[L8P3]
Heuristics for Better Test Case Design

Test case design approaches

Exploratory testing vs scripted testing

Previously, two alternative approaches to test case design were discussed: exploratory vs scripted. That distinction was based on *when* the test cases are designed. In scripted approach, they are designed in advance. In the exploratory approach, they are designed on the fly.

Given next are alternative approaches based on some other aspects of testing.

Black-box vs white-box

This categorization is based on how much of SUT (software under test) internal details are considered when designing test cases. In the *black-box* approach, also known as *specification-based or responsibility-based testing*, test cases are designed exclusively based on the SUT's specified external behavior. In the *white-box* approach, also known as *glass-box or structured or implementation-based testing*, test cases are designed based on what is known about the SUT's implementation, i.e. the code.

Knowing *some* important information about the implementation can help in black-box testing. This kind of testing is sometimes called *gray-box* testing. For example, if the implementation of a sort operation uses an algorithm to sort lists shorter than 1000 items and another to sort lists longer than 1000 items, more meaningful test cases can then be added to verify the correctness of both algorithms.

Test case design techniques

Testing all possible ways of using the SUT requires writing an infinite number of test cases. For example, 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. As you can see from the example above, except for trivial systems, exhaustive testing is not possible!

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 have to be designed to make the best use of testing resources. In particular:

- Testing should be **effective**, i.e. it finds a high % of existing bugs. 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.

Given below are various tools and techniques used to improve E&E (Effectiveness and Efficiency) of testing.

Equivalence partitioning

Consider the testing of the following operation.

> isValidMonth (int m): boolean
>
> Description: checks if m is in the range [1..12]. returns true if m is in the range, false otherwise.

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 failed 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. Testing one of the inputs for a given range *should be* as good as exhaustively testing all inputs in that range.

Equivalence partitioning (EP) is a technique that uses the above observation to improve the efficiency and effectiveness of testing. By dividing possible inputs into groups which are likely to be processed similarly by the SUT, testing every possible input in each group is avoided. Such groups of input are called *equivalence partitions* (or *equivalence classes*). Equivalence partitioning can minimize test cases that are unlikely to find new bugs.

Equivalence partitions are usually derived from the specifications of the SUT. Preconditions and postconditions can also help in identifying partitions. For example, these could be equivalence partitions for the isValidMonth example:

- [MIN_INT ... 0] (below the range that produces 'true')
- [1 … 12] (the range that produces 'true')
- [13 … MAX_INT] (above the range that produces 'true')

Note that the equivalence partitioning technique does not suggest the number of test cases to pick from each partition. It depends on how thorough the test is.

Here's an example from an OO system. Consider the Minesweeper system that was explored previously. What are the equivalence partitions for the newGame() operation of the Logic component? In general, equivalence partitions of all *data participants*[8] that take part in the operation have to be considered. These include

- the target object of the operation call,
- input parameters of the operation call,
- and other data/objects accessed by the operation such as global variables.

Since newGame() does not have any parameters, the only 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 operation might also be included as participants. For example, Minefield object

[8] We use the term "data participants" to mean both objects and primitive values. Note that this is not a standard term

can be considered as another participant of the newGame() operation. Here, the black-box approach is assumed.

Next, identify equivalence partitions for each participant. Will the newGame operation 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 (i.e. before the game starts, minefield does not exist yet), READY (i.e. a new minefield has been created and waiting for player's first move), IN_PLAY, WON, LOST.

Here's another example. Consider the markCellAt(int x, int y) operation of the Logic component. Applying the technique described above, the equivalence partitions for the markCellAt operation can be obtained. The partitions in bold (green) represent valid inputs. Here, a Minefield of size WxH is assumed :

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

Here's another example. Consider the push operation from a DataStack class.

Operation: push(Object o): boolean

Description: Throws MutabilityException if the global flag FREEZE==true
Else, throws InvalidValueException if o==null.
Else, returns false if the stack is full.
Else, puts o at the top of the DataStack and returns true.

