Software Testing, Quality Assurance and Maintenance Winter 2026

Lecture 5 — January 19, 2026

Patrick Lam *version 1*

We've talked about the notion of a *software failure*. Recall that failures are caused by a *fault* in the software which is executed and whose effect propagates to output.

But, who's to say that the output is correct or incorrect? This is the oracle problem. See [BHM+15] for an exhaustive survey of the research literature on the oracle problem as of 2015.

The Oracle Problem

One can say "ask a human", but let's think some more about that.

It really is begging the question, in the strict sense of the term, but I've been known to just take the system's output as being correct when writing test cases. If we're doing that, what we're doing boils down to regression testing (depending on how critically we consider the output).

More fundamentally, though, how should the human know what correct behaviour is?

This one seems easy:

```
def add(x,y):
  return x + y
```

Everyone will agree[1] that `add(1,1)` should return 2.

Now, let's consider root-finding. You have a function `solve_quadratic()` which solves quadratic equations and you want to test it. If you give it $x^2 - 2x - 4 = 0$, how do you know what answer to test for? You need to read the function name and remember high school math. Also, there are the usual edge cases: what should happen if there are no solutions? (Also, don't forget floating-point issues e.g. with comparisons.)

As a human oracle, at a unit test level, you are using the function/class name as the specification. Maybe you also consider the code documentation, if it exists. At all levels, you combine the information that you have with your human experience. Hopefully it is reflective of the user base.

Helping Human Oracles

There's no magic here. Later today, we'll talk about some ways that we can otherwise calculate the right answer, but sometimes there is no alternative to a human looking at the test case and deciding whether it is correct or not, as discussed just above. It comes down to whether the output meets its requirements. Eliciting the requirements is a whole other course.

In terms of helping humans find the right answer, the literature talks about making automatically-generated inputs easier to understand, e.g. in [MSH10].

[1] JavaScript chooses chaos for `"3"+5`.

For instance, if you are testing a function that computes the days between two dates, you can easily manually evaluate the number of days between 12/24/2025 and 12/25/2025. But if one of your inputs is -5455/23195/-30879, that's going to be hard to calculate. (The function under test—not shown—sanitizes invalid inputs so that date gets parsed as 1/31/-30879, but is that a leap year?).

The suggestion is instead to generate inputs that fit expected input profiles. One might start from inputs that developers use as sanity checks. (You sanity check your code, right?) Other helpful hints when creating profiles include sanitizing checks inside the code and variable names. For instance, common knowledge is that valid values for months are between 1 and 12; one can test valid values, 0, negative values, and a value greater than 12.

There's more in [MSH10] about searches starting from normal inputs, genetic algorithms, and generating from distributions.

Concretely, it's also possible to reuse partial inputs. This makes sense from a testing point of view too. Change one thing at a time—it is easier to reason about the change in the output given a single change in the input. Go from input 0/1/2010 to 1/1/2010.

One more comment here. Sensible strings are harder to generate automatically than numbers, since the space of strings is bigger and the space of sensible strings is proportionately smaller. Random strings are good as fuzzed inputs, but you also want strings that pass sanity checks. One can mine the web for strings, or generate strings using metaheuristics (or, these days, LLMs, I guess).

Crowdsourcing. People have tried to ask Mechanical Turk for the right answer, but apparently it's hard.

Reducing the volume of work for human testers. There's what seems to me like the eternal dream of test suite reduction, for which there are no general-purpose solutions that I'm aware of. Also test case reduction, which we'll talk about in the fuzzing module.

Implicit Oracles

Segfaults are always bad. The easiest way to label a test execution as incorrect is when it exhibits a segfault during execution. Same with buffer overflows, though you might need a tool to observe those. But, in general, you don't need to know anything about the domain or the specification to label such tests.

Other types of crashes are also most often incorrect (though an uncaught exception may be better than silently failing). Similarly, livelock, deadlock, and race conditions are undesirable. Memory leaks and performance degradations (with respect to a baseline) are other things that can be automatically detected.

Implicit oracles and assertions are key when fuzzing—since the inputs are automatically generated at high volume, there is going to be no way to check that the outputs are correct. We can only check that the outputs don't break assertions and don't crash the program.

These implicit oracles don't give definitive evidence that the system under test is correctly computing a result.

Specification-based Oracles

What should an implementation do? Well, one can use a *specification* to specify what the implementation should do. A specification is some sort of description of a part of a system. Let's make that a bit more specific.

Model-based specification. There are a lot of modelling languages out there, which one can use to describe (aspects of) system behaviour more or less formally (depending on the language). Specifically, for a model-based specification, one creates a model of system state, typically using sets or relations, and then specifies operations that modify the modelled state.

