Test case design

Introduction

What

★★★☆ Can explain the need for deliberate test case design

Except for trivial SUTs, exhaustive testing is not practical because such testing often requires a massive/infinite number of test cases.

Consider the test cases for adding a string object to a collection:

- Add an item to an empty collection.
- Add an item when there is one item in the collection.
- Add an item when there are 2, 3, ... n items in the collection.
- Add an item that has an English, a French, a Spanish, ... word.
- Add an item that is the same as an existing item.
- Add an item immediately after adding another item.
- Add an item immediately after system startup.
- ...

Exhaustive testing of this operation can take many more test cases.

> Program testing can be used to show the presence of bugs, but never to show their absence!
>
> --Edsger Dijkstra

Every test case adds to the cost of testing. In some systems, a single test case can cost thousands of dollars e.g. on-field testing of flight-control software. Therefore, **test cases need to be designed to make the best use of testing resources.** In particular:

- **Testing should be *effective*** i.e., it finds a high percentage of existing bugs e.g., a set of test cases that finds 60 defects is more effective than a set that finds only 30 defects in the same system.
- **Testing should be *efficient*** i.e., it has a high rate of success (bugs found/test cases) a set of 20 test cases that finds 8 defects is more efficient than another set of 40 test cases that finds the same 8 defects.

For testing to be E&E, each new test you add should be targeting a potential fault that is not already targeted by existing test cases. There are test case design techniques that can help us improve the E&E of testing.

Exercises

Positive vs Negative Test Cases

★★☆☆ Can explain positive and negative test cases

A *positive test case* is when the test is designed to produce an expected/valid behavior. On the other hand, a *negative test case* is designed to produce a behavior that indicates an invalid/unexpected situation, such as an error message.

Consider the testing of the method `print(Integer i)` which prints the value of `i`.

- A positive test case: `i == new Integer(50);`
- A negative test case: `i == null;`

Black Box vs Glass Box

★★☆☆ Can explain black box and glass box test case design

Test case design can be of three types, based on how much of the SUT's internal details are considered when designing test cases:

- ***Black-box* (aka *specification-based or responsibility-based*) approach**: test cases are designed exclusively based on the SUT's specified external behavior.
- ***White-box* (aka *glass-box or structured or implementation-based*) approach**: test cases are designed based on what is known about the SUT's implementation, i.e. the code.
- ***Gray-box* approach**: test case design uses *some* important information about the implementation. For example, if the implementation of a sort operation uses different algorithms to sort lists shorter than 1000 items and lists longer than 1000 items, more meaningful test cases can then be added to verify the correctness of both algorithms.

Black-box and white-box testing

Equivalence partitions

What

★★☆☆ Can explain equivalence partitions

Consider the testing of the following operation.

> `isValidMonth(m)` : returns `true` if `m` (and `int`) is in the range [1..12]

It is inefficient and impractical to test this method for all integer values `[-MIN_INT to MAX_INT]`. Fortunately, there is no need to test all possible input values. For example, if the input value `233` fails to produce the correct result, the input `234` is likely to fail too; there is no need to test both.

In general, **most SUTs do not treat each input in a unique way. Instead, they process all possible inputs in a small number of distinct ways.** That means a range of inputs is treated the same way inside the SUT. ***Equivalence partitioning (EP)* is a test case design technique that uses the above observation to improve the E&E of testing.**

> **Equivalence partition (aka equivalence class)**: A group of test inputs that are likely to be processed by the SUT in the same way.

By dividing possible inputs into equivalence partitions you can,

- **avoid testing too many inputs from one partition.** Testing too many inputs from the same partition is unlikely to find new bugs. This increases the efficiency of testing by reducing redundant test cases.
- **ensure all partitions are tested.** Missing partitions can result in bugs going unnoticed. This increases the effectiveness of testing by increasing the chance of finding bugs.

Basic

★★☆☆ Can apply EP for pure functions

Equivalence partitions (EPs) are usually derived from the specifications of the SUT.

