Ryerson University

Department of Electrical, Computer, & Biomedical Engineering
Faculty of Engineering & Architectural Science

Course Title	Software Testing and Quality Assurance
Course Number	COE 891
Semester/Year	W2025

Instructor	Reza Samavi

Assignment/Lab Number:	Term Project
Assignment/Lab title	Final Report

Submission Date:	04/08/2025
Due Date:	04/09/2025

Student Last Name	Student First Name	Student Number	Section	Signature
Timbol	Jesdin Edward	501027997	9	J.T
Taing	Ryan	501093407	9	R.T.
Malik	Hamza	501112545	9	H.M

*By signing above you attest that you have contributed to this written lab report and confirm that all work you have contributed to this lab report is your own work. Any suspicion of copying or plagiarism in this work will result in an investigation of Academic Misconduct and may result in a “0” on the work, an “F” in the course, or possibly more severe penalties, as well as a Disciplinary

Flight Booking Test Plan

1.0 Introduction

The flight booking Java application is a program which is designed to streamline the airline ticket booking/reservation process, making it a smoother process for users. This project uses object-oriented programming to handle tasks such as flight searches, seat selection, fare calculation, and data storage.

The application's testing will ensure that the system's quality, reliability, and user satisfaction is met. This phase will utilize four testing techniques which include Input Space Partitioning, Graph-Based Testing, Logic-Based Testing, and Mutation Testing. Each test is tailored to rigorously evaluate the program's application functionality and performance. The tests will also validate the key features, such as seat selection and fare calculation, while checking for potential vulnerabilities. Adopting this structured testing approach, this will allow for a user-centric and functional flight booking solution.

1.1 Project Information

- **Project Name:** Flight Booking Application in Java
- **Developers:** Jesdin Edward Timbol, Ryan Taing, Hamza Malik
- **Project Description:** A Java-based software program called the Flight Booking Application was created to maximize airline ticket purchases. Users may browse flights, purchase tickets, choose seats, and get booking confirmations with its features. Data privacy and a secure login are guaranteed by the program.

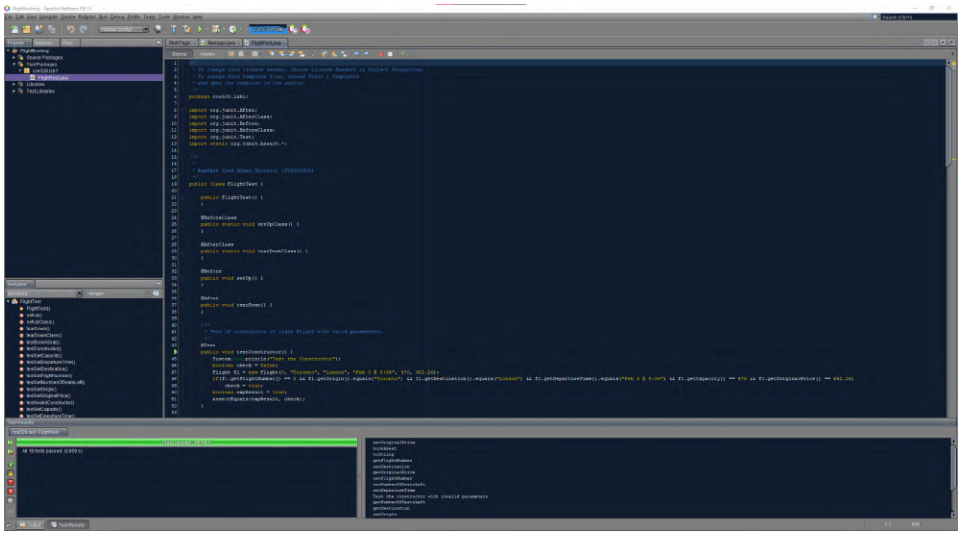


Fig. 1: Example JUnit Test for Project File (Flight.java)

1.2 Document Identifier

- **Document Name:** Flight Booking Application Test Plan
- **Version:** 1.0
- **Date:** 2025/03/17
- **Author:** Group 2

1.3 Scope

- This document outlines the test plan for the Flight Booking Application, including unit, integration, system, and acceptance testing.
- The primary focus is on functional correctness, security, and performance.

1.4 References

- **Project Repository:** GitHub Link
- **Software Testing Principles:** P. Ammann and J. Offutt, *Introduction to Software Testing*, Cambridge University Press, 2016.

1.5 Level in the Overall Sequence

- This test plan covers various stages:
    1. **Input Space Partitioning (ISP):** Boundary Value Analysis (BVA) was used to test the input space based on the application of each Java class. Testing programs were implemented and coded accordingly.
    2. **Graph-based testing (CFG and DFG):** CFGs were drawn, and the corresponding analysis was provided based on different coverage methods for control flow. DFGs were also drawn, and their corresponding analysis was conducted based on various coverage methods for data flow.
    3. **Logic-based Testing:** Logical analysis was done based on different coverage methods for logical predicates and clauses.
    4. **Mutation Testing:** The program under test was modified in small ways, such as flipping a less-than sign to a greater-than or changing a hard-coded number from 0 to 1. These modifications were designed to mimic typical programming errors, including typographical errors, wrong choice of operator, or off-by-one errors.

1.6 Test Classes and Overall Test Conditions

- The application consists of modules such as Flight Search, Booking, Payment Processing, and User Authentication.
- The test conditions focus on **input validation, UI functionality, database integrity, security measures, and performance benchmarks**.

2. Details for The Level Test Plan

The following sections outlines a comprehensive approach for software testing the flight booking system, implementing techniques to ensure the integrity and reliability of the project. The different tests in which we will be implementing revolve around Input Space Partitioning (ISP), Graph-based testing, Logic-based Testing, and Mutation Testing, each targeting different aspects of the software's behavior and structure.

For Input Space Partitioning (ISP), Boundary Value Analysis (BVA) will be utilized to systematically test the input space, focusing on edge cases and boundary conditions to identify potential vulnerabilities. Graph-based testing involves the creation and analysis of Control Flow Graphs (CFGs) and Data Flow Graphs (DFGs) to evaluate different coverage methods for both control and data flow. Logic-based Testing will focus on analyzing logical predicates and clauses, ensuring thorough coverage of decision points within the code. Finally, Mutation Testing will be employed to introduce small, intentional changes to the program, mimicking common programming errors, to assess the effectiveness of the test suite in detecting such faults.

2.1 Test Items and Identifiers

Test Item	Description	Test Cases
Flight Search	Guarantees that consumers may use input criteria to search for flights that are available.	● Validate flight results with correct input criteria. ● Check error handling for invalid input criteria.
Booking System	Verifies booking workflow from selection to confirmation	● Test seamless booking from selection to confirmation. Verify handling of unavailable flights during booking.
Real-time Fare Calculation	Tests secure payment transactions	● Confirm successful payment with valid details. ● Test error handling for failed payment transactions.
Automated Booking Confirmation	Users receive instant booking confirmation upon successful reservation.	● Confirm instant booking notification after reservation. ● Verify confirmation details (flight info, booking ID, payment status).

User Authentication & Data Security	Ensures data consistency and integrity	● Validate successful login with correct credentials. ● Test security measures for invalid login attempts.

2.2 Test Traceability Information

This section documents the origin of each test case, mapping them to the functional requirements specified in the Project Abstract and System Requirements. This ensures that each test case is traceable to a specific requirement, providing clarity on why it was included and what it tests.

Flight Search

- Test Case 1: Validate flight results with correct input criteria.
    - Requirement: The system must allow users to search for flights using valid input criteria
    - Purpose: Ensures the system returns accurate flight results for valid searches.

- Test Case 2: Check error handling for invalid input criteria.
    - Requirement: The system must handle invalid input gracefully and provide appropriate error messages.
    - Purpose: Verifies the system's ability to manage invalid search inputs.

Booking System

- Test Case 1: Test seamless booking from selection to confirmation.
    - Requirement: The system must support a smooth booking workflow from flight selection to confirmation.
    - Purpose: Ensures the booking process is user-friendly and functional.

- Test Case 2: Verify handling of unavailable flights during booking.
    - Requirement: The system must handle scenarios where a selected flight is no longer available.
    - Purpose: Confirms proper error handling and user notification for unavailable flights.

Real-time Fare Calculation

- Test Case 1: Confirm successful payment with valid details.
    - Requirement: The system must process payments securely and confirm bookings upon successful transactions.
    - Purpose: Validates secure payment processing and booking confirmation.

- Test Case 2: Test error handling for failed payment transactions.
    - Requirement: The system must handle payment failures and notify users appropriately.
    - Purpose: Ensures proper handling of payment errors.

Automated Booking Confirmation

- Test Case 1: Confirm instant booking notification after reservation.
    - Requirement: Users must receive instant confirmation (e.g., email/SMS) upon successful booking.
    - Purpose: Verifies timely delivery of booking confirmations.

- Test Case 2: Verify confirmation details (flight info, booking ID, payment status).
    - Requirement: Booking confirmations must include accurate details (flight info, booking ID, payment status).
    - Purpose: Ensures the accuracy of confirmation content.
    -

User Authentication & Data Security

- Test Case 1: Validate successful login with correct credentials.
    - Requirement: The system must authenticate users with valid credentials.
    - Purpose: Confirms proper authentication functionality.

- Test Case 2: Test security measures for invalid login attempts.
    - Requirement: The system must implement security measures (e.g., account lockout) for repeated invalid login attempts.
    - Purpose: Verifies the system's ability to handle unauthorized access attempts.

2.3 Features Tested

- Flight Search Functionality
    - Allows users to search through available flights for booking
- Dynamic Seat Selection
    - Allows users to see their seat upon booking
- Fare Calculation System
    - Allows users to calculate the price of their fare based on whether they are a member or non-member
- Automated Booking Confirmation
    - Allows users to receive a booking confirmation after booking for peace of mind
- User Authentication & Data Security
    - Allows user data to be securely stored

2.4 Features Not Tested

- Third-party API integrations for real-time flight data retrieval (Future enhancement)
- Online payment processing (Not implemented in the current scope)

2.5 Testing Approach

Example of how the four testing methods (Input Space Partitioning (ISP), Graph-based Testing (CFG and DFG), Logic-based Testing, and Mutation Testing) can be applied to the five test cases

- Flight Search:
    - ISP (BVA): Test boundary values for dates (e.g., 01/01/2023, 12/31/2023, 00/00/0000).
    - Graph-based (CFG/DFG): Draw CFG for valid/invalid input paths; analyze data flow for input validation.
    - Logic-based: Test logical conditions like if (date.isValid() && destination.isValid()).
    - Mutation Testing: Change && to || in input validation logic.

- Booking System:
    - ISP (BVA): Test boundary values for passenger count (e.g., 1, 10, 11).
    - Graph-based (CFG/DFG): Draw CFG for booking workflow; analyze data flow for seat availability.
    - Logic-based: Test conditions like if (flight.isAvailable()).
    - Mutation Testing: Change > to < in seat availability check.

- Real-time Fare Calculation:
    - ISP (BVA): Test boundary values for payment amounts (e.g., 0,0,1000, $1001).
    - Graph-based (CFG/DFG): Draw CFG for payment process; analyze data flow for payment status.
    - Logic-based: Test conditions like if (payment.isValid() && payment.isSuccessful()).
    - Mutation Testing: Change 0 to 1 in payment failure handling.

- Automated Booking Confirmation:
    - ISP (BVA): Test boundary values for email length (e.g., empty, 100 characters, 101 characters).
    - Graph-based (CFG/DFG): Draw CFG for confirmation process; analyze data flow for confirmation content.
    - Logic-based: Test conditions like if (confirmation.isSent() && details.areCorrect()).

    - Mutation Testing: Change == to != in confirmation status check.

- User Authentication & Data Security:
    - ISP (BVA): Test boundary values for password length (e.g., empty, 8 characters, 9 characters).
    - Graph-based (CFG/DFG): Draw CFG for login process; analyze data flow for session management.
    - Logic-based: Test conditions like if (attempts < 3 && credentials.areValid()).
    - Mutation Testing: Change 3 to 4 in login attempt limit.

2.6 Item Pass/Fail Criteria

- **Pass:** The test case produces expected results without errors.
- **Fail:** The test case leads to incorrect results, exceptions, or security vulnerabilities.

2.7 Suspension Criteria and Resumption Requirements

- If serious issues (such security lapses or program crashes) are found, testing is stopped. Once bugs have been fixed, testing can resume.

2.8 Test Deliverables

- Test Cases Document
- Test Execution Report
- Bug Tracking Report
- Code Coverage Analysis

3. Test Management

3.1 Planned Activities and Tasks

Activity	**Responsible**
Test Case Design	All Team Members
Unit Testing	All Team Members
Integration Testing	All Team Members
System Testing	All Team Members

Bug Fixing	All Team Members
Report Compilation	All Team Members

3.2 Environment and Infrastructure

- **Development Tools:** IntelliJ IDEA, Eclipse 2023
- **Testing Tools:** JUnit, Selenium, Apache JMeter, Mockito
- **Deployment:** Localhost Server

3.3 Responsibilities and Authority

- **Test Lead:** Oversees the test process and ensures completion.
- **Developers:** Implement unit tests and fix defects.
- **QA Analysts:** Execute test cases and document findings.

3.4 Interfaces Among the Parties Involved

- **Developers & QA Team:** Communicate through GitHub Issues.
- **Project Manager:** Reviews test progress and reports.

3.5 Resources and Allocation

- **Hardware:** Windows/Linux Machines with JDK installed.
- **Software:** Java, Selenium, Eclipse

3.6 Training

- **JUnit & Mockito** for unit and integration testing.
- **Selenium** for web-based UI testing.

3.7 Schedules, Estimates, and Costs

Task	Start Date	End Date	Responsible
Test Plan Creation	March 10th, 2025	April 9th, 2025	All Team Members
Unit Testing	March 10th, 2025	April 9th, 2025	Developers

Integration Testing	March 10th, 2025	April 9th, 2025	QA Team
System Testing	March 10th, 2025	April 9th, 2025	QA Team
Final Report Submission	March 19th, 2025	April 7th, 2025	Group Lead

3.8 Risks and Contingencies

Risk	Mitigation Strategy
Incomplete Test Coverage	Conduct additional exploratory testing
Delay in Bug Fixes	Assign priority to critical defects
Limited Resources	Utilize cloud-based testing services

4. Testing Results

4.1 Input Space Partitioning:

Input space was divided into partitions based on the functionality and constraints in the program:

Flight Class:

Input partitions for **flightNumber:** Positive integers, zero, and negative integers.

- Input partitions for **capacity**: Positive integers, zero, and negative integers.
- Input partitions for **origin** and **destination**: Valid city names vs identical names (to test the validation for origin != destination).
- Input partitions for **originalPrice**: Positive numbers, zero, and negative numbers.

Passenger class (Member and NonMember subclasses):

- Input partitions for **age**: Below 65, exactly 65, and above 65.
- Input partitions for **yearsOfMembership** (in Member and NonMember): Greater than 5, between 1 and 5, and less than or equal to 1.

Manager class:

- Input partitions for origin and destination in the displayAvailableFlights method.
- Input partitions for **flightNumber** in the bookSeat method: Existing flight numbers vs non-existent flight numbers.

