Functional Testing

Software Engineering
Andreas Zeller • Saarland University
Testing
Even more Testing
Testing
Software is manifold
Software is manifold
Software is manifold
Software is manifold
Software is manifold
Software is manifold
Testing

Configurations
What to test?
Dijkstra’s Curse

Testing can only find the presence of errors, not their absence.
Formal Verification

Configurations
Formal Verification

Abstraction

Configurations
Formal Verification

Abstraction

Configurations
Zeller’s Variation on Dijkstra

Verification can only find the *absence* of errors, but never their *presence*
The Best of two Worlds

Abstraction

Configurations
What to test?
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

Functional
  “black box”

Structural
  “white box”
Why Functional?

- Program code not necessary

- Early functional test design has benefits
  reveals spec problems • assesses testability • gives additional explanation of spec • may even serve as spec, as in XP
Why Functional?

• 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
Angle

\[ 2^{32} = 4.294.967.296 \]

different values

Force

\[ 2^{32} = 4.294.967.296 \]

different values
$2^{32} = 4.294.967.296$

different values

$2^{32} = 4.294.967.296$

different values

$\times$

$2^{64} = 18.446.744.073.709.551.616$

different runs
9.223.372.036.854.775.808 Minutes
18.446.744.073.709.551.616
×

1 Minute
Systematic Functional Testing

- Functional specification
- Independently testable feature
- Representative values
- Model
- Test case specifications
- Test case
- Systematic Functional Testing
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
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;
}

• What are the independently testable features?
Testable Features

- Consider a multi-function calculator
- What are the independently testable features?
Testable Features

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

Identify
- Identify
- Identify
- Identify
- Derive
- Derive
- Derive

Generate
Representative Values

- 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
Needles in a Haystack

• To find needles, look systematically

• We need to find out what makes needles special
Systematic Partition Testing

- **Failure (valuable test case)**
- **No failure**

The space of possible input values (the haystack)

- Failures are sparse in the space of possible inputs...
- ...but dense in some parts of the space

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

Functional testing is one way of drawing orange 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>
Boundary Analysis

- Possible test case

- Test at *lower range* (valid and invalid), at *higher range* (valid and invalid), and at *center*
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

• How about a ZIP code that reads
  
  12345'; DROP TABLE orders; SELECT *
  FROM zipcodes WHERE 'zip' = '

• Or a ZIP code with 65536 characters…

• This is security testing
Gutjahr’s Hypothesis

Partition testing is more effective than random testing.
Representative Values

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

Procedures:
- Identify
- Derive
- Generate
Model-Based Testing

• Have a formal model that specifies software behavior
  
  • Models typically come as
    • finite state machines and
    • decision structures
Finite State Machine
**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

- **Path coverage:** Tests cover every path
  Not feasible in practice due to infinite number of paths

- **State coverage:** Every node is executed
  A minimum testing criterion

- **Transition coverage:** Every edge is executed
  Typically, a good coverage criterion to aim for
Transition Coverage
State-based Testing

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

Figure 14.3 State diagram for Account class (adapted from [KIR94])
### Decision Tables

<table>
<thead>
<tr>
<th></th>
<th>Education</th>
<th>Individual</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Education account</strong></td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td><strong>Current purchase &gt; Threshold 1</strong></td>
<td>–</td>
<td>F</td>
</tr>
<tr>
<td><strong>Current purchase &gt; Threshold 2</strong></td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td><strong>Special price &lt; scheduled price</strong></td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td><strong>Special price &lt; Tier 1</strong></td>
<td>–</td>
<td>F</td>
</tr>
<tr>
<td><strong>Special price &lt; Tier 2</strong></td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td><strong>Out</strong></td>
<td>Edu discount</td>
<td>Special price</td>
</tr>
</tbody>
</table>
Condition Coverage

• **Basic criterion**: Test every column
  “Don’t care” entries (−) can take arbitrary values

• **Compound criterion**: Test every combination
  Requires $2^n$ tests for $n$ conditions and is unrealistic

• **Modified condition decision criterion (MCDC)**: like basic criterion, but additionally, modify each T/F value at least once such that the outcome changes
  Again, a good coverage criterion to aim for
## MCDC Criterion

<table>
<thead>
<tr>
<th></th>
<th>Education</th>
<th>Individual</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Education account</strong></td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>Current purchase &gt; Threshold 1</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Current purchase &gt; Threshold 2</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Special price &lt; scheduled price</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>Special price &lt; Tier 1</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Special price &lt; Tier 2</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Out</td>
<td>Edu discount</td>
<td>Special price</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>S</td>
</tr>
</tbody>
</table>
## MCDC Criterion

<table>
<thead>
<tr>
<th></th>
<th>Education</th>
<th>Individual</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Education account</strong></td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td><strong>Current purchase &gt;</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Threshold 1</strong></td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td><strong>Current purchase &gt;</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Threshold 2</strong></td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td><strong>Special price &lt;</strong></td>
<td>T</td>
<td></td>
</tr>
<tr>
<td><strong>scheduled price</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Special price &lt;</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Tier 1</strong></td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td><strong>Special price &lt;</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Tier 2</strong></td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td><strong>Out</strong></td>
<td>Edu discount</td>
<td>Special price</td>
</tr>
</tbody>
</table>
## MCDC Criterion

