Testing Strategies and Automation

CSCE 740 - Lecture 22 - 11/16/2015

We Will Cover

- Additional structural testing strategies
- Test automation
  - Testing requires programming!
- Automated Testing Strategies

Path Coverage

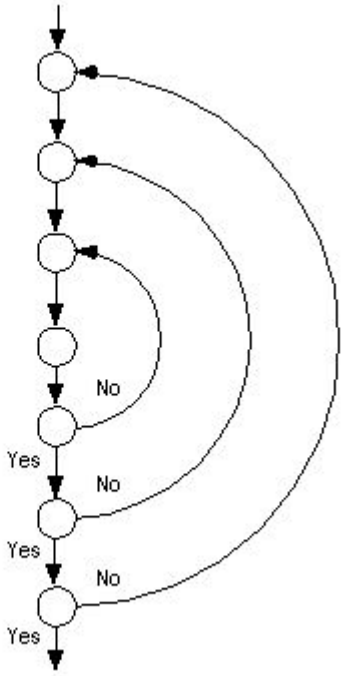
- Path coverage requires that all possible execution paths be covered by tests.
- Theoretically, the strongest coverage metric.
- But… Generally impossible to achieve.
  - Loops result in an infinite number of path variations.

Dealing With Loops

- To make path coverage practical, constrain the number of loop iterations to *representative scenarios*.
- For simple loops, write tests that:
  - Skip the loop entirely.
  - Take one pass through the loop.
  - Take two passes through the loop.
  - Choose an upper bound N, and:
    - M passes, where 2 < M < N
    - (N-1), N, and (N+1) passes

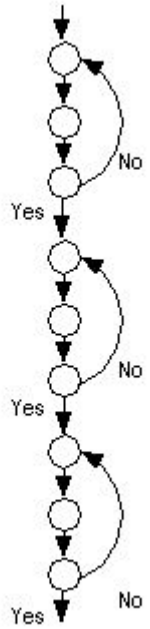
Nested Loops

- Often, loops are nested within other loops.
- For each level, you should execute similar strategies to simple loops.
- In addition:
  - Test innermost loop first with all outer loops executed the minimum number of times.
  - Move one loops out, keep the inner loop at "typical" iteration numbers, and test this layer as you did the previous layer.
  - Continue until the outermost loop tested.



Concatenated Loops

- One loop executes. The next line of code starts a new loop.
- These are generally independent.
  - Most of the time...
- If not, follow a similar strategy to nested loops.
  - Start with bottom loop, hold higher loops at minimal iteration numbers.
  - Work up towards the top, holding lower loops at “typical” iteration numbers.



Why These Loop Strategies?

Why do these loop values make sense?

- In proving formal correctness of a loop, we would establish preconditions, postconditions, and invariants that are true on each execution of the loop, then prove that these hold.
  - The loop executes **zero** times when the postconditions are true in advance.
  - The loop invariant is true on loop entry (**one**), then each loop iteration maintains the invariant (**many**).
    - (invariant and !(loop condition) implies postconditions)
- Loop testing strategies echo these cases.

Identifying the Subpaths

- Number of paths can be limited by identifying a set of subpaths that can be combined to form all paths.
- A control-flow graph has a number of *basis subpaths* equal to:

  number of edges - number of nodes + 2

- This is known as the “cyclomatic complexity” of the control flow graph.

The Subpaths

- The number of paths through this code is exponential.
  - N non-loop branches results in 2^N paths.
- However, there are many overlapping subpaths.
  - number of edges - number of nodes + 2
  - or… number of decision points +1
- We can combine these subpaths to form any path.

```
if (a)      S1;
if (b)      S2;
if (c)      S3;
...
if (x)      SN;
```

1	False	False	False	False
2	True	False	False	Dalse
3	False	True	False	False
4	False	False	True	False
5	False	False	False	True