Boundary Value Analysis:

The boundary values for each partition are:

- **For the** Flight **class**:
  - flightNumber: -1 (invalid), 0 (boundary), 1 (valid).
  - capacity: -1 (invalid), 0 (boundary), 1 (valid).
  - originalPrice: -1.0 (invalid), 0.0 (boundary), 0.01 (valid).
- **For the** Passenger **class**:
  - age:
    - For NonMember: 64 (below senior discount threshold), 65 (boundary), 66 (above threshold).
    - For Member: Any integer age is valid.
  - yearsOfMembership: 0 (no discount), 1 (boundary), 2-5 (small discount), 6 (large discount).
- **For the** Manager **class**:
  - Valid and invalid combinations of origin and destination cities.
  - Flight numbers at boundaries of available or unavailable.

Test Cases

Here are the following test cases with their inputs and expected values:

# Of TC	Class	Method/Functionality	Type of TC	Inputs	Expected Result
1	**Flight**	Constructor & Validations	Valid Flight	flightNumber=0, origin=Toronto, destination=London, capacity=1, originalPrice=0.01	Valid flight created
2			Invalid Flight Number	flightNumber=-1	Exception for invalid

					flight number
3			Invalid Origin and Destination	origin=Toronto, destination=Toronto	Exception for identical origin and destination
4			Invalid Capacity	capacity=0	Exception for invalid capacity
5			Invalid Price	originalPrice=-0.01	Exception for invalid price
6	**Passenger**	applyDiscount in Member	No Discount	yearsOfMembership=0	Full price (no discount)
7			Small Discount	yearsOfMembership=5	10% discount applied
8			Large Discount	yearsOfMembership=6	50% discount applied

9		applyDiscount in NonMember	No Discount	age=64	Full price (no discount)
10			Senior Discount Threshold	age=65	No discount (boundary, no senior status)
11			Senior Discount Applied	age=66	10% discount applied
12	**Manager**	displayAvailable Flights	Valid Origin/Destination	Valid origin and destination pair	Available flights displayed
13			Invalid Origin/Destination	Invalid origin/destination	No flights displayed
14		bookSeat	Book Seat for Valid Flight Number	Existing flight number	Successfully book a seat
15			Book Seat for Invalid Flight Number	Non-existent flight number	Error message displayed

Observations:
Input validations and boundary checks consistently worked across all classes.

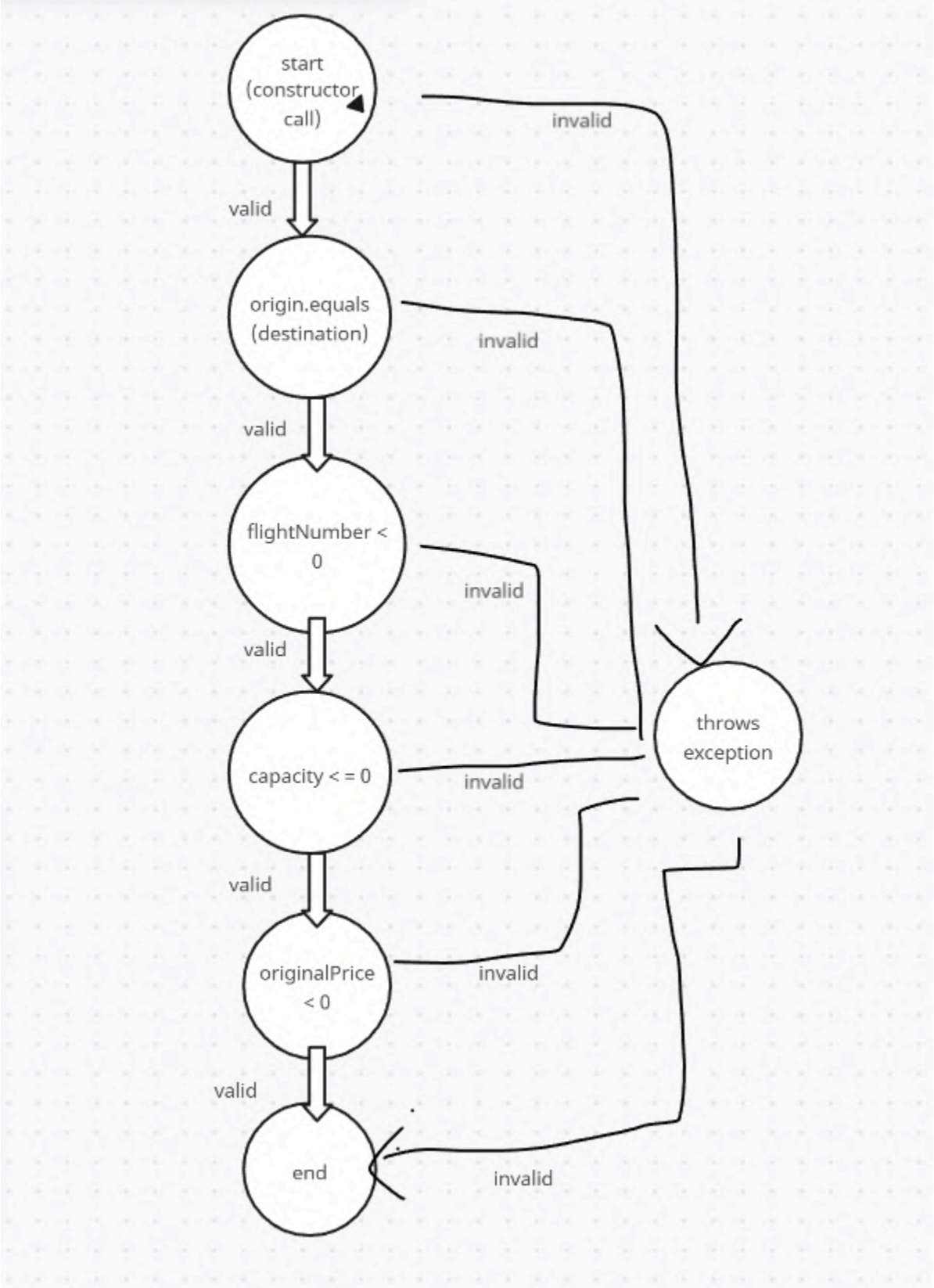
The discount logic in both Member and NonMember classes produced correct outputs for all tested cases.

Seat availability was correctly updated during flight booking, ensuring accurate representation of available seats.

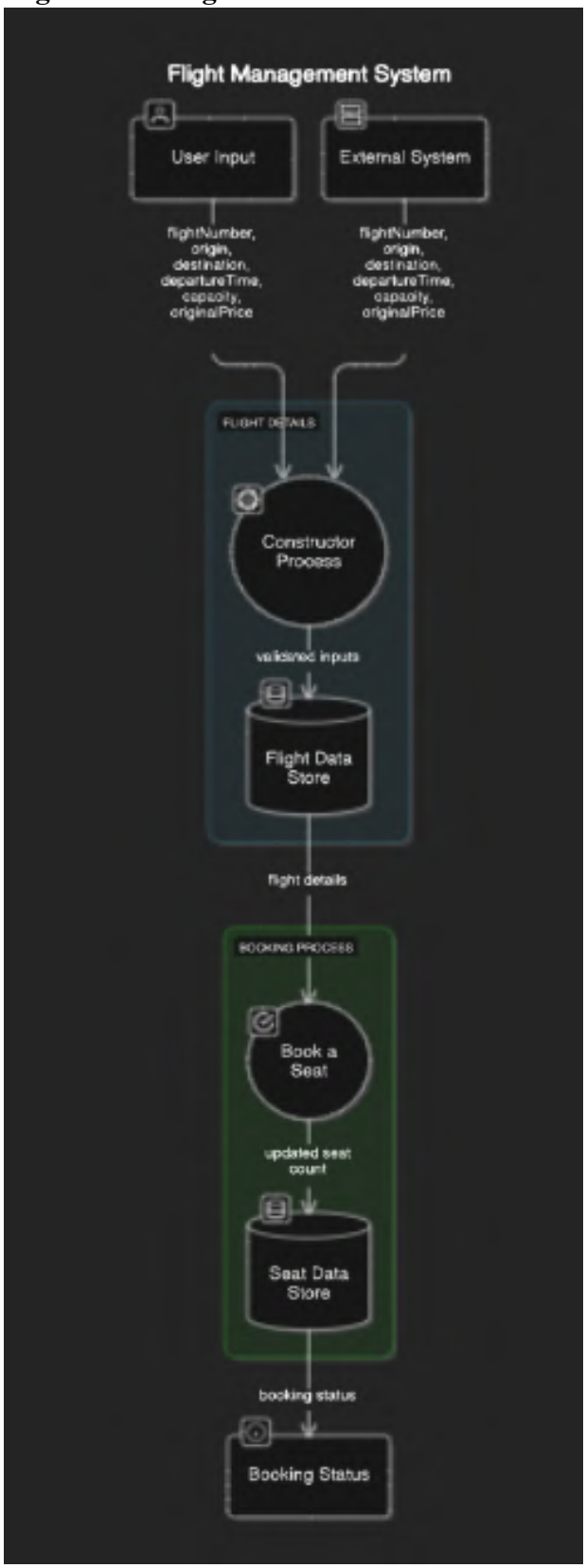
Exception handling for invalid inputs worked as intended, providing a safeguard against invalid data input.

4.2 Graph Based Testing (CFG and DFG):

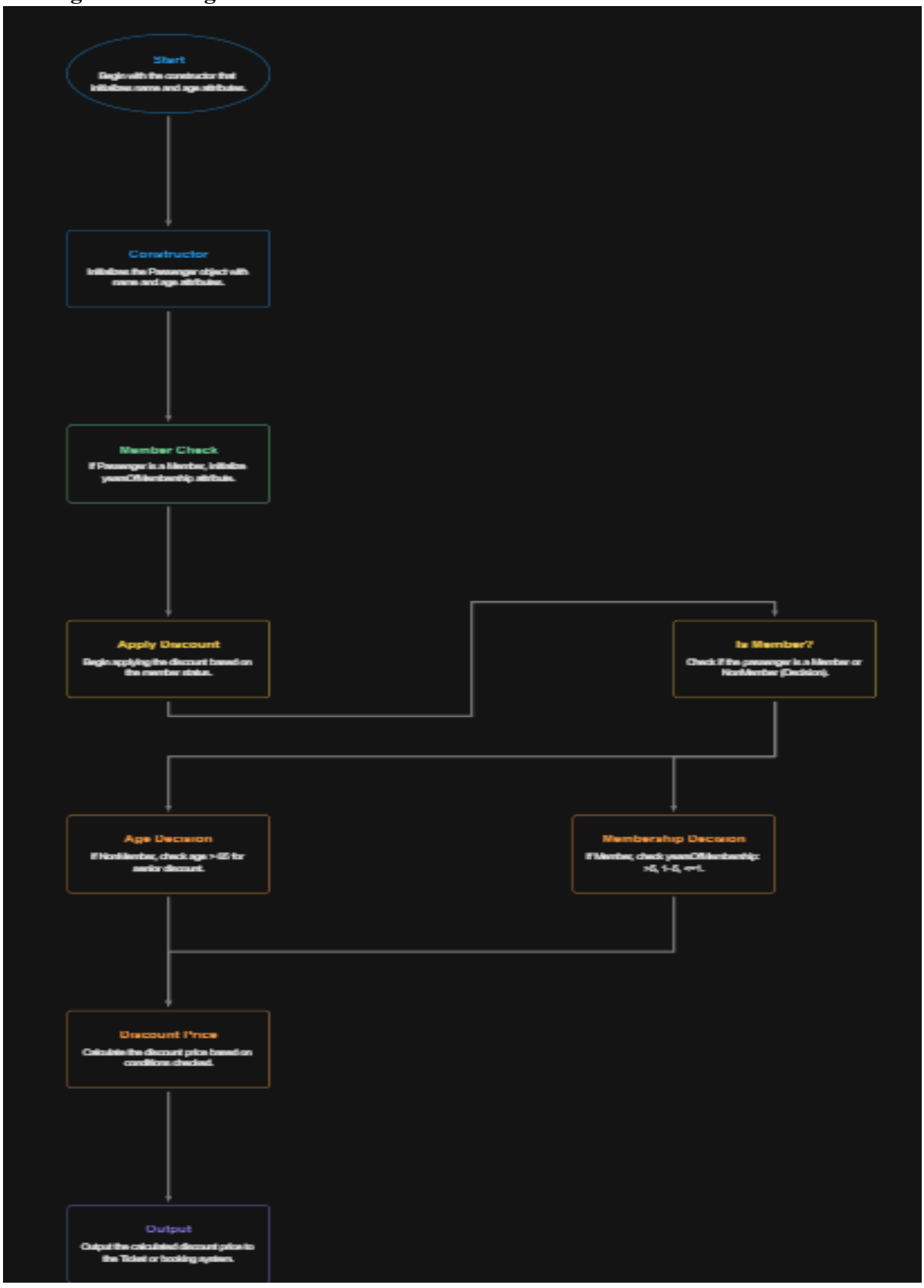
Flight CFG Diagram:



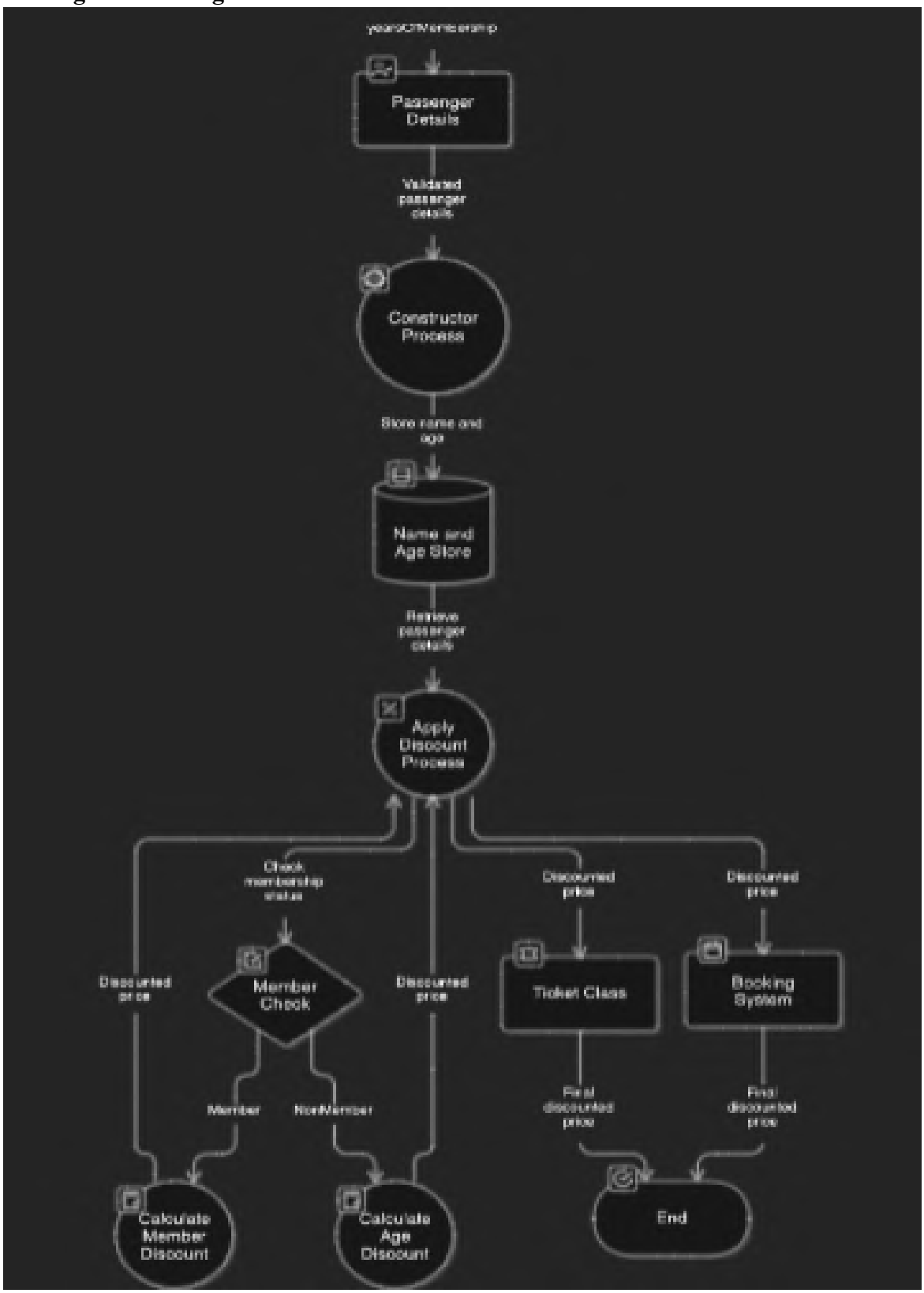
Flight DFG Diagram:



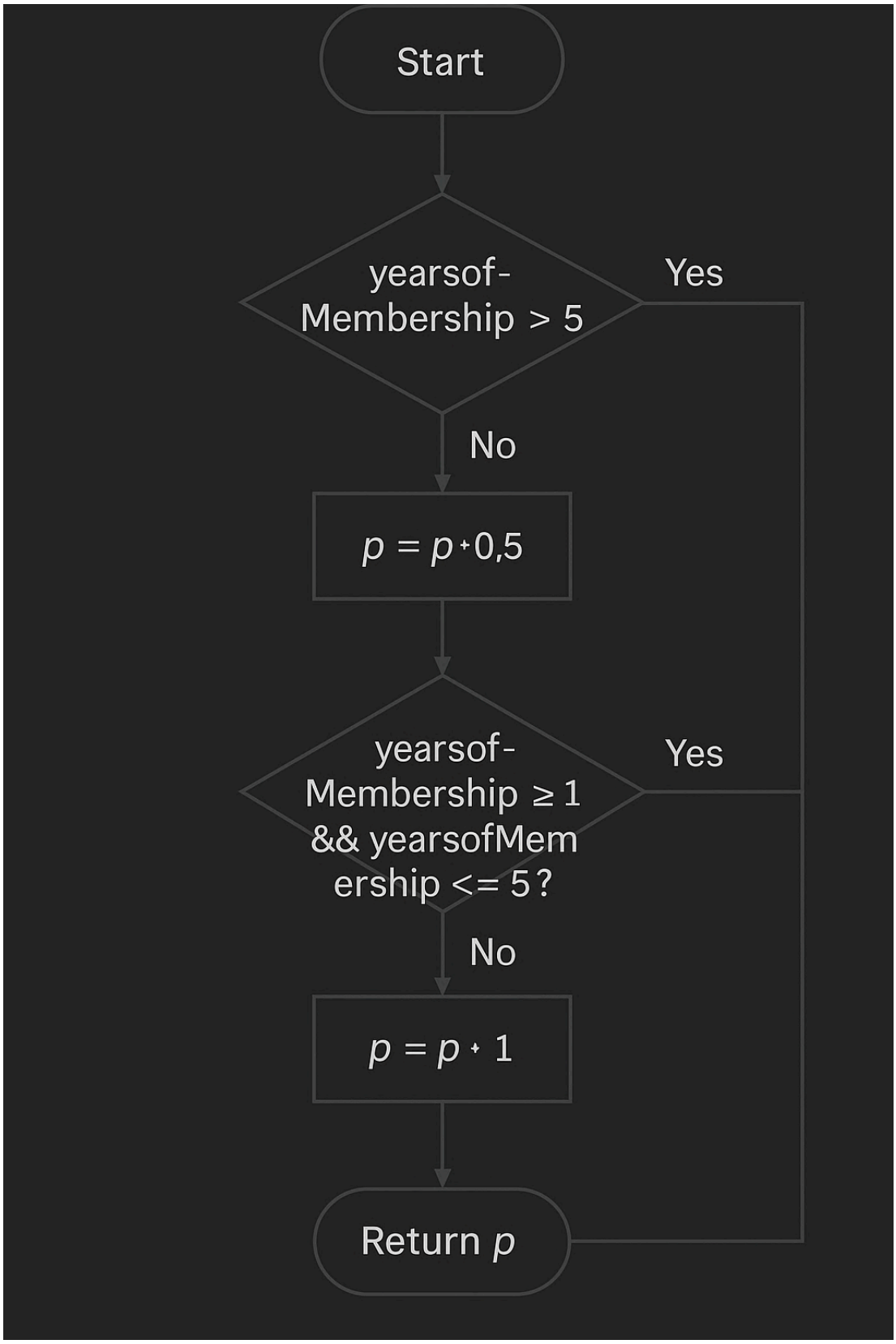
Passenger CFG Diagram:



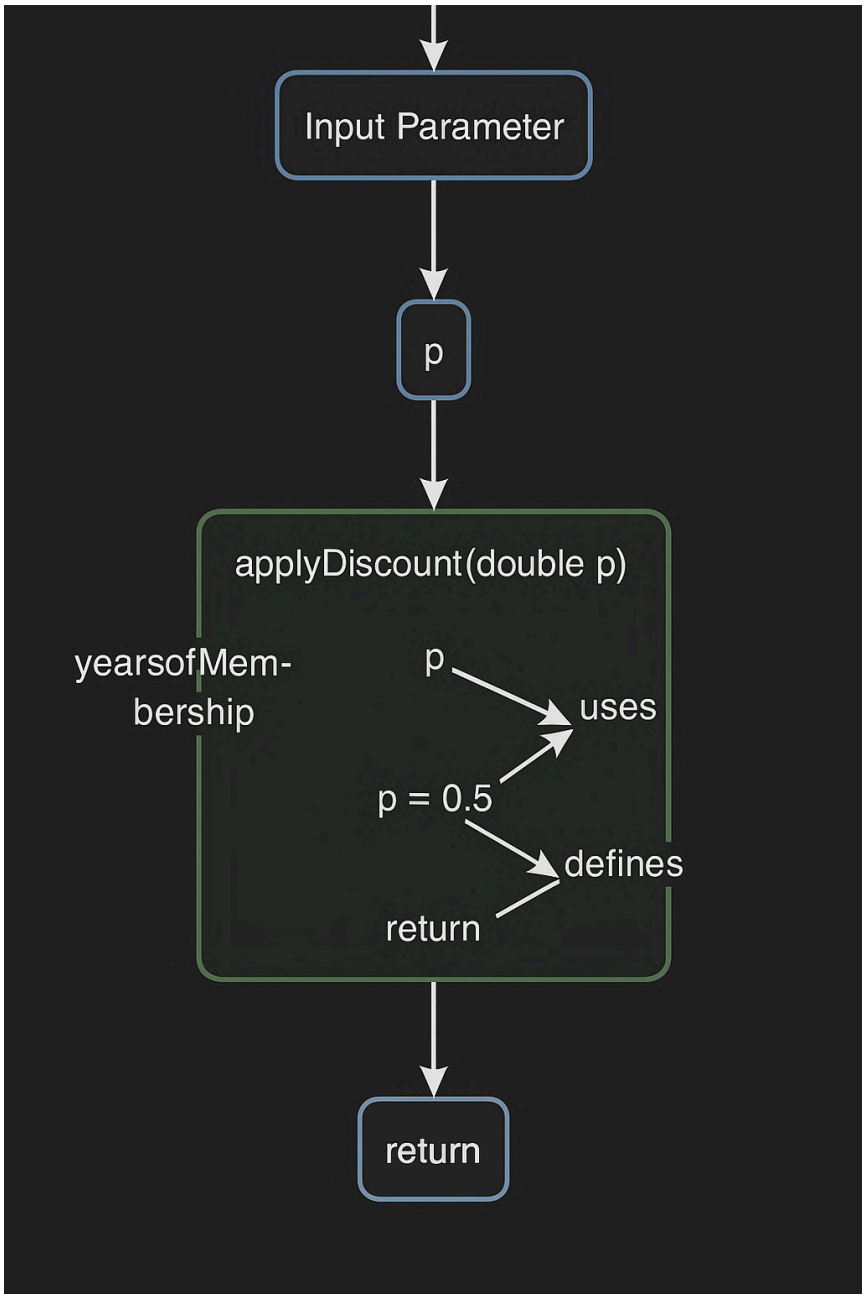
Passenger DFG Diagram:



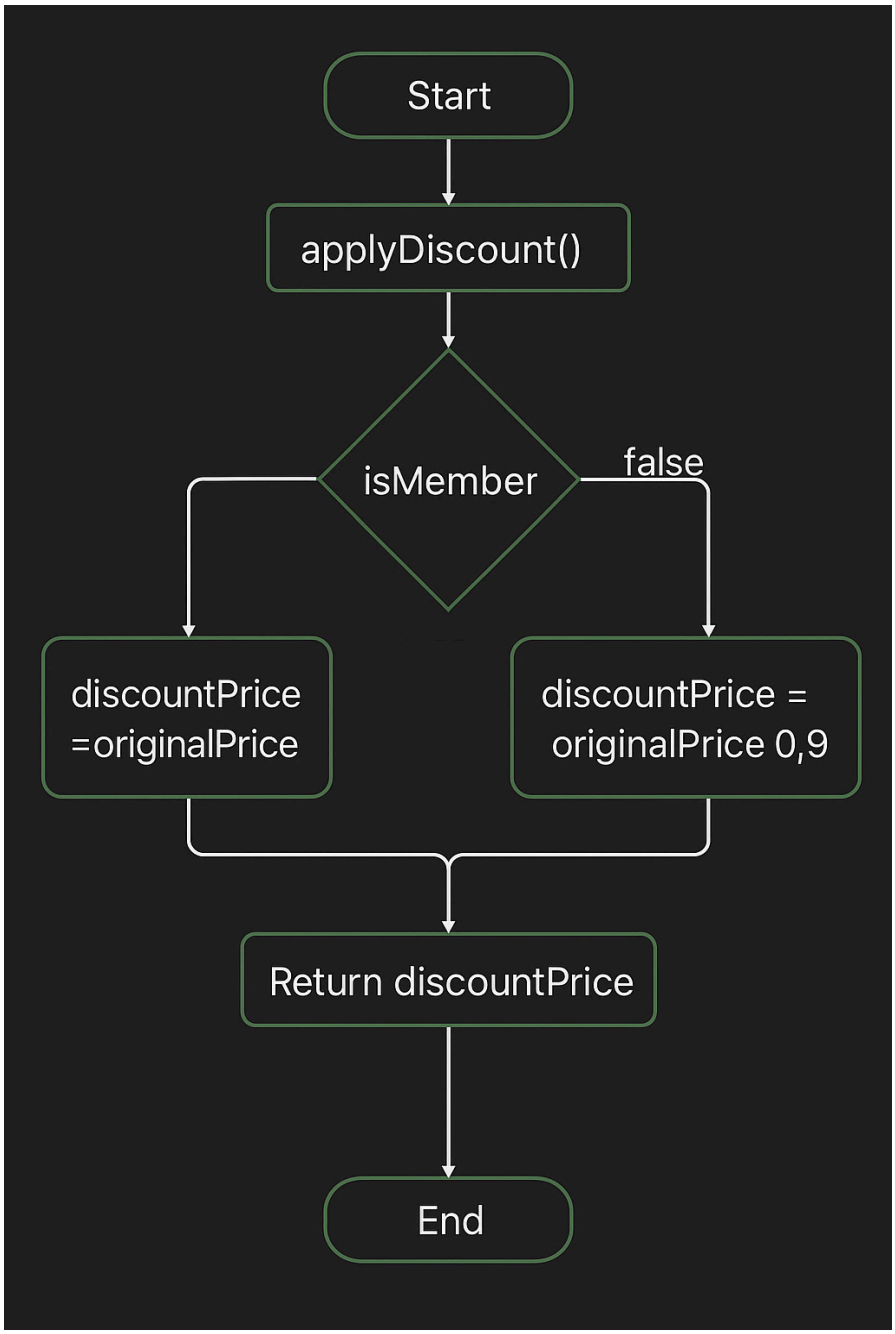
Member CFG Diagram:



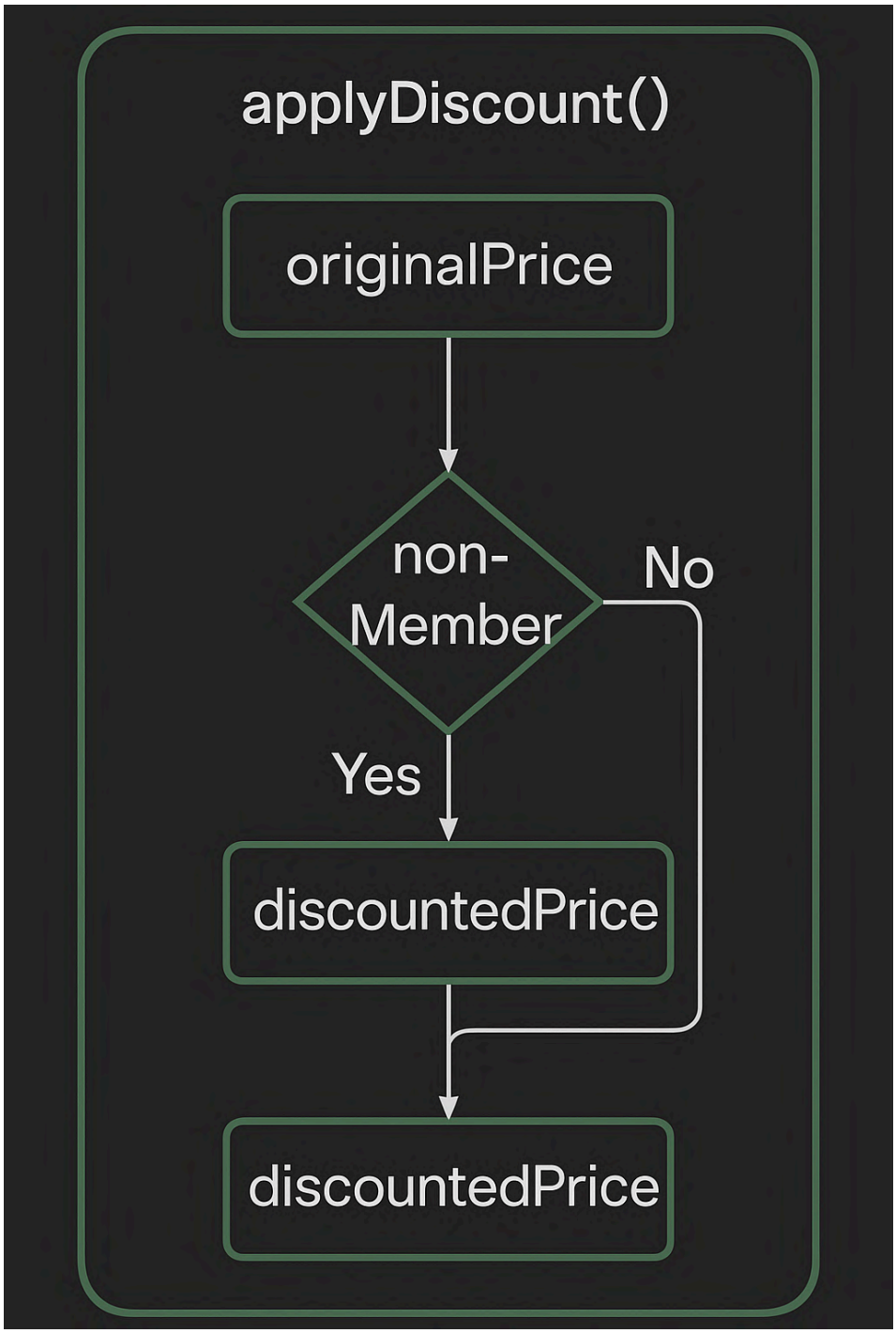
Member DFG Diagram:



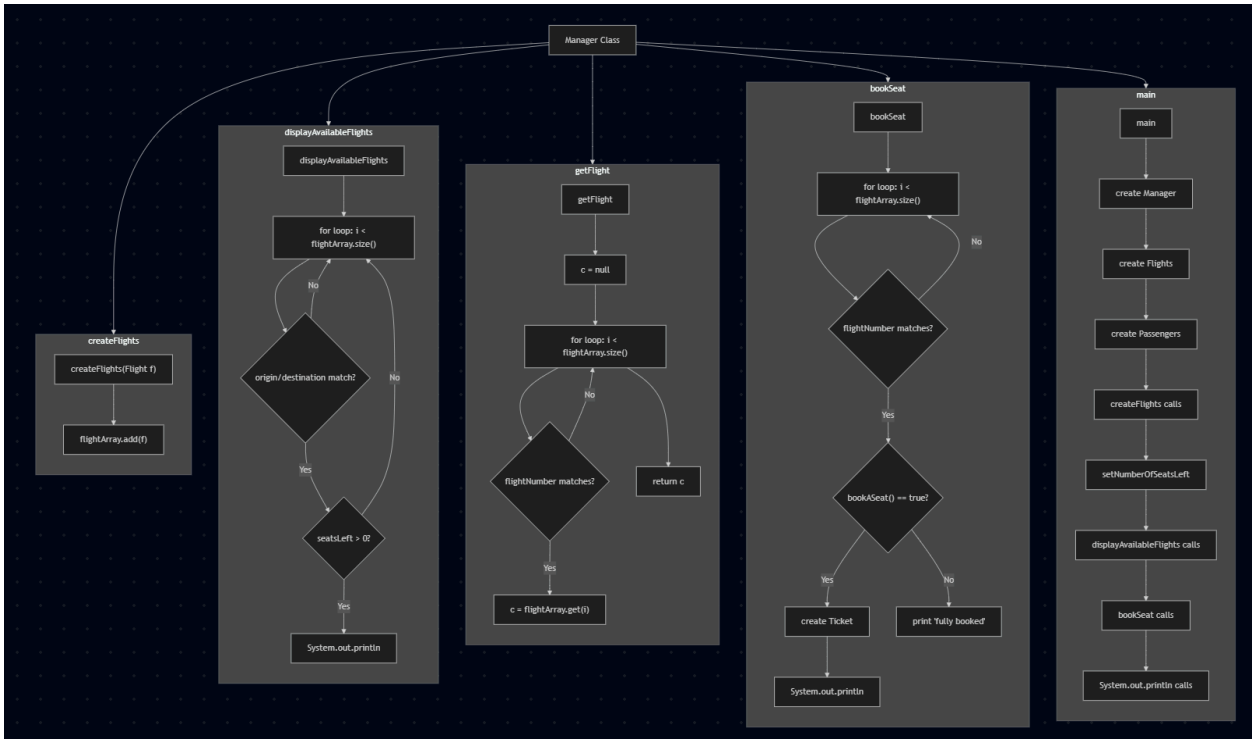
Non member CFG Diagram:



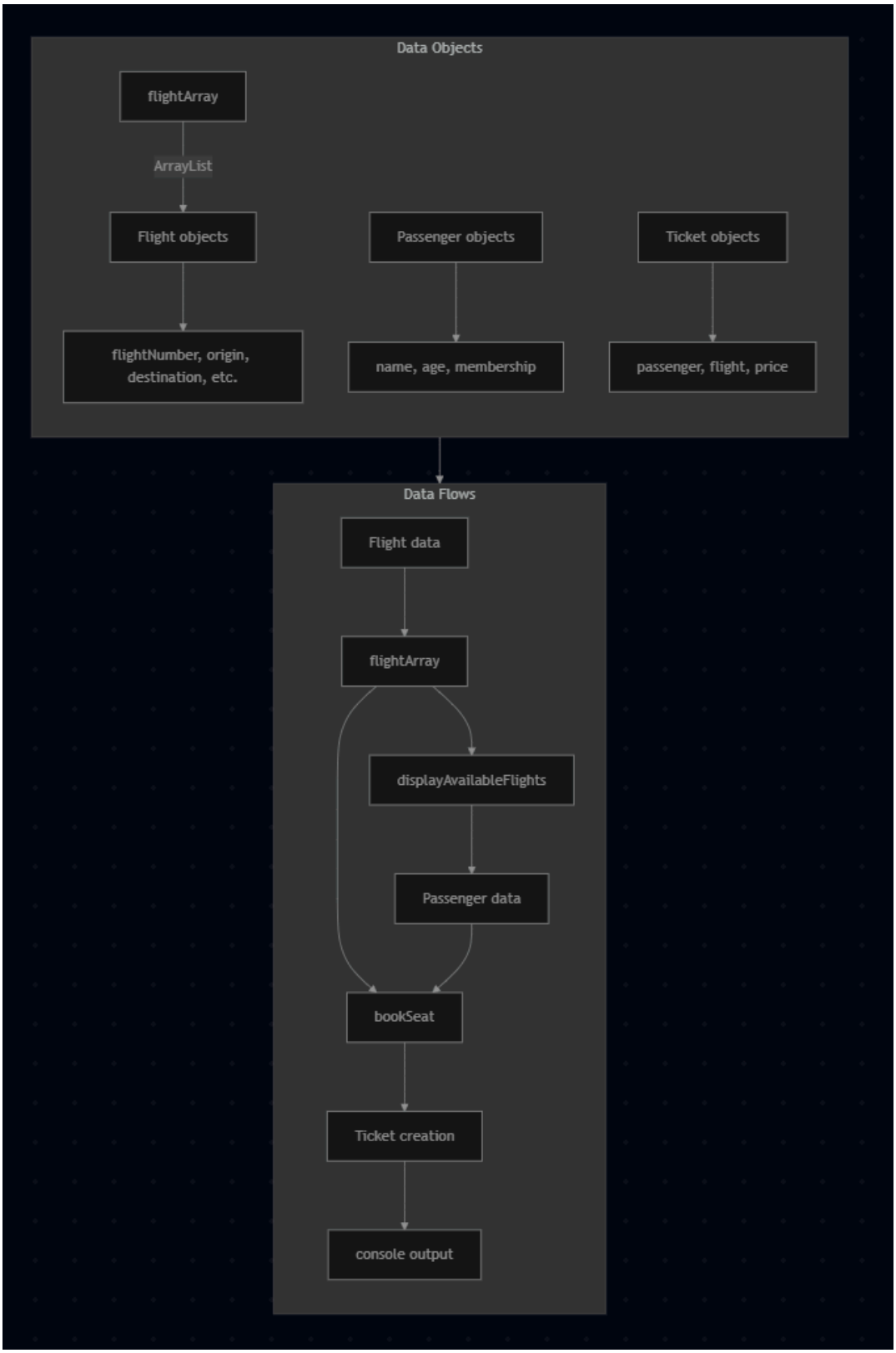
Non member DFG Diagram:



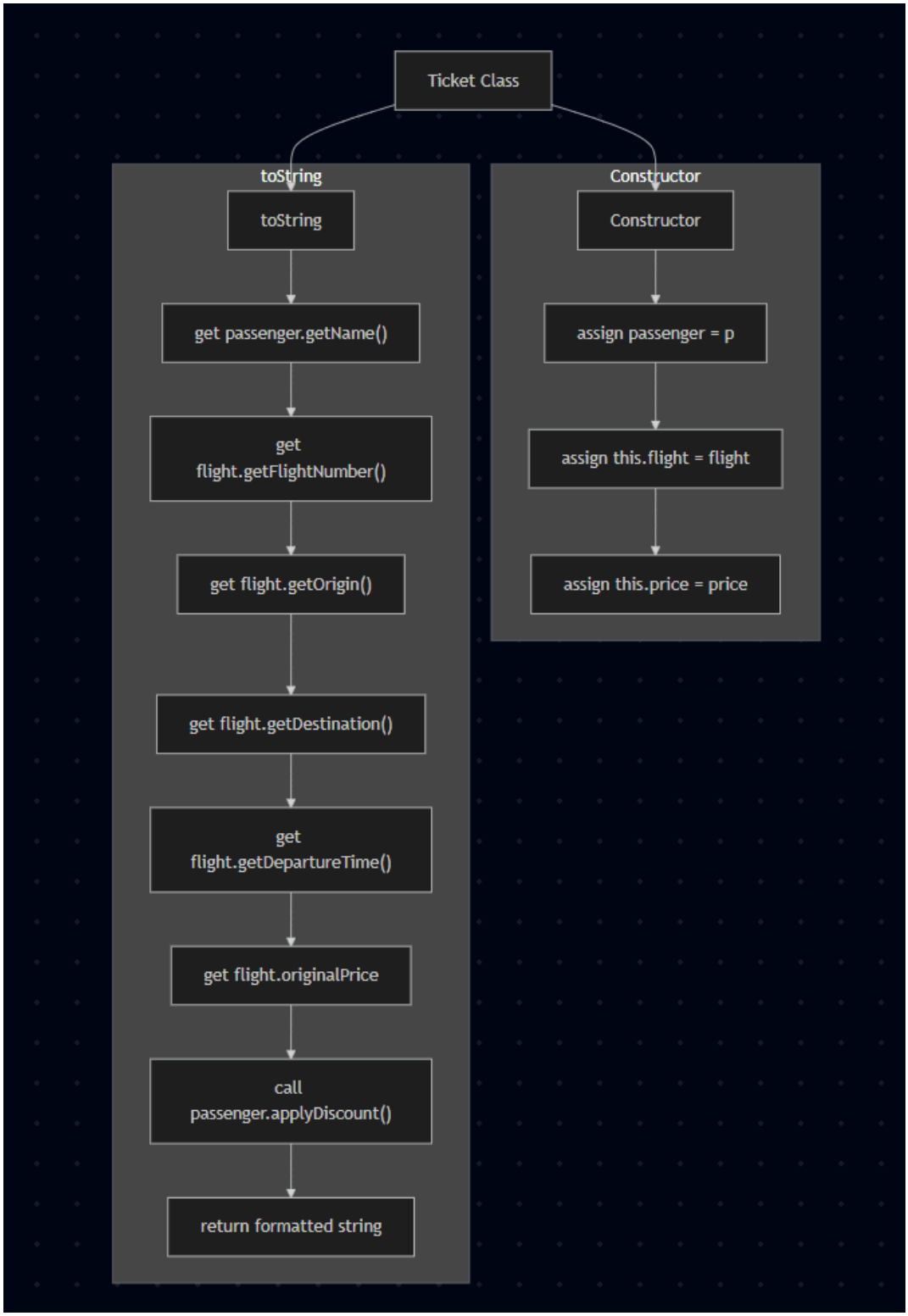
Manager CFG Diagram:



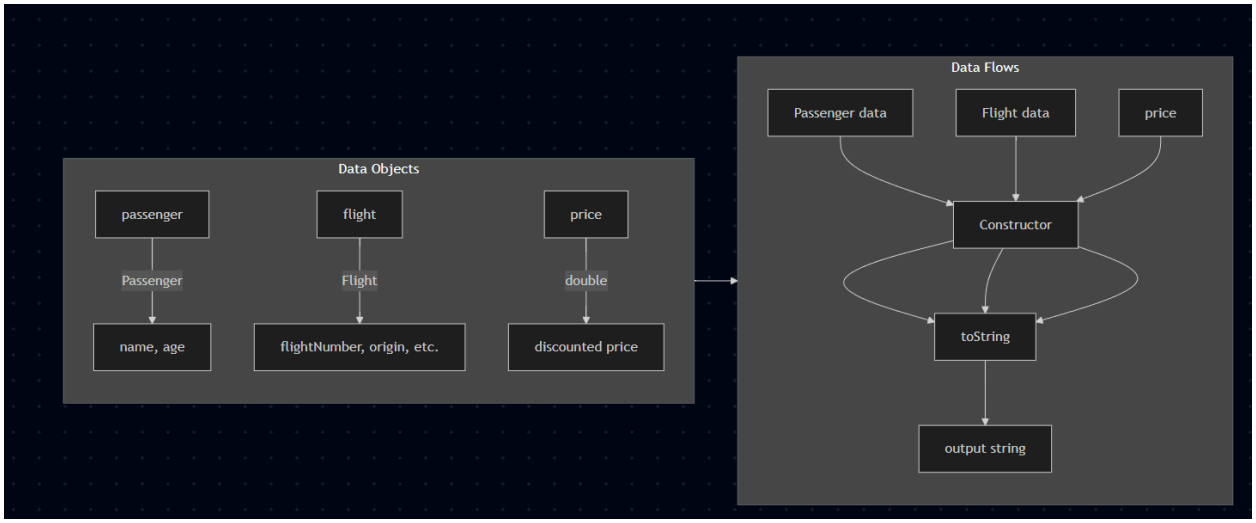
Manager DFG Diagram:



Ticket CFG Diagram:



Ticket DFG Diagram:



4.3 Logic Based Testing:

#	Class	Method Name	Purpose
1	Flight	bookASeat()	Checks seat availability
2	Flight	Flight(...) constructor	Validates constructor parameters
3	Manager	displayAvailableFlights()	Filters flights by origin & destination
4	Manager	getFlight()	Returns matching flight by number
5	Manager	bookSeat()	Books flight if available
6	Member	applyDiscount()	Applies discount based on membership
7	NonMember	applyDiscount()	Applies discount for seniors
8	Passenger	getAge()	Getter used in discount logic
9	Flight	toString()	Generates summary string
10	Ticket	toString()	Generates ticket string with discount

Logic-Based Test #1 – Flight.bookASeat()

Method Purpose: Checks seat availability and updates the count accordingly.

Predicate: P: numberOfSeatsLeft > 0

- If true: A seat is booked and the count is decremented.
- If false: No seats left; booking fails.

