


CS1632, LECTURE 2: TESTING THEORY AND TERMINOLOGY

Wonsun Ahn

Key () concept to
the course

Expected behavior vs observed behavior

Expected vs. Observed Behavior

- *Expected behavior*: What “should” happen
- *Observed behavior*: What “does” happen
- *Testing*: comparing expected and observed behavior
- *Defect*: when expected \neq observed behavior
- Expected behavior is also known as *requirement*

Example

- Suppose we are testing a function `sqrt`:
`// returns the square root of num`
`float sqrt(int num) { ... }`
- When I call `sqrt` with argument 42 ,
`float ret = sqrt(42);`
Expected behavior: `ret == 6.48074069841`
- When `float ret = sqrt(9);` ,
Expected behavior: `ret == 3`
- When `float ret = sqrt(-9);` ,
Mathematically, square root of `-9` can't be a real number,
but requirements should still specify some behavior

THE IMPOSSIBILITY OF EXHAUSTIVE TESTING

- Let's say we want to ensure that `sqrt` is defect-free for all arguments (both positive and negative)
- Assume `arg` is a Java `int` (signed 32-bit integer)
- How many values do we have to test?

4,294,967,296

What if there are two arguments?

- Suppose we are testing a function `add`:
`// return the sum of x and y`
`int add(int x, int y) { ... }`
- How many tests do we have to perform?
(Hint: all combinations of `x` and `y`)

4,294,967,296 \wedge 2

What if the argument is an array?

- Suppose we are testing a function `add`:
`// return sum of elements in A`
`int add(int[] A) { ... }`
- How many tests do we have to perform?
(Note: array `A` can be arbitrarily long)

4,294,967,296 \wedge Infinity

Would testing all the combinations of arguments guarantee that there are no problems?

LOL NOPE

- Compiler issues
- Parallel programming issues (e.g. data races)
- Non-functional issues (e.g. performance)
- Floating-point issues (e.g. loss of precision)
- Systems-level issues (e.g. OS/device-dependent defect)
- Misunderstood requirements

Compiler Issues

- The compiled binary, not your source code, runs on the computer
- What if compiler has a bug? (Rare)
- What if compiler *exposes* a bug in your program? (More frequent)

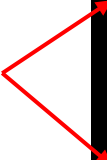
```
int add_up_to (int count) {  
    int sum, i;  /* some C compilers will init sum to 0, others will not */  
    for(i = 0; i <= count; i++) sum = sum + i;  
    return sum;  
}
```

- ☞ Code will work with some compilers but not with others
- You can avoid this issue by using the same compiler with the same compiler options, but sometimes that is not feasible

Parallel programming issues

```
class Main implements Runnable {  
    public static int count = 0;  
    public void run() {  
        for(int i=0; i < 1000000; i++) { count++; }  
        System.out.println("count = " + count);  
    }  
    public static void main(String[] args) {  
        Main m = new Main();  
        Thread t1 = new Thread(m);  
        Thread t2 = new Thread(m);  
        t1.start();  
        t2.start();  
    }  
}
```

Why?



```
$ javac Main.java  
$ java Main  
count = 1868180  
count = 1868180  
$ java Main  
count = 1033139  
count = 1033139
```

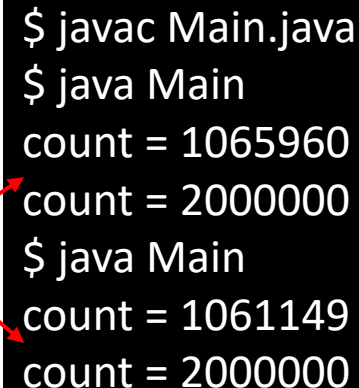
Parallel programming issues

- Why does this happen?
 - Threads t_1 and t_2 run on separate CPUs
 - Two threads try to increment `count` at the same time
 - Often, they step on each other's toes (a data race)
- If there is a data race, result is undefined
 - Java language specifications say so!
 - Every time you run it, you may get a different result
 - Passing a test once does not guarantee correctness
- Worst part: often, result is correct 99% of the time
 - ☞ Must test thousands of times to find defect

Parallel programming issues

```
class Main implements Runnable {  
    public static int count = 0;  
    public void run() {  
        for(int i=0; i < 1000000; i++)  
            synchronized(this) { count++; }  
        System.out.println("count = " + count);  
    }  
    public static void main(String[] args) {  
        Main m = new Main();  
        Thread t1 = new Thread(m);  
        Thread t2 = new Thread(m);  
        t1.start();  
        t2.start();  
    }  
}
```

Solved?



```
$ javac Main.java  
$ java Main  
count = 1065960  
count = 2000000  
$ java Main  
count = 1061149  
count = 2000000
```


Parallel programming issues

- `synchronized` removes the data race
 - Now `count = 2000000` in the end, as expected
 - How?
 - `synchronized` “locks” the code region while incrementing `count` so that other thread can’t interfere
 - But note that value of intermediate `count` is still nondeterministic. Why?
 - Speed of threads t_1 and t_2 are nondeterministic
- 👉 Data-race-free programs can still pose problems

For the purposes of this course...