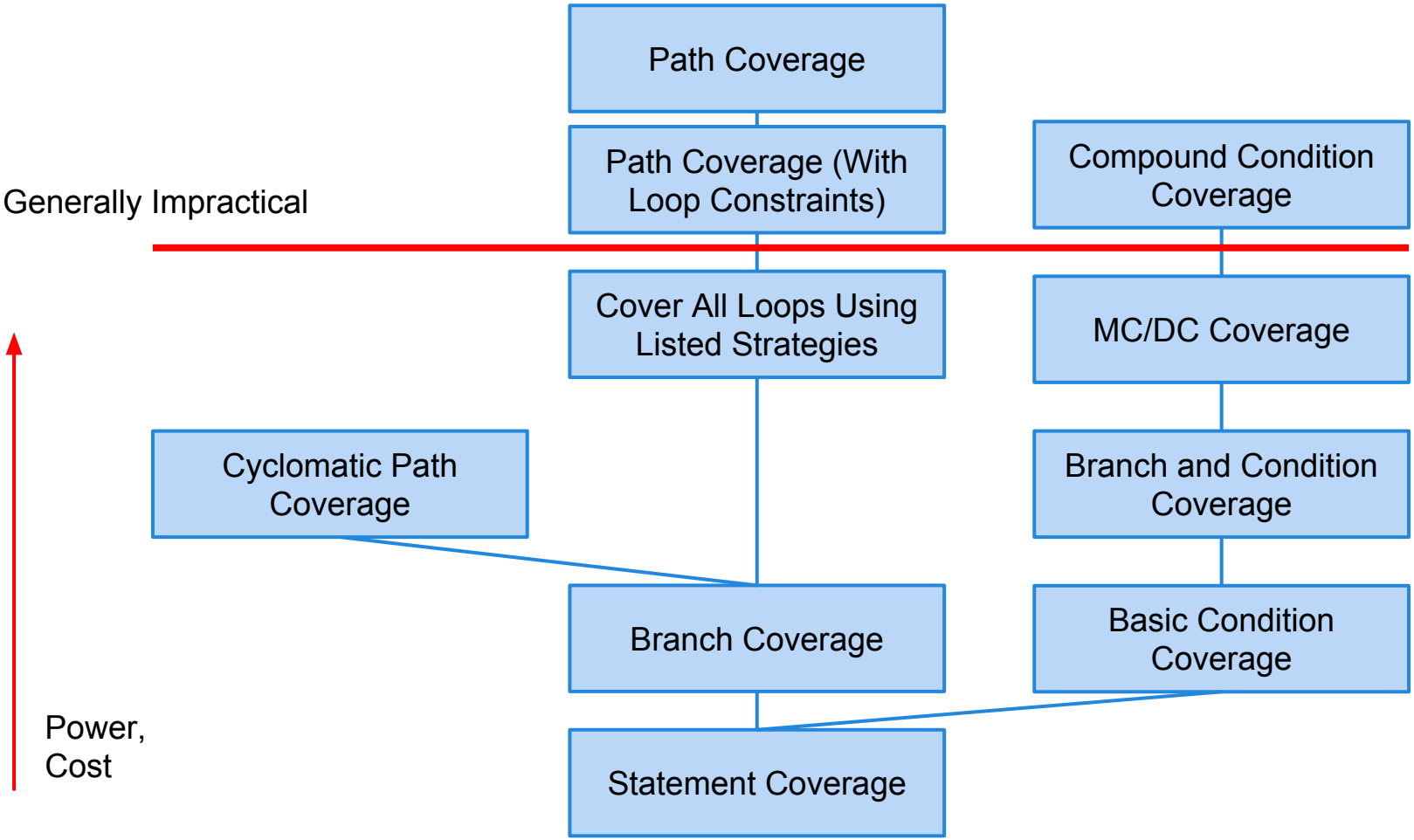
Cyclomatic Testing

- Generally, there are many options for the set of basis subpaths.
- When testing, count the number of independent paths that have already been covered, and add any new subpaths covered by the new test.
  - You are done testing when the number of independent subpaths covered = the cyclomatic complexity.

Uses of Cyclomatic Complexity

- A way to guess “how much testing is enough”.
  - Upper bound on number of tests for branch coverage.
  - Lower bound on number of tests for path coverage.
- Used to refactor code.
  - Components with a complexity > some threshold should be split into smaller modules.
  - Based on the belief that more complex code is more fault-prone.

Which Coverage Metric Should I Use?



Don’t Rely on Metrics

- There is a *small* benefit from using coverage as a stopping criterion.
- But, auto-generating tests with coverage as the goal produces poor tests.
- Two key problems - sensitivity to how code is written, and whether infected program state is noticed by oracle.