Here's an example of an action predicate written in the Alloy programming language[2].

```
pred upload [f : File] {
  f not in uploaded           // guard
  uploaded' = uploaded + f    // effect on uploaded
  trashed' = trashed          // no effect on trashed
  shared' = shared            // no effect on shared
}
```

Elsewhere in the model, we have declared sets of uploaded, trashed, and shared files. Here, we're saying that the upload action has a precondition that file `f` not previously be uploaded, and that it is uploaded after completion of the action. Also, the trashed and shared sets remain unchanged after the action. (Alloy models can also express invariants.)

One can then use the specification to write test cases which verify the specified behaviour, and test implementations using these test cases:

- try to upload an already-uploaded file;
- when uploading a new file, check that the file ends up in the uploaded set, checking that trashed and shared remain the same.

The action predicate is saying what the result of the action is supposed to be—it acts as an oracle. Models can also be verified for internal consistency.

The model uses sets, while a real implementation might use a container data structure—when writing the test case, one has to convert from specification terms to implementation terms.

Modelling in the implementation language. The model above used abstract sets. To some extent, it is possible to specify properties using objects in the implementation langauge.

For a filesystem that lives on the disk, this may be difficult: there may not be an in-memory data structure that represents the set of trashed files.

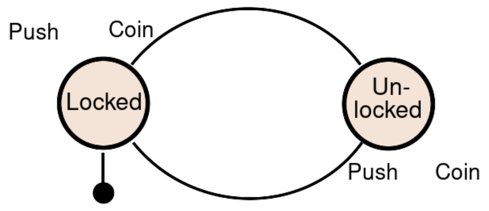
But other times, instead of using sets, the programmer can simply use the program state to specify program properties, and express them using assertions. Preconditions, postconditions, and invariants can also be specified that way, although some conditions are difficult to efficiently verify—for instance, that a linked list is acyclic. Some languages have specialized syntax for specifying preconditions, postconditions, and invariants, but they can be compiled down to assertions inserted in the appropriate places.

[2]`https://practicalalloy.github.io/chapters/behavioral-modeling/index.html`

```
# partially self-verifying appendToList implementation
def appendToList(l,elem):
  l.append(elem)
  assert elem in l
```

Assertions can serve as explicit oracles, in contrast to the implicit oracles we talked about above. You write a test case that arranges and acts, but if your system under test contains assertions, then you have some explicit oracles embedded in the code. Just like the implicit oracles, if an assertion fails, then we know that this is a test case failure. There is a bit more assurance in the event of an assertion success, but I'd feel better about tests derived from model-based specifications.

State Transition Systems. At some level, this is similar to model-based specification, except that there is a finite state machine which describes the overall system state. Wikipedia provides the turnstile example[3]:



One can use the FSM as a test oracle; inserting a coin in a locked turnstile should result in a turnstile in unlocked state. Exercise: write a test case that expresses that idea.

Derived Test Oracles

What about when there are no specifications, or the specifications are insufficient or unhelpful? (As you may have noticed, this is the most common case). Here are some options.

Pseudo-oracles. Consider a straightforward implementation computing Fibonacci numbers imperatively:

```
def fib\_imperative(n):
    a = 1
    b = 1
    next = b
    count = 1
    seq = [1, 1]

    while count <= n:
        count += 1
        a, b = b, next
```

[3]By Chetvorno - Own work, CC0, `https://commons.wikimedia.org/w/index.php?curid=20269475`

```
            next = a + b
            seq.append(next)
    return seq
```

Of course, we can also implement Fibonacci recursively, though it's inefficient.

```
def fib\_recursive(n):
    if n == 0:
        return 0
    elif n == 1 or n == 2:
        return 1
    return fib(n-1) + fib(n-2)
```

This is two versions. (OK, they don't provide exactly the same API.) If they don't match, then at least one of them is wrong.

N-version programming extends that. In any case, we can vote on the most popular answer between different versions to get a "right" answer. It's sort of an oracle, though if all of the versions are programmed to the wrong specification, we still lose.

Regression testing. As mentioned earlier, regression testing uses an earlier version as an oracle. A perfective change in the software may require the right answer to be corrected, because the previous version was wrong. Perhaps the software's specifications have changed.

Textual documentation. Text can also serve as a source of truth. Usually, text requires humans to decipher it and convert it into specifications. Maybe you can use LLMs to read text. I generally trust nothing coming from an LLM.

Specification mining. There is work on automatically deriving specifications from program executions, both in the form of invariants and more general specifications. We won't go into that.

Metamorphic testing. Coming up in the next lecture.

ECE653 note. I am thinking of having an ECE653-only question on the midterm where I ask specific questions which can be answered based on the survey [BHM+15].

References

[BHM+15] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *Transactions on Software Engineering*, 41(5):507–525, May 2015.

[MSH10] Phil McMinn, Mark Stevenson, and Mark Harman. Reducing qualitative human oracle costs associated with automatically generated test data. In *1st International Workshop on Software Test Output Validation*, pages 1–4, 2010.