Validation and Verification

Testing – V&V

Books: Jacobson, Pfleeger, ...

Definition

- **Validation:**
  - Are we building the correct system?
  - Does it implement the required functionality?
  - Is it acceptable to the customer/user?

- **Verification:**
  - Have we built the system correctly?
    - Did we chose the correct algorithm and data structures
    - Did we implement them in a correct way / provable?
  - Is it reliable and robust ?
  - Is it “secure” ?
Some more Definitions

Failure:
- occurs when a program “misbehaves”
- is a property of the system in execution

Fault:
- exists in the program code
- wrong code can be corrected (fixed)

Error:
- human mistake that results in software containing a fault

Testing

The purpose of testing is finding faults!
It is NOT the validation of a program
It supports the verification of a program

Two different goals:
- acceptance tests
  they are intended to demonstrate the functionality of all system components together
- unit tests
  they are intended to assist in the verification of the code of one specific unit
Examine the code by reading it and compare it with the design documents
Try to spot algorithm, data, and logical faults

- Examining the code:
  - the code reflects the interpretation of the design documented in word and diagrams (UML)
  - this “code review” should be performed by a team of experts who are not the implementers
  - “code reviews” come in two forms
    - code inspection
    - code walkthrough
  - Code reviews have been shown to be extraordinarily successful at detecting faults.

- “code reviews” come in two forms
- code walkthrough
  - the developer presents the code and the documentation to a team of reviewers
  - the review team inspects and discusses the material, the developer leads the discussion
  - the atmosphere is informal, the attention is on the code, not the coder
- code inspection
  - similar to walkthrough but more formal
  - review team checks the code and documentation against a prepared list of criteria
  - definitions and use of classes, data types, procedures, or methods are compared with the implementations
  - team will meet as a group and may delegate tasks to individual members
Testing Levels

شدد System testing
- concerns the entire system or application and takes the end-user view of the system (acceptance testing); starts from use-cases

شدد Integration testing
- intention is to verify that the individual components or units are working together correctly
- subsystems, packages, collaborations, blocks and the entire system are tested this way

شدد Unit testing
- one and only one unit is tested as such
- typically a class, block, service module is tested in this manner

Test Cases

شدد A test case (test point) is a particular set of input data to produce an expected output
شدد It is used to demonstrate the behavior of a component or system
شدد The output is observed and compared to the correct (?) and expected result
شدد A test is a finite collection of test cases

شدد Test cases for acceptance tests are derived from the use cases
شدد Test cases for unit tests are derived from the design - and this also applies to the test cases for the integration tests.
Testing Technique

**Regression test**
- is done when changes have been made to the system;
- the purpose is to verify that the old functionality remains, and that
- the corrected or modified code performs as expected;
- very important test but time consuming and tedious, hence
- should be automated.

Testing Techniques

**Operation test**
- the system is tested in normal operation for a longer period of time, normal operation only; might include reconfiguration

**Full-scale test**
- system is run at its maximum scale; all system parameters approach the limit
- can be stretched to Stress testing where the extreme limits are reached

**Performance test (capacity test)**
- establish performance limits; measure the processing ability;
Testing Techniques

- **Overload test**
  - see how the system behaves when it is subjected to overload (“graceful degradation”); how does it survive load peaks; - security holes?

- **Negative test**
  - stress the system beyond what it has been built for – use it in an incorrect manner; stress the system beyond normal use cases; / denial-of-service

- **Ergonomic tests**
  - test the man-machine interface, is the interface consistent in the use cases; are menus logical and readable; are the error messages intelligible;

Testing Strategies

- **The most common way is testing in the reverse order to that used in design and implementation**

Testing should be as interleaved as possible with other development efforts; design, implementation and testing are very interleaved activities performed in an incremental fashion.
Testing Strategies

3 common strategies:
- top-down approach
- bottom-up approach
- sandwich approach

