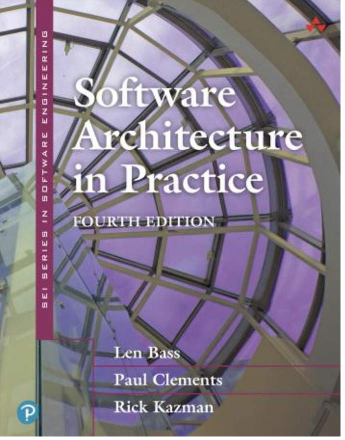
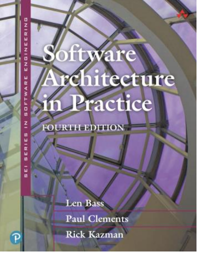

SEI Series in Software Engineering
Software Architecture in Practice
FOURTH EDITION
Len Bass
Paul Clements
Rick Kazman

Chapter 13: Usability

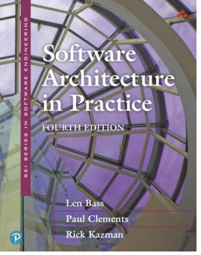
People ignore design that ignores people.

—Frank Chimero



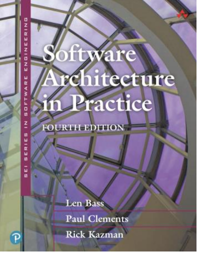
Chapter Outline

- What is Usability?
- Usability General Scenario
- Tactics for Usability
- Tactics-Based Questionnaire for Usability
- Patterns for Usability
- Summary



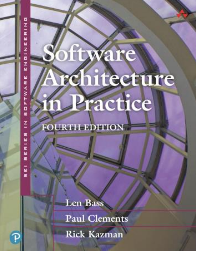
What is Usability?

- Usability is concerned with how easy it is for the user to accomplish a desired task and the kind of user support that the system provides.
- Usability comprises the following areas:
  – *Learning system features*
  – *Using a system efficiently*
  – *Minimizing the impact of user errors*
  – *Adapting the system to user needs*
  – *Increasing confidence and satisfaction*



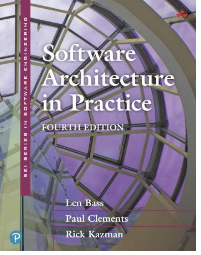
What is Usability?

- There is a strong connection between the achievement of usability and modifiability.
- The user interface design process consists of generating and then testing a user interface design.
- It is highly unlikely that you will get this right the first time, so you should plan to iterate this process—and hence you should design your architecture to make that iteration less painful.



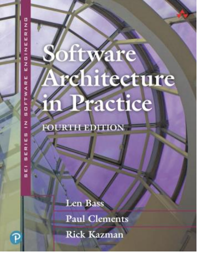
Usability General Scenario

Portion of Scenario	**Description**	**Possible Values**
Source	Where does the stimulus come from?	The end user (who may be in a specialized role, such as a system or network administrator) is the primary source of the stimulus for usability. An external event arriving at a system (to which the user may react) may also be a stimulus source.
Stimulus	What does the end user want?	End user wants to: ▪ Use a system efficiently ▪ Learn to use the system ▪ Minimize the impact of errors ▪ Adapt the system ▪ Configure the system
Environment	When does the stimulus reach the system?	The user actions with which usability is concerned always occur at runtime or at system configuration time.



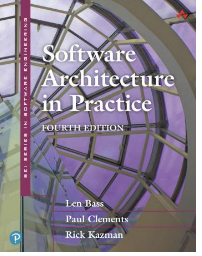
Usability General Scenario

Artifacts	What portion of the system is being stimulated?	Common examples include: ▪ A GUI ▪ A command-line interface ▪ A voice interface ▪ A touch screen
Response	How should the system respond?	The system should: ▪ Provide the user with the features needed ▪ Anticipate the user's needs ▪ Provide appropriate feedback to the user
Response measure	How is the response measured?	One or more of the following: ▪ Task time ▪ Number of errors ▪ Learning time ▪ Ratio of learning time to task time ▪ Number of tasks accomplished ▪ User satisfaction ▪ Gain of user knowledge ▪ Ratio of successful operations to total operations ▪ Amount of time or data lost when an error occurs

