Software Testing
Course’s Code: CSE 453
Black Box Testing

Black Box Testing

- Testing software against a specification of its external behavior without knowledge of internal implementation details
  - Can be applied to software "units" (e.g., classes) or to entire programs
  - External behavior is defined in API does, Functional specs, Requirements specs, etc.
- Because black box testing purposely disregards the program's control structure, attention is focused primarily on the information domain (Le., data that goes in, data that comes out)
- The Goal: Derive sets of input conditions (test cases) that fully exercise the external functionality

Black Box Testing

- Black box testing tends to find different k inds of errors than white box testing
    - Missing functions
    - Usability problems
    - Performance problems
    - Concurrency and timing errors Initialization and termination errors
    - Etc.
- Unlike white box testing, black box testing tends to be applied later in the development process

Equivalence partitioning

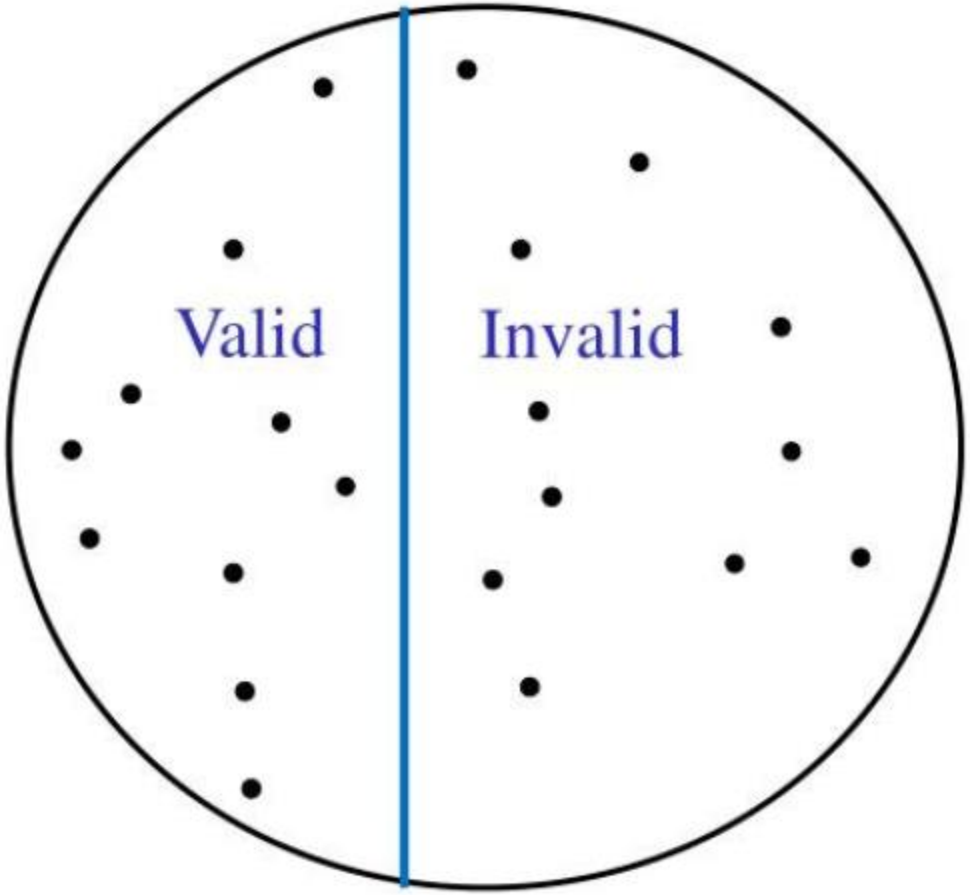
- Typically the universe of all possible test cases is so large that you cannot try them all
- You have to select a relatively small number of test cases to actually run
- Which test cases should you choose?
- Equivalence partitioning helps answer this question

Equivalence partitioning

- Partition the test cases into "equivalence classes"
- Each equivalence class contains a set of "equivalent" test cases
- Two test cases are considered to be equivalent if we expect the program to process them both in the same way (i.e., follow the same path through the code)
- If you expect the program to process two test cases in the same way, only test one of them, thus reducing the number of test cases you have to run

