Software Testing and Quality Assurance – Black Box Testing

Joe Timoney

Black-box testing

- Black-box testing (functional testing) is based on the program specification, without consideration of internal structures of the software
- Aims to verify if the program meets the requirement specification
- Different approaches, each one has strengths and weaknesses

Black Box Testing types

1) Equivalence Partitioning (EP)
2) Boundary Value Analysis (BVA)
3) Combinational Testing
4) Sequential (State-Based) Testing
5) Testing with Random Data
6) Error Guessing/Expert Testing

White Box Test Types

7) Statement Coverage (SC)
8) Branch Coverage/Decision Coverage (BC/DC)
9) Condition Coverage (CC)
10) Decision Condition Coverage (DCC)
11) Multiple Condition Coverage (MCC)
12) Modified Condition/Decision Coverage (MCDC)
13) Path Coverage
14) Data flow (DU Pair) Coverage

Sequence of Testing

- In general, the normal usage of black-box and white-box testing techniques is as follows. Black-box testing is used initially to verify that the software satisfies the specification:

- Use Equivalence Partitioning to verify the basic operation of the software

- If the specification contains boundary values, use Boundary Value Analysis to verify correct operation at the boundaries

- If the specification states different processing for different combinations of inputs, use Combinational Testing to verify correct behaviour for each combination

Sequence of Testing

- If the specification contains state-based behaviour, or different behaviour for different sequences of inputs, then use Sequential Testing to verify this behaviour

- If there are reasons to suspect that there are faults in the code, perhaps based on past experience, then use Error Guessing/Expert Opinion to try and expose them

- If the typical usage of the software is known, then use Random test data to verify the correct operation under these usage patterns

Sequence of Testing

- For each of these tests, measure the statement and branch coverage. Normally the goal is to achieve 100% statement coverage and 100% branch coverage-because these are easy to measure automatically.
- If this has not been achieved, then white-box techniques can be used as follows:
- Use Statement Coverage to ensure that 100% of the statements have been executed.
- Use Branch Coverage to ensure that 100% of the branches have been taken.

Sequence of Testing

- Subsequently, if the code contains complex decisions, or if 100% branch coverage has not been achieved:
- Use Condition Coverage to ensure that every condition has been exercised
- Use Decision/Condition Coverage to ensure that every decision and every condition has been exercised
- Use Multiple Condition Coverage to ensure that every combination of conditions has been exercised

Sequence of Testing

- These white-box test techniques can be further augmented as follows:
- If the code contains complex end-to-end paths, then use Path Testing to ensure coverage of these
- If the code contains complex data usage patterns, then use DU Pair Testing to ensure coverage of these

Sequence of Testing

- In all cases, the decision to proceed with further tests is based on a cost-benefit trade-off: balancing the extra time and work required to do the extra tests justified against the extra confidence they will provide in the software quality.
- Often it is a judgement call as to what level of testing to execute.
- Fault Insertion can subsequently be used to measure the effectiveness of these tests in finding particular faults.

Note:

- Black-box tests can be written before, or in parallel with, the code (as they are based on the specifications).
- It normally serves no purpose to execute white-box tests before black-box tests.
- White-box testing can never be used as a substitute for black-box testing.
- White-box tests must be reviewed, and probably changed, every time the code is modified

Equivalence Partitioning

- Equivalence partitioning (EP) is a type of black-box testing

- In general, its goal is to verify the basic operation of the software

- Equivalence Partitioning is based on selecting representative values of each parameter from the equivalence partitions.

Equivalence Partitioning

- Each equivalence partition for each of the parameters is a test case. Both the inputs and the output should be considered.

- The technique invariably involves generating as few tests as possible: each new test should select data from as many uncovered partitions as possible.

- Error cases should be treated separately to avoid error hiding.

- The goal is to achieve 100% coverage of the equivalence partitions.

EP Test Cases

- Each partition for each input and output is a test case. It is good practice to give each test case for each SUT a unique identifier.
- It is often useful to use the prefix “EP-" for Equivalence Partition test cases.
- Note that the actual numerical values selected from the partitions are not the test cases, they are the test data.

EP Test Data

- Input test data is selected, based on test cases which are not yet covered. Ideally, each normal Test will include as many additional normal test cases as possible. Each error Test must only include one error test case.

- Expected output values are derived from the specification. However, the tester must ensure that all the test cases related to the output parameters are covered. It may be necessary to read the specification “backwards" to determine input values that will result in an output value being in the required equivalence partition.

EP Test Data

- Hint: it is usually easiest to identify test data by going through the test cases in order, selecting the next uncovered test case for each parameter, and then selecting an Equivalence Partition value.

- There is no reason to use different values from the same partition-in fact it is easier to review the test data for correctness if the one particular value is chosen from each partition, and then used throughout

Comment

- Equivalence Partitions provide a minimum level of black-box testing. At least one value has been tested from every input and output partition, using a minimum number of tests.
- These tests are likely to ensure that the basic data processing aspects of the code are correct. But they do not exercise the different decisions made in the code.
- This is important, as decisions are a frequent source of mistakes in the code. These decisions generally reflect the boundaries of input partitions, or the identification of combinations of inputs requiring particular processing.

EP Strengths

- Provides a good basic level of testing.
- Well suited to data processing applications where input variables may be easily identified and take on distinct values allowing easy partitioning.
- Provides a structured means for identifying basic test cases.

EP Weaknesses

- Correct processing at the edges of partitions is not tested.
- Combinations of inputs are not tested.
- The technique does not provide an algorithm for finding the partitions or selecting the test data.

Analysis of software specification

- Parameters
  - Input, output parameters of methods/functions
- Methods (and functions) have explicit and implicit parameters.
  - Explicit parameters are passed in the method call.
  - Implicit parameters are not: for example, in a C program they may be global variables; in a Java program, they may be attributes.

- Both types of parameter must be considered in testing. A complete specification should include all inputs and outputs.

Parameter ranges

- Parameter ranges
  - Natural values (data type) and specification-based ranges of values
  - Limits of ranges [lower value..upper value]
  - Example with Java data type:
    - Age: **short** integer, 16-bit signed two's complement integer,
      - natural range [short.MIN_VALUE..short.MAX_VALUE]: [2^{-15}.. 2^{15}-1] or [-32,768..+32,767]
      - specification-based ranges of Age: [0..120]

Parameter range for **int**

- **int** [Integer.MIN_VALUE..Integer.MAX_VALUE]

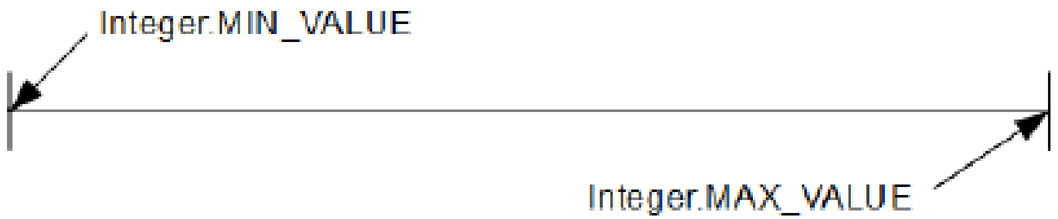


Figure 2.5: Natural Ranges

Parameter Ranges

- Parameter ranges
  - integer: 32-bit signed two's complement integer
  - byte: 8-bit signed two's complement integer
  - long: 64-bit two's complement integer
  - char: 16-bit Unicode character

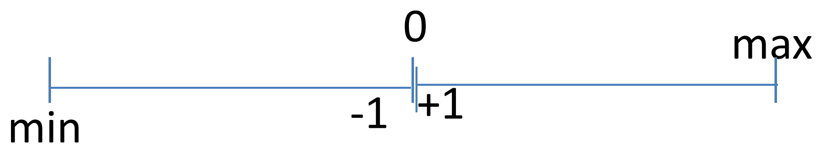
Parameter Ranges

- Natural ranges for types with no natural ordering are treated slightly differently-each value is a separate range containing one value:
    - **boolean** [true][false]
    - **enum** Colour {Red, Blue, Green} [Red][Blue][Green]
- Compound types, such as arrays and classes are more complicated to analyse, though the principles are the same.

Equivalence Partitions

- Equivalence Partitions (EP)
  - EP is a range of values for a parameter for which the specification states equivalent processing
  - Representative value from each partition is selected as test data.
  - Example:
    - Partition 1 (negative number): Integer.MIN_VALUE..-1 -1
    - Partition 2 (0) : 0..0
    - Partition 3(positive number): 1.. Integer.MAX_VALUE
  - Any value in the partition is processed equivalently to any other value

Equivalence Partitions

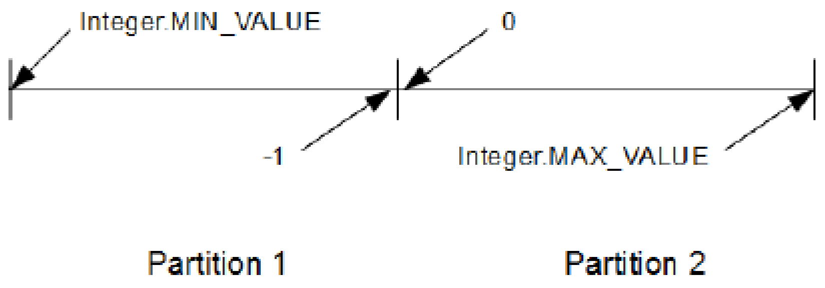


Example isNegative()

- Consider a method,
  - **boolean** `isNegative(int x)`,
- which accepts a single Java `int` as its input parameter. The method returns true if `x` is negative, otherwise false.
- From this specification, two equivalence partitions for the parameter `x` can be identified:
- 1. Integer.MIN VALUE..-1
- 2. 0..Integer.MAX VALUE

Example isNegative()

- These specification-based ranges are called Equivalence Partitions-according to the specification, any value in the partition is processed equivalently to any other value.



Analysis of software specification

- Equivalence Partitions
  - Every value for every parameter is in one partition
  - No values between partition
  - Natural range of the parameter provides the upper and lower limits for partitions
  - Any one value can be selected to represent any other value in the same partition. Traditionally a value in the middle is picked
  - A single test uses a single value in a partition

EP

- Equivalence Partitions are useful for testing the fundamental operation of the software:
- if the software fails using EP values, then it is not worth testing with more sophisticated techniques until the faults have been fixed.