Sensitivity to Structure

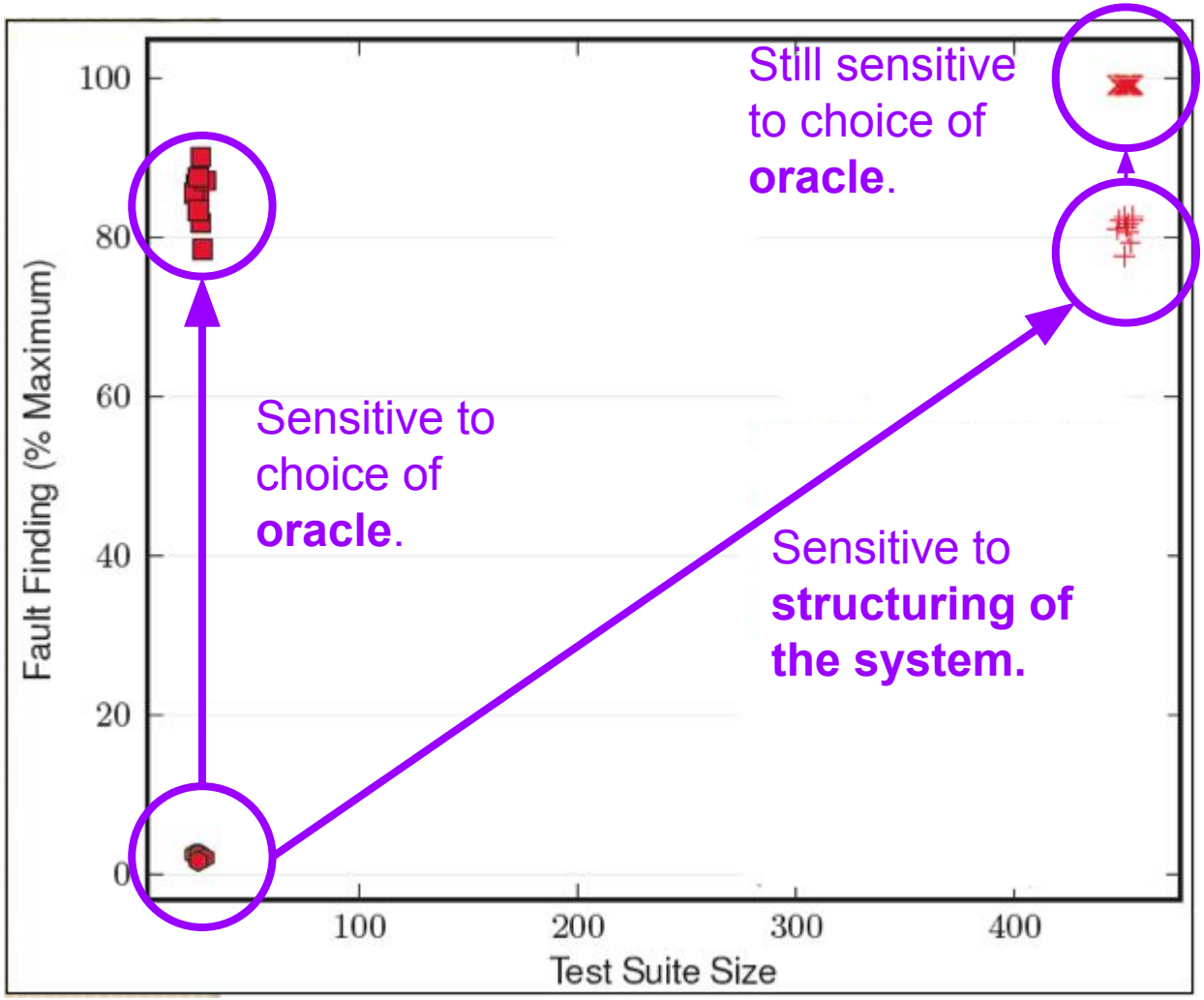
```
expr_1 = in_1 || in_2;
out_1 = expr_1 && in_3;

out_1 = (in_1 || in_2) && in_3;
```

- Both pieces of code do the same thing.
- How code is written impacts the number and type of tests needed.
- Simpler statements result in simpler tests.

Sensitivity to Oracle

- The oracle judges test correctness.
  - We need to choose what results we check when writing an oracle.
- Typically, we check certain output variables.
  - However, masking can prevent us from noticing a fault if we do not check the right variables.
  - We can’t monitor and check all variables.
  - But, we can carefully choose a small number of bottleneck points and check those.
    - Some techniques for choosing these, but still more research to be done.

Coverage Effectiveness



Masking

Why do we care about faults in masked expressions?

- Effect of fault is only masked out for *this* test. It is still a fault. In another execution scenario, it might not be masked.
- We just haven't noticed it yet.
  - The fault isn't gone, we just have bad tests.

What Can Be Done?

