Week 3

Black Box Testing - I

In this section of notes, we shall discuss the following:

- What is blackbox testing?
- What are different types of Black box Testing? - We shall talk about some major types such as Random testing, equivalence testing, category partitioning etc.

What is blackbox testing?

It is a Software Testing method that analyses the functionality of a software/application without knowing much about the internal structure/design of the item that is being tested and compares the input value with the output value.

There are many systematic black-box testing techniques. While it would be nice to have a magic algorithm for translating specifications into a set of tests, in practice, it's often not that simple; where specifications are informal, there is necessarily going to be some level of skill, experience and creativity involved in coming up with an appropriate set of tests based on those informal specs.

Figure 1: Blackbox testing

Blackbox testing is also called specification-based testing or functional testing as it uses the requirements as a standard to devise inputs and pass them onto a program to verify outputs. As there is no knowledge of the program (inside the box), whether it is developed using Java or Python, what data structure is used etc, thus, it is called black box testing. In short, view the program as a black box. Your goal is to be completely unconcerned about the internal behaviour and structure of the program. Instead, concentrate on finding circumstances in which the program does not behave according to its specifications.

We will start with the more formal techniques, and finish up with some heuristics for when those mechanical techniques are not up to the job of interpreting an informal spec.

Types (or Techniques/Strategies) for Black box Testing

There are many types of Black Box Testing, but the following are the ones we are discussing in this unit.

- Random testing
- Equivalence Testing
- Category partitioning
- Boundary Value Analysis
- Combinatorial testing
- Pairwise testing

Random Testing

Random testing involves randomly - which is not the same as arbitrarily - selecting data using a random generator to test the program with. In other words, it is done by selecting randomly generated inputs and the results of output generated are compared with the software specifications to verify if the result is correct or not.

Randomly generating input data can either be very easy, or very hard, depending on the nature of the software under test. Furthermore, even if random input data generation is easy, generating random input data that does anything interesting other than test your input sanity checking can be much more difficult.

Random testing is inexpensive to use, does not have bias and bugs are detected quickly. However, its major drawback is that it is only capable of finding basic (low level) bugs. It is only precise when the specifications are imprecise. This technique will create a problem for continuous integration if different inputs are randomly selected on each test. There are some interesting random testing tools for doing unit testing at the research stage, but none have seen wider industrial adoption.

random testing can be summarized as a four-step procedure:

- Step 1: The input domain is identified.
- Step 2: Test inputs are selected independently from the domain.
- Step 3: The system under test is executed on these inputs. The inputs constitute a random test set.
- Step 4: The results are compared to the system specification. The test is a failure if any input leads to incorrect results; otherwise it is a success.

Let us use an example to further demonstrate it.

Example 1:

Given a code in C++ below:

```
1    int myAbs(int x) {
2        if (x > 0) {
3            return x;
4        }
5        else {
6            return x; // bug: should be '-x'
7        }
8    }
```

We apply the steps given above.

- Step 1: The input domain is absolute value given in integer
- Step 2: Random inputs are {123, 36, -35, 48, 0}.
- Step 3: Each input value is passed to *myAbs* and the resultant value is noted.
- Step 4: Only the value '-35' triggers the bug triggers the bug or it fails as it does not provide the expected result.

Partitioning or Classes

Before discussing more types of blackbox testing techniques we must know what the partitioning is? Given a requirement, we will aim at devising a set of inputs, each tackling one part (or partition), of the program. Programs are usually too complex to be tested with just a single test. There are different cases in which the program is executed, and its execution often depends on various factors, such as the input you pass to the program.

Thus, partitioning is any scheme based on a partition of the input domain. Reassuringly, mathematics and common sense converge to tell us that partition testing schemes are at their most effective when the partitions have homogenous failure behaviour. **That is, either the system performs as expected on every input within the partition, or it reveals a failure on every input within the partition.**

Example 2:

Requirement: Leap year

Given a year as an input, the program should return true if the provided year is leap; false if it is not.