- Let's ignore these issues for now
 - Compiler issues
 - Parallel programming issues
 - Non-functional issues
 - Floating-point issues
 - Systems-level issues
- Testing a sequential program using a single compiler on a single device is hard enough due to the test explosion problem
- Test explosion problem is what we will focus on

Testing = ART + SCIENCE

- Exhaustive testing is impossible
- Goal: achieve “sufficient” test coverage
- How would you define test coverage?
 - Ideally: $\text{defects_found} / \text{total_defects}$
 - But is there a way to measure total_defects ?
(Hint: if you knew all the defects, you wouldn't be testing)
 - So we need a proxy metric for test coverage
(e.g. $\text{lines_of_code_tested} / \text{total_lines_of_code}$)
- Deciding “sufficient” using this inexact proxy is an art
 - In the end, must rely on domain knowledge to decide

Improving Test Coverage

- QA engineers have a limited testing time budget
 - Must choose a subset of tests maximizing test coverage
- Q) Which test maximizes test coverage?
 - Obviously, one that uncovers a new defect
(But as we pointed out, that is impossible to know)
- Q) Which test is likely to uncover a new defect?
 - One that exercises new behavior in the code
 - This is the idea behind *equivalence class partitioning*

Equivalence Class Partitioning

- Partition the input values into “equivalence classes”
 - Equivalence class = group of values with similar behavior
- E.g., equivalence classes for our `sqrt` method:
{positive numbers, 0, negative numbers}
- Behavior for each class:
 - positive numbers: returns a positive number
 - 0: returns 0
 - negative numbers: returns an imaginary number

Equivalence Classes should be *Strictly* Partitioned

- *Strictly*: a value belongs to one and ONLY one class
- If an input value belongs to multiple classes
 - Means you expect input to have two different behaviors
 - Either there is a bug in the requirements or you misunderstood it
- If an input value belongs to no class
 - Means this input does not match any pre-existing behavior
 - Add a new equivalence class for this input

Values do not have to be numeric

- For a spell checker, input values are strings
- Equivalence classes:
{strings_in_dictionary, strings_not_in_dictionary}
- Behaviors:
 - strings_in_dictionary: do nothing
 - strings_not_in_dictionary: red underline string

Values do not have to be numeric

- Input values can be tuna cans
- Equivalence classes:
 {not_expired,
 expired_but_not_smelly,
 expired_and_smelly}
- Behaviors:
 - not_expired: eat
 - expired_but_not_smelly: first feed it to your cat
 - expired_and_smelly: discard

Test Each Equivalence Class

- Pick at least one value from each equivalence class
- Ensures you cover all behavior expected of program
- Gets you good coverage without exhaustive testing!
- How to pick the value? Well, that is part of the art.
 - However, there are some good guidelines!

Interior and boundary values

- Empirical truth:
 - Defects are more prevalent at boundaries of equivalence classes than in the middle.
- Why?
 - Due to the prevalence of off-by-one errors

Off-by-one Error

- Suppose expected behavior is:
 - Method shall take the age of a person as argument
 - Method shall determine whether person can be US president
 - Rule: Person must be 35 years or older to be US president
- Suppose code implementation is:

```
boolean canBePresident(int age) {  
    return age > 35;  
}
```
- Is observed behavior the same as expected behavior?

Equivalence class partitioning

CANNOT_BE_PRESIDENT =
[...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...]

Try to test boundary values

CANNOT_BE_PRESIDENT =
[...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...]

- Test boundary values (shown in **red**)
- In fact, there is a bug at: `age > 35`

Also test interior values

CANNOT_BE_PRESIDENT =
[...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...]

- Testing interior values (shown in green) is also important to see behavior at the interior

Are we done?

CANNOT_BE_PRESIDENT =
[...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...]

- Input values so far: {26, 30, 34, 35, 39, 42}

“Hidden” (IMPLICIT) boundary values

- Boundary values we’ve added so far are explicit – that is, they are defined by requirements
- Some boundaries are implicit – they are generated from the language, hardware, domain, etc.:
 - Language boundaries:
MAXINT, MININT
 - Hardware boundaries:
memory space, hard drive space, etc.
 - Domain boundaries:
weight can’t be negative, score can’t exceed 100, etc.

