# Test Oracles

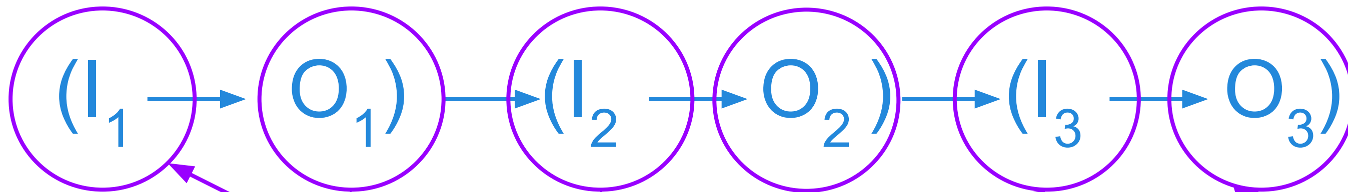
CSCE 747 - Lecture 11 - 02/22/2018

# Software Testing - Back to the Basics

Tests are sequences of **stimuli** and **observations**. We care about input and output.

$$(I_1 \rightarrow O_1) \rightarrow (I_2 \rightarrow O_2) \rightarrow (I_3 \rightarrow O_3)$$

# What Do We Need For Testing?




**Test Inputs**

How we “stimulate” the system

**Test Oracle**

How we check the correctness of the resulting observation.

if $O_n$ = Expected($O_n$)

then… Pass

else… Fail

# We Will Cover

Software Test Oracles

- Where do they come from?
- How do we create them?
- Why are they important for testing?

# Test Oracle - Definition

If a software test is a sequence of activities (*stimuli and observations*), an **oracle** is a predicate that determines whether a given sequence is acceptable or not.

An oracle will respond with a *pass* or a *fail* verdict on the acceptability of any test sequence for which it is defined.

# Test Oracles and Specifications

- In verification, an implementation is checked for conformance to a specification.
- When executing a test case, the correctness of the implementation is checked by an oracle.
- Testing is a form of verification.
- **Is an oracle a specification?**
  - **Is a specification an oracle?**

# Test Oracle Components

An oracle is *an implementation of a specification*.

- **Oracle Information**
  - The information used by the oracle to judge the correctness of the implementation, given the inputs.
  - A specification, in a form that can be used directly by the testing code.
- **Oracle Procedure**
  - Code that uses that information to arrive at a verdict.
  - A form of *automated verification*.
  - Commonly as simple as...

```
(value(system output) == value(expected output))
```

# Oracles are Code

- Oracles must be developed.
  - Like the project, an oracle is built from the requirement specification.
    - … and is subject to interpretation by the developer
    - … and may contain faults
- A faulty oracle can be trouble.
  - May result in false positives - “pass” when there was a fault in the system.
  - May result in false negatives - “fail” when there was not a fault in the system.

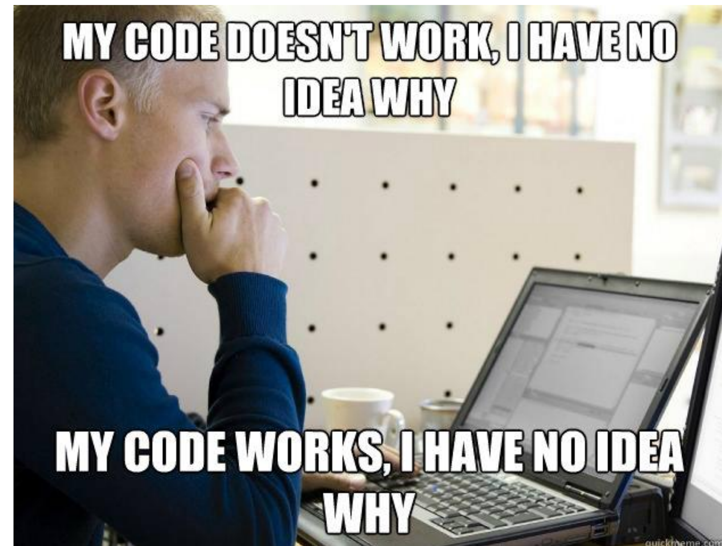
# Oracle Verification

- Verification can be performed on oracles.
  - Does the oracle conform to the specification it was built from?
  - i.e., we could write tests for the oracle.
- Circularity problem.
  - Build code to judge the system, then build code to judge the judge, then what?
- The oracle should be subjected to a lightweight verification process.
  - At least a code review.