A year is a leap year if:

- the year is divisible by 4.
- the year is not divisible by 100.

- exceptions are years that are divisible by 400, which are also leap years.

By looking at the requirements, we can derive the following partitions:

- Year is divisible by 4, but not divisible by 100 = leap year, TRUE
- Year is divisible by 4, divisible by 100, divisible by 400 = leap year, TRUE
- Not divisible by 4 = not a leap year, FALSE
- Divisible by 4, divisible by 100, but not divisible by 400 = not leap year, FALSE

Mechanics:

To find a good set of tests, often referred to as a **test suite**, we break up (or part, or to divide) the testing of a program in classes. Moreover, we have divided the input space of the program in such a way that each class is 1) disjoint, i.e., represents a unique behaviour of the program; in other words, no two partitions represent the same behaviour, 2) can easily verify whether that behaviour is correct or not. So how a domain or problem is divided into different partitions and how test cases or test inputs are assigned to each partition.

Example 3: BMI calculator

For example, imagine that you have developed a BMI calculator. To divide the input space of the program such that each class is disjoint, we try to identify input values that will produce the same unique behaviour. In the case of BMI calculator, the categories are pretty obvious:

- Underweight = < 18.5
- Normal weight = 18.5–24.9
- Overweight = 25–29.9
- Obesity = BMI of 30 or greater

Hence, if we were to test if the program can correctly identify the “Overweight” category, we can choose any input in the range of 25-29.9. Based on the given example, there is “no two partitions that represent the same behaviour”, i.e. no input value should give out an output that is both “Normal Weight” and “Overweight” at the same time.

In a strictly mathematical sense, pretty much all software testing that is not random testing can be viewed as a type of partitioning testing.

The partitions, as you see in example, are not test cases we can directly implement. Each partition might be instantiated by an infinite number of inputs. For example, the partition year is not divisible by 4, we have an infinite number of numbers that are not divisible by 4 that we could use as concrete inputs to the program. It is just impractical to test them all. Then how do we know which concrete input to instantiate for each of the partitions?

Let us move on to different types of partitioning techniques that are applied in Black Box Testing.

Equivalence (class) partitioning

As we discussed above, each partition exercises the program in a certain way. In other words, all input values from one specific partition will make the program behave in the same way. Therefore, any input we select should give us the same result. And we assume that, if the program behaves correctly for one given input, it will work correctly for all other inputs from that partition. This idea of inputs being equivalent to each other is what we call **equivalence partitioning**.

As a practitioner, to conduct equivalence partition testing, this is what you do:

- Identify the input domain of the program, including both valid and invalid inputs.
- Based on your experience, identify parts of that input domain that you believe that the program will treat similarly - and therefore, are likely to have homogeneous failure behaviour. Each equivalence partition should be disjointed - that is, there should be no overlap!
- Write down those equivalence partitions.
- From each equivalence partition, pick a small number of tests. For now, you can assume that you can pick randomly from each equivalence partition.
- For each test input, make sure you know what the expected outputs are.
- Go test!

Given the classes we devised before (in example 2), we know we have 4 test cases in total. We can choose any input in a certain partition. We will use the following inputs for each partition:

- 2016, divisible by 4, not divisible by 100.
- 2000, divisible by 4, also divisible by 100 and by 400.
- 39, not divisible by 4.
- 1900, divisible by 4 and 100, not by 400.

Let us take another example

Example 4:

Requirements:

Temperature monitoring system. The input domain of this system consists of a single variable, temperature. The requirements specification says:

- If the patient's temperature is between 36.5 and 37.5 degrees (inclusive) display a green light.
- If the patient's temperature is below 36.5 degrees, display a blue light.
- If the patient's temperature is above 37.5 degrees, display a red light.

