FACULDADE ENGENHARIA
Departamento de Informática

Black Box Testing#1

(adapted from lecture notes of the CSCI 3060U - Software Quality Assurance unit, J.S. Bradbury, J.R. Cordy, 2018, and lecture notes of the LU - Software Testing unit, L. Buffoni, 2017)

Systematic Testing Methods

Last Lecture: Systematic Methods

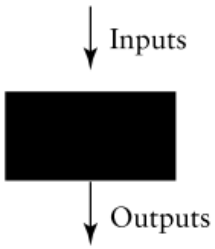
- Recall that to be a systematic test method, we must have
  - a system (rule) for creating tests
  - a measure of completeness
- Need an easy, systematic way to create test cases
  (to know for sure what to test)
- Need an easy, systematic way to run tests
  (to know how to test)
- Need an easy, systematic way to decide when we're done
  (to know when we have enough tests)

FACULDADE ENGENHARIA
Departamento de Informática

Testing in the Software Life Cycle

Last Lecture: Kinds of Tests

- We divide these tests into:



Black Box Testing Methods

- Black box methods – cannot see the software code (it may not exist yet!) – can only base their tests on the requirements or specifications

White Box Testing Methods

- White box (aka glass box) methods – can see the software's code – can base their tests on the software's actual architecture or code

Testing Methods : Black Box Testing

Overview

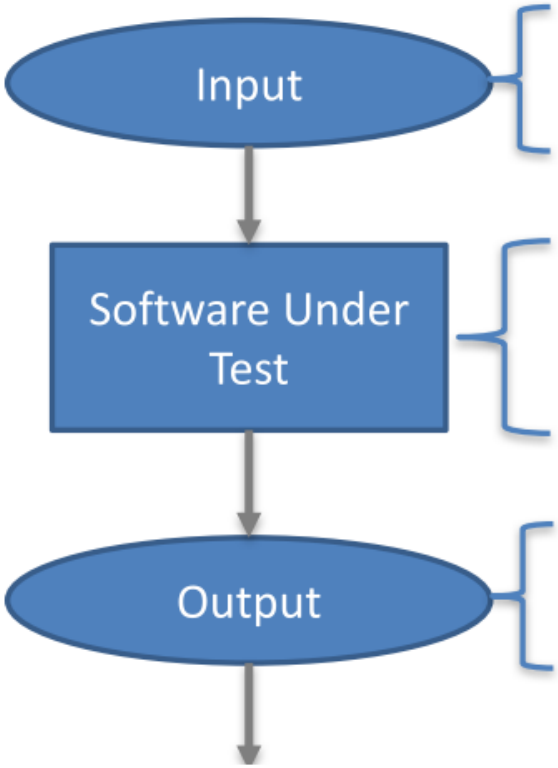
- Today we continue with learning about several kinds of black box methods:
  - Black box method 1:

Systematic functionality coverage testing

  - Black box method 2:

Systematic input coverage testing

Black Box Methods



Based on requirements or specifications, valid and invalid inputs are chosen

Black box testing can be applied on all levels of the system: Unit, Integration, System, Acceptance.

Actual outputs are compared to expected outputs

Black Box Methods

Black Box Methods

- In a black box method, we choose our test cases based solely on the requirements, specification or (sometimes) design documents
- **Advantage:** we can do it independently of the software – on a large project, black box tests can be developed in parallel with the development of the software, saving time
- Normally, black box testing is based on the functional specification (requirements) for the software system

Black Box Methods

Functional Specifications

- Functional specifications can be formal (mathematical), or more often informal (in a natural language such as Portuguese, English, …)
- In either case, the functional specification usually contains at least three kinds of information:
  1. the intended inputs
  2. the corresponding intended actions
  3. the corresponding intended outputs
- Focusing on each one of these separately gives us three different black box systems for testing

Black Box Methods

Three Kinds of Black Box Methods

- Systematic black box methods can be divided into three classes corresponding to these three kinds of information:
  1. input coverage tests, which are based on an analysis of the intended inputs, independent of their actions or outputs
  2. output coverage tests, which are based on an analysis of the intended outputs, independent of their inputs or actions
  3. functionality coverage tests, which are based on a analysis of the intended actions, with or without their inputs and outputs

Systematic Functionality Testing

An Example

- We begin with the third of these, functionality coverage testing
- Functionality coverage attempts to partition the functional specification of the software into a set of small, separate requirements

Example: Suppose that the informal requirements for a program we are to write are as follows:

"Given as input two integers x and y, output all the numbers smaller than or equal to x that are evenly divisible by y. If either x or y is zero, then output zero."

