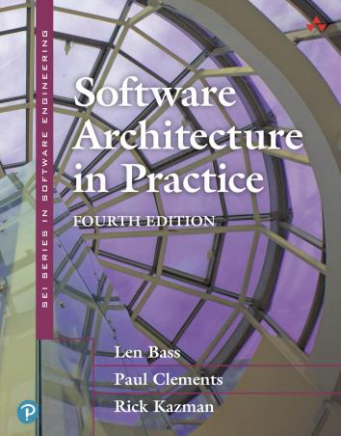
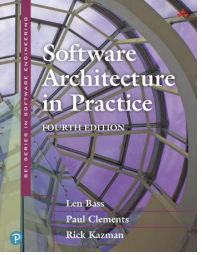


Chapter 12: Testability

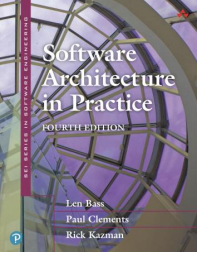
Testing leads to failure, and failure leads to understanding.

—Burt Rutan



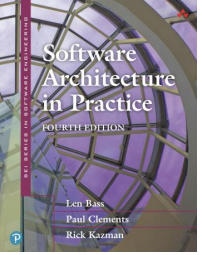
Chapter Outline

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



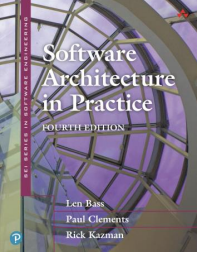
What is Testability?

- Software testability refers to the ease with which software can be made to demonstrate its faults through (typically execution-based) testing.
- Specifically, testability refers to the probability, assuming that the software has at least one fault, that it will fail on its next test execution.
- Intuitively, a system is testable if it “reveals” its faults easily.



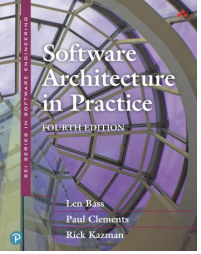
What is Testability?

- For a system to be properly testable, it must be possible to control each component's inputs (and possibly manipulate its internal state) and then to observe its outputs (and possibly its internal state, either after or on the way to computing the outputs).



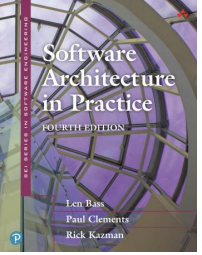
Testability General Scenario

Portion of Scenario	Description	Possible Values
Source	The test cases can be executed by a human or an automated test tool.	One or more of the following: ▪ Unit testers ▪ Integration testers ▪ System testers ▪ Acceptance testers ▪ End users Either run tests manually or use automated testing tools
Stimulus	A test or set of tests is initiated.	These tests serve to: ▪ Validate system functions ▪ Validate qualities ▪ Discover emerging threats to quality
Environment	Testing occurs at various events or life-cycle milestones.	The set of tests is executed due to: ▪ The completion of a coding increment such as a class, layer, or service ▪ The completed integration of a subsystem ▪ The complete implementation of the whole system ▪ The deployment of the system into a production environment ▪ The delivery of the system to a customer ▪ A testing schedule



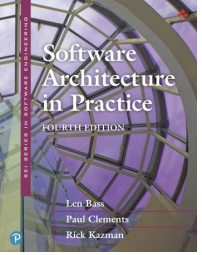
Testability General Scenario

Artifacts	The artifact is the portion of the system being tested and any required test infrastructure.	The portion being tested: ▪ A unit of code (corresponding to a module in the architecture) ▪ Components ▪ Services ▪ Subsystems ▪ The entire system ▪ The test infrastructure
Response	The system and its test infrastructure can be controlled to perform the desired tests, and the results from the test can be observed.	One or more of the following: ▪ Execute test suite and capture results ▪ Capture activity that resulted in the fault ▪ Control and monitor the state of the system
Response measure	Response measures are aimed at representing how easily a system under test "gives up" its faults or defects.	One or more of the following: ▪ Effort to find a fault or class of faults ▪ Effort to achieve a given percentage of state space coverage ▪ Probability of a fault being revealed by the next test ▪ Time to perform tests ▪ Effort to detect faults ▪ Length of time to prepare test infrastructure ▪ Effort required to bring the system into a specific state ▪ Reduction in risk exposure: size(loss) × probability(loss)