📦 These could be EPs for the isValidMonth example:

- [MIN_INT … 0]: **below** the range that produces `true` (produces `false`)
- [1 … 12]: the range that produces `true`
- [13 … MAX_INT]: **above** the range that produces `true` (produces `false`)

When the SUT has multiple inputs, you should identify EPs for each input.

📦 Consider the method `duplicate(String s, int n): String` which returns a `String` that contains `s` repeated `n` times.

Example EPs for `s`:

- zero-length strings
- string containing whitespaces
- …

Example EPs for `n`:

- `0`
- negative values
- …

An EP may not have adjacent values.

📦 Consider the method `isPrime(int i): boolean` that returns true if `i` is a prime number.

EPs for `i`:

- prime numbers
- non-prime numbers

Some inputs have only a small number of possible values and a potentially unique behavior for each value. In those cases, you have to consider each value as a partition by itself.

📦 Consider the method `showStatusMessage(GameStatus s): String` that returns a unique `String` for each of the possible values of s (`GameStatus` is an `enum`). In this case, each possible value of `s` will have to be considered as a partition.

Note that the EP technique is merely a heuristic and not an exact science, especially when applied manually (as opposed to using an automated program analysis tool to derive EPs). The partitions derived depend on how one 'speculates' the SUT to behave internally. Applying EP under a glass-box or gray-box approach can yield more precise partitions.

📦 Consider the EPs given above for the method `isValidMonth`. A different tester might use these EPs instead:

- [1 … 12]: the range that produces `true`
- [all other integers]: the range that produces `false`

📦 Some more examples:

Specification	Equivalence partitions
`isValidFlag(String s): boolean` Returns `true` if `s` is one of [`"F"`, `"T"`, `"D"`]. The comparison is case-sensitive.	[`"F"`] [`"T"`] [`"D"`] [`"f"`, `"t"`, `"d"`] [any other string][null]
`squareRoot(String s): int` Pre-conditions: `s` represents a positive integer. Returns the square root of `s` if the square root is an integer; returns `0` otherwise.	[`s` is not a valid number] [`s` is a negative integer] [`s` has an integer square root] [`s` does not have an integer square root]

Exercises

Intermediate

★★★☆ Can apply EP for OOP methods

When deciding EPs of OOP methods, you need to identify the EPs of all data participants that can potentially influence the behaviour of the method, such as,

- the target object of the method call
- input parameters of the method call
- other data/objects accessed by the method such as global variables. This category may not be applicable if using the black box approach (because the test case designer using the black box approach will not know how the method is implemented).

Consider this method in the `DataStack` class: `push(Object o): boolean`

- Adds `o` to the top of the stack if the stack is not full.
- Returns `true` if the push operation was a success.
- Throws
  - `MutabilityException` if the global flag `FREEZE==true`.
  - `InvalidValueException` if `o` is null.

EPs:

- `DataStack` object: [full] [not full]
- `o`: [null] [not null]
- `FREEZE`: [true][false]

Consider a simple Minesweeper app. What are the EPs for the `newGame()` method of the `Logic` component?

As `newGame()` does not have any parameters, the only obvious participant is the `Logic` object itself.

Note that if the glass-box or the grey-box approach is used, other associated objects that are involved in the method might also be included as participants. For example, the `Minefield` object can be considered as another participant of the `newGame()` method. Here, the black-box approach is assumed.

Next, let us identify equivalence partitions for each participant. Will the `newGame()` method behave differently for different `Logic` objects? If yes, how will it differ? In this case, yes, it might behave differently based on the game state. Therefore, the equivalence partitions are:

- `PRE_GAME`: before the game starts, minefield does not exist yet
- `READY`: a new minefield has been created and the app is waiting for the player's first move
- `IN_PLAY`: the current minefield is already in use
- `WON`, `LOST`: let us assume that `newGame()` behaves the same way for these two values

Consider the `Logic` component of the Minesweeper application. What are the EPs for the `markCellAt(int x, int y)` method? The partitions in **bold** represent valid inputs.

