Software Testing II

Prof. Dr. Holger Schlingloff
Humboldt-Universität zu Berlin
and
Fraunhofer Institute of Computer Architecture
and Software Technology FIRST
Outline of this Lecture Series

• 2006/11/24: Introduction, Definitions, Examples
• 2006/11/25-1: Functional testing
• 2006/11/25-2: Structural testing
• 2006/11/26-1: Model-based test generation
• 2006/11/26-2: Specification-based test generation

• Next week: Your turn!
Outline of This Lecture

- Functional versus structural testing
- Functional testing
  - Unit testing
  - (Integration and systems testing) ➔ afternoon?
Software Engineering Facts

• Typical workload distribution: 10% requirements, 10% architecture, 20% design, 15% implementation, 45% integration and testing

• Siemens: 7% of development is coding

• Quality assurance takes 30-80% of the overall development costs

• Unit testing takes 40-70% of implementation work

• In safety critical systems it may go up to 90%
Specification-based versus Code-based

- Test case derivation source: specification or code?
- Specification-based testing checks whether the specified behaviour is implemented; it cannot find unspecified program behaviours (e.g. viruses)
- Program-code-based testing checks whether the implemented behaviour is correct (with respect to the specification); it cannot find unimplemented requirements (e.g. missing features)
Black-box versus White-Box

- **White-Box**: Structure is openly accessible and visible to the tester; e.g. reading and writing of program variables during test execution
- **Black-Box**: Internals are hidden (e.g. for copyright reasons); access only through documented external interfaces
- **Grey-Box**: Some internal details are revealed for testing purposes (e.g. special testing monitors)
Functional versus Structural Testing

• Focus on
  ▪ “functional”: performed *function*, i.e. actions or activities of the SUT, “what” (external view)
    - e.g. relation between input and output values
  ▪ “structural”: designed *structure*, i.e. components or implementation, “how” (internal view)
    - e.g. data and data types or algorithmic idea

• Often used synonymously:
  ▪ functional test – black-box-test – specification-based test
  ▪ structural test – white-box-test – code-based test
Code-based quality assessment

- Review, audit, walk through
  - four-eyes principle
  - audit rules, Fagan inspection
  - software metrics, coding rules, quality manuals

- Verification
  - model checking, automated theorem proving
  - static analysis

- Unit testing
Outline of This Lecture

- Functional versus structural testing
- Functional testing
  - Unit testing
  - Integration and systems testing
Unit Testing

• Often considered “the” testing
  ▪ first analytical step after coding
  ▪ first execution of actual system parts
  ▪ often done by programmer herself

• The SUT consists of:
  ▪ procedures, functions (in imperative languages)
  ▪ modules, units, classes, interfaces (in oo programs)
  ▪ blocks (in model-based development)

• Same or similar development environment as for SUT itself (compiler, linker, platform, …)
Unit Testing Methodology

• Each unit is tested independently of all the others
  ▪ no external influences or disturbances
  ▪ fault localisation is usually no problem
  ▪ nesting is allowed, poses additional difficulties

• Unit is linked with testing program
  ▪ setting of SUT environment (variables) by testing program
  ▪ invocation of SUT functions with appropriate parameters
  ▪ evaluation of result by comparison of variables
Unit Testing Goals

• Exhibition of faults in the program text
  ▪ wrong calculations, wrong operations
  ▪ incorrect output values, incorrect parameter treatment
  ▪ missing paths, missing cases, missing exception handling
  ▪ duplicate cases, spurious outputs, redundant code
  ▪ wrong loops, wrong boundary values, wrong pointer arithmetic
  ▪ timing and synchronisation problems (difficult!)
  ▪ incorrect (error) messages

• Non-functional properties (efficiency, usability) are of secondary importance
Unit Testing Procedure

• Bottom up:
  ▪ Start with classes which do not depend on others
  ▪ Test all functions in this class
  ▪ Make sure all data fields are accessed and all statements executed (at least)
  ▪ Then test those classes which build solely onto the already tested ones

• Layered view onto the architecture
  ▪ Test cases are grouped into test suites according to this view
Unit Testing Process

• Who?
  - Ideally, programmer and tester are different persons
  - In practice, unit testing is performed by programmers