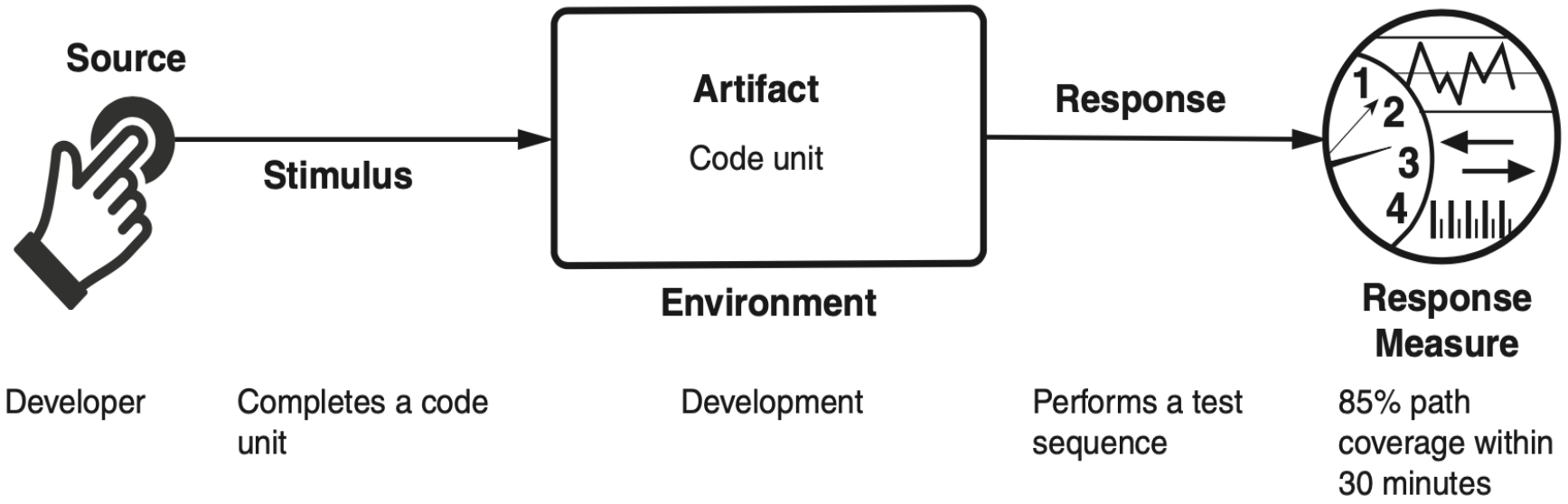


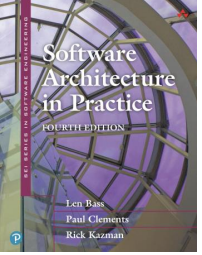
Sample Concrete Testability Scenario

- *The developer completes a code unit during development and performs a test sequence whose results are captured and that gives 85 percent path coverage within 30 minutes.*



