Lecture 15: Dynamic Analysis and Testing I

CS 5150, Spring 2025

Lecture goals

- Write reliable, maintainable tests of various styles, scopes, and sizes
- Leverage dynamic analysis tools to find bugs

Quality Assurance

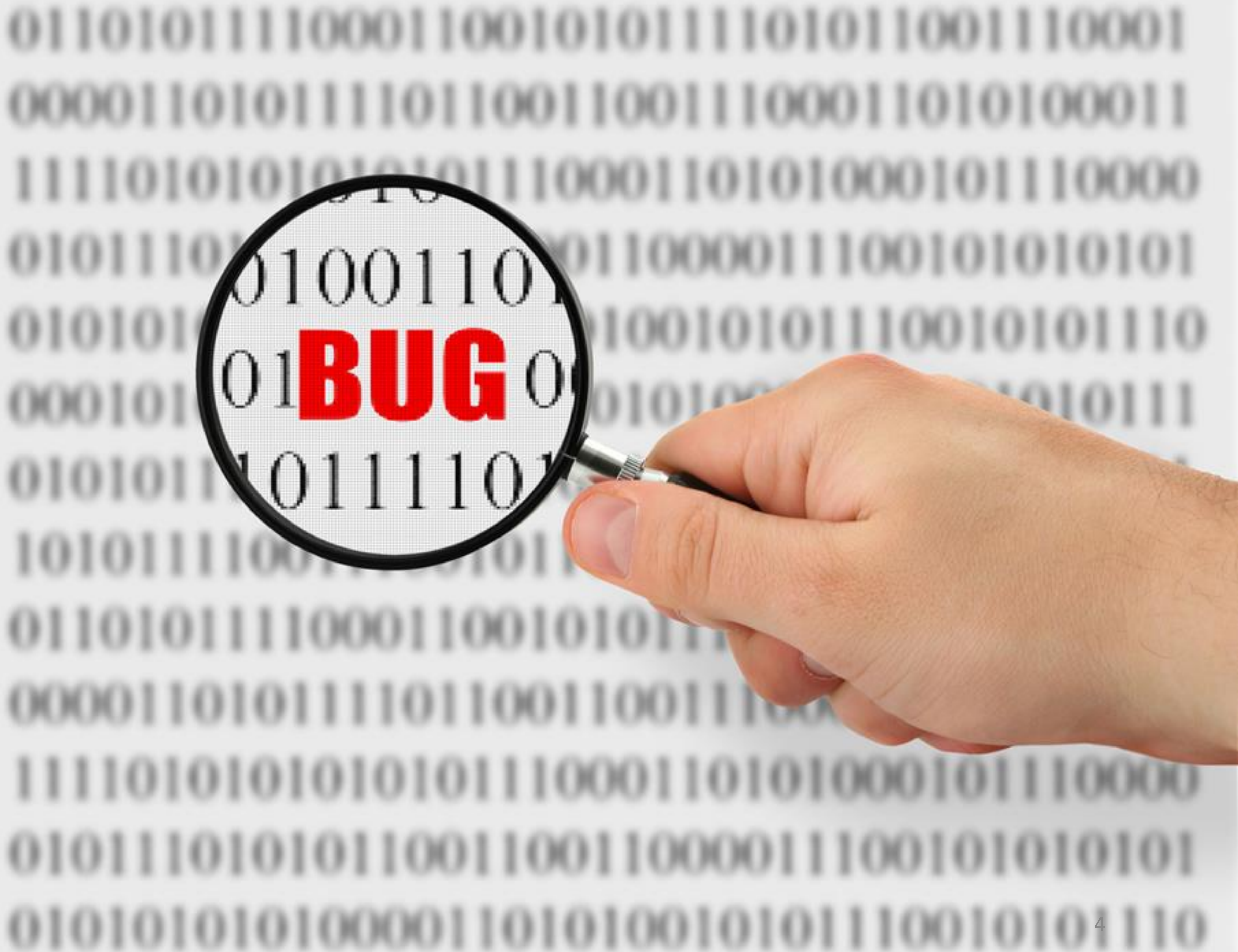
Internal Quality

- Is the code well structured?
- Is the code understandable?
- How well documented is it?

