FUNCTIONAL TESTING

F. Tip and M. Weintraub
ACKNOWLEDGEMENTS

Thanks go to Andreas Zeller for allowing incorporation of his materials
HOW TO TELL IF A SYSTEM MEETS EXPECTATIONS?

Two options:

1. **testing**: execute parts of the program and observe if unexpected behaviors occur

2. **formal verification**: exhaustively enumerate all states of the system, and try to prove that properties to be verified hold in each state.
   - Various techniques, e.g. model checking
THE FIRST COMPUTER BUG (1947)

The First "Computer Bug". Moth found trapped between points at Relay # 70, Panel F, of the Mark II Aiken Relay Calculator while it was being tested at Harvard University, 9 September 1947.

The operators affixed the moth to the computer log, with the entry: "First actual case of bug being found". They put out the word that they had "debugged" the machine, thus introducing the term "debugging a computer program".

In 1988, the log, with the moth still taped by the entry, was in the Naval Surface Warfare Center Computer Museum at Dahlgren, Virginia. The log is now housed at the Smithsonian Institution’s National Museum of American History, who have corrected the date from 1945 to 1947. Courtesy of the Naval Surface Warfare Center, Dahlgren, VA., 1988. NHHC Photograph Collection, NH 96566-KN (Color).
WHAT TO TEST?

Configurations
DIJKSTRA’S CURSE

Testing can only find the presence of errors, but not their absence
FORMAL VERIFICATION
FORMAL VERIFICATION

Design or Specification Level

Level of Abstraction

Configurations
FORMAL VERIFICATION

Level of Abstraction

- Design or Specification Level
- High Level Framework

Configurations
FORMAL VERIFICATION

Level of Abstraction

Design or Specification Level

High Level Framework

Code Level

Configurations
FORMAL VERIFICATION

Level of Abstraction

- Design or Specification Level
- High Level Framework
- Code Level
- Assembly

Configurations
FORMAL VERIFICATION

- Design or Specification Level
- High Level Framework
- Code Level
- Assembly
- OS
- Hardware
ZELLER’S COROLLARY

Verification can only find the *absence* of errors, but never their *presence*
BACK TO TESTING: HOW TO COVER AS MUCH OF THE SPACE AS POSSIBLE?
FUNCTIONAL TESTING – AKA BLACK BOX TESTING
WHITE BOX TESTING IS WHERE YOU TEST BASED ON KNOWING WHAT’S INSIDE THE MODULE
IF WE CANNOT KNOW THE CODE INSIDE, AGAINST WHAT DO WE WRITE TESTS?
IF WE CANNOT KNOW THE CODE INSIDE, AGAINST WHAT DO WE WRITE TESTS?

Specifications
TESTING TACTICS

- Tests based on *spec*
- Test covers as much *specified* behavior as possible

- Tests based on *code*
- Test covers as much *implemented* behavior as possible
WHY DO FUNCTIONAL TESTING?

1. Program code not necessary

2. Early functional test design has benefits
   1. Reveals spec problems
   2. Assesses testability
   3. Gives additional explanation of spec
   4. May even serve as spec, as in XP

Functional/Black Box

Structural/White Box
WHY DO FUNCTIONAL TESTING?

- Best for *missing logic* defects
- Common problem:
  Some program logic was simply forgotten
  Structural testing would not focus on code that is not there
- Applies at all granularity levels
  - unit tests
  - integration tests
  - system tests
  - regression tests
RANDOM TESTING

- Pick possible inputs uniformly
- Avoids designer bias

A real problem: The test designer can make the same logical mistakes and bad assumptions as the program designer (especially if they are the same person)

- But treats all inputs as equally valuable
Angle

Force
INFINITE MONKEY THEOREM
INFINITE MONKEY THEOREM

If you put enough monkeys in front of typewriters and give them enough time, you eventually will get Shakespeare.
$2^{32} = 4,294,967,296$

different values
18,446,744,073,709,551,616 COMBINATIONS

\[
\text{total number of trials} = 2^{32} \times 2^{32} = 2^{64} = 18,446,744,073,709,551,616
\]
Computer scientists are smart, and they can systematically test and analyze programs.
SYSTEMATIC FUNCTIONAL TESTING

- **Functional specification**
- **Independently testable feature**
- **Representative values**
- **Model**
- **Test case specifications**
- **Test case**

**Arrows**
- Identify
- Derive
- Generate
TESTABLE FEATURES

- Decompose system into *independently testable features* (ITF)
- An ITF need not correspond to units or subsystems of the software
- For system testing, ITFs are exposed through user interfaces or APIs
WHAT ARE THE INDEPENDENTLY TESTABLE FEATURES?

class Roots {
    // Solve ax^2 + bx + c = 0
    public roots(double a, double b, double c) {
        ... }

    // Result: values for x
    double root_one, root_two;
}
EVERY FUNCTION IS AN INDEPENDENTLY TESTABLE FEATURE

- Consider a multi-function calculator
- What are the independently testable features?
Try to select inputs that are especially valuable.

