Software Testing

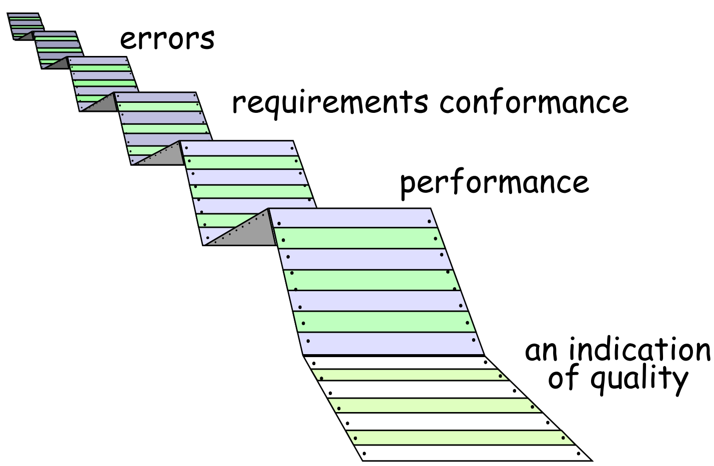
Overview

- What is software testing?
- General testing criteria
- Testing strategies
- OO testing strategies
- Debugging

Software Testing

- Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user.

What Does Testing Show?



Verification and Validation

- Verification refers to tasks to ensure that software correctly implements a specific function
  - "Are we building the product right?"
- Validation refers to tasks to ensure that the built software is traceable to customer requirements
  - "Are we building the right product"?

Who Tests the Software?

developer

Understands the system but, will test "gently" and, is driven by "delivery"

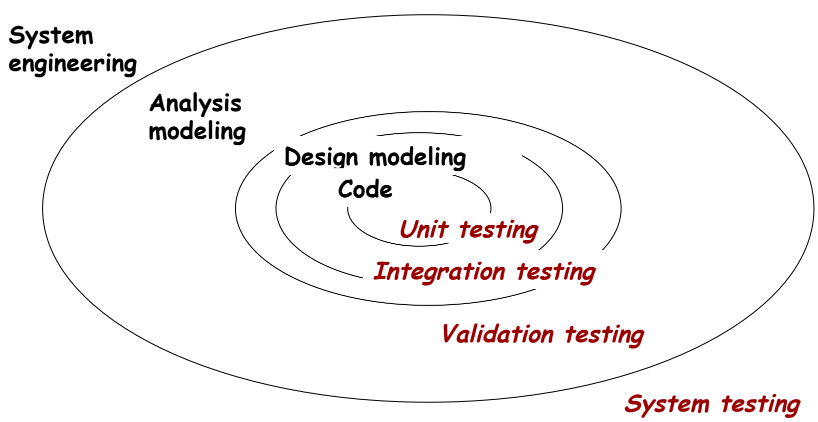
independent tester

Must learn about the system, but, will attempt to break it and, is driven by quality

General Testing Criteria

- Interface integrity
  - Communication and collaboration between components
- Functional validity
  - Algorithm implementation
- Information content
  - Local or global data
- Performance

Testing Strategies

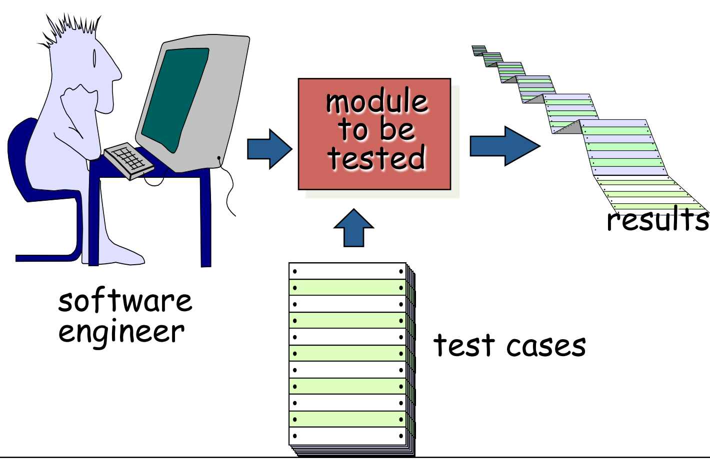


Testing Strategies

- We begin by **"testing-in-the-small"** and move toward **"testing-in-the-large"**
- For conventional software
  - The module is our initial focus
  - Integration of modules follows
- For OO Software
  - OO class is our initial focus
  - Integration of classes via communication and collaboration follows

Strategy 1: Unit Testing

- Verification on the smallest unit of software design



What Are Tested?