External Quality

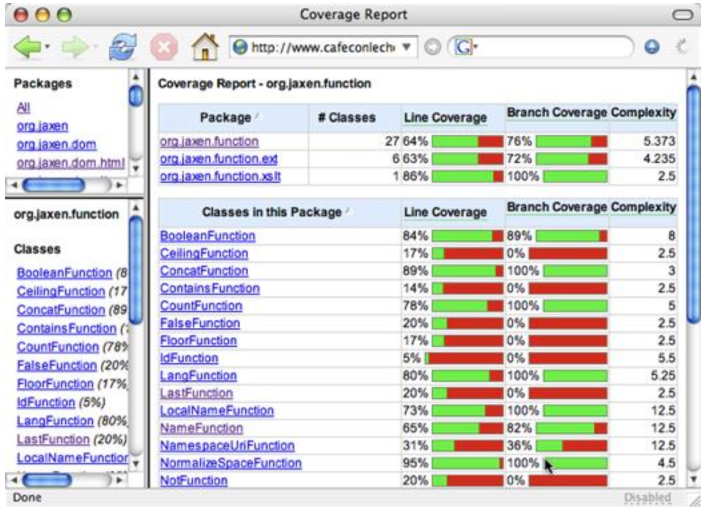
- Does the software crash?
- Does it meet the requirements?
- Is the UI well designed?

Testing



Testing: Basic concepts

- **Test case** (or, simply **test**): an execution of a program with a given test input, including:
  - Input values
  - Sometimes include execution steps
  - Expected outputs (**test oracle**)
- **Test suite**: a finite set of tests
  - Typically run in a sequence
- **Test adequacy**: a measurement to evaluate the test quality
  - Such as code coverage



Testing: Basic concepts

- **Fault:** Specific location(s) in code that is defective/incorrect (static)
- **Error:** An incorrect program state that is triggered when faulty code is executed
- **Failure:** observed behavior != expected behavior
  - Crash, Incorrect result, bad performance, ….
- **Bug:** Commonly used to refer to any of the above
- Other terms: defect