Usually by choosing representatives of equivalence classes that are apt to fail often or not at all.

REPRESENTATIVE VALUES

 Independently testable feature

Representative values

Model

Test case

Test case specifications
LIKE FINDING NEEDLES IN A HAYSTACK

To find bugs systematically, we need to find out what makes certain inputs or behaviors special
SYSTEMATIC PARTITION TESTING

Failure (valuable test case)

No failure

Failures are sparse in some regions of possible inputs ...

... but dense in other

The space of possible input values
(The haystack)

If we systematically test some cases from each part, we will include the dense parts

Functional testing is one way of drawing lines to isolate regions with likely failures
EQUIVALENCE PARTITIONING

<table>
<thead>
<tr>
<th>Input condition</th>
<th>Equivalence classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>range</td>
<td>one valid, two invalid (larger and smaller)</td>
</tr>
<tr>
<td>specific value</td>
<td>one valid, two invalid (larger and smaller)</td>
</tr>
<tr>
<td>member of a set</td>
<td>one valid, one invalid</td>
</tr>
<tr>
<td>boolean</td>
<td>one valid, one invalid</td>
</tr>
</tbody>
</table>

Defining equivalence classes comes from input conditions in the spec. Each input condition induces an equivalence class – valid and invalid inputs.
BOUNDARY ANALYSIS – FINDING ERROR AT THE EDGES

- Possible test case

- Test at lower range (valid and invalid)
- at center
- at higher range (valid and invalid)
EXAMPLE: ZIP CODE

- Input: 5-digit ZIP code
- Output: list of cities

What are representative values to test?
VALID ZIP CODES

1. With 0 cities as output (0 is boundary value)
2. With 1 city as output
3. With many cities as output
INVALID ZIP CODES

4. Empty input

5. 1–4 characters
   (4 is boundary value)

6. 6 characters
   (6 is boundary value)

7. Very long input

8. No digits

9. Non-character data
“SPECIAL” ZIP CODES

1. How about a ZIP code that reads

```
12345'; DROP TABLE orders; SELECT * FROM zipcodes WHERE 'zip' = '
```

2. A ZIP code with 65536 characters...

This is security testing
OR, YOU CAN USE MODELS TO DEFINE TESTS

- Use a formal model that specifies software behavior
- Models typically come as
  - finite state machines and
  - decision structures

- Identify independently testable feature
- Derive representative values
- Identify model
- Derive test case specifications
Finite State Machine for Product Maintenance

Requirements

Maintenance: The Maintenance function records the history of items undergoing maintenance.

If the product is covered by warranty or maintenance contract, maintenance can be requested either by calling the maintenance toll free number, or through the Web site, or by bringing the item to a designated maintenance station.

If the maintenance is requested by phone or Web site and the customer is a US or EU resident, the item is picked up at the customer site, otherwise, the customer shall ship the item with an express courier.

If the maintenance contract number provided by the customer is not valid, the item follows the procedure for items not covered by warranty.

If the product is not covered by warranty or maintenance contract, maintenance can be requested only by bringing the item to a maintenance station. The maintenance station informs the customer of the estimated costs for repair. Maintenance starts only when the customer accepts the estimate. If the customer does not accept the estimate, the product is returned to the customer.

Small problems can be repaired directly at the maintenance station. If the maintenance station cannot solve the problem, the product is sent to the maintenance regional headquarters (if in US or EU) or to the maintenance main headquarters (otherwise).

If the maintenance regional headquarters cannot solve the problem, the product is sent to the maintenance main headquarters.

Maintenance is suspended if some components are not available.

Once repaired, the product is returned to the customer.
COVERAGE CRITERIA

1. **Path coverage**: Tests cover every path
   ✤ *Not feasible in practice*
   
   *Cycles create infinite paths*
   
   *Acyclic graphs can still have an exponential number of paths*

2. **State coverage**: Every node is executed
   ✤ *A minimum testing criterion*

3. **Transition coverage**: Every edge is executed
   ✤ *Typically, a good coverage criterion to aim for*
TRANSITION COVERAGE

Each test case covers a set of transitions

Here, there are five needed to cover each transition once

_one color = one test case_
STATE-BASED TESTING

- Protocols (e.g., network communication)
- GUIs (sequences of interactions)
- Objects (methods and states)

Figure 14.3 State diagram for Account class (adapted from [KIR94])
Some specifications define decision tables, decision trees, or flow charts. We can define tests from these structures.