Systematic Functionality Testing

Requirements Partitioning

- Our first step is to physically partition the functional specification into separate requirements
- In this system we model the separate requirements as independent, even though they are not

Example: Suppose that the informal requirements for a program we are to write are as follows:

"Given as input two integers x and y, output all the numbers smaller than or equal to x that are evenly divisible by y. If either x or y is zero, then output zero."

Requirements Partitioning

"Given as input two integers x and y"

R1. Accept two integers as input.

"output ... the numbers"

R2. Output zero or more (integer) numbers.

"smaller than or equal to x"

R3. All numbers output must be less than or equal to the first input number.

"evenly divisible by y"

R4. All numbers output must be evenly divisible by the second number.

"all the numbers"

R5. Output must contain all numbers that meet both R3 and R4.

"If either x or y is zero, then output zero."

R6. Output must be zero (only) in the case where either first or second input integer is zero.

Test Case Selection

Test Cases for Each Requirement

- We model each partitioned requirement as independent
- We create separate test cases for each partitioned requirement

Example: For the partitioned requirement: *"If either x or y is zero, then output zero."* – R6. Output must be zero (only) in the case where either first or second input integer Is zero.

We might choose the test cases:

R6T1.	0	0	*(both zero)*
R6T2.	0	1	*(x zero, y not)*
R6T3.	1	0	*(y zero, x not)*
R6T4.	1	1	*(neither zero)*

Test Case Selection

Notice these test inputs are the simples possible and make no attempt to be exhaustive

Example: For the partitioned requirement: *"If either x or y is zero, then output zero."* – R6. Output must be zero (only) in the case where either first or second input integer Is zero.

We might choose the test cases:

R6T1.	0	0	*(both zero)*
R6T2.	0	1	*(x zero, y not)*
R6T3.	1	0	*(y zero, x not)*
R6T4.	1	1	*(neither zero)*

A Systematic Method

Black Box Functionality Coverage

- Functionality coverage gives us a system for creating functionality test cases
- It tells us when we are done (i.e., when we have test cases for every partitioned requirement)
- But notice this is *not* the same as acceptance testing – because it treats functional requirements as if they were completely separate, when in fact they are tightly related
  - So it does not replace acceptance testing, we (or the customer) must do that as well
  - Unlike acceptance testing, it is a systematic method, but like other systematic methods, it is only a partial test

Choosing Test Inputs

An Experiment

- Black box testing performs an experiment on the software system
  - We have a hypothesis that the software has certain properties, and we test the hypothesis with our test cases
  - We then observe the results and draw conclusions, in classic scientific method style

Choosing Test Inputs

Experimental Design

- A principle of experimental design is the isolation of "variables"
- This refers to the fact that we should design the experimente such that each possible cause that may affect the outcome (each experimental "variable") can be observed independently
- Thus when an effect is observed we can tell which cause is at work
- The usual way to do this is to design the experiment in steps that only vary one "variable" (possible cause) at a time, keeping everything else constant

Guidelines for Choosing Test Inputs

Choosing Inputs

- For test inputs, this principle means that we should help isolate failure causes, by as much as possible:
  - (1) Consistently choosing the simplest input values possible, in order not to introduce arbitrary variations
  - (2) Keeping everything constant between test cases, varying only one input value at a time (don't try to be "clever" introducing random input variations)
- These principles hold for all systematic test methods, not just this one

Functionality Coverage

Functionality Coverage Methods Review

- Functionality coverage partitions the functional specification into separate requirements to test
- Isolate causes by keeping test input values simple and varying one input value at a time

Input Coverage Testing

Input Coverage

- The second kind of black box testing
- **Idea:** Analyze all the possible inputs allowed by the functional specifications (requirements), create test sets based on the analysis
- Input coverage methods:
  - exhaustive, input partitioning, shotgun, (robustness) boundary
- **Objective:** Show software correctly handles all allowed inputs
- **Question:** What does *"all"* mean?

Exhaustive Testing

What does *"all"* mean?

- Ideally, input coverage testing should try every possible input to the program
- This is called exhaustive testing
- Involves testing the program with every possible input – yields a strong result: virtually certain that the program is correct
- Easy system for test cases, and obvious when done
- But usually impractical, even for very small programs

Exhaustive Testing

What does *"all"* mean?