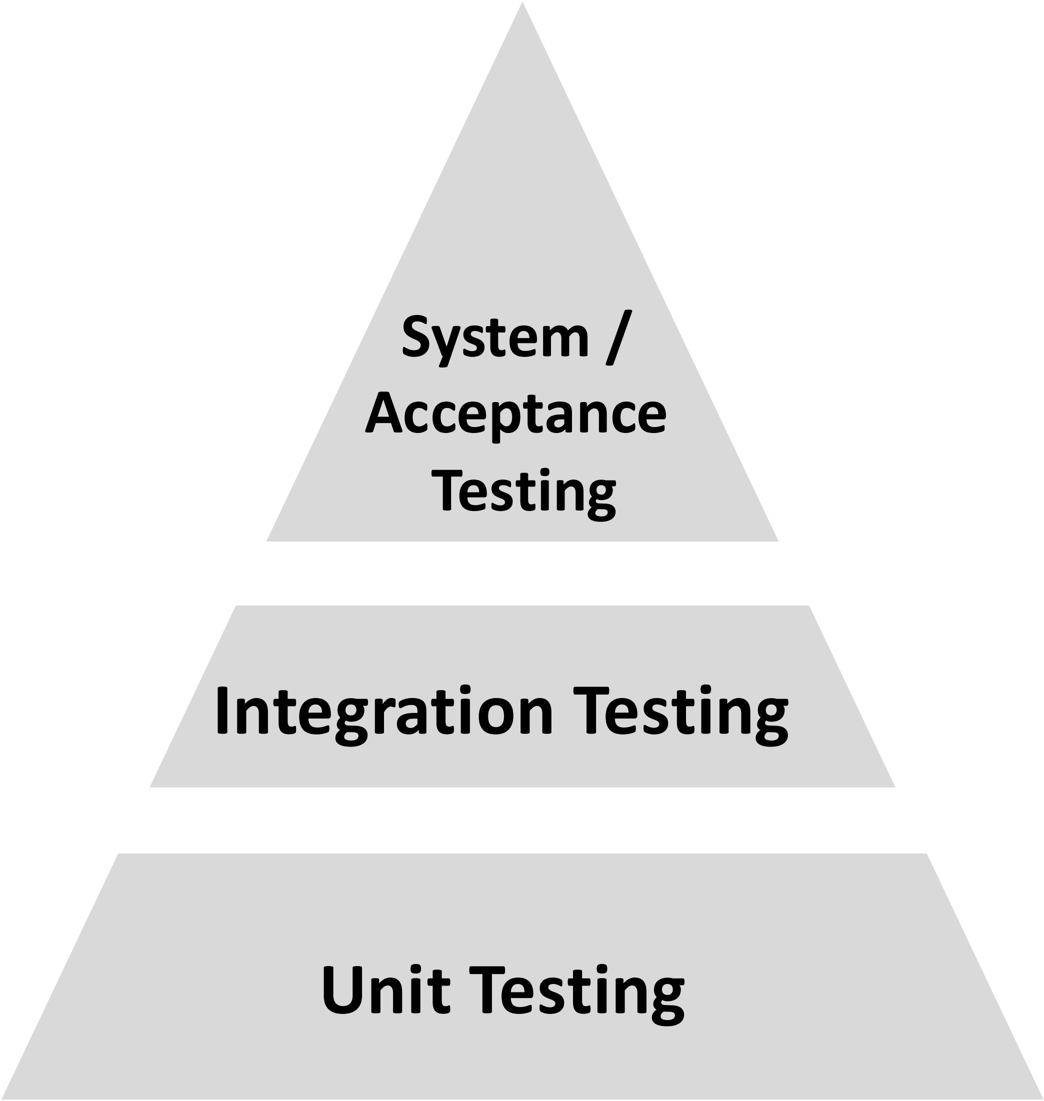
Testing: Basic concepts

- **Testing**: Attempt to trigger failures

- **Debugging**: Attempt to locate faults given a failure

Testing: Levels

- **Unit Testing**
  - Test each single module in isolation
- **Integration Testing**
  - Test the interaction between modules
- **System Testing**
  - Test the system as a whole, by developers
- **Acceptance Testing**
  - Validate the system against user requirements, by customers with no formal test cases



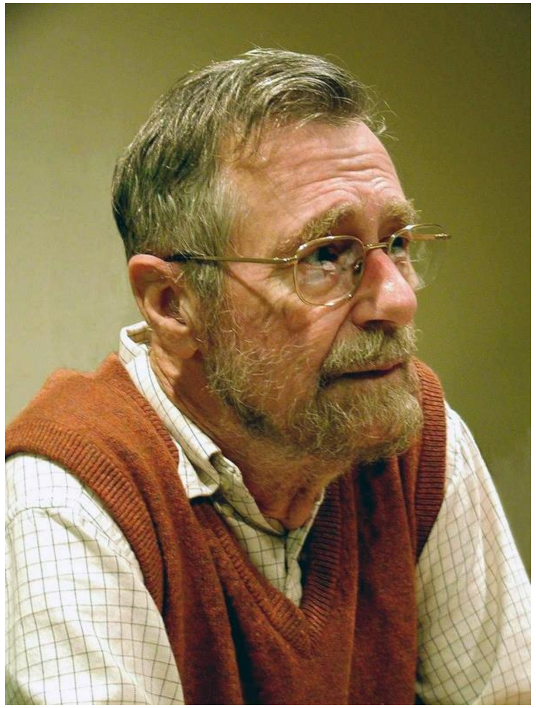
Goals of testing

- Find and prevent bugs
- Improve maintainability (esp. refactoring)
- Clarify intended usage

- To meet these goals, tests themselves should be:
  - Bug-free
  - Maintainable
  - Clearly documented and easy to read
  - **Rigorous**

Principles of Testing #1: Avoid the absence of defects fallacy

- Testing shows the presence of defects
- Testing does not show the absence of defects!
- “no test team can achieve 100% defect detection effectiveness”



Principles of Testing #2:
Exhaustive testing is impossible!

- Consider this simple function:
  - `def is_valid_email(email: str) -> bool:`
  - `...`

- 1 input string, max length: 320, 26 characters + 5 symbols …
  - Inputs to check: 320^31
  - Might take you millions of years …

Principles of Testing #3: Start testing early

- To let tests guide design
- To get feedback as early as possible
- To find bugs when they are cheapest to fix
- To find bugs when they have caused least damage

Principles of Testing #4: Defects are usually clustered

- “**Hot**” components requiring frequent change, bad habits, poor developers, tricky logic, business uncertainty, innovative, size, …
- Use as heuristic to focus test effort

