

# Software Quality Monitoring & Improvement

## Airline Reservation System

Han Xiao - [hxia0019@student.monash.edu](mailto:hxia0019@student.monash.edu)

Yuhang Huang - [yhua0226@student.monash.edu](mailto:yhua0226@student.monash.edu)

Jinhui Yuan - [jyua0030@student.monash.edu](mailto:jyua0030@student.monash.edu)

## 1. Code Quality Analysis Improvement Critique

### 1.1 Critique of the Code Quality Based on ISC/IEC 25010

Airplane			
Criterion	Satisfied?	Reason(s)	Improvement(s)
Functional Correctness	Yes	The code ensures that all fields and details for an airplane, such as airplaneID, businessSitsNumber, crewSitsNumber, etc., are tested. This ensures the correctness of the attributes of the airplane. The code checks that the seat number is within the range [1, 300], which satisfies the requirement for seat number validation.	None
Functional Completeness	Yes	The code covers all the fields and details of the airplane, ensuring that they are tested. This ensures the functional completeness of the airplane class. The code includes the validation of the seat number, ensuring that it falls within the specified range. This contributes to the functional completeness by covering all possible seat number scenarios.	None
Functional Appropriateness	Yes	The code provides the appropriate functionality for setting the attributes of the airplane class. The code includes the necessary unit testing conditions to ensure the correctness and completeness of the airplane attributes, including seat number validation.	None
Time Behaviour	Yes	Through performance analysis and optimization, the system has been ensured to respond to user operations within a reasonable timeframe and maintain stable performance.	None
Resource Utilization	Yes	Through code reviews and performance testing, the efficiency of resource usage during functionality execution have been ensured.	None
Capacity	Yes	Through capacity planning and evaluation, the system has been ensured to have sufficient processing power and storage capacity.	None
User Error Protection	Yes	To ensure the system's availability, the airplane class has designed a series of error alerts for the validation set of the test suite. When the number of seats on the plane does not	None

		meet the specified range, the user is alerted with the message "The number of seats must be within the range of [1, 300]."	
Accessibility	Yes	The user interface and interaction design have been optimized to provide an intuitive experience for all users. For example, when the system expects the user's next input, it will provide a prompt.	None
Fault Tolerance	Yes	After design and testing, the system has a fault tolerance mechanism that can detect and handle errors, ensuring system stability and reliability.	None

Flight			
Criterion	Satisfied?	Reason(s)	Improvement(s)
Functional Correctness	Yes	Provides correct Flight information reading and modification function, which can correctly modify or read the destination, time, code and other information of Flights. But the logic error risk of users inputting the arrival time and departure time of a flight is possible. And there may be a logical error risk in which users input the same departure and destination for a flight.	By adding logic error detection that ensures the <b>Arrival time cannot be earlier than the departure time</b> and <b>The departure and destination for a flight cannot be the same</b> . Providing users with appropriate prompts, the risk of logic errors in user input is reduced.
Functional Completeness	No	Provides a constructor that meets the needs of the system. Allow the system to read or modify Flight information such as destination, time, code, and various other information that needs to be used. But <i>getFlightInfo()</i> will throw an exception when it can't find a suitable flight, which prevents TicketSystem's code about finding a connecting flight from being run.	Returns a NULL value to TicketSystem for further processing when <i>getFlightInfo()</i> can't find a suitable flight, rather than throwing an exception.
Functional Appropriateness	Yes	Flight only provides the methods that the system needs to use, but not the extra methods that the system does not need, such as Code format conversion.	None
Time Behaviour	Yes	All the functions provided by Flight are available in O(1) time complexity a bit. It can meet the requirements of the system and there is no room for optimization.	None
Resource Utilization	Yes	Through code reviews and performance testing, the efficiency and optimization of resource usage during functionality execution have been ensured.	None
Capacity	Yes	Parameters in the Flight class are stored in the system as variables, and capacity can fulfill the requirement of storing any Flight information.	None
User Error Protection	Yes	To ensure the system's availability, the Flight class has designed a series of error alerts for the validation set of the test suite. For example, if the user enters the wrong Flight date information, the system will throw a "Time format error" exception message.	None
Accessibility	Yes	Flight class can be accessed on our system to	None

		anyone who needs to read or modify Flight information.	
Fault Tolerance	Yes	After design and testing, the system has a fault tolerance mechanism that can detect and handle errors, ensuring system stability and reliability. For example, when a user enters a departure time later than the arrival time for a flight, the system will provide a prompt.	None

