Software Testing, Quality Assurance & Maintenance—Lecture 5

Patrick Lam
University of Waterloo

January 19, 2026

Part I

The Oracle Problem

What's the right answer?

cop-out: "ask a human"

Begging the Question

Taking the answer the system computes as the right answer.

(is the basis for regression testing, though)

Simple example: add

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

We all agree about the output of `add(1,1)`.

Right?

(Almost)

(What about `add("3", 5)` in JavaScript?)

High school math

Consider function `solve_quadratic()` for

$$x^2 - 2x - 4 = 0.$$

Need to read the function name and remember high school math.

Also, edge cases: no solutions;
floating-point shenanigans.

Human Oracles: source of truth

Unit level: Mainly use the function name, plus any function documentation (if it exists).

More generally: You use your human experience to say what the answer should be.

Part II

Helping Human Oracles

Sometimes there is no alternative

You may have to
ask a human.

Basis for judgment

Does the output meet the system requirements?

(Eliciting requirements not in scope for this course.)

Easy versus hard inputs

Setting: function `calculate_days_between()`.

What's the answer for:

- 12/24/2025 and 12/25/2025?

What about

- -5455/23195/-30879 and -5460/24100/-30800?

Even if we sanitize/declare negative numbers invalid, some inputs are still easier to check.

Input Profiles

Generate inputs that fit expected input profiles:

Start with developers' sanity-check inputs
(like 12/24/2025 and 12/25/2025 etc).

Also, sanitizing checks inside the code are good places to start.

Months: valid months, 0, -1, 13.

Other options

1. Start from normal inputs,
   use genetic algorithms,
   or generate from distributions.

2. Reuse partial inputs, manually modified;
   change one thing at a time,
   thus, easier to reason about changes in the output.

   e.g. go from 0/1/2010 to 1/1/2010, etc.

Integers vs strings

Even though some integers aren't very good (e.g. 23195),
it's still easier to create integers than strings,
and easier to create strings than e.g. trees.

Space of strings is bigger;
space of sensible strings is proportionally smaller.

Can use random strings as fuzzed inputs,
but also want strings that pass sanity checks.

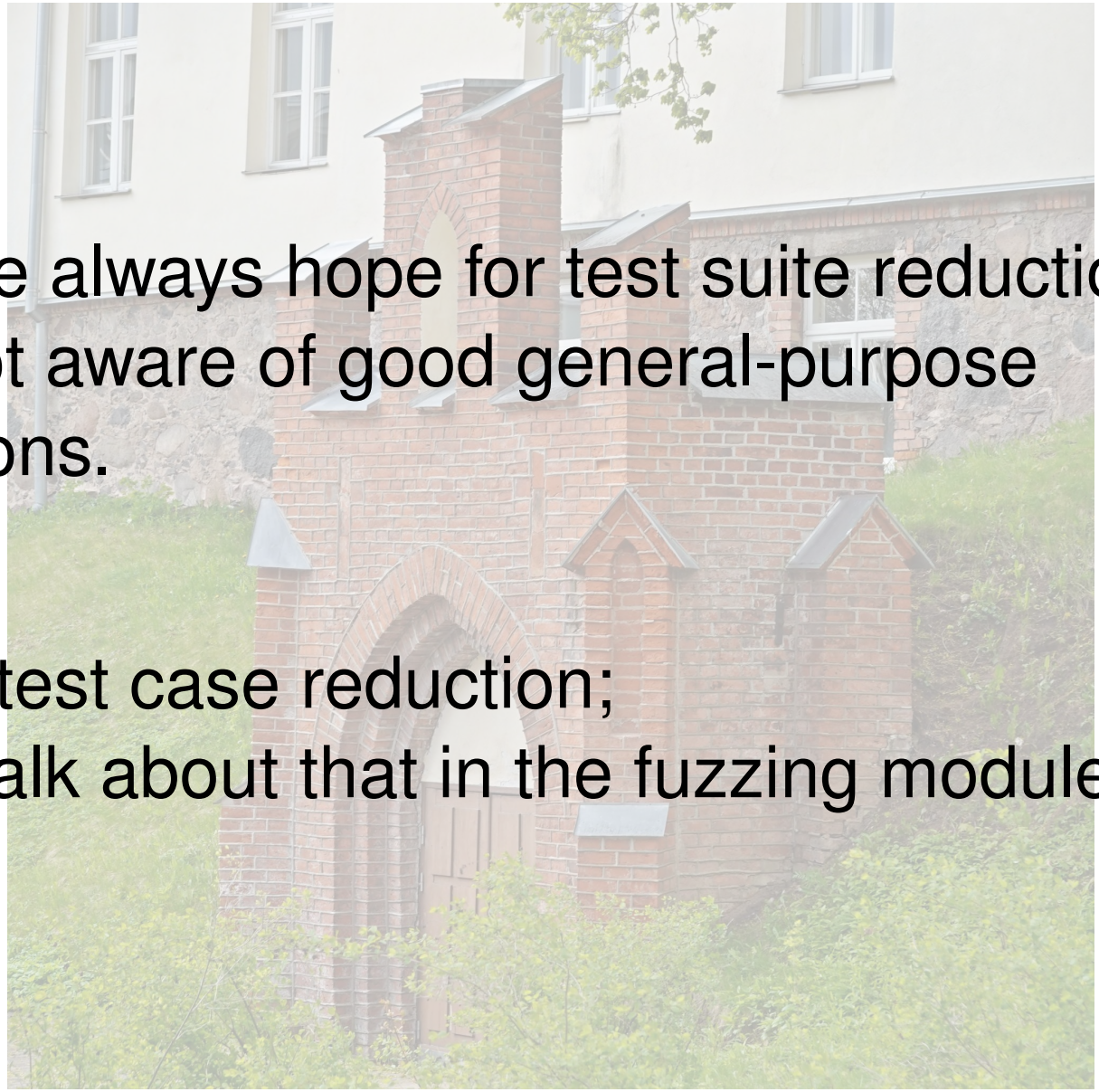
Can mine the web for strings, or generate strings using heuristics (or LLMs).