If you are doing black-box testing, you don't know the code, but your experience as a programmer suggests it's going to look something like this:

```
def update_temp_display(temp):
  if temp < 36.5:
    display_blue()
  elif temp > 37.5:
    display_red()
  else:
    display_green()
```

You might therefore assume that all temperature values that are supposed to cause a blue light to display are going to either all work correctly, or all fail; similarly, for red or green. Thus, you partition the input domain (temperature) as follows:

{<36.5}, {36.5, 37.5}, {>37.5}

Given the specification, there's a reasonable chance that the failure behaviour for each partition is *homogenous*; therefore, if you selected one case from each partition, that would give you a good shot of revealing any faults that exist in the program. It is not guaranteed to be the case. Consider if for some reason the second line of the program erroneously read if temp < 36. A test where temp = 36.2 would reveal a failure, where a test where temp = 35 (in the same partition) would not. But all things considered, these partitions seem reasonably likely to have the expected homogenous failure behaviour.

Question: How to find Equivalence Classes or Partitions?

As we alluded earlier that there is no magic algorithm; it is inherently a subjective process. But there are some guidelines that have proven useful over the years (and use for other partitioning techniques as well to some extent)

Numerical ranges: A good place to start is looking for numerical ranges in inputs. Wherever you have a numerical range defined in the spec, there are likely to be equivalence classes. If there's a single numerical range of valid values between x and y (for the purpose of discussion, assume the range includes x and y), this will typically mean at least three equivalence classes - *less than x, between x and y, and greater than y*. If there are multiple ranges on a single variable, there will be more classes. For instance, consider an online streaming video service that screens R-rated material to adults but blocks it for children. Such a service might have three equivalence classes for age - *less than 0 years old (invalid input), between 0 and 17 inclusive, and 18 and greater*.

Numerical ranges can be used in other contexts too. For instance, when dealing with inputs that are lists of items, the length of that input may form a numerical range.

Input Categories: If you have got inputs that have a property that falls into a small set of clearly defined categories, they can fairly obviously form a set of equivalence classes. If you've got specifically enumerated categories that's trivial, but there can be other types of input categories. A classic example in program testing is the "triangle classifier"; a program that takes three numbers representing the length of the sides of a triangle, and returns the type of triangle - scalene, isosceles, or equilateral. This fairly naturally gives you three equivalence classes:

- Side lengths that form a scalene triangle.
- Side lengths that form an equilateral triangle.
- Side lengths that form an isosceles triangle.

Note: This is different from Category Partitioning which we shall discuss next.

Invalid inputs: Depending on the thing you are testing; groups of invalid inputs are often just as valuable as groups of valid inputs in finding equivalence classes. For instance, considering the triangle classifier, there are a number of "invalid" inputs that could form equivalence classes:

- Side lengths where at least one side length is non-positive.
- Side lengths where the hypotenuse (longest side) is longer than the sum of the two shorter sides.

Looking at outputs to find inputs: Another way to develop equivalence partitions is to do so looking at the outputs and working back to inputs. For instance, consider a program to simulate a simple calculator, with a fixed-size output display buffer eight digits long. This might give you two equivalence classes for test operations:

- Operations with outputs that fit in an eight-digit display.
- Operations that don't.

Category Partitioning Testing

Often, there will be a situation where there will be multiple properties on which you can define equivalence classes, but the behaviour of the system will depend on both properties at the same time.

Key Concept:

The method gives us

- a systematic way of deriving test cases, based on the characteristics of the input parameters,
- minimize the number of tests to a feasible amount.

We first go over the steps of this method and then we illustrate the process with an example.

Steps:

- Identify the parameters, or the input of the program. For example, the parameters your classes and methods receive.
- Derive characteristics of each parameter. For example, an int year should be a positive integer number between 0 and infinite.
    - Some of these characteristics can be found directly in the specification of the program.
    - Others cannot be found from specifications. For example, an input cannot be null if the method does not handle that well.
- Add constraints, as to minimize the test suite.
    - Identify invalid combinations. For example, some characteristics might not be able to be mixed with other characteristics.
    - Exceptional behaviour does not always have to be combined with all the different values of the other inputs. For example, trying a single null input might be enough to test that corner case.
- Generate combinations of the input values. These are the test cases.