- Module interface
  - Information properly flows in and out
- Local data structures
  - Data stored temporarily maintains its integrity

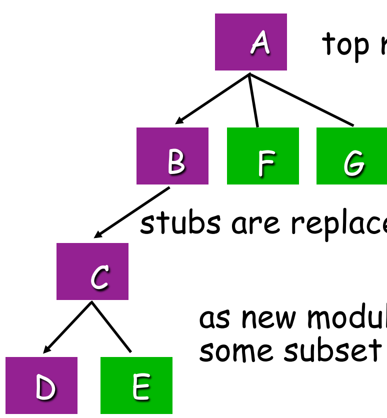
What Are Tested?

- Boundary conditions
  - The module operates properly at boundaries established to limit or restrict processing
- Independent paths
  - All statements in a module have been executed at least once
  - Including error-handling paths

Strategy 2: Integration Testing

- To construct tests to uncover errors associated with interfacing between units
- Two ways to integrate incrementally
  - Top-down integration
  - Bottom-up integration

Top-Down Integration

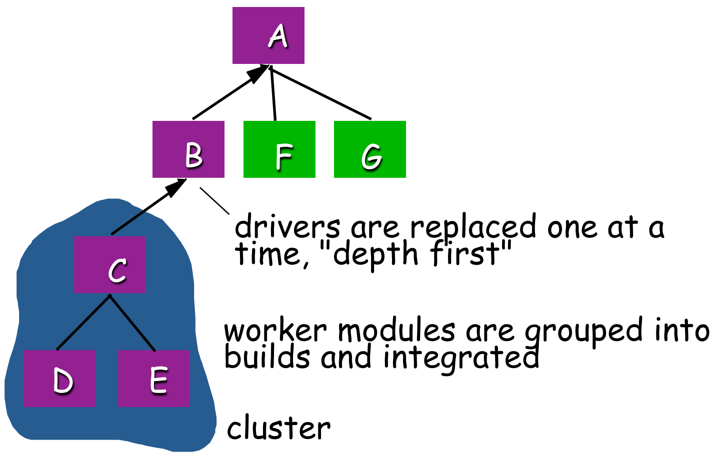


top module is tested with stubs

stubs are replaced one at a time, "depth first"

as new modules are integrated,
some subset of tests is re-run

Bottom-Up Integration



Regression Testing

- As a new module is added in integration testing, **regression testing** reruns some already executed tests to ensure that software changes do not cause problems in verified functions.
  - i.e, no unintended side effect is caused

Smoke Testing

- An integration testing approach **daily conducted** to test time-critical projects
  - "The smoke test should exercise the entire system from end to end. It does not have to be exhaustive, but it should be capable of exposing major problems. The smoke test should be thorough enough that if the build passes, you can assume that it is stable enough to be tested more thoroughly."

Smoke Testing

- Step 1: Software components already implemented are integrated into a "build"
  - A build includes all data files, libraries, reusable modules, and engineered components that are required to implement **one or more product functions**.

Smoke Testing

- Step 2: Tests are designed to expose errors that will keep the build from properly performing its function
  - The intent should be to uncover "show stopper" errors that have the highest likelihood of throwing the software project behind schedule.

Smoke Testing

- Step 3: The build is integrated with other builds, and the entire product is smoke tested daily.
  - The integration approach may be top down or bottom up

Why Smoke Testing?

- Integration risk is minimized
  - Uncover show-stoppers earlier
- The quality of end-product is improved
  - Early exposure of defects in design and implementation
- Error diagnosis and correction are simplified
  - New parts are probably buggy
- Progress is easier to assess

OO Testing Strategies

- Integration testing is mapped to
  - Thread-based testing
    - Integrates the classes required to respond to one input or event for the system
  - Use-based testing
    - integrates the classes required to respond to one use case
  - Cluster testing
    - integrates the classes required to demonstrate one collaboration

Validation Testing

- To check whether software functions as expected by customers
- To ensure that
  - All functional requirements are satisfied
  - All behavioral characteristics are achieved
  - All performance requirements are attained
  - Documentation is correct
  - Other requirements are met

Alpha and Beta Testing

- Alpha testing
  - Acceptance tests by a representative group of end users at the developer's site for weeks or months
- Beta testing
  - "Live" applications of the system in an environment with no developers' presence

Alpha Testing

- Why do we need alpha testing?
  - It is impossible for a developer to foresee how customers will really use a program
- How do people conduct alpha testing?
  - An informal "test drive" or a planned and systematically executed series of tests
  - Users use the software in a natural setting with the developers "looking over the shoulder"

Beta Testing

- Why do we need beta testing?
  - To uncover errors that only end users seem able to find
- How do people conduct beta testing?
  - The customers use the software at end-user sites
  - Customers record all problems that are encountered and report them to developers at regular intervals

System Testing

- A series of different tests to fully exercise the computer-based system
  - Recovery testing
  - Security testing
  - Stress testing
  - Performance testing
  - Deployment testing
- All tests verify that the system is successfully integrated to a larger system