Here are the equivalence partitions for the push operation.

- DataStack object (ds): [full] [not full]
- o: [null] [not null]
- FREEZE: [true][false]

A test case for the push operation can be a combination of the equivalence partitions. Given below is such a test case.

id: DataStack_Push_001
description: checks whether pushing onto a full stack works correctly
input: ds is full, o!=null, FREEZE==false
expected output: returns false, ds remains unchanged

How are equivalence partitions combined and how many test cases to create? This question is addressed later in this handout. Moreover, the expected output should specify the return value as well as the state of all data participants of the operation that may be changed during the operation.

Knowledge of how the SUT behaves is used when deriving equivalence partitions for a given data participant. The table below illustrates some examples. However, note that the EP technique is merely a heuristic and not an exact science. 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.

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]
isPrimeNumber(int i): boolean Returns true if i is a prime number	[prime numbers] [non-prime numbers] * there are too many prime numbers to consider each one as a separate equivalence partition

When a data participant of an SUT is expected to be a subtype of a given type, each subtype that has a bearing on the SUT’s behavior should be treated as a separate equivalence partition. For example, consider the following operation.

> Operation: compare(Expression first, Expression second): boolean
> Description: returns true if both expressions evaluate to the same value

If the Expression is an abstract class which has two sub-classes Sum and Product, then the operation has to be tested for both parameter types Sum and Product.

Boundary Value Analysis

Boundary value analysis is another heuristic that can enhance the E&E of test cases designed using equivalence partitioning. It 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 the boundary are often used in branching instructions etc. where the programmer can make mistakes.

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

When doing boundary value analysis, values around the boundary of an equivalence partition are tested. Typically, three values are chosen: one value from the boundary, one value just below the boundary, and one value just above the boundary. The table below gives some examples of boundary values.

Equivalence partition	Some possible boundary values
[1-12]	0,1,2, 11,12,13
[MIN_INT, 0] *MIN_INT is the minimum possible integer value allowed by the	MIN_INT, MIN_INT+1, -1, 0 , 1

environment.	
[any non-null String]	Empty String, a String of maximum possible length
[prime numbers], ["F"] ["A", "B", "C"]	No specific boundary No specific boundary No specific boundary
[non-empty Stack] *we assume a fixed size stack	Stack with: one element, two elements, no empty spaces, only one empty space

Combining multiple inputs

Often, an SUT can take multiple data participants. Having selected test values for each data participant (using equivalence partitioning, boundary value analysis, or some other technique), how are they combined to create test cases? Consider the following scenario.

Operation to test:

calculateGrade(participation, projectScore, isAbsent, examScore)

Values to test (invalid values are underlined)

participation: 0, 1, 19, 20, <u>21</u>, <u>22</u>

projectScore: A, B, C, D, F

isAbsent: true, false

examScore: 0, 1, 69, 70, <u>71</u>, <u>72</u>

Given next are some techniques that can be used.

All combinations - This technique has a higher chance of discovering bugs. However, the number of combinations can be too high to test. In the above example, there are 6x5x2x6=360 cases to be tested.

At least once – This technique, illustrated in the table below, aims to include every value at least once.

Table 1. Test cases for calculateGrade (V1.0)

Case No	participation	projectScore	isAbsent	examScore	Expected
1	0	A	true	0	...
2	1	B	false	1	...
3	19	C	AVV	69	...
4	20	D	AVV	70	...
5	<u>21</u>	F	AVV	<u>71</u>	Err Msg
6	<u>22</u>	AVV	AVV	<u>72</u>	Err Msg

AVV = Any Valid Value, Err Msg = Error Message

This technique uses one test case to verify multiple input values. For example, test case 1 verifies SUT for participation==0, projectScore==A, isAbsent==true, and examScore==0. However, the expected result for test case 5 could be an error message, because of the invalid input data. This means that it remains unknown whether the SUT works correctly for the projectScore==F input as it is not being used by the other four test cases that do not produce an error message. Furthermore, if the error message was due to participation==21 then it does not guarantee that examScore==71 will also return the correct error message. This is why invalid input values should be tested one at a time, and not combined with the testing of valid input values. Doing this will result in nine test cases, as shown in the table below.

