Defect testing

- Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user.
Objectives

• To understand testing techniques that are geared to discover program faults
• To introduce guidelines for interface testing
• To understand the principles of CASE tool support for testing
Topics covered

- Defect testing
  - Exhaustive Testing
  - Black-box testing
  - Equivalence partitioning
  - White-box ‘Structural’ testing
  - Path testing
- Integration testing
- Interface testing
- Stress testing
- Object-oriented (OO) testing
- Monkey testing
- Testing workbenches
- Automated testing – See JUnit & XUint frameworks
What Testing Shows

- errors
- requirements conformance
- performance
- an indication of quality
Who Tests the Software

**Developer**
- understands the system
- has the source code
- white-box ‘Unit’ testing
- will test "gently"
- driven by delivery ‘schedule’ constraint

**Independent tester**
- must learn about the system
- has no source code
- black-box ‘Acceptance’ testing
- will attempt to break the sys (ME!!)
- driven by quality constraint
Testing phases

Component (Unit) testing

Integration testing

Software developer

Independent testing team
The testing process

• Component (Unit) testing: needs source code (White-box)
  - Testing of individual program components
  - Usually the responsibility of the component developer
  - Tests are derived from the developer’s experience

• Integration testing (Acceptance testing, Black-box testing)
  - Testing of groups of components integrated to create a system or sub-system
  - The responsibility of an independent testing team
  - Tests are based on a system specification (Acceptance testing)
Exhaustive Testing

If there are $10^{14}$ possible paths and we execute one test per millisecond, it would take 3,170 years to test this program!!
Test Case Design

"Bugs lurk in corners and congregate at boundaries ..."

Boris Beizer

OBJECTIVE  to uncover errors
CRITERIA    in a complete manner
CONSTRAINT  with a minimum of effort and time
Test data and test cases

• **Test data**
  - Inputs to test the system

• **Test cases ‘Test scenario’ includes:**
  - Inputs to test the system
  - and the **predicted outputs**
The defect testing process

![Diagram of the defect testing process]

- **Process**
  - Design test cases
  - Prepare test data
  - Run program with test data
  - Compare results to test cases
  - Test results
- **Deliverable**
  - Test reports
  - Test cases
  - Test data
  - Specs of test input & predicted sys output

Fig 20.2 Sommerville
Black-box testing

- Program is considered as a ‘black-box’
- No need to know or access source code
- Functionality testing
- No implementation testing (implementation testing needs source code)
- Test cases are based on the system specification
- Test planning can begin early in the software process
Black-box Testing
“Functionality testing”

Input: Test Case

Program
“No need for source code”

Sys Input

Sys Output

Predicted sys output

Compare

Output: Test Report
Black-box Testing

- Input test data
- System / component
- Output test results

Input causes anomalous behavior
Compare output with predicted values
Outputs reveal presence of defects
Equivalence partitioning

- Objective: Reduce the number of test cases
Equivalence partitioning - cont.

- A way to derive test cases
- Is a **black-box** testing method
- Input data with common characteristics (positive numbers, negative numbers, strings without blanks, etc)
- Each of these classes is an equivalence partition
- Program behaves in an equivalent way for each member of an **Equivalence partition**
- Divides the input domain into classes of data form which test cases can be derived
Equivalence partitioning – cont.

Input

Invalid inputs

Valid inputs

System

Outputs
Equivalence partitioning - cont.

- Guide lines for test case selection for partitions
  - At the **boundaries** of the partition
  - Close to **mid-point** of the partition

![Diagram of Equivalence Partitioning]

- **Equivalence Partition**
  - **Left Boundary Value**
  - **Mid-point Value**
  - **Right Boundary Value**
Equivalence partitioning - Example 1

- Program specs states:
  - Accepts 4 to 10 inputs
  - Each input is 5-digit integer greater than 10,000

- Partition system inputs and outputs into ‘equivalence sets’
Equivalence partitioning - cont.

Example

- Input values between 10000 and 99999:
  - Valid: 99999
  - Invalid: 10000

- Input values between 4 and 10:
  - Valid: 7
  - Invalid: 3, 4, 11

- Input values less than 10000 and more than 99999:
  - Valid: 50000
  - Invalid: 10000, 100000
Equivalence partitioning - cont.

Example 1

• If input is a 5-digit integer between 10,000 and 99,999, 
  equivalence partitions are (fig 20.5 Sommerville)
    <10,000,
    10,000 - 99,999
    and > 99,999

• Test cases - 6 cases
  00000, (invalid special value that may be checked)
  09999, (invalid left boundary value)
  10000, (valid left boundary value)
  50000, (valid mid value)
  99999, (valid right boundary value)
  100000, (invalid right boundary value)

Remember: Bugs lurk in corners & congregate at boundaries
Structural testing ‘White-box testing’

- Glass-box, Clear-box, Transparent-box
- For small program units
- Needs source code

- Objective: is to exercise all program statements
- (not all path combinations)

- Fig 20.8
White-box testing

- Test data
  - Tests
  - Derives
  - Component code
    - known
  - Test outputs
    - known
Path testing

• Each path through the program is executed at least once

• For loops & conditions

• Used at unit testing and module testing levels
Program flow graphs (Diagraph)