Example:

Our gcd program specification from earlier takes any two integers as input - so we would have to test the program with an infinite number of pairs of integers

Even if we limit the input integers to 32 bits each, there are still more than 16,000,000,000,000,000,000 pairs to test

Exhaustive Testing

But sometimes …

- However, sometimes exhaustive testing ***is*** practical (and when it is, we should do it!)

Example: Y2K conversion

The Year 2000 fix automatically applied to 2-digit year comparisons used a conversion like this:

if YY1 > YY2 **then** … //fails when YY1 becomes 00

becomes:

if FourDigit (YY1) > FourDigit (YY2) **then** …

Regardless of how FourDigit is implemented, the comparison change can be exhaustively tested since there are only 100 YY1 values times 100 YY2 values, for a total of 10,000 different pairs, and every case can be *automatically* checked.

Input Partition Testing

Input Partitioning

- However, cases where exhaustive testing is practical are extremely rare
- So we must choose another way to decide when we are done testing inputs
- The most common way is to partition all the possible inputs into equivalence classes which characterize sets of inputs with something in common
- If one case in the equivalence class detects a defect all other test cases in the equivalence class are likely to detect this defect

Example: Recall our example gcd functional specification.

"Given as input two integers x and y, output all the numbers smaller than or equal to x that are evenly divisible by y. If either x or y is zero, then output zero."

Input Partition Testing

Input Partitioning

- The gcd functional specification identifies three special cases for input:
  - the case where x=0,
  - the case where y=0, and
  - the case where neither is zero
- Since the input set is to be integers, we can further partition into negative and positive cases for x and y, giving us the set of input partitions on the next slide

Input Partition Testing

Example:

Partition	x input	y input
P1	0	*non-zero*
P2	*non-zero*	0
P3	0	0
P4	*less than zero*	*less than zero*
P5	*less than zero*	*greater than zero*
P6	*greater than zero*	*less than zero*
P7	*greater than zero*	*greater than zero*

Input Partition Testing

Covering Partitions

- The partitions give us our test cases – all we must do now is design test cases to cover each partition
- For the reasons we saw last time, for each case we choose the simplest input values and vary them as little as possible

Input Partition Testing

Example:

Test	x input	y input
T1	0	1
T2	1	0
T3	0	0
T4	−1	−1
T5	−1	1
T6	1	−1
T7	1	1

- Notice that we do not take into account the intention or actions of the program, only that it handles all its input classes

Input Partition Testing

Catching Errors in Requirements

- Blindly following our input partitioning system led us to create tests for negative input – which probably was not even intended
(although we can't tell that from the requirements, which is really the point)
- Systematic creation of tests from different points of view is intended to expose problems not only in the software, but also in the specification
- As a matter of fact, *most* potential failures caught by systematic testing never actually fail when tested, since they instead are fixed by fixing the requirements or the tests before the tests are actually run!

Input Partition Testing

Advantages of Input Partition Testing

- Input partitioning is what many of us think of intuitively for testing – that is, test the response to each kind of input
- It is generally easy to identify a set of partitions given the functional specification (although it may require insight)
- It is easy to say when we are done (when we have run at least one representative test for each partition)
- It gives us confidence that the program is at least capable of handling (one example of) each different kind of input correctly

Black Box Shotgun Testing

Shotgun Testing

- Black box shotgun testing consists of choosing random values for inputs (with or without worrying about legality) over a large number of test runs
- We then verify that the outputs are correct for the legal inputs, and that program simply did not crash for ilegal ones
- More practically, we usually choose inputs from the legal set and inputs from the illegal set as separate sets of shotgun tests

Black Box Shotgun Testing

- Shotgun Testing

Example:

Test	x input	y input
T1	375	15554
T2	−76	1763
T3	273334	−9762
	... and so on …	

Black Box Shotgun Testing

Systematic?

- Black box shotgun testing is still a black box method – we don't need the code to invent appropriate inputs at random
- But it is not really systematic – although there is a system for choosing test cases (randomly), there is no completion criterion
- So to gain any confidence, we must run a very large number of test cases
- Not really useful, unless there is some way to automate verification of correct output for inputs (because don't want to verify output from thousands of runs by hand!)

