Lecture 14: Dynamic Analysis and Testing I

CS 5150, Spring 2026

Lecture goals

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

Quality Assurance (QA)

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?

How to improve software quality (QA)?

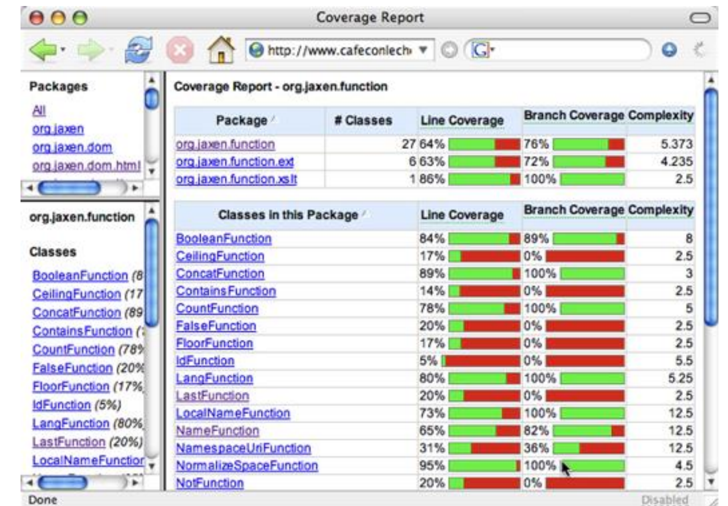
- Code Reviews
- Dynamic Analysis (Testing, Runtime Verification, Sanitizers, …)
- Static Analysis
  - Linters (code conventions, code complexity)
  - Compilers (syntax checks)
  - Static analyzers (security vulnerabilities, data races, …)
- User Testing
- Acceptance Testing

