Software Testing
Lecture Notes 1 (3)

“no issue is meaningful unless it can be put to the test of
decisive verification.”

C.S. Lewis, 1934

“in the beginning of a malady it is easy to cure but difficult
to detect, but in the course of time, not having been either
detected or treated in the beginning, it becomes easy to
detect but difficult to cure.”

Nicolo Machiavelli
*The Prince*, 1513

Contents

• Introduction, Testing process
• Unit Testing:
  – Black-box Testing
  – White-box Testing
• Integration Testing
• System Testing
• Acceptance Testing
Triangle program (simple version)

• The triangle problem is the most widely used example in software testing literature.

• The program accepts three integers, $a$, $b$, and $c$ as input. The three values are interpreted as representing the lengths of sides of a triangle. The program prints a message that states whether the triangle is scalene (oregelbunden), isosceles (likbent) or equilateral (liksidig).

• On a sheet of paper, write a set of test cases (i.e., specific sets of data) that you feel would adequately test this program.

Set of test cases for Triangle program

<table>
<thead>
<tr>
<th>Test case</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>Expected output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>isosceles (likbent)</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Triangle program (improved version)

The program accepts three integers, $a$, $b$, and $c$ as input. The three values are interpreted as representing the lengths of sides of a triangle. The integers $a$, $b$, and $c$ must satisfy the following triangle property (the sum of any pair of sides must be greater than the third side).

$$a < b + c$$

Testing a ballpoint pen

- Does the pen write in the right color, with the right line thickness?
- Is the logo on the pen according to company standards?
- Is it safe to chew on the pen?
- Does the click-mechanism still work after 100,000 clicks?
- Does it still write after a car has run over it?

What is expected from this pen?

**Intended use!!**
Customer Developer

Requirements definition
Requirements specification

Functional requirements
Nonfunctional requirements

Code = System

Design Specification

Error, Fault, Failure

Human error (Mistake, Bug)
Can lead to
Fault (Defect, Bug)
Can lead to
Failure
Types of Faults
(dep. on org. IBM, HP)

- Algorithmic: division by zero
- Computation & Precision: order of op
- Documentation: doc - code
- Stress/Overload: data-str size (dimensions of tables, size of buffers)
- Capacity/Boundary: x devices, y parallel tasks, z interrupts
- Timing/Coordination: real-time systems
- Throughout/Performance: speed in req.
- Recovery: power failure
- Hardware & System Software: modem
- Standards & Procedure: organizational standard; difficult for programmers to follow each other.

A Testing Life Cycle

Putting Bugs IN
Development phases

Finding Bugs
Testing phase

Getting Bugs OUT
Program Behaviors

Specification (expected) ∩ Program (observed) = "Correct" Portion

Specification (expected) ⊆ Program (observed)

Missing Functionality (sins of omission)

Extra Functionality (sins of commission)

Testing Program Behavior

Specification (expected) ∩ Program (observed) = Test Cases (verified)
Basic Approaches

Specification

Program

Functional
(Black Box)
establishes confidence

Structural
(White Box)
seeks faults

April 2007 PUM, Mariam Kamkar, IDA, LiU

Black Box

Inputs

Outputs

Function is understood only in terms of it's inputs and outputs, with no knowledge of its implementation.
Types (strategy) of testing

- **Black-box**: a strategy in which testing is based on requirements and specifications.

- **White-box**: a strategy in which testing is based on internal paths, structure, and implementation.

- **Gray-box**: peek into the “box” just long enough to understand how it has been implemented.

Function is understood only in terms of its implementation.
Contents of a Test Case

"Boilerplate" author, date, purpose, test case ID
Pre-conditions (including environment)
Inputs
Expected Outputs
Observed Outputs
Pass/Fail

Testing level

• Unit testing
• Integration testing
• System testing
• Acceptance testing
Unit & Integration Testing

Objective: to ensure that code implemented the design properly.

Code = System

Design Specification

April 2007  PUM, Mariam Kamkar, IDA, LiU

20
Unit Testing

- Black-box Testing
- White-box Testing

Test Object

Input → Test Object → Output

Oracle

Failure?
Two Types of Oracles

• **Human**: an expert that can examine an input and its associated output and determine whether the program delivered the correct output for this particular input.

• **Automated**: a system capable of performing the above task.