Equivalence Partitioning

- Description
  - Selecting representative values of each parameter from the equivalence partitions
  - The goal is to achieve 100% coverage of the equivalence partitions
    - How to specify EP ?
- Test case
  - Each partition for each input and output is a test case
- Test data
  - Values selected from the partitions

EP Example- fits()

- Specification:
- A plane has 120 seats. A flight can accommodate up to that number of passengers. If the passengers require extra comfort, the effective number of seats is reduced by 40 to ensure that every passenger has an empty seat next to them. The method fits() indicates whether a particular number of passengers can be accommodated with or without extra comfort.

Fits() specification

- **Status** fits(int passengers, boolean comfortFlag)
- **<u>Inputs</u>**
  - passengers: the number of passengers to be carried
  - comfortFlag: flag to indicate whether extra comfort is required
- **<u>Outputs</u>**
- return value:
  - SUCCESS if $\text{passengers} \leq 120$ and !comfortFlag
  - SUCCESS if $\text{passengers} \leq 80$ and comfortFlag
  - FAILURE if passengers>120, or if passengers>80 and comfortFlag
  - ERROR if any inputs are invalid (e.g. passengers<1)

- Status is defined as follows:
  - enum Status { SUCCESS, FAILURE, ERROR };

Processing for Passengers

- There are four types of processing defined:
  - 1. Enough seats are always available (1..80)
  - 2. Enough seats are conditionally available (81..120)
  - 3. Enough seats are never available (121..Integer.MAX VALUE)
  - 4. An error in the input value (≤ 0)

- Note that combinations of input parameters are not considered in Equivalence Partitioning.

Processing for comfortFlag and return value

- For comfortFlag there are two types of processing defined:
  - 1. Don't leave extra space for comfort (false)
  - 2. Leave extra space for comfort (true)
- For return value there are three types of processing defined:
  - 1. Enough seats are available (SUCCESS)
  - 2. Not enough seats are available (FAILURE)
  - 3. An error in the input parameters (ERROR)

Natural Ranges

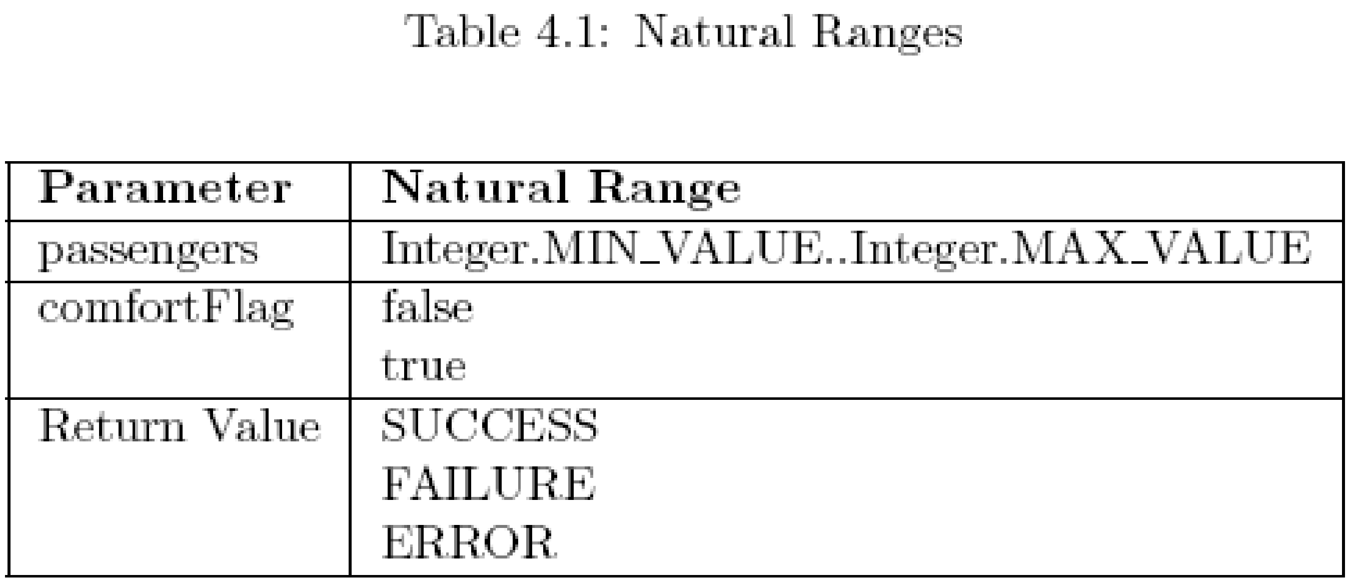
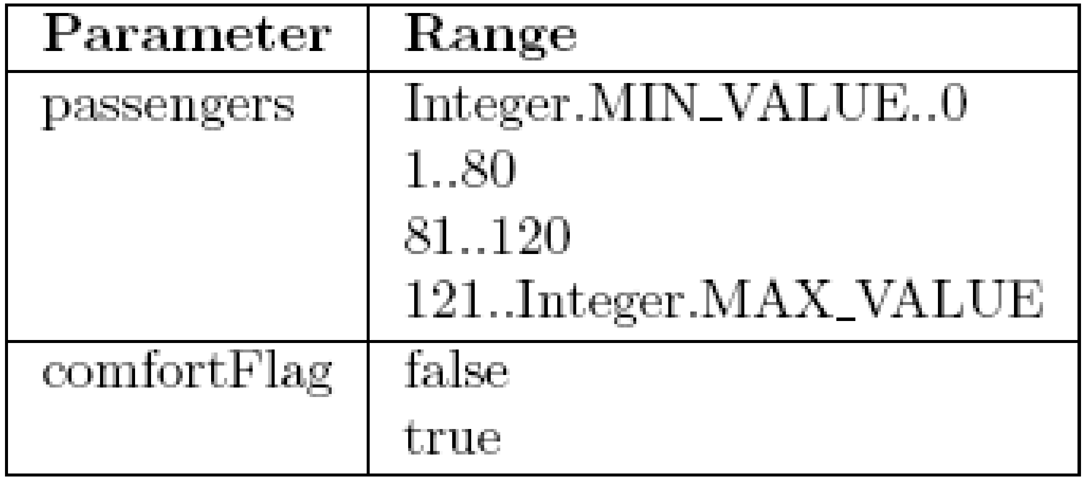


Table 4.1: Natural Ranges

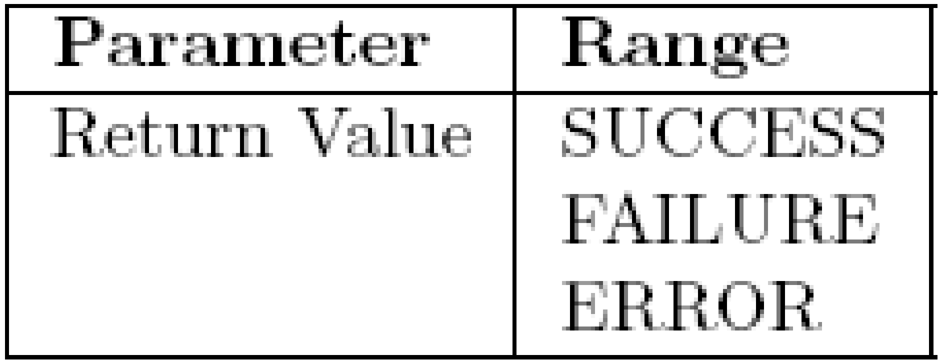
Parameter	**Natural Range**
passengers	Integer.MIN_VALUE..Integer.MAX_VALUE
comfortFlag	false
	true
Return Value	SUCCESS
	FAILURE
	ERROR

Input Partitions



Parameter	**Range**
passengers	Integer.MIN_VALUE..0
	1..80
	81..120
	121..Integer.MAX_VALUE
comfortFlag	false
	true

Output Partitions



Parameter	Range
Return Value	SUCCESS
	FAILURE
	ERROR

Test Cases

Case	Parameter	Range	Test
EP1*	passengers	Integer.MIN_VALUE..0	
EP2		1..80	
EP3		81..120	
EP4		121..Integer.MAX_VALUE	
EP5	comfortFlag	false	
EP6		true	
EP7	Return Value	SUCCESS	
EP8		FAILURE	
EP9*		ERROR	

Each partition is a test case, and requires a unique identifier. Reviewing the test cases for correctness is easier if numeric ranges are shown in order.

An asterisk (*) indicates an error case, which must be tested separately.

Test Data

- Each Test is specified by its associated test data. Each Test requires a unique identifier (unique across all tests for the method under test). The data for each test should include a unique identifier, the test cases covered, the input values, and the expected output value(s).
- Test input data is selected from arbitrary values within the equivalence partitions:
  - normally a central value is selected. Start with the first normal test case (i.e. not an error case). Then complete the tests for all the other normal test cases - for each additional test, data is selected to cover as many additional normal test cases as possible.

Test Data

- Finally, complete the error cases: each input error case must have its own unique test- there can only be one input error case covered by any one test.

