Foundations of Software Testing For Developers

Robert T. Bauer

Course Objectives

- Background — why developers need to focus on testing
- “Extreme” automation
- Test Driven Development
- Reliability Engineering
- What makes for good tests
- How to know when to stop testing

Why focus on testing?

Epistemological Foundation — "Knowledge that . . ."

Can I know that service X will perform as intended when released into environment Y?

There are two ways to know:

- Proof
- Verification

Both are hard and generally can't be done.

Do I know that service X will perform as intended when released into environment Y? Generally, **No!**

Do I have "evidence" consistent with the belief that service X will perform as intended when released into environment Y?

Testing provides the "evidence."

What is a test? A fault? A defect?

A test consists of one or more test cases.

A test case is a formal description that has:

- A starting state
- One or more events to which the system must respond
- The expected response and/or ending state

Test data consists of a collection of starting states (e.g., the initial data in a database table) as well as the events used to test a function, module, subsystem, system or even systems.

A fault occurs when the observed behavior of the system is different from the expected behavior.

A defect is an error in the software (source code).

Faults imply defects (perhaps in the test case rather than the "code"); the converse is not true.

Background and Vocabulary

- Black box - exercise behavior and data specifications
  - Probabilistic Transition Network
  - Orthogonal array, pairwise, equivalence, boundary
- White box - exercise code paths
  - Code coverage (statement, branch, etc.); control and data flow
- Gray box - use either/both in test development
- Defect Detection
  - Testing to demonstrate system features behave as expected
  - Testing to demonstrate system doesn't do "wrong" stuff
- Defect Prevention: Design reviews, Fagan inspections, code analysis, early regression testing, TDD, STeP
- SDLC: Waterfall, Incremental, Spiral, Scrum, Agile
- Tests: Unit, Integration, System

EXTREME Automation

Dijkstra's top-down, step-wise refinement approach is test driven, correct-by-construction, contract-first and design-by-contract. The approach was published February, 1968.

Writing and testing small *chunks* of code lessens the chance that it will be hard and/or take a long time to find defects when a fault is observed.

EXTREME Automation means:

- Easy to run tests over and over again
- Easy to add more tests
- Easy to revert to "previous" working versions
- Easy to check-in dozens of times daily — each check-in *triggers* running automated tests
- Code "automatically" moves through testing platforms/stages/environments
- Continuous Integration & Continuous Deployment

Coding Big Picture

- Write tests for a “feature” before writing code for the feature.
- Always run all the tests - only the “new” tests should fail.
- Periodically stop and refactor code

Hardware Fails?

Assumption: Well-designed, well-engineered, thoroughly tested and properly maintained equipment should never fail in operation.

Reality: Best designed, best manufacturing and best maintenance efforts do not completely eliminate the occurrence of failures.

Practical: Stuff breaks and the more stuff you have, the more likely something is broke. If you have enough stuff, there's always something broke.

Reliability Concepts

Fault Tolerance: Service complying with specification despite faults having occurred or occurring.

Distinguish between:

- Reliable software that accomplishes its task under adverse conditions
- Robust software that indicates a failure correctly without taking down the whole system

Quality — Reliability — Fault Tolerance

Quality: Reliability, efficiency, security, maintainability and (adequate) size

Reliability: Probability of failure-free (software) operation for a specified period of time in a specified environment

Fault Tolerance: $\frac{m}{n}$ where m is the number of tolerable subsystem failures and n is the number of subsystems.

Customers don't differentiate between quality and reliability!

Engineering Software Reliability - Background

- Experiments conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements to determine the probability of failure-free (software) operation for a specified period of time in a specified environment.
- Can be done incrementally; for example, to ensure a new component
  - Meets its reliability objective
  - Doesn't impact other components
- Systems' architectures must not only consider reliability, it must also consider how reliability is measured, assessed and ultimately engineered into systems.

Engineering Software Reliability

- Requirements and Architecture
  - Define “necessary” reliability
  - Specify operational profiles
  - Test design
- Implementation
  - Tests
  - Operational Profiles
- Reliability and Acceptance
  - Execute tests
  - Interpret failure data
  - Create deficiency reports (bug)
  - Evaluate data for acceptance and predict release date

Test Specifications (I)

Traditional unit tests for services:

Component-based feature tests of "atomic" services exercise the functionality of the service to determine whether it satisfies its functional specification.

Integration feature tests of service assemblies exercise the functionality of services that aggregate and/or compose other services to determine whether the assembly satisfies its functional specification.

Failure testing of services and service assemblies induce a variety of failures (out-of-memory, process killed, external operation fails, etc.) to determine whether the service satisfies its functional specification.

Test Specifications (II)

Reliability focused unit tests for services:

Load testing of services and service assemblies determine the point at which the system begins to fail (not satisfy its performance and/or functional specifications). Reliability focused load testing ensures that the service (or system) response times meet the performance specification and that the services are functional under expected production volumes under defined production conditions including peak business conditions. Reliability focused load testing subsumes volume testing.