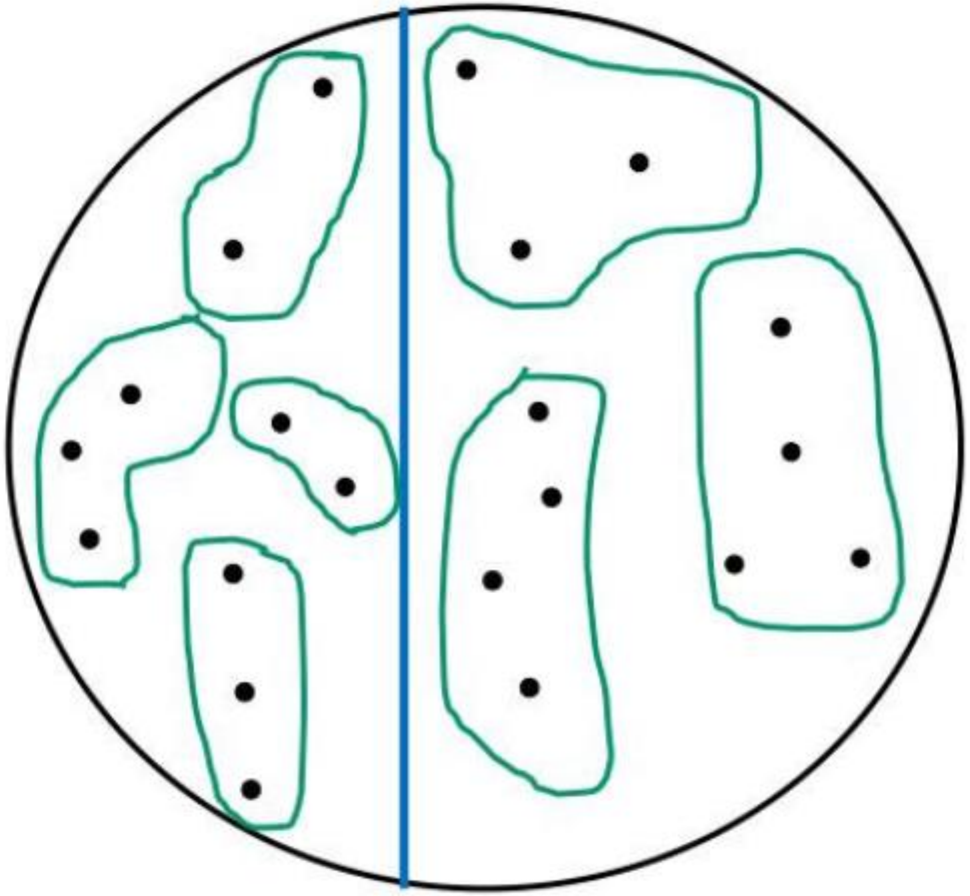
Equivalence partitioning

- First-level partitioning: Valid vs. Invalid test cases



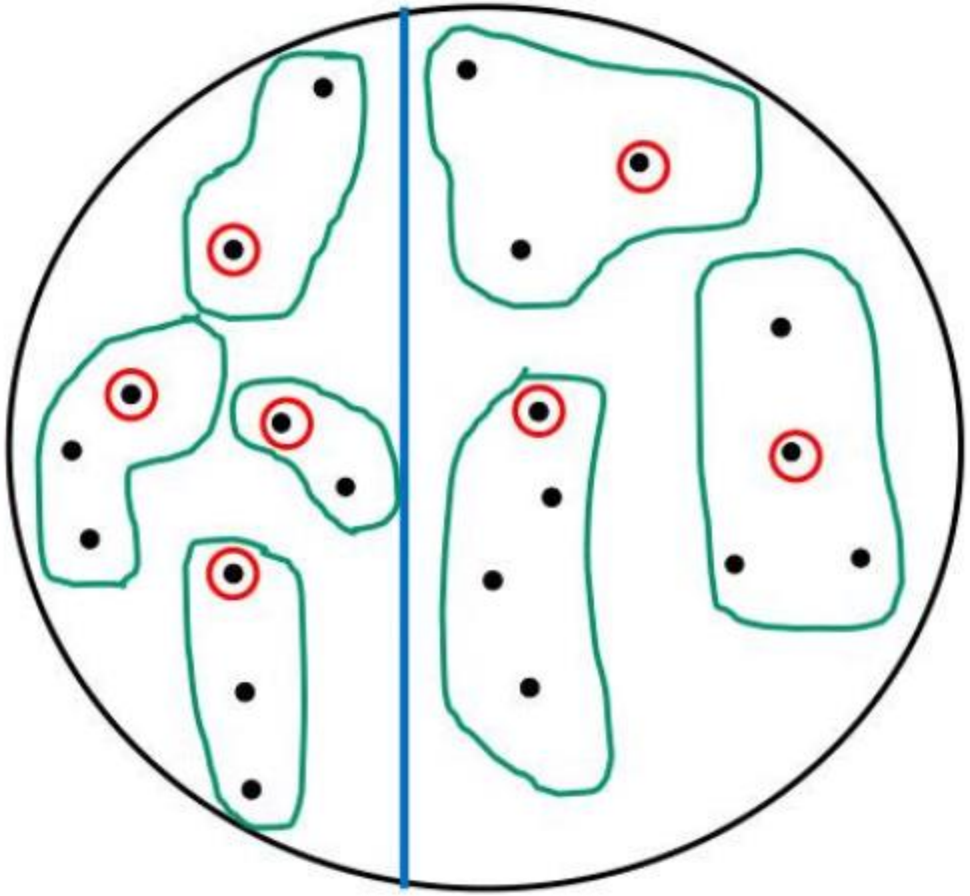
Equivalence partitioning

- Partition valid and invalid test cases into equivalence classes



Equivalence partitioning

- Create a test case for at least one value from each equivalence class



Equivalence partitioning

- When designing test cases, you may use different definitions of "equivalence", each of which will partition the test case space differently
  - Example: `int Add(n1, n2, n3, …)`
    - Equivalence Definition 1: partition test cases by the number of inputs (1, 2, 3, etc.)
    - Equivalence Definition 2: partition test cases by the number signs they contain (positive, negative, both)
    - Equivalence Definition 3: partition test cases by the magnitude of operands (large numbers, small numbers, both)
    - Etc.