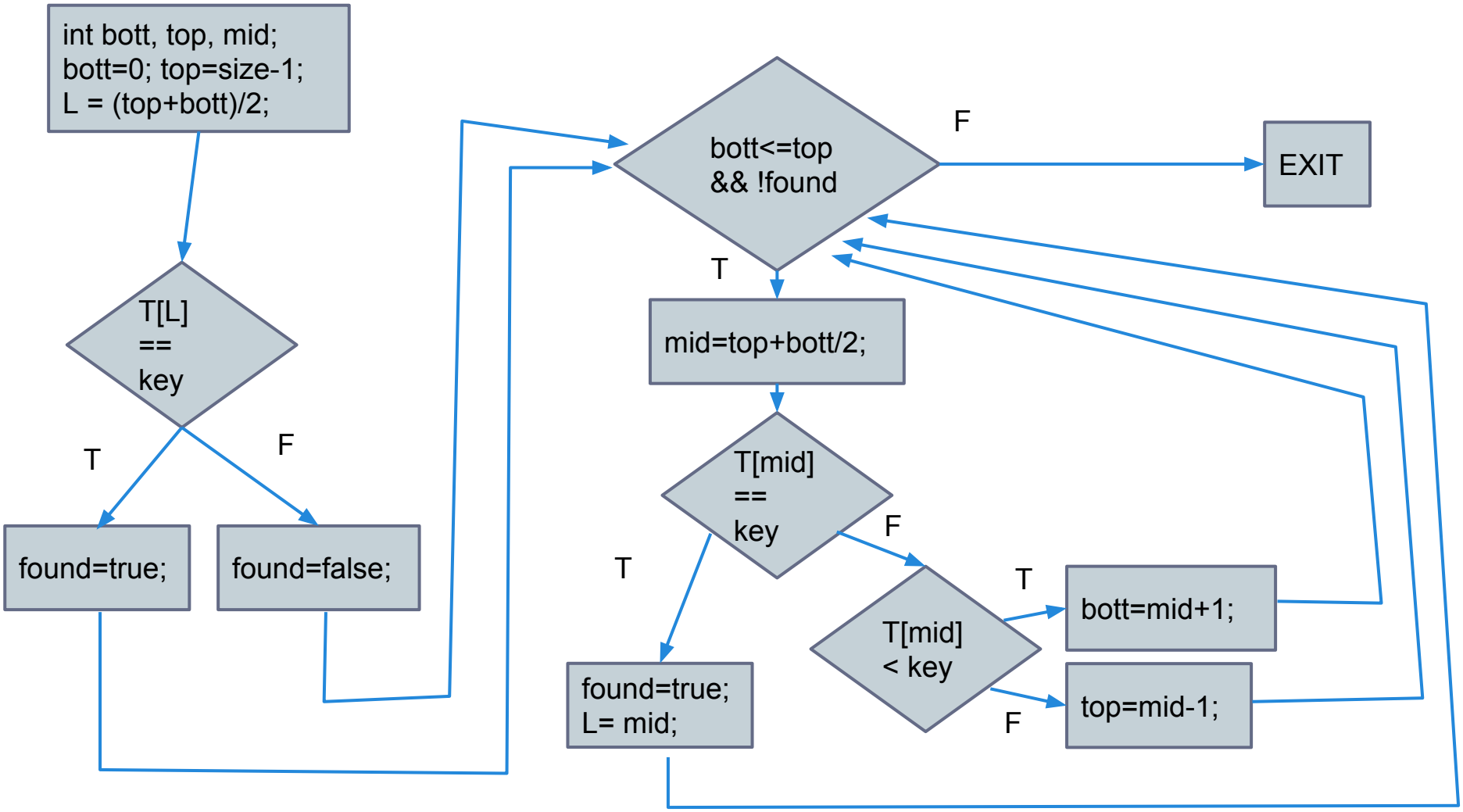
- Take masking into account.
  - Find where masking can take place, and write tests that offer a *masking-clear* path to monitored variable.
  - Write oracles that check values of variables likely to being masked.
- Don’t forget about the requirements and input/output partitions.
  - Require *both* code coverage and coverage of function outcomes and input partitions.
  - More complex tests - input that results in particular outputs while exercising targeted code structures.

Activity: Writing Structure-Based Tests

For the binary-search code:

1. Draw the control-flow graph for the method.
2. Identify the cyclomatic complexity.
   number of edges - number of nodes + 2
   ***number of decision points* + 1**
3. Develop a test suite that exercises the loops using the guidelines presented earlier.

Activity: Writing Structure-Based Tests



Executing Tests

Once you have tests, how do you execute them?

- You could run them and check results by hand
  - Don't do this. Please.
- **Test Automation** is the use of software to separate repetitive tasks from the creative aspects of testing.
  - Control the execution of tests,
  - Compare predicted and actual output,
  - Control the environment and preconditions.

Anatomy of a Test Case

- Test Input
  - Any required input data.
- Expected Output (Test Oracle)
  - What *should* happen, i.e., values or exceptions.
- Initialization
  - Any steps that must be taken before test execution.
- Test Steps
  - Interactions with the system (such as method calls) , and output comparisons.
- Tear Down
  - Any steps that must be taken after test execution to prepare for the next test.

Testing Requires Writing Code

Test scaffolding is a set of programs written to support test automation.

- Not part of the product
- Often temporary

Allows for:

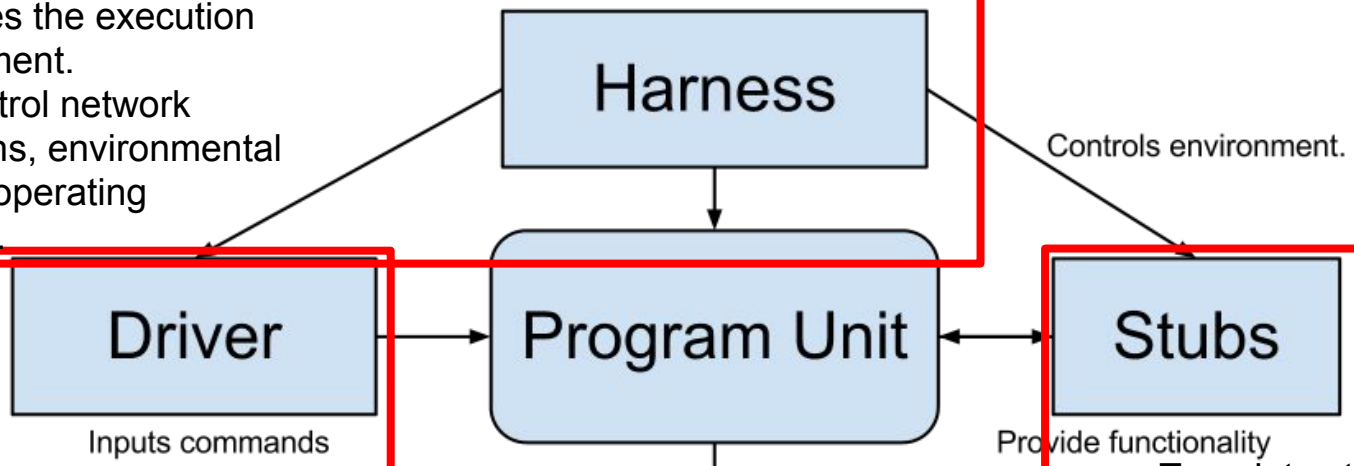
- Testing before all components complete.
- Testing independent components.
- Control over testing environment.