- `Logic`: PRE_GAME, **READY**, **IN_PLAY**, WON, LOST
- `x`: [MIN_INT..-1] **[0..(W-1)]** [W..MAX_INT] (assuming a minefield size of WxH)
- `y`: [MIN_INT..-1] **[0..(H-1)]** [H..MAX_INT]
- `Cell` at `(x,y)`: **HIDDEN**, MARKED, CLEARED

Boundary value analysis

What

★★☆☆ Can explain boundary value analysis

***Boundary Value Analysis (BVA)* is a test case design heuristic that is based on the observation that bugs often result from incorrect handling of boundaries of equivalence partitions.** This is not surprising, as the end points of boundaries are often used in branching instructions, etc., where the programmer can make mistakes.

The `markCellAt(int x, int y)` operation could contain code such as `if (x > 0 && x <= (W-1))` which involves the boundaries of x's equivalence partitions.

BVA suggests that when picking test inputs from an equivalence partition, values near boundaries (i.e. boundary values) are more likely to find bugs.

Boundary values are sometimes called *corner cases*.

Exercises

How

★★★☆ Can apply boundary value analysis

Typically, you should choose three values around the boundary to test: one value from the boundary, one value just below the boundary, and one value just above the boundary. The number of values to pick depends on other factors, such as the cost of each test case.

Some examples:

Equivalence partition	Some possible test values (boundaries are in bold)

[1-12]	0,**1**,2, 11,**12**,13
[MIN_INT, 0] (MIN_INT is the minimum possible integer value allowed by the environment)	**MIN_INT**, MIN_INT+1, -1, **0** , 1
[any non-null String] (assuming string length is the aspect of interest)	**Empty String**, **a String of maximum possible length**
[prime numbers] [“F”] [“A”, “D”, “X”]	No specific boundary No specific boundary No specific boundary
[non-empty Stack] (assuming a fixed size stack)	Stack with: no elements, **one element**, two elements, **no empty spaces**, only one empty space

Combining test inputs

Why

★★☆☆ 🏆 Can explain the need for strategies to combine test inputs

An SUT can take multiple inputs. You can select values for each input (using equivalence partitioning, boundary value analysis, or some other technique).

📦 An SUT that takes multiple inputs and some values chosen for each input:

- Method to test: `calculateGrade(participation, projectGrade, isAbsent, examScore)`
- Values to test:

Input	Valid values to test	Invalid values to test
participation	0, 1, 19, 20	21, 22
projectGrade	A, B, C, D, F	
isAbsent	true, false	
examScore	0, 1, 69, 70,	71, 72

Testing all possible combinations is effective but not efficient. If you test all possible combinations for the above example, you need to test 6x5x2x6=360 cases. Doing so has a higher chance of discovering bugs (i.e. effective) but the number of test cases will be too high (i.e. not efficient). Therefore, **you need smarter ways to combine test inputs that are both effective and efficient.**

Test Input Combination Strategies

★★☆☆ 🏆 Can explain some basic test input combination strategies

Given below are some basic strategies for generating a set of test cases by combining multiple test inputs.

📦 Let's assume the SUT has the following three inputs and you have selected the given values for testing:

SUT: `foo(char p1, int p2, boolean p3)`

Values to test:

Input	Values
p1	a, b, c
p2	1, 2, 3
p3	T, F

The *all combinations* strategy generates test cases for each unique combination of test inputs.

📦 This strategy generates 3x3x2=18 test cases.

Test Case	p1	p2	p3
1	a	1	T
2	a	1	F
3	a	2	T
...	...	...	...
18	c	3	F

The *at least once* strategy includes each test input at least once.

📦 This strategy generates 3 test cases.

