BLACKBOX TESTING

CS-HU 274 Lecture 3

TEST ADEQUACY CRITERIA

Test selection criteria are used to guide the selection of a test suite T

A test selection criterion defines "coverage" items that T should include

- If a test case t includes coverage items that are not in T then add it to t.

The same criteria can be used as test adequacy criteria: $\frac{items\ covered\ by\ T}{total\ items}$

A test adequacy criterion describes how complete T is

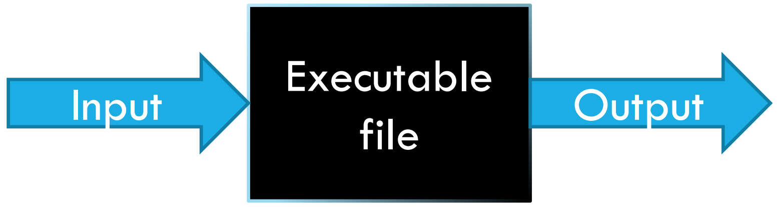
- If T covers 85% of items, then it is better than one with 60% coverage but still could be improved

An adequacy criterion ensures distinct test cases in T with respect to that criterion

We study two families of test adequacy criteria

- Blackbox (this week)
- Whitebox (next week)

BLACKBOX TESTING OVERVIEW



- Program implementation is treated as a “black box”
- No reference to the internal structure of the program
- Can be used for P implemented in different languages
- A test selection criterion derived from a functional/behavioral specification of a program under test
- Applicable to any level of testing: unit, integration and system testing

A BLACKBOX ADEQUACY CRITERION

Deals directly with all possible inputs D of a program P

How can we describe D?

- Depends on its type
- If D is a list containing 1 or 2, then all possible lists D = { (), (1), (2), (1,2), … }
- An infinite numerical domain, e.g., any number D = { … -1,0,1, …}
- A finite domain, e.g., a set of options D = {on, off, skip}

Exhaustive testing

- Execute P on each element of D
- In *general*, this approach is not practical

DEFINING INPUT DOMAIN

1st Step: Define the input domain of a program

m(int x, int y)

- D_x for x is {10,11, …}, i.e., grater than 9 natural numbers
- D_y for y is {…, -4, -2, 0, 2, 4, …}, i.e., even integer numbers
- D_m for m is D_x X D_y = { …, (10,2), (10,0), (10,-2),(11,2), (11,0), (11,-2),…}

n(Option o, boolean y)

- D_o for option is {on, off, skip}
- D_b for boolean is {true, false}
- D_n for n is D_o X D_b = {(on, true), (off, true), (skip, true), (on, false), (off, false), (skip, false)}

Testing all element of an input domain either

- infeasible (cannot be done because D is infinite, e.g., D_m)
- intractable (difficult to do in practice because D takes too many resources to run or too large, e.g., when each input in D_n takes 2 weeks to run)

(GROUP) EXERCISE

Choose 5 inputs that test a program P's behavior the best

- P1: Takes a positive real number and returns its square root.
  - What is the input domain? What are the 5 inputs?
- P2: Takes a line of text and returns the count of the symbol 'k' in the line.
  - What is the input domain? What are the 5 inputs?
- P3: The `pop` method of a stack class.
  - What is the input domain? What are the 5 inputs?

Why did you select those cases?

What was your intuition/intention?

BLACKBOX PRINCIPLE FOR LARGE/INFINITE DOMAINS

Equivalence partitioning of D

- Divide the space of possible inputs D into a finite set of classes.
- The purpose is to select few elements from each equivalence class to test.

Boundary values

- Identify specific values in the equivalent partitions/classes
- They may or more likely to handle incorrectly / cause a failure

Adhere to a systematic approach

- Create a process/algorithm, i.e., identify elementary steps
- Automate the test generation process

EQUIVALENCE PARTITION

2nd Step: using descriptions/specifications of P identify test cases that my reveal classes of errors by executing different parts of programs.