Table 2. Test cases for calculateGrade (V2.0)

Case No	participation	projectScore	isAbsent	examScore	expected
1	0	A	true	0	…
2	1	B	false	1	…
3	19	C	AVV	69	…
4	20	D	AVV	70	…
5	AVV	F	AVV	AVV	…
6	21	AVV	AVV	AVV	Err Msg
7	22	AVV	AVV	AVV	Err Msg
8	AVV	AVV	AVV	71	Err Msg
9	AVV	AVV	AVV	72	Err Msg

Other links between parameters can increase the number of test cases further. For example, assuming that an absent student can only have examScore==0, a link between isAbsent and examScore is established. To cater for the hidden invalid case arising from this, a 10th test case is added for which isAbsent==true and examScore!=0. In addition, test cases 3-9 should have isAbsent==false so that the input remains valid.

Table 0.3. Test cases for calculateGrade (V3.0)

Case No	participation	projectScore	isAbsent	examScore	Expected
1	0	A	true	0	…
2	1	B	false	1	…
3	19	C	false	69	…
4	20	D	false	70	…
5	AVV	F	false	AVV	…
6	21	AVV	false	AVV	Err Msg
7	22	AVV	false	AVV	Err Msg
8	AVV	AVV	false	71	Err Msg
9	AVV	AVV	false	72	Err Msg
10	AVV	AVV	true	!=0	Err Msg

All-pairs – This technique creates test cases so that for any given pair of parameters, all combinations between them are tested. It is based on the observations 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" approach, but higher than the "at least once" approach. The technique for creating such a set of test cases is beyond the scope of this handout.

Testing use cases

Use case testing is straightforward in principle: test cases are simply based on the use cases. It is used for system testing (i.e. testing the system as a whole). 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 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.

Worked examples

[Q1]

a) Explain the concept of *exploratory testing* using Minesweeper as an example.

b) Explain why exhaustive testing is not possible using the newGame operation (from Logic class in the Minesweeper case study) as an example.

[A1]

(a) When we test the Minesweeper by simply playing it in various ways, especially trying out those that are likely to be buggy, that would be exploratory testing.

b) Consider this sequence of test cases:

Test case 1. Start Minesweeper. Activate newGame() and see if it works.

Test case 2. Start Minesweeper. Activate newGame(). Activate newGame() again and see if it works.

Test case 3. Start Minesweeper. Activate newGame() three times consecutively and see if it works.

…

Test case 267. Start Minesweeper. Activate newGame() 267 times consecutively and see if it works.

Well, you get the idea. Exhaustive testing of newGame() is not possible.

[Q2]
Assume students are given matriculation number according to the following format:

[Faculty Alphabet] [Gender Alphabet] [Serial Number] [Check Alphabet]

E.g. CF1234X

The valid value(s) for each part of the matriculation number is given below:

Faculty Alphabet:
- Single capital alphabet
- Only 'C' to 'G' are valid

Gender Alphabet:
- Single capital alphabet
- Either 'F' or 'M' only

Serial Number
- 4-digits number
- From 1000 to 9999 only

Check Alphabet
- Single capital alphabet
- Only 'K', 'H', 'S', 'X' and 'Z' are valid

Assume you are testing the operation isValidMatric(String matric):boolen. Identify equivalence partitions and boundary values for the matriculation number.

