



BASIC TESTING CONCEPTS

CS-HU 274 Lecture 1

GOALS OF TESTING

Improve software quality by finding faults

- “A test is successful if the program fails” (Goodeneogh at al. “Toward Theory of Test Data Selection”)

Premise

- Programs have defects
- Defects can be observed
- Expected observable behaviors of programs are defined
- Few program inputs reveal those defects

Goals

- Find program inputs that cause those defects to surface
- Design clever/sensible heuristic for program input selection

GOALS OF THIS CLASS

Testing is the best effort activity

- None of the input selection strategies guarantee to find a bug
- But for some programs and situations some heuristics work better than others

Input selection heuristics – when to stop testing?

- Blackbox – based on a program's specification/description
- Whitebox – based on a program's code

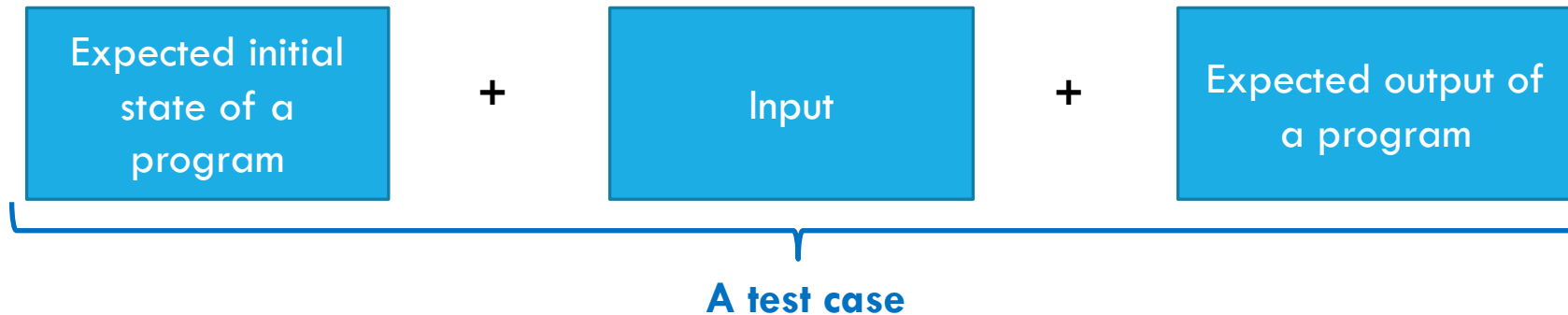
Automate testing

- Testing frameworks
- Automated input generators

Reason about testing strategies

- What is the best input selection heuristic for a given program?
- How can we automate it?

ANATOMY OF A TEST CASE



Initial state

- The start of the program or the beginning of a static method.
- A data-structure in a particular state, e.g., an empty list, a list with one element.

Input

- Concrete values from program's input or method's arguments.
- Sequence of method invocation on an object, e.g., add an element to a list and then remove it.

Expected output (an observable behavior of a program)

- Return value of a method, e.g., the return value is non-negative.
- State of an object, e.g., the list must be empty

FAILURE VS FAULT VS ERROR

Failure


- The observable incorrect behavior of a program (a failed test case)
- Conceptually related to program behavior rather than to code

Fault (bug/defect)

- Related to code
- Necessary condition for the occurrence of a failure

Error

- The cause of a fault
- Usually a human error, e.g., conceptual, typo, copy-paste



For easy debugging, they should be as close as possible in the code

FAILURE VS FAULT VS ERROR

```
1: int times2 (int x){  
2: int y = 0;  
3: y = x*x;  
4: return y;}
```

A Test case:

- Initial: none
- Input: times2(3)
- Expected output: 6

Result 9 represents a **failure**

The failure is due to the **fault** in line 3

The **error** is a typo

FAILURE VS FAULT VS ERROR

```
public class MyList{
    private size = 0;
    private int[] elem;
    public MyList(int size){
        this.size = size;
        elem = new int[size];
    ...
    11: int getLast (){
    12: int indx = size - 1;
    13: return elem[indx];}
    ...
}
```

A Test case:

- Initial:
- Input:
- Expected output:

What is the **failure**?

Where is the **fault**?

What could be the **error**?

