Software Testing

Kristian Sandahl (slides by Mariam Kamkar)
Department of Computer and Information Science
2014-09-23
Requirements

System Design
(Architecture, High-level Design)

Module Design
(Program Design, Detailed Design)

Implementation of Units (classes, procedures, functions)

Unit testing

Module Testing
(Integration testing of units)

System Testing
(Integration testing of modules)

Acceptance Test
(Release testing)

Validation Requirements, Verify Specification

Verify System Design

Verify Module Design

Verify Implementation

Project Management, Software Quality Assurance (SQA), Supporting Tools, Education
How do you test 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!!**
Testing software

- Are the functions giving correct output?
- Are the integrated modules giving correct output?
- Is the entire system giving correct output when used?
- Is the correct output given in reasonable time?
- Is the output presented in an understandable way?
- Was this what we really expected?

- Software testing is an activity in which a program is executed under specified conditions, the results are observed, and an evaluation is made of the program.
Other methods for Validation & Verification

- Formal verification – read about Z in the book
- Model checking
- Prototyping
- Simulation
- Software reviews – lecture to come
• ”Testing shows the presence, not the absence of bugs“

(Edsger Wybe Dijkstra)

• …but you might use experience and statistics to make some kind of assessment.
Basic Definitions – lecture notes

The terminology here is taken from standards developed by the Institute of Electronics and Electrical Engineers (IEEE) Computer Society.

- **Error**: people make errors. A good synonym is mistake. When people make mistakes while coding, we call these mistakes bugs. Errors tend to propagate; a requirements error may be magnified during design and amplified still more during coding.

- **Fault**: a fault is the result of an error. It is more precise to say that a fault is the representation of an error, where representation is the mode of expression, such as narrative text, data flow diagrams, hierarchy charts, source code, and so on. Defect is a good synonym for fault, as is bug. Faults can be elusive. When a designer makes an error of omission, the resulting fault is that something is missing that should be present in the representation. We might speak of faults of commission and faults of omission. A fault of commission occurs when we enter something into a representation that is incorrect. Faults of omission occur when we fail to enter correct information. Of these two types, faults of omission are more difficult to detect and resolve.

- **Failure (anomaly)**: a failure occurs when a fault executes. Two subtleties arise here: one is that failures only occur in an executable representation, which is usually taken to be source code, or more precisely, loaded object; the second subtlety is that this definition relates failures only to faults of commission. How can we deal with failures that correspond to faults of omission?
Error, Fault, Failure

Human error (Mistake, Bug) can lead to Fault (Defect, Bug) which can lead to Failure.
The V-model from the tester perspective

Putting Bugs IN Development phases

Finding Bugs Testing phase

Getting Bugs OUT

LiU EXPANDING REALITY
Program Behaviors

Specification (expected)

"Correct" Portion

Program (observed)

Missing Functionality (sins of omission)

Extra Functionality (sins of commission)
Basic Approaches

**Specification**

R1: Given *input*, the software shall provide *output*.

**Program**

Find *input* and *output* so that *X* is executed.

**Functional (Black Box) establishes confidence**

**Structural (White Box) seeks faults**
Types of Faults

• Discuss in pairs:

• What types of faults do you typically find in your software?

• How can you test to find them?
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
- Throughput/Performance: speed in req.
- Recovery: power failure
- Hardware & System Software: modem
- Standards & Procedure: organizational standard; difficult for programmers to follow each other.
Faults classified by severity
(Beizer, 1984)

1. **Mild**  Misspelled word
2. **Moderate**  Misleading or redundant information
3. **Annoying**  Truncated names, bill for $0.00
4. **Disturbing**  Some transaction(s) not processed
5. **Serious**  Lose a transaction
6. **Very serious**  Incorrect transaction execution
7. **Extreme**  Frequent "very serious" errors
8. **Intolerable**  Database corruption
9. **Catastrophic**  System shutdown
10. **Infectious**  Shutdown that spreads to others
Contents of a Test Case