Test Scaffolding

- A **driver** is a substitute for a main or calling program.
  - Test cases are drivers.
- A **harness** is a substitute for all or part of the deployment environment.
- A **stub** (or **mock object**) is a substitute for system functionality that has not been completed.

Test Scaffolding

- Simulates the execution environment.
- Can control network conditions, environmental factors, operating systems.



- Initializes objects
- Initializes parameter variables
- Performs the test
- Performs any necessary cleanup steps.

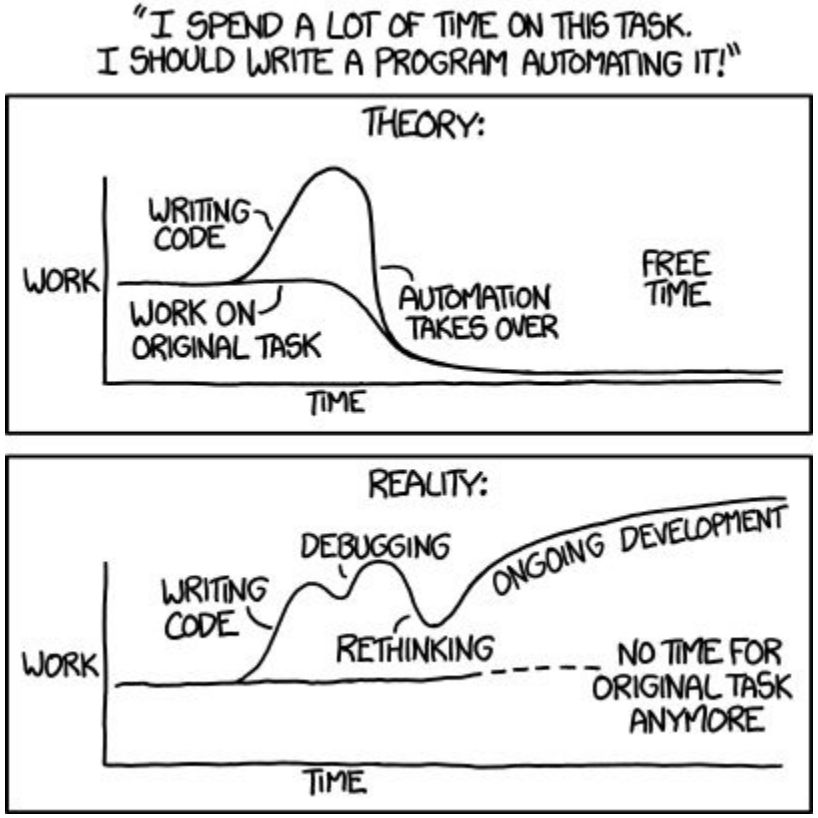
- Templates that provide functionality and allow testing in isolation

- Checks the correspondence between the produced and expected output and renders a test verdict.

Generic vs Specific Scaffolding

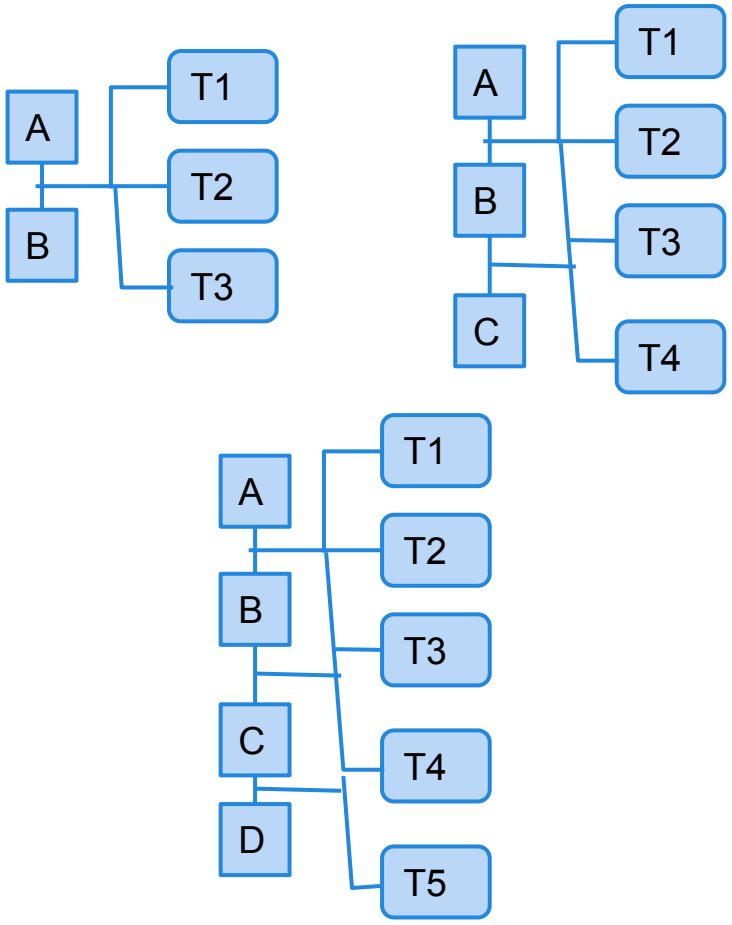
- Simplest driver - one that runs a single specific test case.
- More complex:
  - Common scaffolding for a set of similar tests cases,
  - Scaffolding that can run multiple test suites for the same software (i.e., load a spreadsheet of inputs and run then).
  - Scaffolding that can vary a number of parameters (product family, OS, language).
- Balance of quality, scope, and cost.

