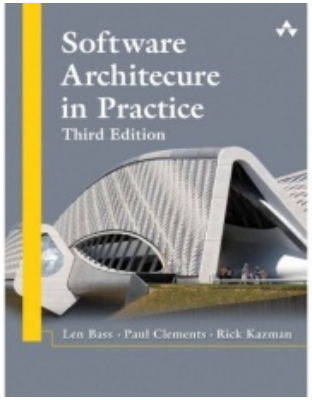
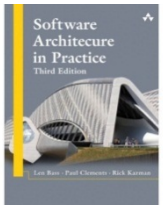
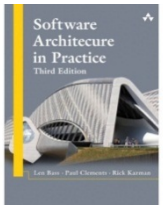


Chapter 10: Testability



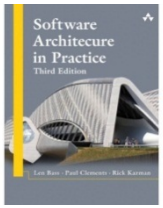
Chapter Outline

- What is Testability?
- Testability General Scenario
- Tactics for Testability
- A Design Checklist for Testability
- Summary



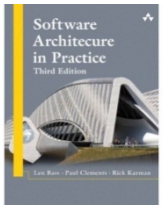
What is Testability?

- 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.
- If a fault is present in a system, then we want it to fail during testing as quickly as possible.



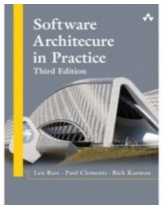
What is Testability?

- 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).



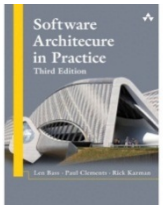
Testability General Scenario

Portion of Scenario	Possible Values
Source	Unit testers, integration testers, system testers, acceptance testers, end users, either running tests manually or using automated testing tools
Stimulus	A set of tests are 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 system; or the delivery of the system to the customer.
Environment	Design time, development time, compile time, integration time, deployment time, run time
Artifacts	The portion of the system being tested
Response	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	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 fault being revealed by the next test; time to perform tests; effort to detect faults; length of longest dependency chain in test; length of time to prepare test environment; reduction in risk exposure (size(loss) * prob(loss))



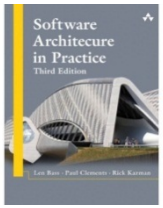
Sample Concrete Testability Scenario

- The unit tester completes a code unit during development and performs a test sequence whose results are captured and that gives 85% path coverage within 3 hours of testing.

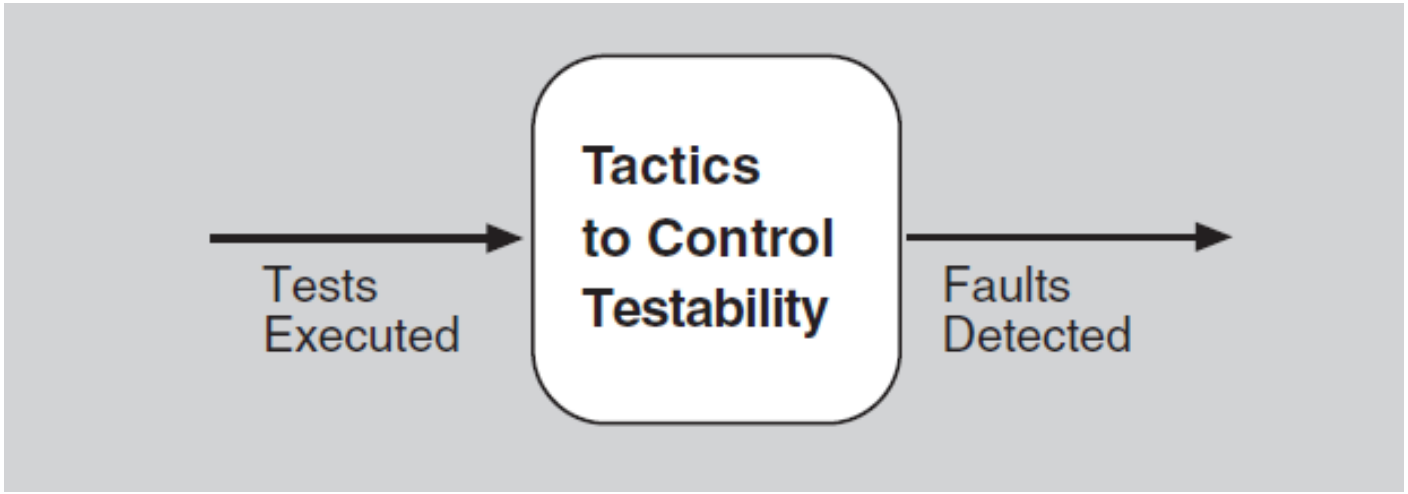


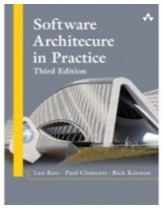
Goal of Testability Tactics

- The goal of tactics for testability is to allow for easier testing when an increment of software development has completed.
- Anything the architect can do to reduce the high cost of testing will yield a significant benefit.
- 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.

