Defect testing

- Testing programs to establish the presence of system defects
Objectives

- To understand testing techniques that are geared to discover program faults
- To introduce guidelines for interface testing
- To understand specific approaches to object-oriented testing
- To understand the principles of CASE tool support for testing
Topics covered

- Defect testing
- Integration testing
- Object-oriented testing
- Testing workbenches
The testing process

- **Component testing**
  - Testing of individual program components
  - Usually the responsibility of the component developer (except sometimes for critical systems)
  - Tests are derived from the developer’s experience

- **Integration 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
Testing phases

Component testing

Integration testing

Software developer

Independent testing team
Defect testing

- The goal of defect testing is to discover defects in programs.
- A *successful* defect test is a test which causes a program to behave in an anomalous way.
- Tests show the presence not the absence of defects.
Testing priorities

- Only exhaustive testing can show a program is free from defects. However, exhaustive testing is impossible.
- Tests should exercise a system's capabilities rather than its components.
- Testing old capabilities is more important than testing new capabilities.
- Testing typical situations is more important than boundary value cases.
Test data and test cases

- **Test data**  Inputs which have been devised to test the system
- **Test cases**  Inputs to test the system and the predicted outputs from these inputs if the system operates according to its specification
The defect testing process

1. Design test cases
2. Prepare test data
3. Run program with test data
4. Compare results to test cases
5. Test cases
6. Test data
7. Test results
8. Test reports
Black-box testing

- An approach to testing where the program is considered as a ‘black-box’
- The program test cases are based on the system specification
- Test planning can begin early in the software process
Black-box testing

Input test data

System

Outputs which reveal the presence of defects

Outputs which reveal the presence of defects

Inputs causing anomalous behaviour

Input test data

Output test results

I

O_e
Equivalence partitioning

- Input data and output results often fall into different classes where all members of a class are related.
- Each of these classes is an equivalence partition where the program behaves in an equivalent way for each class member.
- Test cases should be chosen from each partition.
Equivalence partitioning

[Diagram showing invalid and valid inputs leading to a system, which produces outputs.]
Equivalence partitioning

- Partition system inputs and outputs into ‘equivalence sets’
  - If input is a 5-digit integer between 10,000 and 99,999, equivalence partitions are <10,000, 10,000-99, 999 and > 10, 000

- Choose test cases at the boundary of these sets
  - 00000, 09999, 10000, 99999, 10001
Equivalence partitions

Between 10000 and 99999
Less than 10000
More than 99999

Between 4 and 10
Less than 4
More than 10

Number of input values

Input values

©Ian Sommerville 2000
Software Engineering, 6th edition. Chapter 20
Search routine specification

**procedure** Search (Key : ELEM ; T: ELEM_ARRAY; Found : *in out* BOOLEAN; L: *in out* ELEM_INDEX) ;

**Pre-condition**
-- the array has at least one element
T'FIRST <= T'LAST

**Post-condition**
-- the element is found and is referenced by L
( Found and T (L) = Key)

or
-- the element is not in the array
( not Found and
not (exists i, T'FIRST >= i <= T’LAST, T (i) = Key ))
Search routine - input partitions

- Inputs which conform to the pre-conditions
- Inputs where a pre-condition does not hold
- Inputs where the key element is a member of the array
- Inputs where the key element is not a member of the array
Testing guidelines (sequences)

- Test software with sequences which have only a single value
- Use sequences of different sizes in different tests
- Derive tests so that the first, middle and last elements of the sequence are accessed
- Test with sequences of zero length
# Search routine - input partitions

<table>
<thead>
<tr>
<th>Array</th>
<th>Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single value</td>
<td>In sequence</td>
</tr>
<tr>
<td>Single value</td>
<td>Not in sequence</td>
</tr>
<tr>
<td>More than 1 value</td>
<td>First element in sequence</td>
</tr>
<tr>
<td>More than 1 value</td>
<td>Last element in sequence</td>
</tr>
<tr>
<td>More than 1 value</td>
<td>Middle element in sequence</td>
</tr>
<tr>
<td>More than 1 value</td>
<td>Not in sequence</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input sequence (T)</th>
<th>Key (Key)</th>
<th>Output (Found, L)</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>17</td>
<td>true, 1</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td>false, ??</td>
</tr>
<tr>
<td>17, 29, 21, 23</td>
<td>17</td>
<td>false, ??</td>
</tr>
<tr>
<td>41, 18, 9, 31, 30, 16, 45</td>
<td>45</td>
<td>true, 7</td>
</tr>
<tr>
<td>17, 18, 21, 23, 29, 41, 38</td>
<td>23</td>
<td>true, 4</td>
</tr>
<tr>
<td>21, 23, 29, 33, 38</td>
<td>25</td>
<td>false, ??</td>
</tr>
</tbody>
</table>
Structural testing