[A2]
String length: (less than 7 characters), (7 characters), (more than 7 characters)

For those with 7 characters,

[Faculty Alphabet]: (‘C’, ‘G’), (‘c’, ‘g’), (any other character)

[Gender Alphabet]: (‘F’, ‘M’), (‘f’, ‘m’), (any other character)

[Serial Number]: (1000-9999), (0000-0999), (any other 4- characters string)

[Check Alphabet]: ('K', 'H', 'S', ‘X’, 'Z'), ('k', 'h', ’s’, ‘x’, 'z'), (any other character)

[Q3]
Given below is the overview of the method dispatch(Resource, Task), from an emergency management system (e.g. a system used by those who handle emergency calls from the public about incidents such as fires, possible burglaries, domestic disturbances, etc.). A task might need multiple resources of multiple types. For example, the task 'fire at Clementi MRT' might need two fire engines and one ambulance.

dispatch(Resource r, Task t):void

Overview: This method dispatches the Resource *r* to the Task *t.* Since this can dispatch only one resource, it needs to be used multiple times should the task need multiple resources.

Imagine you are designing test cases to test the method dispatch(Resource,Task). Taking into account equivalence partitions and boundary values, which different inputs will you combine to test the method?

[A3]

Test input for r	Test input for t
• A resource required by the task • A resource not required by the task • A resource already dispatched for another task • null	• A fully dispatched task • A task requiring one more resource • A task with no resource dispatched Considering the resource types required • A task requiring only one type of resources • A task requiring multiple types of resource • null

[Q4]
Given below is an operation description taken from a restaurant booking system. Use equivalence partitions and boundary value analysis technique to design a set of test cases for it.

boolean transferTable (Booking b, Table t)
Description: Used to transfer a Booking *b* to Table *t*, if *t* has enough room.
Preconditions: *t* has room for *b* , b.getTable() != t
Postconditions: b.getTable() == t

[A4]
Equivalence partitions

Booking:
Invalid: null, not null and b.getTable==t
Valid: not null and b.getTable != t

Table:
Invalid: null, not vacant, vacant but doesn't have enough room,

Valid: vacant and has enough room.

Boundary values:

Booking:

Invalid: null, not null and b.getTable==t

Valid:not null and b.getTable != t

Table:

Invalid: null, not vacant, (booking size == table size + 1)

Valid: (booking size == table size), (booking size == table size-1)

Test cases:

Test case	**Booking**	**Table**
1	null	Any valid
2	not null and b.getTable==t	Any valid
3	Any valid	null
4	Any valid	not vacant
5	Any valid	(booking size == table size + 1)
7	Any valid	(booking size == table size)
8	Any valid	(booking size == table size-1)

Note: We can use Bookings of different sizes for different test cases so that we increase the chance of finding bugs. If there is a minimum and maximum booking size, we should include them in those test cases.

[Q5]

Assume you are testing the add(Item) method specified below.

ItemList
add(Item):void contains(Item):boolean count():int

Assume i to be the Item being added.

Preconditions:

i != null [throws InvalidItemException if i == null]

contains(i) == false [throws DuplicateItemException if contains(i) == true]

count() < 10 [throws ListFullException if count() == 10]

Postconditions:

contains(i) == true;

new count() == old count()+1

Invariants: (an "invariant" is true before and after the method invocation).

0 <= count() <= 10

(a) What are the equivalence partitions relevant to testing the add(Item) method?

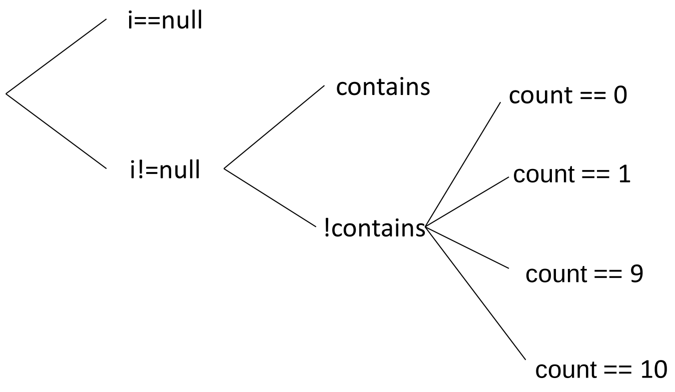
(b) What are the boundary and non-boundary values you will use to test the add(Item) method?