"Boilerplate": author, date, purpose, test case ID

Pre-conditions (including environment)

Inputs

Expected Outputs

Observed Outputs

Pass/Fail
Testing levels

- **Requirements**
  - Validate Requirements, Verify Specification
  - Acceptance Test (Release testing)

- **System Design**
  - System Testing (Integration testing of modules)
  - Verify System Design

- **Module Design**
  - Module Testing (Integration testing of units)
  - Verify Module Design

- **Implementation of Units (classes, procedures, functions)**
  - Unit testing
  - Verify Implementation
### Test table

<table>
<thead>
<tr>
<th>Id</th>
<th>Advanced course credits in Computer Science</th>
<th>Advanced course credits in total</th>
<th>Masters thesis in subject</th>
<th>Total number of credits</th>
<th>M.Sc., Computer Science</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td>120</td>
<td>Computer sc.</td>
<td>120</td>
<td>No</td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td>90</td>
<td>Computer sc.</td>
<td>120</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>90</td>
<td>Physics</td>
<td>120</td>
<td>No</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Can be written from specification
Unit Testing

Objective: to ensure that code implemented the design properly.

Often done by the programmers themselves.
The oracle problem

Input

Test

Failure?

Oracle

Object

Output
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.

R2: “The answer is 42.”
Black-box Testing

1. Exhaustive testing
2. Equivalence class testing (Equivalence Partitioning)
3. Boundary value analysis
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.

- Sum of two 16 bit integers: $2^{32}$ combinations

- One test per ms takes about 50 days.
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

- Invalid equivalence classes
Two-variable example

• Validate loan application forms against the rule:
  • *If you are 18 years and older, you can borrow maximally 100.000, but not less than 10.000.*
  • Variable: age
  • EC1: age < 18
  • EC2: age >= 18
  • Variable: sum
  • EC3: sum < 10.000
  • EC4: 10.000 <= sum <= 100.000
  • EC5: sum > 100.000
Two-variable example, test-cases

<table>
<thead>
<tr>
<th>Test-case id</th>
<th>Age</th>
<th>Sum</th>
<th>Valid form</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32</td>
<td>55.300</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
<td>72.650</td>
<td>No</td>
</tr>
<tr>
<td>3</td>
<td>44</td>
<td>9.875</td>
<td>No</td>
</tr>
<tr>
<td>4</td>
<td>50</td>
<td>60.000</td>
<td>Yes</td>
</tr>
<tr>
<td>5</td>
<td>87</td>
<td>103.800</td>
<td>No</td>
</tr>
</tbody>
</table>

Arbitrary, valid sums

Arbitrary, valid ages
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.
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.
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.
Specification: the program accepts four to eight inputs which are 5 digit integers greater than or equal to 10000.

<table>
<thead>
<tr>
<th>Input values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 10000</td>
</tr>
<tr>
<td>Between 10000 and 99999</td>
</tr>
<tr>
<td>More than 99999</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of input values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 4</td>
</tr>
<tr>
<td>Between 4 and 8</td>
</tr>
<tr>
<td>More than 8</td>
</tr>
</tbody>
</table>
Boundary value analysis

- **Less than 10000**
- **Between 10000 and 99999**
- **More than 99999**
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.
xUnit

- xUnit is a set of tools for regression testing
- x denotes a programming language
- JUnit, for Java is one of the earliest and most popular
- TDDC88 has a lab – do that

- Recommended primer:

  http://www.it-c.dk/~lthorup/JUnit Primer.html
Object Oriented Framework Development

by Marcus Eduardo Markiewicz and Carlos J.P. Lucena
JUnit interface

```java
assertEquals("10 times 0 must be 0", 0, test.imul(10, 0));
```

message if fail  
expected  
actual

Source: Plantir dev zone
Test-Driven Development (TDD)

1. Write a test that fails
2. Make the code work
3. Eliminate redundancy

The mantra of Test-Driven Development (TDD) is “red, green, refactor.”

source: Redmond Developer
Integration testing
Integration Testing strategies

1. Big-bang
2. Bottom-up
3. Top-down
4. Sandwich
Three level functional decomposition tree
Big-Bang testing

Environment:
A, B, C, D, E, F, G, H

System-wide test

Unit test A

Unit test B

... 