Crowdsourcing inputs

People have tried to use Mechanical Turk.

Apparently it's hard to get good results.

Reducing volume of work for human testers



People always hope for test suite reduction.
I'm not aware of good general-purpose solutions.

Also: test case reduction;
we'll talk about that in the fuzzing module.

Part III

Implicit Oracles

Segfaults: always bad

```
plam@poneke ~/c/s/n/c/L05 (main)> cat segfault.c
#include <stdlib.h>

int main()
{
  int * q = malloc(5*sizeof(int));
  q[2000000] = 4;
}
plam@poneke ~/c/s/n/c/L05 (main)> gcc segfault.c -o segfault
plam@poneke ~/c/s/n/c/L05 (main)> ./segfault
fish: Job 5, './segfault' terminated by signal SIGSEGV (Address boundary error)
plam@poneke ~/c/s/n/c/L05 (main) [SIGSEGV]>
```

Can always label as bad: Buffer overflows, segfaults, etc.

Might need a tool (e.g. Valgrind, AddressSanitizer) to identify some undefined behaviour.

Don’t need any domain knowledge here.

Also bad (what is “etc”)

- other crashes
  (but: uncaught exception vs
  quiet quitting?)
- race conditions, livelock, deadlock
- memory leaks, performance degradations

Fuzzing & Implicit Oracles

Preview of fuzzing lectures:

because we autogenerate heaps of inputs, we have no way of checking their outputs.

Can only check implicit oracles: no broken assertions, no crashes, use other sanitizers.

Even less evidence for correctness than artisanal tests.

Part IV

Specification-based Oracles

Enabling Verification

What should the implementation do?

One answer: what the specification says.

Specification: a description of [a part of] a system.

Model-based specification

Use modelling languages to describe system behaviour.

e.g. an Alloy action predicate:

```
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
}
```

says what changes in the model when a file is uploaded—acts as an oracle.
System state is modelled by sets or relations.

Using the Model

Can write test cases verifying the specified behaviour:

- what happens if we upload an already-uploaded file?
- when we upload a file, does the file appear in the uploaded set?

Need to translate between the model's sets and the implementation's data structures.

Modelling in the Implementation Language

Instead of a dedicated modelling language, use the program's data structures.

This is back to writing assertions.

Also: preconditions, postconditions, invariants (possibly with specialized syntax e.g. @ensures).

Modelling example

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

Implementation Language Modelling Challenges

e.g. for a disk-based filesystem, can't verify using in-memory info.

some conditions are expensive to verify, e.g. is a linked list acyclic?

Assertions as Explicit Oracles

The conditions embedded in the assertions are explicit oracles (not implicit).

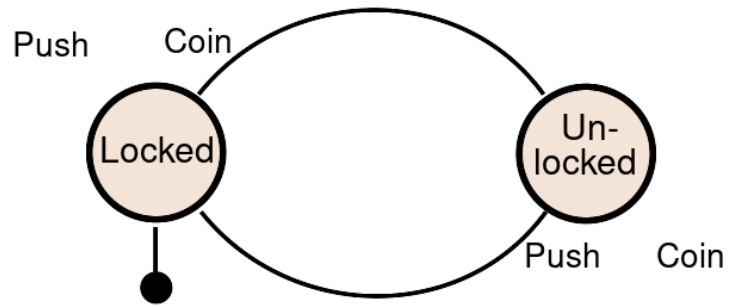
Your test case arranges and acts, and then uses the embedded explicit oracles.

Embedded assertion failure = test case failure (like implicit oracles): a bit more assurance.

Model-based specifications still better.

State Transition Systems

Kind of similar to model-based specification; turnstile example from Wikipedia:



Use the FSM as a test oracle—e.g. inserting a coin in a locked turnstile should result in an unlocked turnstile.

You can write a test to check this.

Part V

Derived Test Oracles

Not Enough Specs

Often, there are no specifications, or the specifications are insufficient or unhelpful.

Derived test oracles of various sorts:

- pseudo-oracles;
- regression testing;
- textual documentation;
- specification mining;
- metamorphic testing.

Pseudo-Oracles

You have to test this imperative implementation:

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

    while count <= n:
        count += 1
        a, b = b, next
        next = a + b
        seq.append(next)
    return seq
```

Recursive fib

The recursive implementation matches the definition of fib closely:

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

Create tests (perhaps automatically) that compare outputs of the two versions.

Generalization: *N*-version programming, vote on the most popular answer.

Sort of an oracle, though if all versions implement the wrong spec, you still lose.

Regression Testing as Pseudo-Oracles

One can consider the earlier version to be a sort of pseudo-oracle.

But a perfective change in the software means that the earlier version was actually wrong, or maybe the specifications have changed.

Textual documentation

Words also serve as a source of truth.

Historically: then deciphered by humans, and converted into specifications.

Specification mining

Lots of work on observing program executions and deriving specifications (invariants and more generally).

Won't talk about that in this course,
but Daikon is one research tool that does this.