Event Application Testing

Event.calculatePrice()

Input Space Partitioning Testing for the method provided

Method signature:

calculatePrice(int numTickets, int ticketsBookedSoFar, LocalDateTime bookingTime)

Characteristic definition:

C1: numTickets validity

- C1A: numTickets is null
- C1B: numTickets < 0
- C1C: numTickets > 0
- C1C: numTickets = 0

C2: ticketsBookedSoFar validity

- C2A: ticketsBookedSoFar is null
- C2B: ticketsBookedSoFar < 0
- C2C: ticketsBookedSoFar > 0
- C2D: ticketsBookedSoFar = 0

C3: bookingTime validity

- C3A: bookingTime is null
- C3B: bookingTime >= event.endTime
- C3C: event.startTime(minus 30days) < bookingTime < event.startTime
- C3D: bookingTime = event.startTime
- C3E: bookingTime < event.startTime(minus 30days)
- C3F: event.startTime < bookingTime < event.endTime

C4: maxTickets (object state)

- C4A: maxTickets = 0
- C4B: maxTickets > 0
- C4C: maxTickets < 0

C5: endTime (object state)

- C5A: this.endTime < now
- C5B: this.endTime > now
- C5C: this.endTime = now

C6: startTime (object state)

- C6A: this.startTime < now
- C6B: this.startTime = now
- C6C: this.startTime > now

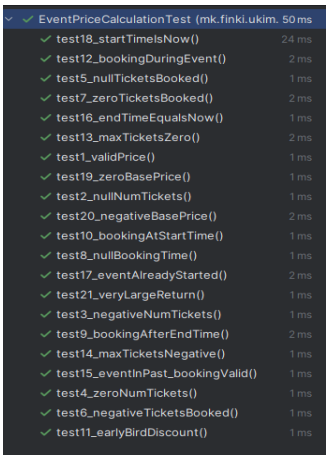
C7: return value

- C7A: returns value = 0
- C7B: returns value < 0
- C7C: returns value >= max.double
- C7D: returns 0 < value < max.double

There it is the ISP table where I used Base Coverage to find the Expected Outputs.

	C1	C2	C3	C4	C5	C6	C7	Expected
1	C1C	C2C	C3C	C4B	C5B	C6C	C7D	Valid price
2	**C1A**	C2C	C3C	C4B	C5B	C6C	C7D	IllegalArgumentException
3	**C1B**	C2C	C3C	C4B	C5B	C6C	C7D	IllegalArgumentException
4	**C1D**	C2C	C3C	C4B	C5B	C6C	C7D	IllegalArgumentException
5	C1C	**C2A**	C3C	C4B	C5B	C6C	C7D	IllegalArgumentException
6	C1C	**C2B**	C3C	C4B	C5B	C6C	C7D	IllegalArgumentException
7	C1C	**C2D**	C3C	C4B	C5B	C6C	C7D	Valid price
8	C1C	C2C	**C3A**	C4B	C5B	C6C	C7D	IllegalArgumentException
9	C1C	C2C	**C3B**	C4B	C5B	C6C	C7D	InvalidDateForBookingException
10	C1C	C2C	**C3D**	C4B	C5B	C6C	C7D	InvalidDateForBookingException
11	C1C	C2C	**C3E**	C4B	C5B	C6C	C7D	Valid price with 10% discount
12	C1C	C2C	**C3F**	C4B	C5B	C6C	C7D	InvalidDateForBookingException
13	C1C	C2C	C3C	**C4A**	C5B	C6C	C7D	ArithmeticException
14	C1C	C2C	C3C	**C4C**	C5B	C6C	C7D	IllegalArgumentException
15	C1C	C2C	C3C	C4B	**C5A**	C6C	C7D	Valid price
16	C1C	C2C	C3C	C4B	**C5C**	C6C	C7D	Valid price
17	C1C	C2C	C3C	C4B	C5B	**C6A**	C7D	InvalidDateForBookingException
18	C1C	C2C	C3C	C4B	C5B	**C6B**	C7D	InvalidDateForBookingException
19	C1C	C2C	C3C	C4B	C5B	C6C	**C7A**	Valid price = 0
20	C1C	C2C	C3C	C4B	C5B	C6C	**C7B**	Valid price < 0 (should not happen)
21	C1C	C2C	C3C	C4B	C5B	C6C	**C7C**	Valid very large price (check for overflow)

Tests Made and Runned



Graph Coverage on the calculatePrice(params) function:

Nodes:

1 2

2 3

2 4

4 5

4 6

6 7

6 8

8 9

8 10

10 11

10 12

12 13

12 14

14 15

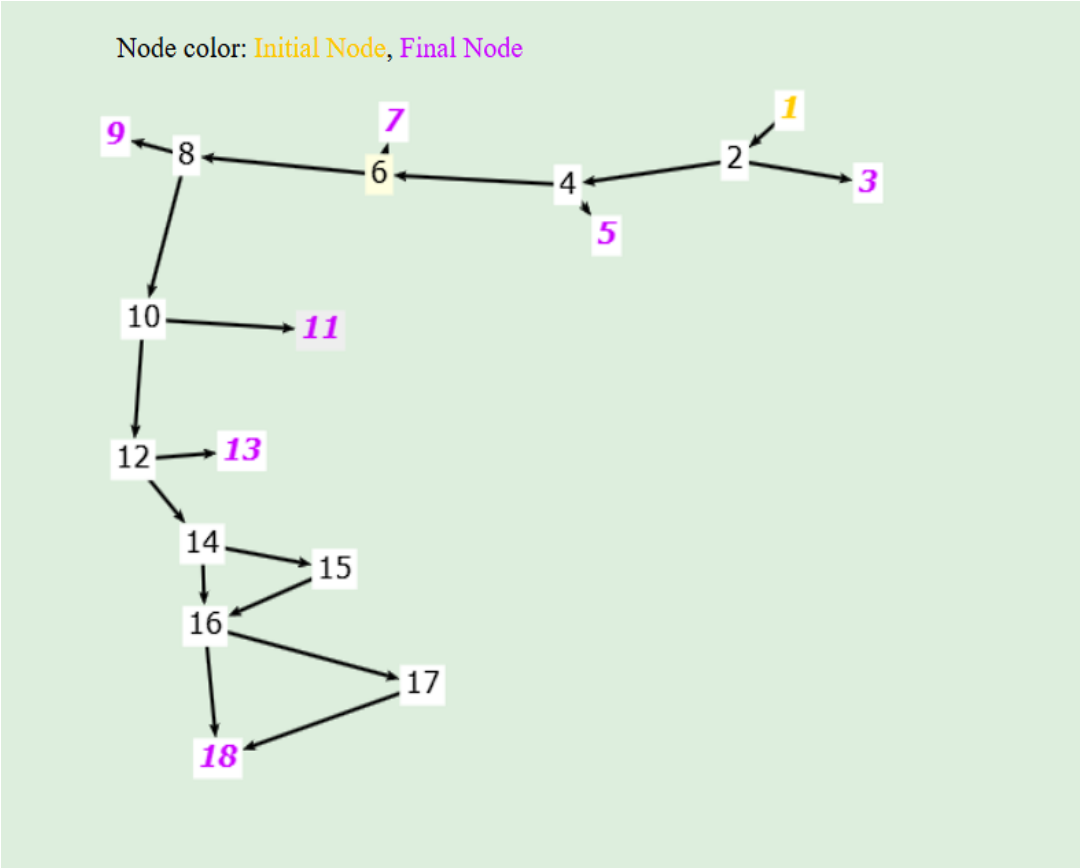
14 16

15 16

16 17

16 18

17 18



Initial node is 1, Ending nodes are 3 5 7 9 11 13 18

Now we did the Prime Paths to find the Test Requirements:

10 requirements are needed for Prime Paths
1. [1,2,4,6,8,10,12,14,15,16,17,18]
2. [1,2,4,6,8,10,12,14,15,16,18]
3. [1,2,4,6,8,10,12,14,16,17,18]
4. [1,2,4,6,8,10,12,14,16,18]
5. [1,2,4,6,8,10,12,13]
6. [1,2,4,6,8,10,11]
7. [1,2,4,6,8,9]
8. [1,2,4,6,7]
9. [1,2,4,5]
10. [1,2,3]

LoyaltyUtils.getLoyaltyDiscount()

Input Space Partitioning

Method definition:

public static String getLoyaltyLevel(int bookingCount)

Characteristics:

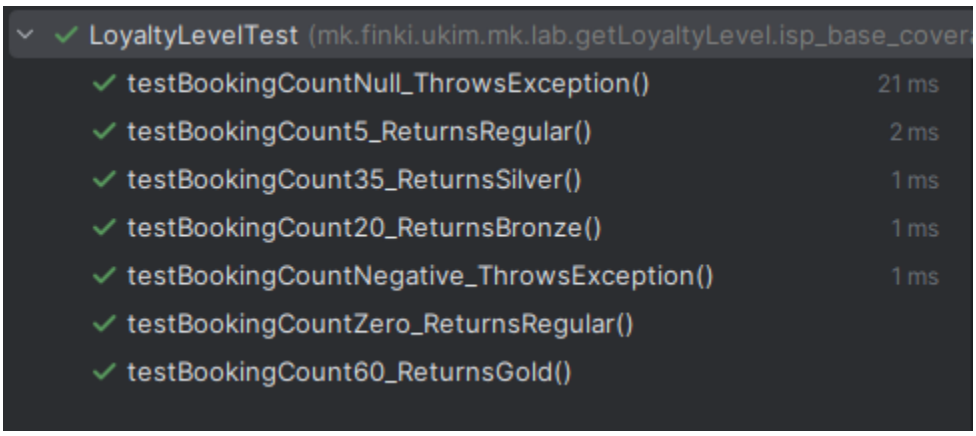
C1: bookingCount value

- C1A: bookingCount < 0
- C1B: bookingCount > 0
- C1C: bookingCount = 0
- C1D: bookingCount = null

C2: return value

- C2A: GOLD
- C2B: SILVER
- C2C: BRONZE
- C2D: REGULAR

	C1	C2	EXPECTED
1	C1B	C2A	T(returns level)
2	C1A	C2A	T (returns exc.)
3	C1C	C2A	T (returns Regular)
4	C1D	C2A	T (returns exc.)
5	C1B	C2B	T (returns Silver)
6	C1B	C2C	T (returns Bronze)
7	C1B	C2D	T (returns Regular)

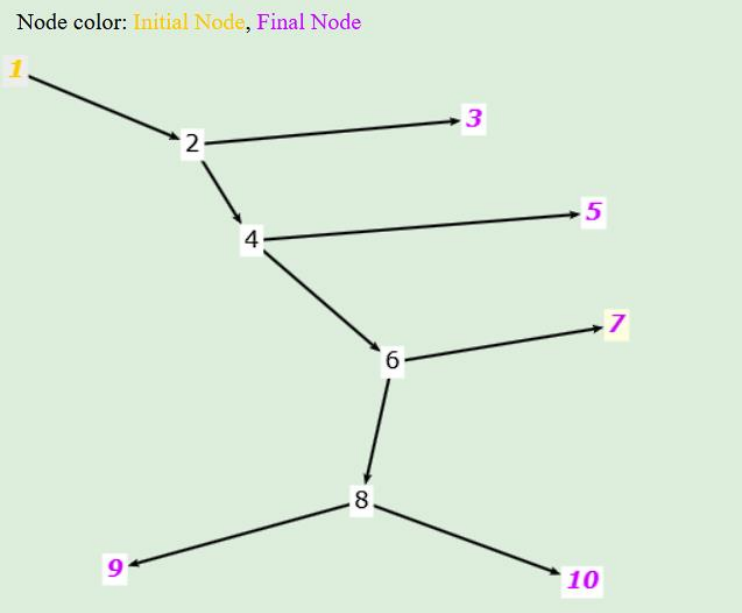


Logical Coverage is not that good to be made on this function, because there is no logical clause where we can test it.

Graph Coverage:

Graph nodes:
1 2
2 3
2 4
4 5
4 6
6 7
6 8
8 9
8 10

Initial is 1, terminals are 3, 5, 7, 9, 10



Using Edge-Pair we got the following TR:

Test Paths	Test Requirements that are toured by test paths directly
[1,2,3]	[1,2,3]
[1,2,4,5]	[1,2,4], [2,4,5]
[1,2,4,6,7]	[1,2,4], [2,4,6], [4,6,7]
[1,2,4,6,8,10]	[1,2,4], [2,4,6], [4,6,8], [6,8,10]
[1,2,4,6,8,9]	[1,2,4], [2,4,6], [4,6,8], [6,8,9]

DU- Path Coverage from bookingCount (starting in 1, and ending in 2,4,6,8)

Variable	All DU Path Coverage
bookingCount	[1,2,3] [1,2,4,5] [1,2,4,6,7] [1,2,4,6,8,9]

EventBookingService.bookEvent()

Input Space Partitioning with Base Choice Coverage

Method Signature:

bookEvent(String attendeeName, String attendeeAddress, Long eventId, Long numTickets)

Characteristics:

C1: atendeeName is ___

- C1A: atendeeName is null
- C1B: atendeeName is empty
- C1C: atendeeName is valid string

C2: atendeeAddress is ___

- C2A: atendeeAddress is null
- C2B: atende atendeeAddress is empty
- C2C: atendeeAddress is valid string

C3: eventId is ___

- C3A: eventId is null
- C3B: min.Long <= eventId <= max.Long and exists in DB
- C3C: min.Long <= eventId <= max.Long and does not exist in DB

C4: numTickets is ___

- C4A: numTickets is null
- C4B: min.Long <= numTickets <= max.Long

C5: Return Behavior

- C5A: Valid booking is created and returned successfully
- C5B: Exception is thrown because event is not found (RuntimeException)
- C5C: InvalidArgumentException
- C5D: Booking is created, but total price is 0.0 (if logic allows this edge case)

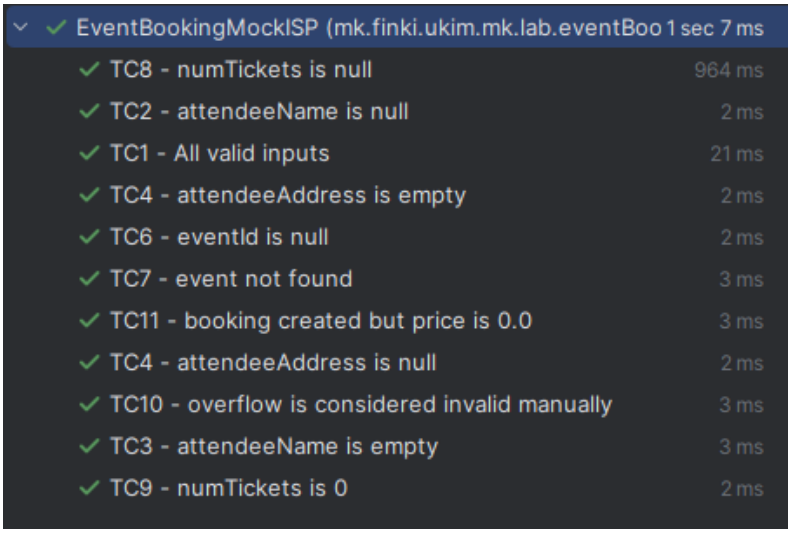
Here it is the table for base choice coverage

	C1	C2	C3	C4	C5	VALID?
1	C1C	C2C	C3B	C4B	C5A	valid
2	C1A	C2C	C3B	C4B	C5C	valid
3	C1B	C2C	C3B	C4B	C5C	valid
4	C1C	C2B	C3B	C4B	C5C	valid
5	C1C	C2A	C3B	C4B	C5C	valid
6	C1C	C2C	C3A	C4B	C5C	valid
7	C1C	C2C	C3C	C4B	C5B	valid
8	C1C	C2C	C3B	C4A	C5C	valid
9	C1C	C2C	C3B	C4B	C5B	valid,but same as 7
10	C1C	C2C	C3B	C4B	C5C	impossible
11	C1C	C2C	C3B	C4B	C5D	valid

Here it is an instance of a Mock, where we use hardcoded behavior of the repositories so we can test the method with ISP (Base Choice Coverage)

```
@ExtendWith(MockitoExtension.class)
class EventBookingMockISP {

    @Mock  4 usages
    private EventRepository eventRepository;

    @Mock  5 usages
    private EventBookingRepository eventBookingRepository;

    @InjectMocks  11 usages
    private EventBookingServiceImplementation eventBookingService;

    private Event event;  3 usages

    @BeforeEach
    void setup() {
        event = new Event( basePrice: 100.0,  maxTickets: 100, LocalDateTime.now().plusDays(10), LocalDateTime.now().plusDays(10).plusHours(3));
    }
```

Tests created:



Mockito Behavior

Also we introduced Mockito Behavior Testin when we were testing the functionality of the method, its cases and returns. We mocked the behaviors of the Service and Repository layers, allowing us to simulate specific scenarios and control the return values.