Automation Trade-Offs



Some common strategies help guide automation.

Incremental Testing



Test pieces of the system as they are completed. Use scaffolding (stubs, drivers) to test in isolation, then swap out for real components to test integration.

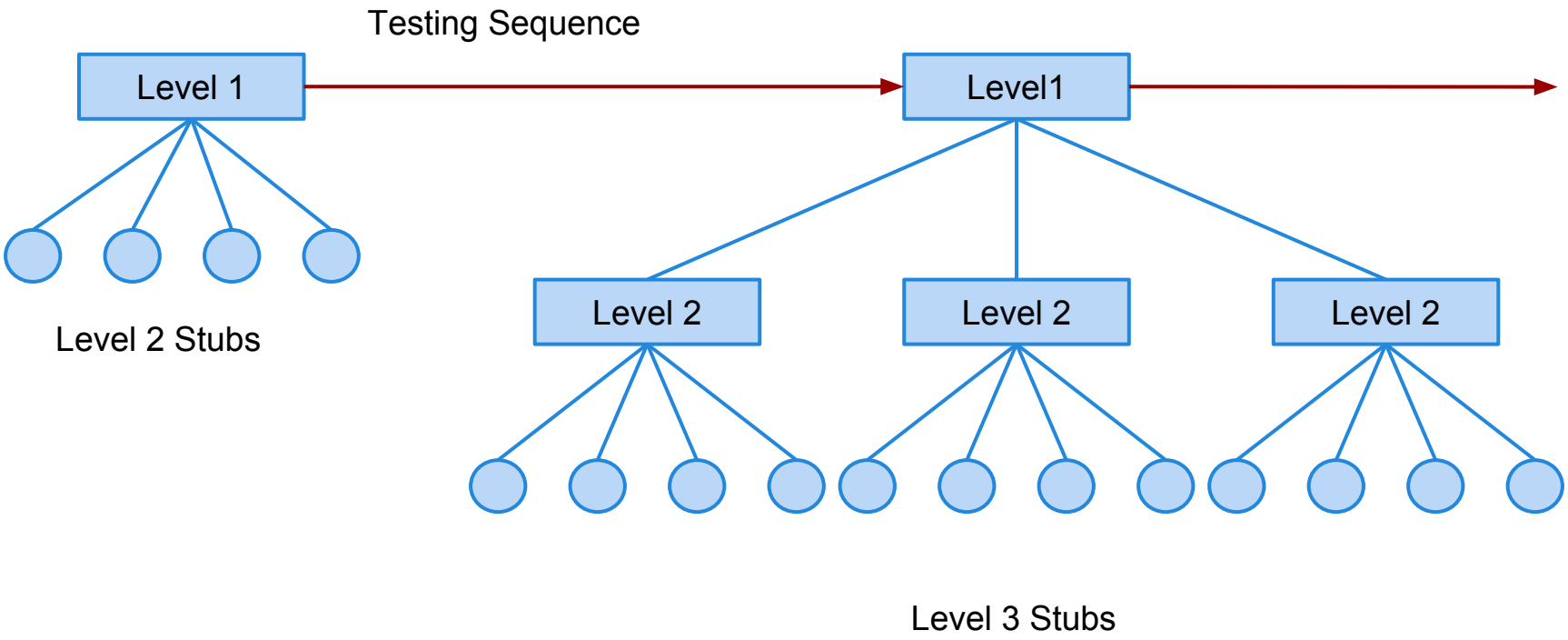
Advantages:

- Easily test components in isolation.
- Discover faults earlier.

Disadvantage:

- Expensive to develop scaffolding.