Test Data

ID	Test Cases Covered	Inputs		Exp. Output
		passengers	comfortFlag	return value
T1.1	EP2,5,7	40	false	SUCCESS
T1.2	EP3,6,8	101	true	FAILURE
T1.3	EP4 [5,8]	200	false	FAILURE
T1.4*	EP1*,9*	-100	false	ERROR

Case	Parameter	Range	Test
EP1*	passengers	Integer.MIN_VALUE..0	
EP2		1..80	
EP3		81..120	
EP4		121..Integer.MAX_VALUE	
EP5	comfortFlag	false	
EP6		true	
EP7	Return Value	SUCCESS	
EP8		FAILURE	
EP9*		ERROR	

Case	Parameter	Range	Test
EP1*	passengers	Integer.MIN_VALUE..0	T1.4
EP2		1..80	T1.1
EP3		81..120	T1.2
EP4		121..Integer.MAX_VALUE	T1.3
EP5	comfortFlag	false	T1.1
EP6		true	T1.2
EP7	Return Value	SUCCESS	T1.1
EP8		FAILURE	T1.2
EP9*		ERROR	T1.4

Test Data notes

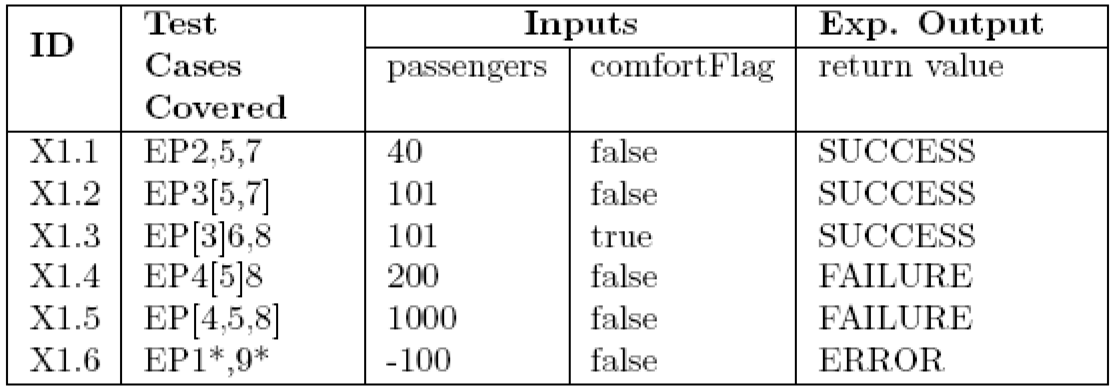
- Note that as an abbreviation “EP2,5,7" is used to mean “EP2 and EP5 and EP7".

- Note that duplicate test cases, already covered by a previous test, are specified in []'s-e.g. [5,8] in T1.3 indicates that EP5 and EP8 have been covered by a previous test. This is both useful in eliminating unnecessary tests, and may also act as a guideline for producing subsequent tests.

- Note that error test T1.4 does NOT cover Test Case EP5, even though the value of comfortFlag is false. This is due to error hiding-only one input error test case, and no input non-error test cases, are covered by an error test.

What not to do!

- Do not provide a separate test for each test case.
- After completing the test data, check whether there are any unnecessary tests.
- Also, check whether reorganisation of the test cases might provide fewer tests.



ID	Test Cases Covered	Inputs		Exp. Output
		passengers	comfortFlag	return value
X1.1	EP2,5,7	40	false	SUCCESS
X1.2	EP3[5,7]	101	false	SUCCESS
X1.3	EP[3]6,8	101	true	SUCCESS
X1.4	EP4[5]8	200	false	FAILURE
X1.5	EP[4,5,8]	1000	false	FAILURE
X1.6	EP1*,9*	-100	false	ERROR

Non-optimal test data set

Boundary Value Analysis

- In general, the goal of BVA is to find faults in the software associated with decisions.
- The type of processing applied depends on the equivalence partitions of the inputs, and the correctness of the decisions tends to be associated with the boundaries of these partitions.

BVA Description

- Programming faults are often related to the incorrect processing of boundary conditions, so an obvious extension to Equivalence Partitioning is to select two values from each partition: the bottom and the top values.
- This doubles the number of tests, but is more likely to find boundary-related programming faults.
- Each boundary value for each parameter is a test case.

BVA Description

- As for Equivalence Partitioning, the number of tests is minimised by selecting data that includes as many uncovered test cases as possible in each new test.
- Error tests are as always considered separately- only one error boundary value is included per error test.
- The goal is to achieve 100% coverage of the boundary values.

BVA Test Cases

- Each boundary value for each partition for each input and output is a test case.
- It is good practice to give each test case for each SUT a unique identifier.
- It is often useful to use the prefix “BVA-" for Boundary Value Analysis test cases.
- Note that, unlike Equivalence Partitions, the values selected from the boundaries are the test cases

BVA Test Data

- Input test data is selected, based on test cases which are not yet covered.
- Ideally, each normal Test will include as many additional normal test cases as possible.
- Each error Test must only include one error test case.

BVA Comment

- There is little published evidence that using Boundary Values improves the effectiveness of testing, but experience indicates that this is likely to cover significantly more possible errors than Equivalence Partitions.
- Note that Boundary Value Analysis provides exactly the same test cases as Equivalence Partitioning for boolean and enumerated parameters

BVA Appraisal

- Strengths
  - Test data values are provided by the technique.
  - Tests focus on areas where faults are more likely to be found.
- Weaknesses
  - Combinations of inputs are not tested.

Boundary Values

- Each Equivalence Partition has an upper and lower boundary value.
- Experience has shown that many software failures are due to the incorrect handling of limits,
- Thus Boundary Values (BV) provide a useful increase in sophistication over Equivalence Partitions. But this is at the cost of doubling the number of tests

Boundary Value Analysis

- For the example isNegative(int x), the boundary values for x are as follows:
  - 1. Integer.MIN VALUE
  - 2. -1
  - 3. 0
  - 4. Integer.MAX VALUE
- And the boundary values for the return value are the same as the equivalence partitions (as it is an enumerated type)-each EP is a range with only one data value:
  - 1. true
  - 2. false

Picking Boundary Values

- Some rules for picking boundary values:
- Every parameter has a boundary value at the top and bottom of every equivalence partition.
- For a contiguous data type, the successor to the value at the top of one partition must be the value at the bottom of the next.
- The natural range of the parameter provides the ultimate maximum and minimum values.
- It is important to note that boundary values do not overlap, and that there is no gap between partitions.

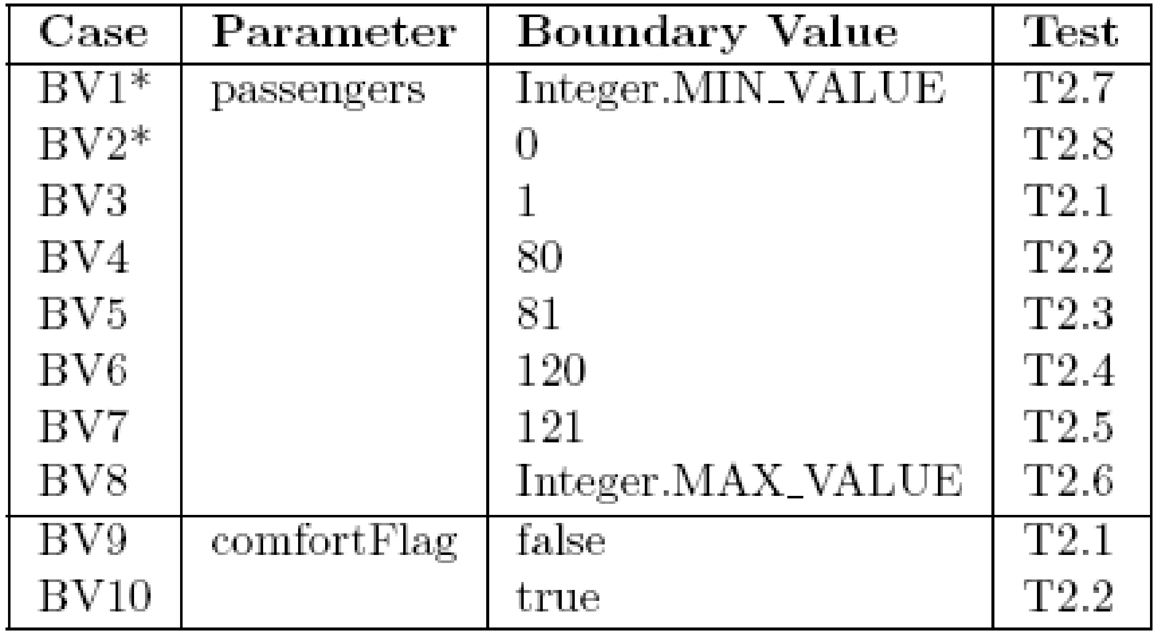
BVA Shorthand

- A convenient shorthand for specifying partitions and their boundary values is as follows:
- x: [Integer.MIN VALUE..-1][0..Integer.MAX VALUE]
- return value: [true][false]

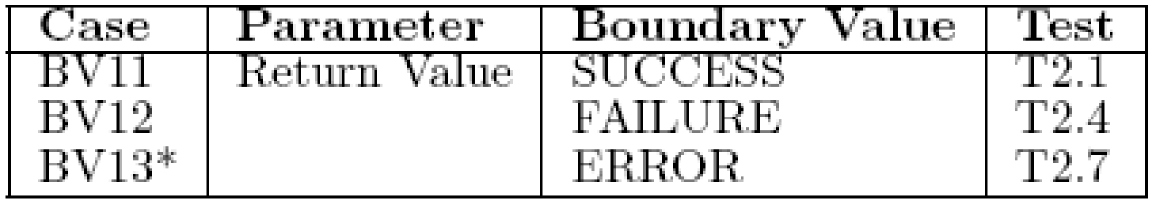
BVA Test Cases

- Boundary Values are the upper and lower values for each Equivalence Partition.
- Having identified the partitions, identifying the boundary values is straightforward.
- Test Cases: Each boundary value is a test case. Approach this systematically: unless there is a good reason not to, consider boundary values in increasing order (or in the order in which the values are defined).
- However, it is good practice to deal with all the error tests separately.