- Sometime called white-box testing
- Derivation of test cases according to program structure. Knowledge of the program is used to identify additional test cases
- Objective is to exercise all program statements (not all path combinations)
White-box testing

- Test data
- Component code
- Test outputs

Tests Derives
class BinSearch {

    // This is an encapsulation of a binary search function that takes an array of
    // ordered objects and a key and returns an object with 2 attributes namely
    // index - the value of the array index
    // found - a boolean indicating whether or not the key is in the array
    // An object is returned because it is not possible in Java to pass basic types by
    // reference to a function and so return two values
    // the key is -1 if the element is not found

    public static void search ( int key, int [] elemArray, Result r ) {
        int bottom = 0 ;
        int top = elemArray.length - 1 ;
        int mid ;
        r.found = false ; r.index = -1 ;
        while ( bottom <= top ) {
            mid = (top + bottom) / 2 ;
            if ( elemArray [mid] == key ) {
                r.index = mid ;
                r.found = true ;
                return ;
            } // if part
            else {
                if ( elemArray [mid] < key )
                    bottom = mid + 1 ;
                else
                    top = mid - 1 ;
            } //else
        } // while loop
    } // search
} //BinSearch
Binary search - equiv. partitions

- Pre-conditions satisfied, key element in array
- Pre-conditions satisfied, key element not in array
- Pre-conditions unsatisfied, key element in array
- Pre-conditions unsatisfied, key element not in array
- Input array has a single value
- Input array has an even number of values
- Input array has an odd number of values
Binary search equiv. partitions

Equivalence class boundaries

Elements < Mid

Elements > Mid

Mid-point
## Binary search - test cases

<table>
<thead>
<tr>
<th>Input array (T)</th>
<th>Key (Key)</th>
<th>Output (Found, L)</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>17</td>
<td>true, 1</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td>false, ??</td>
</tr>
<tr>
<td>17, 21, 23, 29</td>
<td>17</td>
<td>true, 1</td>
</tr>
<tr>
<td>9, 16, 18, 30, 31, 41, 45</td>
<td>45</td>
<td>true, 7</td>
</tr>
<tr>
<td>17, 18, 21, 23, 29, 38, 41</td>
<td>23</td>
<td>true, 4</td>
</tr>
<tr>
<td>17, 18, 21, 23, 29, 33, 38</td>
<td>21</td>
<td>true, 3</td>
</tr>
<tr>
<td>12, 18, 21, 23, 32</td>
<td>23</td>
<td>true, 4</td>
</tr>
<tr>
<td>21, 23, 29, 33, 38</td>
<td>25</td>
<td>false, ??</td>
</tr>
</tbody>
</table>
Path testing

- The objective of path testing is to ensure that the set of test cases is such that each path through the program is executed at least once.
- The starting point for path testing is a program flow graph that shows nodes representing program decisions and arcs representing the flow of control.
- Statements with conditions are therefore nodes in the flow graph.
Program flow graphs

- Describes the program control flow. Each branch is shown as a separate path and loops are shown by arrows looping back to the loop condition node.
- Used as a basis for computing the cyclomatic complexity.
- Cyclomatic complexity = Number of edges - Number of nodes + 2
Cyclomatic complexity

- The number of tests to test all control statements equals the cyclomatic complexity
- Cyclomatic complexity equals number of conditions in a program
- Useful if used with care. Does not imply adequacy of testing.
- Although all paths are executed, all combinations of paths are not executed
while bottom <= top

if (elemArray [mid] == key

(if (elemArray [mid]< key

bottom > top

Binary search flow graph
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
- A dynamic program analyser may be used to check that paths have been executed
Integration testing

- 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
Incremental integration testing

Test sequence 1

Test sequence 2

Test sequence 3
Approaches to integration testing

- **Top-down testing**
  - Start with high-level system and integrate from the top-down replacing individual components by stubs where appropriate

- **Bottom-up testing**
  - Integrate individual components in levels until the complete system is created

