

# EE360T/382V Software Testing

khurshid@ece.utexas.edu

March 28, 2017

# Overview

## Today

- Chapter 6: Practical considerations

## Next time

- Exam 2 discussion
- Specification-based testing

# EE360T/382V Software Testing

khurshid@ece.utexas.edu

## Practical considerations (Chapter 6)\*

\*Introduction to Software Testing by Ammann and Offutt

# Overview

Chapters 1—5 provide a practical “toolbox” for testing based on various test criteria

- Studying these criteria lays the foundation for more effective testing

Integrating effective testing strategies into the overall development process helps improve software quality

- A key element of moving to level 3 or 4 of test process maturity (recall from Chapter 1)

This chapter discusses various issues involved with applying test criteria during software development

- The basic philosophy is to approach the issues as technical problems and seek technical solutions

# Outline

Regression testing

Integration and testing

- Stubs and drivers
- Class integration and test order

Test process

Test plans

Identifying correct outputs

- Direct verification
- Redundant computations
- Consistency checks
- Data redundancy

# 6.1 Regression testing

Process of *re-testing* software that has been *modified*, say due to a bug fix or a feature addition

Major part of the testing effort in commercial software

Typically automated but can still take a long time

- Regression suites accumulate over time

Increasing efficiency is hard – a small change can impact many parts of code

Some key questions:

- What to do when regression tests fail?
- What tests to include in a regression suite?
- Which tests in a regression suite to run?

# Regression test failure

A regression test is evaluated by comparing the test pass/fail results on previous and current versions

- If they differ, the regression test fails

There are three possibilities for regression test failure:

- The software is buggy
  - Fix the software
- The test inputs are no longer valid, e.g., a GUI widget no longer exists
  - Modify or remove the test
- The test output is no longer valid
  - Update the test

# Regression suite maintenance

As software ages, regression suites grow larger

- Running them can take hours or days
- Using more machines can reduce the time but are all tests really needed?

Different policies can direct which tests to keep, e.g.,

- For each bug report, have a test that detects the bug
- Keep tests that provide a certain level of coverage

Removing tests is tricky

- Not increasing code coverage?
- Not having found a fault so far?
- Having found the same fault as other tests?



# Regression test selection

What tests to run given a specific change in code?

Various test selection techniques exist

- E.g., *change-impact analysis*
  1. Compute what parts of software are *impacted* by the change
  2. Run tests that exercise those parts

A selection technique is:

- **Inclusive** if it includes *modification-revealing* tests
- **Precise** if it omits *non-modification-revealing* tests
- **Efficient** if using it is faster than running all tests

# Example: minimum value program

## V1

```
/*01*/ static int min(int x, int y, int z) {  
/*02*/     if (x < y) {  
/*03*/         if (x < z) {  
/*04*/             return x;  
/*05*/         } else {  
/*06*/             return y;  
/*07*/         }  
/*08*/     }  
/*09*/     if (y < z) {  
/*10*/         return y;  
/*11*/     }  
/*12*/     return z;  
/*13*/ }
```

## Test suite

```
@Test public void xismin() {  
    assertEquals(1, min(1, 2, 3));  
}  
  
@Test public void yismin() {  
    assertEquals(1, min(2, 1, 3));  
}  
  
@Test public void zismin() {  
    assertEquals(1, min(2, 3, 1));  
}
```

# Example: min – bug fix (V2)

## V2

```
/*01*/ static int min(int x, int y, int z) {  
/*02*/     if (x < y) {  
/*03*/         if (x < z) {  
/*04*/             return x;  
/*05*/         } else {  
/*06*/             return z;  
/*07*/         }  
/*08*/     }  
/*09*/     if (y < z) {  
/*10*/         return y;  
/*11*/     }  
/*12*/     return z;  
/*13*/ }
```

## Test suite

```
@Test public void xismin() {  
    assertEquals(1, min(1, 2, 3));  
}  
  
@Test public void yismin() {  
    assertEquals(1, min(2, 1, 3));  
}  
  
@Test public void zismin() {  
    assertEquals(1, min(2, 3, 1));  
}
```

# Example: coverage

## V1

```
/*01*/ static int min(int x, int y, int z) {  
/*02*/     if (x < y) {  
/*03*/         if (x < z) {  
/*04*/             return x;  
/*05*/         } else {  
/*06*/             return y;  
/*07*/         }  
/*08*/     }  
/*09*/     if (y < z) {  
/*10*/         return y;  
/*11*/     }  
/*12*/     return z;  
/*13*/ }
```

## Test suite

```
@Test public void xismin() {  
    assertEquals(1, min(1, 2, 3));  
}  
  
@Test public void yismin() {  
    assertEquals(1, min(2, 1, 3));  
}  
  
@Test public void zismin() {  
    assertEquals(1, min(2, 3, 1));  
}
```

# Example: test selection for V2

V2 was created by editing only one statement on Line 6

Test execution on V1 has the following coverage matrix:

Test/Statement	2	3	4	6	9	10	12
xismin	X	X	X				
yismin	X				X	X	
zismin	X	X		X			

Execution of the first two tests on V2 must have the same result as before

- No need to re-run them
- Only 1 out of 3 tests needs to be re-run

# Example: min – refactoring (V3)

V2

```
/*01*/ static int min(int x, int y, int z) {  
/*02*/     if (x < y) {  
/*03*/         if (x < z) {  
/*04*/             return x;  
/*05*/         } else {  
/*06*/             return z;  
/*07*/         }  
/*08*/     }  
/*09*/     if (y < z) {  
/*10*/         return y;  
/*11*/     }  
/*12*/     return z;  
/*13*/ }
```

V3

```
static int min(int x, int y, int z) {  
    if (x < y) {  
        if (x < z) {  
            return x;  
        } else {  
            return z;  
        }  
    }  
    return (y < z) ? y : z;  
}
```

## 6.2 Integration and testing

Software systems typically consist of many components that may have different developers

**Unit testing** tests individual components

**Integration testing** tests interfaces among components (that are already unit tested)

- Typically performed on an *incomplete* system, e.g., a tester may check interaction among 2 components
  - Requires **scaffolding**, i.e., extra code to support integration and testing
    - **Stub** – emulates unimplemented functionality
    - **Driver** – emulates calls to code under test


# Stubs

A stub must enable compilation of components under test, e.g.,

- Stub of a method must have compatible signature and return type

A stub must emulate functionality that allows testing

A stub can be implemented in several ways:

- 
- Return a constant value
  - Return a random value
  - Return value from table lookup
  - Return value entered by user during test execution
  - Return value based on a formal specification



# Class-integration and test order

**CITO problem** -- what order to integrate and test?

- A key decision when integrating multiple components

In OO programs, classes can have various *dependencies*

- A class may use methods in another class
- A class may inherit from another class
- A class may aggregate (in its fields) objects of another class

The *dependency graph* may have cycles, e.g., C calls a method in D and D calls a method in C

An approach: break cycles; use minimal stubbing overall

## 6.4 Test plan

Its key contents are:

- How the tests were created
- Why the tests were created
- How the tests will be run

ANSI/IEEE standard 829-1983 defines it as:

- *A document describing the scope, approach, resources, and schedule of intended test activities. It identifies test items, the features to be tested, the testing tasks, who will do each task, and any risks requiring contingency planning.*

## 6.5 Identifying correct outputs

**Oracle problem** – “Is the output correct?”

- Often a complex problem to solve

This section discusses four techniques to check outputs:

- Direct verification of outputs
  - Behavioral specifications
- Redundant computations
  - Alternative implementations
- Consistency checks
  - Class invariants or other partial properties, e.g., tree has no cycle, output is one of the inputs, ...
- Data redundancy -- compare outputs for different inputs

# Direct verification of outputs

Automated checking using an executable specification

- Implement code that checks expected properties
  - `boolean oracle(Object input, Object output);`

Same check can run against many inputs

Cost is often high

- Need to implement complex checking logic
  - E.g., relation between pre-state and post-state
    - `add` correctly adds the given element to a set

# Example: specifying sort

Consider a method to sort an integer array

- `void sort(int[] arr);`

Post-condition: arr is in sorted order

- Pre-state: [8, 92, 8, 14]
- Post-state: [1, 2, 3, 4] – allowed by (weak) spec

△Post-condition: no new elements introduced

- Post-state: [92, 14, 8, 8] – still allowed by spec

△Post-condition: ascending order

- Post-state: [8, 14, 14, 92] – still allowed by spec

△Post-condition: pre-state is a permutation of post-state

# Example: executable check

```
static boolean checkSort(int[] arr) {  
    int[] copy = Arrays.copyOf(arr, arr.length);  
    sort(arr);  
    for(int x: arr) {  
        if (count(x, arr) != count(x, copy)) return false;  
    }  
    for (int i = 0; i < arr.length - 1; i++) {  
        if (arr[i] > arr[i + 1]) return false;  
    }  
    return true;  
}
```

Similar in complexity to the method under test!

# Redundant computation

Aka differential testing

- Use another implementation, e.g., one that has slower performance
  - Check if the outputs match

Can check many test executions

Can be expensive

May have coincidental failures

- Both programs may fail on the same input

Applies naturally in regression testing

- Check outputs of current version against outputs of previous version

# Data redundancy

Compare outputs for different inputs

- Partial solution
  - Checks for specific faults

E.g., use knowledge about *identities*

- Pushing an element on a stack followed by popping it leaves the stack in its original state
- $\min(1, 2, 3) == \min(2, 1, 3)$
- $\sin(a + b) == \sin(a).\cos(b) + \cos(a).\sin(b)$



# Outline

Regression testing

Integration and testing

- Stubs and drivers
- Class integration and test order

Test process

Test plans

Identifying correct outputs

- Direct verification
- Redundant computations
- Consistency checks
- Data redundancy

?/!