Add implicit boundary values

CANNOT_BE_PRESIDENT =
[**MININT**,...-2,-**1**,**0**,1,...,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,...,**MAXINT**]

- **MININT**, **MAXINT**: language boundaries
- -**1**, **0**: domain boundaries (age can't be negative)
- Inputs: {**MININT**, -**1**, **0**, **26**, **30**, **34**, **35**, **39**, **42**, **MAXINT**}

Finding the Off-by-one Error

- Now let's feed these inputs to our code:

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

Inputs: {MININT, -1, 0, 26, 30, 34, 35, 39, 42, MAXINT}

- Remember, expected behavior was:
 - Person must be 35 years or older to be US president
- A defect is found with input 35:
 - Expected behavior: Can be president
 - Observed behavior: Cannot be president

Base, edge, and corner cases

- **Base case:** An interior value, OR an expected use case
- **Edge case:** A boundary value, OR an unexpected use case
- **Corner case (or pathological case):**
Value outside of normal operating parameters, OR
multiple edge cases happening simultaneously

Base, edge, and corner cases: Example

- Suppose a cat scale has these operating envelopes:
 - Weight between 0 – 100 lbs
 - Temperature between 0 – 120 F
- Base cases: (10 lbs, 60 F), (20 lbs, 70 F), ...
- Edge cases: (**100 lbs**, 70 F), (10 lbs, **0 F**), ...
- Corner cases: (**300 lbs**, 70 F), (**100 lbs**, **120 F**), ...
- Why test corner cases?
 - Even if scale isn't expected to operate correctly for 300 lbs, user still cares what happens (i.e. does it break the scale?)

Boundary values from source code

- So far, boundary values were found external to source code
 - Explicit boundaries: from requirements
 - Implicit boundaries: from knowledge of language, domain, ...
- Suppose implementation for `canBePresident` was:

```
boolean canBePresident(int age) {  
    return age >= 35 && age <= 65;  
}
```
- Age limit of 65 is not in the requirements but it is in code
 - Wouldn't you want to test it?
 - Doing this is called *white box testing*

Black-, white, and grey-box testing

- **Black-box testing:**
 - Testing with no knowledge of interior structure or source code
 - Tests are performed from the user's perspective
 - Can be performed by lay people who don't know how to program
- **White-box testing:**
 - Testing with explicit knowledge of the interior structure and codebase
 - Tests are performed from the developer's perspective
 - Test inputs are crafted to exercise specific lines of code
 - Testing may involve explicitly calling specific methods (unit testing)
- **Grey-box testing:**
 - Testing with some knowledge of the interior structure and codebase
 - Knowledge comes from partial code inspection or a design document
 - Performed from the user's perspective, but informed by knowledge

Black-box testing examples

- Testing a website using a web browser
- Testing a game by actually playing it
- Testing a script against an API endpoint
- Any type of beta test

White-box testing examples

- Crafting input to exercise specific program paths
 - Choosing a boundary value not present in the requirements but present in the source code
 - Intentionally choosing inputs that cause exceptions (and observing whether handled correctly)
- Explicitly calling methods from a testing script
 - Testing that a function returns the correct result
 - Testing that instantiating a class creates a valid object

Grey-box testing examples

- *Reviewing code* and noticing that bubble sort is used. Then writing a *user-facing test* involving a large input.
- *Reviewing code* in a web app and noticing user input is not properly sanitized. Then writing a *user-facing test* which attempts SQL code injection.
- *Reading a design document* and noticing a network connection through which lots of data passes. Then writing a *user-facing test* that stresses that connection.

Static vs dynamic testing

- We talked a great deal about choosing good inputs
 - But is this all there is to testing?
- Dynamic testing = code is executed
 - Relies on good inputs for good coverage
- Static testing = code is not executed
 - There are no inputs since code is not executed
 - Relies on analyzing the code to find defects

Dynamic testing

- What we have been talking about so far
 - Code is executed under certain circumstances (e.g. input values, compiler, OS, runtime library, etc.)
 - **Observed results** are compared with **expected results**
- More commonly used in industry
 - Programmers know to do this even w/o being taught
 - Can be done w/o special tools or training
 - Majority of the class will be about dynamic testing

Static testing

- Code is analyzed by a person or testing tool
- Examples:
 - Code walkthroughs and reviews by a person
 - Code analysis using a tool
 - Linting
 - Model checking
 - Complexity analysis
 - Code coverage
 - Finite state analysis
 - ... COMPILING!

Now Please Read Textbook Chapters 2-4