Test Case	p1	p2	p3
1	a	1	T
2	b	2	F
3	c	3	VV/IV

VV/IV = Any Valid Value / Any Invalid Value

The *all pairs* strategy creates test cases so that for any given pair of inputs, all combinations between them are tested. It is based on the observation that a bug is rarely the result of more than two interacting factors. The resulting number of test cases is lower than the *all combinations* strategy, but higher than the *at least once* approach.

📦 This strategy generates 9 test cases:


See steps

Test Case	p1	p2	p3
1	a	1	T
2	a	2	T
3	a	3	F
4	b	1	F
5	b	2	T
6	b	3	F
7	c	1	T
8	c	2	F
9	c	3	T

A variation of this strategy is to test all pairs of inputs but only for inputs that could influence each other.

📦 Testing all pairs between p1 and p3 only while ensuring all p2 values are tested at least once:

Test Case	p1	p2	p3
1	a	1	T
2	a	2	F
3	b	3	T
4	b	VV/IV	F
5	c	VV/IV	T
6	c	VV/IV	F

The *random* strategy generates test cases using one of the other strategies and then picks a subset randomly (presumably because the original set of test cases is too big).

There are other strategies that can be used too.

Heuristic: Each Valid Input at Least Once in a Positive Test Case

★★★☆ 🏆 Can apply heuristic 'each valid input at least once in a positive test case'

Consider the following scenario.

SUT: `printLabel(String fruitName, int unitPrice)`

Selected values for `fruitName` (invalid values are underlined):

Values	Explanation
Apple	Label format is round
Banana	Label format is oval
Cherry	Label format is square
Dog	Not a valid fruit

Selected values for `unitPrice`:

Values	Explanation
1	Only one digit
20	Two digits
0	Invalid because 0 is not a valid price
-1	Invalid because negative prices are not allowed

Suppose these are the test cases being considered.

Case	fruitName	unitPrice	Expected
1	Apple	1	Print label
2	Banana	20	Print label
3	Cherry	0	Error message "invalid price"
4	Dog	-1	Error message "invalid fruit"

It looks like the test cases were created using the *at least once* strategy. After running these tests, can you confirm that the square-format label printing is done correctly?

- Answer: No.
- Reason: `Cherry` -- the only input that can produce a square-format label -- is in a negative test case which produces an error message instead of a label. If there is a bug in the code that prints labels in square-format, these tests cases will not trigger that bug.

In this case, a useful heuristic to apply is **each valid input must appear at least once in a positive test case**. `Cherry` is a valid test input and you must ensure that it appears at least once in a positive test case. Here are the updated test cases after applying that heuristic.

Case	fruitName	unitPrice	Expected

Case	fruitName	unitPrice	Expected
1	Apple	1	Print round label
2	Banana	20	Print oval label
2.1	Cherry	VV	Print square label
3	VV	0	Error message “invalid price”
4	Dog	-1	Error message “invalid fruit"

VV/IV = Any Invalid or Valid Value VV = Any Valid Value

Heuristic: No More Than One Invalid Input In A Test Case

★★★☆ 🏆 Can apply heuristic ‘no more than one invalid input in a test case’

Consider the test cases designed in [Heuristic: each valid input at least once in a positive test case].

After running these test cases, can you be sure that the error message “invalid price” is shown for negative prices?

- Answer: No.
- Reason: `-1` -- the only input that is a negative price -– is in a test case that produces the error message “invalid fruit”.

In this case, a useful heuristic to apply is **no more than one invalid input in a test case**. After applying that, you get the following test cases.

