Black-Box Testing

Focus: I/O behavior. If for any given input, we can predict the output, then the module passes the test.
- Almost always impossible to generate all possible inputs ("test cases")

Goal: Reduce number of test cases by equivalence partitioning:
- Divide input conditions into equivalence classes
- Choose test cases for each equivalence class.
  (Example: If an object is supposed to accept a negative number, testing one negative number is enough)

Black-Box Testing (Continued)

Selection of equivalence classes (No rules, only guidelines):
- Input is valid across range of values. Select test cases from 3 equivalence classes:
  - Below the range
  - Within the range
  - Above the range
- Input is valid if it is from a discrete set. Select test cases from 2 equivalence classes:
  - Valid discrete value
  - Invalid discrete value

Another solution to select only a limited amount of test cases:
- Get knowledge about the inner workings of the unit being tested => white-box testing

White-Box Testing

Focus: Thoroughness (Coverage). Every statement in the component is executed at least once.

Four types of white-box testing:
- Statement Testing
- Loop Testing
- Path Testing
- Branch Testing

White-Box Testing (Continued)

Statement Testing (Algebraic Testing):
- Test single statements (Choice of operators in polynomials, etc)

Loop Testing:
- Cause execution of the loop to be skipped completely.
- Loop to be executed exactly once
- Loop to be executed more than once

Path testing:
- Make sure all paths in the program are executed

Branch Testing (Conditional Testing):
- Make sure that each possible outcome from a condition is tested at least once.

Comparison of White & Black-Box Testing

White-box Testing:
- Potentially infinite number of paths have to be tested
- White-box testing often tests what has been implemented, instead of what should be implemented.
- Cannot detect missing use cases

Black-box Testing:
- Potential combinatorial explosion of test cases (valid & invalid data)
- Often not clear whether the selected test cases uncover a particular error

Comparison of White & Black-Box Testing

Both types of testing are needed.
White-box testing and black box testing are the extreme ends of a testing continuum.
Any choice of test case lies in between and depends on the following:
- Number of possible logical paths
- Nature of input data
- Amount of computation
- Complexity of algorithms and data structures
The Four Testing Steps

1. Select what has to be measured
   - Completeness of requirements
   - Code tested for reliability
   - Design tested for cohesion

2. Decide how the testing is done
   - Code inspection
   - Proofs
   - Black-box, white-box,
   - Select integration testing strategy (big bang, bottom up, top down, sandwich)

3. Develop test cases
   - A test case is a set of test data or situations that will be used to exercise the unit (code, module, system) being tested or about the attribute being measured.

4. Create the test oracle
   - An oracle contains the predicted results for a set of test cases.
   - The test oracle has to be written down before the actual testing takes place.

Guidance for Test Case Selection

Use analysis knowledge about functional requirements (black-box):
   - Use cases
   - Expected input data
   - Invalid input data

Use design knowledge about system structure, algorithms, data structures (white-box):
   - Control structures
     - Test branches, loops, ...
   - Data structures
     - Test records fields, arrays, ...

Use implementation knowledge about algorithms:
   - Force division by zero
   - Use sequence of test cases for interrupt handler

Unit-testing Heuristics

1. Create unit tests as soon as object design is completed:
   - Black-box test: Test the use cases & functional model
   - White-box test: Test the dynamic model
   - Data-structure test: Test the object model

2. Develop the test cases
   - Goal: Find the minimal number of test cases to cover as many paths as possible

3. Cross-check the test cases to eliminate duplicates
   - Don’t waste your time!

4. Desk check your source code
   - Reduces testing time

5. Create a test harness
   - Test drivers and test stubs are needed for integration testing

6. Describe the test oracle
   - Often the result of the first successfully executed test

7. Execute the test cases
   - Don’t forget regression testing
   - Re-execute test cases every time a change is made.

8. Compare the results of the test with the test oracle

Component-Based Testing Strategy

The entire system is viewed as a collection of subsystems (sets of classes) determined during the system and object design.