Equivalence partitioning

- When designing test cases, you may use different definitions of “equivalence”, each of which will partition the test case space differently
  - Example: `string Fetch(URL)`
    - Equivalence Definition 1: partition test cases by URL protocol (“http”, “https”, “ftp”, “file”, etc.)
    - Equivalence Definition 2: partition test cases by type of file being retrieved (HTML, GIF, JPEG, Plain Text, etc.)
    - Equivalence Definition 3: partition test cases by length of URL (very short, short, medium, long, very long, etc.)
    - Etc.

Equivalence partitioning

Input	Valid Equivalence Classes	Invalid Equivalence Classes
A integer N such that: -99 <= N <= 99	?	?
Phone Number Area code: [200, 999] Prefix: (200, 999] Suffix: Any 4 digits	?	?

Equivalence partitioning

Input	Valid Equivalence Classes	Invalid Equivalence Classes
A integer N such that: -99 <= N <= 99	[-99, -10] [-9, -1] 0 [1, 9] [10, 99]	?
Phone Number Area code: [200, 999] Prefix: (200, 999] Suffix: Any 4 digits	?	?

Equivalence partitioning

Input	Valid Equivalence Classes	Invalid Equivalence Classes
A integer N such that: -99 <= N <= 99	[-99, -10] [-9, -1] 0 [1, 9] [10, 99]	< -99 > 99 Malformed numbers {12-, 1-2-3, …} Non-numeric strings {junk, 1E2, $13} Empty value
Phone Number Area code: [200, 999] Prefix: (200, 999] Suffix: Any 4 digits	?	?

Equivalence partitioning

Input	Valid Equivalence Classes	Invalid Equivalence Classes
A integer N such that: -99 <= N <= 99	[-99, -10] [-9, -1] 0 [1, 9] [10, 99]	< -99 > 99 Malformed numbers {12-, 1-2-3, …} Non-numeric strings {junk, 1E2, $13} Empty value
Phone Number Area code: [200, 999] Prefix: (200, 999] Suffix: Any 4 digits	555-5555 (555)555-5555 555-555-5555 200 <= Area code <= 999 200 < Prefix <= 999	?

Equivalence partitioning

Input	Valid Equivalence Classes	Invalid Equivalence Classes
A integer N such that: -99 <= N <= 99	[-99, -10] [-9, -1] 0 [1, 9] [10, 99]	< -99 > 99 Malformed numbers {12-, 1-2-3, …} Non-numeric strings {junk, 1E2, $13} Empty value
Phone Number Area code: [200, 999] Prefix: (200, 999] Suffix: Any 4 digits	555-5555 (555)555-5555 555-555-5555 200 <= Area code <= 999 200 < Prefix <= 999	Invalid format 5555555, (555)(555)5555, etc. Area code < 200 or > 999 Area code with non-numeric characters *Similar for Prefix and Suffix*

Equivalence partitioning

Example with JUnit:

We will test the behavior of withdrawing money from a bank account. An amount from $1 to $500 is considered valid. Otherwise, any value greater than $500 and less than $1 is considered invalid.

It is impossible to test all values because it is a waste of time, and the test case number will exceed 500. Here is how we use Equivalence Partitioning techniques to maximize test coverage:

Equivalence partitioning

Example with JUnit:

Value	Expected Outcome	Group	Representative Value
< 1	Invalid	Partition 1	-1
1 to 500	Valid	Partition 2	200
> 500	Invalid	Partition 3	550

By applying Equivalence Partitioning, we divide possible inputs into groups or levels (3 groups as above) in which the system will respond as the same. Then we select one representative value from each partition to do testing.

Equivalence partitioning

Example with JUnit:

```
@Test    public void testWithdrawValidAmount() {
        String output = captureOutput(() -> BankAccount.withdraw(-1));
        assertEquals(“Invalid", output);
}
@Test    public void testWithdrawValidUpperBoundary() {
        String output = captureOutput(() -> BankAccount.withdraw(200));
        assertEquals("Valid", output);
}
 @Test    public void testWithdrawValidLowerBoundary() {
        String output = captureOutput(() -> BankAccount.withdraw(550));
        assertEquals(" Invalid", output);
}
```