Partition 1

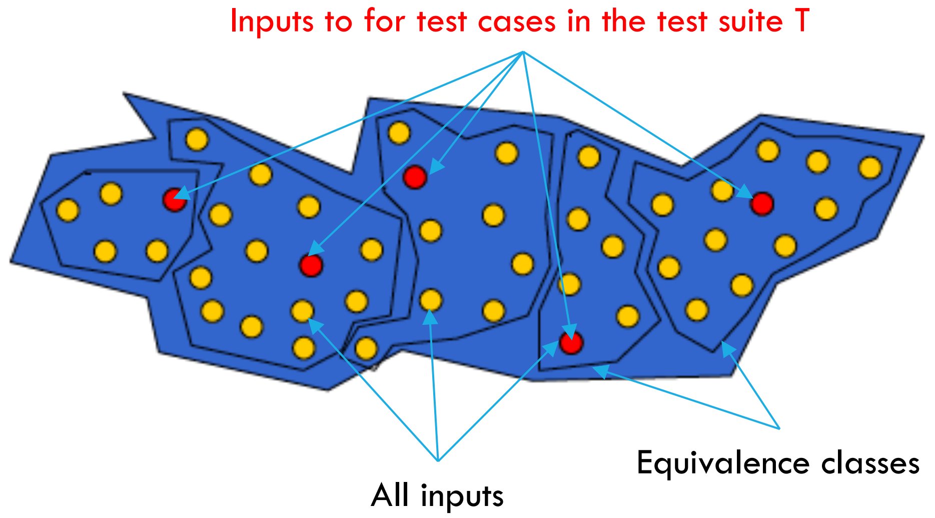
- Class 1: empty list = {()}
- Class 2: non-empty lists = {(1), (2), (11), …}

Partition 2

- Class 1: empty list = {()}
- Class 2: non-empty even length lists = {(1,1), (1,2), …}
- Class 3: non-empty odd length lists = {(1), (1,2,1), …}

3rd Step: Partition the domain into classes from which to select a test input

A class is a set of inputs whose elements are likely to be treated the same by the program



EXAMPLE: EQUIVALENCE PARTITION

Functional description of factorial program

/* -the input cannot be negative, and the program cannot compute factorial of inputs greater than 200.

-for small values up to and not including 20 the program calculates the exact value.

-for larger values it returns an approximate factorial value in a floating-point format. The admissible error is 0.1% of the exact value. */

```
factorial(int n)
```

Possible equivalence classes

- D1: $n < 0$
- D2: $0 \leq n < 20$
- D3: $20 \leq n \leq 200$
- D4: $n > 200$

Chose one test case per equivalence class

- Inputs = { -5, 5, 150, 250 }

EQUIVALENCE PARTITION OBSERVATION

Defining an equivalence partition is an iterative process

- Start with few and if there are still resources available then define more classes (refine partition)
- If there are too many partitions, then merge some of them into one equiv. class (coarsen partition)

D can have more than one equivalence partitions for a given method

- Partition 1: Class 11: non-negative integers, Class 12: negative integers
- Partition 2: Class 21: odd integers, Class 22: even integers
- Two inputs {-2, 3} cover all 4 equivalence classes

An equiv. partition captures the program description, but we want to refine it

A program P with no detailed description of inputs, or just a name: sqrt(int x)

- Think about on what input values P more likely to process differently, e.g., execute different parts of P
- How would you implement such program

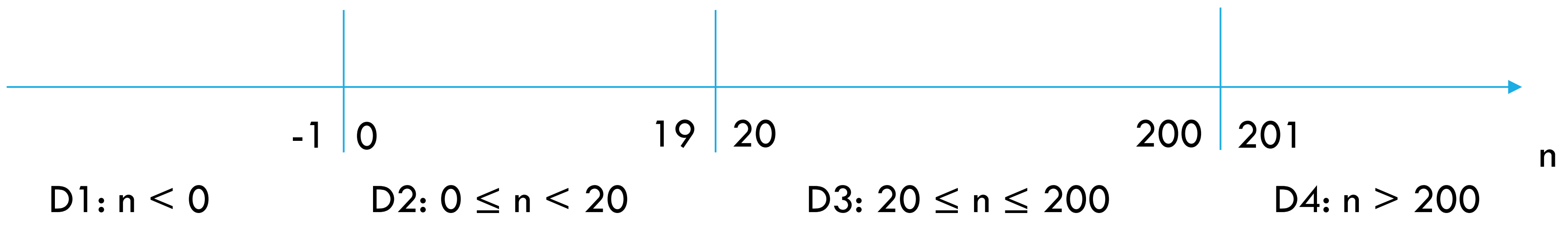
TESTING BOUNDARY CONDITIONS

Selecting inputs from equivalent classes

4th Step: select those values that create boundaries with other equiv. classes and one value that does not (”middle” value).

