

# CS1632: TESTING THEORY AND TERMINOLOGY

Wonsun Ahn

# Expected vs. Observed Behavior

- **Expected behavior:** What “should” happen
- **Observed behavior:** What “does” happen
- **Defect:** when expected  $\neq$  observed behavior
- **Testing:** checking expected  $==$  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 `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 ^ Infinity

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

# LOL NOPE

- Issues not covered by exhaustive input testing
  - Compiler issues
  - Systems-level issues (e.g. OS/device-dependent defect)
  - Parallel programming issues (e.g. data races)
- The same input must be tested multiple times
  - On different compilers, OSes, devices, ...
  - (Potentially) many times on same compiler / OS / device

# 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;          /* Is sum == 0? Not necessarily! */  
    for(i = 0; i <= count; i++) sum = sum + i;  
    return sum;  
}
```

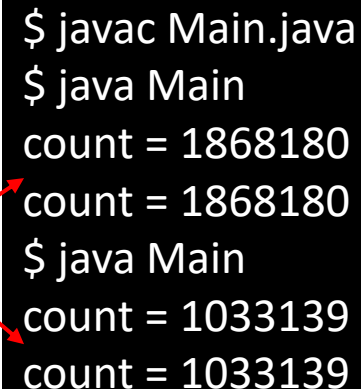
- Behavior is undefined according to C language specifications
- Compiler can generate code with arbitrary behavior

☞ Need to exhaustively verify with all compilers and compiler options!

# 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  $t_2$  execute concurrently
    - 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
    - Result depends on relative speed of threads  $t_1$  and  $t_2$
- ☞ Running 1000+ times may not cover all behavior

# For the purposes of this Chapter...

- Let's ignore these issues for now
  - Combinatorial testing issues
  - Compiler issues
  - Systems-level issues
  - Parallel programming issues
- Exhaustive input value testing is hard enough
  - a.k.a. “test explosion problem”
  - This is what we will focus on in this chapter
- We will address the other issues later 😊

# Equivalence Classes

Achieving Test Coverage Efficiently



# Defining Test Coverage

- Goal of testing: achieve good test coverage
  - **Test coverage**: measure of how well code has been tested
  - Ideally,  $test\_coverage = defects\_found / total\_defects$
- But is there a way to measure *total\_defects*?
  - If we knew, we wouldn't need to do any testing!
  - Impossible to measure true test coverage
- Is there a good proxy that estimates true test coverage?
  - **Statement coverage** =  $statements\_tested / total\_statements$
  - Rationale: if a high percentage of statements are tested  
👉 likely that a high percentage of defects are found
  - Other proxies out there: method coverage, path coverage, ...

# Improving Test Coverage

- QA engineers have a limited testing time budget
  - Since true test coverage is impossible to measure, must choose tests maximizing proxy coverage metric
  - Most commonly, maximizing statement coverage
- Which tests are likely to maximize coverage?
  - Tests that exercise all required program behaviors
  - If tests exercise only one specific program behavior → likely to have low statement coverage
  - This is the idea behind *equivalence class partitioning*

# Equivalence Class Partitioning

- Partition the input values into “equivalence classes”
  - Equivalence class = group of values with same behavior
- E.g. equivalence classes for our `sqrt` method:  
*{nonnegative\_numbers, negative\_numbers}*
- Behavior for each equivalence class:
  - *nonnegative\_numbers*: returns square root of number
  - *negative\_numbers*: returns NaN (not a number)

# Equivalence Classes should be *Strictly* Partitioned

- **Strictly:** each value belongs to one and only one class
- If an input value belongs to multiple classes
  - Means requirements specify two different behaviors for the input
  - Either requirements are inconsistent, or you misunderstood them
- If an input value belongs to no class
  - Means requirements do not specify a behavior for the input
  - Either requirements are incomplete, or you misunderstood them

# Values can be Strings

- For a spell checker, input values are strings
- Equivalence classes:  
 $\{strings\_in\_dictionary, strings\_not\_in\_dictionary\}$
- Behavior for each equivalence class:
  - *strings\_in\_dictionary*: do nothing
  - *strings\_not\_in\_dictionary*: red underline string

# Values can also be Objects

- Input values can be tuna cans
- Equivalence classes:  
*{not\_expired, expired\_and\_not\_smelly, expired\_and\_smelly}*
- Behavior for each equivalence class:
  - *not\_expired*: eat
  - *expired\_and\_not\_smelly*: feed it to your cat (kidding)
  - *expired\_and\_smelly*: discard

# Testing 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 many values should I pick? And what values?
  - There is no exact science.
  - But there are some good empirical guidelines!

# Defects are Prevalent at Boundaries

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



# Defects are Prevalent at Boundaries

- Suppose requirements are:
  - Age shall be given as commandline argument
  - If age is **35 years or older**, system shall print “can be US president”.
  - Otherwise, system shall print “cannot be US president”.
- Suppose code implementation is:

```
public static void main(String[] args) {  
    int age = Integer.parseInt(args[0]);  
    if (age > 35) { // See the off-by-one error?  
        System.out.println("can be US president");  
    } else {  
        System.out.println("cannot be US president");  
    }  
}
```

# 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...]

# Always 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...]

- Always test boundary values (shown in **red**).
- Will catch the off-by-one error at **35**: `age > 35`

## Also Test a few 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 (in green) is also important.
- Who knows? Code behavior may change in the middle.

# 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...]

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

# Implicit (hidden) boundary values

- **Explicit boundaries:** specified in requirements
  - Boundaries that are defined by equivalence classes
- **Implicit boundaries:** not in the requirements
  - “Naturally” occurring in language, hardware, domain
  - Language boundaries: MAXINT, MININT
  - Hardware boundaries: memory space, hard drive space
  - Domain boundaries:  $\text{weight} \geq 0$ ,  $0 \leq \text{score} \leq 100$ , etc.
- We need to check implicit boundaries as well. Why?

# Implicit boundaries should not change behavior

- Why do we check **explicit boundaries**?
  - To verify behavior **changes** when boundary is crossed.
- Why do we check **implicit boundaries**?
  - To verify behavior **does not change** on the boundary.
- Checking correct handling of implicit boundaries:
  - Crossing MAXINT boundary for input value
    - Handle larger numbers gracefully with no int overflow
  - Crossing memory space limit due to large input
    - Handle gracefully (possibly by moving data to disk)

# Add implicit boundary values

CANNOT\_BE\_PRESIDENT =

[**MININT-1**, **MININT**, ..., **-1**, **0**, **1**, ..., **26**, **27**, **28**, **29**, **30**, **31**, **32**, **33**, **34**]

CAN\_BE\_PRESIDENT =

[**35**, **36**, **37**, **38**, **39**, **40**, **41**, **42**, **43**, **44**, ..., **MAXINT**, **MAXINT+1**]

- language boundaries: **MININT** <= age <= **MAXINT**
- domain boundaries: age >= **0** (age is non-negative)
- Inputs: {**MININT-1**, **MININT**, **-1**, **0**, **26**, **30**, **34**, **35**, **39**, **42**, **MAXINT**, **MAXINT+1**}



# Finding Defects using our Inputs

- Now, let's feed the inputs to our code:  
{MININT-1, MININT, -1, 0, 26, 30, 34, 35, 39, 42, MAXINT, MAXINT+1}

```
public static void main(String[] args) {  
    int age = Integer.parseInt(args[0]);  
    if (age > 35) { // The off-by-one error.  
        System.out.println("can be US president");  
    } else {  
        System.out.println("cannot be US president");  
    }  
}
```

- Which inputs would find defects?

# Finding Defects using our Inputs

- Defect 1: off-by-one-error found with input **35**:
  - Expected behavior: prints “can be US president”
  - Observed behavior: prints “cannot be US president”
- Defect 2: int overflow error found with input **MAXINT+1**:
  - Expected behavior: prints “can be US president”
  - Observed behavior: throws java.lang.NumberFormatException
- Defect 3: int underflow error found with input **MININT-1**:
  - Expected behavior: prints “cannot be US president”
  - Observed behavior: throws java.lang.NumberFormatException

# Base, edge, and corner cases

- **Base case:** A typical use case
  - Interior value of equivalence class for normal operation
- **Edge case:** A use case at the limit of allowed use
  - Boundary value of equivalence class for normal operation
- **Corner case (or pathological case):**
  - Value far 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 300 lbs?
  - Even if scale isn't expected to operate correctly for 300 lbs, user still cares what happens (i.e. does it break the scale?)

Categories of Testing:  
Black / White / Gray  
Dynamic / Static

# Black-, white, and gray-box testing

- **Black-box testing:**

- Testing with **no knowledge** of interior structure 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

- **Gray-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

- Tests are performed using only UI
- 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
  - Penetration testing on a website

# White-box testing examples

- Tests are performed by both...
  - Using UI to exercise specific program paths
  - Explicitly calling methods from a testing script
- Examples
  - Choosing inputs to exercise specific parts of an algorithm
  - Choosing inputs causing exceptions and checking handling
  - Testing that a method call returns the correct result
  - Testing that instantiating a class creates a valid object
- Unlike black-box, can measure **statement coverage**



# 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

- Code is executed under various test scenarios
  - Varying input values, compilers, OSes, etc.
  - **Observed results** are compared with **expected results**
  - Hard to achieve 100% test coverage
- Examples:
  - Manual testing
  - Unit testing
  - System testing
  - Performance testing

# Static testing

- Code is analyzed by a person or testing tool
  - While checking whether correctness rules are followed
  - 100% test coverage achieved for all code analyzed
  - Even when check passes, defects can still occur at runtime
- Examples:
  - Code reviews by a person
  - Code analysis using a tool
    - Compilers
    - Linters
    - Bug pattern finders
    - Code coverage analysis
    - Model checkers

Now Please Read Textbook Chapters 2-4