Case	fruitName	unitPrice	Expected
1	Apple	1	Print round label
2	Banana	20	Print oval label
2.1	Cherry	VV	Print square label
3	VV	0	Error message “invalid price”
4	VV	-1	Error message “invalid price"
4.1	Dog	VV	Error message “invalid fruit"

VV/IV = Any Invalid or Valid Value VV = Any Valid Value

Exercises

Mix

★★★☆ 🏆 Can apply multiple test input combination techniques together

Consider the calculateGrade scenario given below:

- SUT: `calculateGrade(participation, projectGrade, isAbsent, examScore)`
- Values to test: invalid values are underlined
  - participation: 0, 1, 19, 20, 21, 22
  - projectGrade: A, B, C, D, F
  - isAbsent: true, false
  - examScore: 0, 1, 69, 70, 71, 72

To get the first cut of test cases, let's apply the *at least once* strategy.

Test cases for calculateGrade V1

Case No.	partici- pation	projectGrade	isAbsent	examScore	Expected
1	0	A	true	0	...
2	1	B	false	1	...
3	19	C	VV/IV	69	...
4	20	D	VV/IV	70	...
5	21	F	VV/IV	71	Err Msg
6	22	VV/IV	VV/IV	72	Err Msg

VV/IV = Any Valid or Invalid Value, Err Msg = Error Message

Next, let's apply the *each valid input at least once in a positive test case* heuristic. Test case 5 has a valid value for `projectGrade=F` that doesn't appear in any other positive test case. Let's replace test case 5 with 5.1 and 5.2 to rectify that.

Test cases for calculateGrade V2

Case No.	partici- pation	projectGrade	isAbsent	examScore	Expected
1	0	A	true	0	...
2	1	B	false	1	...
3	19	C	VV	69	...
4	20	D	VV	70	...
5.1	VV	F	VV	VV	...
5.2	21	VV/IV	VV/IV	71	Err Msg
6	22	VV/IV	VV/IV	72	Err Msg

VV = Any Valid Value VV/IV = Any Valid or Invalid Value

Next, you have to apply the *no more than one invalid input in a test case* heuristic. Test cases 5.2 and 6 don't follow that heuristic. Let's rectify the situation as follows:

Test cases for calculateGrade V3

Case No.	partici- pation	projectGrade	isAbsent	examScore	Expected
1	0	A	true	0	...
2	1	B	false	1	...
3	19	C	VV	69	...
4	20	D	VV	70	...
5.1	VV	F	VV	VV	...
5.2	21	VV	VV	VV	Err Msg
5.3	22	VV	VV	VV	Err Msg
6.1	VV	VV	VV	71	Err Msg
6.2	VV	VV	VV	72	Err Msg

Next, you can assume that there is a dependency between the inputs `examScore` and `isAbsent` such that an absent student can only have `examScore=0`. To cater for the hidden invalid case arising from this, you can add a new test case where `isAbsent=true` and `examScore!=0`. In addition, test cases 3-6.2 should have `isAbsent=false` so that the input remains valid.

Test cases for calculateGrade V4

Case No.	partici- pation	projectGrade	isAbsent	examScore	Expected
1	0	A	true	0	...
2	1	B	false	1	...
3	19	C	false	69	...
4	20	D	false	70	...
5.1	VV	F	false	VV	...
5.2	21	VV	false	VV	Err Msg
5.3	22	VV	false	VV	Err Msg
6.1	VV	VV	false	71	Err Msg
6.2	VV	VV	false	72	Err Msg
7	VV	VV	true	!=0	Err Msg

Exercises

Statements about test input combinations

Combine test inputs for the `consume` method

More

Testing Based on Use Cases

★★★☆ Can explain test case design for use case based testing

Use cases can be used for system testing and acceptance testing. For example, the main success scenario can be one test case while each variation (due to extensions) can form another test case. However, note that use cases do not specify the exact data entered into the system. Instead, it might say something like `user enters his personal data into the system`. Therefore, the tester has to choose data by considering equivalence partitions and boundary values. The combinations of these could result in one use case producing many test cases.

To increase the E&E of testing, high-priority use cases are given more attention. For example, a scripted approach can be used to test high-priority test cases, while an exploratory approach is used to test other areas of concern that could emerge during testing.

Recap

Exercises

★★★☆ Can combine test case design techniques

Exercises