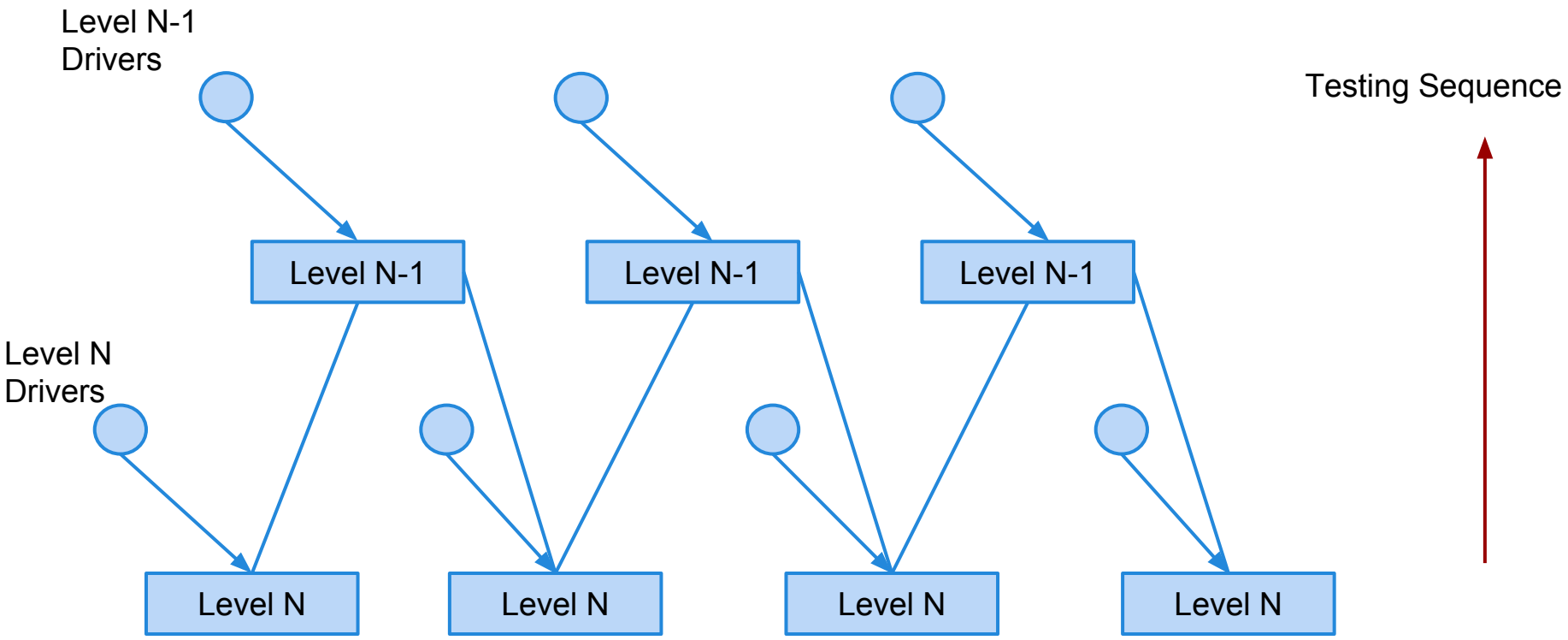
Top-Down Testing



Top-Down Testing

- Start with the high levels of a system (based on control-flow, data-flow, or architecture) and work your way downwards.
  - Use in conjunction with top-down development.
- Very good for finding architectural or integration errors.
- May need system infrastructure in place before testing is possible.
- Requires large effort in developing stubs.