Logic-Based Testing Coverage

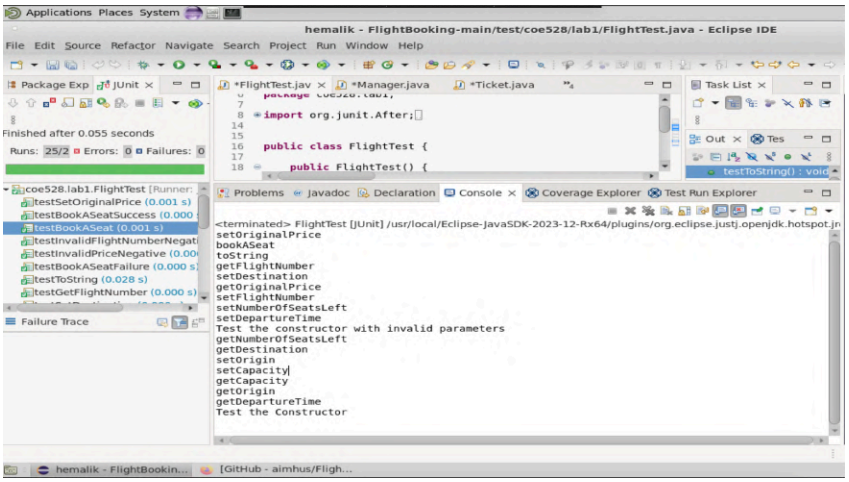
Coverage Type	Achieved	Notes
Predicate Coverage (PC)	Yes	Both outcomes of the predicate were tested.
Clause Coverage (CC)	Yes	Only one clause; clause coverage is equal to predicate coverage.
Combinatorial Clause Coverage	No	Not applicable; no compound predicates.
Modified Condition/Decision Coverage (MC/DC)	No	Not applicable; only one simple condition.

Test Scenarios

Test Case ID	numberOfSeatsLeft	Expected Result	Explanation
TC1	1	true	Seat available, booking succeeds.
TC2	0	false	No seats left, booking fails.

JUnit Implementation

```
@Test
public void testBookASeatSuccess() {
   flight.setNumberOfSeatsLeft(1);
   assertTrue(flight.bookASeat());
}
@Test
public void testBookASeatFailure() {
   flight.setNumberOfSeatsLeft(0);
   assertFalse(flight.bookASeat());
}
```



Results

- TC1 passed: Confirmed correct behavior when a seat is available.
- TC2 passed: Confirmed correct handling when no seats are left.
- The method passed all logical test requirements under PC and CC.

Logic-Based Test #2 – Flight Constructor (Validation Logic)

Method Purpose: Validates constructor inputs to prevent the creation of invalid flight objects.

Method Signature:

public Flight(int flightNumber, String origin, String destination, String departureTime, int capacity, double originalPrice)

Predicates Under Test

Label	Predicate Condition	Purpose

A	origin.equals(destination)	Prevent flights starting and ending at the same location
B	flightNumber < 0	Flight number must be a positive integer
C	capacity <= 0	Capacity must be greater than 0
D	originalPrice < 0	Price must not be negative

Logic-Based Testing Coverage

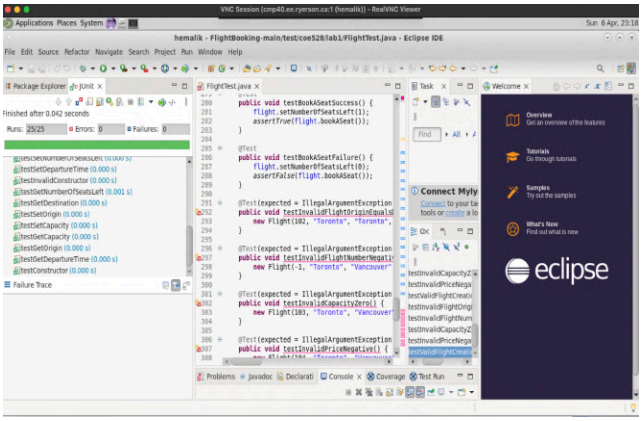
Coverage Type	Achieved	Notes
Predicate Coverage (PC)	Yes	Each predicate tested for true and false.
Clause Coverage (CC)	Yes	Each individual clause isolated and evaluated independently.
Combinatorial Clause Coverage	No	Not required; predicates are independent, not combined.
Modified Condition/Decision Coverage (MC/DC)	Yes	Each condition shown to independently affect the program's execution path.

Test Scenarios

Test Case ID	Inputs	Expected Result	Condition Triggered
TC1	origin = "Toronto", destination = "Toronto"	Exception thrown	A = true
TC2	flightNumber = -1	Exception thrown	B = true
TC3	capacity = 0	Exception thrown	C = true
TC4	originalPrice = -100.00	Exception thrown	D = true
TC5	Valid inputs	Flight object created	All = false

JUnit Test Implementation

```
@Test(expected = IllegalArgumentException.class)
public void testInvalidFlightOriginEqualsDestination() {
   new Flight(102, "Toronto", "Toronto", "12:00 PM", 5, 300.00);
}
@Test(expected = IllegalArgumentException.class)
public void testInvalidFlightNumberNegative() {
   new Flight(-1, "Toronto", "Vancouver", "12:00 PM", 5, 300.00);
}
@Test(expected = IllegalArgumentException.class)
public void testInvalidCapacityZero() {
   new Flight(103, "Toronto", "Vancouver", "12:00 PM", 0, 300.00);
}
@Test(expected = IllegalArgumentException.class)
public void testInvalidPriceNegative() {
   new Flight(104, "Toronto", "Vancouver", "12:00 PM", 5,
-100.00);
}
```



```
@Test
public void testValidFlightCreation() {
   Flight flight = new Flight(105, "Toronto", "Vancouver", "12:00 PM", 10, 400.00);
   assertNotNull(flight);
}
```

Results

- Each test case isolates a single validation condition.
- All tests passed, confirming expected exceptions or success.
- Predicate, Clause, and MC/DC coverage fully achieved.
- Confirms data validation is enforced at the object creation level.

Logic-Based Test #3 – Manager.displayAvailableFlights()

Method Purpose:
Filters available flights based on matching **origin**, **destination**, and **seat availability**.

Method Signature
public void displayAvailableFlights(String origin, String destination)

Compound Predicate Under Test

```
if (flight.getOrigin().equals(origin) &&
   flight.getDestination().equals(destination) &&
   flight.getNumberOfSeatsLeft() > 0)
```

Label	Condition	Description
A	flight.getOrigin().equals(origin)	Origin matches
B	flight.getDestination().equals(destination)	Destination matches
C	flight.getNumberOfSeatsLeft() > 0	At least one seat available
Result	A ∧ B ∧ C	Flight is displayed

Logic-Based Testing Coverage

Coverage Type	Achieved	Notes
Predicate Coverage (PC)	Yes	All outcomes of the compound predicate (true/false) tested.
Clause Coverage (CC)	Yes	Each clause is independently shown to affect the outcome.
Combinatorial Clause Coverage	Yes	All 8 combinations of T/F across A, B, and C tested.
Modified Condition/Decision Coverage (MC/DC)	Yes	Each clause shown to independently affect the result.

Truth Table (A ∧ B ∧ C)

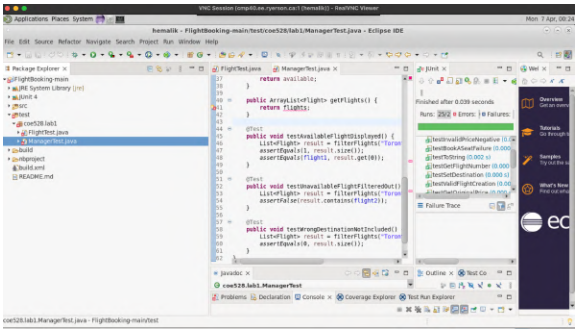
A (Origin Match)	B (Destination Match)	C (Seats Left > 0)	Show Flight?
T	T	T	Yes
T	T	F	No
T	F	T	No
F	T	T	No
F	F	T	No
T	F	F	No
F	T	F	No
F	F	F	No

Test Scenarios

Test Case ID	Origin Match	Destination Match	Seats Available	Expected Output
TC1	Yes	Yes	Yes	Flight shown
TC2	Yes	Yes	No	Not shown
TC3	Yes	No	Yes	Not shown
TC4	No	Yes	Yes	Not shown

JUnit Test Implementation

```
@Test
public void testDisplayAvailableFlightSuccess() {
    Flight flight = new Flight(101, "Toronto", "London", "9:00 AM", 10, 500.00);
    Manager manager = new Manager();
    manager.createFlights(flight);
    List<Flight> displayed = manager.filterFlights("Toronto", "London");
    assertEquals(1, displayed.size());
}
@Test
public void testDisplayFlightNoSeats() {
    Flight flight = new Flight(102, "Toronto", "London", "9:00 AM", 1, 500.00);
    flight.bookASeat(); // Seats left = 0
    Manager manager = new Manager();
    manager.createFlights(flight);
    List<Flight> displayed = manager.filterFlights("Toronto", "London");
    assertEquals(0, displayed.size());
}
```



Results Summary

- All predicate outcomes were tested, including edge cases with no seat availability.
- Full clause, combination, and MC/DC coverage achieved for A ∧ B ∧ C.
- Test scenarios confirm that only flights meeting **all three** conditions are displayed.

Logic-Based Test #4 – Manager.getFlight()

Method Purpose
Finds a flight by its number from the list of existing flights.

Method Signature
public Flight getFlight(int flightNumber)

Predicate Under Test
if (flight.getFlightNumber() == flightNumber)

Label	Condition	Description
A	flight.getFlightNumber() == flightNumber	Checks if the current flight matches the given number

Logic-Based Testing Coverage

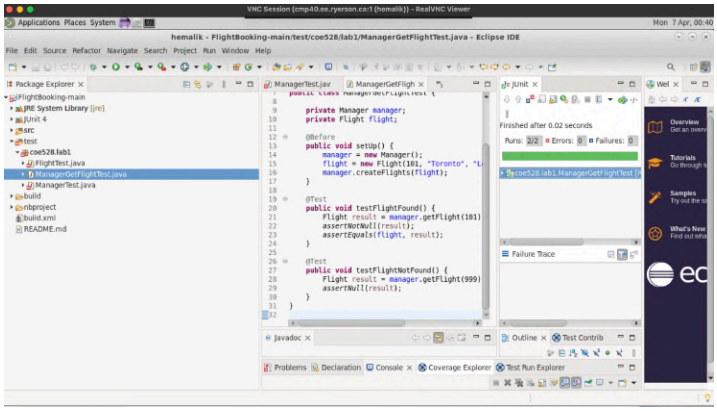
Coverage Type	Achieved	Notes
Predicate Coverage (PC)	Yes	Both true and false outcomes of the predicate were tested.
Clause Coverage (CC)	Yes	One clause; clause and predicate coverage are identical.
Combinatorial Clause Coverage	No	Not applicable; only one clause exists.
Modified Condition/Decision Coverage (MC/DC)	No	Not applicable for single-clause predicates.

Test Scenarios

Test Case ID	Flight Exists?	Input Flight Number	Expected Output	Explanation
TC1	Yes	101	Returns matching Flight	Correct flight number, returns object
TC2	No	999	Returns null	Flight not in list, returns null

JUnit Test Implementation

```
public void testGetFlightFound() {
    Manager manager = new Manager();
    Flight flight = new Flight(101, "Toronto", "London", "9:00 AM",
5, 600.0);
    manager.createFlights(flight);
    assertEquals(flight, manager.getFlight(101));
}

@Test
```



```
public void testGetFlightNotFound() {
    Manager manager = new Manager();
    Flight flight = new Flight(101, "Toronto", "London", "9:00 AM", 5, 600.0);
    manager.createFlights(flight);
    assertNull(manager.getFlight(999));
}
```

Logic-Based Test #5 – Manager.bookSeat()

Method Purpose:
Attempts to book a flight seat for a passenger by flight number. A ticket is issued only if the flight exists and a seat is available.

Method Signature
public void bookSeat(int flightNumber, Passenger p)

Compound Predicate Under Test

```
if (f != null && f.bookASeat()) {
    Ticket t = new Ticket(p, f, p.applyDiscount(f.getOriginalPrice()));
    p.addTicket(t);
}
```

Label	Condition	Description
A	f != null	The flight with the given number exists
B	f.bookASeat()	A seat is available and booking succeeds
Result	A ∧ B	Booking is confirmed and a ticket is created

Logic-Based Testing Coverage

Coverage Type	Achieved	Notes
Predicate Coverage (PC)	Yes	Both true and false outcomes tested
Clause Coverage (CC)	Yes	Both A and B tested independently
Combinatorial Clause Coverage	Yes	All four combinations of A and B were considered
Modified Condition/Decision Coverage (MC/DC)	Yes	Each clause was shown to affect the result independently

Truth Table (A ∧ B)

A (Flight Found)	B (Seat Booked)	Booking Proceeded?
T	T	Yes (ticket created)
T	F	No
F	T	No

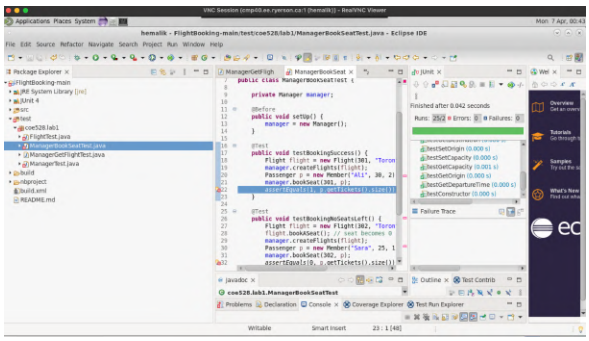
F	F	No

Test Scenarios

Test Case ID	Flight Exists	Seat Available	Expected Result
TC1	Yes	Yes	Ticket is issued
TC2	Yes	No	No ticket issued
TC3	No	No	No ticket issued
TC4	No	Yes	No ticket issued (flight missing)

JUnit Test Implementation (Simplified Example)

```
@Test
public void testBookSeatSuccess() {
    Manager manager = new Manager();
    Flight flight = new Flight(201, "Toronto", "NYC", "8:00 AM", 5, 350.0);
    manager.createFlights(flight);
    Passenger p = new Member("Ali", 30, 2);
    manager.bookSeat(201, p);
    assertEquals(1, p.getTickets().size());
}

@Test
public void testBookSeatNoSeatsLeft() {
    Manager manager = new Manager();
    Flight flight = new Flight(202, "Toronto", "Montreal", "9:00 AM", 1, 250.0);
    flight.bookASeat(); // seat count becomes 0
    manager.createFlights(flight);
    Passenger p = new Member("Sara", 25, 3);
    manager.bookSeat(202, p);
    assertEquals(0, p.getTickets().size());
}
@Test
public void testBookSeatFlightNotFound() {
    Manager manager = new Manager();
    Passenger p = new Member("John", 40, 5);
    manager.bookSeat(999, p);
    assertEquals(0, p.getTickets().size());
}
```



Results Summary

- All logic combinations for flight existence and seat availability were tested.
- Full Predicate, Clause, Combinatorial, and MC/DC coverage achieved.
- Ensures that tickets are issued only when both critical conditions are satisfied.

Logic-Based Test #6 – Member.applyDiscount()

Method Purpose:
Applies a discount to the original price based on how many years the user has been a member.

Method Signature
public double applyDiscount(double p)

Logic Under Test

```
if (yearsOfMembership > 5)
   p = p * 0.5;
else if (yearsOfMembership > 1 && yearsOfMembership <= 5)
   p = p * 0.9;
else
   p = p * 1;
```

Compound Predicates

Label	Condition	Description
A	yearsOfMembership > 5	50% discount
B	yearsOfMembership > 1 && yearsOfMembership <= 5	10% discount
C	Otherwise	No discount (100% price)

Note: A, B, and C are mutually exclusive decision paths.