We used the following methods:

- verify(...) – a method that is checking interactions with your mocks
- when(...).then(...) – this method is setting behavior to the mocks

EventService.isEventForAdultsOnly()

Logic Coverage applied to the method

Method signature: public boolean isEventForAdultsOnly(Event event)

Predicate to test: (isBigEvent || isLocationOverused) && !isBadRated && isAtNight

Clauses:

a - isBigEvent

b - isLocationOverused

c - isBadRated

d - isAtNight

Predicate written with clauses: (a | b) & !c & d

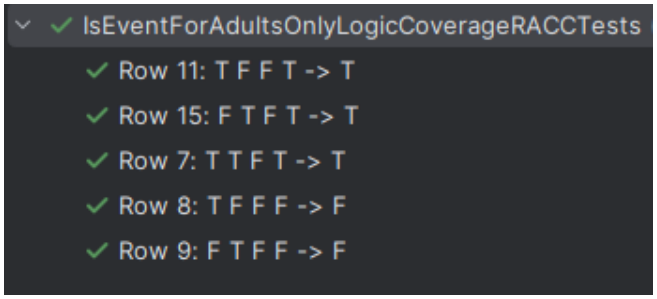
row index	a	b	c	d	P	Pa	Pb	Pc	Pd
1	T	T	T	T	F			T	
2	T	T	T	F	F				
3	T	T	F	T	T			T	T
4	T	T	F	F	F				T
5	T	F	T	T	F			T	
6	T	F	T	F	F				
7	T	F	F	T	T	T		T	T
8	T	F	F	F	F				T
9	F	T	T	T	F			T	
10	F	T	T	F	F				
11	F	T	F	T	T		T	T	T
12	F	T	F	F	F				T
13	F	F	T	T	F				
14	F	F	T	F	F				
15	F	F	F	T	F	T	T		
16	F	F	F	F	F				

We are going to make tests, using all 3 possible coverages:

- RACC

Major Clause	Set of possible tests
a	(7,15)
b	(11,15)
c	(1,3), (5,7), (9,11)
d	(3,4), (7,8), (11,12)

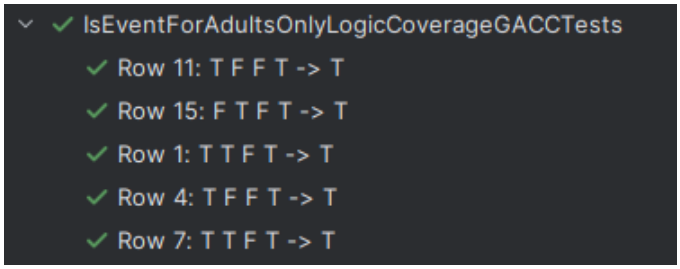
We get 7,8,9,11,15 rows as test cases rows



- GACC

Major Clause	Set of possible tests
a	(7,15)
b	(11,15)
c	(1,3), (1,7), (1,11), (5,3), (5,7), (5,11), (9,3), (9,7), (9,11)
d	(3,4), (3,8), (3,12), (7,4), (7,8), (7,12), (11,4), (11,8), (11,12)

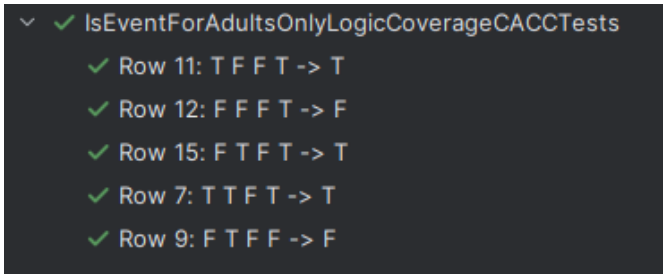
We get 1,4,7,11,15 rows as test cases rows



- CACC

Major Clause	Set of possible tests
a	(7,15)
b	(11,15)
c	(1,3), (1,7), (1,11), (5,3), (5,7), (5,11), (9,3), (9,7), (9,11)
d	(3,4), (3,8), (3,12), (7,4), (7,8), (7,12), (11,4), (11,8), (11,12)

We get 7,9,11,12,15 rows as test cases rows



This parameterized test class demonstrates an excellent approach to testing complex business logic. The key insight is that instead of writing 10 separate test methods that would largely duplicate code, we created it as data-driven testing structure that:

1. Encapsulates test scenarios in a custom TestCase class
2. Provides easy way to test with different combinations of inputs
3. Maintains a single test method that handles all scenarios consistently

This pattern is especially useful when testing methods with multiple conditional branches, as it helps achieve comprehensive path coverage while avoiding code duplication and making the expected behavior transparent to anyone reading the tests.

EventService.getDynamicPricesForAllEvents()

Graph Coverage

Graph Nodes:

1 2

2 10

2 3

3 4

3 6

6 8

6 7

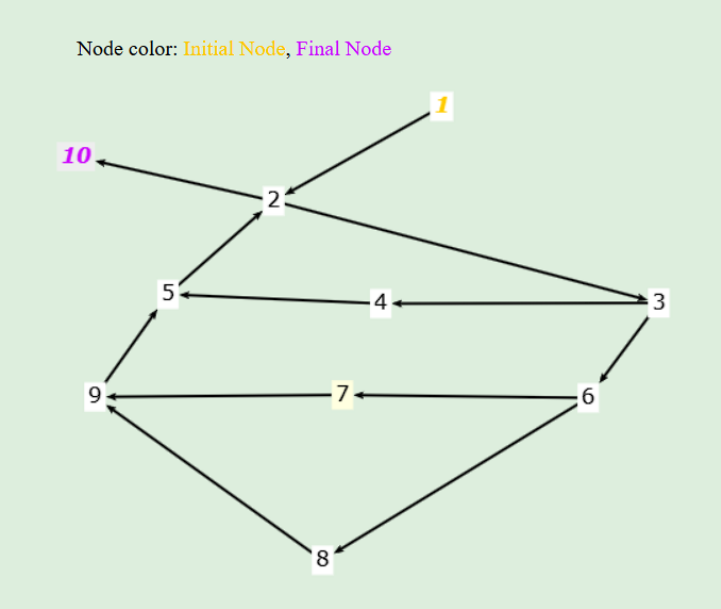
8 9

7 9

4 5

9 5

5 2



Initial node is 1, and terminal node is 10

Now we are going to do Prime Paths and we get around 29 TR:

29 requirements are needed for Prime Paths
1. [3,6,7,9,5,2,3]
2. [3,6,7,9,5,2,10]
3. [6,8,9,5,2,3,4]
4. [6,7,9,5,2,3,4]
5. [6,8,9,5,2,3,6]
6. [3,6,8,9,5,2,3]
7. [1,2,3,6,7,9,5]
8. [1,2,3,6,8,9,5]
9. [2,3,6,8,9,5,2]
10. [3,6,8,9,5,2,10]
11. [2,3,6,7,9,5,2]
12. [6,7,9,5,2,3,6]
13. [9,5,2,3,6,8,9]
14. [4,5,2,3,6,7,9]
15. [9,5,2,3,6,7,9]
16. [5,2,3,6,7,9,5]
17. [5,2,3,6,8,9,5]
18. [4,5,2,3,6,8,9]
19. [8,9,5,2,3,6,7]
20. [8,9,5,2,3,6,8]
21. [7,9,5,2,3,6,8]
22. [7,9,5,2,3,6,7]
23. [3,4,5,2,3]
24. [3,4,5,2,10]
25. [1,2,3,4,5]
26. [2,3,4,5,2]
27. [5,2,3,4,5]
28. [4,5,2,3,4]
29. [1,2,10]

Now we are going to write the test cases:

Test Cases for Prime Path Coverage

TC1: Valid Event – Not Sold Out

Test Path: 1 → 2 → 3 → 6 → 8 → 9 → 5 → 2 → 10

Covers TR:

[1,2,3,6,8,9,5,2,10]

[3,6,8,9,5,2,10]

TC2: Valid Event – Sold Out

Test Path: 1 → 2 → 3 → 6 → 7 → 9 → 5 → 2 → 10

Covers TR:

[3,6,7,9,5,2,10]

[6,7,9,5,2,10]

TC3: Null or Malformed Event

Test Path: 1 → 2 → 3 → 4 → 5 → 2 → 3 → 6 → 8 → 9 → 5 → 2 → 10

Covers TR:

[3,4,5,2,3,6,8]

[4,5,2,3,6,8,9]

[1,2,3,4,5,2,3,6,8]

TC4: Multiple Events – Sold Out and Not Sold Out

Test Path: 1 → 2 → 3 → 6 → 7 → 9 → 5 → 2 → 3 → 6 → 8 → 9 → 5 → 2 → 10

Covers TR:

[6,7,9,5,2,3,6,8]

[7,9,5,2,3,6,8]

[5,2,3,6,7,9,5,2]

[6,8,9,5,2,3,6]

TC5: Null + Sold Out + Valid

Test Path: 1 → 2 → 3 → 4 → 5 → 2 → 3 → 6 → 7 → 9 → 5 → 2 → 3 → 6 → 8 → 9 → 5 → 2 → 10

Covers TR:

[4,5,2,3,6,7,9]

[4,5,2,3,6,7,9,5,2]

[8,9,5,2,3,6,7]

[9,5,2,3,6,7,9]

TC6: All Types – Edge & Backtrack Paths

Test Path: 1 → 2 → 3 → 4 → 5 → 2 → 3 → 6 → 7 → 9 → 5 → 2 → 3 → 4 → 5 → 2 → 10

Covers TR:

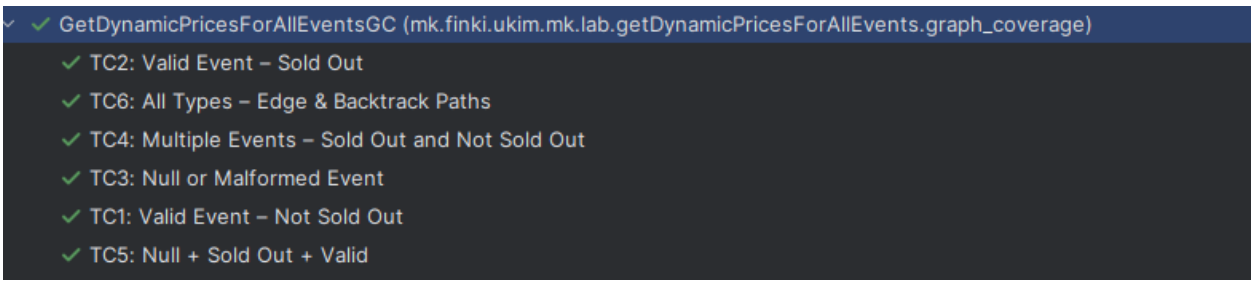
[3,4,5,2,10]

[2,3,4,5,2]

[5,2,3,4,5]

[1,2,10]

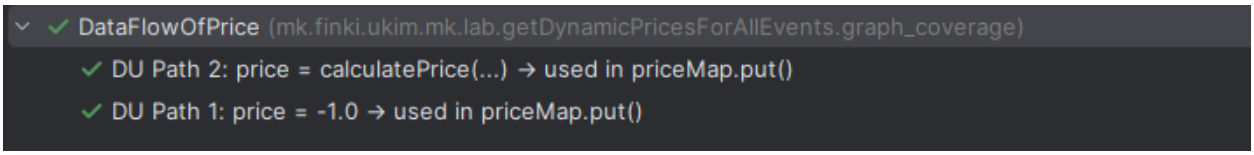
[4,5,2,3,4]



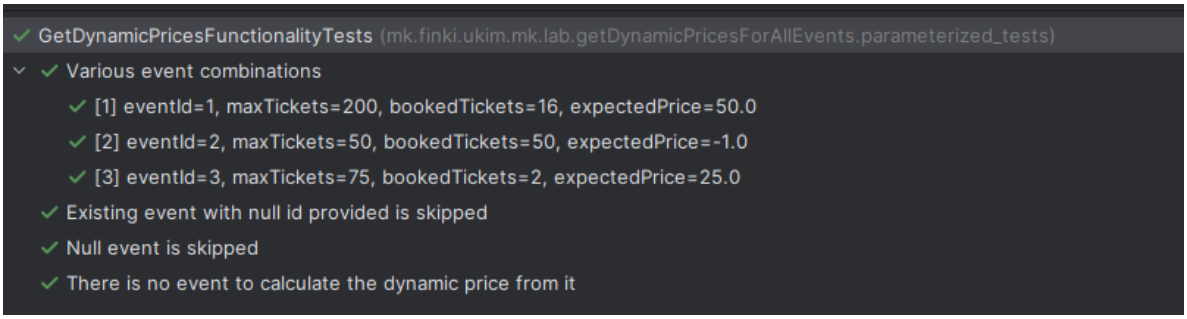
Du Paths coverage for the usage and definitions of price variable.

All DU Path Coverage for all variables are:

Variable	All DU Path Coverage
price	[1,2,3,6,7,9,5,2,10] [1,2,3,6,8,9,5,2,10]



We inserted some parameterized tests that are going to test the method itself too.



LoyaltyUtils. getLoyaltyDiscount

Method Signature:

public static double getLoyaltyDiscount(Integer bookingCount)

Input Space Partitioning with Base Choice

Characteristics:

C1 – bookingCount

- C1A: bookingCount == null
- C1B: bookingCount <0
- C1C: bookingCount ==0
- C1D: bookingCount >0

C2 – value of return number(x)

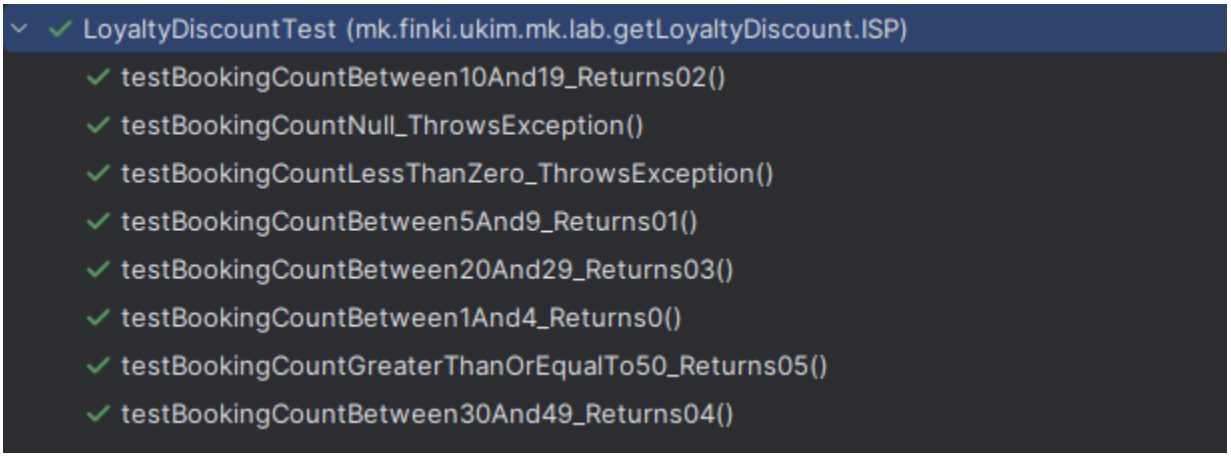
- C2A: x==null
- C2B: x<0
- C2C: x==0.0
- C2D: 0.0<x<=0.1
- C2E: 0.1<x<=0.2
- C2F: 0.2<x<=0.3
- C2G: 0.3<x<=0.4
- C2H: 0.4<x<=0.5
- C2J: x>0.5

For best choice we choose C1D C2D – T

Below, we added the table where we see all combinations for Base Choice method.

row	C1	C2	Possible?	Reason
1	C1D	C2D	Yes	bookingCount > 0 → if it's 5–9 → returns 0.1 (valid range)
2	C1A	C2D	Yes	null → throws exception, never returns any numeric value
3	C1B	C2D	Yes	negative → throws exception
4	C1C	C2D	No	bookingCount == 0 → returns 0.0 → not in 0.0 < x <= 0.1
5	C1D	C2A	No	never returns null (return type is double, not Double)
6	C1D	C2B	No	never returns negative value
7	C1D	C2C	Yes	bookingCount > 0 but if bookingCount ∈ [1–4], return is 0.0
8	C1D	C2E	Yes	bookingCount ∈ [10–19] → return in (0.1–0.2]
9	C1D	C2F	Yes	bookingCount ∈ [20–29] → return in (0.2–0.3]
10	C1D	C2G	Yes	bookingCount ∈ [30–49] → return in (0.3–0.4]
11	C1D	C2H	Yes	bookingCount ≥ 50 → return in (0.4–0.5]
12	C1D	C2J	No	function never returns > 0.5 (max is 0.5)

So we wrote all the test combinations that are possible to distinguish (total of 8)