# Where Do We Get Test Oracles?

**Most commonly:**

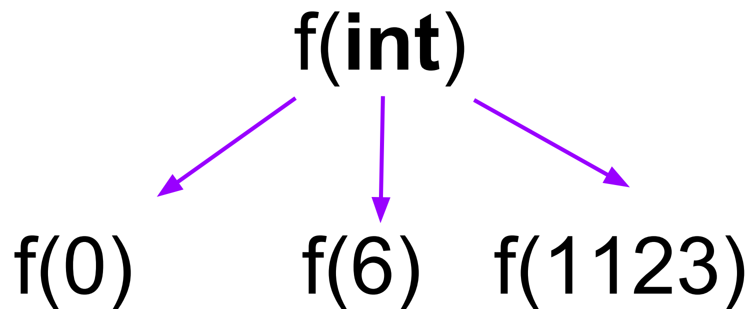
Developers write oracles by hand, tied to a particular test case.




- Large amount of manual effort and time required to create tests
- Will not be able to run many tests. Did you choose the right inputs?

# The “Test Oracle Problem”

We are good at coming up with new input...




But, it is **much harder** to automatically check results for multiple tests.

f(0) = **?**

f(6) = **?**

f(1123) = **?**

# Oracle Trade-Offs

- We can specify the exact output behavior expected from the system.
  - This is very hard to do for more than one test at a time.
- Or, trade *precision* for *generality*.
  - Specify properties that should be obeyed by a function.
  - Build a model of a function.
  - Check for types of anomalies that all programs can suffer from.

# Test Oracle

- A test oracle is **complete** if it can offer a verdict for any set of test input.
- A test oracle is **sound** if it offers the right verdict for any test case that it can offer a verdict for.
- A test oracle is **correct** if it is both sound and complete.
  - It is *partially correct* if it is sound, but not complete.

# Judging the Judge

**What properties do we seek in an oracle?**

- Cost to Build (Per Test and Overall)
  - How much effort goes into building it?
  - Want **low cost**.
- Accuracy of Verdicts
  - Can it give the wrong answer?
  - Want **high accuracy**.
- Completeness
  - Can it offer verdicts for multiple test cases?
  - What kind of situations is it useful for?
  - Want **high completeness**.

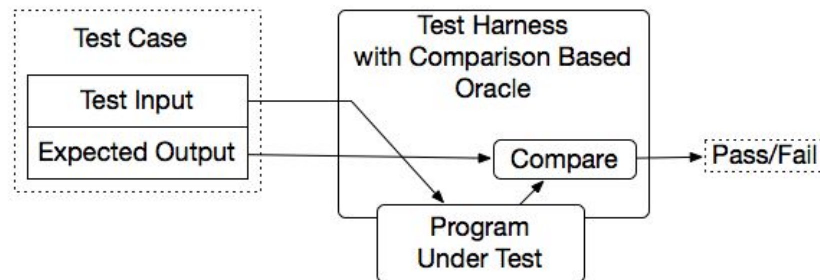
# Types of Oracles

- **Specified Oracles**
  - Developers, using the requirements, formally specify properties that correct behavior should follow.
- **Derived Oracles**
  - An oracle is derived from development artifacts or system executions.
- **Implicit Oracles**
  - An oracle judges correctness using properties expected of many programs.
- **Human Oracles**
  - How do you handle the lack of an oracle?