Black-box / Closed-box Testing

• incorrect or missing functions
• interface errors
• performance error
Block-Box Testing Techniques

• **Definition**: a strategy in which testing is based on requirements and specifications.

• **Applicability**: all levels of system development
  - Unit
  - Integration
  - System
  - Acceptance

• **Disadvantages**: never be sure of how much of the system under test (SUT) has been tested.

• **Advantages**: directs tester to choose subsets to tests that are both efficient and effective in finding defects.

---

Black-box Testing

1. Exhaustive testing
2. Equivalence class testing (Equivalence Partitioning)
3. Boundary value analysis
4. Decision table testing
5. Use case testing
1. Exhaustive testing

- **Definition**: testing with every member of the input value space.

- Input value space: the set of all possible input values to the program.

2. Equivalence Class Testing

- Equivalence Class (EC) testing is a technique used to reduce the number of test cases to a manageable level while still maintaining reasonable test coverage.

- Each EC consists of a set of data that is treated the same by the module or that should produce the same result. Any data value within a class is *equivalent*, in terms of testing, to any other value.
Identifying the Equivalence Classes

Taking each input condition (usually a sentence or phrase in the specification) and partitioning it into two or more groups:

- Input condition
  - range of values \( x: 1-50 \)
- Valid equivalence class
  - \( 1 < x < 50 \)
- Invalid equivalence classes
  - \( x < 1 \)
  - \( x > 50 \)

Guidelines

1. If an input condition specifies a range of values; identify one valid EC and two invalid EC.
2. If an input condition specifies the number (e.g., one through 6 owners can be listed for the automobile); identify one valid EC and two invalid EC (- no owners; - more than 6 owners).
3. If an input condition specifies a set of input values and there is reason to believe that each is handled differently by the program; identify a valid EC for each and one invalid EC.
4. If an input condition specifies a “must be” situation (e.g., first character of the identifier must be a letter); identify one valid EC (it is a letter) and one invalid EC (it is not a letter)
5. If there is any reason to believe that elements in an EC are not handled in an identical manner by the program, split the equivalence class into smaller equivalence classes.
Identifying the Test Cases

1. Assign a unique number to each EC.

2. Until all valid ECs have been covered by test cases, write a new test case covering as many of the uncovered valid ECs as possible.

3. Until all invalid ECs have been covered by test cases, write a test case that cover one, and only one, of the uncovered invalid ECs.

Applicability and Limitations

• Most suited to systems in which much of the input data takes on values within ranges or within sets.

• It makes the assumption that data in the same EC is, in fact, processed in the same way by the system. The simplest way to validate this assumption is to ask the programmer about their implementation.

• EC testing is equally applicable at the unit, integration, system, and acceptance test levels. All it requires are inputs or outputs that can be partitioned based on the system’s requirements.
Equivalence partitioning

Specification: the program accepts four to eight inputs which are 5 digit integers greater than 10000.
3. Boundary Value Testing

Boundary value testing focuses on the boundaries simply because that is where so many defects hide. The defects can be in the requirements or in the code.

The most efficient way of finding such defects, either in the requirements or the code, is through inspection (Software Inspection, Gilb and Graham’s book).

Technique

1. Identify the ECs.

2. Identify the boundaries of each EC.

3. Create test cases for each boundary value by choosing one point on the boundary, one point just below the boundary, and one point just above the boundary.
### Boundary Value Analysis

#### Test Cases for Triangle Program (upper bound 200)

<table>
<thead>
<tr>
<th>Test case</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>Expected output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>100</td>
<td>1</td>
<td>isosceles (likbent)</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>100</td>
<td>2</td>
<td>isosceles</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>100</td>
<td>199</td>
<td>isosceles</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>100</td>
<td>200</td>
<td>not a triangle</td>
</tr>
<tr>
<td>…</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Applicability and Limitations

Boundary value testing is equally applicable at the unit, integration, system, and acceptance test levels. All it requires are inputs that can be partitioned and boundaries that can be identified based on the system’s requirements.

4. Decision Table Testing

Decision tables are an excellent tool to capture certain kinds of system requirements and to document internal system design. They are used to record complex business rules that a system must implement.