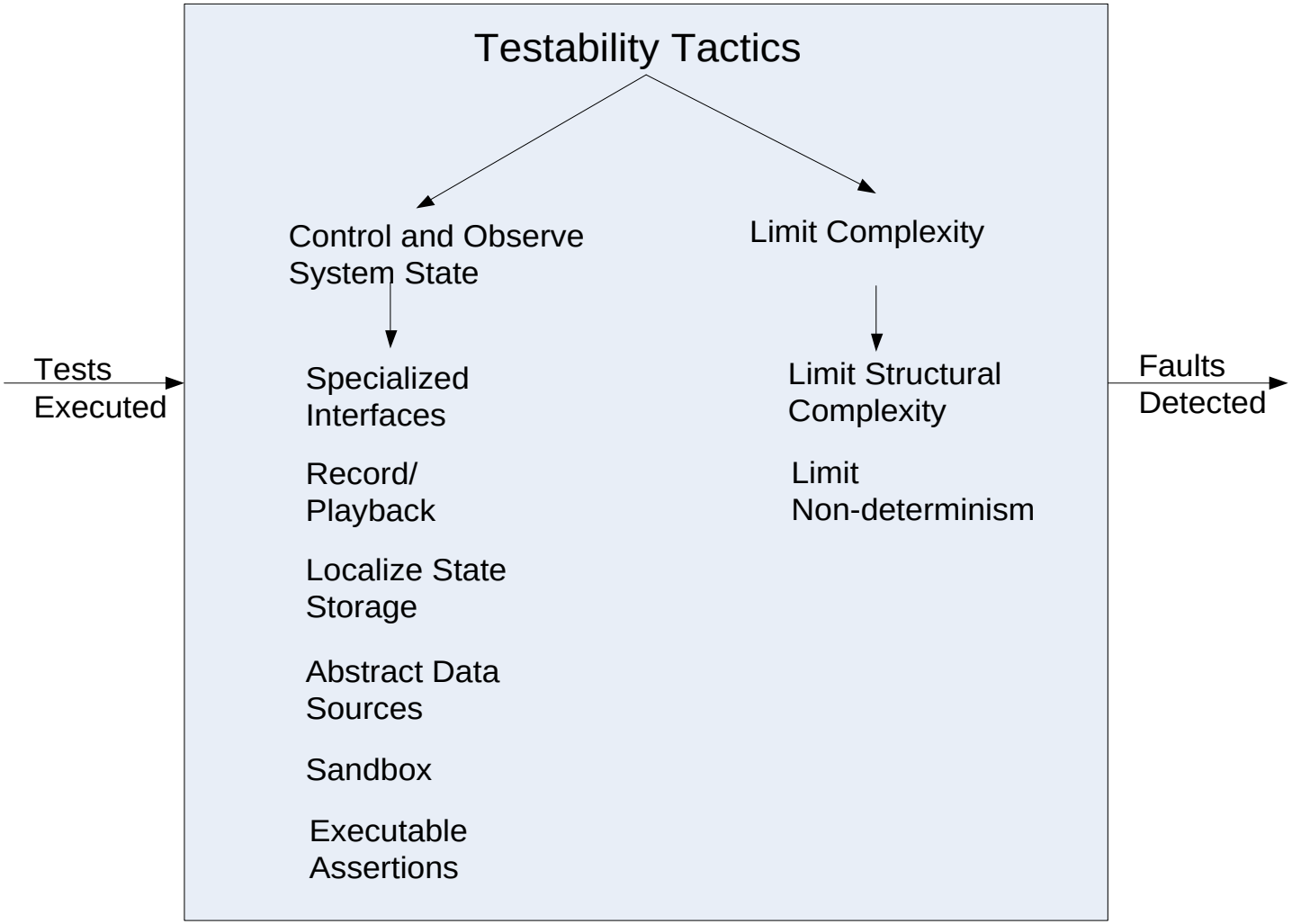


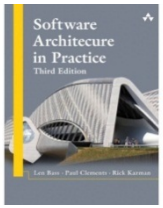
Goal of Testability Tactics





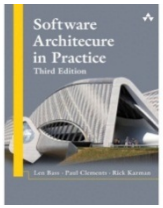
Testability Tactics





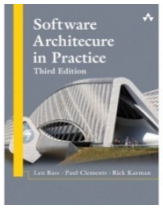
Control and Observe System State

- Specialized Interfaces: to control or capture variable values for a component either through a test harness or through normal execution.
- Record/Playback: capturing information crossing an interface and using it as input for further testing.
- Localize State Storage: To start a system, subsystem, or module in an arbitrary state for a test, it is most convenient if that state is stored in a single place.



