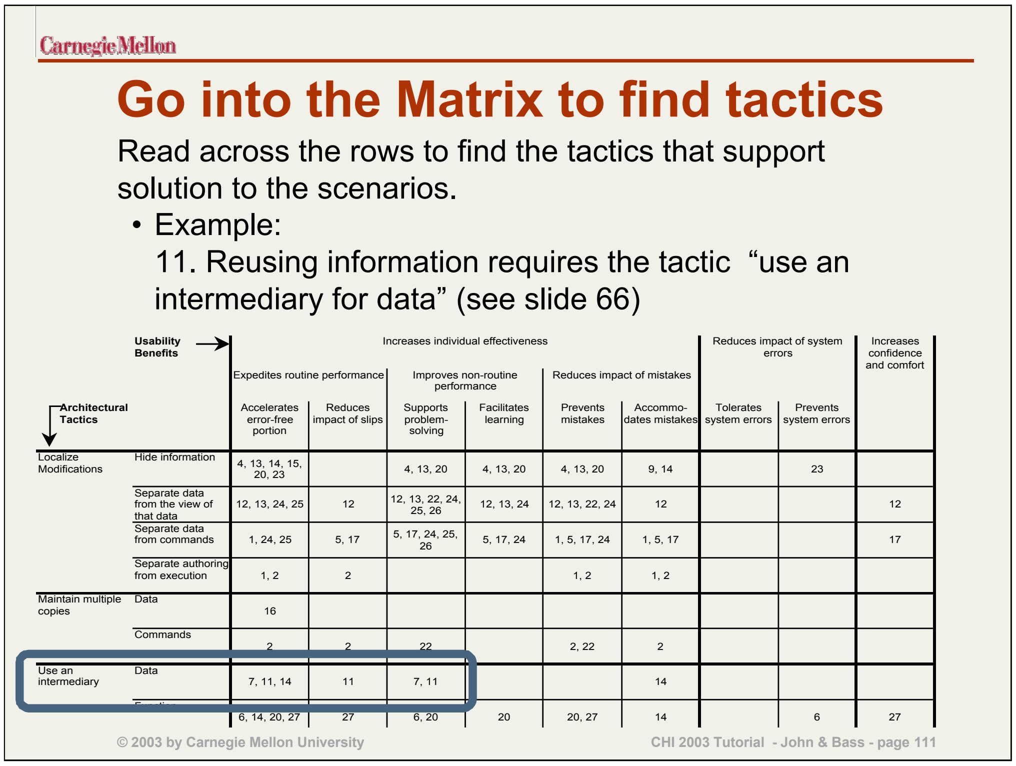
CarnegieMellon

Make the scenarios concrete for your system

Example: 11. Reusing information in inventory processing
A user may wish to move a large number of items from one warehouse inventory list to another. Reentering this information by hand provides opportunity for many errors to occur and go unnoticed. Users should be able to select many items and assign them all to the new inventory list (equivalent of cut and paste).

In practice, users and other stakeholders should participate in making the scenarios concrete.

