Testing

Non-execution based testing

- Walk-throughs
- Inspections
  - Peer-to-peer
  - Development Organization-to-customer

Typically, in non-execution based testing an “interesting” piece of code is selected either by the developer or by the customer and then this piece of code is studied, commented, and suggested upon.

The developer might choose a piece of code that is particularly clever/tricky in order to get feedback. A development organization might choose a piece of code that chose how well it has fulfilled its obligation to its customer.

Execution based testing

Behavioral properties of programs:

1. Utility – the program responds correctly to <u>valid</u> input.
2. Robustness – the program recovers well from <u>invalid</u> input.
3. Reliability – a measure of the frequency and criticality of product failure.
   i. Meantime between failures
   ii. Meantime to repair
4. Performance – a product performs well if it meets its runtime constraints.

<u>Observation</u>: Execution based program testing can be very effective in showing the presence of bugs, but it is hopelessly inadequate for showing their absence (Dijkstra ’72).

Why? *Because exhaustive program testing is <u>not</u> possible.*

Example: Consider a program with 20 inputs; each input can accept 4 distinct values. In order to test this program exhaustively we have to consider 4^{20} different test cases ($\cong 10^{12}$).

Assume that we can write these test cases; also assume that each test case takes 30secs to execute. This implies that it would take more than 1,000,000 years to execute the complete test suite with the 4^{20} test cases.

This is clearly impossible even for a simple program with only 20 inputs.

In order to test our programs effectively we need to rely on appropriate test data selection in order to establish some confidence in the program.

“Testing is a process of inferring certain behavioral properties of a product based on the results of executing the product in a known environment with selected inputs” – Goodenough ’79.

The key to good testing is to select a set of inputs that lets us infer as much information as possible from the behavior of the program:

- Black box testing; equivalence testing with boundary value analysis.

Black box testing

Treat program as a black box and infer correctness from studying the program behavior on the test cases (as opposed to Glass Box Testing – structural testing).

> *Example:* Suppose a specification states that a DB had to handle 1 – 16,383 records.
>
> Typically we approach testing a specification like this by picking some record values in the interval of 1-16,383. Once we have tested the product with a couple of values from this interval, additional testing with values from this interval does not really increase our confidence in the product. This is because the interval 1-16,383 constitutes an <u>equivalence class</u>, any number in an equivalence class is as good a test case as any other number.

This gives rise to the notion of <u>equivalence testing</u>.

> *Example Cont’d*: Our specification above actually defines three equivalence classes:
>
> 1. Less than 1 record.
> 2. 1-16,383 records.
> 3. More than 16,383 records.
>
> Using equivalence testing in order to test the DB program we only need to show that the product works for 3 test cases, one from each equivalence class.

It has been shown in empirical studies that most software problems occur at specification boundaries. In order to avoid those problems we can introduce an even more powerful testing methodology called <u>equivalence testing with boundary value analysis</u>:

- Choose values for test cases on and around equivalence class boundaries (probability of finding a problem is dramatically increased).

> *Example Cont’d*: Our specification defines two equivalence class boundaries,
>
> - Lower boundary: 1
> - Upper boundary: 16,383

We want to design test cases around those boundaries:

Test Case 1: 0 (member of first equivalence class, below lower boundary)
Test Case 2: 1 (member of second equivalence class, right on lower boundary)
Test Case 3: 2 (member of second equivalence class, right above lower boundary)
Test Case 4: 788 (member of second equivalence class – optional)
Test Case 5: 16,382 (member of second equivalence class, below upper boundary)
Test Case 6: 16,383 (member of second equivalence class, on upper boundary)
Test Case 7: 16,384 (member of third equivalence class, above upper boundary)

Guidelines for Equivalence Testing with Boundary Value Analysis

- For each range (L,U), select six test cases: <L, =L, >L and <U, =U, >U.
- For each set S, select 2 test cases: $\in S$ and $\notin S$.
- For each precise value P, select two cases: P and anything else.

Functional Testing

Unfortunately, values are not everything, modern GUIs have many more features than just input values.

- Functional testing (or functional analysis)

Each item of functionality or function of the product is identified, that is, each functional requirement has to be covered at least once.

Test data are devised in order to test each functionality or function separately.

Test Case Format

Each test case should include the following:

- Short description of purpose.
- Pass/fail criteria
- Input specification
- Expected results
- Procedural steps – "how to"
- Traceability, with each test case note down which one(s) of the original requirements it covers.
- If appropriate equivalence classes and boundary value analysis.