Graph Coverage

On the picture below we can see the graph we got

1 2

2 3

2 4

4 5

4 6

6 7

6 8

8 9

8 10

10 11

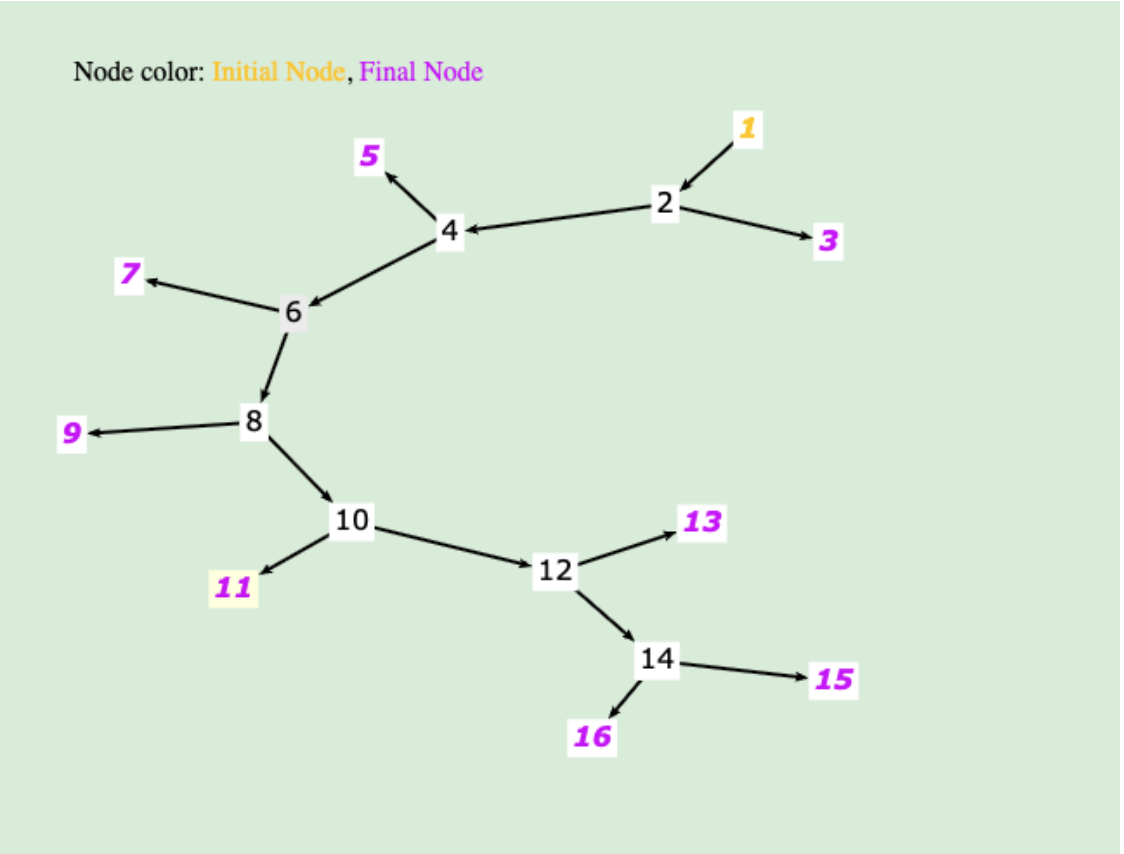
10 12

12 13

12 14

14 15

14 16

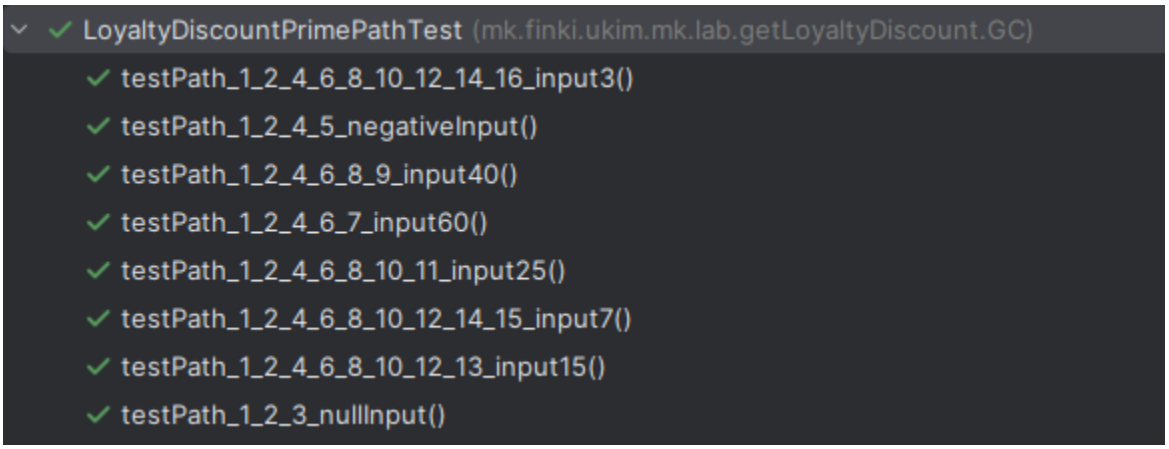


Initial node is 1, and terminal nodes are 3 5 7 9 11 13 15 16

We used prime paths to design our test cases, aiming to cover all important and independent paths through the graph for stronger logic coverage. We got the following Test Requirements:

8 test paths are needed for Prime Path Coverage using the prefix graph algorithm

Test Paths	Test Requirements that are toured by test paths directly
[1,2,3]	[1,2,3]
[1,2,4,5]	[1,2,4,5]
[1,2,4,6,7]	[1,2,4,6,7]
[1,2,4,6,8,9]	[1,2,4,6,8,9]
[1,2,4,6,8,10,11]	[1,2,4,6,8,10,11]
[1,2,4,6,8,10,12,13]	[1,2,4,6,8,10,12,13]
[1,2,4,6,8,10,12,14,15]	[1,2,4,6,8,10,12,14,15]
[1,2,4,6,8,10,12,14,16]	[1,2,4,6,8,10,12,14,16]

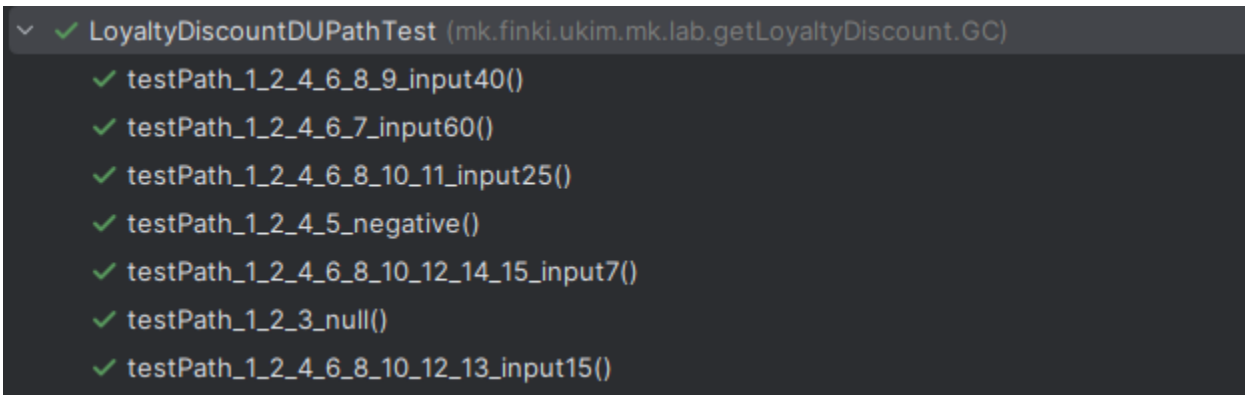


Then we used Du-Paths for the bookingCount variable and we got the following coverage result:

All DU Path Coverage for all variables are:

Variable	All DU Path Coverage
bookingCount	[1,2,3] [1,2,4,5] [1,2,4,6,7] [1,2,4,6,8,9] [1,2,4,6,8,10,11] [1,2,4,6,8,10,12,13] [1,2,4,6,8,10,12,14,15]

From this, we test the variable defs and uses with the following tests:



We wrote several parameterized unit tests and pure unit tests so we can test the method logic itself too.

Source Type	Method Name	Purpose
@ValueSource	testLoyaltyDiscount_ValueSource	Verifies that common values (3–60) **don't throw exceptions**
@CsvSource	testLoyaltyDiscount_CsvSource	Checks the **exact discount output** for given booking counts
@EnumSource	testLoyaltyDiscount_EnumSource	Ensures predefined levels (LOW, MEDIUM, HIGH) are handled without error
@MethodSource	testLoyaltyDiscount_MethodSource	**Tests edge cases**, including null and negative values, expecting exceptions

Location.isValidLocation()

Method Signature:

public boolean isValidLocation()

Logic Coverage

This method contains a clear structure that precisely fits for testing Logic Coverage, because has a predicate with several clauses.

Clause:

A – name.isBlank()

B – address.isBlank()

C – capacity.matches("\\d+")

D – description.length() >= 10

Predicate: P = !A ∧ !B ∧ C ∧ D

Here it is the truth table provided:

Row#	P	A	B	C	D	P	PP	PA	PB	PC	PD
1	T	T	T	T	T		T				
2	T	T	T	T			T				
3	T	T	T		T		T				
4	T	T	T				T				
5	T	T		T	T		T	T			
6	T	T		T			T				
7	T	T			T		T				
8	T	T					T				
9	T		T	T	T		T		T		
10	T		T	T			T				
11	T		T		T		T				
12	T		T				T				
13	T			T	T	T	T	T	T	T	T
14	T			T			T				T
15	T				T		T			T	
16	T						T				
17		T	T	T	T	T	T				
18		T	T	T		T	T				
19		T	T		T	T	T				
20		T	T			T	T				
21		T		T	T	T	T	T			
22		T		T		T	T				
23		T			T	T	T				
24		T				T	T				
25			T	T	T	T	T		T		
26			T	T		T	T				
27			T		T	T	T				
28			T			T	T				
29				T	T		T	T	T	T	T
30				T		T	T				T
31					T	T	T			T	
32						T	T				

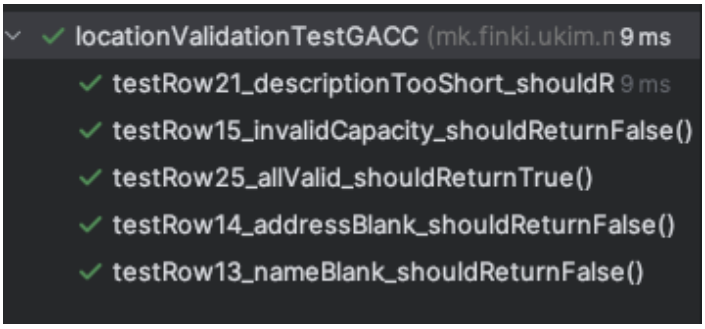
We tried GACC and RACC in the function and we got the following results:

- GACC

The following result for GACC is based on the truth table on the right:

Major Clause	Set of possible tests
P	(1,17), (1,18), (1,19), (1,20), (1,21), (1,22), (1,23), (1,24), (1,25), (1,26), (1,27), (1,28), (1,29), (1,30), (1,31), (1,32), (2,17), (2,18), (2,19), (2,20), (2,21), (2,22), (2,23), (2,24), (2,25), (2,26), (2,27), (2,28), (2,29), (2,30), (2,31), (2,32), (3,17), (3,18), (3,19), (3,20), (3,21), (3,22), (3,23), (3,24), (3,25), (3,26), (3,27), (3,28), (3,29), (3,30), (3,31), (3,32), (4,17), (4,18), (4,19), (4,20), (4,21), (4,22), (4,23), (4,24), (4,25), (4,26), (4,27), (4,28), (4,29), (4,30), (4,31), (4,32), (5,17), (5,18), (5,19), (5,20), (5,21), (5,22), (5,23), (5,24), (5,25), (5,26), (5,27), (5,28), (5,29), (5,30), (5,31), (5,32), (6,17), (6,18), (6,19), (6,20), (6,21), (6,22), (6,23), (6,24), (6,25), (6,26), (6,27), (6,28), (6,29), (6,30), (6,31), (6,32), (7,17), (7,18), (7,19), (7,20), (7,21), (7,22), (7,23), (7,24), (7,25), (7,26), (7,27), (7,28), (7,29), (7,30), (7,31), (7,32), (8,17), (8,18), (8,19), (8,20), (8,21), (8,22), (8,23), (8,24), (8,25), (8,26), (8,27), (8,28), (8,29), (8,30), (8,31), (8,32), (9,17), (9,18), (9,19), (9,20), (9,21), (9,22), (9,23), (9,24), (9,25), (9,26), (9,27), (9,28), (9,29), (9,30), (9,31), (9,32), (10,17), (10,18), (10,19), (10,20), (10,21), (10,22), (10,23), (10,24), (10,25), (10,26), (10,27), (10,28), (10,29), (10,30), (10,31), (10,32), (11,17), (11,18), (11,19), (11,20), (11,21), (11,22), (11,23), (11,24), (11,25), (11,26), (11,27), (11,28), (11,29), (11,30), (11,31), (11,32), (12,17), (12,18), (12,19), (12,20), (12,21), (12,22), (12,23), (12,24), (12,25), (12,26), (12,27), (12,28), (12,29), (12,30), (12,31), (12,32), (13,17), (13,18), (13,19), (13,20), (13,21), (13,22), (13,23), (13,24), (13,25), (13,26), (13,27), (13,28), (13,29), (13,30), (13,31), (13,32), (14,17), (14,18), (14,19), (14,20), (14,21), (14,22), (14,23), (14,24), (14,25), (14,26), (14,27), (14,28), (14,29), (14,30), (14,31), (14,32), (15,17), (15,18), (15,19), (15,20), (15,21), (15,22), (15,23), (15,24), (15,25), (15,26), (15,27), (15,28), (15,29), (15,30), (15,31), (15,32), (16,17), (16,18), (16,19), (16,20), (16,21), (16,22), (16,23), (16,24), (16,25), (16,26), (16,27), (16,28), (16,29), (16,30), (16,31), (16,32)
A	(5,13), (5,29), (21,13), (21,29)
B	(9,13), (9,29), (25,13), (25,29)
C	(13,15), (13,31), (29,15), (29,31)
D	(13,14), (13,30), (29,14), (29,30)

Tests chosen: (31,21), (21,13), (25,13), (13,15), (13,14) - (13,14,15,21,25,31)

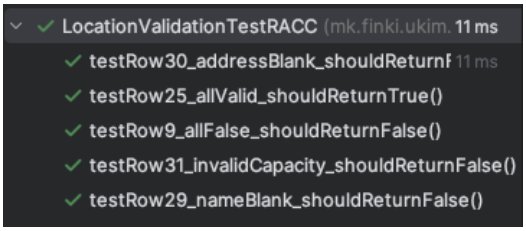


- RACC

The following result for RACC is based on the truth table on the right:

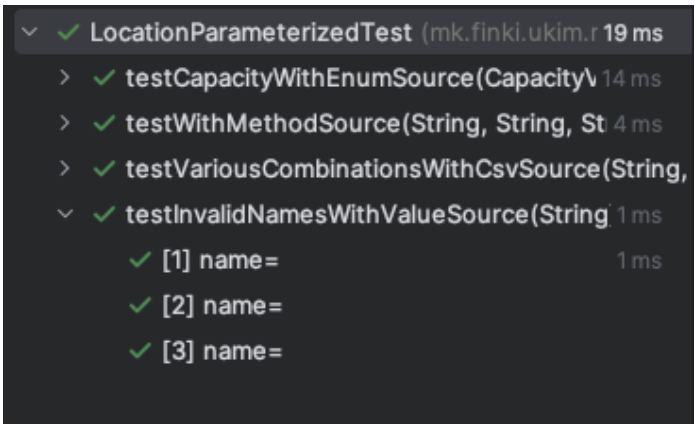
Major Clause	Set of possible tests
P	(1,17), (2,18), (3,19), (4,20), (5,21), (6,22), (7,23), (8,24), (9,25), (10,26), (11,27), (12,28), (13,29), (14,30), (15,31), (16,32)
A	(5,13), (21,29)
B	(9,13), (25,29)
C	(13,15), (29,31)
D	(13,14), (29,30)

Tests chosen: (29,30), (29,31),(25,29),(21,29),(9,25) – (9, 21, 25, 29, 31)



We wrote some base tests too, with parameters in them. On the table bellow you can see a description for the tests and which approach we used to create them

Source Type	Used For Testing…
@ValueSource	One invalid field (blank names)
@CsvSource	Many test cases in one block (including expected)
@MethodSource	Programmatic creation of test arguments
@EnumSource	Enum values affecting capacity validation



BookingService.addToCart()

Method Signature:

BookingCart addToCart(String selectedEvent, Integer numTickets, String username, String address, HttpSession session)

Input Space Partitioning with Base Choice Coverage

Characteristics:

C1 - selectedEvent

- C1A - selectedEvent==null
- C1B - selectedEvent== “ ”
- C1C – selectedEvent != “ ”

C2 - numTicketsC2A - numTickets==null

- C2B - numTickets < 0
- C2C – numTickets== 0
- C2D– numTickets> 0

C3 - username

- C3A - username==null
- C3B - username== “ ”
- C3C – username != “ ”

C4- address

- C4A - address==null
- C4B - address == “ ”
- C4C – address!= “ ”

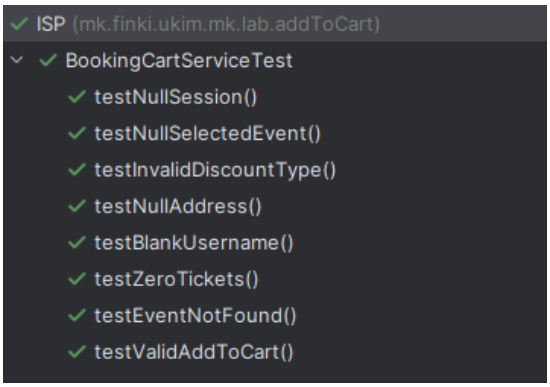
C5- address

- C5A - session==null
- C5B - session!= null

Best choice: C1C C2D C3C C4C C5B

Table:

row	C1	C2	C3	C4	C5	Possible	Reason
1	C1C	C2D	C3C	C4C	C5B	yes	All values valid and session exists
2	C1A	C2D	C3C	C4C	C5B	yes	selectedEvent == null → eventService.findByName(null) → throws exception
3	C1B	C2D	C3C	C4C	C5B	yes	selectedEvent == "" → likely returns null → event.getId() fails
4	C1C	C2A	C3C	C4C	C5B	yes	numTickets == null → will throw exception in event.calculatePrice()
5	C1C	C2B	C3C	C4C	C5B	yes	numTickets < 0 → invalid logic → no check → unexpected behavior
6	C1C	C2C	C3C	C4C	C5B	yes	numTickets == 0 is allowed (might result in 0 price, but not error)
7	C1C	C2D	C3A	C4C	C5B	yes	username == null → acceptable unless used later for display/email
8	C1C	C2D	C3B	C4C	C5B	yes	username == "" is syntactically valid
9	C1C	C2D	C3C	C4A	C5B	yes	address == null is accepted in code
10	C1C	C2D	C3C	C4B	C5B	yes	address == "" is accepted in code
11	C1C	C2D	C3C	C4C	C5A	yes	session == null → throws NullPointerException