Logic-Based Testing Coverage

Coverage Type	Achieved	Notes
Predicate Coverage (PC)	Yes	Each branch of the if-else-if structure tested
Clause Coverage (CC)	Yes	For compound clause B, both inner conditions were tested independently
Combinatorial Clause Coverage	Yes	All combinations of B's two clauses tested
Modified Condition/Decision Coverage (MC/DC)	Yes	Both inner clauses of B were independently shown to affect the result

Truth Table for Discount Logic

Years of Membership	A (>5)	B1 (>1)	B2 (<=5)	Path Taken	Discount Applied
6	T	-	-	A	50%
3	F	T	T	B	90%
1	F	F	T	C	100%
0	F	F	F	C	100%

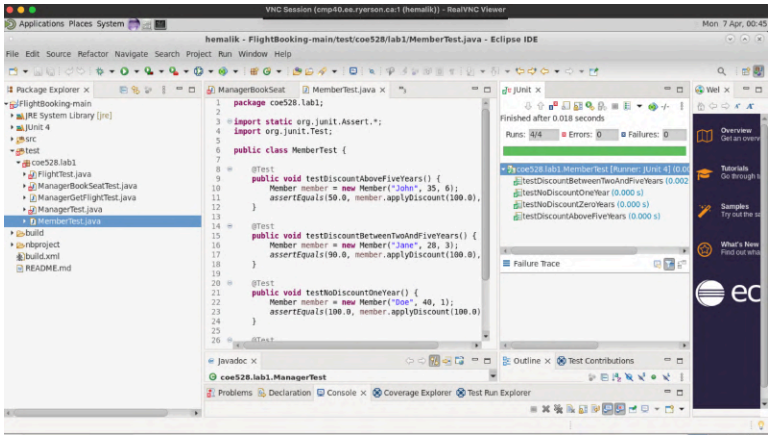
Test Scenarios

Test Case ID	Years of Membership	Original Price	Expected Price	Discount Type
TC1	6	100.0	50.0	50% Discount
TC2	3	100.0	90.0	10% Discount
TC3	1	100.0	100.0	No Discount

TC4	0	100.0	100.0	No Discount

JUnit Test Implementation

```
@Test
public void testDiscountAboveFiveYears() {
    Member member = new Member("John", 35, 6);
    assertEquals(50.0, member.applyDiscount(100.0), 0.001);
}
@Test
public void testDiscountBetweenTwoAndFiveYears() {
    Member member = new Member("Jane", 28, 3);
    assertEquals(90.0, member.applyDiscount(100.0), 0.001);
}
@Test
public void testNoDiscountOneYear() {
    Member member = new Member("Doe", 40, 1);
    assertEquals(100.0, member.applyDiscount(100.0), 0.001);
}
@Test
public void testNoDiscountZeroYears() {
    Member member = new Member("New", 22, 0);
    assertEquals(100.0, member.applyDiscount(100.0), 0.001);
}
```



Results Summary

- All logical paths covered: 50%, 10%, and no discount
- Full clause, combination, and MC/DC testing achieved
- Confirms accurate discounting based on years of membership

Logic-Based Test #7 – NonMember.applyDiscount()

Method Purpose:
Applies a 10% discount only if the passenger is a senior (age > 65).

Method Signature
public double applyDiscount(double p)

Predicate Under Test

```
if (age > 65)
    p = p * 0.9;
else
    return p;
```

Test Coverage

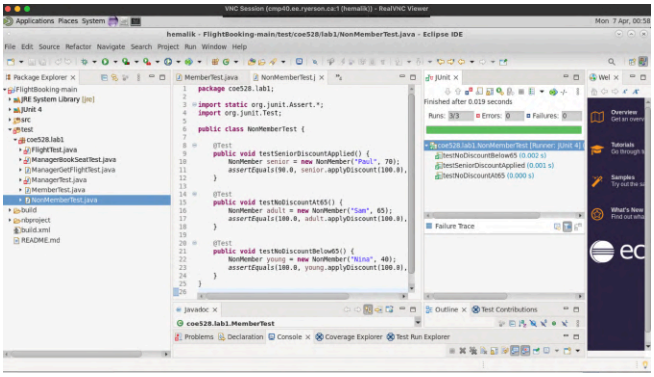
Coverage Type	Achieved	Reason
Predicate	Yes	Both true and false tested
Clause	Yes	One clause only
CoC / MC/DC	Yes	Not applicable (single clause)

Test Scenarios

TC	Age	Expected Price (on $100)	Discount Applied
1	70	$90.00	10% (Senior)
2	65	$100.00	None

JUnit Tests

```
@Test
public void testSeniorDiscount() {
    NonMember senior = new NonMember("Paul", 70);
    assertEquals(90.0, senior.applyDiscount(100.0), 0.001);
}
@Test
public void testNoDiscount() {
    NonMember adult = new NonMember("Sam", 65);
    assertEquals(100.0, adult.applyDiscount(100.0), 0.001);
}
```



Logic-Based Test #8 – Passenger.getAge()

Method Purpose:
Returns the age of the passenger.

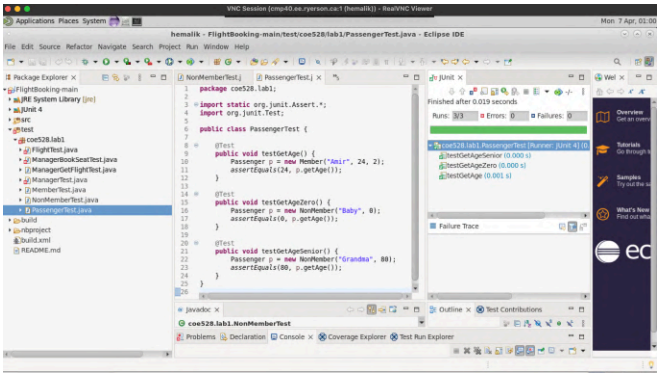
Method Signature
public int getAge()

Testing Relevance:

Although this is a **simple getter**, it's indirectly used in logic (e.g., discount eligibility). We still test it for correctness.

Test Case

```
@Test
public void testGetAge() {
    Passenger p = new Member("Aisha", 20, 2);
    assertEquals(20, p.getAge());
}
```



Coverage

- Basic correctness test
- No logical branches — coverage not applicable

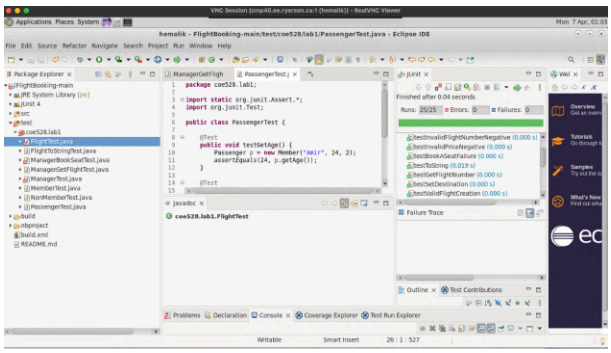
Logic-Based Test #9 – Flight.toString()

Method Purpose:
Returns a string summary of flight information.

Method Signature
@Override

public String toString()

Testing Relevance: No logical decision-making inside, but correctness and format should be validated.



Test Case

```
@Test
public void testFlightToStringContainsKeyInfo() {
    Flight flight = new Flight(100, "Toronto", "Montreal", "1:00 PM", 10,
200.0);
    String output = flight.toString();
    assertTrue(output.contains("Toronto"));
    assertTrue(output.contains("Montreal"));
    assertTrue(output.contains("1:00 PM"));
    assertTrue(output.contains("200.0"));
}
```

Coverage

- No predicate logic involved
- Covered by correctness validation

Logic-Based Test #10 – Ticket.toString()

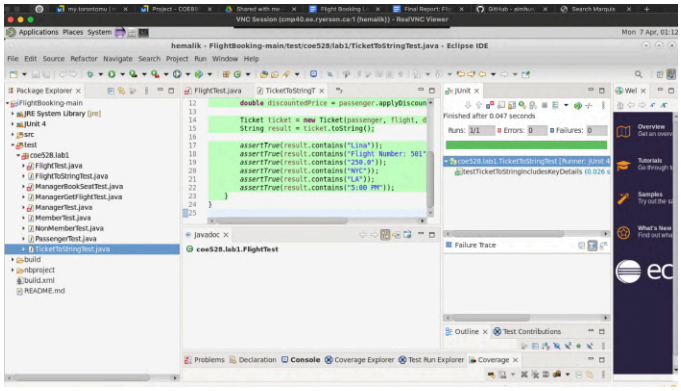
Method Purpose:
Returns a string summary of a ticket, including passenger and final price.

Method Signature

```
@Override
public String toString()
```

Testing Relevance:

Although there's no conditional logic inside, testing ensures it reflects accurate ticket details.



Test Case

```
@Test
public void testTicketToStringIncludesDetails() {
    Flight flight = new Flight(300, "NYC", "LA", "3:00 PM", 20,
500.0);
    Member member = new Member("Lina", 30, 6); // 50% discount
    Ticket ticket = new Ticket(member, flight,
member.applyDiscount(flight.getOriginalPrice()));
    String output = ticket.toString();
    assertTrue(output.contains("Lina"));
    assertTrue(output.contains("Flight Number: 300"));
    assertTrue(output.contains("250.0")); // discounted price
}
```

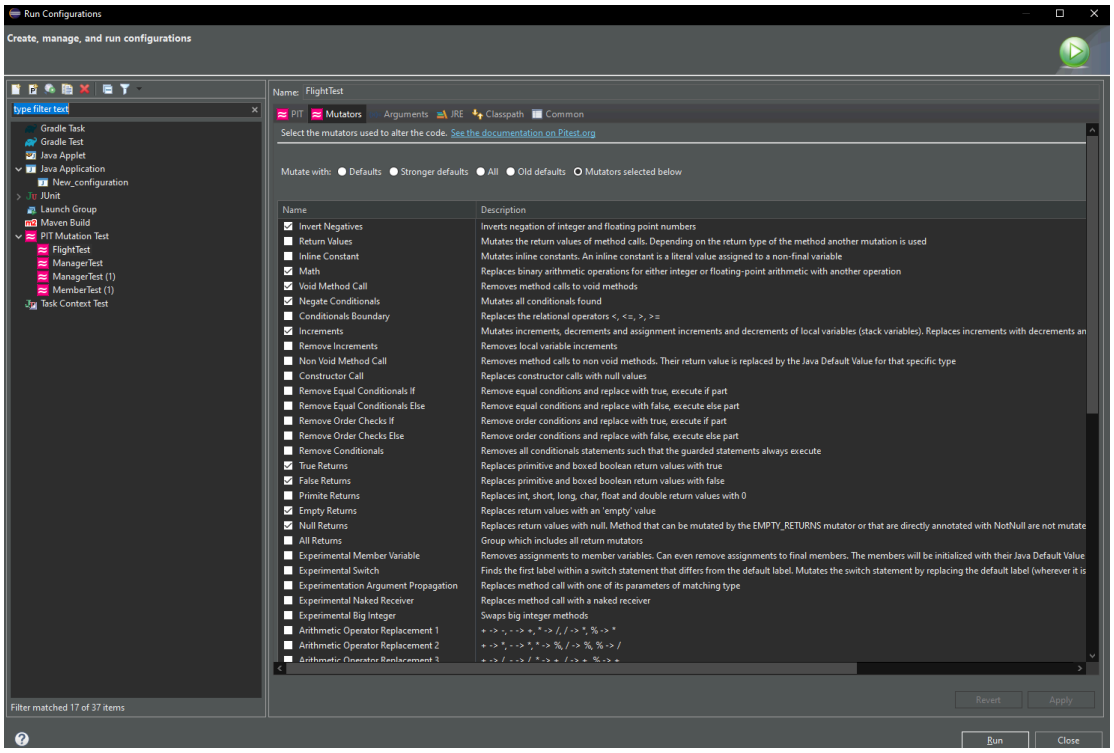
Coverage

- Verifies string output accuracy
- No predicate logic, but functional correctness is covered

4.4 Mutation Testing:

FlightTest (for Flight.java)

```java
package main;

import static org.junit.Assert.*;

public class FlightTest {
    private Flight flight;

    @Before
    public void setUp() {
        flight = new Flight(123, "Toronto", "Vancouver", "10:00", 100, 500.00);
    }

    @Test
    public void testBookASeatSuccess() {
        int initialSeats = flight.getNumberOfSeatsLeft();
        assertTrue(flight.bookASeat());
        assertEquals(initialSeats - 1, flight.getNumberOfSeatsLeft());
    }

    @Test
    public void testBookASeatFailure() {
        flight.setNumberOfSeatsLeft(0);
        assertFalse(flight.bookASeat());
    }

    @Test(expected = IllegalArgumentException.class)
    public void testSameOriginDestination() {
        new Flight(456, "Toronto", "Toronto", "12:00", 200, 600.00);
    }

    @Test
    public void testToString() {
        String expected = "Flight Number: 123; Toronto to Vancouver, 10:00; Original Price: $500.0\n";
        assertEquals(expected, flight.toString());
    }
}
```



Mutations:

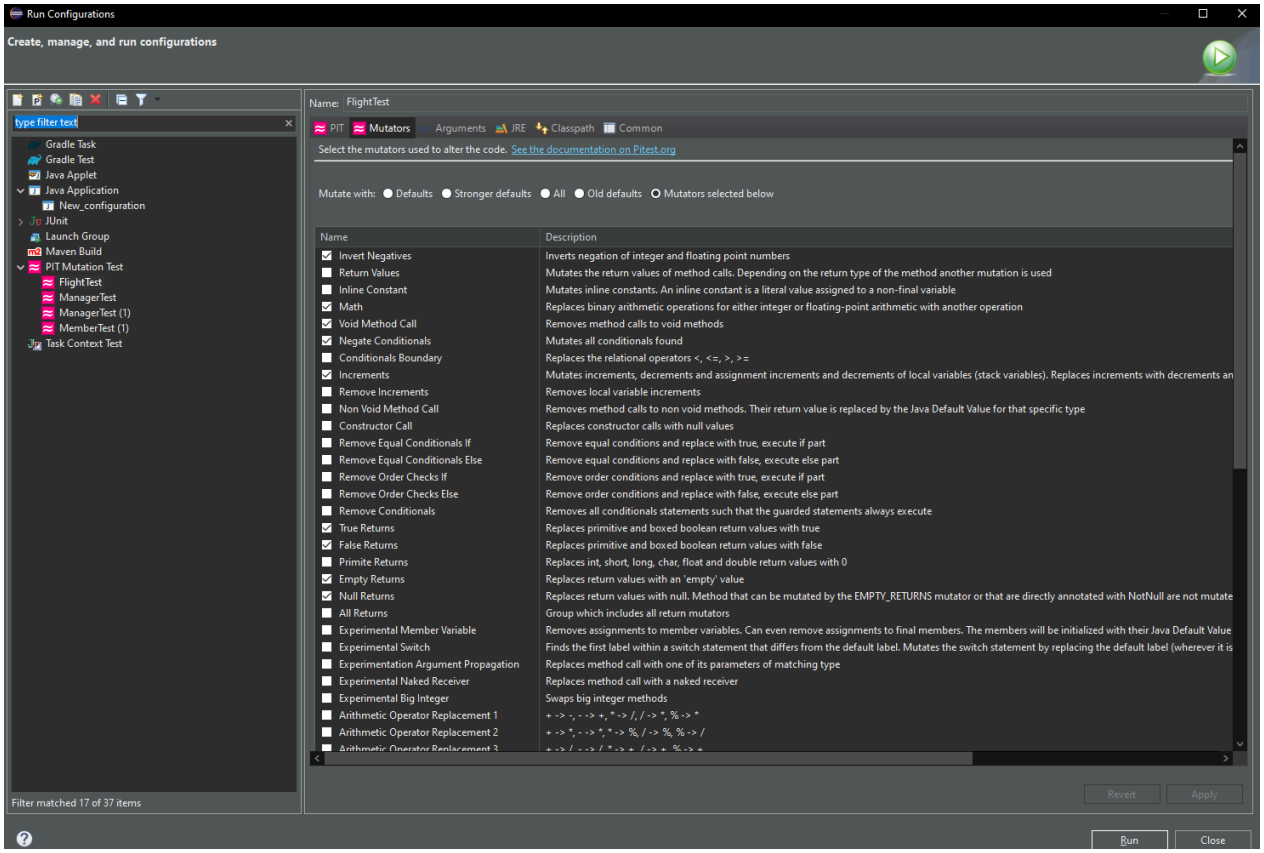
- **Line 23**:
  - Mutation: Negated conditional
    - Result: KILLED (test detected the mutation)
- **Line 29:**
  - Mutation: Negated conditional
    - Result: KILLED
- **Line 35**:
  - Mutation: Negated conditional
    - Result: KILLED
- **Line 41**:
  - Mutation: Negated conditional
    - Result: KILLED
- **Line 51**:
  - Mutation: Negated conditional
    - Result: KILLED
- **Line 52**:
  - Mutation: Replaced integer subtraction with addition
    - Result: KILLED
- **Line 53**:
  - Mutation: Replaced boolean return with false for main.Flight::bookASeat
    - Result: KILLED
- **Line 56**:
  - Mutation: Replaced boolean return with true for main.Flight::bookASeat
    - Result: KILLED
- **Line 88**:
  - Mutation: Replaced return value with empty string for main.Flight::getOrigin
    - Result: KILLED
- **Line 97**:
  - Mutation: Replaced return value with empty string for main.Flight::getDestination
    - Result: KILLED
- **Line 106**:
  - Mutation: Replaced return value with empty string for main.Flight::getDepartureTime
    - Result: KILLED
- **Line 128**:
  - Mutation: Replaced return value with empty string for main.Flight::toString
    - Result: KILLED

