Software Testing Part 1

CS350 Introduction to Software Engineering

Shin Yoo

LIKE...
HOW DO I EVEN START?
makeameme.org

Let’s start with the lucrative question(s)

- Not just one, but TWO million dollar question**s** in Software Testing
  - The virtue of knowing when to stop.
  - The wisdom of knowing what is right.

Knowing when to stop.

Software Testing

- The most widely adopted form of dynamic program analysis that is performed to gain confidence in software quality, both functional and non-functional
- Strongly constrained by the inherent limitations of dynamic analysis, but also the most intuitive and easy to adopt (in theory)
- Its mantra is essentially: “believe me, I have tried this, and it works”
    - Strictly speaking, this is better than nothing, but I would also understand if you do not want to trust your life on this dude… right?

How do we know whether a software system is *correct*?

Rationalists vs. Empiricists

“It is correct because I proved that certain errors do not exist in the system”
(Formal Verification)

“It is correct because I tried it several times and it ran okay”
(Software Testing)

How do we know whether a software system is *correct*?

Rationalists vs. Empiricists

static

dynamic

Why do we not entirely trust the dude on the right?

(I mean, Lord Francis Bacon)

- ?

The Famous Dijkstra Quote

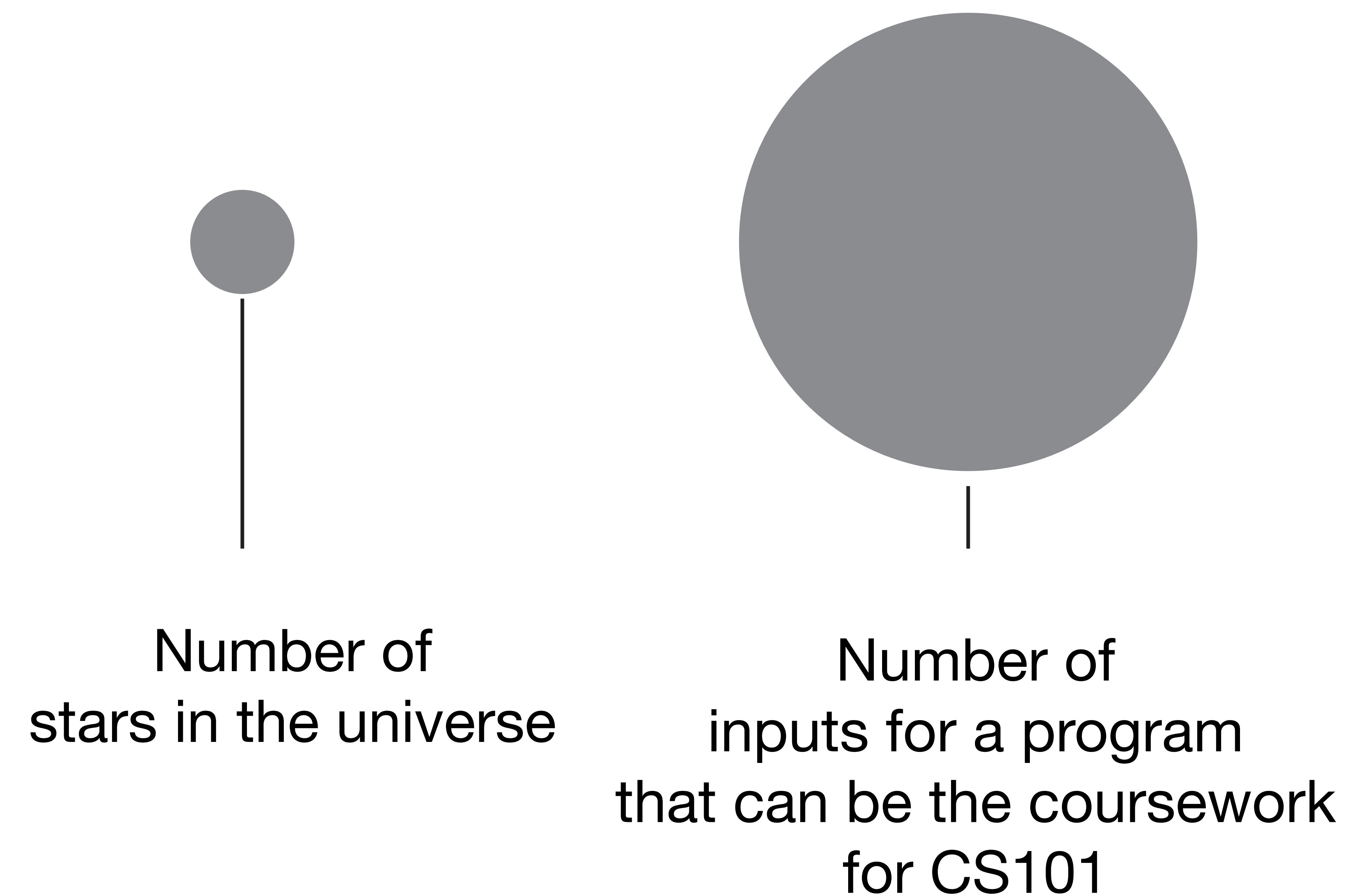
- “Program testing can be used to show the presence of bugs, but never to show their absence.”