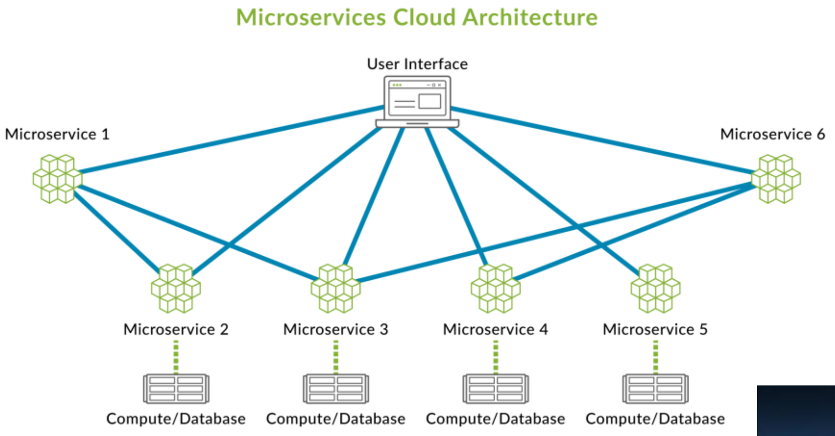
Principles of Testing #5: The pesticide paradox

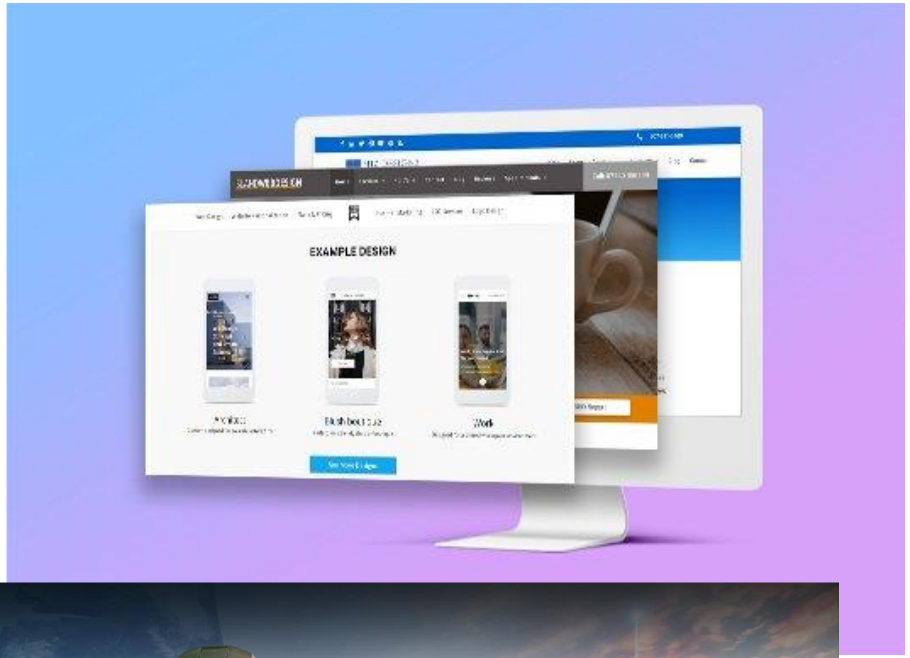
"Every method you use to prevent of find bugs leaves a residue of subtler bugs against which those methods are ineffectual"

- Re-running the same test suite again and again on a changing program gives a false sense of security

- Testing must **evolve** with software!

Principles of Testing #6:
Testing is context-dependent





Principles of Testing #7: Verification is not validation

- Verification:
  - Does the software system meet the requirements specifications?
  - Are we building the **software right**?

- Validation
  - Does the software system meet the user’s real needs?
  - Are we building the **right software**?

Test coverage

- Ways to measure "how much code" was tested
  - Function coverage
  - Statement (line) coverage
  - Branch coverage
  - Condition/decision coverage
  - Loop coverage
  - Path coverage
  - …
- Coverage analysis can reveal gaps in testing

- Example:
  `if (a>b && c!=25) { d++; }`

- Required cases for condition/decision coverage:
  - `a<=b`
  - `a>b && c==25`
  - `a>b && c!=25`

Poll: PollEv.com/cs5150sp25

```
double[] boxFilter(double[] x) {
  var y = new double[x.length];
  for (int i = 0; i < x.length; ++i) {
    var xl = x[i];  var xr = x[i];
    if (i > 0) { xl = x[i-1]; }
    if (i < x.length-1) { xr = x[i+1]; }
    y[i] = (xl + x[i] + xr)/3.0;
  }
  return y;
}
```

Coverage targets

- *Any statement not covered by a test is code you expect your client/users to run before you do*

  - By this philosophy, 100% line coverage would be a minimum target
    - But chasing coverage metrics with low-quality tests can be self-defeating
  - Tests take time to write, review, and run; must consider cost/benefit ratio