• When?
  - Ideally, in parallel to the implementation
  - In practice, often after implementation is finished
    - “test it before anybody else can see it”
  - eXtreme programming: before implementation begins
    - tests available during implementation
    - find design errors before even starting to implement
An Example

```java
public final class IMath {
    /*
     * Returns an integer approximation
     * to the square root of x.
     */
    public static int isqrt(int x) {
        int guess = 1;
        while (guess * guess < x) {
            guess++;
        }
        return guess;
    }
}

/** A class to test the class IMath. */
public class IMathTestNoJUnit {
    /** Runs the tests. */
    public static void main(String[] args) {
        printTestResult(0);
        printTestResult(1);
        printTestResult(2);
        printTestResult(3);
        printTestResult(4);
        printTestResult(7);
        printTestResult(9);
        printTestResult(100);
    }
    private static void printTestResult(int arg) {
        System.out.print("isqrt(" + arg + ") ==> ");
        System.out.println(IMath.isqrt(arg));
    }
}
```

Beispiel: Yoonsik Cheon, University of Texas at El Paso, www.cs.utep.edu/~cheon/cs3331/notes/unit-testing.ppt
Discussion

• What is the output of the test?
• What advantage (if any) does this method have compared to manual testing?
• Which types of errors can be found, which can’t be found this way?
• What problems do you see in this procedure?
• What could be improved?
JUnit

- Controlled test execution and evaluation
- Public domain
- Integrated into IDEs (e.g. Eclipse)
- Supports testing by developers
- Eliminates tedious work in writing tests

```java
import junit.framework.*;
public class IMathTest extends TestCase {
    public void testIsqrt() {
        assertEquals(0, IMath.isqrt(0));
        assertEquals(1, IMath.isqrt(1));
        ...
        assertEquals(10, IMath.isqrt(100));
    }
    public static Test suite() {
        return new TestSuite(IMathTest.class);
    }
    public static void main (String[] args) {
        junit.textui.TestRunner.run(suite());
    }
}
```
JUnit Criticism

• Advantages
  ▪ automated, repeatable test execution
  ▪ test suites linked to program code, for each class a test class
  ▪ full flexibility e.g. for test data import, access to internal (public)
  ▪ automated evaluation of assertions
  ▪ possibility to test exception handling
  ▪ integration with IDE

• Disadvantages
  ▪ No test case selection
  ▪ No test oracle
Unit Test Criticism

- Writing Unit Test Cases is a work which is closely related to the implementation and to the code, often done by implementers
  - "programmers know their code best", but also tend to overlook their own errors (no redundancy)
  - "why bother with a specification if the code is available", but also the user perspective may be lost
  - "unit tests can help in debugging", but they may not demonstrate the correctness of the SUT
  - "background information available", but undocumented assumptions may also have other impacts as well
How to Select Test Cases?

• For complexity reasons it is not possible to test all possible inputs to a program
  - 32-bit integer $\rightarrow 10^{10}$ values
  - (month, day, year) $\rightarrow 12 \times 31 \times 700 = 260,000$ combinations

• **Test selection problem**: given an upper size to the test suite, which subset of all possible test cases has the highest error-uncovering probability?
  - equivalence classes
  - boundary values
  - decision tables
Equivalence Class Method

• 1st step: partition the input domain into a finite number of equivalence classes (w.r.t. potential errors)
  ▪ e.g. equilateral triangle, i.e. three equal positive integers
  ▪ e.g. [-maxint,-1] [0] [1,3] [4, maxint]

• 2nd step: choose one representative from each class
  ▪ e.g. (2,2,2)
  ▪ e.g. -3, 0, 2, 7

• 3rd step: combine representatives into test cases
  ▪ only feasible combinations
How to build equivalence classes

• Look at the domains for input and output
• For each parameter there are valid and invalid classes
  ▪ enumerations: contained / not contained
  ▪ different computation paths: for each path a valid, plus one invalid class
  ▪ outputs which are calculated differently: one class for each case
  ▪ input preconditions: one valid and one invalid class for each input
• Split a class if you have reasons to assume that the elements are treated differently
How to combine representative values

- Complete: cartesian product of representative classes
  - Usually too many test cases, combinatorial explosion

- Heuristic: Choose according to the following strategy
  - Test cases which cover many previously uncovered cases
  - Test cases covering exactly one invalid class
  - Pair-wise combination of values

ex: (2,2,2) and (2,2,3), (-7,1,2), (5,“a”,2)
Example

public final class IMath {

    public static int idiv (int x, y) {
        /* Returns the integer quotient
           of the two input values */
        ...
    }
}

• Which are the test cases arising by the equivalence class method?
• Which important cases are not covered?
Discussion

• Pro
  ▪ systematic procedure
  ▪ reasonable number of test cases
  ▪ well-suited for “small” functions with pre- and post-conditions

• Contra
  ▪ selection of test cases by heuristics
  ▪ interaction between parameter values neglected
  ▪ with complex parameter sets many equivalence classes
Improvements

• Boundary value analysis
• Decision table methods
  ▪ Classification tree method