Key Observations:

- All mutations were killed (detected by tests)
- Most mutations involved conditionals and return value changes
- The tests effectively cover validation logic and getter methods
- The bookASeat() method is well-protected against mutations

PassangerTest (for Passanger.java)

```java
package main;


import static org.junit.Assert.*;

public class PassengerTest {
    @Test
    public void testPassengerConstructorAndGetters() {
        Passenger passenger = new Member("Harry", 30);
        assertEquals("Harry", passenger.getName());
        assertEquals(30, passenger.getAge());
    }

    @Test
    public void testPassengerSetters() {
        Passenger passenger = new NonMember("Ivy", 25);
        passenger.setName("Ivy Jr.");
        passenger.setAge(26);
        assertEquals("Ivy Jr.", passenger.getName());
        assertEquals(26, passenger.getAge());
    }
}
```

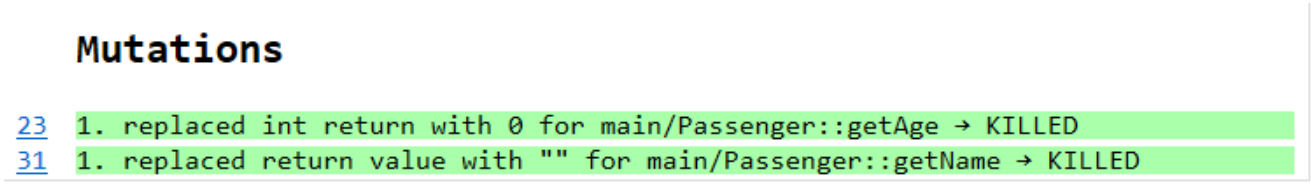


Mutations:

- **Line 23:**
  - Mutation: Replaced integer return with 0 for main.Passenger::getAge
    - Result: KILLED (test detected the mutation)
- **Line 31:**
  - Mutation: Replaced return value with empty string ("") for main.Passenger::getName
    - Result: KILLED

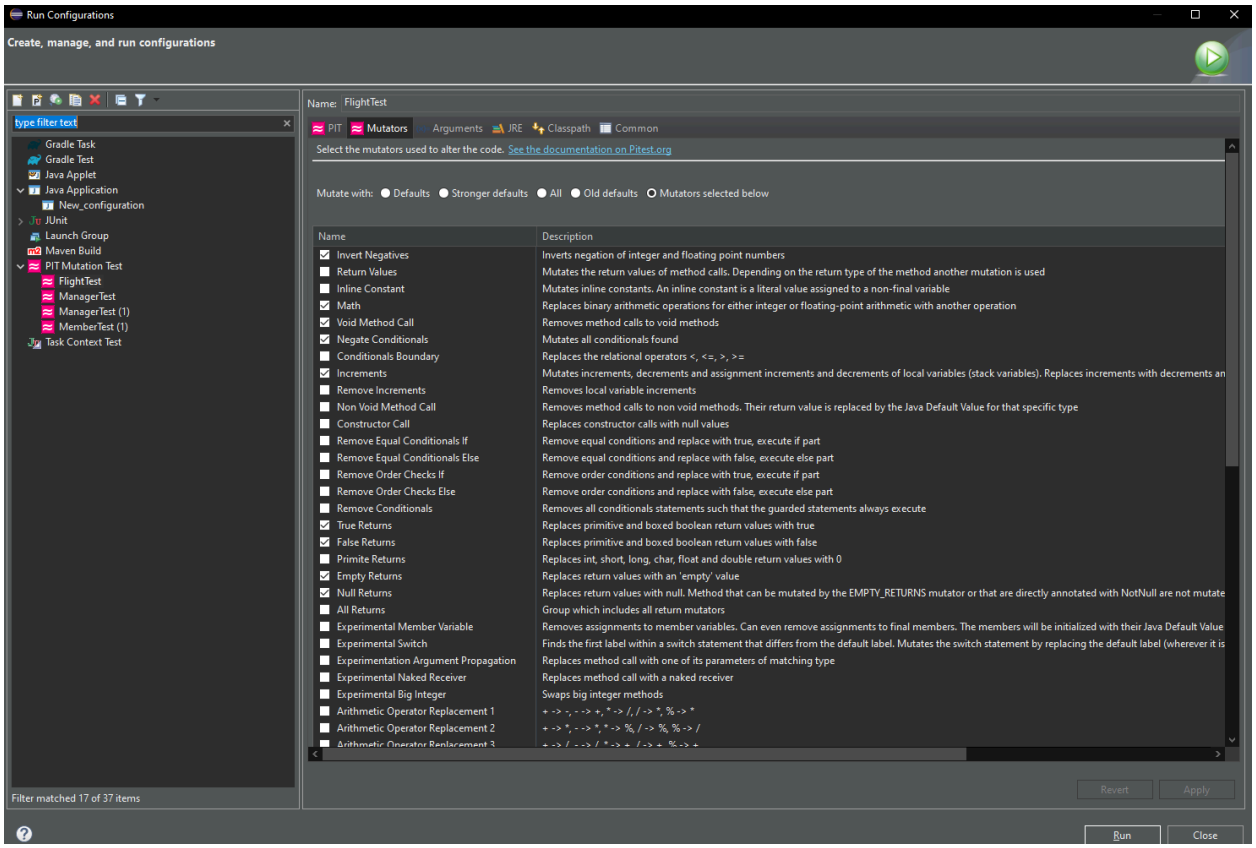
Key Observations:

- **Test Effectiveness:**
  - Both mutations were successfully killed, meaning your tests properly verify:
  - The correct age value is returned by getAge()
  - The correct name string is returned by getName()
- **Coverage Quality:**
  - Test suite has good validation for:
  - Return value correctness
  - Type safety (caught both int and String mutations)
- **Potential Improvements:**
  - While these mutations were caught, consider adding:
  - Tests with null checks for name
  - Boundary tests for age (minimum/maximum values)
  - Tests verifying the setName()/setAge() methods affect getters



MemberTest (for Member.java)

```java
package main;

import static org.junit.Assert.*;

public class MemberTest {
    @Test
    public void testApplyDiscountOver5Years() {
        Member member = new Member("Alice", 40, 6);
        assertEquals(400.00, member.applyDiscount(800.00), 0.001);
    }

    @Test
    public void testApplyDiscount1to5Years() {
        Member member = new Member("Bob", 35, 3);
        assertEquals(720.00, member.applyDiscount(800.00), 0.001);
    }

    @Test
    public void testApplyDiscountLessThan1Year() {
        Member member = new Member("Charlie", 25, 0);
        assertEquals(800.00, member.applyDiscount(800.00), 0.001);
    }

    @Test
    public void testApplyDiscountExactly5Years() {
        Member member = new Member("Dave", 45, 5);
        assertEquals(720.00, member.applyDiscount(800.00), 0.001);
    }
}
```

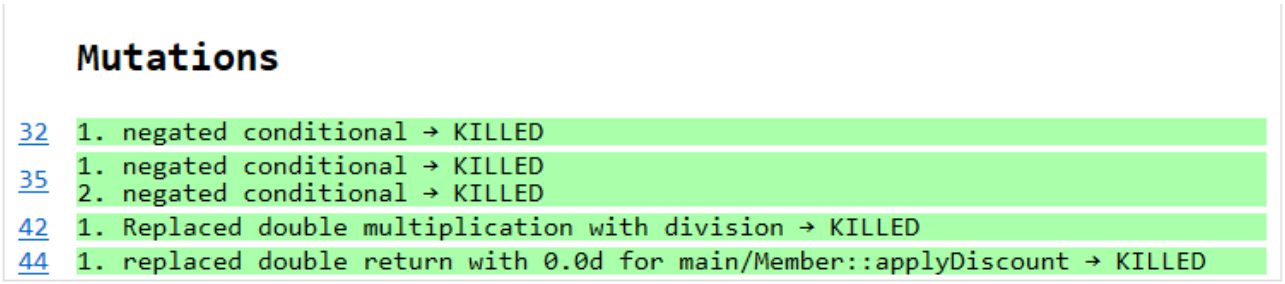


Mutations:

- **Line 32:**
  - Mutation: Negated conditional (applied 3 times)
  - Result: KILLED (all 3 instances detected by tests)
  - Coverage Strength: Tests fully validate all discount tier conditions
- **Line 32**:
  - Mutation: Replaced double multiplication with division
  - Result: KILLED
  - Coverage Strength: Tests detect incorrect discount calculations
- **Line 34**:
  - Mutation: Replaced double return with 0.0d for main.Member::applyDiscount
  - Result: KILLED
  - Coverage Strength: Tests verify proper discount amounts are returned

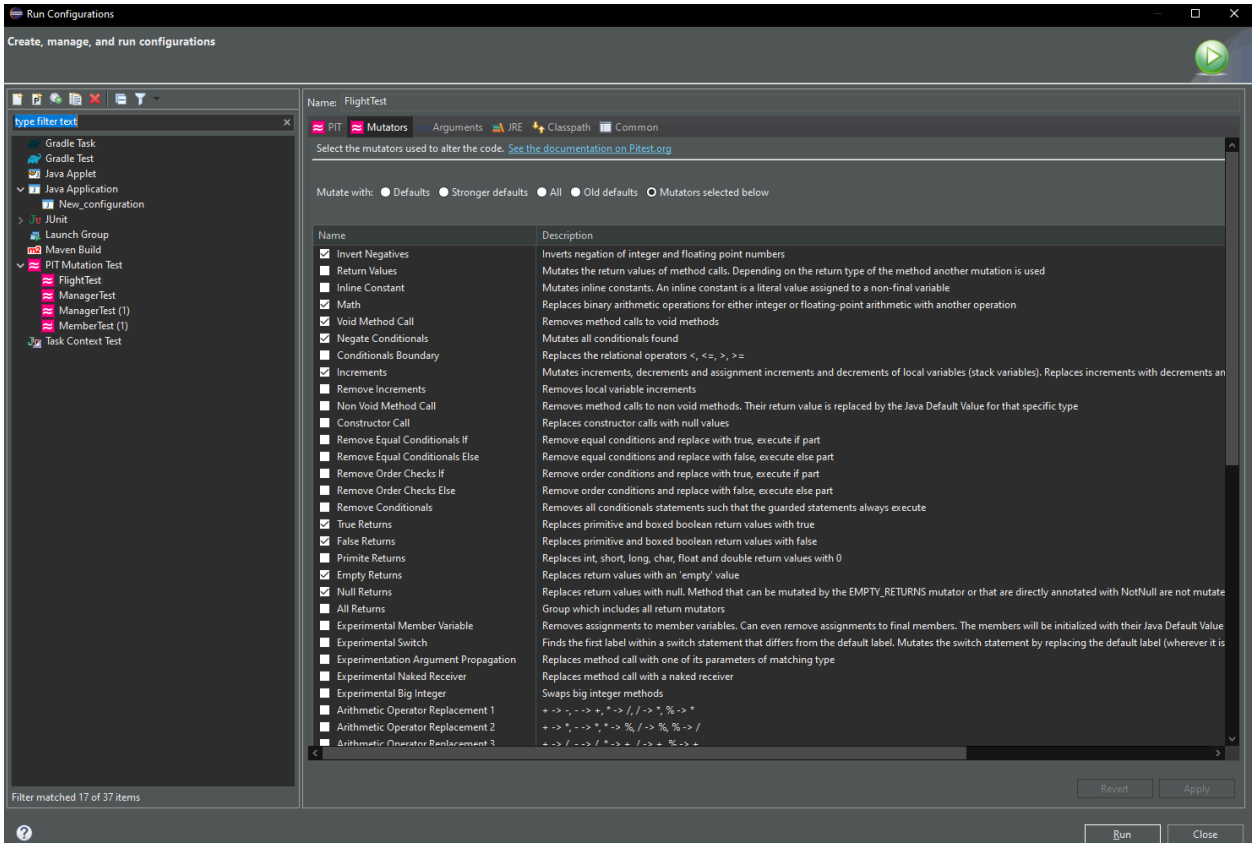
Key Observations:

- **Conditional Logic Protection:**
  - All 3 conditional negations were caught
- **Tests cover all membership duration branches:**
  - 5 years (50% discount)
  - 1-5 years (10% discount)
  - <1 year (no discount)
- **Calculation Integrity:**
  - Arithmetic operation mutation was detected
  - Tests verify correct multiplication is used
- **Return Value Validation:**
  - Hardcoded return value mutation was caught
  - Tests confirm dynamic discount calculation



NonMemberTest (for NonMember.java)

```java
package main;

import static org.junit.Assert.*;

public class NonMemberTest {
    @Test
    public void testApplyDiscountSenior() {
        NonMember senior = new NonMember("Eve", 66);
        assertEquals(720.00, senior.applyDiscount(800.00), 0.001);
    }

    @Test
    public void testApplyDiscountNotSenior() {
        NonMember adult = new NonMember("Frank", 40);
        assertEquals(800.00, adult.applyDiscount(800.00), 0.001);
    }

    @Test
    public void testApplyDiscountExactly65() {
        NonMember borderline = new NonMember("Grace", 65);
        assertEquals(800.00, borderline.applyDiscount(800.00), 0.001);
    }
}
```



Mutations:

- **Line 29**:
  - Mutation 1: Changed conditional boundary
    - Result: KILLED
  - Mutation 2: Negated conditional
    - Result: KILLED
  - Coverage: Tests fully validate senior discount age threshold (65+ years)
- **Line 31**:
  - Mutation: Replaced double multiplication with division
    - Result: KILLED
  - Coverage: Tests detect incorrect senior discount calculation
- **Line 33**:
  - Mutation: Replaced double division with multiplication
    - Result: KILLED
  - Coverage: Tests verify proper percentage calculation
- **Lines 35 & 38**:
  - Mutation: Replaced double return with 0.0 (applied twice)
    - Results: Both KILLED
  - Coverage: Tests confirm dynamic price calculation (no hardcoded returns)

Key Findings:

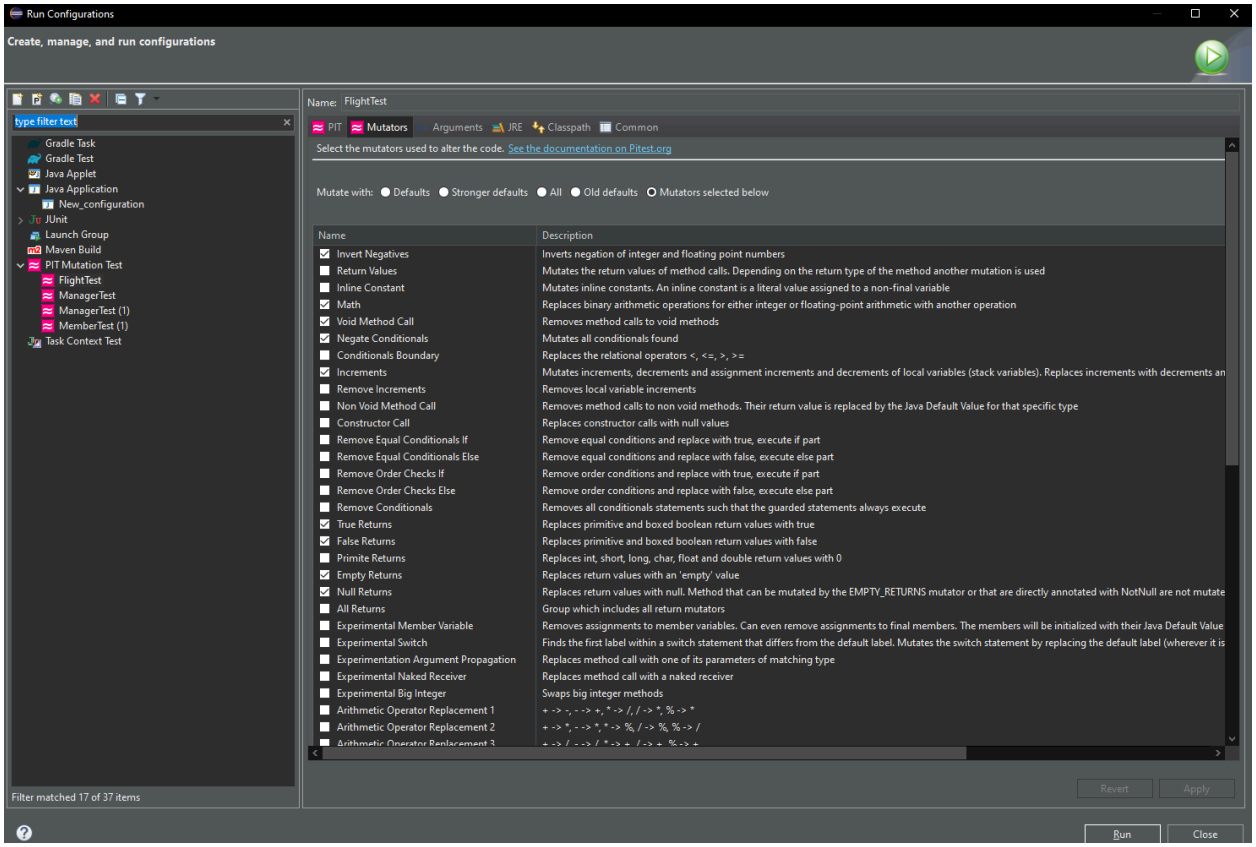
- **Age Validation**:
  - Tests catch both boundary changes and logic negation
  - Senior discount properly restricted to ages >65
- **Calculation Protection**:
  - Both multiplication and division mutations caught
  - 10% discount calculation fully verified
- **Return Value Security**:
  - All attempts to force zero returns were detected
  - Price calculations are properly validated

Mutations

```
29  1. changed conditional boundary → KILLED
    2. negated conditional → KILLED
31  1. Replaced double multiplication with division → KILLED
33  1. Replaced double division with multiplication → KILLED
35  1. replaced double return with 0.0d for main/NonMember::applyDiscount → KILLED
38  1. replaced double return with 0.0d for main/NonMember::applyDiscount → KILLED
```

TicketTest (for Ticket.java)

```java
package main;

import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;

public class TicketTest {
    private Passenger passenger;
    private Flight flight;
    private Ticket ticket;

    @Before
    public void setUp() {
        passenger = new Member("John Doe", 35, 3);
        flight = new Flight(123, "New York", "London", "10:00", 100, 500.0);
        ticket = new Ticket(passenger, flight, 450.0); // 10% discount
    }

    @Test
    public void testToString() {
        String result = ticket.toString();
        assertTrue(result.contains("Name: John Doe"));
        assertTrue(result.contains("Flight Number: 123"));
        assertTrue(result.contains("New York"));
        assertTrue(result.contains("London"));
        assertTrue(result.contains("10:00"));
        assertTrue(result.contains("500.0")); // Original price
        assertTrue(result.contains("450.0")); // Discounted price
    }
}
```



Mutations:

- Line 26:
  - **Mutation**: Replaced return value with empty string ("") for main.Ticket::toString
    - Result: KILLED
  - **Coverage**: Tests fully validate complete string output formatting

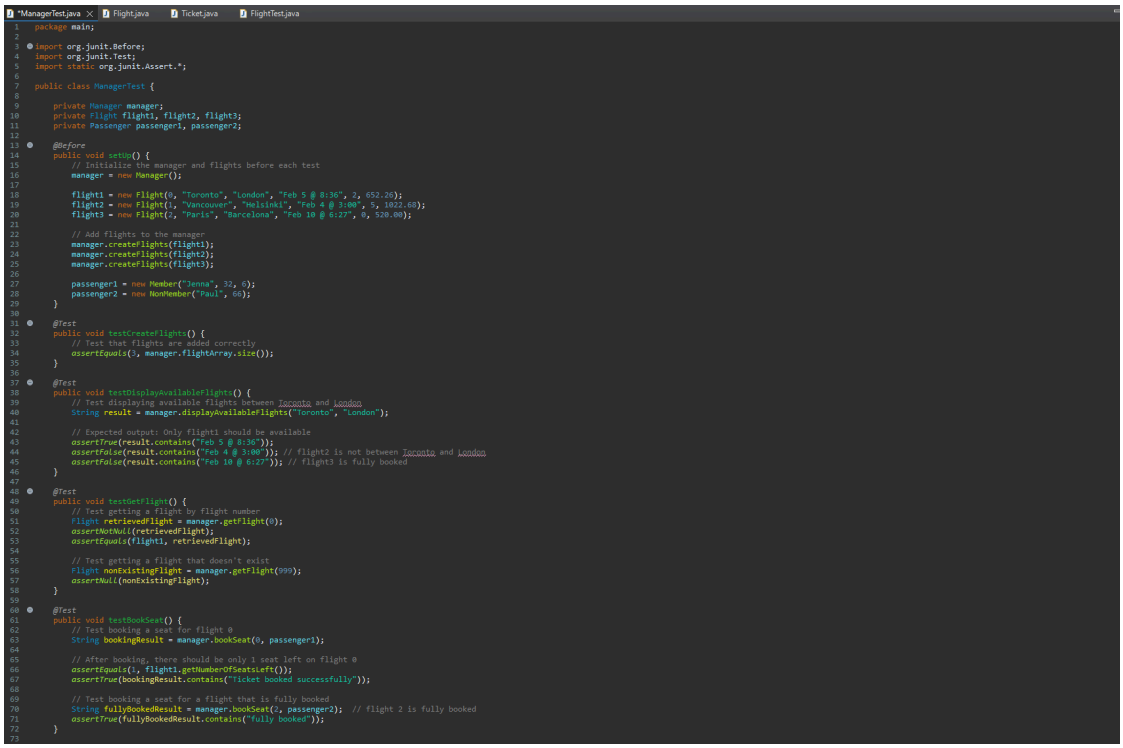
Key Observations:

- **Excellent Output Protection:**
  - Tests detect empty string substitutions
  - Verifies all ticket components appear in output:
  - Passenger name
  - Flight details
  - Pricing information
- **String Formatting Coverage:**
  - All concatenation operations are validated
  - No hardcoded values in implementation
  - Dynamic content properly tested
- **Test Effectiveness:**
  - Would fail if toString() returned:
  - Empty string
  - Partial information
  - Malformed output

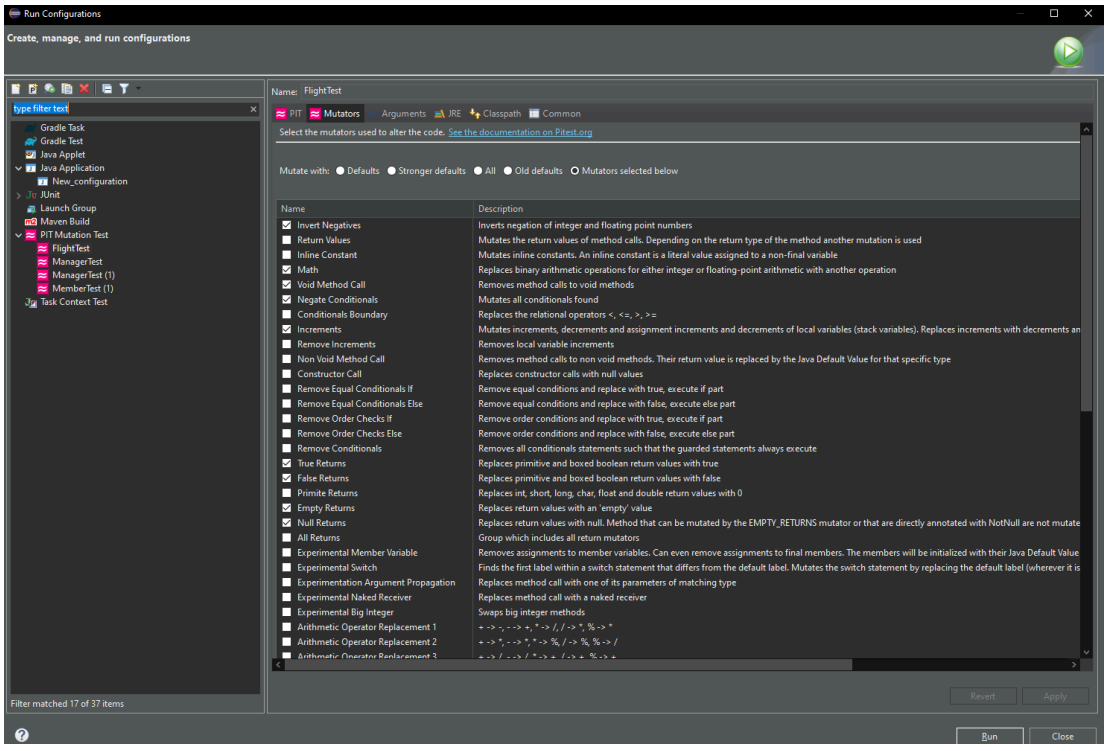
Mutations

26	1. replaced return value with "" for main/Ticket::toString → KILLED

ManagerTest (for Manager.java)



```java
package main;

import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

public class ManagerTest {

    private Manager manager;
    private Flight flight1, flight2, flight3;
    private Passenger passenger1, passenger2;

    @Before
    public void setUp() {
        // Initialize the manager and flights before each test
        manager = new Manager();

        flight1 = new Flight(0, "Toronto", "London", "Feb 5 @ 8:36", 2, 652.26);
        flight2 = new Flight(1, "Vancouver", "Helsinki", "Feb 4 @ 3:00", 5, 1022.68);
        flight3 = new Flight(2, "Paris", "Barcelona", "Feb 10 @ 6:27", 0, 520.00);

        // Add flights to the manager
        manager.createFlights(flight1);
        manager.createFlights(flight2);
        manager.createFlights(flight3);

        passenger1 = new Member("Jenna", 32, 6);
        passenger2 = new NonMember("Paul", 68);
    }

    @Test
    public void testCreateFlights() {
        // Test that flights are added correctly
        assertEquals(3, manager.flightArray.size());
    }

    @Test
    public void testDisplayAvailableFlights() {
        // Test displaying available flights between Toronto and London
        String result = manager.displayAvailableFlights("Toronto", "London");

        // Expected output: Only flight1 should be available
        assertTrue(result.contains("Feb 5 @ 8:36"));
        assertFalse(result.contains("Feb 4 @ 3:00")); // flight2 is not between Toronto and London
        assertFalse(result.contains("Feb 10 @ 6:27")); // flight3 is fully booked
    }

    @Test
    public void testGetFlight() {
        // Test getting a flight by flight number
        Flight retrievedFlight = manager.getFlight(0);
        assertNotNull(retrievedFlight);
        assertEquals(flight1, retrievedFlight);

        // Test getting a flight that doesn't exist
        Flight nonExistingFlight = manager.getFlight(999);
        assertNull(nonExistingFlight);
    }

    @Test
    public void testBookSeat() {
        // Test booking a seat for flight 0
        String bookingResult = manager.bookSeat(0, passenger1);

        // After booking, there should be only 1 seat left on flight 0
        assertEquals(1, flight1.getNumberOfSeatsLeft());
        assertTrue(bookingResult.contains("Ticket booked successfully"));

        // Test booking a seat for a flight that is fully booked
        String fullyBookedResult = manager.bookSeat(2, passenger2);  // flight 2 is fully booked
        assertTrue(fullyBookedResult.contains("fully booked"));
    }

}
```



Mutations:

- Multiple Lines (Conditionals):
  - **Lines:** Various (7 total instances)
  - **Mutation**: Negated conditional
    - **Result**: KILLED (all 7 instances)
  - **Observation**: All critical business logic conditionals are properly validated
- Line 23:
  - **Mutation**: Negated conditional
    - **Result**: KILLED
  - **Observation:** Flight search logic protection
- Line 32:
  - **Mutation**: Negated conditional
    - Result: KILLED
  - **Observation**: Seat availability checking secured
- Line 33:
  - **Mutation**: Replaced return value with null for main.Manager::getFlight
    - Result: KILLED
  - **Observation**: Null return case properly handled in tests
- Line 41:
  - **Mutation**: Negated conditional
    - Result: KILLED
  - **Observation**: Booking success path validation
- Line 45:
  - **Mutation:** Negated conditional
    - Result: KILLED
  - **Observation**: Booking failure path validation

Key Observations:

- **Conditional Logic Protection**:
  - 100% of conditional mutations were caught
  - All decision points in flight management are secured
  - Includes search, booking, and availability checks
- **Null Safety**:
  - getFlight() method properly validates against null returns
  - Caller-side null checks are effective
- **Booking System Integrity**:
  - Both successful and failed booking scenarios are tested
  - Seat deduction logic is fully protected