Boundary Value Analysis

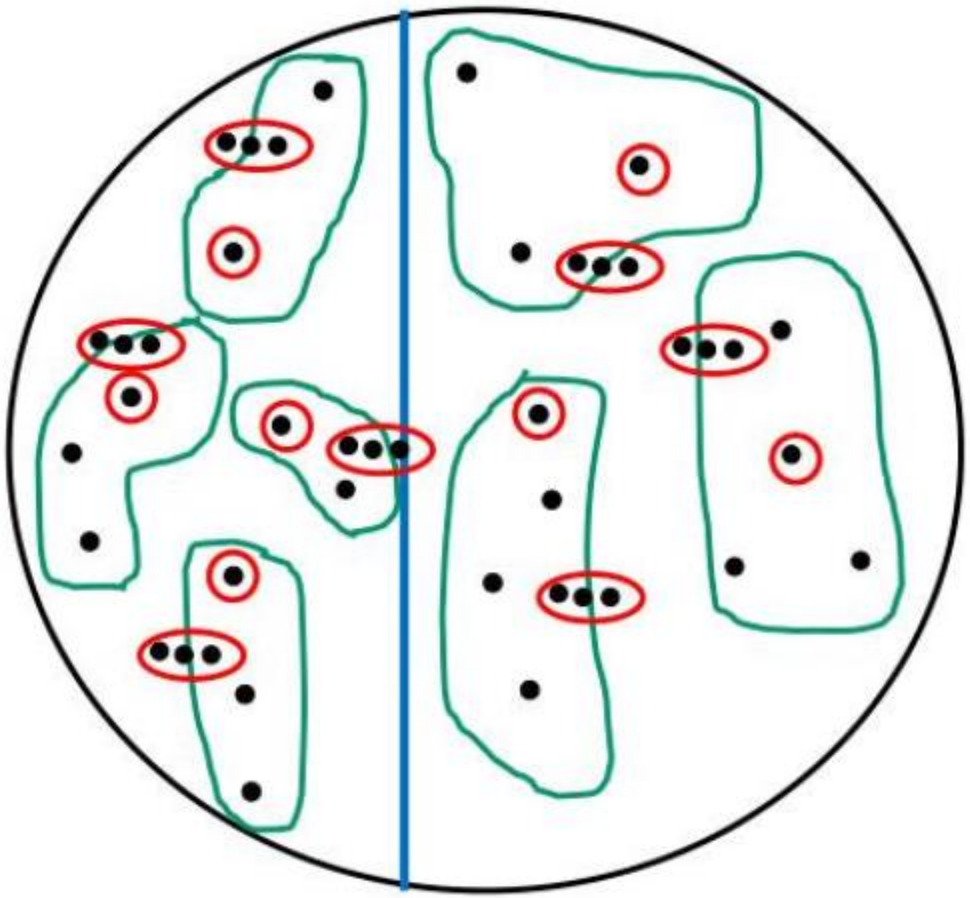
- When choosing values from an equivalence class to test, use the values that are most likely to cause the program to fail

- Errors tend to occur at the boundaries of equivalence classes rather than at the "center"
  - If (200 < areaCode && areaCode < 999) { // valid area code }
  - Wrong!
  - If (200 <= areaCode && areaCode <= 999) { // valid area code }
  - Testing area codes 200 and 999 would catch this error, but a center value like 770 would not

- In addition to testing center values, we should also test boundary values
  - Right on a boundary
  - Very close to a boundary on either side

Boundary Value Analysis

- Create test cases to test boundaries of equivalence classes



Boundary Value Analysis

Input	Boundary Cases
A number N such that: -99 <= N <= 99	?
Phone Number Area code: [200, 999] Prefix: (200, 999] Suffix: Any 4 digits	?

Boundary Value Analysis

Input	Boundary Cases
A number N such that: -99 <= N <= 99	?
Phone Number Area code: [200, 999] Prefix: (200, 999] Suffix: Any 4 digits	?

Boundary Value Analysis

Input	Boundary Cases
A number N such that: -99 <= N <= 99	-100, -99, -98 -10, -9 -1, 0, 1 9, 10 98, 99, 100
Phone Number Area code: [200, 999] Prefix: (200, 999] Suffix: Any 4 digits	?

Boundary Value Analysis

Input	Boundary Cases
A number N such that: -99 <= N <= 99	-100, -99, -98 -10, -9 -1, 0, 1 9, 10 98, 99, 100
Phone Number Area code: [200, 999] Prefix: (200, 999] Suffix: Any 4 digits	Area code: 199, 200, 201 Area code: 998, 999, 1000 Prefix: 200, 199, 198 Prefix: 998, 999, 1000 Suffix: 3 digits, 5 digits

Boundary Value Analysis

Example:

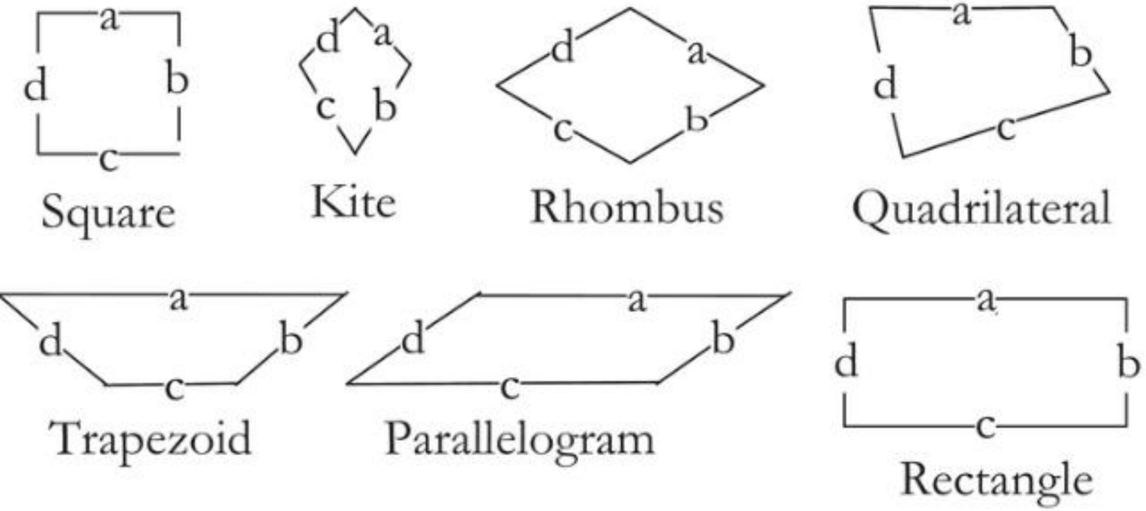
The Quadrilateral Program It accepts four integers, side1, side2, side3 and side4, as input. These are taken to be sides of a four-sided figure and they must satisfy the following conditions:

1. c1. $1 \leq a \leq 200$ (top)
2. c2. $1 \leq b \leq 200$ (left side)
3. c3. $1 \leq c \leq 200$ (bottom)
4. c4. $1 \leq d \leq 200$ (right side)

The output of the program is the type of quadrilateral determined by the four sides: Square, Rectangle, Trapezoid, Kite or General. (Since the problem statement only has information about lengths of the four sides, a square cannot be distinguished from a rhombus, similarly, a parallelogram cannot be distinguished from a rectangle.)

Boundary Value Analysis

Example:



1. A *square* has two pairs of parallel sides (a||c, b||d), and all sides are equal (a = b = c = d).
2. A *kite* has two pairs of equal sides, but no parallel sides (a = d, b = c).
3. A *trapezoid* has one pair of parallel sides (a||c) and one pair of equal sides (b = d).
4. A *rectangle* has two pairs of parallel sides (a||c, b||d) and two pairs of equal sides (a = c, b = d).
5. A *scalene quadrilateral* has four sides, none equal and none parallels (aka a trapezium).
6. Other options can create a *General* quadrilateral.

Boundary Value Analysis

Example:

Normal Boundary Value Analysis:

Test cases will be created by holding the value of all variables except one at their nominal values, and allowing the one variable assume **5 input variable values** : minimum value (**min**), just above the minimum (**min+1**), a nominal value (**nom**), just below their maximum (**max-1**), and maximum value (**max**).

1. Min: 1
2. Min+1: 2
3. Nom: 100
4. Max -1: 199
5. Max : 200

Boundary Value Analysis

Example:

Normal Boundary Value Analysis:

topSide	bottomSide	leftSide	rightSide	Expected
100	100	100	1	General
100	100	100	2	General
100	100	100	100	Square
100	100	100	199	General
100	100	100	200	General
100	100	1	100	General
100	100	2	100	General
100	100	199	100	General
100	100	200	100	General
....				

Boundary Value Analysis

Example:

Normal Boundary Value Analysis:

```
@DisplayName("Normal Boundary Value Test")
@ParameterizedTest(name="topSide= {0},bottomSide= {1},leftSide= {2},"
        + "rightSide={3},ExpectedResult= {4}")
@CsvSource({"100,100,100,1,General","100,100,100,2,General","100,100,100,100,Square"
    ,"100,100,100,199,General","100,100,100,200,General",
    "100,100,1,100,General","100,100,2,100,General",
    "100,100,199,100,General","100,100,200,100,General",
    "100,1,100,100,Trapezoid","100,2,100,100,Trapezoid",
    "100,199,100,100,Trapezoid","100,200,100,100,Trapezoid",
    "1,100,100,100,Trapezoid","2,100,100,100,Trapezoid",
    "199,100,100,100,Trapezoid","200,100,100,100,Trapezoid"})
void testQuadrilateral1(int topSide, int bottomSide,int leftSide,int rightSide,String result) {
    Q.setSide(topSide, bottomSide, leftSide, rightSide);
    assertEquals(result,Q.classify());
}
```

Boundary Value Analysis

Example:

Robust Boundary Value Analysis:

Test cases will be created by holding the value of all variables except one at their nominal values, and allowing the one variable assume **7 input variable values** : just bellow the minimum **(min-1)**, minimum value **(min),** just above the minimum **(min+1),** a nominal value **(nom),** just below their maximum **(max-1),** and maximum value **(max),** just above the maximum **(max+1).**

1. Min -1: 0
2. Min: 1
3. Min +1: 2
4. Nom: 100
5. Max -1: 199
6. Max : 200
7. Max +1 : 201

Boundary Value Analysis

Example:

Robust Boundary Value Analysis:

topSide	bottomSide	leftSide	rightSide	Expected
100	100	100	**1**	General
100	100	100	**0**	OUT_OF_RANGE
100	100	100	**2**	General
100	100	100	**100**	General
100	100	100	**199**	General
100	100	100	**200**	General
100	100	100	**201**	OUT_OF_RANGE
…………..				
201	100	100	100	OUT_OF_RANGE

Boundary Value Analysis

Example:

Robust Boundary Value Analysis:

```java
@DisplayName("Robust Boundary Value Test")
@ParameterizedTest(name="topSide= {0},bottomSide= {1},leftSide= {2},"
        + "rightSide={3},ExpectedResult= {4}")
@CsvSource({"100,100,100,1,General","100,100,100,0,OUT_OF_RANGE","100,100,100,2,General",
    "100,100,100,100,Square","100,100,100,199,General","100,100,100,200,General",
    "100,100,100,201,OUT_OF_RANGE",
    "100,100,0,100,OUT_OF_RANGE","100,100,1,100,General","100,100,2,100,General",
    "100,100,199,100,General","100,100,200,100,General","100,100,201,100,OUT_OF_RANGE",
    "100,0,100,100,OUT_OF_RANGE","100,1,100,100,Trapezoid","100,2,100,100,Trapezoid",
    "100,199,100,100,Trapezoid","100,200,100,100,Trapezoid","100,201,100,100,OUT_OF_RANGE",
    "0,100,100,100,OUT_OF_RANGE","1,100,100,100,Trapezoid","2,100,100,100,Trapezoid",
    "199,100,100,100,Trapezoid","200,100,100,100,Trapezoid","201,100,100,100,OUT_OF_RANGE"})
void testQuadrilateral2(int topSide, int bottomSide,int leftSide,int rightSide,String result) {
    Q.setSide(topSide, bottomSide, leftSide, rightSide);
    assertEquals(result,Q.classify());
}
```

Decision Table

- Specifications often contain business rules to define
  - functions of the system
  - conditions or decisions under which each function operates
- Individual conditions are simple, but the overall effects of these logical conditions can become quite complex
- As testers, we need to able to assure ourselves that every combinations of these conditions might occur has been tested
  - need to capture all decisions in a way that enables us to explore their combinations
  - Decision Table is the mechanism that is used to capture the logical decisions in a precise and compact way

Decision Table

- A decision table lists all the input conditions that can occur and all the actions that can arise from them
- The resulting decision tables are easy to understand, and can be validated by domain experts.
- Furthermore, testers can use decision tables for the systematic derivation of test cases,
    - in order to verify that the system under test correctly implements the required conditional logic.
- A decision table is structured into a table as rows
    - the conditions at the top of the table
    - possible actions at the bottom
    - business rules which involve some combinations of conditions to produce some combinations of actions are arranged across the top

Decision Table

- A decision table is structured into a table as rows
  - the conditions at the top of the table
  - possible actions at the bottom
  - each column represents a single business rule shows how input conditions are combined to produce actions

	Rule 1	Rule 2	Rule 3
Condition 1	T	F	T
Condition 2	T	T	T
Condition 3	T	dc	F
Action 1	Y	N	Y
Action 2	N	Y	Y

Decision Table

- Rule 1 requires all conditions to be true to generate action 1
- Rule 2 results in action 2 if condition 1 is false and condition 2 is true, but does not depends on condition 3
- Rule 3 requires conditions 1 and 2 to be true and condition 3 to be false
- Hyphen “dc” in decision table represents a “don’t care” entry.

	Rule 1	Rule 2	Rule 3
Condition 1	T	F	T
Condition 2	T	T	T
Condition 3	T	dc	F
Action 1	Y	N	Y
Action 2	N	Y	Y

Example-Phone Subscriptions

- A Dutch phone company, lets users click various options and then determines a price per month.

- The price per month is determined by two conditions. The conditions are:
  - whether the subscription is for national, which is cheaper, or international, more expensive, that’s on the first row
  - the second condition is whether the customer is willing to renew automatically, which would also be cheaper.
  - At the bottom in bold, we see the actual outcome or action, as determined by the conditions. In this case the outcomes are different monthly prices.

		Variants			
Conditions	International?	F	F	T	T
	Auto-renewal?	T	F	T	F
Action	**Price/month**	**10**	**15**	**30**	**32**

Example-Phone Subscriptions

- In this example, there are 4 rules
    - The first rule describes the cheapest subscription, 10 Euros per month, limited to national usage and with an obligation to automated renewal
    - The fourth column is the most expensive one, 32 Euros per month with usage across the world, and the possibility to cancel any moment.

		Variants			
Conditions	International?	F	F	T	T
	Auto-renewal?	T	F	T	F
Action	**Price/month**	**10**	**15**	**30**	**32**

Example-Phone Subscriptions

- As a slightly more complex example, consider an extra condition, shown in the third row.
  - If you as a potential buyer are a loyal customer already, you deserve a reduction.
  - Here, we assume a customer can either get loyalty reduction, or auto-renewal reduction, but not both.

	Rule 1	**Rule 2**	**Rule 3**	**Rule 4**	**Rule 5**	**Rule 6**
International?	F	F	F	T	T	T
Auto-renewal?	T	dc	F	T	dc	F
Loyal?	dc	T	F	dc	T	F
Price/month	10	10	15	30	30	32

Example-Phone Subscriptions

- The first column indicates that if we have auto-renewal already, T-value, whether the customer is loyal as well does not matter, DC-value.

- Likewise, if the customer is loyal, second column, picking auto-renewal makes no difference.

	Rule 1	**Rule 2**	**Rule 3**	**Rule 4**	**Rule 5**	**Rule 6**
International?	F	F	F	T	T	T
Auto-renewal?	T	dc	F	T	dc	F
Loyal?	dc	T	F	dc	T	F
Price/month	10	10	15	30	30	32

Example-Phone Subscriptions

- These don't care values are essentially an abbreviation for two separate columns, for the true and false case, which are identical except for the DC value.

- If we expand each DC-value, we get, in this table, 4 extra variants or columns.

- We can see that variants one and three are exactly the same, even though they were derived from expansions of different DC-cells, shown in orange.

- The table is in this case well designed, in the sense that these two variants share the same action, the reduced price is 10 Euros in both cases.