Exhaustive Testing

- Can we test each and every program with all possible inputs, and guarantee that it is correct every time? Surely then it IS correct.
- Assuming that the program execution only depends on the input, this is in theory possible: we will observe ALL POSSIBLE EXECUTIONS so we will also detect all failures…. but can we?
- Consider the triangle program
  - Takes three 32bit integers, tells you whether they can form three sides of a triangle, and which type if they do.
  - How many possible inputs are there?

Exhaustive Testing

- 32bit integers: between -2^{31} and $2^{31}-1$, there are 4,294,967,295 numbers
- The program takes three: all possible combination is close to 8^{28}
- Approximated number of stars in the known universe is 10^{24}
- It's going to take some time.



Is program execution finite?

- Any non-trivial program with reasonable interaction with users and environments can result in practically infinite number of possible executions.
- Exhaustive testing is usually infeasible, period.
- Any testing is essentially a **sampling activity**.
    - You may do it manually by writing the tests yourself, or you may do it automatically, but it does not change the fact that you are sampling from this infinite space.
    - The difficulty lies in the fact that we do not really know much about this space of inputs: where are the good ones?

Knowing when to stop

That is, when have I tested enough?

- You cannot execute all possible inputs, because there are too many.
- You can only sample a few that you can practically run.
- But it is under-approximation of program behaviour, so you are never sure - it is always nicer to run more!
- When do I stop? When CAN I stop?

Modelling Fault Detection

- We can try to model the process of sampling test inputs. For example, we can assume that the probability of sampling an input that will reveal a fault in program P to be…
- Is this too much simplification?