Activity: Brainstorm difficult testing scenarios

Difficult testing scenarios

- Error codes & exceptions from library and system calls
  - Out of memory
  - Out of disk space
  - Incomplete I/O
  - Transient I/O error (EAGAIN)
  - Timeouts
- Unbounded blocking
- Crash/power loss
  - Corrupted data
- Malicious intent
- Concurrency
  - High lock contention
  - Race conditions
  - Caching & memory ordering
  - True concurrency vs. multitasking
- Portability
  - Unsupported capabilities
  - Platform differences
- Performance
  - NUMA Non-Uniform Memory Access
  - Big.LITTLE
  - Disk I/O (bandwidth, latency)
  - Network I/O (bandwidth, latency)

Beyoncé rule

- *"If you liked it, then you shoulda put a test on it"*

- Manages responsibility during large-scale refactoring
  - *Infrastructure team* must ensure all tests pass before committing
  - If functionality breaks, *product team* must fix it (and add more tests)

- Aim for sufficient coverage so that *you* (and your teammates) would be okay being held responsible for a production breakage in uncovered code

Example: SQLite

- 640x more test code than application code
- 100% branch test coverage
- OOM, I/O errors, crashes
  - Use abstractions to wrap malloc, I/O operations
- Boundary values
- Regression tests
- Valgrind: memory debugging, memory leak detection, and profiling.
- Fuzz testing

- https://www.sqlite.org/testing.html

Kinds of testing

- Styles
  - Exploratory
  - Smoke tests
  - Black box
  - Glass box
  - Fuzz testing
  - Dynamic analysis

  (Black box, Glass box, Fuzz testing: Can synthesize with boundary value analysis, coverage feedback)
- Scopes
  - Unit tests
  - Integration tests
  - End-to-end tests
- Sizes
  - Small: fast, deterministic (in-process)
  - Medium: multi-process, allow blocking calls (single machine)
  - Large: Multi-node
- Purpose
  - Prevent reoccurrence of bugs (regression tests)
  - Prepare for release (acceptance tests, beta testing)
  - Ensure operating health (self tests)

Example: Aerospace testing

- Unit tests
  - Ensure thorough coverage
  - Verify independent implementations
- Smoke tests
  - Small-scale integration test
  - Ensure configs are valid
- Regression tests
  - Catch any change to behavior (ensure refactoring changes are non-functional)
  - Ensure control algorithms achieve mission objectives
- Checkpoint/restore tests
- Exploratory tests
  - Logged data posted to reviews
- Software-in-the-loop
  - Medium-scale integration test
  - Leverage virtualization, preloading, hardware simulation
  - Subsystem and end-to-end scope
- Hardware-in-the-loop
  - Large-scale integration test
  - Verify non-functional requirements
- Vehicle-in-the-loop
  - Large-scale integration test
  - Verify a particular "production unit"
- Formal test deliverables

Flaky vs. brittle tests

Flaky

- Non-deterministic failures
  - Multi-process/multi-node infrastructure failures
  - Timeouts
  - Randomness
    - Always log seed
  - Concurrency
    - Difficult to reproduce

Brittle

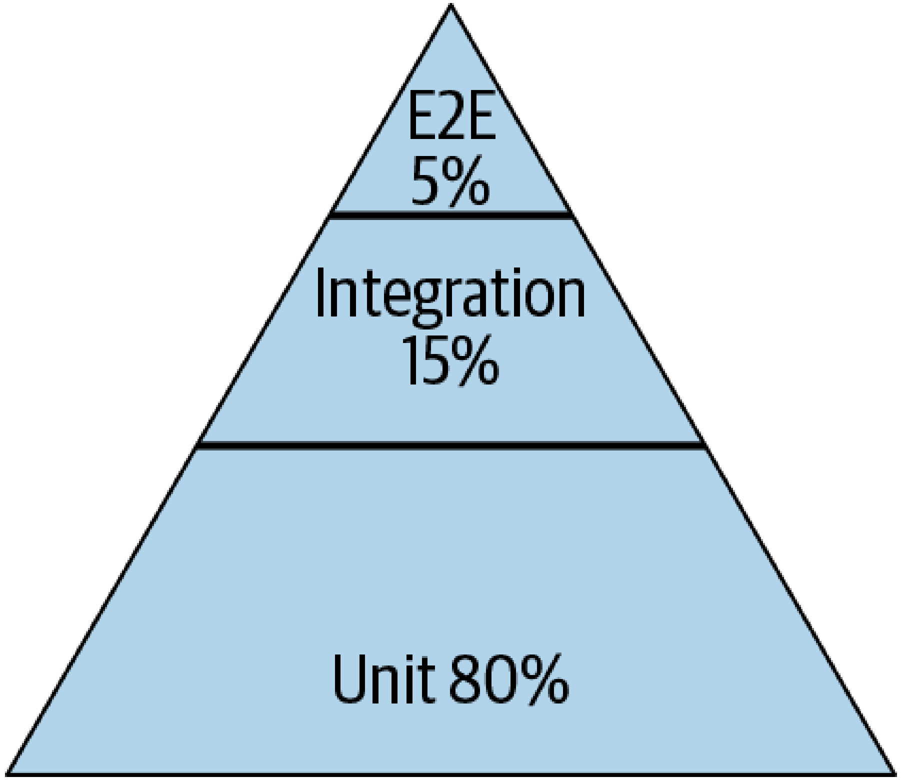
- "High maintenance"
  - Leverage private functionality
  - Depend on private state
  - Assume behavior beyond the spec
    - e.g., checking interactions instead of state