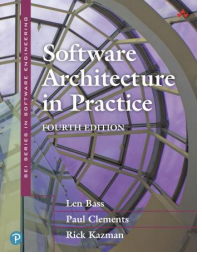
Sample Concrete Testability Scenario



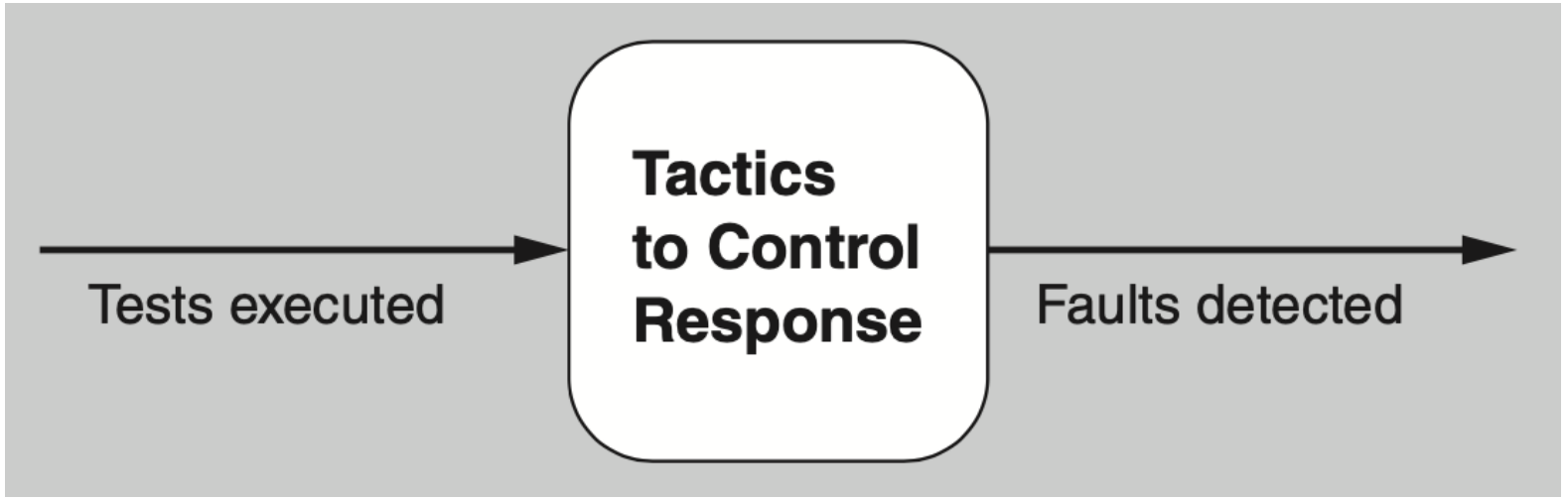


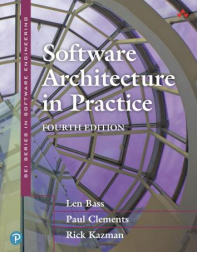
Goal of Testability Tactics

- Tactics for testability are intended to promote easier, more efficient, more capable testing.
- There are two categories of tactics for testability:
  - The first category deals with adding controllability and observability to the system.
  - The second deals with limiting complexity in the system's design.

