SAPM
Testability

Testability

- Testability is **not** testing.
- A system or element of a system is testable if it is possible to test it in the way required by a particular development or maintenance process.
- The concern of the software architect is that the architecture is structured in order to ensure desired testing can take place.

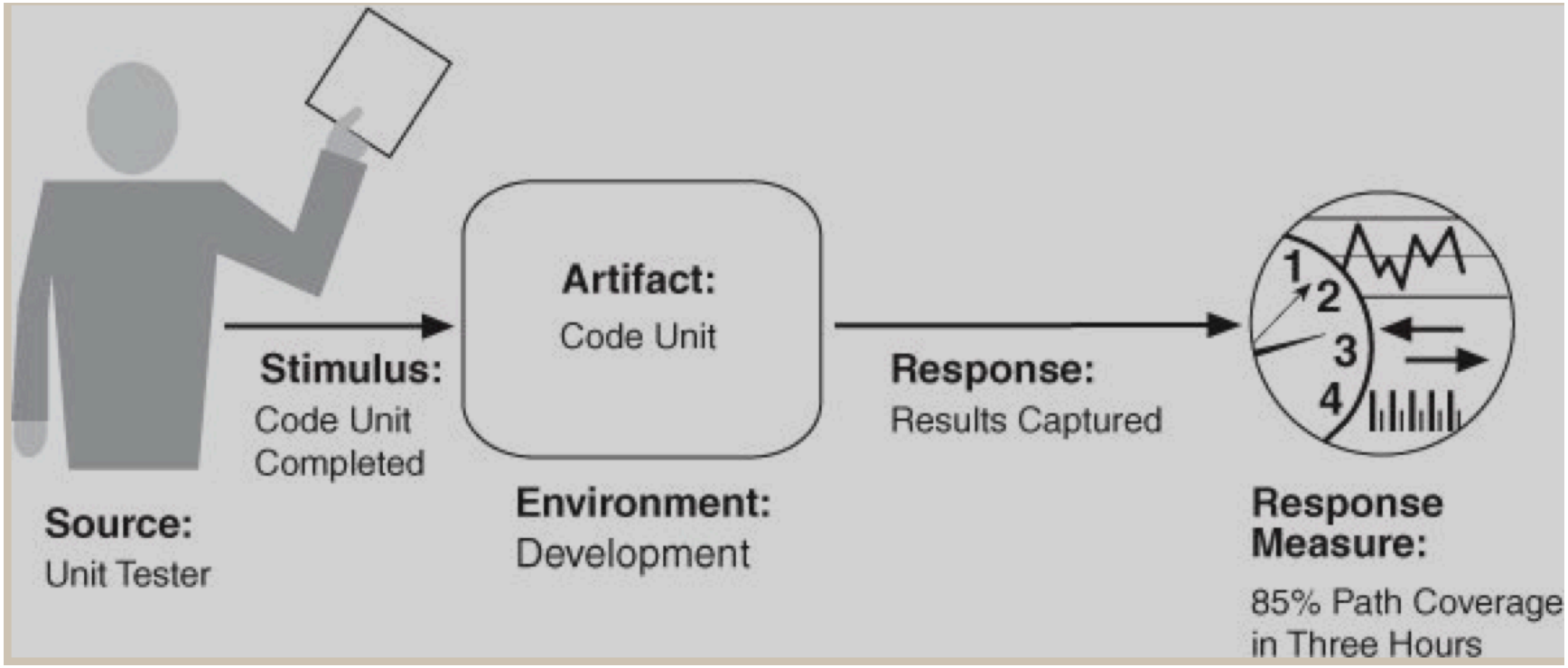
Testability Concerns

- Unlike the other examples we have considered (availability, performance and security) testability is concerned with the code structure rather than the connector/ component view or deployment view.
- The system elements we consider are code modules and the relationships are dependencies involved in building the code for components.

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 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, 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))

Concrete Scenario



Coverage Concrete Scenario

- **Source:** Regression Tester
- **Stimulus:** Completion of maintenance development to repair a critical bug
- **Artifact:** Modules for the full system
- **Environment:** Maintenance Development
- **Response:** Results from path coverage tool
- **Response Measure:** Path coverage is better than 95% of non-looping paths inside modules

Development Concrete Scenario

- **Source:** Developer/Tester
- **Stimulus:** Completion of new functionality to add transaction logging to a system.
- **Artifact:** Logging Modules and log database
- **Environment:** Development
- **Response:** Results from tests of simulated operation of the system.
- **Response Measure:** Consistency between log and the simulated data.

Testability Tactics

Coverage Scenario

- We might identify that there is one particularly large module with a very large number of loop-free paths that confounds the regression test.
- If it could be broken into smaller modules with lower complexity that could allow the regression test to achieve higher path coverage.
- This is an example of a tactic based on limiting complexity (or at least attempting to control it)

Development

- It may be that we cannot pass the scenario because we do not have adequate access to the log database from the tests.
- If we restructure the software architecture to introduce specific interfaces that provide better access to the database this may resolve the issue.
- This is an example where we are attempting to change what can be observed to it is possible to check if the system passes the test.

Checklist: Allocation of responsibilities

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.

Checklist: Coordination Model

Coordination Model	Ensure the system's coordination and communication mechanisms: ▪ Support the execution of a test suite and capture 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 nondeterminism

Checklist: Data Model

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

Checklist: Mapping among elements

Mapping among Architectural Elements	Determine how to test the possible mappings of architectural elements (especially mappings of processes to processors, threads to processes, and 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.

Checklist: Resource Management

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 do the following: ▪ Test resource limits ▪ Capture detailed resource usage for analysis in the event of a failure ▪ Inject new resource limits into the system for the purposes of testing ▪ Provide virtualized resources for testing

Checklist:Binding Time

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.

Checklist: Choice of Technology

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

- Testability becomes increasingly important in an environment where development is test driven.
- Software Architecture has a role to play in helping with the choice of static structure for the code of a system.
- The main ways we can improve testability is via changing the observability of phenomena or by limiting complexity these both have an impact on our ability to test.