Prioritize the scenarios and choose the most beneficial ones to analyze first



Carnegie Mellon

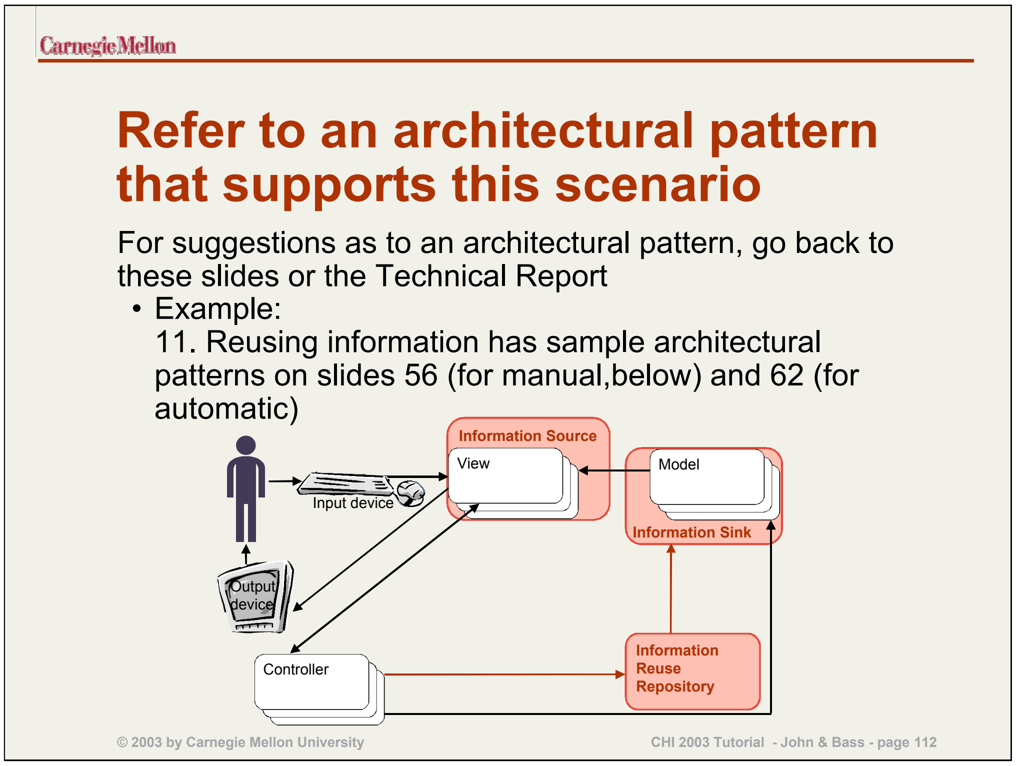
Go into the Matrix to find tactics

Read across the rows to find the tactics that support solution to the scenarios.

- Example:

  11. Reusing information requires the tactic “use an intermediary for data” (see slide 66)

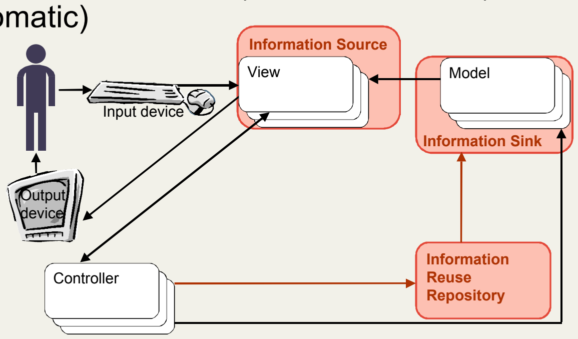
	Usability Benefits →	Increases individual effectiveness						Reduces impact of system errors		Increases confidence and comfort
		Expedites routine performance		Improves non-routine performance		Reduces impact of mistakes				
Architectural Tactics ↓		Accelerates error-free portion	Reduces impact of slips	Supports problem-solving	Facilitates learning	Prevents mistakes	Accommo-dates mistakes	Tolerates system errors	Prevents system errors	
Localize Modifications	Hide information	4, 13, 14, 15, 20, 23		4, 13, 20	4, 13, 20	4, 13, 20	9, 14		23	
	Separate data from the view of that data	12, 13, 24, 25	12	12, 13, 22, 24, 25, 26	12, 13, 24	12, 13, 22, 24	12			12
	Separate data from commands	1, 24, 25	5, 17	5, 17, 24, 25, 26	5, 17, 24	1, 5, 17, 24	1, 5, 17			17
	Separate authoring from execution	1, 2	2			1, 2	1, 2			
Maintain multiple copies	Data	16								
	Commands	2	2	22		2, 22	2			
Use an intermediary	Data	7, 11, 14	11	7, 11			14			
	Function	6, 14, 20, 27	27	6, 20	20	20, 27	14		6	27