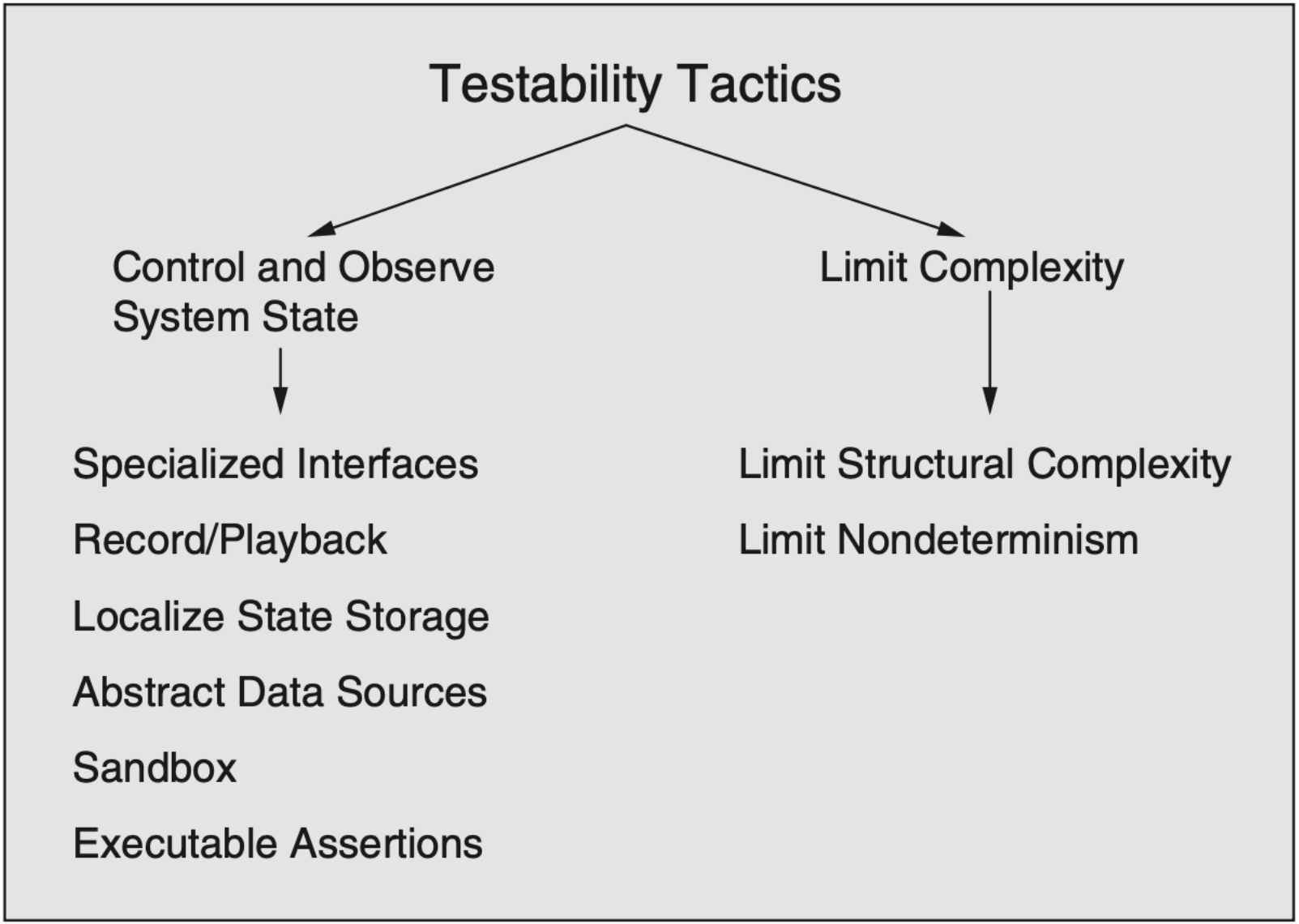


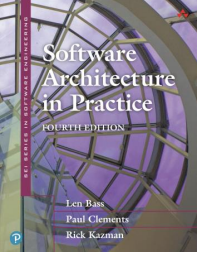
Goal of Testability Tactics





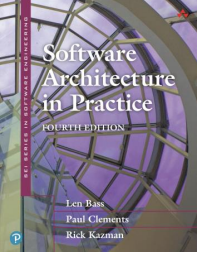
Testability Tactics





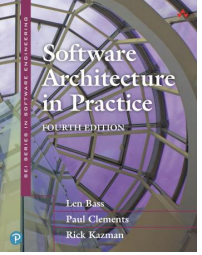
Control and Observe System State

- *Specialized interfaces.* Having specialized testing interfaces allows you to control or capture variable values for a component through the application of a test harness or through normal execution.
- *Record/playback*. The state that caused a fault is often difficult to re-create. Recording the state when it crosses an interface allows that state to be used to re-create the fault.
- *Localize state storage*. To start a component in an arbitrary state for a test, it is convenient if state is stored in a single place. By contrast, if the state is buried or distributed, this approach becomes difficult. The state can be fine-grained, even bit-level, or coarse-grained to represent broad abstractions or overall operational modes.



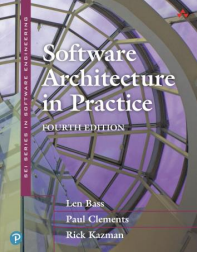
Control and Observe System State

- *Abstract data sources.* Similar to the case when controlling a program's state, the ability to control its input data makes it easier to test. Abstracting the interfaces lets you substitute test data more easily.
- *Sandbox*. "Sandboxing" refers to isolating an instance of the system from the real world to enable experimentation that is unconstrained by any worries about having to undo the consequences of the experiment. Testing is facilitated by the ability to operate the system in such a way that it has no permanent consequences, or so that any consequences can be rolled back.
- *Executable assertions*. With this tactic, assertions are (usually) hand-coded and placed at desired locations to indicate when and where a program is in a faulty state. The assertions are often designed to check that data values satisfy specified constraints.



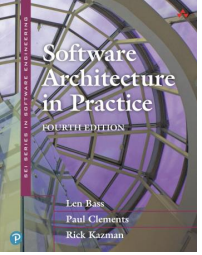
Limit Complexity

- *Limit structural complexity.* This tactic includes avoiding or resolving cyclic dependencies between components, isolating and encapsulating dependencies on the external environment, and reducing dependencies between components.
- *Limit nondeterminism.* The counterpart to limiting structural complexity is limiting behavioral complexity. This tactic involves finding all sources of nondeterminism, and weeding them out.



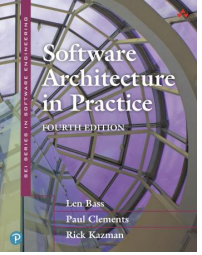
Tactics-Based Questionnaire for Testability

Tactics Group	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Control and Observe System State	Does your system have **specialized interfaces** for getting and setting values? Does your system have a **record/playback** mechanism? Is your system's **state storage localized**? Does your system **abstract its data sources**? Can some or all of your system operate in a **sandbox**? Is there a role for **executable assertions** in your system?				