<table>
<thead>
<tr>
<th></th>
<th>Education</th>
<th>Individual</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Education account</strong></td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td><strong>Current purchase &gt;</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Threshold 1</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td><strong>Current purchase &gt;</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Threshold 2</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td><strong>Special price &lt;</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>scheduled price</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td><strong>Special price &lt;</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tier 1</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td><strong>Special price &lt;</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tier 2</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Edu discount</th>
<th>Special price</th>
<th>No discount</th>
<th>Special price Tier 1 discount</th>
<th>Special price Tier 2 discount</th>
<th>Special price Tier 2 discount</th>
<th>Special price Tier 2 discount</th>
<th>Special price Tier 2 discount</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Out</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## MCDC Criterion

<table>
<thead>
<tr>
<th></th>
<th>Education</th>
<th>Individual</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Education account</strong></td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td><strong>Current purchase &gt; Threshold 1</strong></td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td><strong>Current purchase &gt; Threshold 2</strong></td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td><strong>Special price &lt; scheduled price</strong></td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td><strong>Special price &lt; Tier 1</strong></td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td><strong>Special price &lt; Tier 2</strong></td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td><strong>Out</strong></td>
<td>Edu discount</td>
<td>Special price</td>
</tr>
</tbody>
</table>

- T: True
- F: False
- Special price:
  - Tier 1 discount
  - Tier 2 discount
  - Special Price
Weyuker’s Hypothesis

The adequacy of a coverage criterion can only be intuitively defined.
Learning from the past
Pareto’s Law

Approximately 80% of defects come from 20% of modules
Model-Based Testing

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

Arrows indicate:
- identify
- derive
- generate
Deriving Test Case Specs

• Input values enumerated in previous step
• Now: need to take care of combinations
• Typically, one uses models and representative values to generate test cases

Diagram:
- Test case specifications
- Test case
- Model
- Representative values
  - identify
  - derive
  - generate
Combinatorial Testing

Server

Apache

MySQL

Oracle

OS

Windows

Linux

Database
Combinatorial Testing

• Eliminate invalid combinations
  IIS only runs on Windows, for example

• Cover *all pairs* of combinations
  such as MySQL on Windows and Linux

• Combinations typically generated
  automatically
  and – hopefully – tested automatically, too
Pairwise Testing
Testing environment

- Millions of configurations
- Testing on dozens of different machines
- All needed to find & reproduce problems
Deriving Test Case Specs

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

Arrows indicate:
- Identify
- Derive
- Generate
Deriving Test Cases

- Implement test cases in code
- Requires building *scaffolding* – i.e., drivers and stubs
Unit Tests

• Directly access units (= classes, modules, components…) at their programming interfaces

• Encapsulate a set of tests as a single syntactical unit

• Available for all programming languages (JUnit for Java, CPPUNIT for C++, etc.)
A test case…

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

http://www.askigor.org/status.php?id=sample
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

public class URLTest extends TestCase {
    private URL askigor_url;

    // Create new test
    public URLTest(String name) { super(name); }

    // Assign a name to this test case
    public String toString() { return getName(); }

    // Setup environment
    protected void setUp() {
        askigor_url = new URL("http://www.askigor.org/" + "status.php?id=sample");
    }

    // Release environment
    protected void tearDown() { askigor_url = null; }
}
// Test for protocol (http, ftp, etc.)
public void testProtocol() {
    assertEquals(askigor_url.getProtocol(), "http");
}

// Test for host
public void testHost() {
    int noPort = -1;
    assertEquals(askigor_url.getHost(), "www.askigor.org");
    assertEquals(askigor_url.getPort(), noPort);
}

// Test for path
public void testPath() {
    assertEquals(askigor_url.getPath(), "/status.php");
}

// Test for query part
public void testQuery() {
    assertEquals(askigor_url.getQuery(), "id=sample");
}
// Set up a suite of tests
public static Test suite() {
    TestSuite suite = new TestSuite(URLTest.class);
    return suite;
}

// Main method: Invokes GUI
public static void main(String args[]) {
    String[] testCaseName = {
        URLTest.class.getName()
    };
    // junit.textui.TestRunner.main(testCaseName);
    junit.swingui.TestRunner.main(testCaseName);
    // junit.awtui.TestRunner.main(testCaseName);
}

JUnit
Deriving Test Cases

1. **Functional specification**
2. **Independently testable feature**
3. **Representative values**
4. **Model**
5. **Test case specifications**
6. **Test case**

- **Identify**
- **Derive**
- **Generate**
Systematic Functional Testing

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

Arrows indicate:
- Identify
- Derive
- Generate
Systematic Functional Testing

- Functional specification
  - Identify
  - Independently testable feature
  - Derive
  - Representative values
  - Model
- Test case
  - Generate
  - Test case specifications

Systematic Partition Testing
- Failure (valuable test case)
- No failure
- Failures are sparse in the space of possible inputs...
- ...but dense in some parts of the space
- The space of possible input values (the haystack)
- Functional testing is one way of drawing orange lines to isolate regions with likely failures

Testing Tactics
- Functional "black box"
- Structural "white box"
- MC/DC Criterion

MC/DC Criterion

<table>
<thead>
<tr>
<th></th>
<th>Education</th>
<th>Individual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Education</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>Current price &gt; Threshold 1</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Current price &gt; Threshold 2</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Special price &lt; Tier 1</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>Special price &lt; Tier 2</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Out</td>
<td>Edu discount</td>
<td>Special price</td>
</tr>
</tbody>
</table>