Aside: random numbers

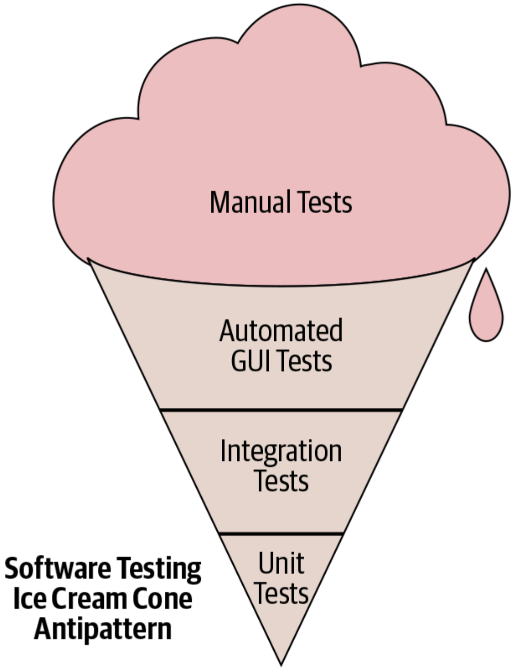
- In most settings, random numbers should be *deterministic*
  - Enables reproducibility, reduces test flakiness
  - Exceptions (in production): cryptography, gambling
- Recommended approach
  - Application starts with a specified global seed (and logs it)
  - Each component constructs a private RNG by combining global seed with unique instance name
  - Alternative for parallel computation: sequence queries, use RNG that can "fast forward" state
- Advantages
  - Results independent of amount of parallelism
  - Results do not change if "peripheral" components are added or removed

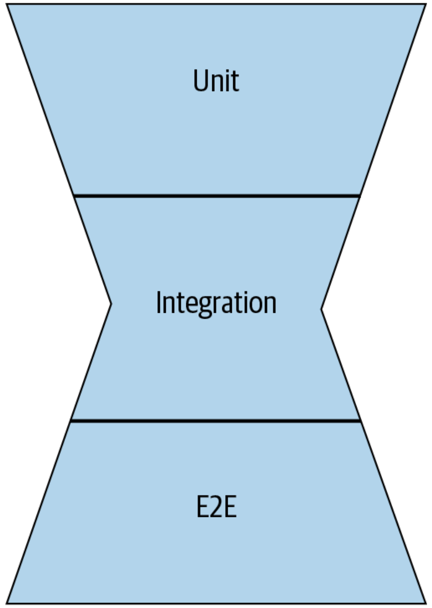
Test scope

Good



Bad





From *Software Engineering at Google*, Ch. 11

Test scope

Small scope

- Limited coverage (per test)
  - But coverage is orthogonal
- May require awkward setup (dependency injection, mock objects)
- Can be written simultaneously with the code-under-test
- Easy to diagnose
  - Limited amount of code is executed
  - Easier to understand procedure and results
- Typically faster
  - Can run more often

Large scope

- Extensive coverage (per test)
  - Much coverage is redundant
  - Most results are not checked (false sense of security)
- May be easier to set up than mid-scoped tests
  - But total configuration harder to reason about
- Depends on whole system
  - Bugs may not be found until later
- Difficult to diagnose
  - Slows down debugging when bugs are found
- Typically slower