Example 5:

Requirement: Christmas discount

The system should give a 25% discount on the raw amount of the cart when it is Christmas. The method has two input parameters: the total price of the products in the cart and the date. When it is not Christmas it just returns the original price, otherwise it applies the discount.

By using the category partition method:

1. We have two input parameters:
    a. The current date
    b. The total price
2. Now for each input parameter we define the following characteristics:
    a. Based on the requirements, the only important characteristic is that the date can be either Christmas or not.
    b. The price can be a positive number, or maybe 0 for some reason. Technically the price can also be a negative number. This is an exceptional case, as you cannot really pay a negative amount.
3. The number of characteristics and parameters is not too high in this case. Still we know that the negative price is an exceptional case. Therefore, we can test that with just one combination instead of with both a date that is Christmas and not Christmas.
4. We combine the other characteristics to get the test cases. These are the following:
    a. Positive price on Christmas
    b. Positive price not on Christmas

c. Price of 0 on Christmas
d. Price of 0 not on Christmas
e. Negative price on Christmas

Now we can implement these test cases. Each of the test cases corresponds to one of the partitions that we want to test.

Let us take another example in a detailed manner to explain category partitioning.

Example 6:

Requirement: ATM Machine

The ATM machine can allow users to process savings, cheque, and credit card accounts, and can accept deposits and withdrawals from any type of account. However, the handling of each type of transaction is quite different.

Intuitively, this leads to six equivalence classes:

- Deposits to savings accounts.
- Deposits to credit card accounts.
- Deposits to cheque accounts.
- Withdrawals from savings accounts.
- Withdrawals from credit card accounts.
- Withdrawals from cheque accounts.

But then you might have another "thing" that affects the system behaviour and justifies decomposing those equivalence classes further. For example, whether the accounts are with the ATM operator's bank or with another bank. At some point, ad hoc ways of splitting these things up will become unmanageable and lead to you missing things.

Rather than going straight to "equivalence classes", you apply a multi-step process to generate them.

1. **First, from the specification, identify the things that affect the behaviour of the part of the system being tested, which become "categories".**

This includes both obvious input parameters and "environment conditions". Environment conditions are things in the environment the system is being executed in, that can affect its behaviour. In our bank ATM system, potential categories would be Account Type, and Transaction Type.

2. **The next step is to identify "choices" for each category. The choices for a category essentially correspond to an equivalence partition of that single category (aka variable).**

Again, an example is clearer: for the Account Type category we could have the choices savings, credit card, and cheque, and for the Transaction Type category we could have the choices deposit and withdrawal.

While these choices are, um, categorical (sorry for the overloading of the words) choices can be defined in any way that's appropriate for the category. Say for instance, that the system enforces that you can't withdraw more than your current balance/credit limit from an account. Then you would add another category, *Transaction Amount*, where the choices are specified as *Transaction Amount* <= *balance* and *Transaction Amount* > *balance*. As long as the choices are a) clearly defined, and b) non-overlapping, that's perfectly fine.

3. **Once you have your set of categories and choices, you list all the possible combinations. If you have n categories, each of which has c_k choices, you'll end up with c_1 * c_2 * c_k test frames.**

Here's the test frames for the three categories that are Account Type, Transaction Type, Transaction Amount in our ATM example:

- credit card, deposit, <=balance
- credit card, deposit, >balance
- savings, deposit, <=balance
- savings, deposit, >balance
- cheque, deposit, <=balance
- cheque, deposit, >balance
- credit card, withdrawal, <=balance
- credit card, withdrawal, >balance
- savings, withdrawal, <=balance
- savings, withdrawal, >balance
- cheque, withdrawal, <=balance
- cheque, withdrawal, >balance