BVA Test Cases

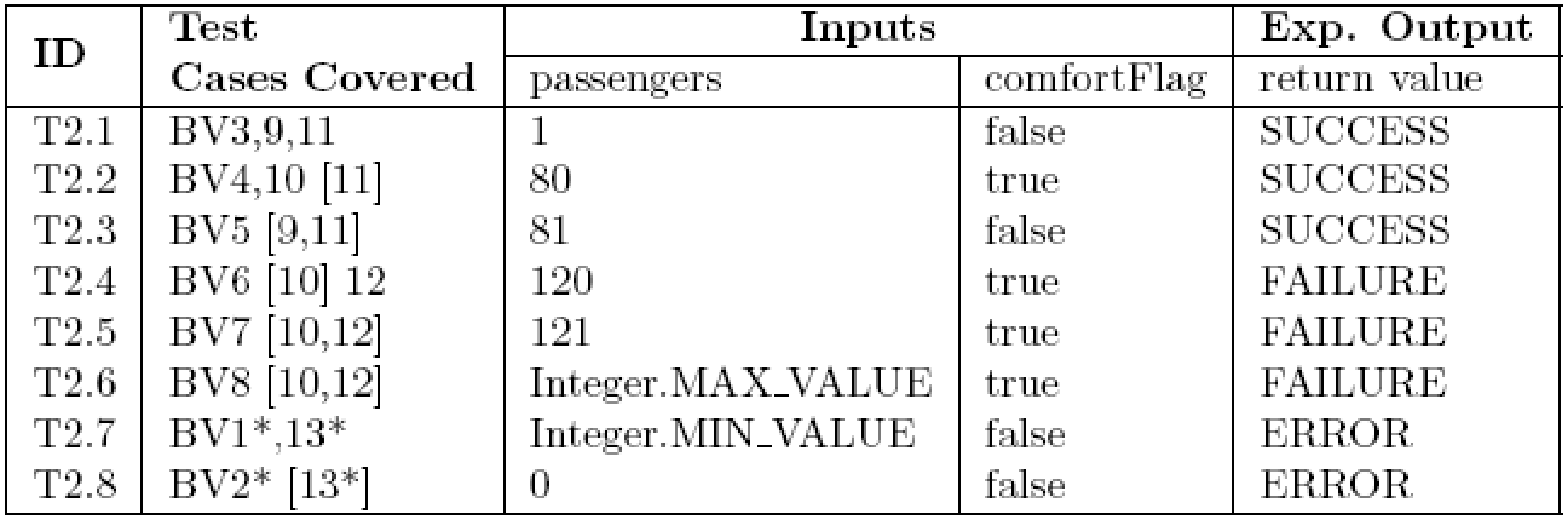


Case	Parameter	Boundary Value	Test
BV1*	passengers	Integer.MIN_VALUE	T2.7
BV2*		0	T2.8
BV3		1	T2.1
BV4		80	T2.2
BV5		81	T2.3
BV6		120	T2.4
BV7		121	T2.5
BV8		Integer.MAX_VALUE	T2.6
BV9	comfortFlag	false	T2.1
BV10		true	T2.2



Case	Parameter	Boundary Value	Test
BV11	Return Value	SUCCESS	T2.1
BV12		FAILURE	T2.4
BV13*		ERROR	T2.7

BVA Test Data



ID	Test Cases Covered	Inputs		Exp. Output
		passengers	comfortFlag	return value
T2.1	BV3,9,11	1	false	SUCCESS
T2.2	BV4,10 [11]	80	true	SUCCESS
T2.3	BV5 [9,11]	81	false	SUCCESS
T2.4	BV6 [10] 12	120	true	FAILURE
T2.5	BV7 [10,12]	121	true	FAILURE
T2.6	BV8 [10,12]	Integer.MAX_VALUE	true	FAILURE
T2.7	BV1*,13*	Integer.MIN_VALUE	false	ERROR
T2.8	BV2* [13*]	0	false	ERROR

BVA Tests

- At the expense of approximately twice the number of tests, the minimum and maximum value of each Equivalence Partition has been tested at least once, using a minimum number of tests.

- However, combinations of different boundary values have not been exhaustively tested: in this example there is a limit of 128 candidate combinations (not all are possible).

Combinations of Values

- Some programs do simple data processing just based on the input values. Other programs exhibit different behaviour based on the combinations of input values.

- Handling complex combinations correctly is likely to be a source of faults, and they need to be analysed to derive associated test cases.

Combinations of Values

- There are a number of different techniques for identifying relevant combinations, such as Cause-Effect Graphs and Decision Tables.
- The recommended technique is Truth Tables- they are simplest to create, and tests can be derived directly from them.

Combinational Testing

- The analysis of combinations involves identifying all the different combinations of input *causes* to the software and their associated output *effects*.

- The causes and effects are described as logical statements (or predicates), based on the specification of the software.

- These expressions specify the conditions required for a particular variable to cause a particular effect.

Combinational Testing

- To identify a minimum subset of possible combinations that will test all the different behaviours of the program, a truth table is created.

- The inputs (“Causes") and outputs (“Effects") are specified as Boolean expressions (using predicate logic). Combinations of the causes are the inputs that will generate a particular response from the program.

Combinational Testing

- These causes and Effects are combined in a Boolean graph or truth table that describes their relationship.
- Test cases are then constructed that will cover all possible combinations of Cause and Effect.
- For *N* independent causes, there will therefore be a total of 2^N different combinations. The truth table specifies how the software should behave for each combination.

isNegative(int x)

- There is a single cause:
  - 1. x<0 (which can be true or false)
- And a single effect:
  - 1. the return value (which can be true or false)
- Notes:
- For mutually exclusive expressions, such as " x<0" and " x>=0", only one form of the expression is needed, as it can take on the value true or false.

- For Boolean variables, both expressions are not needed.
  - " variable==true" and “variable==false"
- The single expression " variable" (here, return value) can take on the value true or false.

isZero(int x)

- Consider the method boolean isZero(int x) which returns true if x is zero, and otherwise returns false.
- There is a single cause:
  - 1. x==0
- And a single effect:
  - 1. the return value

isLargest(int x, int y)

- Consider the method `int largest(int x, int y)`, which returns the largest of the two input values.
- The causes are:
  - 1. x>y
  - 2. x<y
- And the effects are:
  - 1. return value==x
  - 2. return value==y

isLargest(int x,int y) note

- Where there are three possible situations (here, x<y, x==y, and x>y) you need at least two expressions to cover them:
  - 1. If x>y, then x<y is false.
  - 2. If x<y, then x>y is false.
  - 3. If x==y, then x>y and x<y are both false.
- Where the effect is for an output (here the return value) to take on one of the input values, it is important to list all the possibilities, as they are not mutually exclusive.
- In this case, when x is equal to y, the output is equal to both x and y.

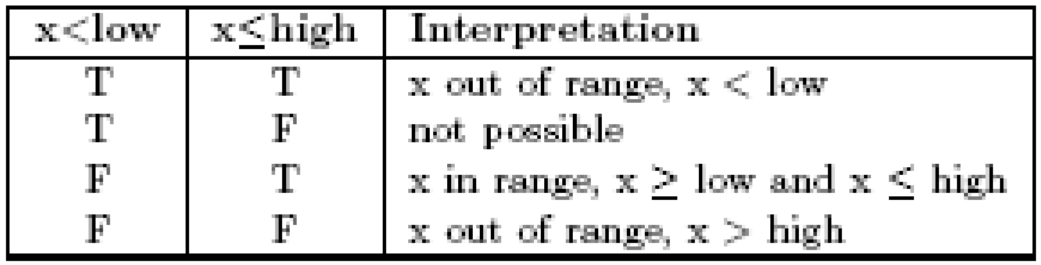
inRange(int x, int low, int high)

- Consider the method boolean inRange(int x, int low, int high), which returns true iff $low \leq x \leq high$ (if high<low, the output is undefined).
- It would be possible to create 3 causes as follows:
  - 1. x<low
  - 2. $low \leq x \leq high$
  - 3. x>high

Mutually Exclusive Rules

- However, mutually exclusive rules (where only one can be true) make for large and cumbersome truth tables, and a preferred set of causes is as follows:
  - 1. $x<low$
  - 2. $x \leq high$

Rules for inRange(int x, int low, int high)



x<low	x≤high	Interpretation
T	T	x out of range, $x < low$
T	F	not possible
F	T	x in range, $x \geq low$ and $x \leq high$
F	F	x out of range, $x > high$

This both reduces the number of causes, and allows for greater combinations of causes (as they are no longer completely mutually exclusive).

Effect for inRange(int x, int low, int high)

- There is only one effect for a boolean:
  - 1. return value

Truth Tables

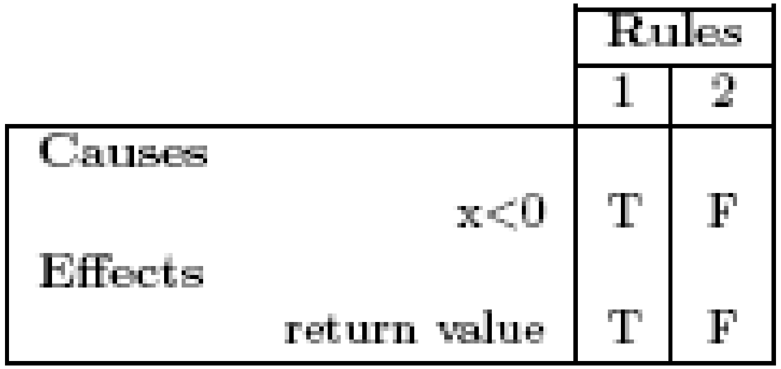
- A Truth Table is used to map the causes and effects through rules. Each rule states that under a particular combination of input causes, a particular set of output effects should result. It is critical to note that only one rule may be "active" at a time: the truth table is invalid if any input matches more than one rule!

- To generate the truth table, each cause is listed in a separate row, and then a different column is used to identify each combination of causes that creates a different effect. Each column is referred to as a" Rule" in the truth table{each rule is a different test case.

Truth Tables

- The truth tables for the three examples are shown next. Note that "T" is used as shorthand for true, and "F" for false.

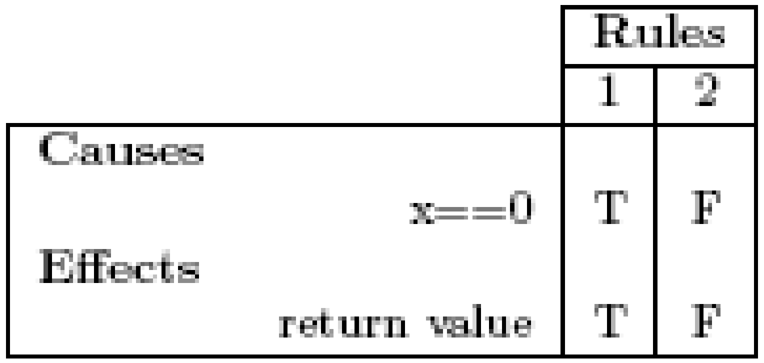
isNegative()



	Rules	
	1	2
Causes		
x<0	T	F
Effects		
return value	T	F

Rule 1 states that if x<0, then the return value is true.
Rule 2 states that if !(x<0), then the return value is false.

