IS2545, LECTURE 2:
TESTING THEORY AND TERMINOLOGY

KEY (🔑) CONCEPT TO THE COURSE

EXPECTED BEHAVIOR VS OBSERVED BEHAVIOR

EXPECTED BEHAVIOR VS OBSERVED BEHAVIOR

You need to know what “should” happen under some circumstances, then check to see if that behavior actually occurred.

For example, assume I have a function foo, which accepts an integer, a, and returns a float. What should happen if I send in the value a = 42?

This is a simple idea, but it’s the “Fundamental Theorem of Testing” (although note that we may violate it later…)

EXAMPLE

Assume foo is supposed to return the square root of the passed in value a.

When I send in the value a = 42, then I expect to be returned the value 6.48074069841.

When I send in the value a = 9, then I expect to be returned the value 3.

When I send in the value a = -1, then I expect….

THE IMPOSSIBILITY OF EXHAUSTIVE TESTING

- Let's say we want to ensure that our square root method will never fail, no matter what we send in. Assume we are using a standard Java int (signed 32-bit integer)
- How many values do we have to test?

4,294,967,296

WHAT ABOUT A MEDIUM-SIZED, 1000-METHOD JAVA PROGRAM?

- Assume that each method has one int argument and returns one primitive value.
- Remember that methods in a java-like language could theoretically influence other methods (e.g., setting global variables, calling other methods, mutating objects, etc.)

4,294,967,296 ^ 1000

WOULD HAVING THAT MANY TESTS GUARANTEE THAT THERE ARE NO PROBLEMS WITH THE SYSTEM UNDER TEST?

LOL NOPE

- Data races?
- Compiler issues?
- Non-functional issues (performance, usability, etc.)?
- Floating-point issues?
- Integration issues?
- Systems-level issues?
- Ambiguous or misunderstood requirements?

TESTING = ART + SCIENCE

- There are techniques for testing which can reduce the number of tests necessary for sufficient test coverage.
- We will need to define what we mean by “sufficient test coverage”.
- We will also require domain knowledge.

EQUIVALENCE CLASS PARTITIONING

- We can partition the testing parameters into "equivalence classes"
  - Equivalence class = a natural grouping of values with similar behavior
- For example, in our square root method:
  - Negative numbers (input) -> Imaginary numbers (output)
  - 0 -> 0
  - Positive numbers -> Positive numbers

EQUIVALENCE CLASSES ARE STRICTLY PARTITIONED

- For any given input value, it must belong to one and ONLY one equivalence class (strictly partitioned)
  - If there are values that seem like they belong in multiple equivalence classes, you either need:
    - Multiple partitionings
    - Another equivalence class

EXAMPLE

- Assume you have a program which will return the square root of an int, and if the number is whole (e.g., 1 or 2, but not 1.342), it should print it out in red, otherwise it will print it out in black.
- You can have two partitionings:
  - (the positive/0/negative partitioning on the previous slide)
  - Another partitioning:
    - Number is whole -> output printed in red
    - Number is not whole -> output printed in black
- Therefore, for every value, there are multiple partitionings to check

THEY DO NOT HAVE TO BE NUMERIC

- On Twitter, if you follow somebody, you see all of their tweets, unless they are writing directly to somebody you do not follow.
- Equivalence classes:
  - You do not follow person A -> DO NOT see the tweet
  - You do follow person A, they are not writing directly to somebody -> see the tweet
  - You do follow person A, they are writing directly to person B, whom you also follow -> see the tweet
  - You do follow person A, they are writing directly to person B, whom do you not follow -> DO NOT see tweet

TEST EACH EQUIVALENCE CLASS

- Pick at least one value from each equivalence class
- This will ensure you capture behavior from each “class” of possible behavior
- Will find a good percentage of defects without exhaustive testing!
- We reduced the problem something a human can do! Woo-hoo!
- How to pick the input? Well, that is part of the art.
  - However, there are some good guidelines!

INTERIOR AND BOUNDARY VALUES

- Theory: Problems are more prevalent on the boundaries of equivalence classes than in the middle.

WHY?

```
public boolean canBePresidentOfUnitedStates(int age) {
    return age > 35;
}
```

EQUIVALENCE CLASS PARTITIONING

CANNOT_BE_PRESIDENT = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34]

CAN_BE_PRESIDENT = [35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64….INFINITY]

WHERE ARE PROBLEMS LIKELY?

CANNOT_BE_PRESIDENT = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34]

CAN_BE_PRESIDENT = [35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64….INFINITY]

TRY TO ENSURE THAT YOU TEST BOUNDARY AND INTERIOR VALUES

CANNOT_BE_PRESIDENT = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34]

CAN_BE_PRESIDENT = [35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64....INFINITY]

- Are we missing anything?

“HIDDEN” (IMPLICIT) BOUNDARY VALUES

- The boundary values we have gone over already are explicit – that is, they are defined, or at least able to be deduced from, the requirements of the problem itself.
- Some boundaries are implicit – they are generated from the domain, architecture, hardware, or other elements:
  - MAXINT, MININT
  - Maximum precision of a floating point value
  - Allocation limitation (memory, hard drive space, network bandwidth, etc.)
  - Undefined values

BASE, EDGE, AND CORNER CASES

- Base case – An element in an equivalence class that is not around a boundary (interior value), OR an expected use case.
- Edge case – An element in an equivalence class that is next to a boundary (boundary value), OR an unexpected use case.
- Corner case (or pathological case) – A case which can only occur outside of normal operating parameters, or a combination of multiple edge cases.

BLACK-, WHITE, AND GREY-BOX TESTING

- Black-box testing: Testing with no knowledge of the interior structure or code of the application. Tests are often performed from the user's perspective, looking at the system as a whole.
- White-box testing: Testing with explicit knowledge of the interior structure and codebase, and directly testing that code. Tests are often at a lower level (e.g., testing individual methods or classes)
- Grey-box testing: Testing with knowledge of the interior structure and codebase of the system under test, but not directly testing the code. Tests are similar to black-box tests, but are informed by the tester's knowledge of the codebase.

BLACK-BOX TESTING EXAMPLES

- Accessing a website, using a browser, to look for flaws
- Running a script against an API endpoint
- Checking to see that changing fonts in a word processor shows the correct font

WHITE-BOX TESTING EXAMPLES

- Testing that a function returns the correct result
- Testing that instantiating an object creates a valid object
- Checking that there are no unused variables in a method

GREY-BOX TESTING EXAMPLES

- Reviewing code, and noticing that bubble sort is used. Then write a user-facing test involving a large input size.
- Reviewing code and noticing an off-by-one error. Then write a user-facing test which checks that boundary value.

STATIC VS DYNAMIC TESTING

- Dynamic testing = code is executed (at least some of it)
- Static testing = code is not executed

DYNAMIC TESTING

- If you’re thinking about testing, this is probably what you are thinking about.
    - Code is executed under certain circumstances (e.g. input values, environment variables, etc.)
    - Observed results are then compared with expected results
- The majority of the class will consists of dynamic testing
- Much more commonly used in industry

STATIC TESTING

- The code is reviewed by a person or external program, without being executed
- Examples:
  - Code walkthroughs and reviews
  - Requirements analysis
  - Source Code Analysis
    - Linting
    - Model checking
    - Complexity analysis
    - Code coverage
    - Finite state analysis
    - … COMPILING!

SEE YOU NEXT TIME