Exploratory testing

- Applications
  - Developers check how existing code behaves
  - Developers "gut check" new code
  - Demonstrate functionality in a scenario of interest with complicated setup
  - QA testing (test behaviors developers often overlook)
- Tools
  - Application itself
  - REPL (JShell, iPython)
  - Dynamic analysis tools (valgrind, callgrind)
- Drawbacks
  - Not reproducible
    - Results may depend on unique context
    - Good habit to log all interactions
  - Good to think about expectations before running test, but if you can express what you expect, just write a unit test
  - Quality varies with tester
    - Can't measure coverage
- Other tools: **Selenium** for browsers

Unit tests

- Narrow scope (typically a single function or a single class)
- Focus on publicly-visible, fully-specified behavior
  - Check state, not process
- Write for clarity
  - Okay to be repetitive
  - Avoid new abstractions or logic

- Bad example:
  - When registering a new user, the system first generates a password, then tries to insert a new auth table row, throwing an exception if insertion failed (name already taken)
- Better example:
  - After registering a new user whose name is not taken, a new row will exist in the database with their username and password
  - If attempting to register a new user whose name is already taken, an exception is thrown

Behavior-driven development (BDD)

- Structuring tests around methods can make them brittle, hard to read
  - Try to test too many behaviors at once
- Better to structure tests around scenarios
- **Arrange-act-assert** format
  - "Given …, when …, then …"
  - Analogous to User Stories preamble

- "Given two accounts, the first of which has at least $100, when transferring $100 from the first to the second account, then both account balances should reflect the transfer"
- Test frameworks can help make tests self-documenting

BDD example

```
"A Stack" should "pop values in last-in-first-out order" in {
  val stack = new Stack[Int]
  stack.push(1)
  stack.push(2)
  stack.pop() should be (2)
  stack.pop() should be (1)
}

it should "throw NoSuchElementException if an empty stack is popped" in {
  val emptyStack = new Stack[Int]
  a [NoSuchElementException] should be thrownBy {
    emptyStack.pop()
  }
}
```

BDD example output

```
A Stack
- should pop values in last-in-first-out order
- should throw NoSuchElementException if an empty stack
  is popped
Run completed in 76 milliseconds.
Total number of tests run: 2
Suites: completed 1, aborted 0
Tests: succeeded 2, failed 0, canceled 0, ignored 0,
  pending 0
All tests passed.
```

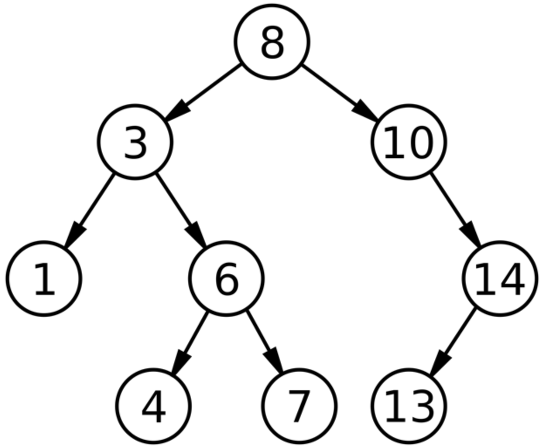
BDD example 2

```
info("As a TV set owner")
info("I want to be able to turn the TV on and off")
info("So I can watch TV when I want")
info("And save energy when I'm not watching TV")

Feature("TV power button") {
  Scenario("User presses power button when TV is off") {
    Given("a TV set that is switched off")
    val tv = new TVSet
    assert(!tv.isOn)
    When("the power button is pressed")
    tv.pressPowerButton()
    Then("the TV should switch on")
    assert(tv.isOn)
  }
  Scenario("User presses power button when TV is on") {
    Given("a TV set that is switched on")
    val tv = new TVSet
    tv.pressPowerButton()
    assert(tv.isOn)
    When("the power button is pressed")
    tv.pressPowerButton()
    Then("the TV should switch off")
    assert(!tv.isOn)
  }
}
```

Activity: Design tests using BDD

```
class BinarySearchTree {
private Node root; // root node
private int size; // number of nodes in the tree
static class Node {
private Node left; // left child
private Node right; // right child
}
public BinarySearchTree insert(int N);
public BinarySearchTree delete(int N);
public BinarySearchTree search(int N);
public BinarySearchTree succ(int N);
public BinarySearchTree pred(int N);
public int getSize();
```



Task: What kind of tests would you add?

