Black-Box Testing

5 October 2016

Types of black-box testig
Equivalence class partitioning
Boundary testing
Cause-effect analysis
Exploratory testing

"Black-Box" Testing

Product is viewed as an opaque system
   (no access to internal details – this includes source)

Why black-box testing ?
   applicable to any product
   no effort for source code analysis
   applicable from simple to complex
   and in a variety of situations

Types of black-box testing [Kaner]

Or: *where do we start testing from ?*

Function testing

test each function in isolation; basic functionality

tests are credible, easy to evaluate, not very powerful

Domain testing

essence: sample equivalence classes through representatives

initially one variable at a time, then combinations

well-chosen values ⇒ powerful, informative tests

Specification-based testing

tests for every claim in the specificatin/req. list/model/manual

conformance is very significant; choose representative tests

can go deeper: find errors/omissions/ambiguities/limit cases in spec

Types of black-box testing [Kaner, cont.]

Risk-based testing

- imagine a way program could fail, test for it
- tests must be *powerful, credible, motivating*

Stress testing: several definitions

- 1) under burst of activity
- 2) at/beyond specified limits, to cause failure (IEEE std.)
- 3) to see *how* the program fails (important!)

Regression testing

- test set designed for reuse after every program change
- no longer powerful, but well documented for maintenance

User testing

- real, not simulated users (beta testing)
- using specified scenarios, or freely
- credible, motivating, not always powerful (depends on user)

Types of black-box testing [Kaner, cont.]

Scenario Testing
- specific use case; may be model-based
- credible, motivating, easy to evaluate, complex
- going deeper: use scenario in limit / hostile case

State-model-based testing
- model: finite-state automaton
- analyze model, then product with model-based tests

High-volume automated testing

Exploratory testing
- actively guides testing process
- designs new tests based on info offered by existing tests

Test Strategies [Kaner, Black-Box Testing course]

1. Start with simple (obvious) tests (grave if they fail)
2. Test each function, understand behavior before criticizing.
3. Test broadly before deeply. Cover program before focusing.
4. More powerful tests, boundary conditions
5. Expand scope, look for challenges
6. Freestyle exploratory testing

Equivalence class partitioning [Myers]

Analyze domain of values for each variable or input,
identify sets for which we assume tests behave alike
⇒ used to generate a set of "interesting" conditions for testing

Desirable: a test case should cover several relevant conditions (should reduce number of conditions to analyze by more than one)

For every condition: tests with *valid* and *invalid* values

Myers suggests using a table of the form

Condition	Valid equiv. classes	Invalid equiv. classes

How to choose equivalence classes

Depending on the variable type / domain:

For an *interval*:
- one valid case (inside), two invalid ones (on both sides)
  - will refine for boundary testing

For a fixed (speficied) number:
- one valid case, two invalid cases (larger, smaller)

For enumeration type: each value, plus an invalid one

Combining equivalence classes into test cases:
- cover as many *valid classes* with one test case
- generate a separate test for each *invalid class*
  - (if combined, an invalid condition may mask another)

Example to work through

Declaring dimensions of an array in FORTRAN [Myers]

`DIMENSION` *array-descrp* (, *array-descrp*)*

array-descrp ::= *name* (*dim* (, *dim*)*)

name ::= *letter* (*letter* | *digit*)*   (1..6 chars)

dim ::= [*lower-bound* :] *upper-bound*

bound ::= *int-constant* | *name*

-65534 $\leq$ *lower-bound* $\leq$ *upper-bound* $\leq$ 65535

lower-bound e implicit 1

Boundary testing

Refines equivalence class partitioning in two ways:

1) each limit of an equivalence class covered by a test
   implicitly: also values above / below limit

2) derive tests also from domain of *output* values, not just input
   (not just input value domain)

Working example [Burnstein]: identifiers of 3–15 alphanumeric chars, the first two being letters

Constraints (each with equivalence classes/boundary conditions):
   alphanumeric characters
   length (min - 1, min, intermediate, max, max + 1)
   first two chars

Testing using cause-effect analysis

Equivalence partitioning does not focus on combining conditions

Principle: in a combination of conditions, each factor should be covered

Steps:

decompose spec into manageable-size components
identify causes: input conditions/equivalence classes
identify effects: output conditions/change of state
express specificatin as set of rules or Boolean diagram
generate tests

Testing using cause-effect analysis

Example [Myers]

> *The character in column 1 must be an A or a B. The character in column 2 must be a digit. In this situation, the file update is made. If the first character is incorrect, message X12 is issued. If the second character is not a digit, message X13 is issued.*

Tests are generated starting from output (effect)
successively setting the causes that should produce this effect

- for an OR condition, each *true* cause individually
- for an AND condition, each *false* cause individually

similar to MC/DC coverage, but on the *specification*, not on code

Higher-level strategies: Exploratory testing

cf. James Bach:
simultaneous *learning, design* and *execution* of tests

situation-dependent

results obtained from tests determine subsequent testing

Bug finding strategies

[James Whittaker, How to Break Software]

Test perspectives:

1. User interface
   black-box: inputs, outputs
   open box: focus on state, interactions

2. System interface
   file system
   operating system (concurrency, memory, network, etc.)

What kind of tests to try ?

Invalid inputs (wrong type – e.g. objects/images/files of the wrong kind; small/large size, limit values)

is error handled ? with meaningful error messages ?

Forcing *reinitialization*: input null/invalid values.
Does the system revert to default values?

Inputs with *invalid characters* / control chars / special chars

Buffer overflow

not only when data is input, but also on future use
(limits may be different)

Combinations / interactions between inputs

two large inputs; one large and one small

Repetitive testing (loop traversal)

memory usage; (re)initialization problems

What kind of tests to try? (cont.)

Explore *one* input in *different contexts*

different answers: are all cases handled?

Generating *invalid outputs*

sometimes in a roundabout way (e.g. 29 feb. 2000 → 2001

UI attacks: refresh screen (done completely?)

Try to overstep internal limits

e.g. create table of maximum size, then add a row

Computations with invalid operators / operands

Test recursive inclusions (frame in frame; footnote in footnote, etc.)