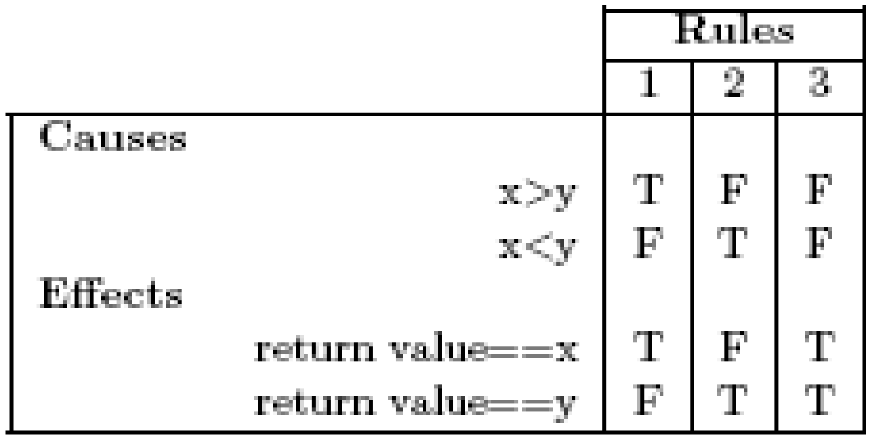
Truth Table for isZero()



	Rules	
	1	2
Causes		
x==0	T	F
Effects		
return value	T	F

Rule 1 states that if x==0, then the return value is true.
Rule 2 states that if !(x==0), then the return value is false.

Truth Table for isLargest()

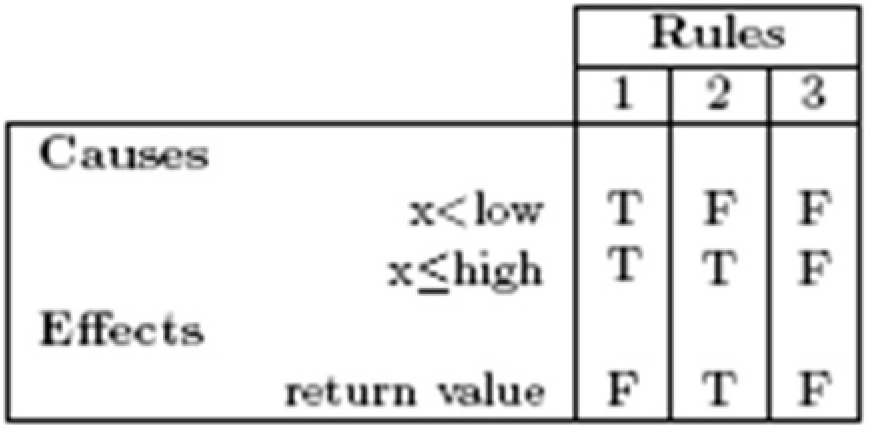


	Rules		
	1	2	3
Causes			
x>y	T	F	F
x<y	F	T	F
Effects			
return value==x	T	F	T
return value==y	F	T	T

. Rule 1 states that if (x>y) and !(x<y), then the return value is x.
. Rule 2 states that if !(x>y) and (x<y), then the return value is y.
. Rule 3 states that if !(x>y) and !(x<y), implying that (x==y), then the return value is equal to both x and y.

There is no Rule 4 as it is not logically possible

Truth Table for inRange()



	Rules		
	1	2	3
Causes			
$x < low$	T	F	F
$x \leq high$	T	T	F
Effects			
return value	F	T	F

Note that there is no Rule 4, as the combination " (x<low) and !(x ≤ high)" is not logically possible

. Rule 1 states that if (x<low) and (x≤high), then the return value is false.
. Rule 2 states that if !(x<low) and (x ≤ high), then the return value is true.
. Rule 3 states that if !(x<low) and !(x ≤ high), then the return value is false.

Don’t Care Conditions

- "Don't care" conditions exist where the value of a cause has no impact on the effect. These "Don't care" conditions are used to reduce the number of rules where the same output will be generated irrespective of whether the cause is true or false.

- In the worst case, if there are no "Don't care" conditions, *N* causes will create 2^N rules.

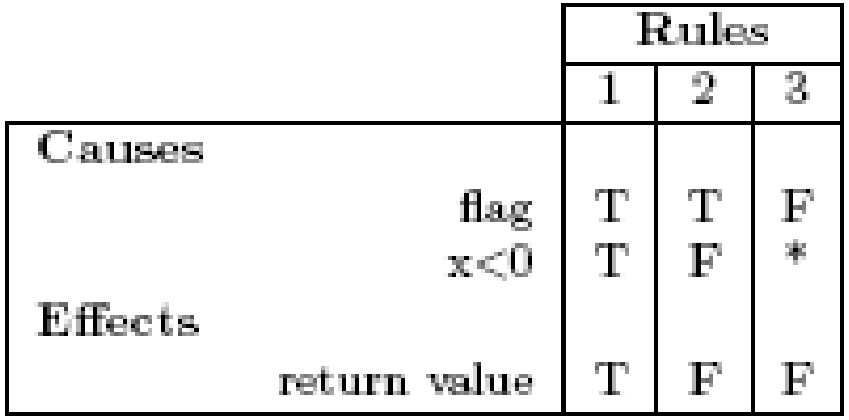
- "Don't care“ conditions are represented by using " *" for the causes in a truth table. The " *" is also used where an effect might be either true or false-the value is not determined by the specification.

condIsNeg(int x, boolean flag)

- `boolean condIsNeg(int x, boolean flag)` returns true when x is negative and flag is true, but always returns false otherwise.

- The causes are:
  - 1. flag
  - 2. x<0
- and the effect is:
  - return value (which can be true or false)

Truth Table for condIsNeg()



	Rules		
	1	2	3
Causes			
flag	T	T	F
x<0	T	F	*
Effects			
return value	T	F	F

Rule 1 states that when x<0 and flag, the return value is true.
Rule 2 states that when !(x<0) and flag, then the return value is false.
Rule 3 states that when !ag, the return value is false.

Note the don't-care condition for the cause x<0 when the flag is false

Finding *Don’t Care* conditions

- To systematically introduce don’t-care conditions, find two rules with exactly the same effect, and just one different value for a cause, and merge them using a don't care for that cause.
- The rules need not be next to each other.
- Merging mutually exclusive causes is difficult, and takes great care-another reason to avoid mutually exclusive causes.

Exercise

- In the development of truth tables, especially when not experienced in their use, it is useful to develop a series of candidate tables.
- The initial table has all the combinations systematically included, starting at all true, and ending with all false.
- The next candidate table has the impossible rules removed.
- The final table introduces don't-care conditions.

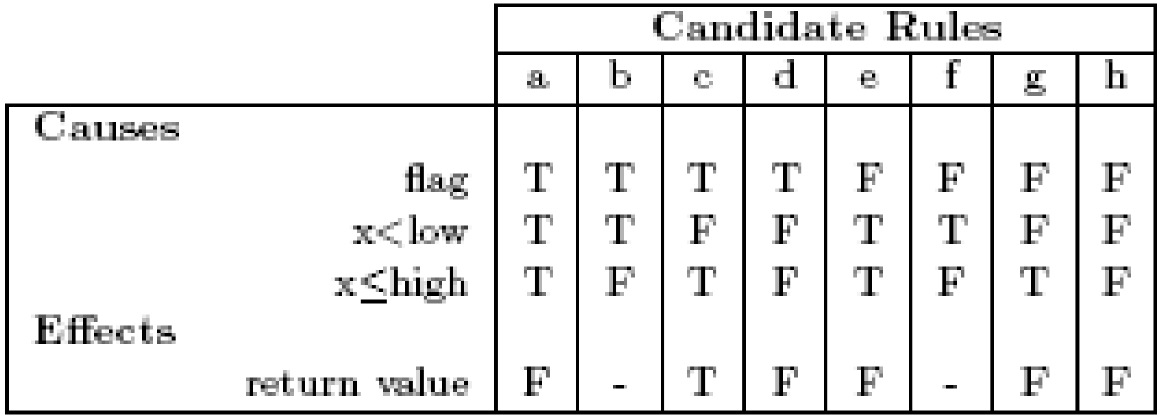
condInRange()

- `boolean condInRange(int x, int low, int high, boolean flag)`
- returns true iff the flag is true, and low $\leq$ x $\leq$ high (as before, if high<low, the output is undefined).
- Note that "-" is used for the effects on an impossible rule, and letters are used in place of numbers for the candidate rules to prevent confusion with the final rules.

Causes and effects

- The causes are:
  - 1. flag
  - 2. $x<low$
  - 3. $x\leq high$
- and the effect is:
  - 1. return value

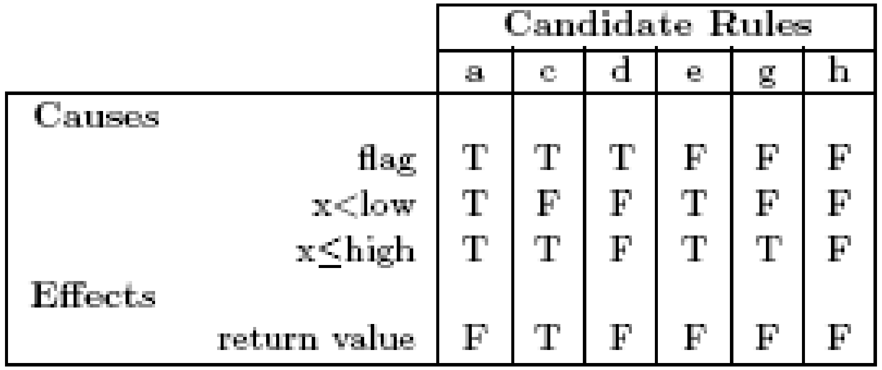
Initial Candidate Truth Table for condInRange()



	Candidate Rules							
	a	b	c	d	e	f	g	h
Causes								
flag	T	T	T	T	F	F	F	F
x<low	T	T	F	F	T	T	F	F
$x \le high$	T	F	T	F	T	F	T	F
Effects								
return value	F	-	T	F	F	-	F	F