4. **Sometimes, you can have a situation where not all categories are relevant - if so, you can exclude the irrelevant categories from the test frame - if this results in duplicate test frames, you can delete those.**

For instance, if you added another choice to the Transaction Type category, balance check, there is no "Transaction Amount" applicable. That's OK. You don't have to include an entry from a particular category if it's not applicable. Formally, Ostrand and Balcer states that test frames may have "zero or one choice" from each category. Personally, I would prefer the

convention of adding a "not applicable" choice to categories and then requiring a choice from each category - it's directly equivalent.

Sometimes the categories are hidden. Let's take another example.

Example 7:

Requirement: Chocolate Bars

A factory produces chocolates packed with a fixed amount of weight, calculated in kilos. A package should store a *X* number of kilos, which is based on the inputs from users. There are two types of chocolates available, namely small bars (1 kilo each) and big bars (5 kilos each). Based on the input given by the users, we should calculate the number of small bars to use, assuming we always use big bars before small bars. Return -1 if it cannot be done.

The input of the program is

1. the number of small bars available in the factory,
2. the number of big bars available in the factory,
3. the total amount of kilos to store in the package

The output of the program is

1. Total number of small bars needed to use
2. Return -1 if it cannot be done

For example:

Test input: small=7, big=3, total=20

Test output: 5 (We always use big bars before small bars - hence big x 3 = 15kg. 5 small bars are needed to fill in the package. We have 7 small bars available in the factory. Hence, the requirement can be met)

Explanation:

In this example, the partitions are a bit more **"hidden"**. We must really understand the problem in order to derive the partitions. You should spend some time (try to even implement it!!) in understanding it.

Now, let's think about the classes/partitions:

- **Need only small bars**. A solution that only uses the provided small bars.
- **Need only big bars.** A solution that only uses the provided big bars.
- **Need Small + big bars**. A solution that must use both small and big bars.
- **Not enough bars.** A case in which it is not possible, because there are not enough bars.
- **Not from the specs:** An exceptional case.

For each of these classes, we can devise concrete test cases:

- **Need only small bars**. small = 1, big = 1, total = 10
- **Need only big bars**. small = 5, big = 3, total = 10
- **Need Small + big bars**. small = 5, big = 3, total = 17
- **Not enough bars**. small = 4, big = 2, total = 3
- **Not from the specs**: small = 4, big = 2, total = -1

What is the total number of test cases?

These examples show why deriving good test cases is challenging. Specifications can be complex, and we need to fully understand the problem!

Summary

- Blackbox testing uses the software's functionality as a basis to test the software.
- Input is passed through the system and output is used as a means to make a judgement.
- Random testing involves randomly - which is not the same as arbitrarily - selecting data using a random generator to test the program with
- Equivalence partitioning
- In category partitioning we identify different categories and partition them into different types to extract test cases.

References

1. https://www.guru99.com/black-box-testing.html
2. Software Testing: From Theory to Practice accessible at https://sttp.site/
3. Chapter 4 of the Foundations of software testing: ISTQB certification. Graham, Dorothy, Erik Van Veenendaal, and Isabel Evans, Cengage Learning EMEA, 2008.
4. Chapter 10 of the Software Testing and Analysis: Process, Principles, and Techniques. Mauro Pezzè, Michal Young, 1st edition, Wiley, 2007.
5. Ostrand, T. J., & Balcer, M. J. (1988). The category-partition method for specifying and generating functional tests. Communications of the ACM, 31(6), 676-686.
6. Jeng, B., & Weyuker, E. J. (1994). A simplified domain-testing strategy. ACM Transactions on Software Engineering and Methodology (TOSEM), 3(3), 254-270.
7. http://burtleburtle.net/bob/math/jenny.html
8. http://www.pairwise.org/tools.asp