Testing

Assuring external quality

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

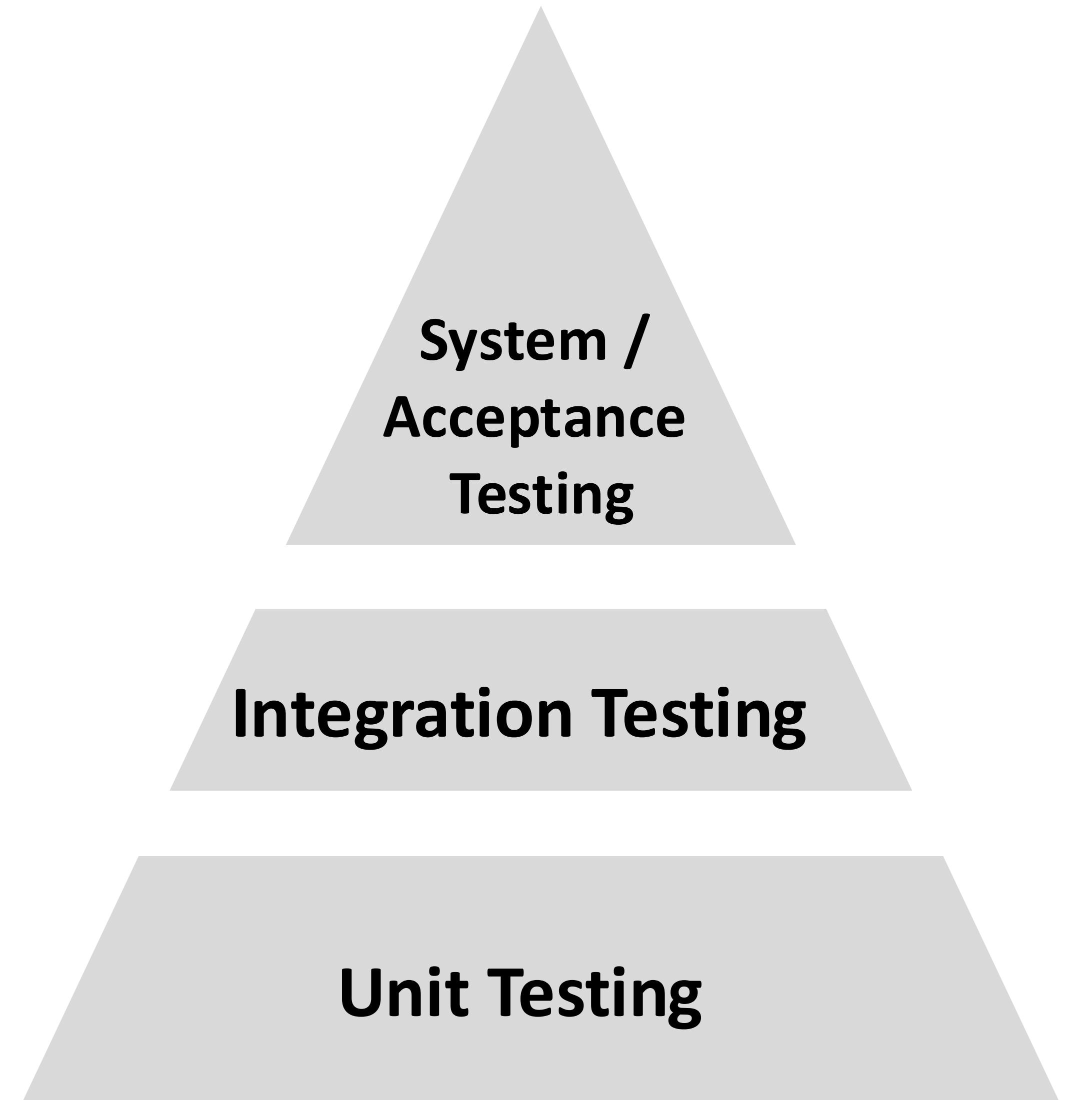
Testing: Basic concepts

- **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



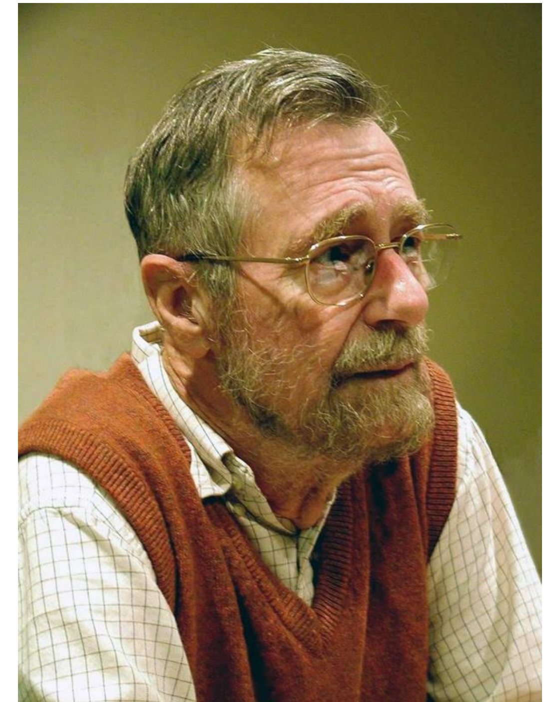
Goals of testing

- 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

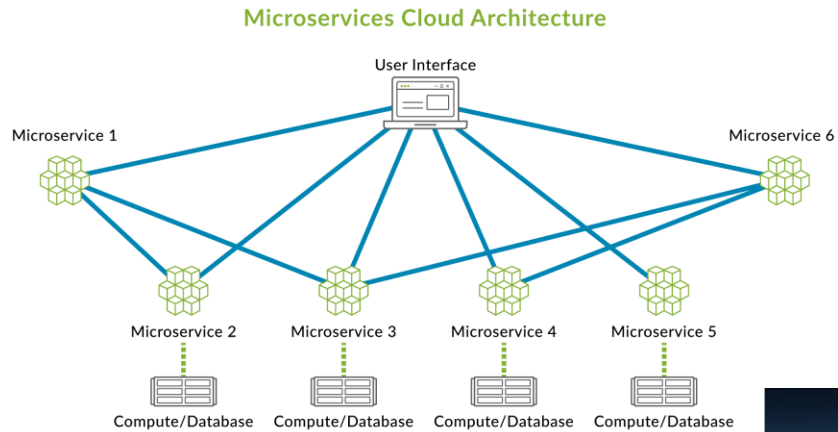
Principles of Testing #5: The pesticide paradox