In addition, they can serve as a guide to creating test cases.
The general format of a decision table:

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Rule 1</th>
<th>Rule 2</th>
<th>…</th>
<th>Rule P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condition-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Condition-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>…</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Condition-m</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Actions</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Action-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Action-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>…</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Action-n</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A decision table converted to a test case table:

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Test Case 1</th>
<th>Test Case 2</th>
<th>…</th>
<th>Test Case P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condition-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Condition-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>…</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Condition-m</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expected Results</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Action-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Action-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>…</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Action-n</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


**A decision table with "don’t care" entry**

<table>
<thead>
<tr>
<th></th>
<th>Rule 1</th>
<th>Rule 2</th>
<th>Rules 3,4</th>
<th>Rule 5</th>
<th>Rule 6</th>
<th>Rules 7,8</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>C2</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>C3</td>
<td>T</td>
<td>F</td>
<td>_</td>
<td>T</td>
<td>F</td>
<td>_</td>
</tr>
<tr>
<td>A1</td>
<td>X</td>
<td>X</td>
<td>_</td>
<td>X</td>
<td>_</td>
<td>_</td>
</tr>
<tr>
<td>A2</td>
<td>X</td>
<td>_</td>
<td>_</td>
<td>X</td>
<td>_</td>
<td>_</td>
</tr>
<tr>
<td>A3</td>
<td>X</td>
<td>X</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
</tr>
<tr>
<td>A4</td>
<td>X</td>
<td>X</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
</tr>
</tbody>
</table>

• _ : “don’t care” entry. The don’t care entry has two major interpretations: the condition is irrelevant, or the condition does not apply. Sometimes the “n/a” symbol for this latter interpretation.

• **Limited entry decision tables**: all the conditions are binary.

• **Extended entry decision tables**: conditions are allowed to have several values.

• Decision tables are deliberately declarative (as opposed to imperative); no particular order is implied by the conditions, and selected actions do not occur in any particular order.

---

**Triangle program (new conditions)**

The program accepts three integers, \(a\), \(b\), and \(c\) as input. The three values are interpreted as representing the lengths of sides of a triangle. The integers \(a\), \(b\), and \(c\) must satisfy the following conditions:

- **C1**: \(1 \leq a \leq 200\)
- **C2**: \(1 \leq b \leq 200\)
- **C3**: \(1 \leq c \leq 200\)
- **C4**: \(a < b + c\)
- **C5**: \(b < a + c\)
- **C6**: \(c < a + b\)
**Decision table for the Triangle problem**

<table>
<thead>
<tr>
<th>Rule 1</th>
<th>Rule 2</th>
<th>Rule 3</th>
<th>Rule 4</th>
<th>Rule 5</th>
<th>Rule 6</th>
<th>Rule 7</th>
<th>Rule 8</th>
<th>Rule 9</th>
<th>Rule 10</th>
<th>Rule 11</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1: a &lt; b + c?</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>C2: b &lt; a + c?</td>
<td>_</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>C3: c &lt; a + b?</td>
<td>_</td>
<td>_</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>C4: a = b?</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>C5: a = c?</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>C6: b = c?</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

**Choice of conditions:** Here we expand the old condition (C1: a, b, c form a triangle?) to a more detailed view of the three inequalities of the triangle property. If any one of these fails, the three integers do not constitute sides of a triangle.
Applicability and Limitations

Decision table testing can be used whenever the system must implement complex business rules when these rules can be represented as a combination of conditions and when these conditions have discrete actions associated with them.

5. Use Case Testing

- Test cases that exercise a system’s functionalities from start to finish.

- Use cases were created by Ivar Jacobsen in his book Object-Oriented Software Engineering: A use case driven approach.
Unified Modeling Language Notation

- **Use case**: a scenario that describes the use of a system by an actor to accomplish a specific goal.
- **Actor**: a user, playing a role with respect to the system, seeking to use the system to accomplish something worthwhile within a particular context.
- **Scenario**: a sequence of steps that describe the interactions between the actor and the system.
- The set of test cases makes up the functional requirements of a system.

---

Applicability and Limitations

- **System and acceptance testing.**
Black-box Testing

Summary

1. Exhaustive testing
2. Equivalence class testing (Equivalence Partitioning)
3. Boundary value analysis
4. Decision table testing
5. Use case testing

Testing level

Summary

- Unit testing
- Integration testing
- System testing
- Acceptance testing