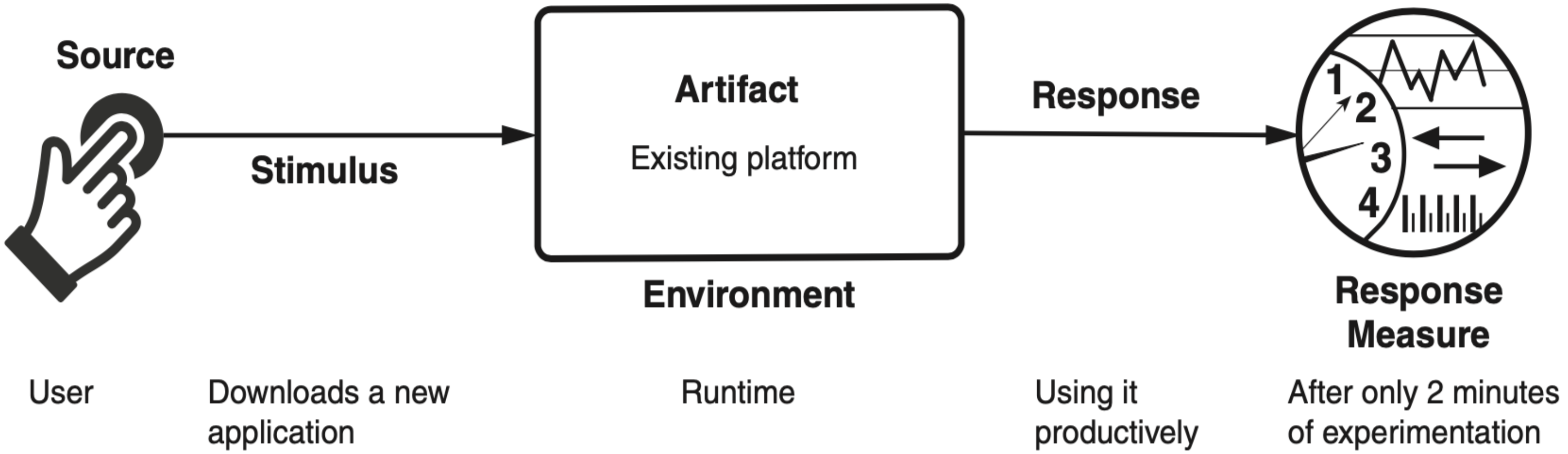


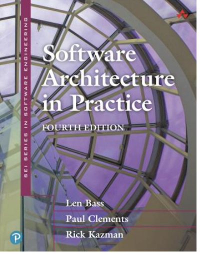
Sample Concrete Usability Scenario

- *The user downloads a new application and is using it productively after 2 minutes of experimentation.*