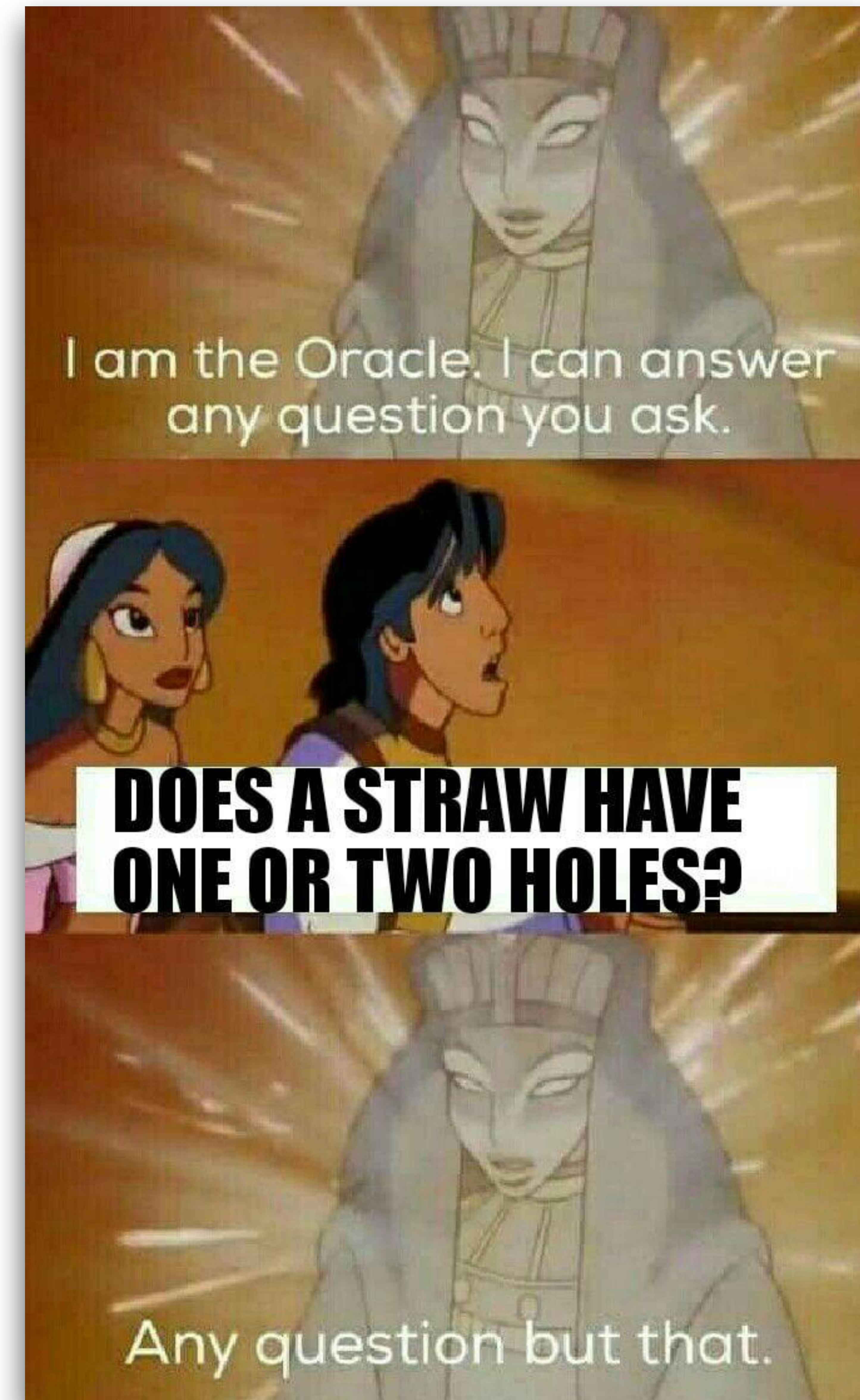
Test Adequacy Metric

- We try to define a tangible and measurable goal of testing, accepting the fact that exhaustive or complete testing is not doable in practice.
- What are such goals that we can set?
  - “Each requirement/use case scenario should have a corresponding test case”
  - “Each branch in the source code should be executed at least once”
- Compromised, yes, but we need to be systematic nonetheless.

Knowing what is right.

Test Oracle

- You have sampled an input.
- The program executes it and produces an output.
- Now you have to check whether the output is as expected.
- The checking mechanism is called a test oracle.
- But how?



Difficulties of Writing Test Oracles

- Ideally, you want something (a program?) that can compute the right answer for any input you choose…. wait… what?
- That something needs to understand the domain, the semantic, and the design.
  - Specification: perhaps your specification is at least computable, if not executable, like logical formulas?
  - Current version: if you want to assume that the current behaviour can be a reference, you can capture it to act as an oracle
  - Human: you think hard about the input, the program, and the specification, and write down the answer.

Implicit vs. Explicit Oracles

- Implicit Oracles: things that should not happen regardless of the semantic and the logic of this specific program
    - Crash, null pointer dereference, infinite loop, etc
- Explicit Oracles: things that should happen due to the specific requirement/business logic of your program
    - Assertions
- Checking for implicit oracles only will not catch all bugs.

