Black Box Testing

Author: Kelsey Fargas

5 May 2015

Overview

This lecture discusses the approaches, characteristics, and concept of black box testing. Compared to white box testing, which is testing against the code itself, black box testing is testing against the specifications to see if the code implemented does what is specified. White box testing is testing functionality where black box testing focuses on the domain of the input data. There are two main approaches: identification of representative values and derivation of a model. Note that the quality of testing is highly dependent on the requirements given to the development team. If a requirement hasn't been specified or explicitly stated, then it cannot be tested.

Important Slides

A list of the important slides, and an explanation of why they are important. (At least one slide per concept explained, easy or difficult)

Slide n°3: This slide contains an example of the specifications for the given function "foo". The specification is an example of black box testing. It's a fairly easy function, but note that you want to compare the specifications to the function to see if the function does what is specified. Does the function do what is specified? (Answer: No)

```
int foo (int param){
    int result;
    result = param/2;
    return (result);
}
```

Specification for foo:

1. Inputs an integer *param*
2. Outputs an integer that is:

- 1. Half the value of *param* if *param* is even
- 2. **Full value of *param* in all other cases**

Slide n°4: This slide has the general approaches to testing which compares the details of black box and white box. Main take aways from these are:

- Black box is based on a functional specification of the software
- Basically, you're writing testing code based on specificiations (ex. SRS) and seeing if they align with each other

(Not too certain about the details, but there was definitely a question on the midterm about comparing black box vs. white box testing).

Slide n°5: This slide contains the characteristics of black box testing. Main take aways:

- Doesn't consider the implemented control structure and focuses on the domain of the data
- Human-intensive activity
- Different black-box criteria:
  - category partition method
  - state-based techniques
  - combinatorial approach
  - catalog based techniques

Slide n°6: Here is a General Approach to Black Box Testing:

1. **Equivalence partitioning of the input space**

- Condense the possible inputs into a finite set with the intent of picking 1+ test cases in each classes
  - Find groups of test cases with similar types of errors *(Slide 7)*
- We're assuming all the conditions in one partition will be treated the same by the software, so if one partition works, we assume all conditions in that partition will work and vice versa.
- Main point: we want to reduce testing time so that we can perform less tests if we expect outputs to be the same for similar functions

- Ex. Testing for input that accepts # 1-1000, write one test case instead of 1000. Divide each test case in a class (valid and invlaid inputs).

2. **Identification of boundary values *(Slide 8)***

- Identify specific values that could be handled incorrectly
- Since errors tend to occur at the boundaries, we should test them
- Complementary to EP: after identifying the classes, choose 1+ boundary values for each class
- Ex. Testing for input that accepts #1-1000, select range of 0-999 or 2-1001, which are just below and above the extreme edges of input

3. **Generation of test cases *(Slide 9)***

- Main point: you want to select test cases according to the interface.
- Suitably combining values for all inputs
- Cartesian product
- Problem: scalability
- Use domain knowledge
  - Reduce redundant tests
  - Eliminate logical contradictions
  - Test exceptional conditions once

4. **All of the above using a Systematic Approach**

- Break down the test generation process into smaller steps
  - Decoupling of different activities
  - Dividing brain-intensive form steps that can be automated
  - Monitoring of the test process

Category partition process State-based process for testing

Slide n°12: Category-Partition Method:

Generates a set of test cases from a functional specification in several steps (A-F):

- A: Identify independently testable features
- B: Identify categories
- C: Partitioning the categories into choices
- D: Identity constraints among choices
- E: Produce and evaluate test-case specifications
- F: Generate test cases from test-case specifications

Slide n°14: Step A: Identifying Independently testable features

- For each feature, identify:
  - Parameters of the feature (Ex. `<pattern> <filename>`)
  - Other elements of the execution environment on which the feature depends (Ex. *File*)
- Environment "parameters" are treated identially to input parameters

Slide n°16: Step B: Identify Categories

- Categories are elementary characteristics of the parameters
  - Ex: Cateogies for parameter *pattern*
    - Explicit: writen in the specifications
      - Enclosing quotes, blanks, quotes within the pattern
    - Implicit: Derivable from experience
      - Length of the pattern

Slide n°20: This is a good reference for an example of how to identify the categories of each parameter, whether it be input or environmental

Slide n°21: Step C: Partitioning the Categories into Choices

- Identify choices: specific classes of values for each category
- Basically, break down categories–> choices

Slide n°25: This is a good reference for an example of how to break down each cateogires of the parameters into specific classes

Example of the parameter "pattern":

- Presence of enclosing quotes
  - Not enclosed
  - Enclosed
  - Incorrect
- Presence of blanks
  - None
  - One
  - Many

Slide n°26 - 30: Step D: Identify Constraints Among Choices

- Choose a choice(from Step C) –> Find a constraint
- Combination identifies a partition of the input spaces and is a possibly meaningful test case specification
- Takeaway: Using constraints gives way to more sepcific semnatic constraints and eliminates less meaningful combos
- Example of meaningless/simply less meaningful:
  - *Empty file* and *more than one occurence of the pattern in the file*
  - Type *property*: eliminates combinations
    - [property X], [if X]
    - Permissions on the file
      - Non-readable
      - Readable [property validFile]
- Situation that we want to test at most one time:
  - Type *error* and *single*: indicate choice that don't have to appreare in more than one combo.
    - [error], [single]
    - Number of matches in a line
      - None [single]
      - One [if validFile]
      - Many [if validFile]

Slide n°31: Step E: Produce and Evaluate Test-Case Specifications

- You can estimate the number of test cases and automatically generate test-case specifications for a set of choices and constraints
- Number and type of constraints can change based on the number of test cases

Slide n°32:

- Generation can be (Semi) automated–> but it depends on how formal description of choices
  - Ex. Type of application, expertise of the testers

Slide n°34: Finite State Machines

- **Nodes**: state of the system
- **Edges**: the transitions between states
- **Edge attributes**: events and actions
- (*Recommendation: I bet he'll ask to define node, edges, and edge attributes)

Slide n°36: Finite State Machines: Approach

- Identify boundaries–> inputs–> states–> outputs–> Build table of the states (state + event->state) –> Build table of the outputs (state + event-> output)–> Design and run tests

Slide n°38: This is a good example of the bottling machine

Exam Questions

A list of questions that will probably appear in the final exam.

- What is the difference between an input parameter vs. environmental parameter?
- Define nodes, edges, and edge attributes.
-