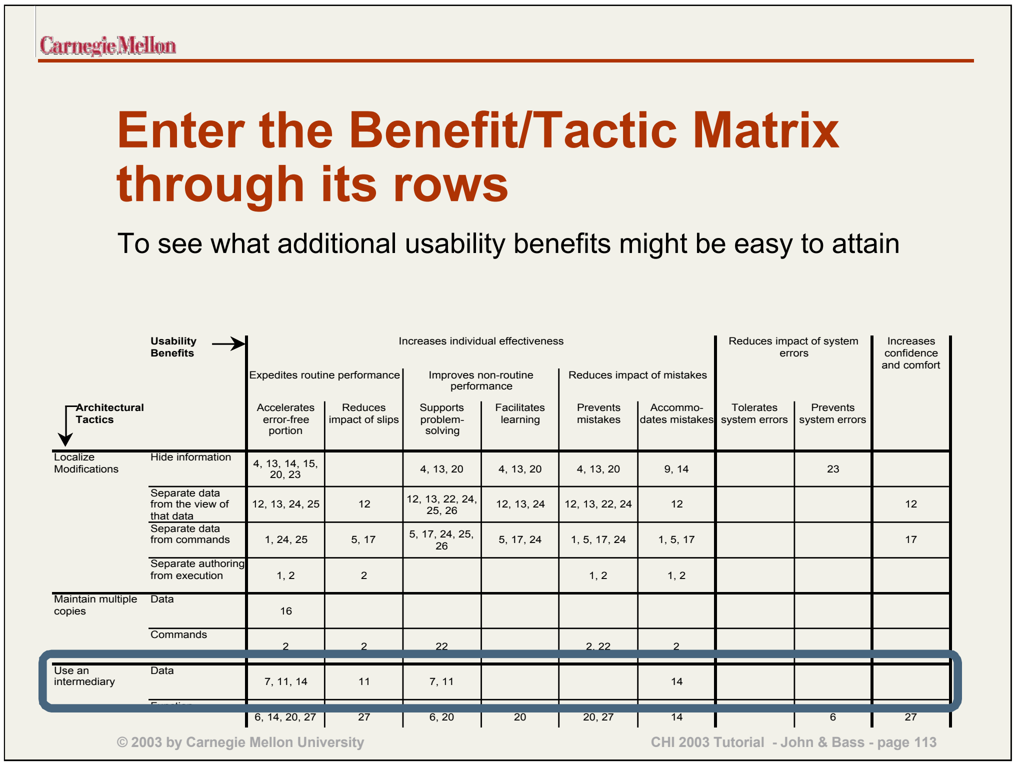


Refer to an architectural pattern that supports this scenario

For suggestions as to an architectural pattern, go back to these slides or the Technical Report

- Example:
  11. Reusing information has sample architectural patterns on slides 56 (for manual,below) and 62 (for automatic)





Carnegie Mellon

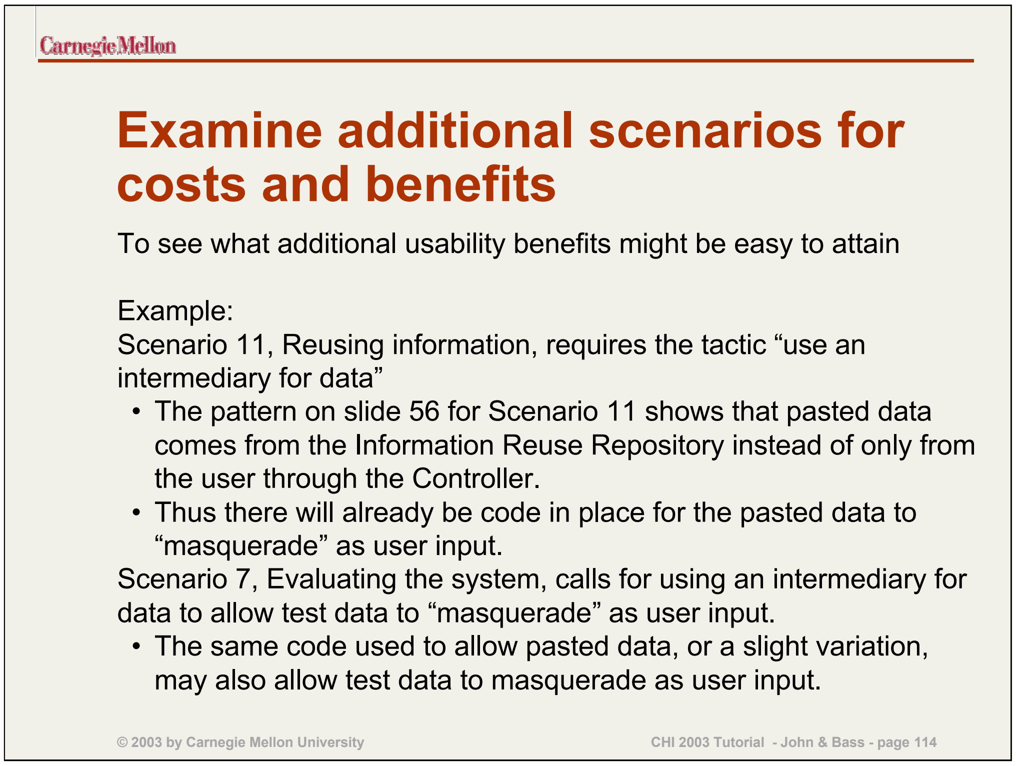
Enter the Benefit/Tactic Matrix through its rows