- **In practice, most integration involves a combination of these strategies**
Top-down testing

Level 1

Level 2

Level 2

Level 2

Level 3 stubs

Level 2 stubs

Testing sequence

Level 1
Bottom-up testing

Test drivers

Level N

Level N

Level N

Level N

Level N

Testing sequence

Test drivers

Level N–1

Level N–1

Level N–1

Level N–1
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

- Test observation
  - Problems with both approaches. Extra code may be required to observe tests
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

- **Procedural interfaces**
  - Sub-system encapsulates a set of procedures to be called by other sub-systems

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

- **Interface misuse**
  - A calling component calls another component and makes an error in its use of its interface e.g. parameters in the wrong order

- **Interface misunderstanding**
  - A calling component embeds assumptions about the behaviour of the called component which are incorrect

- **Timing errors**
  - The called and the calling component operate at different speeds and out-of-date information is accessed
Interface testing guidelines

- Design tests so that parameters to a called procedure are at the extreme ends of their ranges
- Always test pointer parameters with null pointers
- Design tests which cause the component to fail
- Use stress testing in message passing systems
- In shared memory systems, vary the order in which components are activated
Stress testing

- Exercises the system beyond its maximum design load. Stressing the system often causes defects to come to light.
- Stressing the system test 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 testing

- The components to be tested are object classes that are instantiated as objects.
- Larger grain than individual functions so approaches to white-box testing have to be extended.
- No obvious ‘top’ to the system for top-down integration and testing.
Testing levels

- Testing operations associated with objects
- 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 associated with an object
  - Setting and interrogating all object attributes
  - Exercising the object in all possible states

- Inheritance makes it more difficult to design object class tests as the information to be tested is not localised
Weather station object interface

<table>
<thead>
<tr>
<th>WeatherStation</th>
</tr>
</thead>
<tbody>
<tr>
<td>identifier</td>
</tr>
</tbody>
</table>

- Test cases are needed for all operations
- Use a state model to identify state transitions for testing
- Examples of testing sequences
  - Shutdown ♦ Waiting ♦ Shutdown
  - Waiting ♦ Calibrating ♦ Testing ♦ Transmitting ♦ Waiting
  - Waiting ♦ Collecting ♦ Waiting ♦ Summarising ♦ Transmitting ♦ Waiting

| identifier reportWeather () calibrate (instruments) test () startup (instruments) shutdown (instruments) |
Object integration

- Levels of integration are less distinct in object-oriented systems
- Cluster testing is concerned with integrating and testing clusters of cooperating objects
- Identify clusters using knowledge of the operation of objects and the system features that are implemented by these clusters
Approaches to cluster testing

- **Use-case or scenario testing**
  - Testing is based on user interactions with the system
  - Has the advantage that it tests system features as experienced by users

- **Thread testing**
  - Tests the system’s response to events as processing threads through the system

- **Object interaction testing**
  - Tests sequences of object interactions that stop when an object operation does not call on services from another object
Scenario-based testing

- Identify scenarios from use-cases and supplement these with interaction diagrams that show the objects involved in the scenario.
- Consider the scenario in the weather station system where a report is generated.
Collect weather data

CommsController
request (report)
acknowledge ()
report ()
summarise ()
reply (report)

WeatherStation

WeatherData

send (report)
acknowledge ()

Weather station testing

- Thread of methods executed
  - CommsController:request
  - WeatherStation:report
  - WeatherData:summarise

- Inputs and outputs
  - Input of report request with associated acknowledge and a final output of a report
  - Can be tested by creating raw data and ensuring that it is summarised properly
  - Use the same raw data to test the WeatherData object
Testing workbenches

- Testing is an expensive process phase. Testing workbenches provide a range of tools to reduce the time required and total testing costs.
- Most testing workbenches are open systems because testing needs are organisation-specific.
- Difficult to integrate with closed design and analysis workbenches.
A testing workbench

- Source code
- Dynamic analyser
- Program being tested
- Simulator
- Test manager
- Test data
- Test results
- Test predictions
- File comparator
- Report generator
- Test results report
- Specification
- Test data generator
- Oracle
- Report generator

- Execution report
Tetsing workbench adaptation

- Scripts may be developed for user interface simulators and patterns for test data generators
- Test outputs may have to be prepared manually for comparison
- Special-purpose file comparators may be developed
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.