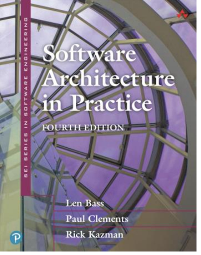
Sample Concrete Usability Scenario



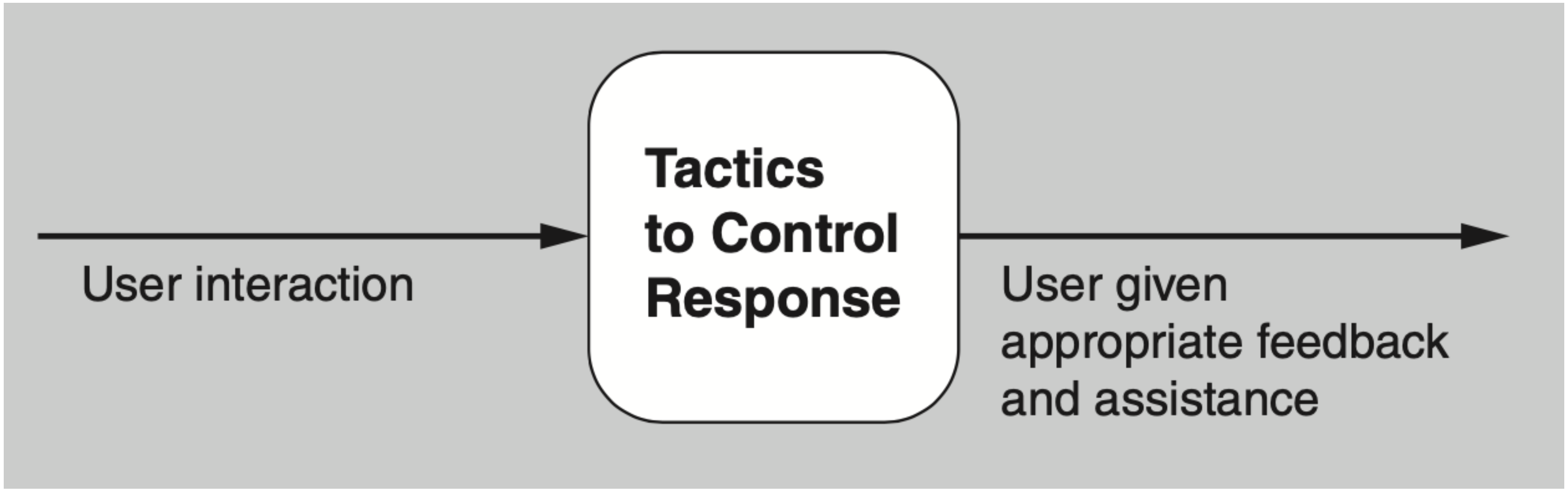


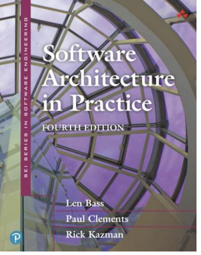
Goal of Usability Tactics

- There are two categories of tactics for Usability:
  - The first category deals with supporting user initiative.
  - The second deals with supporting system initiative.