To see what additional usability benefits might be easy to attain

Architectural Tactics	Usability Benefits	Increases individual effectiveness						Reduces impact of system errors		Increases confidence and comfort
		Expedites routine performance		Improves non-routine performance		Reduces impact of mistakes				
		Accelerates error-free portion	Reduces impact of slips	Supports problem-solving	Facilitates learning	Prevents mistakes	Accommo-dates mistakes	Tolerates system errors	Prevents system errors	
Localize Modifications	Hide information	4, 13, 14, 15, 20, 23		4, 13, 20	4, 13, 20	4, 13, 20	9, 14		23	
	Separate data from the view of that data	12, 13, 24, 25	12	12, 13, 22, 24, 25, 26	12, 13, 24	12, 13, 22, 24	12			12
	Separate data from commands	1, 24, 25	5, 17	5, 17, 24, 25, 26	5, 17, 24	1, 5, 17, 24	1, 5, 17			17
	Separate authoring from execution	1, 2	2			1, 2	1, 2			
Maintain multiple copies	Data	16								
	Commands	2	2	22		2, 22	2			
Use an intermediary	Data	7, 11, 14	11	7, 11			14			
	[illegible]	6, 14, 20, 27	27	6, 20	20	20, 27	14		6	27

© 2003 by Carnegie Mellon University CHI 2003 Tutorial - John & Bass - page 113

Scenario 7 and 14 use the same tactic as Scenario 11.



Examine additional scenarios for costs and benefits

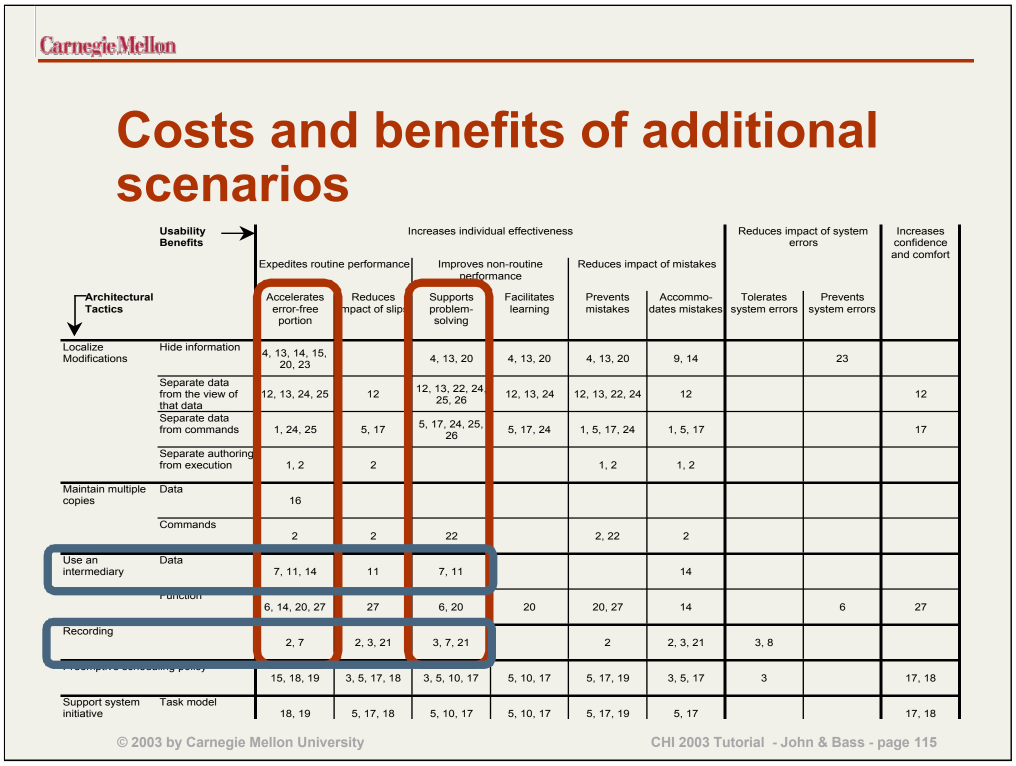
To see what additional usability benefits might be easy to attain