- More likely those values are used in conditional statements, switch statements
- Exposing bugs in code’s logic – off by one errors

```
factorial(int n)
```



{ -5, 5, 150, 250 } U {-1,0} U {19,20} U {200,201}

Middle values

Boundary values

GENERAL RULES FOR BOUNDARY CONDITIONS

For each range [a,b] listed in either input or output description of P choose five cases

1. Values less than a
2. Value equal to a
3. Value greater than a but less than b
4. Value equal to b
5. Value greater than b

For open intervals n > 200, n < 0

- Maximum value, e.g., Integer.MAX_VALUE
- Minimum value, e.g., Integer.MIN_VALUE
- Can expose overflow bugs

For sets select two values

- In the set
- Not in the set

For equality select two values

- Equal
- Not equal

For sets or lists select two/three cases

- Empty
- Nonempty
- Max elements (if cardinality is fixed)

TEST-DRIVEN DEVELOPMENT

Goal is to rapidly crate code that is testable and correct.

Commonly used in agile development (and should be used in your assignments too)

Test early and test often!

Idea:

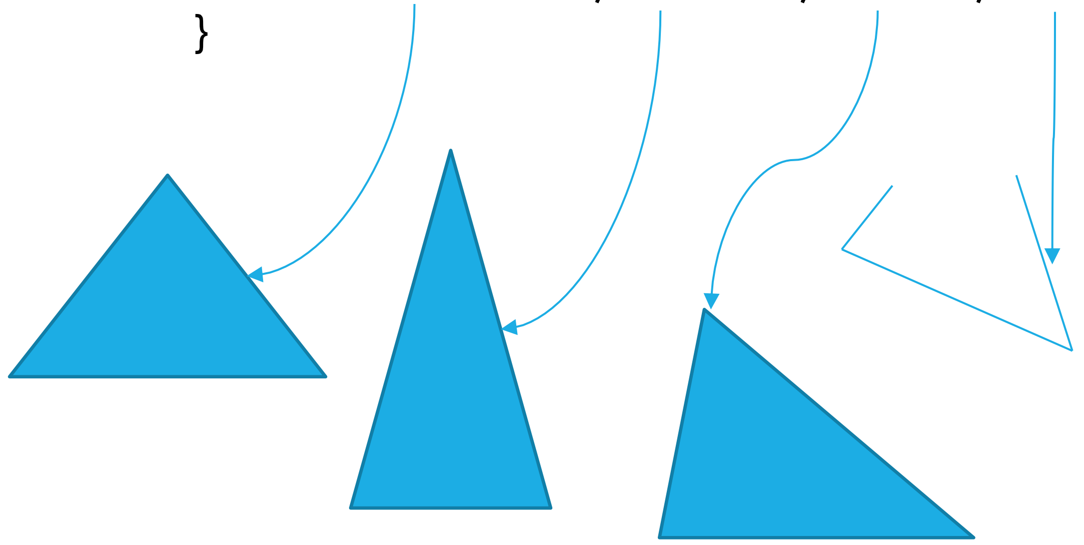
- Developers write a JUnit test suite first (for a given interface)
- Programs until all test cases pass

Cycle: RED, GREEN, REFACTOR

- In the beginning all test cases fail (JUnit is in RED)
- Implements functionality to pass tests (JUnit is getting more GREEN)
- Tests are all pass (JUnit is in GREEN)
- Then the developer can optimize the code (i.e., REFACTOR)

IN-CLASS (GROUP)WORK: TRIANGLE CLASSIFIER

```
enum TriangleType{
            EQUILATERAL, ISOSCELES, SCALENE, INVALID
}
```



```
static TtriangleType classify(int a, int b, int c)
```

What is the input domain for classify?

What is the program description, i.e., relations between a,b, and c?

What are the equivalent classes?

What are the boundaries between those classes?

Write inputs for each equivalent class.

Write inputs for boundary conditions.

Next class: 1)quiz and 2)continue with this exercise