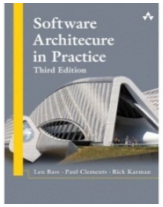
Control and Observe System State

- Abstract Data Sources: Abstracting the interfaces lets you substitute test data more easily.
- Sandbox: isolate the system from the real world to enable experimentation that is unconstrained by the worry about having to undo the consequences of the experiment.
- Executable Assertions: assertions are (usually) hand coded and placed at desired locations to indicate when and where a program is in a faulty state.



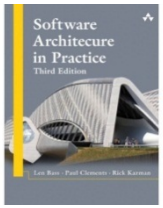
Limit Complexity

- Limit Structural Complexity: avoiding or resolving cyclic dependencies between components, isolating and encapsulating dependencies on the external environment, and reducing dependencies between components in general.
- Limit Non-determinism: finding all the sources of non-determinism, such as unconstrained parallelism, and weeding them out as far as possible.



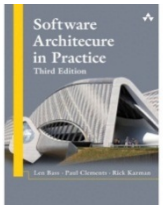
Exercise

- Write a quality attribute scenario on testability for POS
- Apply tactics to handle the scenario



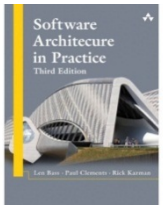
Design Checklist for Testability

Allocation of Responsibilities	**Determine which system responsibilities are most critical and hence need to be most thoroughly tested.** **Ensure that additional system responsibilities have been allocated to do the following:** • **execute test suite and capture results (external test or self-test)** • **capture (log) the activity that resulted in a fault or that resulted in unexpected (perhaps emergent) behavior that was not necessarily a fault** • **control and observe relevant system state for testing** **Make sure the allocation of functionality provides high cohesion, low coupling, strong separation of concerns, and low structural complexity.**



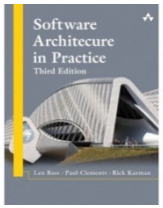
Design Checklist for Testability

Coordination Model	**Ensure the system's coordination and communication mechanisms:** • support the execution of a test suite and capture of the results within a system or between systems • support capturing activity that resulted in a fault within a system or between systems • support injection and monitoring of state into the communication channels for use in testing, within a system or between systems • do not introduce needless non-determinism



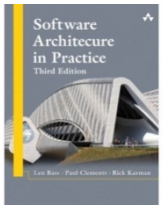
Design Checklist for Testability

Data Model	**Determine the major data abstractions that must be tested to ensure the correct operation of the system.** . **Ensure that it is possible to capture the values of instances of these data abstractions.** . **Ensure that the values of instances of these data abstractions can be set when state is injected into the system, so that system state leading to a fault may be re-created.** . **Ensure that the creation, initialization, persistence, manipulation, translation, and destruction of instances of these data abstractions can be exercised and captured**



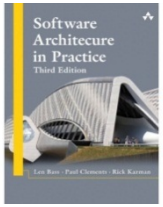
Design Checklist for Testability

Mapping Among Architectural Elements	**Determine how to test the possible mappings of architectural elements (especially mappings of processes to processors, threads to processes, modules to components) so that the desired test response is achieved and potential race conditions identified.** **In addition, determine whether it is possible to test for illegal mappings of architectural elements.**



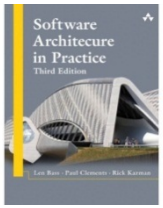
Design Checklist for Testability

Resource Management	**Ensure there are sufficient resources available to execute a test suite and capture the results.** **Ensure that your test environment is representative of (or better yet, identical to) the environment in which the system will run.** **Ensure that the system provides the means to:** **. test resource limits** **. capture detailed resource usage for analysis in the event of a failure** **. inject new resources limits into the system for the purposes of testing** **. provide virtualized resources for testing**



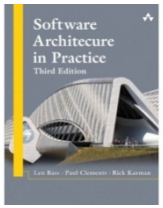
Design Checklist for Testability

Binding Time	**Ensure that components that are bound later than compile time can be tested in the late bound context.** **Ensure that late bindings can be captured in the event of a failure, so that you can re-create the system's state leading to the failure.** **Ensure that the full range of binding possibilities can be tested.**



Design Checklist for Testability

Choice of Technology	**Determine what technologies are available to help achieve the testability scenarios that apply to your architecture. Are technologies available to help with regression testing, fault injection, recording and playback, and so on?** **Determine how testable the technologies are that you have chosen (or are considering choosing in the future) and ensure that your chosen technologies support the level of testing appropriate for your system. For example, if your chosen technologies do not make it possible to inject state, it may be difficult to re-create fault scenarios.**



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 are a major class of testability tactics.
- Complex systems are difficult to test because of the large state space in which their computations take place, and because of the larger number of interconnections among the elements of the system. Consequently, keeping the system simple is another class of tactics that supports testability.