Example:
Scenario 11, Reusing information, requires the tactic “use an intermediary for data”

- The pattern on slide 56 for Scenario 11 shows that pasted data comes from the Information Reuse Repository instead of only from the user through the Controller.
- Thus there will already be code in place for the pasted data to “masquerade” as user input.

Scenario 7, Evaluating the system, calls for using an intermediary for data to allow test data to “masquerade” as user input.

- The same code used to allow pasted data, or a slight variation, may also allow test data to masquerade as user input.



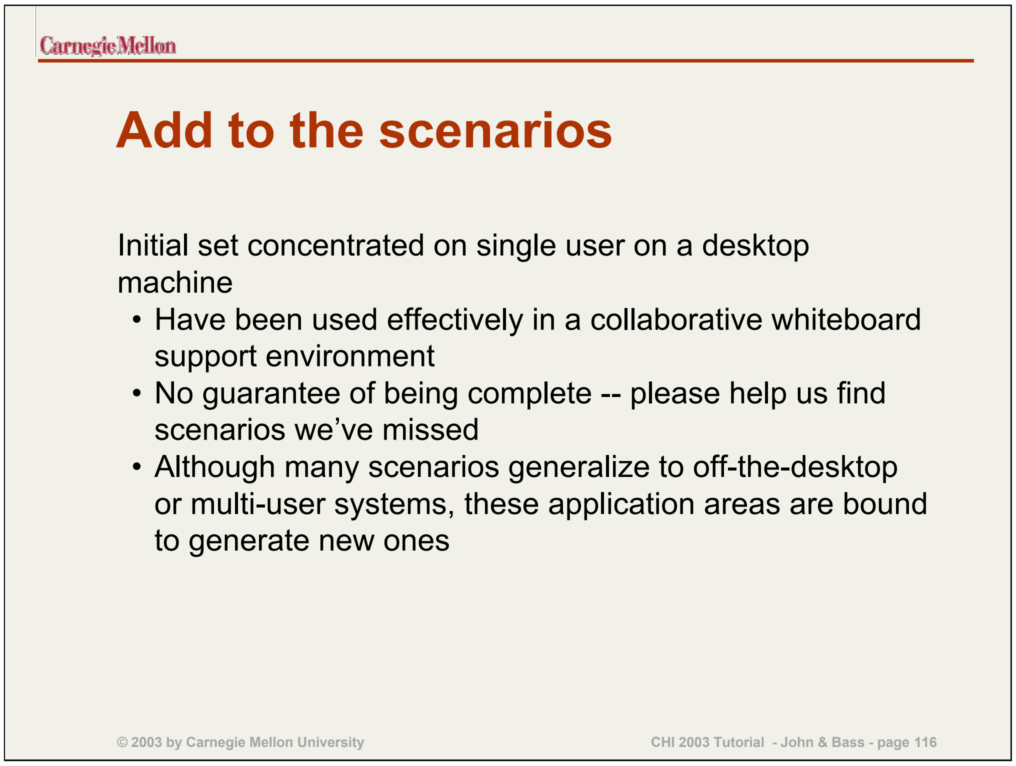
Carnegie Mellon

Costs and benefits of additional scenarios

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

© 2003 by Carnegie Mellon University CHI 2003 Tutorial - John & Bass - page 115

The tactic of using an intermediary for data will already be in place to support Scenario 11. It is possible that the code that implements this tactic may be reused or modified to support Scenario 7. In addition, to get the benefits provided by Scenario 7 (accelerates error-free portion of routine performance of the development team and supports problem solving of the development team), the tactic of recording must be added. The design team can assess the costs of adding that tactic versus the potential usability benefits.