Input Partition Shotgun Testing

A Hybrid Method

- Shotgun testing is nevertheless interesting, because it tries lots of different inputs
- We can use it to strengthen input partition testing by applying the shotgun method to choose random input values within each partition
  - That way we both have the confidence of input partition testing and the additional confidence that our simple input values are not the only ones that work
  - However, we still need automated output verification to be practical
  - And we must be careful about experimental design – we should run the ordinary simple input partition tests first, *then* load the shotgun

Input Robustness Testing

Robustness

- Robustness is the property that a program doesn't crash or halt unexpectedly, no matter what the input
  - Robustness testing tests for this property
- Two kinds of robustness testing:

(1) Shotgun robustness testing (random garbage input)
(2) Boundary value robustness testing

Input Boundary Testing

Boundary Values

- Even when programs behave well with input values well outside their expected range, it is typical that failures come at the boundaries of the legal or expected range of values
- For this reason, black box testers often create boundary value tests to check that the program is robust with inputs on the edge
- Unlike shotgun testing, boundary value testing is a systematic test method, because it has both
  - an easy way to choose test cases, and
  - an easy way to know when we are done (when all boundary values have been tested)

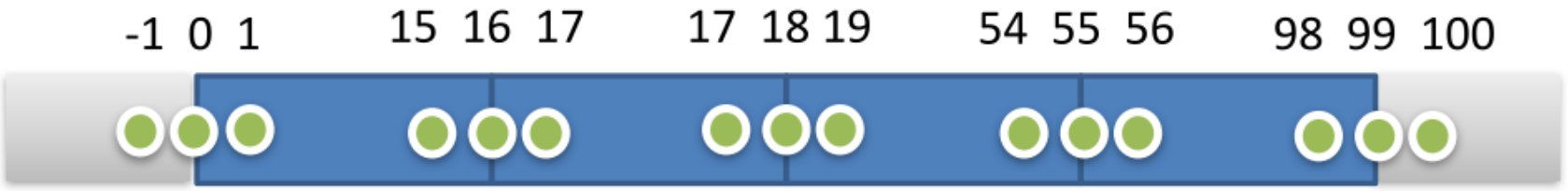
Input Boundary Testing

- Boundary value testing focuses on the boundaries between equivalence classes, simply because that is where so many defects hide. The defects can be in the requirements or in the code.

- Method
  - Identify the equivalence classes.
  - Identify the boundaries of each equivalence class.
  - Create test cases for each boundary value by choosing one point on the boundary, one point just below the boundary, and one point just above the boundary.

Input Boundary Testing - Example

Age	Employment status
0-15	Don't hire
16-17	Can hire part-time
18-54	Can hire full-time
55-99	Don't hire



Input Coverage Methods

Input Coverage Methods Review

- Exhaustive testing is usually impractical, but we can approximate it using input partitioning
- Shotgun testing can be added to input partitioning to give additional confidence
- Robustness testing checks for crashes on unexpected or unusual input, such as the boundaries of the input range

Exercise#1

- Which of the following is a form of functional testing?

a) Boundary value analysis
b) Usability testing
c) Performance testing
d) Security testing

Exercise#2

- Which one of the following options is categorized as a black-box test technique?

a) Techniques based on analysis of the architecture.

b) Techniques checking that the test object is working according to the technical design.

c) Techniques based on the expected use of the software.

d) Techniques based on formal requirements.

Exercise#3

- Use case testing is useful for which of the following?

1. Designing acceptance tests with users or customers
2. Making sure the mainstream business processes are tested
3. Finding defects in the interaction between components
4. Identifying the maximum and minimum values for every input field
5. Identifying the percentage of statements exercised by a set of tests

Select the correct answer:

a) 1, 2 and 3

b) 2, 4 and 5

c) 1, 2 and 4

d) 3, 4 and 5

Exercise#4

- Consider a program ‘Addition’ with two input values x and y and it gives the addition of x and y as an output. The range of both input values are given as:

$$100 \leq x \leq 300$$
$$200 \leq y \leq 400$$

- Create suitable test cases for the example above.

Exercise#5

- Consider a program for the determination of division of a student based on the marks in three subjects. Its input is a triple of positive integers (e.g. mark1, mark2, and mark3) and values are from interval [0, 100]. The division is calculated according to the following rules:

Average of Marks Obtained	Division
75-100	First division with distinction
60-74	First division
50-59	Second division
40-49	Third division
0-39	Fail

- Average=(mark1+mark2+mark3)/3
- Design the boundary value test cases.

Summary

- Black box methods include input coverage, output coverage, functionality coverage
  - Functionality coverage partitions the functional specification into separate requirements to test
  - Input coverage methods analyze the set of possible inputs specified and create tests to cover them


HOW
WHY
WHERE
WHEN
WHERE
WHY
WHERE
WHO
WHEN
WHAT
WHAT
WHY
WHEN
WHAT
WHO
WHEN
WHERE
WHY
HOW