Rules (b) and (f) are impossible and can be removed,

Candidate Truth Table for condInRange() with impossible rules removed

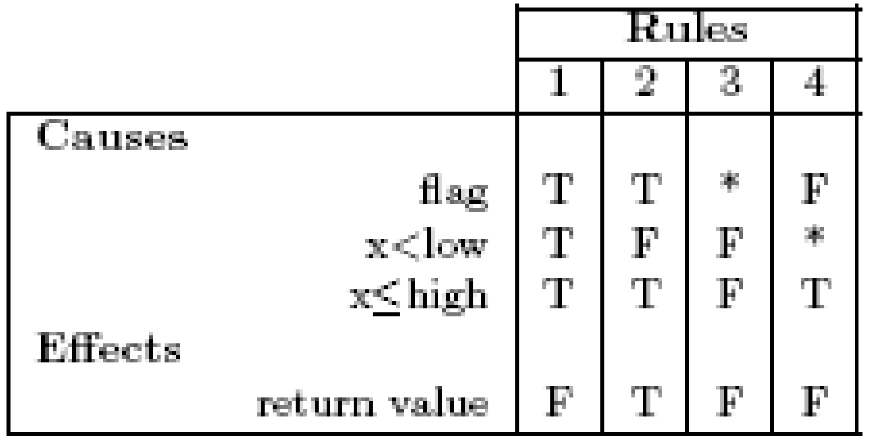


	Candidate Rules					
	a	c	d	e	g	h
Causes						
flag	T	T	T	F	F	F
x<low	T	F	F	T	F	F
$x \leq$high	T	T	F	T	T	F
Effects						
return value	F	T	F	F	F	F

First, rules (e) and (g) have the same effect, and one different cause: x<low is T in (e) and F in (g). They can be merged into a single rule, using a "*" for cause (x<low).

Then, rules (d) and (h) can be merged. They have the same effect, and just one different cause value (for flag).

Final Truth Table for condInRange()

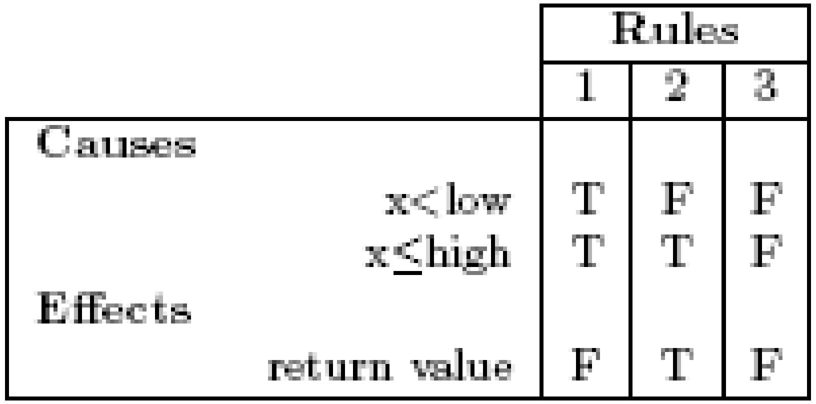


	Rules			
	1	2	3	4
Causes				
flag	T	T	*	F
x<low	T	F	F	*
$x \le$ high	T	T	F	T
Effects				
return value	F	T	F	F

Partial Truth Tables

- Another technique to reduce table size is to use partial truth tables.
- These only contain a subset of the causes, the others being defined to have a constant value.
- For example, the table on the previous slide is a partial truth table for condInRange() where $low \leq high$.

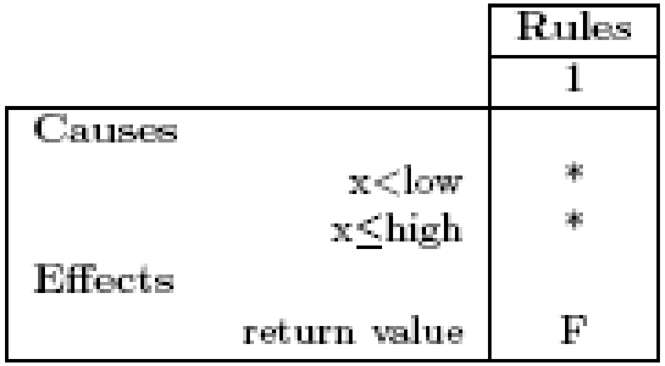
Truth Table for condInRange()



	Rules		
	1	2	3
Causes			
$x < low$	T	F	F
$x \leq high$	T	T	F
Effects			
return value	F	T	F

This is a partial truth table for condInRange() where flag==true, and low≤high.

Truth Table for condInRange()



	Rules
	1
Causes	
x<low	*
x≤high	*
Effects	
return value	F

This is a partial truth table for condInRange() where flag==false and low≤high.

Partial Truth Tables and Error Hiding

- Partial truth tables are used to avoid handling error conditions.
- Due to error hiding, errors tend to hide all other behaviour. Unless there are interesting combinations of inputs that cause errors, it is usual to present truthtables without error conditions.
- This is shown for condInRange(), where the error condition low>high is not considered in the truth-table.
- However if errors are caused by a combination of input values, and not just particular values or relationships, then they should be included in the truth-table.

Combinational Testing

- The rules in a truth table are used to identify the output value for all the possible combinations of inputs.
- Typically error cases are not covered in a truth table, in order to reduce its size.
- However, if combinations of inputs can cause errors, then a separate table should be used, covering just the error outputs.

Analysis for Causes and Effects

- It takes practice to identify reasonable causes: The partitions provide a good starting point.
- There is no one right answer - typically the causes can be stated in a large number of different (but equivalent) ways.
- Note: comfortFlag is boolean, so it is redundant to state “comfortFlag==true".

Analysis for Causes and Effects

- The number of causes should be minimized to reduce the size of the truth table-in particular, where a parameter has a range of values that provide a particular response, this can be expressed as a single cause.
- The (non-error) causes for this program, taken from the specification, can be expressed as follows:
  - passengers $\leq$80
  - passengers $\leq$ 120
  - comfortFlag

Analysis for Effects

- When considering the non-error combinations for testing, only the non-error effects need be considered:
  - return value == SUCCESS
  - return value == FAILURE

Truth Table

- The rules must be (a) complete, and (b) independent. One rule, and exactly one rule, must be selected by any combination of the input causes. Do not include impossible combinations of causes.
- Develop the rules systematically, starting with all F at the left-hand side of the table, and ending with all T at the right-hand side (or, alternatively, starting with all T, and ending with all F).
- Use don't care conditions (indicated by a '*') when the value of a cause has no impact on the effect of the rule.
- It sometimes helps to develop the full truth table first, and then remove redundant and impossible rules
- Note that we are not considering error cases: so the following truth tables are defined to always meet the conditions: passengers>0

Test Cases

- Each rule from the truth-table is a test case. If an SUT includes multiple truth-tables (for example, in a class) then it is useful to give each test case a unique identifier.

- Often a truth-table will only include normal cases, due to the number of rules required to describe all the possible error cases.

- If error cases are included, then it is often in a separate table for clarity.

Test Data

- Input test data is selected, based on test cases (rules) which are not yet covered. Each Test will cover exactly one test case (or rule).
- Expected output values are derived from the specification (the truth-table).

Picking Test Data

- It is usually easiest to identify test data by going through the test cases (rules) in order, and selecting a value for each parameter that matches the required causes.
- As for equivalence partitions, it can be easier to review a test for correctness if as few different values as possible are used for each parameter.
- For expected output, the value from the specification must, if the technique has been followed properly, match the required effect.

Comment

- The number of tests is reduced using `don't-care" conditions where the value of a particular cause has no effect on the output.
- This means that Combinational Testing does not test all the combinations of causes

Comment

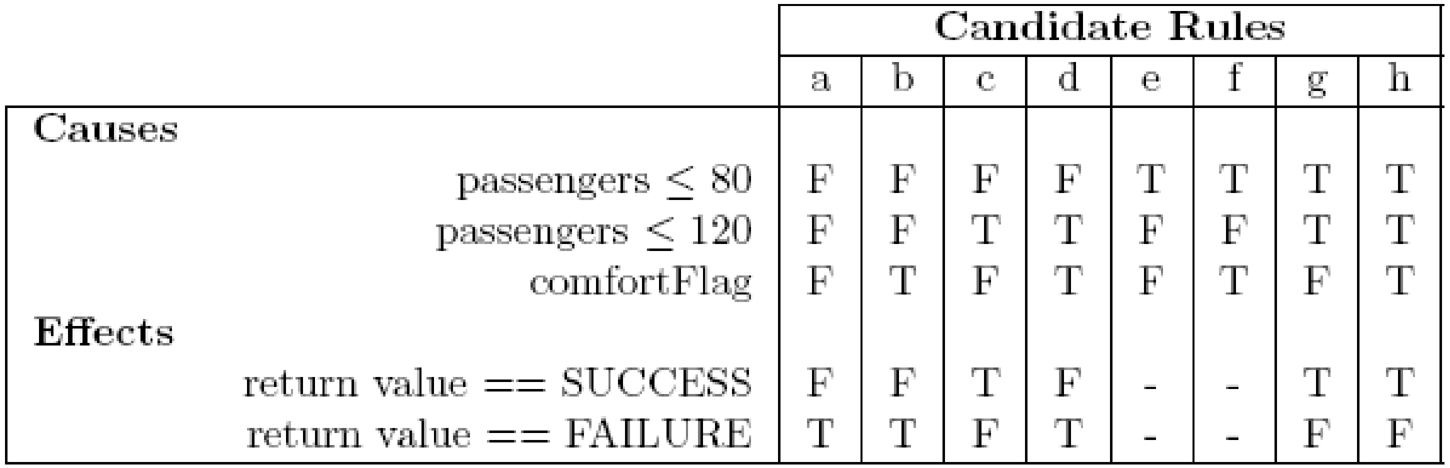
- Testing all the possible combinations of causes and effects would cause a very large number of tests for a typical program. Picking a minimum number of rules, based on using “don't care" conditions reduces the number of tests significantly-at the cost of reducing the test coverage.

- It should be noted that Equivalence Partitions/Boundary Values are complementary to Combinational Testing. There is again little published evidence as to the effectiveness of truth table testing, but experience in programming indicates that this is likely to cover different errors from Equivalence Partitions and Boundary Values.