The order in which the subsystems are selected for testing and integration determines the testing strategy:
   - Big bang integration (nonincremental)
   - Bottom up integration
   - Top down integration
   - Sandwich testing
   - Variations of the above

For the selection use the system decomposition from the System Design.
Bottom-up Testing Strategy
The subsystems in the lowest layer of the call hierarchy are tested individually. Then the next subsystems are tested that call the previously tested subsystems. This is done repeatedly until all subsystems are included in the testing. Special program needed to do the testing, Test Driver:
• A routine that calls a particular subsystem and passes a test case to it

Pros and Cons of Bottom-Up Integration Testing
Bad for functionally decomposed systems:
• Tests the most important subsystem last
Useful for integrating the following systems:
• Object-oriented systems
• real-time systems
• systems with strict performance requirements

Top-down Testing Strategy
Test the top layer or the controlling subsystem first. Then combine all the subsystems that are called by the tested subsystems and test the resulting collection of subsystems. Do this until all subsystems are incorporated into the test. Special program is needed to do the testing, Test stub:
• A program or a method that simulates the activity of a missing subsystem by answering to the calling sequence of the calling subsystem and returning back fake data.

Pros and Cons of Top-Down Integration Testing
Test cases can be defined in terms of the functionality of the system (functional requirements).
Writing stubs can be difficult: Stubs must allow all possible conditions to be tested.
Possibly a very large number of stubs may be required, especially if the lowest level of the system contains many methods.
One solution to avoid too many stubs: Modified top-down testing strategy
• Test each layer of the system decomposition individually before merging the layers.
• Disadvantage of modified top-down testing: Both, stubs and drivers are needed.

Sandwich Testing Strategy
Combines top-down strategy with bottom-up strategy The system is viewed as having three layers:
• A target layer in the middle
• A layer above the target
• A layer below the target
• Testing converges at the target layer
How do you select the target layer if there are more than 3 layers?
• Heuristic: Try to minimize the number of stubs and drivers.

Pros and Cons of Sandwich Testing
Top and bottom layer tests can be done in parallel. Does not test the individual subsystems thoroughly before integration.
Solution: Modified sandwich testing strategy
Modified Sandwich Testing Strategy

**Test in parallel:**
- Middle layer with drivers and stubs
- Top layer with stubs
- Bottom layer with drivers

**Test in parallel:**
- Top layer accessing middle layer (top layer replaces drivers)
- Bottom accessed by middle layer (bottom layer replaces stubs)

Steps in Component-Based Testing

1. Based on the integration strategy, **select a component** to be tested. Unit test all the classes in the component.
2. Put selected component together; do any preliminary fix-up necessary to make the integration test operational (drivers, stubs).
3. Do **functional testing**: Define test cases that exercise all use cases with the selected component.

Steps in Component-Based Testing

4. Do **structural testing**: Define test cases that exercise the selected component
5. Execute **performance tests**
6. **Keep records** of the test cases and testing activities.
7. **Repeat** steps 1 to 6 until the full system is tested.

Performance Testing

- **Stress Testing:**
  - Stress limits of system (maximum # of users, peak demands, extended operation)
- **Volume testing:**
  - Test what happens if large amounts of data are handled
- **Configuration testing:**
  - Test the various software and hardware configurations
- **Compatibility test:**
  - Test backward compatibility with existing systems
- **Security testing:**
  - Try to violate security requirements
- **Timing testing:**
  - Evaluate response times and time to perform a function
- **Environmental test:**
  - Test tolerances for heat, humidity, motion, portability
- **Quality testing:**
  - Test reliability, maintainability & availability of the system
- **Recovery testing:**
  - Tests system’s response to presence of errors or loss of data.
- **Human factors testing:**
  - Tests user interface with user

Test Cases for Performance Testing

- **Push the (integrated) system to its limits.**
- **Test how the system behaves when overloaded.**
- Can bottlenecks be identified? (First candidates for redesign in the next iteration)
- **Try unusual orders of execution**
- Call a receive() before send()
- **Check the system’s response to large volumes of data**
- If the system is supposed to handle 1000 items, try it with 1001 items.
- **What is the amount of time spent in different use cases?**
- Are typical cases executed in a timely fashion?

Acceptance Testing

**Goal:** Demonstrate system is ready for operational use
- Choice of tests is made by client/sponsor
- Many tests can be taken from integration testing
- Acceptance test is performed by the client, not by the developer.

Majority of all bugs in software is typically found by the client after the system is in use, not by the developers or testers. Therefore two kinds of additional tests:
Acceptance Testing

**Alpha test:**
- Sponsor uses the software at the developer’s site.
- Software used in a controlled setting, with the developer always ready to fix bugs.

**Beta test:**
- Conducted at sponsor’s site (developer is not present)
- Software gets a realistic workout in target environment
- Potential customer might get discouraged