Object-Oriented Programming
Software testing

CSIE Department, NTUT
Woei-Kae Chen
Software Testing (1)

- Software testing should use information from both specification and program.
  - Specification-based testing and interface-based testing belong to black-box testing.
  - Program-based testing and combined specification and program-based criteria belong to white-box testing.

- Black-box testing
  - Black-box testing treats the program under test as a “black box.” No knowledge about the implementation is assumed.

- White-box testing
  - The tester has access to the details of the program under test and performs the testing according to such details.
Three basic approaches to software testing

- **Structural testing**
  - Specifies testing requirements in terms of the coverage of a particular set of elements in the structure of the program or the specification.

- **Fault-based testing**
  - Focuses on detecting faults (i.e., defects) in the software.

- **Error-based testing**
  - Requires test cases to check the program on certain error-prone points according to our knowledge about how programs typically depart from their specifications.
Software Testing: Coverage criteria

- **Statement coverage**
  - testers are required to generate test cases to execute every statement in the program at least once.

- **Branch coverage**
  - requires that all control transfers in the program under test are exercised during testing.

- **Path coverage**
  - requires that all the execution paths from the program’s entry to its exit are executed during testing.

- **Mutation**
  - A program with a planted fault is called a mutant of the original program.
  - A way to measure how well test cases are designed is to plant some artificial faults into the program and check if they are detected by the test.
Software Testing: Statement coverage (1)

```cpp
class X {
    . . .
    void f() {
        . . .
    }
};

class XTest {
    . . .
    void testf() {
        X x;
        x.f();
        CPPUNIT_ASSERT(...);
    }
};
```

Function under test

Test case(s)

Statement coverage: are all statements executed?
class X {
    . . .
    void f(bool x) {
        . . .
        if (x) {
            . . .
        }  
    }
};

class XTest {
    . . .
    void testf() {
        X x;
        x.f(false);
        CPPUNIT_ASSERT(...);
    }
    . . .
};

There are unexecuted statements
class X {
    . . .
    void f(bool x) {
        . . .
        if (x) {
            . . .
            class XTest {
            }
            . . .
            void testf() {
                X x;
                x.f(true);
                CPPUNIT_ASSERT(…);
            }
        }
    }
};

All statements are executed
Software Testing: Branch coverage (1)

```cpp
class X {
    ...  
    void f(bool x) {
        ...  
        if (x) {
            ...  
            class XTest {
                ...  
                void testf() {
                    X x;
                    x.f(true);
                    CPPUNIT_ASSERT(...);
                }
            }
        }
    }
};
```

Branch coverage: are all control transfer exercised?

All statements are executed

Unexercised
class X {
    . . .
    void f(bool x) {
        . . .
        if (x) {
            . . .
        }
        . . .
    }
    . . .
};

class XTest {
    . . .
    void testf() {
        X x;
        x.f(true);
        CPPUNIT_ASSERT(…);
        . . .
        x.f(false);
        CPPUNIT_ASSERT(…);
    }
    . . .
};

Is reusing x a good idea?

Two test cases

All control transfers are exercised
class X {
    . . .
    void f(int x) {
        . . .
        do {
            x--;
            . . .
        } while (x>0);
        . . .
    }
};

class XTest {
    . . .
    void testf() {
        X x;
        x.f(0);
        CPPUNIT_ASSERT(...);
    }
};

Statement coverage? Yes
Branch coverage? No
JUnit FAQ: Best Practices

- **Test-first programming**
  - Tests should be written before the code.
  - Test-first programming is practiced by only writing new code when an automated test is failing.
  - When all the tests pass, you know you're done!
  - When a bug is reported, first write unit test(s) to expose the bug(s), then fix them.
    - This makes it almost impossible for that particular bug to resurface later.
  - Good tests tell you how to best design the system for its intended use.
  - Test-driven development is a lot more fun than writing tests after the code seems to be working.
Do I have to write a test for everything?

- No, just test everything that could reasonably break.
  - Investments in testing are equal investments in design.
- If defects aren't being reported, and your design responds well to change, then you're probably testing enough.
- If you're spending a lot of time fixing defects and your design is difficult to grow, you should write more tests.
- If something is difficult to test, it's usually an opportunity for a design improvement.
JUnit FAQ: Best Practices

How simple is “too simple to break”? 
  – If it can't break on its own, it's too simple to break.
  – Example
    • `getX()` method cannot break unless the compiler is also broken. Therefore, don't test `getX()`.
    • `setX()` method is also too simple to break. However, if it does any parameter validation, you likely need to test it.
JUnit FAQ: Best Practices

**How often should I run my tests?**
- Run all your unit tests as often as possible
  - Ideally every time the code is changed.
- Make sure all your unit tests always run at 100%.
- Frequent testing gives you confidence that your changes didn't break anything.
- For larger systems, you may just run specific test suites that are relevant to the code you're working on.
- Run all the tests of the system at least once per day (or night).
JUnit FAQ: Best Practices

- **What do I do when a defect is reported?**
  - Write a failing test that exposes the defect
    - When the test passes, you know the defect is fixed!
  - This is a learning opportunity
    - Perhaps the defect could have been prevented by being more aggressive about testing everything that could reasonably break.

- **Why not just use print?**
  - It requires that output be scanned manually every time the program is run to ensure that the code is doing what's expected.
  - Tests should retain its value over time.

- **Why not just use a debugger?**
  - The same as that of using print.
JUnit FAQ: Best Practices

- **Testing Idioms**
  - Code a little, test a little, code a little, test a little...
  - Begin by writing tests for the areas of code that you're most worried about breaking.
  - Write tests that have the highest possible return on your testing investment.
  - When you need to add new functionality to the system, write the tests first.
  - If you find yourself debugging using System.out.println(), write a test case instead.
  - The next time someone asks you for help debugging, help them write a test.
  - Don't deliver software that doesn't pass all of its tests.
Homework #4: Example

- What if output format changes
- What if input format changes