Appraisal

- Strengths
  - Exercises combinations of test data
  - Expected outputs are created as part of the process
- Weaknesses
  - The truth tables can sometime be very large. The solution is to identify subproblems, and develop separate tables for each.
  - Very dependent on the quality of the specification - more detail means more causes and effects, which takes more time to test; less detail means less causes and effects, but less effective testing

Candidate Full Truth table for fits()

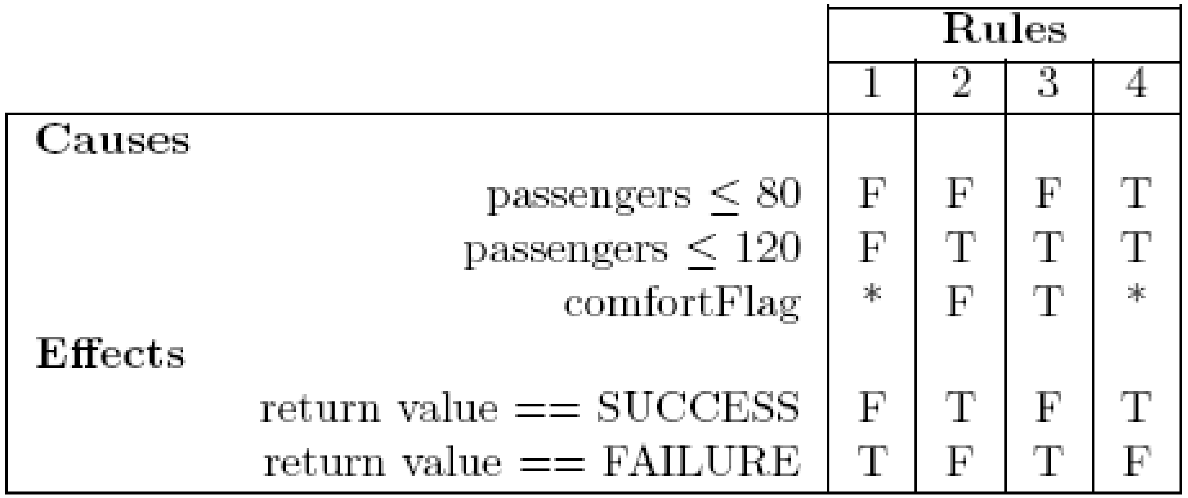


	Candidate Rules							
	a	b	c	d	e	f	g	h
Causes								
passengers ≤ 80	F	F	F	F	T	T	T	T
passengers ≤ 120	F	F	T	T	F	F	T	T
comfortFlag	F	T	F	T	F	T	F	T
Effects								
return value == SUCCESS	F	F	T	F	-	-	T	T
return value == FAILURE	T	T	F	T	-	-	F	F

Reducing the Truth Table

- 1. Candidate rules a and b have the same effect, and only one different cause value (for comfortFlag), so can be merged using a don't care for comfortFlag
- 2. Rules c and d must be retained, they have different effects
- 3. Rules e and f are impossible, and can be removed
- 4. Rules g and h have the same effect, and only one different cause value (for comfortFlag), so can be merged using a don't care for comfortFlag

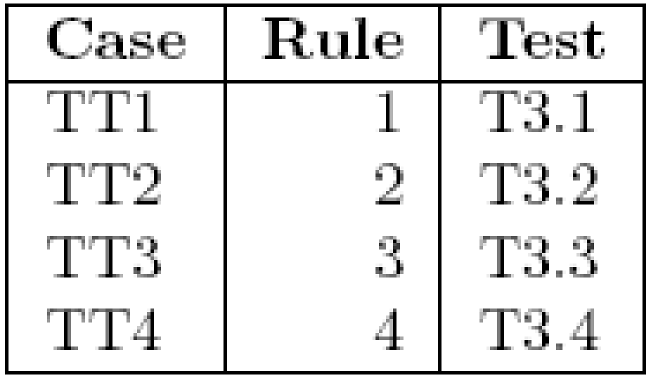
Reduced Truth Table for fits()



	Rules			
	1	2	3	4
Causes				
passengers ≤ 80	F	F	F	T
passengers ≤ 120	F	T	T	T
comfortFlag	*	F	T	*
Effects				
return value == SUCCESS	F	T	F	T
return value == FAILURE	T	F	T	F

Test Cases

- Each rule in the truth table (column) is a test case,



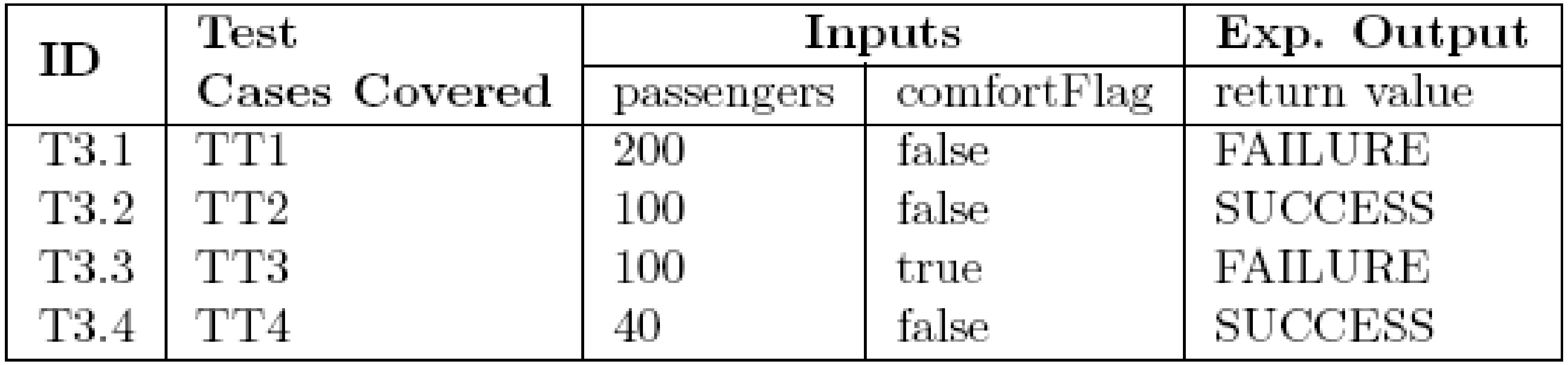
Case	Rule	Test
TT1	1	T3.1
TT2	2	T3.2
TT3	3	T3.3
TT4	4	T3.4

Note: if multiple partial truth tables are used, then the rules for each table must be numbered uniquely (e.g. Rule 1 in Table 1 as 1.1, Rule 1 in Table 2 as 2.1, etc.).

Test Data

- Each test case must be covered in a separate test-it is not possible to have multiple combinations in the same test.
- Test input data is selected by picking values that satisfy the causes and effects for the rule to be tested.

Test Data



ID	**Test Cases Covered**	**Inputs**		**Exp. Output**
		passengers	comfortFlag	return value
T3.1	TT1	200	false	FAILURE
T3.2	TT2	100	false	SUCCESS
T3.3	TT3	100	true	FAILURE
T3.4	TT4	40	false	SUCCESS

Random Testing

- Random testing can be used to achieve one or more of a number of goals-the main ones are as follows:
  - To use a probabilistic approach to search for faults not found with the previous, systematic approaches.
  - To use a probabilistic approach and demonstrate the correctness of software in a broader range of scenarios that explored with the previous, systematic approaches.
  - To estimate quality statistics, such as the mean time to failure (MTTF), by applying inputs which statistically match the expected inputs in real use, and measuring the failure rate.

Random Testing

- Test data is generated using random number generators. The distribution may be uniform, or chosen to mimic, in a statistical sense, the type of inputs that the program will receive in real use. If the specification is clearly written and thorough, then it should be possible to find the set(s) of possible input values.
- Typically uniformly distributed random numbers are used in Unit Testing, as the code may well be expected to work in a variety of different scenarios or within different programs. Statistically relevant distributions of random numbers are more often used in System Testing, where specific scenarios or customer usage patterns may be known.

Random Testing

- The overall goal of Random Testing is to achieve a “reasonable" coverage of the possible values for each input parameter, based on its distribution.

- This can be determined heuristically (using, for example, 10 random values), or based on a statistical sample size determined from the required confidence in the coverage

Test Cases

- Each test case is represented by a set of (random) input values, one for each parameter.
- If the test is fully automated, then each test case is represented by a distribution of values for a particular parameter. This will normally include the upper and lower limits, and the distribution to be used between these limits to select a random value.
- Each test case should be given a unique identifier.

Test Data

- Input test data is selected, normally automatically, based on the test cases.
- Expected output values are derived from the specification. This may be manual or automated. Automating the interpretation of a specification to produce the expected output is difficult. Three approaches are:

  - 1. Select the output partition at random first, then select random output values from this partition, and then select matching input values (which may be random or non-random depending on the specification).

  - 2. Write a Test Oracle in a higher-level language, which is more likely to be correct.

  - 3. Use post-conditions to determine the validity of an output after it is produced (rather than producing the expected output value before the test is run).

Comment

- Random test data generation is straightforward to implement and leads to a fast generation of test cases. However, calculating the Expected Output from the specification is as time consuming as for EP and BV.

- If the distribution/histogram of the real-world input data is known, then this provides a mathematical basis for selecting a set of input test case values. The measured test failure rate then provides an indication of the expected failure rate in use.

- However, if the distribution is not known, then the basis for choosing the random data may not reflect its use, and the failure rate cannot be predicted from the results.

- Furthermore, the random input data obtained may not have a sufficient set of illegal or extreme values, or even combinations of valid values, that will test the program thoroughly.

Appraisal

- Strengths
  - Fast generation of test cases
  - Can offer a mathematical basis for selecting an appropriate set of input values
- Weaknesses
  - Possibly an insufficient set of extreme or illegal values may be tested
  - If the distribution of the input is unknown the input values chosen may not reflect typical usage

Note: Regression and Stability Testing

- Randomisation can also be used to select a small set of tests from a large suite in order to execute the suite more quickly. This can be particularly useful for <u>Regression Testing</u>. The effectiveness of this depends on how the random tests are selected.
- More sophisticated techniques may be “directed", using feedback from each test to select data for the following test.
- Random data selection is sometimes used for <u>Stability Testing</u>, to ensure that no input data value causes the software to crash or raise unexpected exceptions. This technique is easy to implement in an automated manner, but is unlikely to find faults except in low-quality code.