Unit test H
Driver

- A pretend module that requires a sub-system and passes a test case to it

setup driver
SUT(x)
verification

SUT

Black-box view

System Under Test
Bottom-up testing

Diagram showing a hierarchical structure with nodes labeled A, B, C, D, E, F, G, H, and arrows indicating the flow from Level 1 to Level 2 to Level 3.
Is bottom-up smart?

- If the basic functions are complicated, error-prone or has development risks
- If bottom-up development strategy is used
- If there are strict performance or real-time requirements

Problems:
- Lower level functions are often off-the shelf or trivial
- Complicated User Interface testing is postponed
- End-user feed-back postponed
- Effort to write drivers.
Stub

- A program or a method that simulates the input-output functionality of a missing subsystem by answering to the decomposition sequence of the calling sub-system and returning back simulated or "canned" data.

```
Service(x)
Stub
Check x
Return y;
end
```
Top-down testing

A, B, C, D

A, B, E, F, C, D, G, H
Is top-down smart?

- Test cases are defined for functional requirements of the system
- Defects in general design can be found early
- Works well with many incremental development methods
- No need for drivers

Problems:
- Technical details postponed, potential show-stoppers
- Many stubs are required
- Stubs with many conditions are hard to write
Sandwich testing

Target level

A, B, C, D
E, F, B
G, H, D

A, B, E, F, C, D, G, H
System Testing
System functional requirements

Function test

Functioning systems

Performance test

Other software requirements

Verified validated software

Integrated modules

Customer requirements spec.

Acceptance test

Accepted system

Installation test

User environment

System In Use!
Function testing/Thread testing

*(testing one function at a time)*

**functional requirements**

A function test checks that the integrated system performs its function as specified in the requirement

- **Guidelines**
  - use a test team *independent* of the designers and programmers
  - know the expected actions and output
  - test both valid and invalid input
  - never modify the system just to make testing easier
  - have stopping criteria
Performance Testing
nonfunctional requirements

• Stress tests
• Volume tests
• Configuration tests
• Compatibility tests
• Regression tests
• Security tests
• Timing tests

• Environment tests
• Quality tests
• Recovery tests
• Maintenance tests
• Documentation tests
• Human factors tests / usability tests
Software reliability engineering

- Define target failure intensity
- Develop operational profile
- Plan tests
- Execute test
- Apply data to decisions
Acceptance Testing

OK
Acceptance Testing

customers, users need

- **Benchmark test**: a set of special test cases

- **Pilot test**: everyday working
  - Alpha test: at the developer’s site, controlled environment
  - Beta test: at one or more customer site.

- **Parallel test**: new system in parallel with previous one
Installation Testing

users site

Acceptance test at developers site

‡ installation test at users site,
otherwise installation test might not be needed!
Termination Problem
How decide when to stop testing

- The main problem for managers!
- Termination takes place when
  - resources (time & budget) are over
  - found the seeded faults
  - some *coverage* is reached
Control-flow based coverage

Statement coverage

All statements executed
Control-flow based coverage

Branch coverage

All decision branches tried
Control-flow based coverage

Full path coverage

All possible paths executed
GUI Testing

• GUI application is **event driven**; users can cause any of several events in **any order**

• GUI applications offer one small benefit to testers:
  • There is a **little need** for integration testing

• Unit testing is typically at the “button level”; that is **buttons have functions**, and these can be tested in the usual **unit-level sense**.

• The essence of **system-level testing** for GUI applications is to exercise the event-driven nature of application

A wide range of GUI testing tools has appeared on the market over the past few years.

TDDC88 has a lab on Selenium
Smoke test

- Important selected tests on module, or system
- Possible to run fast
- Build as large parts as possible as often as possible
- Run smoke tests to make sure you are on the right way