Graph Coverage

0 1

1 2

1 15

2 3

2 16

3 4

3 17

4 5

4 18

5 6

5 19

6 7

7 8

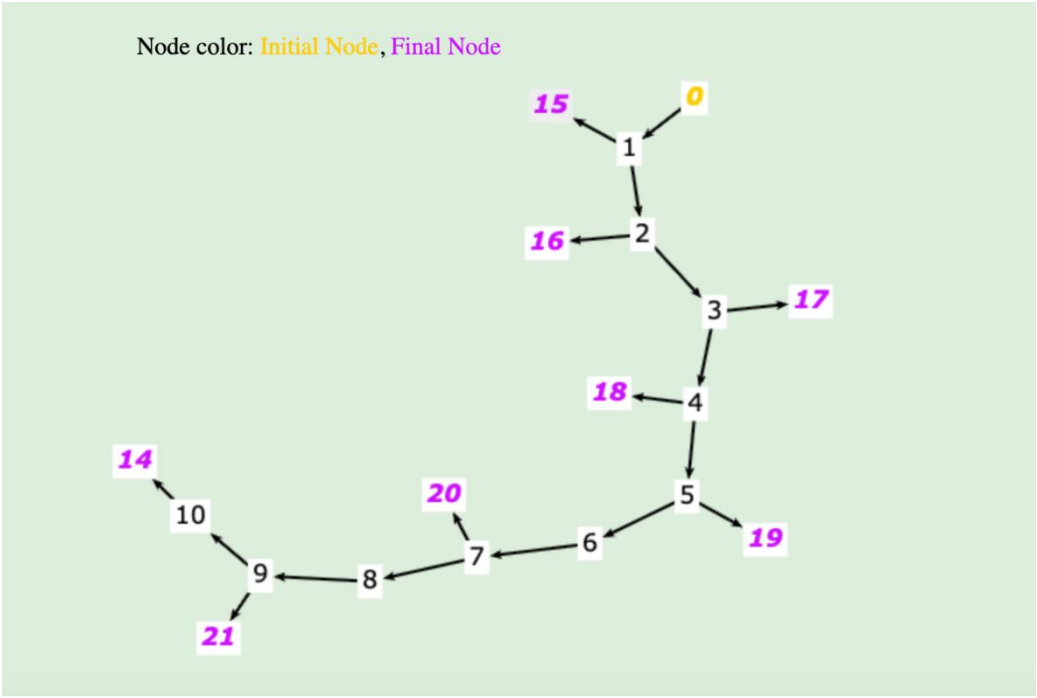
7 20

8 9

9 10

9 21

10 14

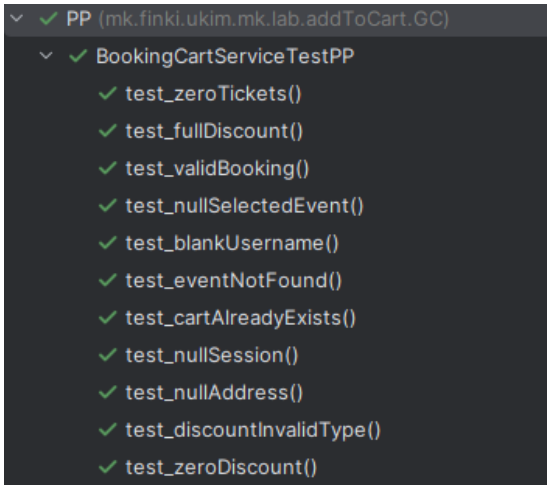


Initial node is 0, terminal nodes are 14 15 16 17 18 19 20 21

Graph Coverage with Prime Paths

8 test paths are needed for Prime Path Coverage

Test Paths	Test Requirements that are toured by test paths directly
[0,1,2,3,4,5,6,7,8,9,10,14]	[0,1,2,3,4,5,6,7,8,9,10,14]
[0,1,2,3,4,5,6,7,8,9,21]	[0,1,2,3,4,5,6,7,8,9,21]
[0,1,2,3,4,5,6,7,20]	[0,1,2,3,4,5,6,7,20]
[0,1,2,3,4,5,19]	[0,1,2,3,4,5,19]
[0,1,2,3,4,18]	[0,1,2,3,4,18]
[0,1,2,3,17]	[0,1,2,3,17]
[0,1,2,16]	[0,1,2,16]
[0,1,15]	[0,1,15]

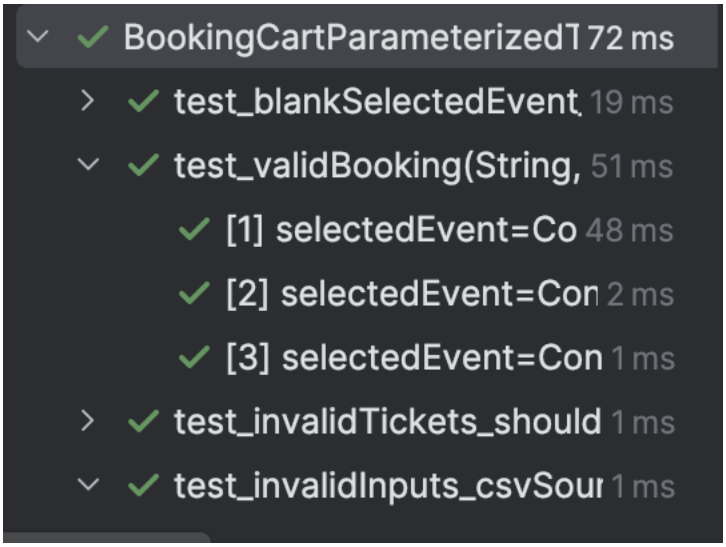


Graph Coverage with All Du-Paths Coverage for event

All DU Path Coverage for all variables are:

Variable	All DU Path Coverage
event	[0,1,2,3,4,5,6,7,20] [0,1,2,3,4,5,6,7,8,9,10,14]

Same as all the methods, we wrote some base tests so we can test the functionality of the method and its behavior on different scnearios



EventService.save_event()

Method Signature:

void save_event(Long id, String name, String description, double popularityScore, Long locationID, LocalDateTime from, LocalDateTime to,double basePrice, int maxTickets)

Graph Coverage

0 1

1 2

1 3

3 4

3 5

5 6

5 7

7 8

7 9

9 10

9 11

11 12

11 13

13 14

13 15

15 16

15 17

17 18

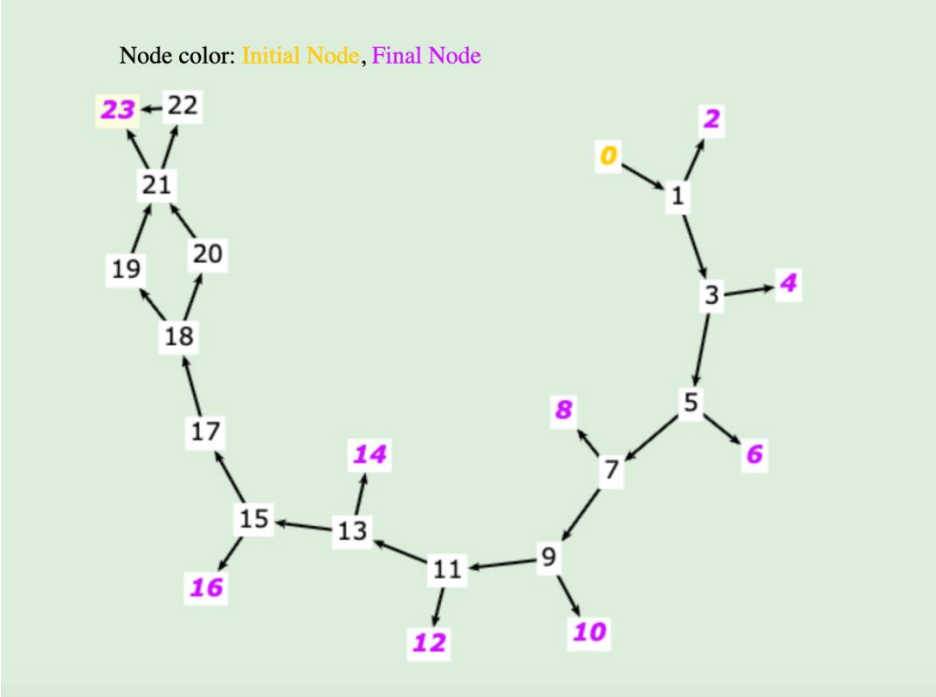
18 19

19 20

20 21

21 22

22 23



Initial node is 0, and terminal are 2 4 6 8 10 12 14 16 23

First we are going to introduce the realisation of GC with Prime Paths

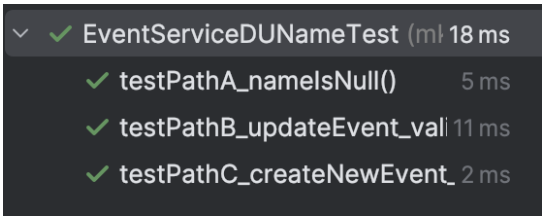
12 test paths are needed for Prime Path Coverage