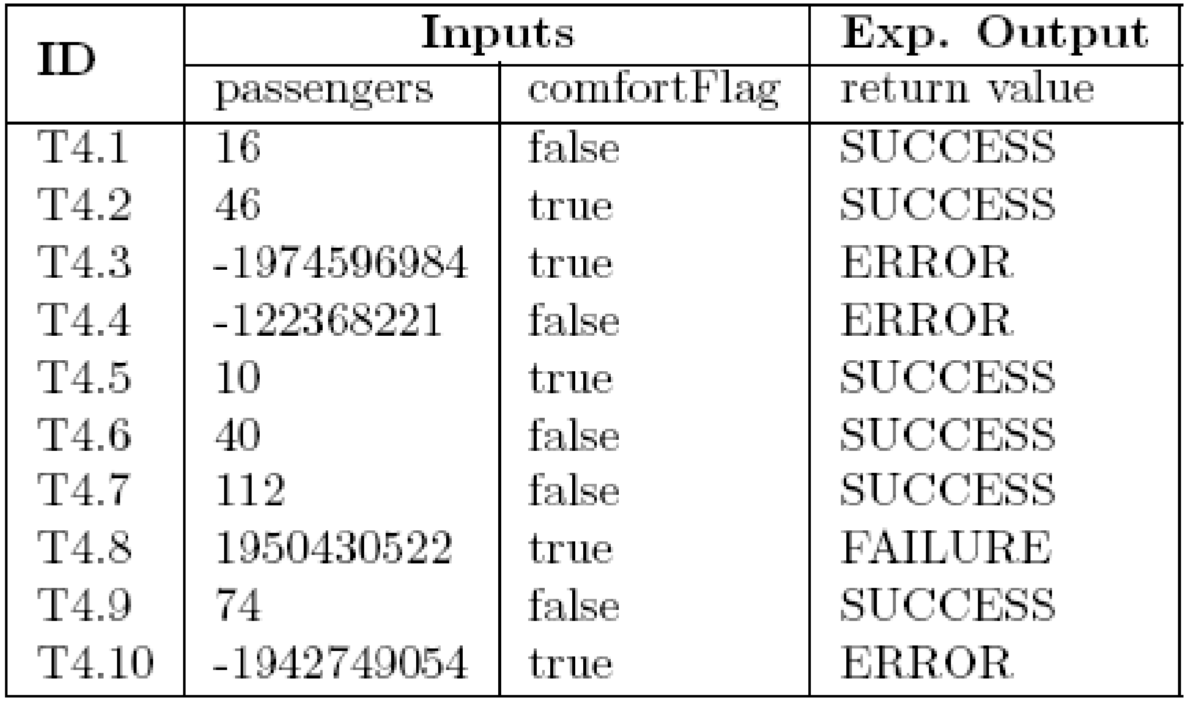
Random Testing

- Random data for Unit Testing can be easily generated by first randomly selecting which partition the output value is to be in, and then randomly selecting appropriate input values.
- Uniformly distributed random numbers are generally used for Unit Testing

Test Data

- The random values shown here have been generated (using a Java program and the Random class) as follows:
- 1. Select the result at random (SUCCESS, FAILURE, or ERROR)
- 2. Select the comfortFlag at random
- 3. Select a value for passengers from the range of values that will cause the required result:
    - ERROR: from Integer.MIN VALUE to 0
    - SUCCESS and false: from 1 to 120
    - SUCCESS and true: from 1 to 80
    - FAILURE and false: from 121 to Integer.MAX VALUE
    - FAILURE and true: from 81 to Integer.MAX VALUE

Test Data



ID	Inputs		Exp. Output
	passengers	comfortFlag	return value
T4.1	16	false	SUCCESS
T4.2	46	true	SUCCESS
T4.3	-1974596984	true	ERROR
T4.4	-122368221	false	ERROR
T4.5	10	true	SUCCESS
T4.6	40	false	SUCCESS
T4.7	112	false	SUCCESS
T4.8	1950430522	true	FAILURE
T4.9	74	false	SUCCESS
T4.10	-1942749054	true	ERROR

Note that random data values are normally selected at runtime using test automation, and not specified beforehand as shown here.

Error Guessing

- The goal of error guessing, in general, is to try find faults in the software based on the experience of domain experts, or software experts.
- This is an ad-hoc approach, based on intuition and experience. Test data is selected that is likely to expose faults in the code. Some typical examples of inputs likely to cause problems are given on the next slide.

Typical Errors

- Empty or null strings, arrays, lists, and class references. These may find code that does not check for empty or non-null values before using them.

- Zero as a value, or as a count of instances or occurrences. These may find divide-by-zero faults.

Typical Errors

- Spaces or null characters in strings. This may find code that does not process strings correctly, or does not trim whitespace before trying to extract data from the string.

- Negative numbers. These may find faults in code that only expects to receive positive numbers.

Test Cases

- The tester selects values which are likely to produce errors. Each value is a test case.
- Each test case should have a unique identifier.
- This technique can produce both normal and error test cases.
- The values selected are those that are likely to expose faults in the code, they are not necessarily illegal values.

Test Data

- Input test data is selected, based on test cases which are not yet covered.
- As for other test techniques, error cases should be executed individually.
- Expected output values are derived from the specification.
- It may be required to read the specification “backwards" to determine input values for output parameter test cases.

Comment

- With experienced testers, this can be a very effective complement to other testing techniques.

- It depends on how well the testers know the types of mistakes that the developers are likely to make, or mistakes that have a high impact on the final product.

Appraisal

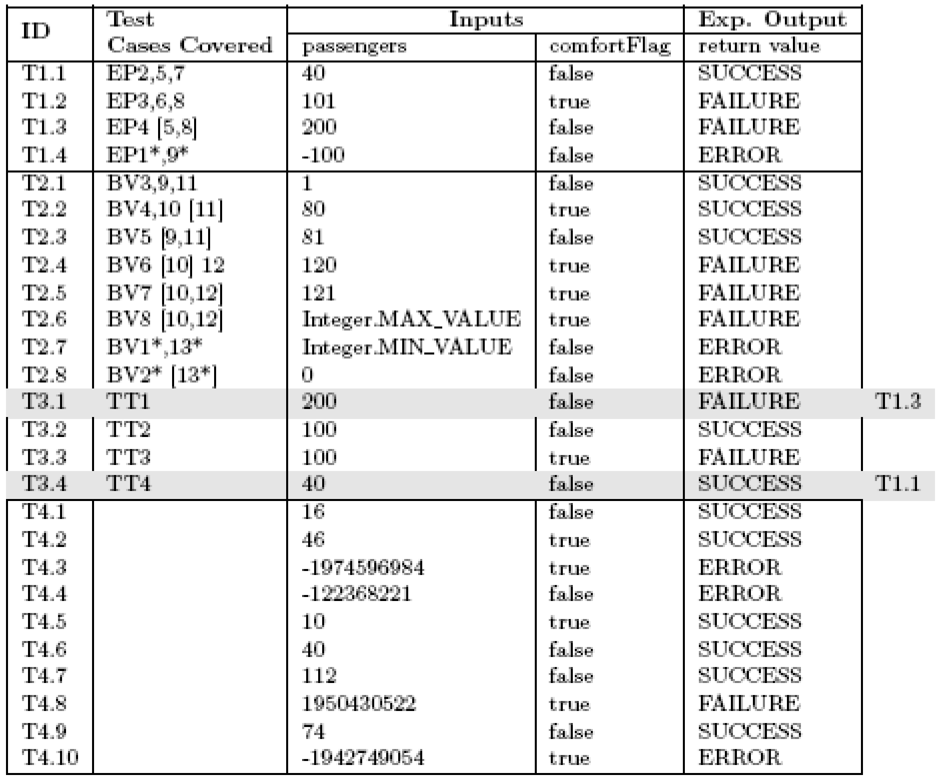
- Strengths
  - Intuition can frequently provide an accurate basis for finding faults.
  - The technique is very efficient, as it focuses on likely faults.
- Weaknesses
  - The technique relies on experienced testers-but they are not always available.
  - The ad-hoc nature of the approach means it is hard to ensure completeness of the testing.

Elimination of Duplicate Tests

- The use of 'standard' values simplifies the task of identifying identical tests.

- In cases where this has not been done, then further test elimination can be achieved by identifying equivalent tests, and making minor changes to the test data.

Elimination of Duplicate Tests



ID	Test Cases Covered	Inputs		Exp. Output	
		passengers	comfortFlag	return value	
T1.1	EP2,5,7	40	false	SUCCESS	
T1.2	EP3,6,8	101	true	FAILURE	
T1.3	EP4 [5,8]	200	false	FAILURE	
T1.4	EP1*,9*	-100	false	ERROR	
T2.1	BV3,9,11	1	false	SUCCESS	
T2.2	BV4,10 [11]	80	true	SUCCESS	
T2.3	BV5 [9,11]	81	false	SUCCESS	
T2.4	BV6 [10] 12	120	true	FAILURE	
T2.5	BV7 [10,12]	121	true	FAILURE	
T2.6	BV8 [10,12]	Integer.MAX_VALUE	true	FAILURE	
T2.7	BV1*,13*	Integer.MIN_VALUE	false	ERROR	
T2.8	BV2* [13*]	0	false	ERROR	
T3.1	TT1	200	false	FAILURE	T1.3
T3.2	TT2	100	false	SUCCESS	
T3.3	TT3	100	true	FAILURE	
T3.4	TT4	40	false	SUCCESS	T1.1
T4.1		16	false	SUCCESS	
T4.2		46	true	SUCCESS	
T4.3		-1974596984	true	ERROR	
T4.4		-122368221	false	ERROR	
T4.5		10	true	SUCCESS	
T4.6		40	false	SUCCESS	
T4.7		112	false	SUCCESS	
T4.8		1950430522	true	FAILURE	
T4.9		74	false	SUCCESS	
T4.10		-1942749054	true	ERROR	