CS1632, LECTURE 2: TESTING THEORY AND TERMINOLOGY

Wonsun Ahn

Key (🔑) concept to the course

Expected behavior vs observed behavior

Expected 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 if there are 1000 method calls?

- Assume each method accepts a 32-bit int argument
- Remember: methods in a Java-like language can have side-effects (not pure functions)
  - Can access global variables, global data structures
  - Mean a method call is affected by all previous calls

4,294,967,296 ^ 1000

What if argument is an object reference?

- That object could be a list data structure, a tree, a graph, …
- How many shapes can a tree take?

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

LOL NOPE

- Compiler issues
- Parallel programming issues
- Non-functional issues (performance, usability, …)
- Floating-point issues
- Integration issues
- Systems-level issues
- Ambiguous or misunderstood requirements

LOL NOPE

- Compiler issues
  - The compiled binary, not your source code, runs on the computer
  - Depending on compiler and compile options, many different binaries can be produced! Besides the binary you used for testing.
  - What if the compiler has a bug? (Rare)
  - What if the 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, some will not */
  for(i = 0; i <= count; i++) sum = sum + i;
  return sum;
}
```

  - Big problem for C/C++, less so for Java since code is run on JVM (In above, Java Virtual Machine always initializes all variables to 0)

LOL NOPE

- Compiler issues
- Parallel programming issues
  - If there is a data race, result is undefined
    - Doesn’t matter whether you are using C/C++ or Java
    - Worst part: for many data races, result is correct 99% of time, masking the bug
  - Even without data race, result is often nondeterministic (Depending upon the respective speed of each thread)
    - To thoroughly test, you have to vary the speed of each thread

LOL NOPE

- Compiler issues
- Parallel programming issues
- Non-functional issues (performance, usability, …)
- 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.
- Defining what “sufficient test coverage” means is subjective.
- Must rely on domain knowledge to decide.

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 belong to multiple equivalence classes, you probably need another equivalence class
- Example:
  - Right handed people -> writes with right hand
  - Left handed people -> writes with left hand

  Jane can write with both hands. Which equivalence class does she belong to?
  - Add “Ambidextrous people -> writes with both hands”

Multiple partitionings

- Assume in the previous square root method, if the result contains a decimal point (e.g. *1.3* or *2.23i*), it prints in red, otherwise in black.
- Now we have two partitionings:
  - The positive / 0 / negative partitioning on the previous slide
  - The decimal / non-decimal partitioning on this slide:
    - Number contains decimal -> output printed in red
    - Number does not contain decimal -> output printed in black
- A value can belongs to two equivalence classes, but those equivalence classes must belong to different partitionings
  - Value *1.3* belongs to “positive” and “decimal” equivalence classes
  - But “positive” and “decimal” belong to different partitionings

Values 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

Values do not have to be numeric

- Suppose Twitter only allows alphanumeric [A-Za-z0-9] characters, and tweets must contain at least one character. Tweets that contain any invalid characters are not posted.
- Equivalence classes (NV = number of valid chars, NI = number of invalid chars):
  - (NV >= 1, NI == 0) -> Post the tweet
  - (NV == 0, NI == 0) -> DO NOT post the tweet
  - (NI >= 1) -> DO NOT post the tweet (note NV is irrelevant here)

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 to something manageable!
- How to pick the value? 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?

- Suppose expected behavior is:
  - Method shall take citizenship and age as arguments
  - Method shall determine whether a person can be US president according to a set of rules
  - Rule 1: Person must be a US citizen to be US president
  - Rule 2: Person must be 35 years or older to be US president

- Suppose implementation is:

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

- 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 both boundary and 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…]

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

Try to test both boundary and 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 is also important to see behavior in interior

Try to test both boundary and 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…]

- **Are we done?**

“Hidden” (IMPLICIT) boundary values

- The boundary values we have gone over already are explicit – that is, they are defined by the requirements of the problem itself.
- Some boundaries are implicit – they are generated from the domain, language, hardware, etc.:
    - MAXINT, MININT
    - Maximum precision of a floating point value
    - Allocation limitation (memory, hard drive space, etc.)
    - Physical world boundaries (weight can’t be negative, Y2K won’t happen, etc.)
        - Side note: Y2K did happen and anti-gravity may yet happen

Add implicit boundary values

CANNOT_BE_PRESIDENT =
[MININT,…,-1,0,1,…
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,
…,MAXINT]

- MININT, MAXINT: hardware boundaries
- 0: physical world boundaries (age can’t be negative)

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 interior structure or source code
  - Tests are 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
  - Tests are performed at the code-level (e.g. testing individual methods or classes)
- **Grey-box testing**:
  - Testing with some knowledge of the interior structure and codebase
  - Knowledge may come from partial inspection of code or a design document
  - Tests are performed from the user’s perspective, but informed by tester’s knowledge

Black-box testing examples

- Testing a website using a web browser
- Running a script against an API endpoint
- Checking to see that changing fonts in a word processor works

White-box testing examples

- Testing that a function returns the correct result
- Testing that instantiating a class creates a valid object
- Checking that there are no unused variables
- Checking that exceptions are caught and handled

Grey-box testing examples

- *Reviewing code* and noticing that bubble sort is used. Then writing a *user-facing test* involving a large input size.
- *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 or cross site scripting.
- *Reading a design document* and noticing a critical network connection through which a lot of data passes through. Then writing a *user-facing test* that stresses that network connection.

Static vs dynamic testing

- Dynamic testing = code is executed (at least the part that is exercised in that test run)
- Static testing = code is not executed

Dynamic testing

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

Static testing

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

Now Please Read Textbook Chapters 2-4