- Flow Graph (Diagraph):
  - nodes representing program decisions
  - arcs (edges) representing the flow of control
  - Ignore sequential statements (assignments, procedures calls, I/O)

- Statements with conditions are therefore nodes in the flow graph

- Cyclomatic complexity =
  Number of edges - Number of nodes + 2
Cyclomatic complexity

- Cyclomatic complexity = \textbf{number of tests} to test \textbf{all control} statements

- Cyclomatic complexity equals number of conditions in a program

- Although all paths are executed, all combinations of paths are not executed
Path testing
Binary search flow graph

1. while bottom <= top
2. if (elemArray[mid] == key)
3. 8. if (elemArray[mid] < key)
4. 9. bottom > top
5. 5. (if (elemArray[mid] < key)
6. 7. (if (elemArray[mid] < key)

Independent paths

- **Cyclomatic complexity** = $11 - 9 + 2 = 4$

- Thus 4 independent paths:
  - 1, 2, 3, 8, 9
  - 1, 2, 3, 4, 6, 7, 2
  - 1, 2, 3, 4, 5, 7, 2
  - 1, 2, 3, 4, 6, 7, 2, 8, 9

- Test cases should be derived so that all of these paths are executed
Integration testing (black-box)

- Tests complete systems or subsystems composed of integrated components

- Integration testing should be black-box testing with tests derived from the specification

- Main difficulty is localising errors
- Incremental integration testing reduces this problem - (XP)
Incremental integration testing

Recall: XP Continuous Integration of: Stories + (Acceptance & Unit Tests)

Minimal system

Test sequence 1
A
B
T1
T2
T3

Test sequence 2
A
B
C
T1
T2
T3
T4

Test sequence 3
A
B
C
D
T1
T2
T3
T4
T5

T1, T2, T3 are repeated to insure right interaction of C with A & B
Approaches to integration testing

• **Top-down integration testing**
  - Start with high-level system and integrate from the top-down replacing individual components by **stubs**
  - **Stubs** have the **same interface** as the components but **very limited functionality**

• **Bottom-up integration testing (XP)**
  - Integrate and test low-level components (or stories in XP), with **full functionality**, before developing higher level components, until the complete system is created

• In practice, combination of both
Top-down integration testing

Test Sequence 1

Level 2 stubs

Testing sequence

Level 2

Level 3 stubs

Test Sequence 2

...
Bottom-up integration testing
Testing approaches

• Architectural validation
  - Top-down integration testing is better at discovering errors in the system architecture

• System demonstration
  - Top-down integration testing allows a limited demonstration at an early stage in the development

• Test implementation
  - Often easier with bottom-up integration testing
Interface testing

• Takes place when **modules** or **sub-systems** are **integrated** to create larger systems

• Objectives are to detect faults due to interface errors or invalid assumptions about interfaces

• Particularly important for **object-oriented development** as **objects** are defined by their **interfaces**
Interface testing
Interfaces types

- **Parameter interfaces**
  - Data passed from one procedure to another

- **Shared memory interfaces**
  - Block of memory is shared between procedures

- **Message passing interfaces**
  - Sub-systems request services from other sub-systems
Interface testing guidelines

- Design tests so that parameters to a called procedure are at the extreme ends of their ranges

- Use stress testing in message passing systems
Stress testing

- Exercises the system beyond its maximum design load.
- Stressing the system often causes defects to come to light.
- Stress testing failure behaviour.
  - Systems should not fail catastrophically.
  - Stress testing checks for unacceptable loss of service or data.

- Particularly relevant to distributed systems which can exhibit severe degradation as a network becomes overloaded.
Object-oriented (OO) testing

- The components to be tested are object classes that are instantiated as objects
Object-oriented testing: Testing levels

- Testing operations (methods) associated with objects
  
  Remember: XP Unit Testing by developers on implemented operations (methods)

- Testing object classes
- Testing clusters of cooperating objects
- Testing the complete OO system
Object class testing

- Complete test coverage of a class involves:
  - Testing all operations (methods “implemented operation”) associated with an object
  - Setting and interrogating all object attributes
  - (Recall: XP Unit testing for all methods and attributes of a class)

- **Inheritance makes it more difficult** to design object class tests
  - the information to be tested is **not localised**
Automated testing

- Important XP practice (See Shodan metrics)

- Highly combined with Continuous Integration practice

- In addition to Application code
  - Write Test code to automate testing

- Disadvantage:
  - Extra Cost for the extra code to automate testing
  - “Testing code” vs Application code

- Advantages:
  - Easy – just press a button
  - Fast
  - Higher quality guaranteed

- Frameworks
  - JUnit & NUnit XUint
Key points

• Test parts of a system which are commonly used rather than those which are rarely executed
• Equivalence partitions are sets of test cases where the program should behave in an equivalent way
• Black-box testing is based on the system specification
• Structural testing identifies test cases which cause all paths through the program to be executed
Key points

• Test coverage measures ensure that all statements have been executed at least once.
• Interface defects arise because of specification misreading, misunderstanding, errors or invalid timing assumptions
• To test object classes, test all operations, attributes and states
• Integrate object-oriented systems around clusters of objects