Carnegie Mellon

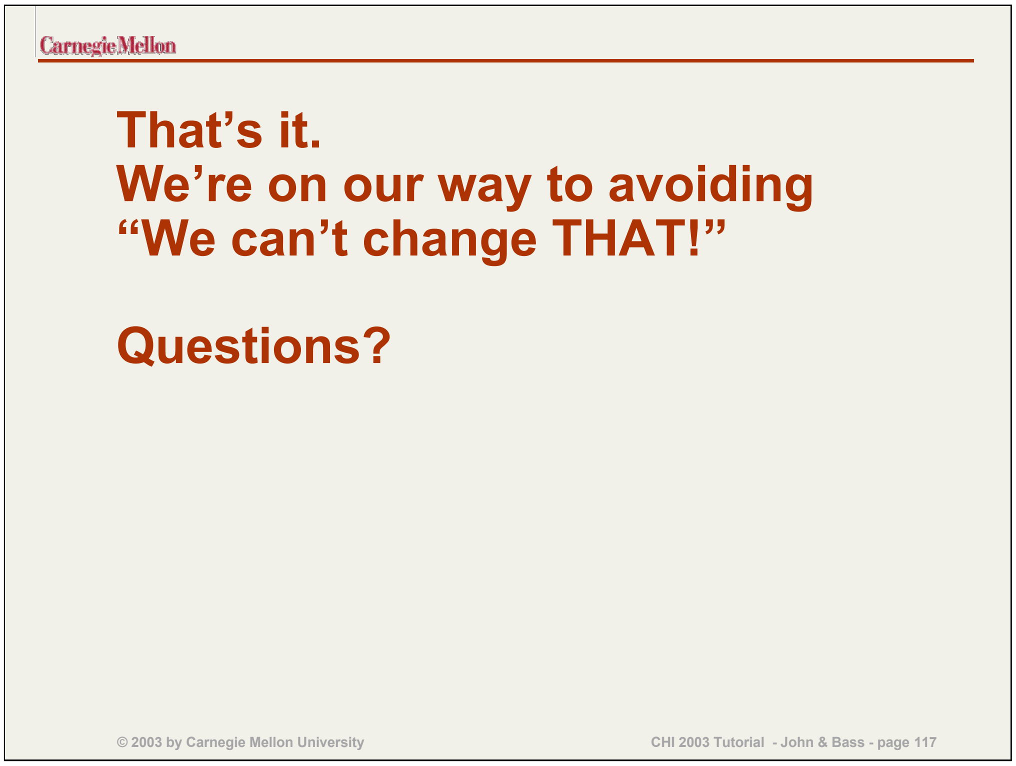
Add to the scenarios

Initial set concentrated on single user on a desktop machine

- Have been used effectively in a collaborative whiteboard support environment
- No guarantee of being complete -- please help us find scenarios we've missed
- Although many scenarios generalize to off-the-desktop or multi-user systems, these application areas are bound to generate new ones

On the paper we hand out, write any general scenario you can think of that is important in the systems you use and/or design. We'll collect them and update our list and architectural analyses.

If you think of any others, or would like updates on this work, please send us e-mail at
ljb@sei.cmu.edu
bej@cs.cmu.edu


Carnegie Mellon

That’s it.
We’re on our way to avoiding “We can’t change THAT!”

Questions?

Appendix I
General Usability Scenarios

(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)

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 tactics 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 online encyclopedia without consulting the table of contents. If the system takes too long to display the new data or if the user must execute a cumbersome command sequence to arrive at her or his destination, the user's time will be wasted. System designers should strive to ensure that users can navigate within a view easily and attempt to keep wait times reasonably short. (See: Working at the User's Pace)

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.

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

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

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

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
Software Architecture Tactics Hierarchy

(Adapted 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 and Bass, L, Clements, P., and Kazman, R. Software Architecture in Practice, 2nd edition, 2003. Addison-Wesley Longman.
In the original version of CMU/SEI-2001-TR-005 and several of our previously published papers, we used the term "mechanisms" instead of "tactics" to refer to techniques for addressing the usability scenarios. The tactics given here extend the tactics given in Chapter 5 of Software Architecture in Practice.)