Testing Environment

input

stubs

test harness

test object

output

expected output
Test Planning

1. Establishing test objectives
2. Designing test cases
3. Writing test cases
4. Testing test cases
5. Executing tests
6. Evaluating test results

The Testing Process
The Testing Process

Test Planning

- planning begins early / with UseCases
- at the latest it starts with the development
- acceptance tests can be designed right from the Use Cases
- testing guidelines are established early
- incremental testing is possible when new functionality has been added - regressions
- a test log should be kept during the entire process

Test Identification

- identify what should be tested
- estimate required resources
- define guiding principles for the specification and execution of the tests

Test Specification

- describe the test and its purpose on a functional level
- describe how the test case is to be executed, including a complete procedural description
- each test case should be documented
Unit Testing

- This is the lowest level of testing
- normally done by the developer himself
- unit tests are performed for
  - classes / methods
  - blocks, components
  - packages, subsystem
- the larger the units the more formal the testing
- in traditional systems a unit test is often the test of a procedure or subroutine
- In object-oriented software it concerns classes
  - tests are carried out a higher level,
  - and are more complex

Testing of Object-Oriented Software

- a unit is not only a set of routines, but also contains encapsulated state
- the state may affect the behavior and the correctness of the unit
- the state may be changed by the execution of a routine of the unit
- additional complexity is added by
  - inheritance
  - polymorphism

Testing of object-oriented software may require support by powerful debugging tools.
Equivalence Partitioning

Testing "typical" sets of inputs
- Equivalence sets partition ranges of inputs and initial conditions that are expected to produce the "same" set of results;
- Equivalence partitioning relates to the commonality and variances among the different situations in which a system is expected to work;
- If situations are equivalent or essentially similar to one another, it is adequate to test only one of them, not all;
- although equivalence is usually intuitively obvious, it is necessary to be careful about what is assumed to be equivalent;
- Example: a password file allows 8 to 16 characters, more or less are invalid; - 3 equivalence sets.

Unit Testing

Two different views of the test object:
- Specification testing - "black box" approach
  - only the external interfaces are visible
  - tests are based on externally visible behavior
- Structural testing - "white box" approach
  - internal design and structure of the module is visible
  - perform code reviews
    - walkthrough
    - inspection
  - Test cases can be designed to execute all possible paths and branches (coverage)
Black-Box Testing

- Purpose is to verify the input-output relations of a unit
- Specification testing:
  - goal is to verify the specified behavior in the unit’s interface
  - verifies WHAT the unit does, not HOW it does it
  - send stimuli with different parameters to the unit
  - observe the responses (results returned) and possible changes of state
  - Build test cases for the different “equivalence sets”

Black-Box Testing - 2

- "equivalence sets" are concerned with the parameter domain (range of values or refs)
  - equivalence partitioning reduces the number of test cases
  - objective is to select a reasonably small number of representative test cases
    (examples: loop testing where we need test cases for (1) startup (2) mid-loop execution, and (3) termination
    - determined by a set of conditions for which an object is supposed to exhibit similar behavior
    - build one test case for each equivalence class
      (example stack: (1) empty (2) loaded (3) full – failures usually show up at the boundaries).
Black-Box Testing - 3

State-based testing:

- tests the interaction between the operations
- monitors changes that happen to the object's attributes
- especially important for state-controlled objects described by state transition diagrams (statecharts)

<table>
<thead>
<tr>
<th>stimulus</th>
<th>state</th>
<th>s0</th>
<th>s1</th>
<th>s2</th>
<th>s3</th>
<th>s4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stimulus_1</td>
<td>ok</td>
<td>ok</td>
<td>wrong response</td>
<td>ok</td>
<td>fail</td>
<td></td>
</tr>
<tr>
<td>Stimulus_2</td>
<td>fail</td>
<td></td>
<td>ok</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stimulus_3</td>
<td>ok</td>
<td></td>
<td>slow check it</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stimulus_4</td>
<td>fail</td>
<td></td>
<td>ok</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

White-Box Tests 11/23

Purpose is "structural testing":