	Rule 1	**Rule 2**	**Rule 3**	**Rule 4**	**Rule 5**	**Rule 6**
International?	F	F	F	T	T	T
Auto-renewal?	T	dc	F	T	dc	F
Loyal?	dc	T	F	dc	T	F
Price/month	10	10	15	30	30	32

International?	F	F	F	F	F	T	T	T	T	T
Auto-renewal?	T	T	T	F	F	T	T	T	F	F
Loyal?	T	F	T	T	F	T	F	T	T	F
Price/month	**10**	**10**	**10**	**10**	**15**	**30**	**30**	**30**	**30**	**32**

Example-Phone Subscriptions

- If we omit the duplicate columns we arrive at the simpler table shown here. With three conditions and 8 variants it shows the maximum number of variants.

International?	F	F	F	F	F	T	T	T	T	T
Auto-renewal?	T	T	T	F	F	T	T	T	F	F
Loyal?	T	F	T	T	F	T	F	T	T	F
Price/month	**10**	**10**	**10**	**10**	**15**	**30**	**30**	**30**	**30**	**32**

Expanded and
De-Duplicated

International?	F	F	F	F	T	T	T	T
Auto-renewal?	T	T	F	F	T	T	F	F
Loyal?	T	F	T	F	T	F	T	F
Price/month	**10**	**10**	**10**	**15**	**30**	**30**	**30**	**32**

Example-Phone Subscriptions

- In general, if we have N conditions, rows, this leads to 2^N possible variants, columns.

- The mobile plan example given here is relatively simple, with just three conditions.

- Decision tables in practice can have many more conditions. For example, the actual mobile plan has several more conditions, bandwidth limits, phone minutes, type of previous subscription, etc. This then easily leads to at least six conditions and 2^6 = 64 variants.

Larger Decision Tables

Decision tables can have many conditions

In general: *N* conditions: 2^*N* variants

Omitted / non-specified variants?

Indicate what “default” behavior is.

Example-Phone Subscriptions: Testing Decision Table

- How to create test cases from a decision table?
  - Two ways
    - create one test case per variant (Column) listed in the table

    Or
    - MC/DC Coverage: Modified Condition / Decision coverage.

All explicit variants: 6

All possible variants: 2^3 = 8

International?	F	F	F	T	T	T
Auto-renewal?	T	dc	F	T	dc	F
Loyal?	dc	T	F	dc	F	F
Price/month	**10**	**10**	**15**	**30**	**30**	**32**

Each condition T/F: 2 cases (TTT, FFF)

All decisions / every unique outcome: 4

Each condition AND all decisions = (M)C/DC

Example-Phone Subscriptions: Testing Decision Table

- MC/DC demands the following: Every condition should yield both true and false, Every outcome or action should be taken at least once, Every condition should be shown to individually determine the decision outcome,

MC/DC: Modified Condition / Decision Coverage

1. **Conditions:** Each condition should be once true, once false
2. **Decisions:** Each action should be taken at least once
3. **Modified:** Each condition should individually determine the outcome

For each condition require two test cases that *only* differ in outcome and *that* condition

Example-Phone Subscriptions: Testing Decision Table

Finding an "MC/DC Cover"

- Expand decision table
- Pick variants with unique outcome
- Combine with others so that they differ in one condition only

	v1	*v2*	*v3*	*v4*	*v5*	*v6*	*v7*	*v8*
International?	F	F	F	F	T	T	T	T
Auto-renewal?	T	T	F	F	T	T	F	F
Loyal?	T	F	T	F	T	F	T	F
Price/month	**10**	**10**	**10**	**15**	**30**	**30**	**30**	**32**

Finding an "MC/DC Cover"

- Expand decision table
- Pick variants with unique outcome
- Combine with others so that they differ in one condition only

	v1	*v2*	*v3*	*v4*	*v5*	*v6*	*v7*	*v8*	*mc/dc*	*action*
International?	F	F	F	F	T	T	T	T	v4,v8	15 ,32
Auto-renewal?	T	T	F	F	T	T	F	F		
Loyal?	T	F	T	F	T	F	T	F		
Price/month	**10**	**10**	**10**	**15**	**30**	**30**	**30**	**32**		

Example-Phone Subscriptions: Testing Decision Table

Finding an “MC/DC Cover”

- Expand decision table
- Pick variants with unique outcome
- Combine with others so that they differ in one condition only

	v1	v2	v3	v4	v5	v6	v7	v8
International?	F	F	F	F	T	T	T	T
Auto-renewal?	T	T	F	F	T	T	F	F
Loyal?	T	F	T	F	T	F	T	F
Price/month	**10**	**10**	**10**	**15**	**30**	**30**	**30**	**32**

Finding an “MC/DC Cover”

- Expand decision table
- Pick variants with unique outcome
- Combine with others so that they differ in one condition only

	v1	v2	v3	v4	v5	v6	v7	v8	mc/dc	action
International?	F	F	F	F	T	T	T	T	v4,v8	15 ,32
Auto-renewal?	T	T	F	F	T	T	F	F	v4,v2	10
Loyal?	T	F	T	F	T	F	T	F		
Price/month	**10**	**10**	**10**	**15**	**30**	**30**	**30**	**32**		

Example-Phone Subscriptions: Testing Decision Table

Finding an “MC/DC Cover”