# Specified Oracles

# Specified Oracles

**Specified Oracles** judge behavior using a human-created specification of correctness.




- Most oracles are specified oracles.
  - Any manually-written test case has a specified oracle.

# Expected-Value Oracles

Simplest oracle - what is the expected value given this input?

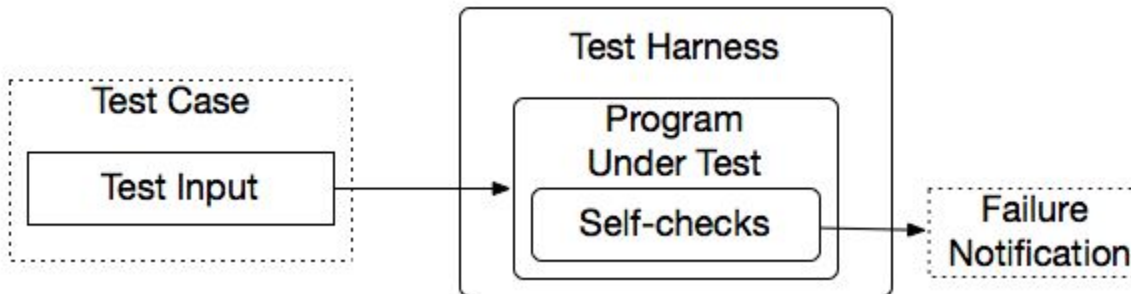
```
int expected = 7;
int actual = max(3, 7);
assertEquals(expected, actual);
```

- Usually not reusable.
- How can we extend this to multiple tests?

# Self-Checks as Oracles

Rather than comparing actual values, use properties about results to judge sequences.

```
@Test
public void multiplicationOfZeroIntegersShouldReturnZero() {
  // Tests
  assertEquals("10 x 0 must be 0", 0, tester.multiply(10, 0));
  assertEquals("0 x 10 must be 0", 0, tester.multiply(0, 10));
  assertEquals("0 x 0 must be 0", 0, tester.multiply(0, 0));
}
```




```
@Test
public void propertiesOfSort (String[] input) {
  // Tests
        String[] sorted = quickSort(input);
        assert(sorted.size >= 1, "This array can’t be empty.")
}
```