A Reliable Oracle

- Oracle should be reliable, if not deterministic: what if the same execution can be sometimes labelled as correct, and other times as incorrect?
- A good test design is
  - A meaningful stimulus (i.e., input given to the program)
  - A reliable observation and judgement
- Reliability becomes harder to achieve as systems become larger: we will discuss this in the next testing lecture (test flakiness).

Let’s revisit the lucrative questions

- Not just one, but TWO million dollar question**s** in Software Testing
  - The virtue of knowing when to stop **(test adequacy and stopping criteria):** a fundamental limitation
  - The wisdom of knowing what is right **(test oracle):** particularly challenging to any effort towards automated testing
- All testing techniques have to prepare their own answer to these.
- All testing research are attempts to answer these questions.

So is it all futile? NO!

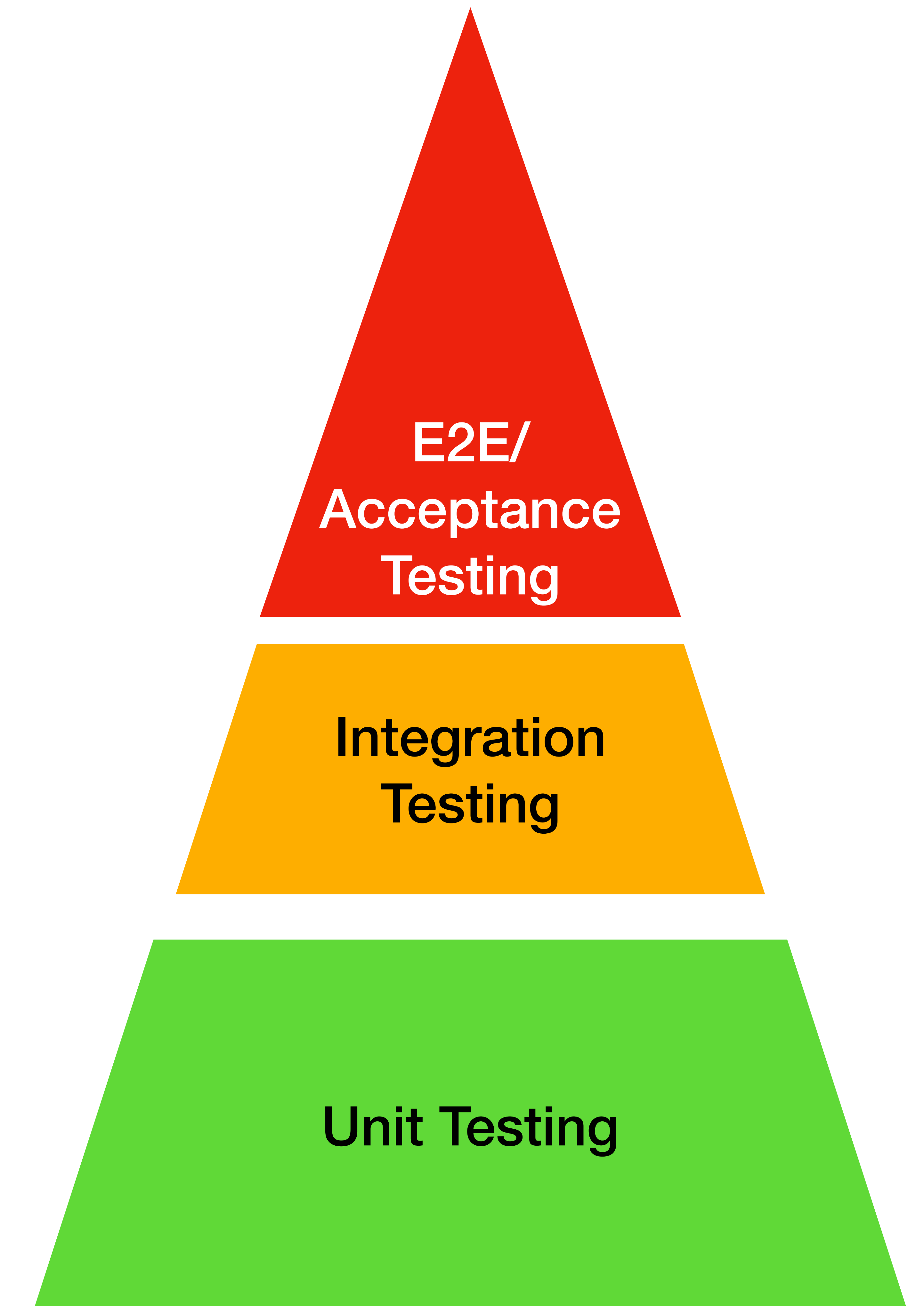
- Despite the difficulties, testing is the most widely adopted validation & verification technique: it plays a central role in contemporary process model.
- If we cannot define a finite set of tests that will detect all faults, we should do it continuously, adapting to changes in the code all the time.
    - You start with a core set of tests that reflect your understanding of the requirements. Later, you will find bugs, for each of which you add a test to ensure that the same thing does not happen again.
- Ideas like TDD takes the oracle problem upside down: if you do not know the oracle, you do not have a spec. Here, test cases become a partial specification. Because it is partial, you keep adding tests whenever you find a hole.
    - Still an active research area because automation is such a huge attraction in testing. Latest trend is of course machine learning…