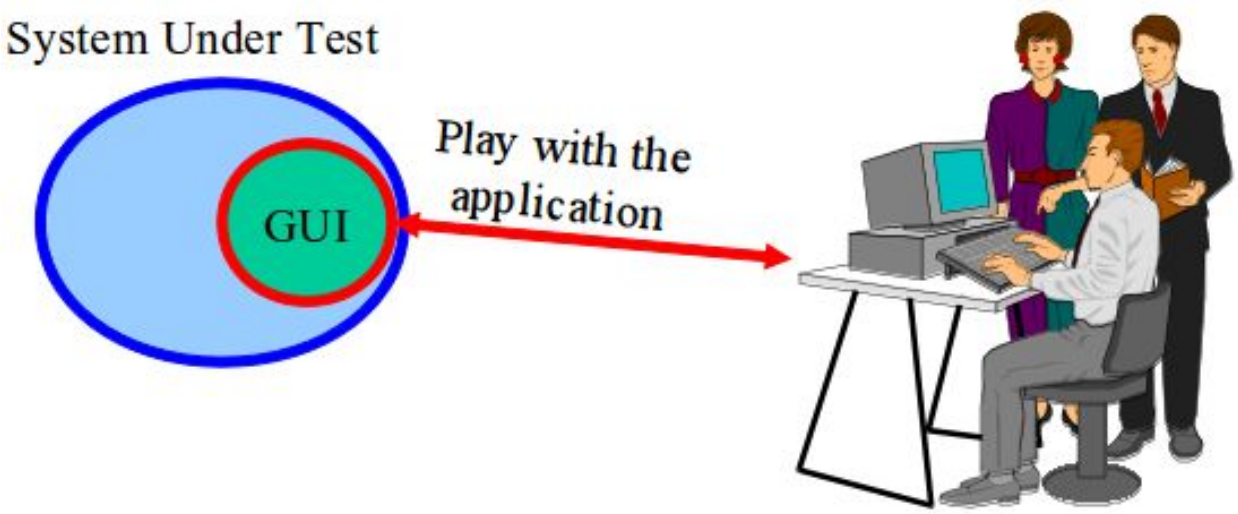
Bottom-Up Testing



Bottom-Up Testing

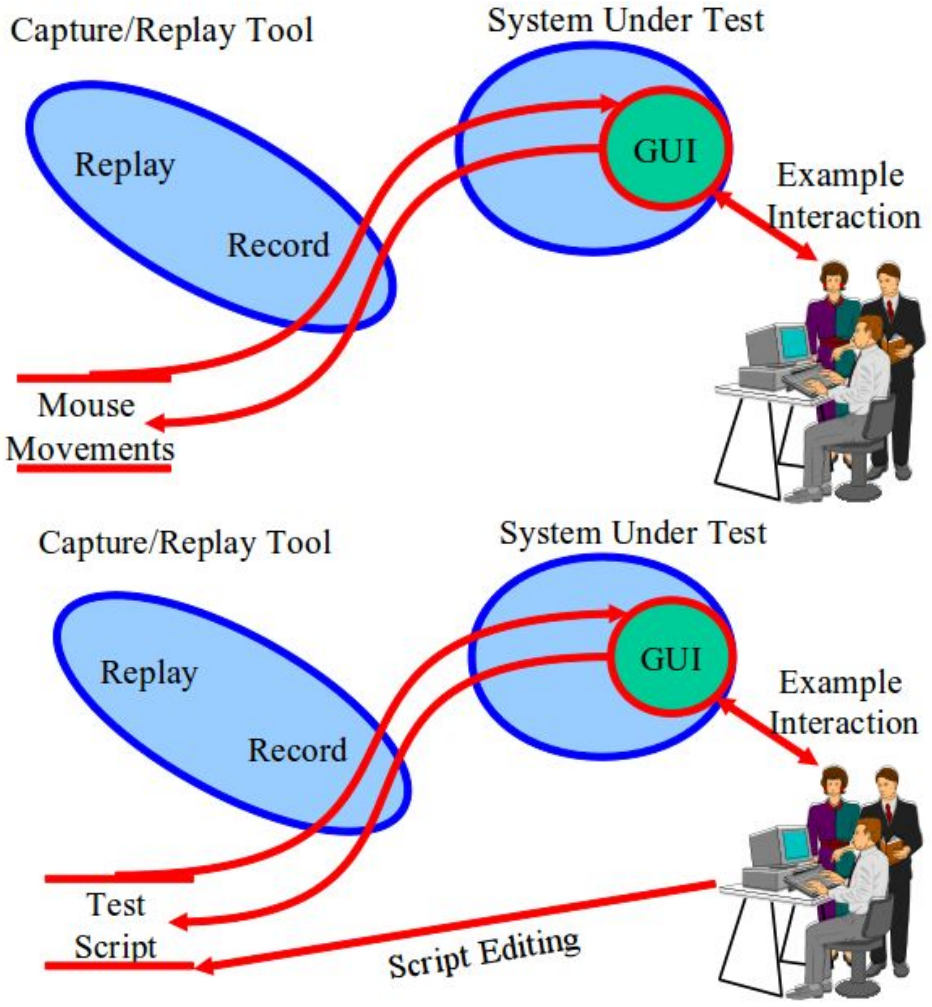
- Start with the lower levels of a system (based on control-flow, data-flow, or architecture) and work your way upwards.
  - Use in conjunction with bottom-up development.
- Appropriate for object-oriented systems.
- Necessary for testing critical infrastructure.
- Does not find major design problems, but very good at testing individual components.
- Requires high effort in developing drivers.

What About Graphical Interfaces?



- Graphical components of projects often tested manually by real users.
- Heavily tested during alpha/beta testing.

Capture and Replay



1. Have a human interact with the system, walking through several different scenarios.
2. Record their mouse motions and clicks during these scenarios.
3. Take these test cases and modify them to create additional tests.

Six Essentials of Testing

Adapted from Software Testing in the Real World, Edward Kit; Addison-Wesley, 1995

- The quality of the test process determines the success of the test effort.
- Prevent defect migration by using early life-cycle testing techniques.
  - Start testing early.
- The time for software testing tools is now.

Six Essentials of Testing

- A real person must take responsibility for improving the testing process.
- Testing is a professional discipline requiring trained, skilled people.
- Cultivate a positive team attitude of creative destruction.

The Key to Effective Testing: Offering the Right Incentives

We Have Learned

- Loop strategies to make path coverage more realistic.
- How coverage criteria relate in terms of cost and power.
- Test automation can be used to lower the cost and improve the quality of testing.
- Automation involves creating drivers, harnesses, stubs, and oracles.
- Systems can be tested in a top-down or bottom-up style.

Next Time

- We’ve looked at developing test inputs…
- What about the expected output?
  - Writing Test Oracles
- Homework - questions?