Test Case Generation

Background

- Main objectives of a project: High Quality & High Productivity
- Quality has many dimensions
  - reliability, maintainability, interoperability etc.
- Reliability - most important
- Reliability: ???
- More defects => more chances of failure =>
- Hence quality goal: **Have as few defects as possible in the delivered software!**

Faults & Failure

- **Failure:** - behavior of the sw is different from expected/specified.
- **Fault:** cause of software failure
- Fault = bug = defect
- Failure implies presence of defects
- Defect has the potential to cause failure.
- Defect is environment, project specific

Role of Testing

- Identify remaining defects after review
- Reviews cannot catch all defects
- There will be requirement defects, design defects and coding defects in code
- **Testing:**
  - Detects defects
  - plays a critical role in ensuring quality.

Detecting defects in Testing

- During testing, execute a program with a set of test cases
- Failure during testing=> defects are present
- No failure => confidence grows, **but can not say "defects are absent"**
- Defects detected through failures
- To detect defects – must cause failures

Test Oracle

- To check if a failure has occurred when executed with a test case, we need to know the correct behavior
  - We need a test oracle, which is often a human
- Human oracle makes each test case expensive as someone has to check the correctness of its output

Common Test Oracles

specifications and documentation,

other products (ex, an oracle for a software program might be a second program that uses a different algorithm to evaluate the same mathematical expression as the product under test)

an *heuristic oracle* that provides approximate results or exact results for a set of a few test inputs,

a *statistical oracle* that uses statistical characteristics,

a *consistency oracle* that compares the results of one test execution to another for similarity,

a *model-based oracle* that uses the same model to generate and verify system behavior,

or a human being's judgment (i.e. does the program "seem" to the user to do the correct thing?).

Role of Test cases

- Ideally we would like the following for test cases
  - No failure implies “no defects” or “high quality”
  - If defects present, then some test case causes a failure
- Psychology of testing is important
  - should be to ‘reveal’ defects (not to show that it works!)
  - test cases must be “destructive”

Role of Testing

- Role of test cases is clearly very critical
  - Only if test cases are “good”, the confidence increases after testing

Test case design

- During test planning, design a set of test cases that will detect defects present
- Criteria needed to guide test case selection
- Two approaches to design test cases
  - functional or black box
  - structural or white box
- Both are complimentary

Black Box testing

- Software tested to be treated as a block box
- Specification for the black box is given
- Use the expected behavior of the system to design test cases
- Determine test cases solely from specification.
- Internal structure of code **not** used for test case design

Black box Testing…

- Premise: Expected behavior is specified.
  - So, just test for specified expected behavior
- Its implemented is not an issue.
- For modules:
  - specification produced in design specify expected behavior
- For system testing,
  - SRS specifies expected behavior

Black Box Testing…

- Most thorough functional testing - exhaustive
  - Software is designed to work for an input space
  - Test with all elements in the input space
- Infeasible - too high a cost
- Need better method for selecting test cases
- Several proposed approaches..

Equivalence Class partitioning

- Divide the input space into equivalent classes
- If the software works for a test case from a class then it is likely to work for all
- Equivalent classes can reduce the set of test cases
- Getting ideal equivalent classes is impossible
- Approximate it by identifying classes for which different behavior is specified

Equivalence class partitioning…

- Rationale: specification requires same behavior for elements in a class
- Software likely to be constructed such that it either fails for all or for none.
  - E.g. if a function was not designed for negative numbers then it should fail for all the negative numbers
- For robustness, should form equivalent classes for invalid inputs also

Equivalent class partitioning..

- Every condition specified as input is an equivalent class
- Define invalid equivalent classes also
- E.g. range 0< value<Max specified
  - one range is the valid class
  - input < 0 is an invalid class
  - input > max is an invalid class
- Whenever an entire range may not be treated uniformly - split into classes

Equivalence class…

- Once eq classes selected for each of the inputs, test cases have to be selected
  - Select each test case covering as many valid equivalence classes as possible
  - Or, have a test case that covers at most one valid class for each input
  - Plus a separate test case for each invalid class

Equivalence Testing Example

- Given the specs for a db states:
  - The system must be able to handle any number of records from 1 through 16383
- If the system can handle 34 records and 16383 records, chances are it will work for, say, 5251 records
- In fact the chances of detecting a fault, if present, are likely to be equally good for any test case in the range

Equivalence Testing

- Conversely, if the product works correctly for any one test in the range 1 through 16383, it will probably work for any other case in the range
- *An equivalence class*

Equivalence Testing

- The specified range of number of records the system must be able to handle defines three different equivalent classes:
  - Equiv class 1:
  - Equiv class 2:
  - Equiv class 3:

Equivalence Testing

- Testing the db using equivalent classes requires that a test case from each equivalent class be selected:
  - Result of Test case from equivalent class 2
  - Result of test cases from class 1 and class 3
- A successful test case (Remember what that means?)

Boundary Value Analysis

- To maximize the chances of finding such fault, a high-payoff technique is boundary value analysis:

- Hence when testing the db system above, seven test cases should be selected ..
  - Can you list them?

Equivalence Classes and Boundary Value Analysis

Test case	Records	Description
1		

Equivalence Class contd.

- The above applies to input specs but an equally powerful technique is to examine the output specs
- Ex. Suppose the minimum SS (FICA) deduction from any one paycheck permitted by the tax code is $0.00, and the maximum is $6,342.00 corresponding to $130,280.00
- What should the test cases for SS deduction include for testing this system?

- It should include input data that are expected to result in deductions of:

Combination

- Testing both input and output specs with a combination of equivalence classes and boundary analysis values is a valuable technique for generating a relatively small set of test data with a high probability of uncovering an as yet undiscovered fault

Functional Testing

- Base test data on the functionality of the module
- Each function implemented in the module is identified. For example a typical function for a computerized warehouse product:
  - “get next db record”, or
  - “determine whether quantity on hand is below the reorder point”

Functional Testing

- After determining all the functions of the module, devise test data to test each function separately. Now take this a step further:
  - If modules consists of a hierarchy of lower-level functions, connected together by control structures of structured programming, functional testing proceeds recursively. See next example:

Functional Testing

- Given a higher-level function of the form

  (higher-level function) := if (conditional expression)
  (lower-level function 1);
  else
  (lower-level function 2);

- Since (*conditional expression*), (*lower-level function 1*), and (*lower-level function 2*) have been subjected to functional testing, (higher-level function) can be tested using branch coverage … a glass-box technique

Functional Testing

- In practice, high-level functions are not constructed this simply, rather, the functions are intertwined
- To determine faults, then, functional analysis is required

Glass-Box Testing

- Select test cases on the basis of examination of the code, rather than the specs
- Examples of forms of glass-box testing:
  - Statement
  - Branch
  - Path coverage

Statement Coverage

- Simplest form of glass-box testing
- Run a series of test cases to ensure every statement is executed at least once
- Use CASE tools to keep record of the tests
- Weakness?

Branch Coverage

- An improvement over statement coverage
- Run a series of tests to ensure all branches are tested at least once
- Use tool to keep track of which branches have (not) been tested
  - Example:
    - Btool
    - General Coverage Tools (GCT)
- Structural Tests

Path Coverage

- The most powerful form of structural tests
- Test all paths
- For product with loops, the number of paths can be large $\rightarrow$ poses difficulty
- Criteria for selecting paths?

Path Coverage

- Linear Code Sequences
  - Identify the set of points L from which control flow may jump
  - Set L includes entry and exit points and branch statements
  - Linear code sequences are those which begin at an element L and end at an element L