<table>
<thead>
<tr>
<th>Type of Purchaser</th>
<th>Educational Purchaser</th>
<th>Individual Purchaser</th>
</tr>
</thead>
<tbody>
<tr>
<td>Education account</td>
<td>T T</td>
<td>F F F F F F F F T T T</td>
</tr>
<tr>
<td>Current purchase &gt; Threshold 1</td>
<td>– – F F T T – –</td>
<td></td>
</tr>
<tr>
<td>Current purchase &gt; Threshold 2</td>
<td>– – – – F F T T T T</td>
<td></td>
</tr>
<tr>
<td>Special price &lt; scheduled price</td>
<td>F T F T – – – –</td>
<td></td>
</tr>
<tr>
<td>Special price &lt; Tier 1</td>
<td>– – – – F T – –</td>
<td></td>
</tr>
<tr>
<td>Special price &lt; Tier 2</td>
<td>– – – – – – F T T</td>
<td></td>
</tr>
<tr>
<td>Outcome</td>
<td>Edu discount Special price</td>
<td>No discount Special price Tier 1 discount Special price Tier 2 discount Special Price</td>
</tr>
</tbody>
</table>
CONDITION COVERAGE

- **Basic Criterion**: each condition should be evaluated once using each possible setting
  - “Don’t care” entries (–) can take arbitrary values

- **Compound Criterion**: Evaluate every possible combination of values for the conditions

- **Decision Coverage**: the expression should be evaluated once so it results in each possible outcome

- **Modified Condition/Decision Coverage (MC/DC)**
  - Each decision takes every possible outcome
  - Each condition in a decision takes every possible outcome
  - Each condition in a decision is shown to independently affect the outcome of the decision.
  - used in safety-critical avionics software
  - details in Pezze + Young, “Software Testing and Analysis”, Chapter 14
LEARNING FROM THE PAST
PARETO’S LAW

Approximately 80% of defects come from 20% of modules
DERIVING TEST SPEC’S

Functional specification → Independently testable feature

identify

Identify

Representative values → model

derive

generate

Test case specifications → Test case

derive

generate
COMBINATORIAL TESTING

Windows
Oracle
MySQL
Apache
IIS
Linux
OS
Server
Database
COMBINATORIAL TESTING

1. Eliminate invalid combinations
   ✦ IIS only runs on Windows, for example

2. Cover all pairs of combinations
   such as MySQL on Windows and Linux

3. Combinations typically generated automatically and – hopefully – tested automatically, too
PAIRWISE TESTING MEANS TO COVER EVERY SINGLE PAIR OF CONFIGURATIONS
RUNNING A TEST

A test case...

1. *sets up an environment for the test*
2. *tests* the unit
3. *tears down* the environment again

*Tests are organized into suites*
TESTING A URL CLASS

http://www.askigor.org/status.php?id=sample
package junitexample;

public class Calculator {
    int add(int value1, int value2) {
        return value1 + value2;
    }
    int subtract(int value1, int value2) {
        return value1 - value2;
    }
    int multiply(int value1, int value2) {
        return value1 * value2;
    }
    int divide(int value1, int value2) {
        return value1 / value2;
    }
}
package junitexample;

import junit.framework.TestCase;

public class CalculatorTest extends TestCase {
    private Calculator calc;
    public CalculatorTest(String s){
        super(s);
    }
    // called before each test
    protected void setUp() throws Exception {
        super.setUp();
        calc = new Calculator();
    }
    // called after each test
    protected void tearDown() throws Exception {
        super.tearDown();
    }
    ...
    // test for the add() method
    public final void testAdd() {
        assertEquals(calc.add(20, 30), 50);
    }
    // test for the subtract() method
    public final void testSub() {
        assertEquals(calc.subtract(20, 10), 10);
    }
    // test for the multiply() method
    public final void testMult() {
        assertEquals(calc.multiply(9, 11), 99);
    }
    // test for the divide() method
    public final void testDiv() {
        assertEquals(calc.divide(18, 2), 9);
    }
    ...
}
JUnit integration in Eclipse
writing tests before you implement functionality involves extra effort, but…

… it forces you to think about the problem you are trying to solve more concretely
	-
and formulate a solution more quickly

…and you will regain the time spent on unit tests by catching problems early
	-
and reduce time spent later on debugging
RECOMMENDATIONS FOR WRITING GOOD TESTS

- write tests that cover a partition of the input space, and that cover specific features
- achieve good code coverage
- create an automated, fast running test suite, and use it all the time
- have tests that cover your system’s tests at different levels of functionality
- set up your tests so that, when a failure occurs, it pinpoints the issue so that it does not require much further debugging
EXTRA
TESTING ENVIRONMENTS ARE OFTEN COMPLEX

- Millions of configurations
- Testing on dozens of different machines
- All needed to find and reproduce problems
DEFECT SEVERITY

- An assessment of a defect’s impact
- Can be a major source of contention between dev and test

**Critical** *Show stopper.* The functionality cannot be delivered unless that defect is cleared. It does not have a workaround.

**Major** Major flaw in functionality but it still can be released. There is a workaround; but it is not obvious and is difficult.

**Minor** Affects minor functionality or non-critical data. There is an easy workaround.

**Trivial** Does not affect functionality or data. It does not even need a workaround. It does not impact productivity or efficiency. It is merely an inconvenience.