Recovery Testing

- To force the software to fail in a variety of ways and verify that recovery is properly performed
  - Automatic recovery
    - Evaluate whether initialization, check pointing mechanisms, data recovery, and restart are correct
  - Manual recovery
    - Evaluate Mean-Time-To-Repair(MTTR) to determine whether it is acceptable

Security Testing

- To check whether the security protection mechanisms will actually protect the software from improper break through by:
  - acquiring passwords through externally
  - using hacking software
  - browsing/modifying sensitive data
  - intentionally causing system crash/errors

Security Testing

- Given enough time and resources, good security testing will ultimately penetrate a system
- The goal of the system designer is to make penetration cost higher than the value of the information that will be obtained

Stress Testing

- To execute a system by demanding resources in abnormal quantity, frequency, or volume
  - "How high can we crank this up before it fails ?"
    - Design tests that generate ten interrupts per second, when one or two is the average rate
    - Increase the input data rates by an order of magnitude to see how input functions will respond
    - Design tests that may cause excessive hunting for disk-resident data

Performance Testing

- To test the run-time performance of software within the context of an integrated system
- It is usually coupled with stress testing
- It requires both hardware and software instrumentation to
  - measure resource utilization, e.g., processor cycles
  - monitor execution states

Deployment (Configuration) Testing

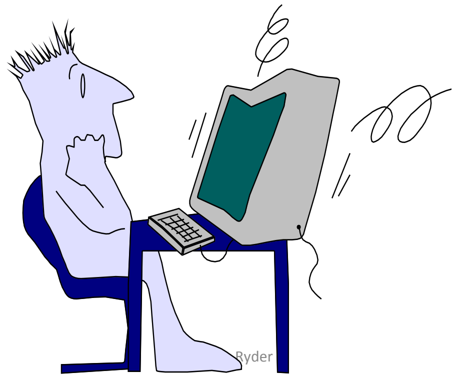
- To ensure the software works in all different operating systems that it is to operate
  - Execute the software in each environment
  - Examine all installation procedures, installer software, and user documentation

When Should We Stop Testing?

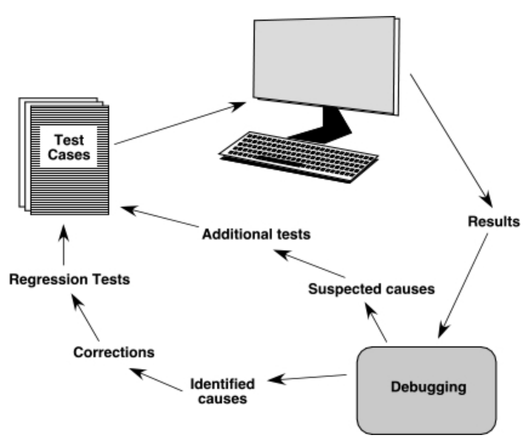
- When we detect some pre-defined number of errors
  - Use predictive models for estimation
- Examine number of errors found per unit of time
  - Decide whether to continue based on slope of graphs
- In reality -- WHEN YOU RUN OUT OF TIME

Debugging

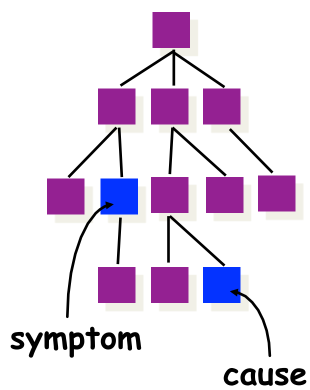
- When a test uncovers an error, debugging is the process to diagnose the root cause and further remove the error



The Debugging Process



Why Is Debugging So Difficult?



- symptom and cause may be geographically separated
- symptom may disappear when another problem is fixed
- cause may be due to a combination of non-errors
- cause may be due to a system or compiler error
- cause may be due to assumptions that everyone believes
- symptom may be intermittent

N. Meng, B. Ryder 37

The Art of Debugging

- Debugging tactics
  - Brute force/testing
  - Backtracking
  - Cause elimination

N. Meng, B. Ryder 38

Brute Force

- Record as much information as you can
  - Take memory dumps, collect runtime traces, print or log program states
- Pros
  - It can work when all other methods fail
- Cons
  - Waste effort and time
  - Too much information to be useful

Backtracking

- Beginning at the site where a symptom has been uncovered, the source code is traced backward (manually) until the cause is found
- Pros
  - Simple, good for small programs
- Cons
  - As the number of source lines increases, the number of potential backward paths may become unmanageably large

Cause Elimination

- Data related to the error occurrence are organized to isolate potential causes
  - A "cause hypothesis" is devised and the data is used to prove or disprove the hypothesis
  - A list of all possible causes is developed and tests are conducted to eliminate each