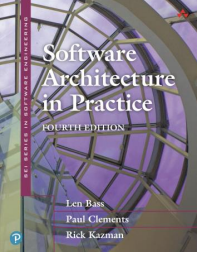
Tactics-Based Questionnaire for Testability

Limit Complexity	Does your system **limit structural complexity** in a systematic way? Is there nondeterminism in your system, and is there a way to control or **limit this nondeterminism**?



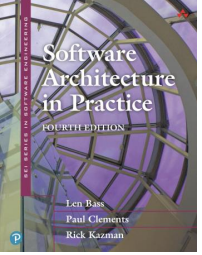
Dependency Injection Pattern for Testability

- In the dependency injection pattern, a client's dependencies are separated from its behavior.
- This pattern makes use of inversion of control. Unlike in traditional declarative programming, where control and dependencies reside explicitly in the code, inversion of control dependencies means that control and dependencies are provided from, and injected into the code, by some external source.



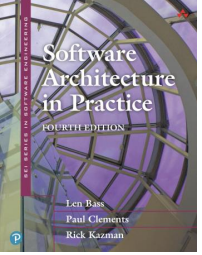
Dependency Injection Pattern Benefits

- Benefits:
  - Test instances can be injected, and these test instances can manage and monitor the state of the service.
  - Thus the client can be written with no knowledge of how it is to be tested.



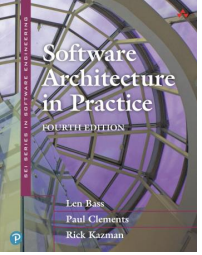
Dependency Injection Pattern Tradeoffs

- Tradeoffs:
  - Dependency injection makes runtime performance less predictable, because it might change the behavior being tested.
  - Adding this pattern adds a small amount of up-front complexity and may require retraining of developers to think in terms of inversion of control.



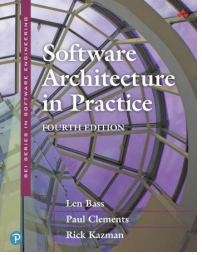
Strategy Pattern

- In the strategy pattern, a class's behavior can be changed at runtime.
- This pattern is often employed when multiple algorithms can be employed to perform a given task, and the specific algorithm to be used can be chosen dynamically.
- The class simply contains an abstract method for the desired functionality, with the concrete version of this method being selected based on contextual factors.



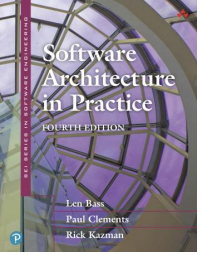
Strategy Pattern Benefits

- This pattern makes classes simpler, by not combining multiple concerns (such as different algorithms for the same function) into a single class.



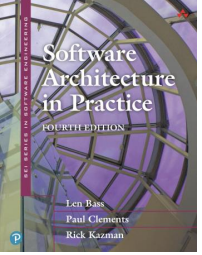
Strategy Pattern Tradeoffs

- The strategy pattern, like all design patterns, adds a small amount of up-front complexity. If the class is simple or if there are few runtime choices, this added complexity is likely wasted.
- For small classes, the strategy pattern can make code slightly less readable. However, as complexity grows, breaking up the class in this way can enhance readability.



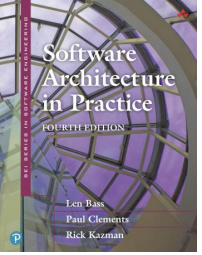
Intercepting Filter Pattern

- The intercepting filter pattern is used to inject pre- and post-processing to a request or a response between a client and a service.
- Any number of filters can be defined and applied, in an arbitrary order, to the request before passing the request to the eventual service.



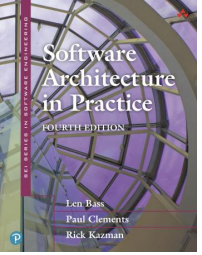
Intercepting Filter Pattern Benefits

- This pattern, like the strategy pattern, makes classes simpler, by not placing all of the pre- and post-processing logic in the class.
- Using an intercepting filter can be a strong motivator for reuse and can dramatically reduce the size of the code base.



Intercepting Filter Pattern Tradeoffs

- If a large amount of data is being passed to the service, this pattern can be highly inefficient and can add a nontrivial amount of latency, as each filter makes a complete pass over the entire input.



Summary

- Ensuring that a system is easily testable has payoffs both in terms of the cost of testing and the reliability of the system.
- Controlling and observing the system state is a major class of testability tactics.
- Complex systems are difficult to test because of their large state space, and because of the many interconnections among the elements. Consequently, limiting complexity is another class of tactics that supports testability.