- Expand decision table
- Pick variants with unique outcome
- Combine with others so that they differ in one condition only

	v1	v2	v3	v4	v5	v6	v7	v8
International?	F	F	F	F	T	T	T	T
Auto-renewal?	T	T	F	F	T	T	F	F
Loyal?	T	F	T	F	T	F	T	F
Price/month	**10**	**10**	**10**	**15**	**30**	**30**	**30**	**32**

Finding an “MC/DC Cover”

- Expand decision table
- Pick variants with unique outcome
- Combine with others so that they differ in one condition only

	v1	v2	v3	v4	v5	v6	v7	v8	mc/dc	action
International?	F	F	F	F	T	T	T	T	v4,v8	15 ,32
Auto-renewal?	T	T	F	F	T	T	F	F	v4,v2	10
Loyal?	T	F	T	F	T	F	T	F	v8,v7	30
Price/month	**10**	**10**	**10**	**15**	**30**	**30**	**30**	**32**		

- 4 Test Cases are: V2,V4,V8 and V7.

Example-Phone Subscriptions: Testing Decision Table

MC/DC: N+1 Test Cases

For a table with N conditions and yes/no actions, N+1 test cases suffice to obtain an MC/DC cover

Implementation of Decision Table based Testing

```java
@Test
void internationalExpensive() {
    PhonePlan plan = new PhonePlan();

    plan.setInternational(true);
    plan.setAutoRenewal(false);
    plan.setLoyal(false);

    assertThat(plan.pricePerMonth())
            .isEqualTo(32);
}
```

Junit Parameterized Tests

```java
@ParameterizedTest
@CsvSource({
        "true,  false, false, 32",
        "true,  false, true,  30",
        "false, false, false, 15",
        "false, true,  false, 10"
})
void testPlan(boolean inter,
              boolean renew,
              boolean loyal,
              int expected) {
    PhonePlan plan = new PhonePlan();

    plan.setInternational(inter);
    plan.setAutoRenewal(renew);
    plan.setLoyal(loyal);

    assertThat(plan.pricePerMonth())
            .isEqualTo(expected);

}
```

Example 2 - Decision Table based Testing

- A supermarket has a loyalty schema that is offered to all customers.
- Loyalty cardholders enjoy the benefits of either additional discounts on all purchases or the acquisition of loyalty points
- Loyalty points can be converted into vouchers for the supermarket or to equivalent points in schemas run by partners
- Customers without a loyalty card receive an additional discount only if they spend more that $100 on any one visit to the store
- otherwise only the special offers offered to all customers apply

Example 2 - Decision Table based Testing

		RULE 1	RULE 2	RULE 3	RULE 4	RULE 5
Conditions	**Customer loyalty card?**	**T**	**T**	**T**	**F**	**F**
	Special Offer selected?	**T**	**F**	**F**	**T**	**F**
	Spend>$100	**dc**	**dc**	**dc**	**T**	**F**
Actions	**Additional Discount**	**F**	**F**	**T**	**T**	**F**
	Loyalty Points-Voucher	**F**	**T**	**F**	**F**	**F**
	Loyalty Points-Equivalent Points	**T**	**F**	**F**	**F**	**F**
	Special offer applied	**T**	**F**	**F**	**T**	**F**

- 5 Test Cases are: Rule 1, Rule 2, Rule 3, Rule 4, Rule 5. (Without applying MC/DC)

Example 3 - Decision Table based Testing

- The triangle program accepts three integers, a, b, and c, as input
- These are taken to be the sides of a triangle
- The integers a, b, and c must satisfy the following conditions:

C1: $a<b+c$
C2: $b<a+c$
C3: $c<a+b$

- The output of the program may be: Equilateral, Isosceles, Scalene, Not-a-triangle and Impossible

Example 3- Decision Table based Testing

		R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11
Conditions	**a<b+c?**	F	T	T	T	T	T	T	T	T	T	T
	b<a+c?	-	F	T	T	T	T	T	T	T	T	T
	c<a+b?	-	-	F	T	T	T	T	T	T	T	T
	a=c?	-	-	-	T	T	T	F	T	F	F	F
	b=c?	-	-	-	T	T	F	T	F	T	F	F
	a=b?	-	-	-	T	F	T	T	F	F	T	F
Actions	**Not a triangle**	T	T	T								
	Scalene											T
	Isosceles								T	T	T	
	Equilateral				T							
	Impossible					T	T	T				

- 11 Test Cases are: R1-R11. (Without applying MC/DC)

Example 4- Decision Table Based Testing

- A mutual insurance company has decided to float its shares on the stock exchange and is offering its members rewards for their past custom at the time of flotation

- Anyone with a current policy will benefit provided it is a ‘with-profits’ policy and they have held it since 2001

- Those who meet these criteria can opt for either a cash payment or an allocation of shares in the new company

- Those who have held a qualifying policy for less than the required time will be eligible for a cash payment but not for shares.

Example 4- Decision Table Based Testing

		Rule 1	Rule 2	Rule 3	Rule 4
Conditions	**Current policy holder**	Y	Y	Y	N
	Policy holder since 2001	N	Y	N	dc
	‘With-profits’ policy	Y	Y	N	dc
Actions	**Eligible for cash payment**	Y	Y	N	N
	Eligible for share allocations	N	Y	N	N

- 4 Test Cases are: RULE1-RULE4