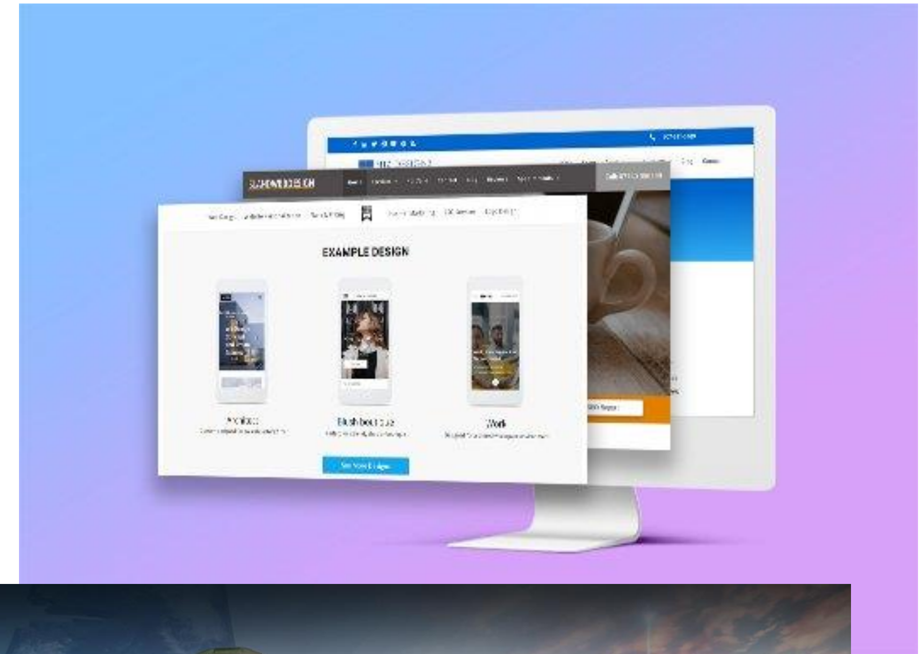
“Every method you use to prevent to find bugs leaves a residue of subtler bugs against which those methods are ineffectual”

- 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**?

How to design/write tests?

- **Unplanned/exploratory**: Add some unit tests, no planning
- **Specification-based testing ("black box"):** Derive test cases from specifications/requirements/user stories
  - Boundary value analysis
  - Equivalence classes
  - Combinatorial testing
  - Random testing
- **Structural testing ("white box"):** Derive test cases to cover implementation "paths":
  - Line coverage, branch coverage

Specification-based Testing

- Tests are based on the specification/requirements/user stories
- **Advantages**:
  - Avoids implementation bias
  - Robust to changes in implementation
  - Tests don’t require familiarity with the code
  - Tests can be developed before implementation

```java
/**
* Computes the final shipping charge for a single order.
*
* Specification:
* - Start from a base shipping amount determined by package weight and distance.
* - Weight tiers should affect pricing (example: <= 5 lb, 5-20 lb, > 20 lb).
* - Distance tiers should affect pricing (example: <= 50 mi, 51-300 mi, > 300 mi).
* - Apply a shipping-speed adjustment based on shippingType: "standard": no speed multiplier, "express": add 25% to the running total, "overnight": add 60% to the running total.
* - Add a fragile-handling surcharge when isFragile is true: +$5 when weight <= 10 lb, +$12 when weight > 10 lb,
* - Apply customer pricing policy based on customerType:"regular": no discount, "premium": 10% discount after surcharges, "business": 15% discount after surcharges.
* - Reject invalid input values (such as negative weight/distance, null/blank shippingType, or unknown customerType) by signaling an error rather than returning a misleading price.
* - Return the total as a non-negative dollar amount representing the final cost
* after all surcharges and discounts.
*/
public static double calculateShippingCost(double weight, int distance, String shippingType, boolean isFragile, String customerType)
```

What about exhaustive testing?

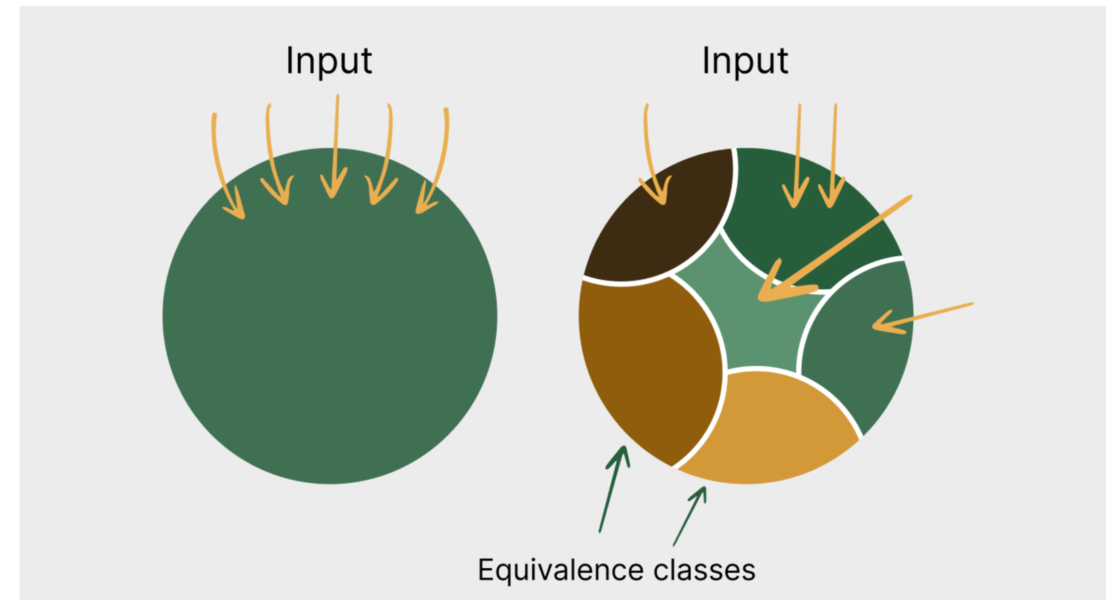
- **Naïve Idea**: Try all possible inputs?!
- Remember the email example? Will take forever!
  - **Weight**: 1g to >100kg
  - **Distance**: 10 miles to 5000 miles
  - **ShippingType**: 2
  - **isFragile**: 2