Test doubles

- How to write unit-scoped tests with complex dependencies?
  - Using external services makes tests "larger"
    - Depending on specialty hardware is very constraining
  - Can be difficult to get complex objects into appropriate state
  - Can be difficult to trigger a corner-case response (e.g. I/O errors)
- Examples of external dependencies?

- Options
  - Use real dependencies anyway (highest fidelity and coverage)
  - Use fakes & simulators (good option; requires investment)
  - Use stubbing/mocks (convenient, but dangerous)
    - Beware temptation of interaction testing
- Design for testing
  - Dependency injection: *pass in* dependencies instead of using Singletons or constructing your own

Stubbing and mocking frameworks

- Create subclasses of dependencies whose methods return values specified by the test
  - Frameworks like **Mockito** make this easy, even with static types
- Enables interaction testing
  - Checking whether code-under-test calls methods on dependencies in the way we expect

Example:

var userAuth = new UserAuthorizer(
  mockPermissionDb);

when(**mockPermissionDb**.getPermission(
  user1, ACCESS)).thenReturn(EMPTY);

userAuth.grantPermission(ACCESS);

verify(**mockPermissionDb**).addPermission(
  user1, ACCESS);

Dangers of stubbing & interaction testing

- Increases brittleness
  - When refactoring the real dependency, must also change everyone's stubs
- Reduced fidelity
- Decreases clarity
  - Pollutes tests for one class with a different class's API
- Depends on implementation details rather than on observable state
  - May be appropriate to test for "side effects"

Integration tests

- Broader scope
  - Check that multiple components interface correctly
  - Check behavior of subsystems
- Tend to be larger in size
  - SoA requires multiple processes
  - Non-trivial data, config can be slow
  - Aim for smallest test possible
    - Split pipelines into pairwise interactions
- Larger tests require non-trivial infrastructure, can be flaky
  - Fakes
  - Lightweight substitutions
    - In-memory databases
  - Hermetic services
    - Leverage virtualization to deploy isolated instances of service dependencies
  - Record/replay I/O
    - Trades flakiness for brittleness

Integration environments

- Production
  - Highest fidelity, esp. for load
  - Failures affect real users
  - Canarying: deploy to subset of production systems
    - E.g., internal users, early access
    - Can lead to version skew – incompatibility between concurrently-running components
  - Feature flags: Allow operators to quickly toggle between new and old implementation
- Staging
  - Ideally configured just like production
  - Potentially high infrastructure cost, limited availability
  - Often can't duplicate production load
  - Failures do not harm users
  - Can practice disaster recovery

Chaos engineering

- Originated at Netflix (ChaosMonkey)
- High-reliability, distributed systems must tolerate failure
- Recovery procedures are often not sufficiently rehearsed – painful, risky

- Deliberately inject failures *in production environment*
  - Tests system resiliency under realistic load
  - Encourages recovery automation

Continuous integration ("CI")

- Build and test whole systems regularly
  - Discover issues earlier
  - Reduce integration pain through automation and isolation of issues
  - Test beyond single developer's resources
  - Eliminate reliance on developers' discipline
  - Continuously monitor readiness of code
- Applies to both development and release
  - Continuous build+test
  - Continuous delivery

CI decisions

- *How* to compose systems along release workflow
- *Which* tests to run *when* along release workflow
- Typical setup
    - Pre-submit test suite gates all merges
        - Compilation and fast tests relevant to affected code
    - Post-submit test suite verifies subset of commits on trunk
        - Contains larger, more integrated tests
        - Blesses commits that pass as "green"
    - Release promotion pipeline verifies candidates for release
        - Contains even larger tests, may require dedicated resources

Automation, speed, & infrastructure

- Builds, tests, and deployment must be automated and reliable
  - Ideally completely reproducible
- Most steps must be fast to avoid impeding productivity
  - Cache build products
  - Skip unaffected tests
  - Parallelize & invest in compute resources
- Benefits from tooling
  - Integration with version control and code review
    - Pre-merge and pre-release gates
    - "Last-known-good" branch (new work should branch from here, not trunk)
  - Bisect breakages
  - Log all results
  - Automatically rerun flaky tests

Multi-system CI

- Without monorepo, need to assemble system from several asynchronously-versioned repositories
- Large integration tests can't check every revision/combination
- Objective: identify "configurations" (revision combinations) suitable for promotion (larger-scale testing, release)