Test Specifications (III)

Reliability focused unit tests for services:

Stress testing of services and service assemblies is meant to ensure the robustness, availability and error handling under heavy load. We want to verify that the service/assembly doesn't crash, hang, etc. under conditions of insufficient resources, unusually high request rates, high concurrency, intermittent connections and environmental failures.

Stress testing is typically done in two phases. In phase stress tests are executed; in phase two stress and functional tests are concurrently executed.

Test Specification Example - Component Feature

Service X takes a mdn and returns a customer id.

Component feature tests in test plan:

- S-Normal-1: Valid mdn returns valid customer id
- S-Normal-2: Malformed mdn returns mdn error
  - How many ways to create a malformed mdn?
  - Is it wise to pick just one?
- S-Normal-3: Invalid mdn returns invalid mdn error

Do we have enough functional tests?

Test Specification Example - Integrated Feature

Service I takes a mdn and returns a customer object. It does this by "composing" Service X which takes a mdn and returns a customer id with Service Y that takes a customer id and returns a customer object.

Integrated feature tests include:

- S-Integrated-1: Valid MDN returns valid customer obj
- S-Integrated-2: Malformed MDN returns MDN error
- S-Integrated-3: Invalid MDN returns invalid MDN error

We aren't driving the "unit" tests for invalid/malformed customer id's; do you agree that we don't need to do this here?

Do you think we need S-Integrated-2 and S-Integrated-3 for the service assembly?

Do we have enough integrated tests?

Test Specification Example - Failure

Service I takes a mdn and returns a customer object. It does this by "composing" Service X which takes a mdn and returns a customer id with Service Y that takes a customer id and returns a customer object.

Failure tests include:

- S-Failure-1: return malformed business object
- S-Failure-2: Service Y times out
- S-Failure-3: Service Y unavailable
- S-Failure-4: Service X unavailable
- S-Failure-5: Service Y throws exception
- S-Failure-6: Service I times out

Do we have enough failure tests?

Test Specification Example - Performance

Why is performance considered to be a non-functional specification?

Let's take a look at a performance specification:

Service Y should be as quick as possible.

Test Specification Example - Performance

Service Y is called approximately 5 times per second. The average response time of the external service it queries takes 2.3 seconds. The external service's response is exponentially distributed with a minimum response of 100 msec, an SLA of 12 seconds and a form factor equal to 0.38.

The average payload returned to Service Y is less than 1 Kb. Payloads exceeding 10 Kb may occur.

The external service periodically sends throttle responses and the probability of it being unavailable is 0.001.

Service Y shall introduce a latency, excluding context switches, not to exceed 1 second.

Test Specification Example - Volume

Service Y that takes a customer id and returns a customer object.

Volume tests include:

- S-Vol-1: Return 1 Kb customer object
- S-Vol-1: Return 10 Kb customer object
- S-Vol-1: Return 100 Kb customer object
- S-Vol-1: Return 1 Mb customer object
- S-Vol-1: Return 10 Mb customer object
- S-Vol-1: Return 100 Mb customer object

Do we have enough volume tests? How many times do we execute them?

Test Specification Example - Performance

Service Y that takes a customer id and returns a customer object.

Performance tests include:

- S-Perf-1: Drive 10 transactions per second to service that returns 10 Kb response. Ensure this rate is sustained for 24 hours. Use external service with similar (or somewhat slower) response characteristics as actual external service.

Do we have enough performance tests?

Test Specification Example - Load

Service Y that takes a customer id and returns a customer object.

Load tests include:

- S-Load-1: Drive 10 transactions per second to service that returns 1 Kb response. Use external service with similar response characteristics as actual external service. Execute for 24 hours.
- S-Load-1: Drive 10 transactions per second to service that returns 100 Kb response. Use external service with similar response characteristics as actual external service. Execute for 24 hours.
- S-Load-1: Drive 10 transactions per second to service that returns 10 Mb response. Use external service with similar response characteristics as actual external service. Execute for 24 hours.

Do we have enough load tests?

Test Specification Example - Stress

Service Y that takes a customer id and returns a customer object.

Stress tests include:

- S-Stress-1: Drive 50 transactions per second to service that returns 100 Kb response. Ensure this rate is sustained for 24 hours. Use external service with similar (or somewhat slower) response characteristics as actual external service.
- S-Stress-2: Execute S-Stress-1 and simultaneously execute functional tests.
- S-Stress-2: Execute S-Stress-2 and simultaneously execute load and volume tests.

Do we have enough stress tests?

When to Stop Testing

When do we have enough tests?

- Code coverage
- Operational profile
- Critical Behavior Identified
  - Must work!
  - Affects goodwill - initial login takes too long
  - Cost of failure high - backup/restore
- Code Review

You must have confidence that the evidence supports your belief that the service will perform as intended when released into production.