  ….
- **Key problem**: choosing test suite
  - **Small enough**: finishes in reasonable amount of time
  - **Large enough**: covers most of the scenarios; high confidence!

- Alternative: **Heuristics**

Equivalence Partitioning

- Identify sets with same behavior (**equivalence class**)
- Try one input from each set
- Equivalence classes derived from specifications (e.g., cases, input range, error conditons, fault models)
- Requires domain-knowledge



```java
/**
* Computes the final shipping charge for a single order.
*
* Specification:
* - Start from a base shipping amount determined by package weight and distance.
* - Weight tiers should affect pricing (example: <= 5 lb, 5-20 lb, > 20 lb).
* - Distance tiers should affect pricing (example: <= 50 mi, 51-300 mi, > 300 mi).
* - Apply a shipping-speed adjustment based on shippingType: "standard": no speed multiplier, "express": add 25% to the running
total, "overnight": add 60% to the running total.
* - Add a fragile-handling surcharge when isFragile is true: +$5 when weight <= 10 lb, +$12 when weight > 10 lb,
* - Apply customer pricing policy based on customerType:"regular": no discount, "premium": 10% discount after surcharges,
"business": 15% discount after surcharges.
* - Reject invalid input values (such as negative weight/distance, null/blank shippingType, or unknown customerType) by signaling an
error rather than returning a misleading price.
* - Return the total as a non-negative dollar amount representing the final cost
* after all surcharges and discounts.
*/
public static double calculateShippingCost(double weight, int distance, String shippingType, boolean isFragile, String customerType)
```

What are some equivalence classes here?

Boundary-value analysis

- **Key insight**: Errors often occur at the boundaries of a variable value
- For each variable, select:
  - Minimum
  - Min+1
  - Max-1
  - 0
  - Other domain-specific corner cases

```
/**
* Computes the final shipping charge for a single order.
*
* Specification:
* - Start from a base shipping amount determined by package weight and distance.
* - Weight tiers should affect pricing (example: <= 5 lb, 5-20 lb, > 20 lb).
* - Distance tiers should affect pricing (example: <= 50 mi, 51-300 mi, > 300 mi).
* - Apply a shipping-speed adjustment based on shippingType: "standard":
total, "overnight": add 60% to the running total.
* - Add a fragile-handling surcharge when isFragile is true: +$5 when weigh
* - Apply customer pricing policy based on customerType:"regular": no dis
"business": 15% discount after surcharges.
* - Reject invalid input values (such as negative weight/distance, null/blan
error rather than returning a misleading price.
* - Return the total as a non-negative dollar amount representing the final
* after all surcharges and discounts.
*/
public static double calculateShippingCost(double weight, int distance, String shippingType, boolean isFragile, String customerType)
```

Parameters	Domains
Weight	<5, [5, 20), [20, 100]
Distance	<50, 300
isFragile	T,F
customerType	Regular, premium, business, …,

What are some boundary values here?

Pairwise testing

- **Key Insight**: some problems only occur as the result of an interaction between parameters/components
- Examples of interactions:
  - Bug occurs when **business** customer type ships a **fragile** item (pairwise interaction)
  - Bug occurs when **premium** customer ships **non-fragile** item weighing more **than 300 lb** (three-way interaction)
  - …

- **Claim: Considering pairwise interactions finds about 50% to 90% of defects**

How to measure test adequacy?

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/cs5150sp26

```
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;
}
```

How many test cases are required for full branch coverage?

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