Some Terminology

- Test Case: test input + test oracle
- Test Suite: a collection of test cases
- Test Harness: a set-up or an environment in which test executions take place
- Test Framework: something that helps you to organise your test cases/test suites, and often also to execute them in batches automatically (JUnit, PyTest, etc)
- Mock/Stub: helper object, written so that you can run test against incomplete system (partial/incomplete simulation of real bejaviour, pre-determined return values, etc)

Types of Testing: Granularity

- Unit testing: targets individual units, such as classes or functions.
- Integration testing: targets interfaces between modules related to the same functionality
- Acceptance testing: targets the entire system combined



Types of Testing: Visibility

- Black Box Testing: you treat the target system as a black box, i.e., no access to the source code - testing depends on observations of inputs and outputs (i.e., functional behaviour)
- White Box Testing: you have access to the source code - you can analyse and even modify the source code to extract information about the target system (e.g., coverage profiling, boundary constant extraction, etc)
- Gray Box Testing: a hybrid (speed of black box + richer information of white box)

Types of Testing: Stages

- Local Testing: the developer runs a few simple tests on a local machine
- Pre-submit Test: the CI/CD pipeline checks an incoming commit by running a (reduced) test suite - the result is either part of the code review, or actually a pre-requisite for the code review.
  - Pre-submit tests need to be simple and fast (e.g., unit level)
- Post-submit Test: a system after merge is tested.
  - Post-submit tests need to be thorough (e.g., integration and acceptance tests)

Types of Testing: Objectives

- Structural Testing: aims to achieve coverage (i.e., to execute as much code as possible, to maximise the chance of detecting faults at arbitrary locations)
- Functional Testing: to check functional behaviour against specification, can combined with structural testing
- Stress Testing: aims to test the capability of the system to retain performance under heavy workloads (this is a non-functional property)
- Usability Testing: observes how easy users find it to use the system
- Reliability Testing: aims to test whether the system can perform reliably for a long period of time
- …

A few basic concepts

Exercise

- Recall that triangle program: it takes three integers, which are interpreted to represent lengths of three sides of a triangle. The program will return whether they form scalene (all three sides and angles are different), isosceles (two sides and two angles are identical), or equilateral (all three sides and angles are the same) triangles.
- Write down test cases for this program :)

Equivalence Class

- We can group inputs based on the program behaviour against them.
  - For example, there are certain inputs that we expect to get the “equilateral” result.
  - In principle, it should be okay to sample one from each of such “classes”
- We can base our definition of equivalence on the specification.
  - However, will the source code have actually implemented the same classes?

Boundary Value Analysis

- Perhaps we need to specifically sample values that are near the boundaries between equivalence classes: may catch one-off errors in boundary conditions.
- Note that this still does not guarantee complete detection of class boundary errors…

Next Lecture

- We will look at individual techniques/research agendas in more detail.