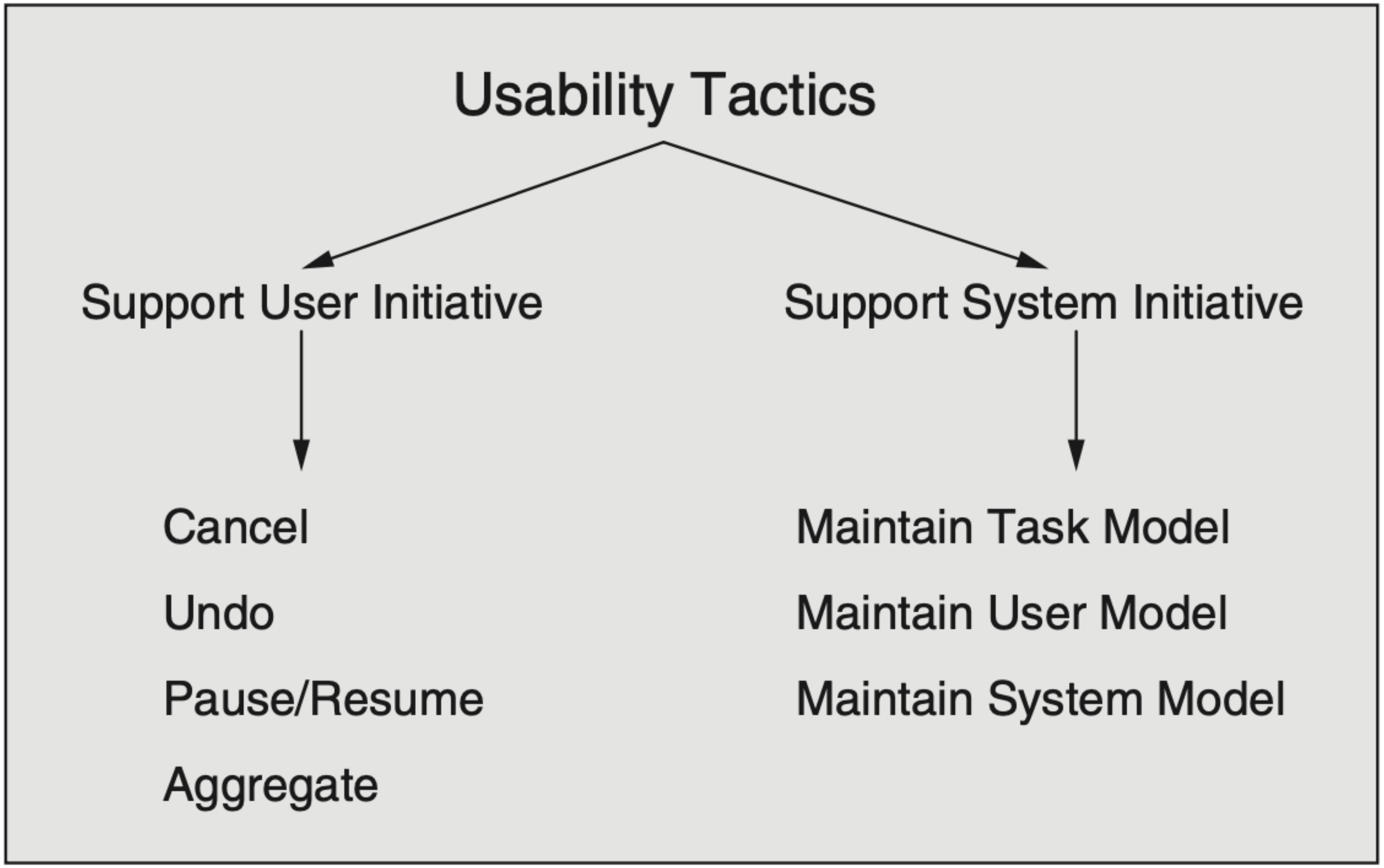


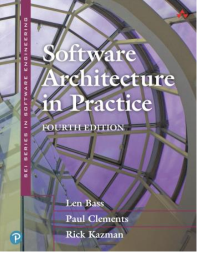
Goal of Usability Tactics


User interaction
Tactics to Control Response
User given appropriate feedback and assistance



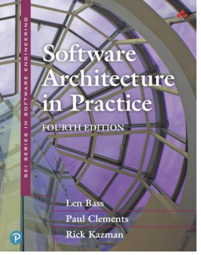
Usability Tactics





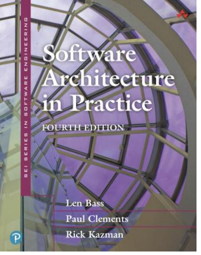
Support User Initiative

- *Cancel*. When the user issues a cancel command, the system must be listening for it; the activity being canceled must be terminated; resources being used by the canceled activity must be freed; and components that are collaborating with the canceled activity must be informed so that they can also take appropriate action.
- *Undo*. To support the ability to undo, the system must maintain enough information about system state so that an earlier state may be restored, at the user’s request.
- *Pause/resume*. When a user has initiated a long-running operation it is often useful to provide the ability to pause and resume the operation. Pausing a long-running operation may be done to temporarily free resources so that they may be reallocated to other tasks.
- *Aggregate*. When a user is performing repetitive operations, or operations that affect a large number of objects in the same way, it is useful to provide the ability to aggregate the lower-level objects into a single group, so that the operation may be applied to the group, thus freeing the user from the drudgery, and potential for mistakes, of doing the same operation repeatedly.



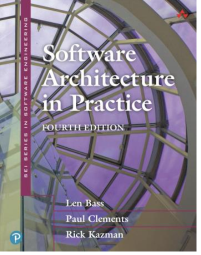
Support System Initiative

- *Maintain task model*. The task model is used to determine context so the system can have some idea of what the user is attempting to do and provide assistance.
- *Maintain user model*. This model explicitly represents the user's knowledge of the system, the user's behavior in terms of expected response time, and other aspects.
- *Maintain system model*. The system maintains an explicit model of itself. This is used to determine expected system behavior so that appropriate feedback can be given to the user.



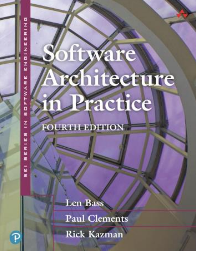
Tactics-Based Questionnaire for Usability

Tactics Group	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Support User Initiative	Is the system able to listen to and respond to a **cancel** command?				
	Is it possible to **undo** the last command, or the last several commands?				
	Is it possible to **pause** and then **resume** long-running operations?				
	Is it possible to **aggregate UI objects** into a group and apply operations on the group?				



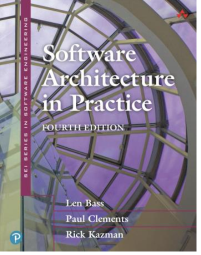
Tactics-Based Questionnaire for Usability

Support System Initiative	Does the system maintain a **model of the task**?
	Does the system maintain a **model of the user**?
	Does the system maintain a **model of itself**?



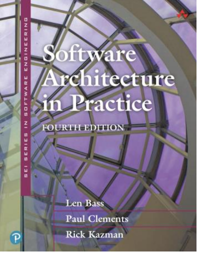
Model-View Controller Pattern

- MVC is likely the most widely known pattern for usability.
- This pattern is focused on separating the model—the underlying “business” logic of the system—from its realization in one or more UI *views*.
- The model sends updates to a *view*, which a user would see and interact with.
- User interactions are transmitted to the *controller*, which interprets them and sends operations to the *model*, which changes its state in response.
- The reverse path is where the model might be changed and the controller would send updates to the view.



