EE360T/382V Software Testing

khurshid@ece.utexas.edu

Chapter 1*: Introduction

February 3, 2020

*Introduction to Software Testing by Ammann and Offutt

Overview

Today – Chapter 1: Introduction

Last class – Basic discrete math, Java, JUnit

Next class – Chapter 2 – Graph coverage criteria

- Read: Sections 2.1 and 2.2

Chapter 1: Outline

Activities of a test engineer

- Traditional testing levels based on software activity
- Beizer's testing levels based on test process maturity

Software testing basics

Coverage criteria for testing

Activities of test engineers

Test engineers perform a few key activities:

- Design tests by creating test requirements
- Transform requirements into values and scripts that form executable tests
- Execute tests against the software under test
- Evaluate results of test execution

A test manager is in charge of one or more test engineers

Traditional testing levels based on software activity

Acceptance testing – check if the software is acceptable to the user

System testing – check the system as a whole

Integration testing – check interactions among different modules

Module testing – check how each modules behaves

Unit testing – check how small units of code (e.g., methods) behave individually

Beizer's testing levels based on test process maturity

Level 0 – there is no difference between testing and debugging

Level 1 – the purpose of testing is to show software works

Level 2 – the purpose of testing is to show software does not work

Level 3 – the purpose of testing is not to prove anything specific, but to reduce the risk of using the software

Level 4 – testing is a mental discipline that helps all IT professionals develop higher quality software

Software testing basics

D1.1 Validation – process of evaluating software at the end of software development to ensure compliance with intended usage

- Are we building the *right* system?

D1.2 Verification – process of determining whether the products of a given phase of the software development process fulfil the requirements established during the previous phase

- Are we building the *system* right?

“Bugs” in IEEE 610.12-1990

Fault – incorrect lines of code

Error – faults cause incorrect (unobserved) state

Failure – errors cause incorrect (observed) behavior

Example faulty code

```
public static int numZero(int[] x) {
    // postcondition: if x == null throw
     //   NullPointerException, else return
     //   the number of occurrences of 0 in x
    int count = 0;
    for (int i = 1; i < x.length; i++) {
        if (x[i] == 0) {
            count++;
        }
    }
    return count;
}
```

Fault/failure model

Three conditions must be satisfied for a failure to be observed:

- Reachability – faulty location(s) in code must be reached
- Infection – after execution of faulty code, the program state must be incorrect
- Propagation – the incorrect state must some program output to be incorrect

Two practical issues

D1.11 Observability – how easy is it to observe program behavior in terms of its outputs, effects on the environment, and hardware/software components?

D1.12 Controllability – how easy is to provide the program the needed inputs in terms of values, operations and behaviors

Test case and test suite

D1.9 Test input values – input values necessary to execute program under test

D1.17 Expected results – output that will be produced by execution of a correct program for the given inputs

D1.17 Test case – composed of test input values and expected results (as well as prefix/postfix values) necessary for execution and evaluation of the program

D1.18 Test suite – set of test cases

Coverage criteria for testing

D1.20 Test requirement – specific element of the software artifact that a test case must cover (satisfy)

- Some requirements may be infeasible

D1.21 Coverage criterion – rule or collection of rules that impose test requirements on a test suite

- Criterion defines requirements precisely

D1.22 Coverage – given test requirements *TR* for coverage criterion *C*, test suite *T* satisfies *C* if and only if for each *tr* ϵ *TR*, at least one test in *T* satisfies *tr*

D1.23 Coverage level – given requirements *TR* and suite *T*, coverage level is

Two ways to use coverage criteria

Test generation – create tests to satisfy a desired criterion

- Can form a basis for automated test input generation

Test evaluation – measure the coverage of a given test suite, i.e., use coverage as a *metric*

- Commonly used in practice
- Supported by several tools, e.g., EclEmma
- 100% code coverage does not imply no faults

Subsumption

D1.24 Criteria subsumption – criterion C_1 subsumes criterion C_2 if and only if every test suite that satisfies C_1 also satisfies C_2

Note: must be true for every test suite

Example: if a test suite covers every branch, it covers every statement

Black-box or white-box?

D1.25 Black-box testing – deriving tests from external descriptions

- E.g., specs, designs, and requirements

D1.26 White-box testing – deriving tests from the source-code internals

- E.g., branches, conditions, and statements

Modern testing techniques often use both black-box and white-box approaches

Criteria based on structures

The textbook focuses on four kinds of structures to define criteria:

- Graphs (Chapter 2)
  - E.g., control-flow graphs (CFGs)
- Logical expressions (Chapter 3)
  - E.g., if-conditions
- Input domain characterization (Chapter 4)
  - E.g., sorted array
- Syntactic structures (Chapter 5)
  - E.g., mutation

?/!