Test Paths	Test Requirements that are toured by test paths directly
[0,1,3,5,7,9,11,13,15,17,18,19,21,22,23]	[0,1,3,5,7,9,11,13,15,17,18,19,21,22,23]
[0,1,3,5,7,9,11,13,15,17,18,20,21,22,23]	[0,1,3,5,7,9,11,13,15,17,18,20,21,22,23]
[0,1,3,5,7,9,11,13,15,17,18,19,21,23]	[0,1,3,5,7,9,11,13,15,17,18,19,21,23]
[0,1,3,5,7,9,11,13,15,17,18,20,21,23]	[0,1,3,5,7,9,11,13,15,17,18,20,21,23]
[0,1,3,5,7,9,11,13,15,16]	[0,1,3,5,7,9,11,13,15,16]
[0,1,3,5,7,9,11,13,14]	[0,1,3,5,7,9,11,13,14]
[0,1,3,5,7,9,11,12]	[0,1,3,5,7,9,11,12]
[0,1,3,5,7,9,10]	[0,1,3,5,7,9,10]
[0,1,3,5,7,8]	[0,1,3,5,7,8]
[0,1,3,5,6]	[0,1,3,5,6]
[0,1,3,4]	[0,1,3,4]
[0,1,2]	[0,1,2]

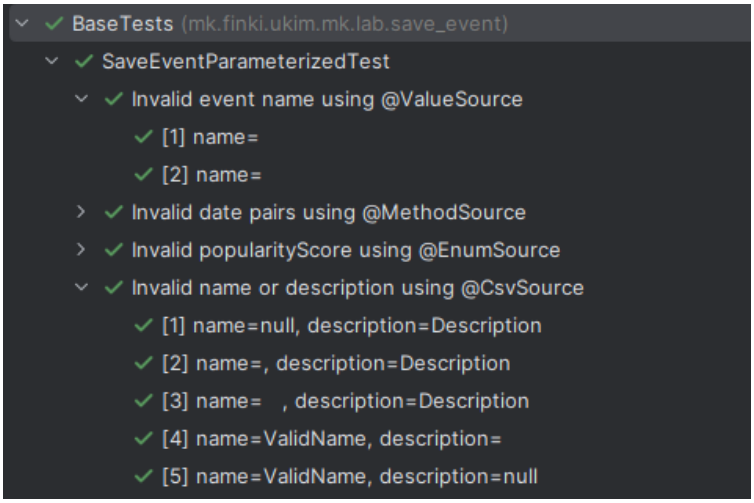
And we do with All Du-Path coverage for name variable

All DU Path Coverage for all variables are:

Variable	All DU Path Coverage
name	[0,1,2] [0,1,3,5,7,9,11,13,14] [0,1,3,5,7,9,11,13,15,16]



In addition to structural coverage techniques, we also designed a set of base tests to validate method behavior under both valid and invalid input conditions. These tests help ensure that the method respond appropriately to various edge cases, including incorrect or unexpected inputs.



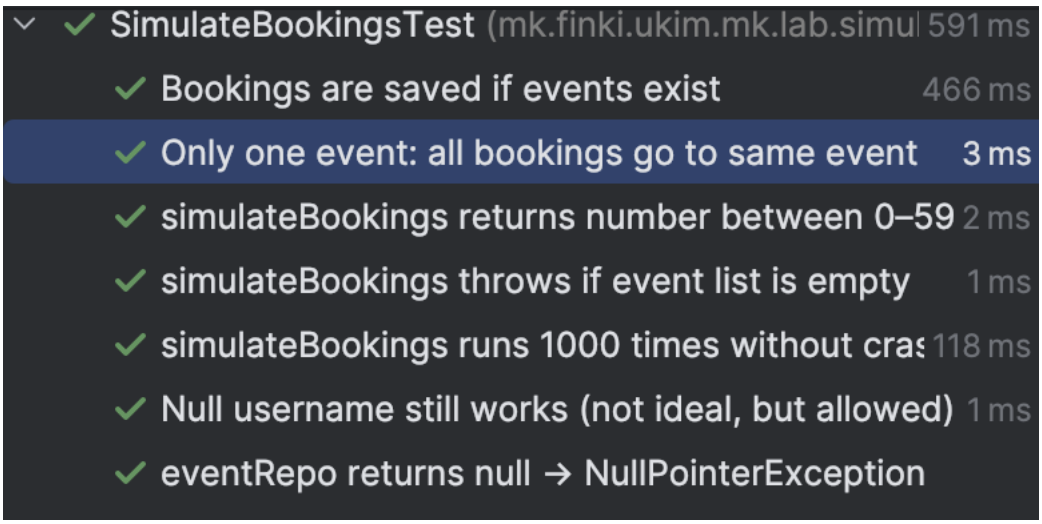
LoginService.simulateBookings()

During the testing of this application, we were using several times @Mock, so we can mock a behavior to some repository mostly (as an interface) and to check some cases while the testing.

Now we chose this function to be tested with Mocks only. The method simulates a random number (0–59) of event bookings for a given user. It selects random events from the database and creates EventBooking entries using a booking repository.

The usage of mockito is in:

a) Mock EventRepository and EventBookingRepository
b) Simulate scenarios like:
    - Available events
    - No events
    - Single event reuse
    - Null usernames
c) Verify how many times bookings are saved
d) Assert correctness of fields (username, address, price)



Spring MVC Test framework

EventBookingControllerTest

The EventBookingControllerTest class verifies the behavior of the EventBookingController using the Spring MVC Test framework.

Now we are going to list the scenarios we covered with this method:

1. /cart/add – Add to Cart
   This endpoint tests the functionality of adding an event to the booking cart. It simulates a valid booking by mocking selectedEvent, numTickets, username, and address, then verifies redirection to /cart/view and checks that the cart is correctly stored in the session.
2. /cart/view – View Cart
   The /cart/view endpoint is tested under two conditions.
   - First, it simulates a session where the cart already exists and contains booking items, *verifying that the model includes the existing cart*.
   - Second, it handles the case where *no cart is present in the session, asserting that a new empty BookingCart is initialized and correctly added to the model*.
3. /cart/confirm – Confirm Cart
   The /cart/confirm endpoint is tested by simulating a session that contains a populated BookingCart. The test mocks the eventBookingService.saveBooking() method to handle the booking process. It then verifies that the user is redirected to the bookingConfirmation page. Additionally, the test asserts that key session attributes—such as username, attendeeAddress, numTickets, selectedEvent, and totalPrice—are correctly set during the confirmation flow.

Tools & Setup for testing the controllers

- @WebMvcTest(EventBookingController.class) to test controller in isolation
- MockMvc to simulate HTTP requests/responses
- Mockito to mock dependencies:
  - EventService
  - EventBookingService
  - BookingCartService
- .with(csrf()) and .with(user(...)) for authenticated (secured requests)

EventControllerTest

The EventControllerTest class provides unit tests for the EventController, which manages event-related operations in the application such as listing, filtering, adding, editing, and deleting events.

Method	Purpose	Covered Scenarios
testGetEventsPage()	Tests the /events GET endpoint	Ensures events are loaded into the model and the correct view (listEvents) is returned.
testFilterList()	Tests the /events/filter_events POST endpoint	Verifies that filtering by text and rating works, the correct model attributes are added (event_list, bookedTickets, eventPrices), and view is listEvents.
testAddNewEvent()	Tests /events/add-form GET endpoint (admin only)	Verifies the add event form is returned with a list of all locations.
testDeleteEventFromList()	Tests /events/delete/{id} GET endpoint (admin only)	Confirms that an event can be deleted and user is redirected to /events.
testSaveEvent()	Tests /events/add POST endpoint	Verifies a new or updated event is saved and the user is redirected to the event list.
testBookEvent()	Tests /events/book_event POST endpoint	Ensures that booking an event stores the user, ticket number, and event name in session and redirects to /eventBooking.
testEditEvent()	Tests /events/edit/{id} GET endpoint (admin only)	Verifies that the edit form is pre-filled with event and location data.

LocationControllerTest

The LocationControllerTest class verifies the behavior of the LocationController, which manages HTTP requests for creating, listing, and deleting event locations. It uses MockMvc and Mockito to simulate and validate the controller's behavior without starting a full web server.

Test Method Name	Description
testLocationsPage()	Tests GET /locations endpoint. Ensures the list of locations is fetched and available as location_list in the model.
testDeleteLocationFromList()	Tests GET /locations/delete/{id}. Confirms that the location is deleted and redirects to the /locations page.
testAddNewLocation()	Tests GET /locations/add-loc. Validates that the add-location form view is correctly returned.
testSaveLocation()	Tests POST /locations/add. Validates saving a location with the correct parameters, and redirection to the /locations page.

LoginControllerTest

This test verifies that the login page is rendered correctly. Since the application does not implement actual login logic, this is the only test needed.

The only scenario that is covered here is the GET /login scenario, that is testing the user has no session on the application, to redirect him on the login page.

LogoutControllerTest

The LogoutControllerTest verifies the behavior of the logout route in the application. It checks whether accessing the /logout endpoint triggers a redirection to the /login page as expected. This controller does not perform actual logout logic but serves as a redirection handler post-logout. The test confirms if accessing GET /logout returns a 3xx redirection status and the redirection to /login.

Conclusion for Spring MVC Testing Framework

The Spring MVC Test framework provides a powerful and efficient way to test Spring web applications by allowing developers to simulate HTTP requests and verify controller behavior without deploying the application to a server. It supports thorough testing of request mappings, model attributes, view resolution, and session handling.