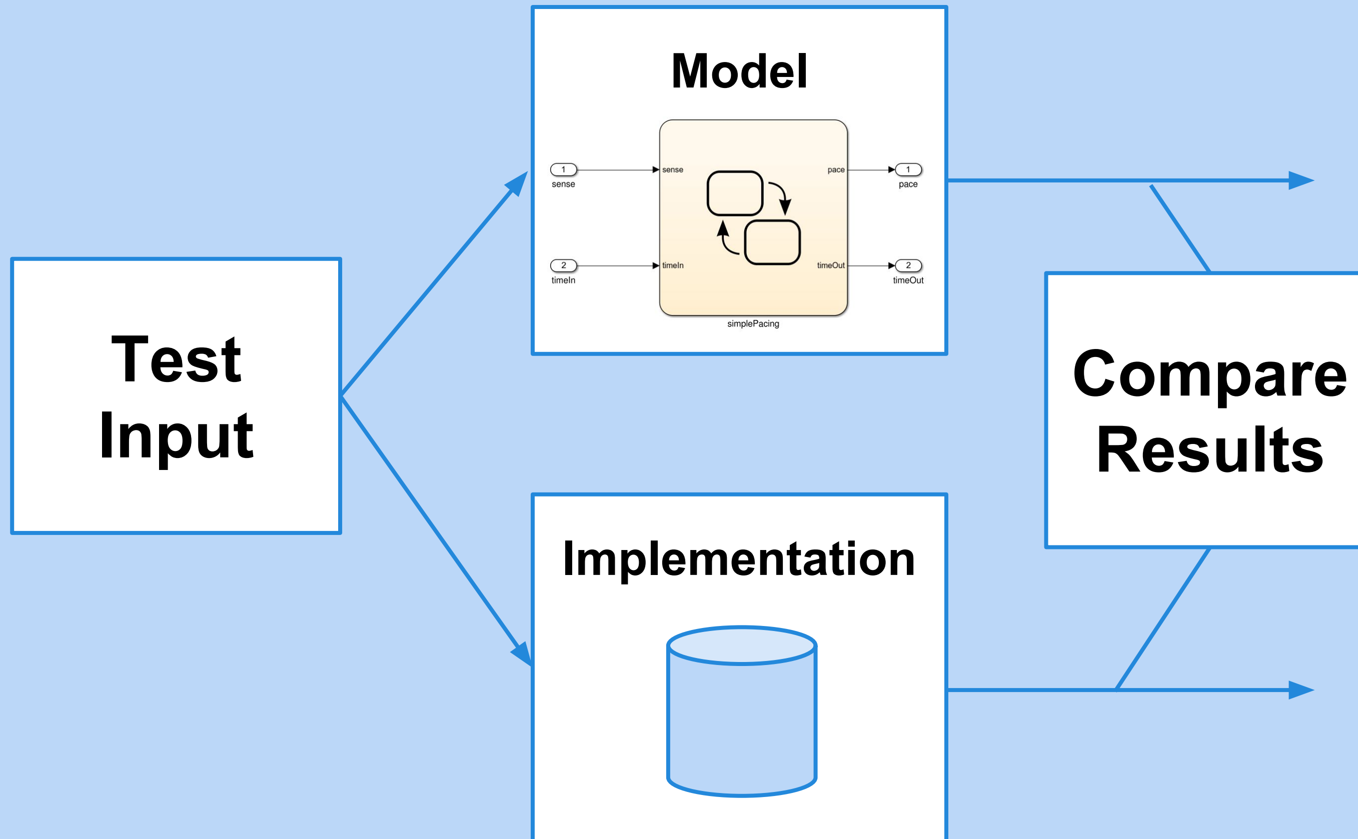
Take the form of assertions, contracts, and other logical properties.

# Self-Checks

- Usually written at the function level.
  - For one method or one high-level “feature”.
  - Properties based on behavior of that function.
- Work for any input to that function.
- Only accurate for those properties.
  - Faults may be missed if the specified properties are obeyed.
  - More properties = more expensive to write.

# System Models as Oracles

**Models could potentially serve as a "universal" test oracle**



# Problem: Abstraction

- Like in finite state verification, models require abstraction.
  - Models are useful for requirements analysis, but may not reflect operating conditions.
  - May get “fail” verdict because the system’s behavior does not match, but the system acted correctly.
- Models are highly reusable, but less accurate than other oracles.

# Derived Oracles

# Derived Oracles

If no specified oracle exists, **oracles can be automatically derived** from existing sources of information:

- Project Artifacts
  - Documentation
  - Existing tests
  - Other versions of the system
- Program Executions
  - Invariant detection
  - Specification mining

# Pseudo Oracles (N-Version Programming)

An alternate version of the program as an oracle.

Does `output(V1) = output(V2)`?

- Pseudo Oracle because we know the two don’t agree, but don’t know which is wrong or why.

Also called N-Version programming, where multiple designs are implemented, or same design is implemented by independent teams.

Genetic programming can (through genetic algorithms) automatically produce multiple implementations.

# Regression Testing

When changes are made to a system, rerun your tests. Any existing tests that passed previously should still pass.

An older version of your program can be the oracle.

- Do new features break working features?
- Do bug fixes break working features?
- If requirements have changed, you do NOT want the output to match for features related to the requirement.

# Metamorphic Relations

If you have test cases, you can generate partial oracles for follow-up tests by deriving **metamorphic relations** between tests.