(c) Design a set of test cases to test the add(Item) method by considering the equivalence partitions and boundary values from your answers to (a) and (b) above.

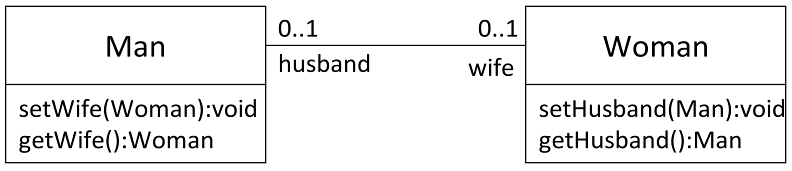
[A5]
(a)

i: i != null, i == null
list: contains(i)==true, contains(i)==false, count() < 10, count() == 10
list == null should NOT be considered.

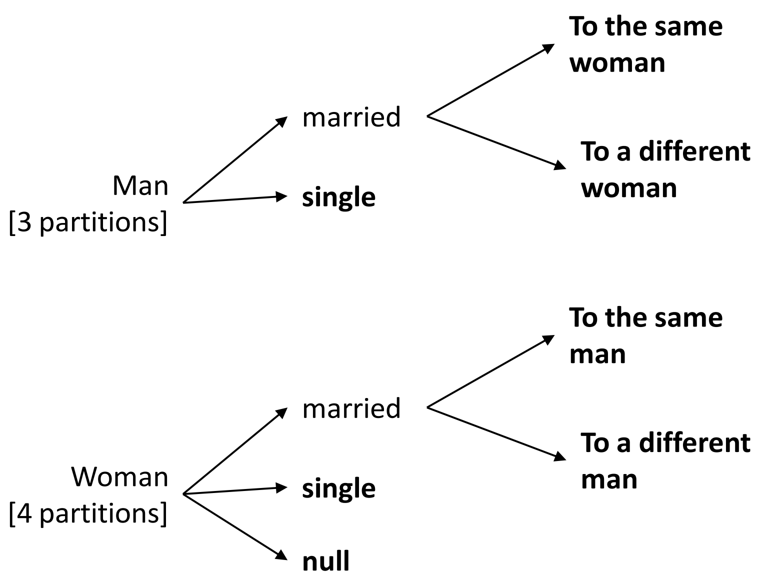
(b) list: count()==0, count()==9, count()==10; count()== [1|2|3|4|5|6|7|8] (1 preferred)
(c)



[Q6]
Use equivalence partitions and boundary values to choose test inputs for testing the setWife operation in the Man class.



[A6]



Partitioning 'married' as 'to same woman' and 'to different woman' seems redundant at first. Arguments for having it:

- The behavior (e.g. the error message shown) may be different in those two situations.
- The 'to same woman' partition has a risk of misunderstanding between developer and user. For example, the developer might think it is OK to ignore the request while the users might expect to see an error message.

[Q7]

b) Identify a set of equivalence partitions for testing isValidDate(String date) method. The method checks if the parameter date is a day that falls in the period 1880/01/01 to 2030/12/31 (both inclusive). The date is given in the format yyyy/mm/dd.

[A7]
Initial partitions: [null] **[10 characters long]**[shorter than 10 characters][longer than 10 characters]

For 10-character strings:

- **c1-c4:** [not an integer] [less than 1880] [**1880-2030 excluding leap years][leap years within 1880-2030 period]**[2030-9999]
- **c5: ['/']**[not '/']
- **c6-c7:** [not an integer][less than 1]**[2][31-day months: 1,3,5, 7,8, 10,12][30-day months: 4,6,9,11]** [13-99]
- **c8: ['/']**[not '/']
- **c9-c10:** [not an integer][less than 1]**[1-28][29][30][31]**[more than 31]

In practice, we often use 'trusted' library functions (e.g. those that come with the Java JDK or .NET framework) to convert strings into dates. In such cases, our testing need not be as thorough as those suggested by the above analysis. However, this kind of thorough testing is required if you are the person implementing such a trusted component.