- Not only the externally visible specifications are visible but the complete implementation (code and internal structure);
- Verify that the internal structure is correct;
- It is desirable to cover all possible combinations of
  - parameters
  - variable values
  - execution paths
- The effectiveness is frequently given as "test coverage"
- The minimum is to execute each D-D (decision-to-decision) path at least once.
Exhaustive Tests

Execute all possible combinations.
Simple example: test a floating point (FP) unit:

\[ R = \{64\text{-bit FPs}\}; \quad x, y \in R; \]

Speed of the FP unit: 1 MFLOP/sec = 10^6 FLOP/sec

Time required for an exhaustive test:

\[ 2^{64} \times 2^{64} / \text{speed} \approx 3 \times 10^{38} / 10^6 \text{ [sec]} \]

\[ \text{or roughly } 8 \times 10^{24} \text{ [years]} \]

White-Box Tests

Test coverage:

- the absolute minimum is to execute each statement at least once;
- more ambitious goal: exercise all pairs of DD paths at least once; the number of test cases increases very rapidly with the number of decisions/branches;
- “complete path testing” (coverage) is only possible in limited local scopes;
### Examples

```cpp
boolean intset::member(int t)
{
    int I = 0;
    int u = cursize - 1;
    //binary search
    while (I <= u) {
        int m = (I+u)/2;
        if (t<x[m])
            u = m+1;
        else if (t>x[m])
            I = m+1;
        else return true;
    }
    return false;
}
```

### Example White Box Test

```cpp
class WhiteBox
{
    public WhiteBox (int height, int width, boolean flag)
    {
        value = new Table_element[MAX][MAX];
        for (int i = 0; i < height; i++)
            for (int j = 0; j < width; j++)
                value[i][j] = new Table_element(i,j,this);
        if (flag == true)
            value[i][j].setFlag();
    }

    // end constructor
    .
    .
    .
}
```

- **Boundary values for height = 0,1, MAX - 1, MAX;**
- **Same boundary values for width as for height**
- **Flag values: true, false**
Example: Test Coverage

```java
public void checkoutResource(Patron patron, Resource resource)
{
    if(patron.getTotalResourcesChecked() <= 15)
    { // patron may check out at most 15 items
        if(patron.getStatus() != HAS_OVER_DUE)
        { if(resource.getStatus() == AVAILABLE)
            { resource.checkout(patron); }
        } else
        { new ErrorDialog("Resource is NOT available"); }
    } else
    { new ErrorDialog("Patron has Overdue Resources out"); }
    else
    { new ErrorDialog("Patron already has 15 items checked"); }
}
```
Example: Including Inheritance

```java
public void checkoutResource(Patron patron, Resource resource) {
    
    if (patron.getTotalResourcesChecked() <= 15) { // patron may check out at most 15 items
        if (patron.getStatus() != HAS_OVER_DUE) {
            if (resource.getStatus() == AVAILABLE) {
                resource.checkout(patron);
            } else {
                new ErrorDialog("Resource is NOT available");
            }
        } else {
            new ErrorDialog("Patron has Overdue Resources out");
        }
    } else {
        new ErrorDialog("Patron already has 15 items checked");
    }
}
```

Possible Patron Subclasses
- Faculty
- Student
- Library staff
- Local resident

Possible Resource Subclasses
- Book
- Software
- Reserve material
- Music CD
- Reference resource
- Video
- Online research resources

Integration Testing

4 common strategies:

- bottom-up approach
- top-down approach
- "big bang" approach
- sandwich approach
Example Component Hierarchy

Example Bottom Up Integration
Example: Top Down Integration

Modified Top-Down Integration
Final Remarks

Reassuring the quality of a software product is a big problem;
Code walk-throughs and formal reviews assist in verifying the code (and design!);
Testing according to a systematic test plan is important;
Define equivalence sets and test them thoroughly;
Automate regression testing and keep a test log;
Validate the expected results;
Verify the unexpected behavior;
Run final acceptance tests for various load situations.