- A metamorphic relation is a necessary property of a function:
  - A property of a sin function is that sin(x) = sin (pi - x).
  - Thus, sin(x) and sin(pi - x) have the same expected output.
- If these relationships are violated, then there is a bug.
- Can be an equation or more general properties specified between inputs.

# Invariant Detection

Invariants (pre/post-conditions) can be specified as a form of test oracle. If they are not known in advance, there are algorithms that can detect them from program executions.

- Testers take a set of tests that the program is known to produce correct behavior for.
- Properties true of all observed executions are extracted for methods, loops, and conditional statements.

# Model Inference

From system executions, we can derive a state machine model of system execution. As we observe more executions, we refine the model.

Major problem of both invariant and model detection: **Accuracy!**

- The more executions observed, the more accurate, but requires much more effort.
- What do we do about this?

# Implicit Oracles

# Implicit Oracles

Implicit oracles require no domain knowledge or specification, but instead can be applied to check properties that are expected of any runnable program.

Implicit oracles often detect particular anomalies, such as network irregularities or deadlock. These are faults that do not require expected output to detect.

# Uses of Implicit Oracles

Implicit oracles can be built to detect:

- Concurrency Issues
  - Deadlock, livelock, and race detection
- Violations of properties related to non-functional attributes of the system
  - Performance properties
  - Robustness
  - Memory access and leaks

# Fuzzing

Fuzzing is a way to find implicit anomalies.

- Generate random (or fuzz inputs).
- Attack the system with these inputs.
  - Generation and attacks guided by “attack profiles” that reflect certain malicious use scenarios.
- Report anomalies with the test sequence that caused them.

Used to detect security vulnerabilities.

# Human Oracles

# The Human Oracle

- If no automation is possible or no specification exists, a human can always judge output manually.
- Not ideal, but surprisingly common in practice.

# Handling the Lack of Oracles

Even if there is no oracle, there are techniques that can reduce the *human oracle cost* through:

- Quantitative reduction in the amount of work the tester has to do for the same amount of fault-detection potential.
- Qualitative increase in the ease of evaluating testing results.

# Quantitative Cost Reduction

Test suites can be unnecessarily large:

- Tests that cover too few testing goals or scenarios.
- Tests that are unnecessarily long, with redundant method calls.

Human oracle cost can be reduced by cutting out either of these.

- Test suite reduction techniques cut tests that cover redundant code structure or do not penetrate deeply into the code.
- Test case reduction techniques attempt to remove unnecessary test steps.

# Qualitative Cost Reduction

Not all test cases are equally understandable by human testers. Automated test generation often produces test inputs that do not match the expected usage of a program, and humans have trouble judging the results of such tests.

Some test generation approaches allow the seeding of human knowledge or use usage profiles to help generate input.

# Crowdsourcing the Oracle

Recent development - outsource the oracle problem to many different human oracles.

Several services exist for this now - Amazon Mechanical Turk, Mob4Hire, MobTest, uTest.

Users cannot be expected to have much domain knowledge, so understandability of test inputs and documentation are very important.

# Placing the Oracles

| | Cost(T/O) | Accuracy | Completeness |
|---|---|---|---|
| Manual Specification | L/H | H | L |
| Behavioral Model | H/L | M-H | H |
| Self-Checks | L/M | H | M |
| N-Version Programming | L-H/L | L-H | M-H |
| Metamorphic Testing | L/M | H | M |
| Invariant Detection | L/L | L-H | M |
| Implicit Oracles | L/L | H | L |

# We Have Learned

- Test Oracles judge the correctness of sequences of stimuli and observations.
- Oracles are implementations of specifications.
- Oracles can be:
  - specified (expected values, models, assertions)
  - derived from correct executions or project artifacts
  - built to detect implicit properties
  - humans asked to check results

# Next Time

- **No class Tuesday!**
- Fault-Based Testing
    - Reading - Ch. 15
- Homework 2 due March 6th.
    - Any questions?