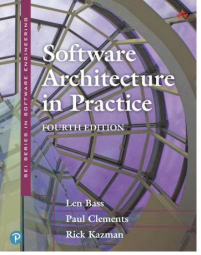
Model-View Controller Pattern Benefits

- Benefits:
  - Because MVC promotes clear separation of concerns, changes to one aspect of the system, such as the layout of the UI (the view), often have no consequences for the model or the controller.
  - Additionally, because MVC promotes separation of concerns, developers can be working on all aspects of the pattern relatively independently and in parallel. These separate aspects can also be tested in parallel.
  - A model can be used in systems with different views, or a view might be used in systems with different models.



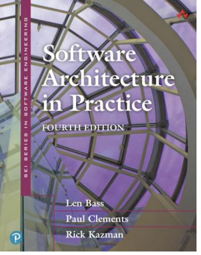
Model-View Controller Pattern Tradeoffs

- Tradeoffs:
  - MVC can become burdensome for complex UIs, as information is often sprinkled throughout several components.
  - For simple UIs, MVC adds up-front complexity that may not pay off in downstream savings.
  - MVC adds a small amount of latency to user interactions.



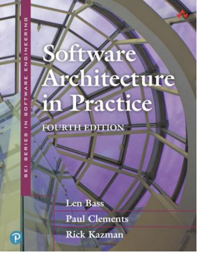
Observer Pattern

- The observer pattern is a way to link some functionality with one or more views. This pattern has a *subject*—the entity being observed—and one or more *observers* of that subject.
- Observers need to register themselves with the subject; then, when the state of the subject changes, the observers are notified.



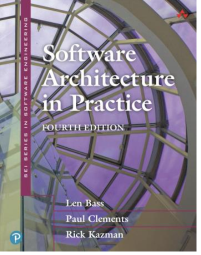
Oberver Pattern Benefits

- This pattern separates some underlying functionality from the concern of how, and how many times, this functionality is presented.
- The observer pattern makes it easy to change the bindings between the subject and the observers at runtime.



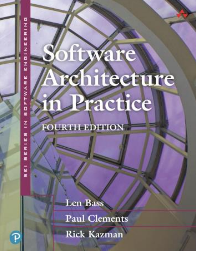
Observer Pattern Tradeoffs

- The observer pattern is overkill if multiple views of the subject are not required.
- The observer pattern requires that all observers register and de-register with the subject.
- If observers neglect to de-register, then their memory is never freed, which effectively results in a memory leak. In addition, this can negatively affect performance, since obsolete observers will continue to be invoked.
- Observers may need to do considerable work to determine if and how to reflect a state update, and this work may be repeated for each observer.



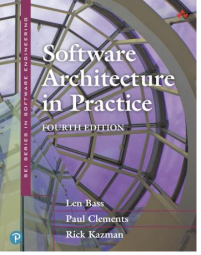
Memento Pattern

- The memento pattern is a common way to implement the undo tactic. This pattern features three major components: the *originator*, the *caretaker*, and the *memento*.
- The originator is processing some stream of events that change its state (originating from user interaction).
- The caretaker is sending events to the originator that cause it to change its state.
- When the caretaker is about to change the state of the originator, it can request a memento—a snapshot of the existing state—and can use this artifact to restore that existing state if needed, by simply passing the memento back to the originator.



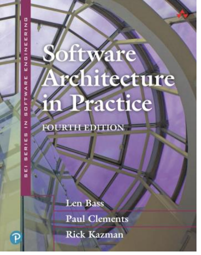
Memento Pattern Benefits

- The obvious benefit of this pattern is that you delegate the complicated process of implementing undo, and figuring out what state to preserve, to the class that is actually creating and managing that state.
- In consequence, the originator’s abstraction is preserved and the rest of the system does not need to know the details.



Memento Pattern Tradeoffs

- Depending on the nature of the state being preserved, the memento can consume arbitrarily large amounts of memory, which can affect performance.
- In some programming languages, it is difficult to enforce the memento as an opaque abstraction.



Summary

- Architectural support for usability involves both allowing the user to take the initiative in circumstances such as cancelling a long running command, undoing a completed command, and aggregating data and commands.
- To predict user or system response, the system must keep a model of the user, the system, and the task.