COINCIDENTAL CORRECTNESS

```
1: int times2 (int x){  
2: int y = 0;  
3: y = x*x;  
4: return y;}
```

```
11: int getLast (){  
12: int indx = size - 1;  
13: return elem[indx];}
```

Not all test cases result in a failure when executing a fault

- times(2) does not reveal the failure – it is coincidentally correct on input 2
- MyList l1 = new MyList(1);
l1.add(3);
getLast() = 3;
does not reveal the failure – it is coincidentally correct on all but the empty list

ORACLE

A mechanism that determines whether test passed or failed

- Has two parts: **defining** expected output and **checking** expected output.

Used to access whether a test is successful or not

An oracle can be:

Human (tedious, error prone)

- A person examines program outputs to determine whether it behaves correctly

Partially automated (considerable initial investment)

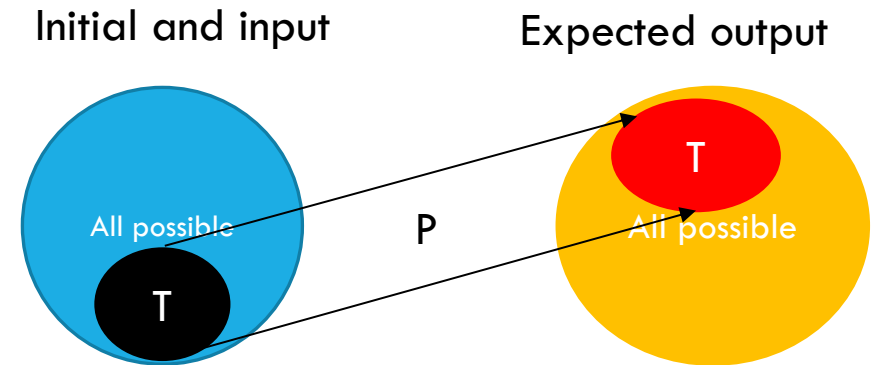
- A run-time environment checks against expected behavior, e.g., Java assertions, assertEquals in Junit.
- Correct behavior: explicit and implicit
- Explicit: a human still needs to define the correct behavior, i.e., what values to put inside assertions. Usually defined for a set of inputs, e.g., if `list.size() ≤ 0` then `list.getLast = -1`
- Implicit: no run-time exceptions such as null-pointer or division by zero.

EXAMPLES OF TEST CASE DEFINITIONS

| Test Case | Initial | Input | Expected |
|-----------|----------------------|-------------------------|------------------------|
| TC 1 | none | times2(2) | returns 4 |
| TC 2 | none | times2(3) | returns 6 |
| TC 3 | none | times2(-2) | returns -4 |
| TC 4 | List is empty | Add element x | getLast() returns x |
| TC 5 | List has one element | Remove element | isEmpty() returns true |
| TC 6 | List has 5 elements | Remove elements 5 times | isEmpty() return true |

TEST SUITE

- A test suite T is a set of test cases
- T is a subset of all possible test cases
- T is an ideal test suite for a program P if it guarantees that P does not fail on all possible test cases
- In general, it is impossible to find an ideal T
- We try our best by defining a suitable test selection criteria
 - Blackbox (week 2)
 - Whitebox (week 3)



CONCEPT CHECK

What is the goal of testing?

What is a test case?

What is a failed test case?

What are differences between failure, fault and error?

What are desired characteristics of a test suite?

AUTOMATED TEST CASE GENERATION

Generating test cases manually is a tedious, but effective approach.

- Few but good quality test cases
- Take long time

Write programs that generate test cases automatically.

- Commonly oracle is a run-time system (e.g., no null pointer exception, no division by zero or other uncaught exceptions)
- Essentially run on its own until test case generation is complete
- Large number of test cases, but 90% are not very useful/equivalent
- Create “clever” techniques to make such generation more effective “smarter”

Need for testing frameworks



Overview of JUnit

Unit testing – tests a specific small part of a code, e.g., a method

Testing framework for Java (TestNG is another popular one)

Tests are placed in a separate “test” top-level directory, which is different from “src” (where the source code goes)

NEXT CLASS

Quiz 1 over the basic testing concepts is due before class.

Working with JUnit framework

- Writing test cases
- Automating JUnit test case generation
- Random testing
- P1 is assigned (should be completed in-class)

Make sure to have Eclipse installed/open and CS-HU374-Public repo imported (see the syllabus link, and I can record a video on how to do it)