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

Table 2 Software Architecture Hierarchy

Localize expected changes

- Maintain semantic consistency
- Separate data from commands
- Separate data from the view of that data
- Separate authoring from execution

Maintain multiple copies

- Data
- Commands

Use an intermediary

- Data
- Function

Recording

Preemptive scheduling Policy

Support system initiative

- Task model
- User model
- System model

An item that affects the range of these tactics is how broadly they are shared. That is, embedding a use of a tactic in the infrastructure and making it available to any application is more far reaching than keeping a tactic within a single application or within a set of applications. We do not capture this range consideration within the description of the tactics.

1 Localize expected changes

Although there is no necessarily a precise relationship between the number of modules affected by a set of changes and the cost of implementing those changes, keeping modifications restricted to a small set of modules will generally reduce the cost. The goal of these tactics is to assign responsibilities to modules such that anticipated changes will be limited in scope. We identify four tactics to accomplish this.

1.1 Hide information

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

1.2 Separate data from commands

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

1.3 Separate data from the view of that data

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

1.4 Separate authoring from execution

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

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

2 Maintain multiple copies

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

2.1 Data

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

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

2.2 Commands

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

3 Use an intermediary

Using an intermediary is a tactic intended to reduce the coupling between distinct elements.

3.1 Data

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

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

3.2 Function

An intermediary function is interposed between various alternative methods of accomplishing a particular service. Terms such as a “virtual device,” a “virtual tool kit,” a “strategy pattern,” and a “factory pattern” describe this tactic. Binding between the service requester and the alternative service provider service may be done before or at execution time. In any case, the service requester uses a single interface to interact with the function intermediary, and the function intermediary translates the information received specifically for the alternative chosen.

4 Recording

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

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

5 Preemptive scheduling policy

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

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

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

6 Support system initiative

The terms “system initiative” and “user initiative” are used to differentiate whether the system is merely responding to the user or taking the initiative to offer some information or action without the user explicitly requesting. Examples of system initiative are progress bars indicating how close to completion a particular task is, making assumptions about the user to control scrolling rate or making assumptions about the task to fix misspellings. System initiative is supported by keeping a model (either implicit or explicit) within the system which allows the model to be used for prediction. Different models can predict different types of items. Each model requires various types of input to accomplish its prediction. Clearly identifying the models that the system uses to predict either its own behavior or the user's intention enables designers to tailor and modify those models either dynamically or off-line during development.

6.1 Task model

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

6.2 User model

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

6.3 System

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

Appendix IV
Benefits/Tactics Matrix

In this appendix, we present a matrix that puts the benefits hierarchy on one axis and the tactics hierarchy on the other. Each cell contains the general usability scenarios that correspond to the tactics and benefit hierarchies.

On the one hand, this matrix reproduces the information presented in Sections 4 and 6 in CMU/SEI-2001-TR-005. On the other, the matrix provides additional benefits. The software design team can decide which usability benefits are most valued in a particular project, use the matrix to focus on the general scenarios providing those benefits to see which are applicable to the project, and then read off the software tactics necessary to implement those scenarios. The team can use this information to generate the architecture or to evaluate an existing architecture to see what usability risks might be inherent in their design. Alternatively, the team could look at the tactics included in a current system design and use the matrix to discover which general usability scenarios could be implemented using those tactics, and which additional usability scenarios could be addressed with only small changes to the architecture. We expect this matrix to be the vehicle for referencing the work presented here and thereby increase its utility beyond the linear format of prose and diagrams.

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

KEY

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

References

References in Usability and Software Architecture

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.
Downloadable from
`www.sei.cmu.edu/publications/documents/01.reports/01tr005.html`

References in software engineering and software architecture cited in the slides or appendices

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.
Downloadable from
`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., 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.

Klein, M. & Bachmann, F. (2000). *Quality Attribute Design Primitives* (CMU/SEI-2000-TN-2000-017). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University.
Downloadable from
`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 cited in the slides or appendices

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.

Miller, D. P. & Swain, A. D. (1987) Human Error and Human Reliability. In Gavriel Salvendy, ed. *Handbook of Human Factors*, New York: John Wiley and Sons, Inc., pp. 219-257.

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.