[Q8]
Consider the following operation that activates an alarm when the landing gear (i.e. wheels structure) of an airplane should be deployed but has not been deployed.

```
isAlarmToBeSounded(timeSinceTakeOff,
                   timeToLand,
                   altitude,
                   visibility,
                   isGearAlreadyDeployed):boolean
```

Here is the logic for the operation. Landing gear must be deployed whenever the plane is within 2 minutes before landing or after takeoff, or within 2000 feet from the ground. If visibility is less than 1000 feet, then the landing gear must be deployed whenever the plane is within 3 minutes from landing or lower than 2500 feet. The operation should return true if the landing gear is not deployed (i.e. isGearAlreadyDeployed ==false) on meeting a deployment condition. Assume that the smallest unit of measurement for time is 1s and for distance it is 1 feet. Also, assume that invalid input such as negative values will not be produced by the sensors. takeoff_max, landing_max, altitude_max, visibility_max are the maximum values allowed by the instruments.

(a) List the equivalence partitions of the conditions that can lead to a decision about whether the alarm should be sounded.

(b) List the boundary and non-boundary values you will test, using no more than one non-boundary value per equivalence partition.

(c) Typically, for a critical system such as the "landing gear alarm system", all combinations of equivalence partitions must be checked. Ignoring that requirement for a moment, design a minimal set of test cases that includes each equivalence partition at least once. Use only boundary values.

(d) Now, you have been told that the correct functioning of the alarm is critically important if the plane is within 2 minutes of landing no matter what the other conditions are. How would you modify the test cases? Use only boundary values.

[A8]
(a) Equivalence partitions:

timeSinceTakeOff:	[0s .. 2m] [2m1s .. takeoff_max]
timeToLand :	[land_max.. 3m1s][3m .. 2m1s] [2m .. 0s]
altitude:	[0 .. 2000], [2001 .. 2500], [2501 .. altitude_max]
visibility:	[0 .. 1000], [1001 .. visibility_max]
isGearAlreadyDeployed:	[true] [false]

(b) take all values mentioned in (a), and one non-boundary value for each partition. For example,

timeSinceTakeOff: 0s, 1m, 2m, 2m1s, 3m, takeoff_max

(c) To test every partition at least once, we need only three test cases.

timeSinceTakeOff	**timeToLand**	**altitude**	**visibility**	**isGearAlreadyDeployed**
0s	land_max	0	0	[true]
2m1s	3m	2001	1001	[false]
-any value-	2m	2501	-any value-	-any value-

[Not required by the question] To test every boundary value at least once, we need six test cases.

timeSinceTakeOff	**timeToLand**	**altitude**	**visibility**	**isGearAlreadyDeployed**
0s	land_max	0	0	[true]
2m0s	3m1s	2000	1000	[false]
2m1s	3m	2001	1001	-any value-
takeoff_max	2m1s	2500	visibility_max	-any value-
-any value-	2m	2501	-any value-	-any value-
-any value-	0s	altitude_max	-any value-	-any value-

(d) Values 2m and 0s for the timeToLand should be combined with every possible combination of the other four inputs.

For TimeToLand ==2m -> 4x6x4x2 -> 192 test cases

For TimeToLand == 0s -> 4x6x4x2 -> 192 test cases

For TimeToLand == land_max, 3m1s, 3m, 2m1s -> 4 test cases

In total we need 192+192+4=388 test cases.

We can also test a non-boundary value from the critical equivalence partition (e.g. timeToLand ==1m), in which case, we will have an additional 192 test cases.