<b>FlightCollection</b>			
<b>Criterion</b>	<b>Satisfied?</b>	<b>Reason(s)</b>	<b>Improvement(s)</b>
Functional Correctness	Yes	Provides correct Flight List information reading and adding function, which can correctly add or get flight instances from the list.	None
Functional Completeness	Yes	Provides a constructor that meets the needs of the system. Allow the system to read or modify Flight information such as destination, time, code, and various other information that needs to be used.	None
Functional Appropriateness	Yes	FlightCollection only provides the methods that the system needs to use, but not the extra methods that the system does not need, and all the methods will be used.	None
Time Behaviour	Yes	All the functions provided by FlightCollection are available in under O(n) time complexity. It can meet the requirements of the system and there is no room for optimization.	None
Resource Utilization	Yes	Through code reviews and performance testing, the efficiency and optimization of resource usage during functionality execution have been ensured.	None
Capacity	Yes	Parameters in the FlightCollection class are stored in the system as variables in list format. Its capacity can fulfill the requirement of storing any amount of Flight instances.	None
User Error Protection	Yes	To ensure the system's availability, the FlightCollection class has designed a series of error alerts for the validation set of the test suite. For example, if the user adds a Flight which already exists, the class will throw an "Already existing" exception message.	None
Accessibility	Yes	FlightCollection class can be accessed on our system to anyone who needs to get and add flight instances.	None
Fault Tolerance	Yes	After design and testing, the system has a fault tolerance mechanism that can detect and handle errors, ensuring system stability and reliability.	None

<b>Passenger</b>			
<b>Criterion</b>	<b>Satisfied?</b>	<b>Reason(s)</b>	<b>Improvement(s)</b>
Functional Correctness	Yes	The code checks for the required fields when creating or returning a passenger, including first name, last name, age, gender, phone number, email, and passport number. This ensures the correctness	None

		<p>of the required fields.</p> <p>The validation patterns for phone numbers and email addresses ensure that they follow the expected format requirements.</p> <p>The length restriction on the passport number ensures that it does not exceed 9 characters, as required.</p>	
Functional Completeness	Yes	After thorough requirements analysis and integration testing, the completeness of the functionality has been achieved. The code enforces the completeness of all fields, meaning that all fields need to be provided when creating or returning a passenger. The validation of phone numbers, email addresses, and passport numbers ensures the completeness of these fields.	None
Functional Appropriateness	Yes	The code provides the appropriate functionality for the creation and retrieval of passenger objects. The validation of phone numbers, email addresses, and passport numbers uses appropriate patterns and rules.	None
Time Behaviour	Yes	Through performance analysis and optimization, the system has been ensured to respond to user operations within a reasonable timeframe and maintain stable performance.	None
Resource Utilization	Yes	Through code reviews and performance testing, the efficiency and optimization of resource usage during functionality execution have been ensured.	None
Capacity	Yes	Through capacity planning and evaluation, the system has been ensured to have sufficient processing power and storage capacity.	None
User Error Protection	Yes	All fields are required, which reduces errors caused by users' missed input. The mobile phone number verifies the format and reduces illegal input. The mailbox verifies the format and reduces illegal input. It limits the length of passport numbers and reduces illegal entry. Firstname, last name, age and gender are mandatory when adding passengers, reducing errors caused by missing required items. Providing error messages to inform the user that the input data format is wrong.	None
Accessibility	Yes	The user interface and interaction design have been optimized to provide an intuitive experience for all users. For example, when the system expects the user's next input, it will provide a prompt.	None
Fault Tolerance	Yes	After design and testing, the system has a fault tolerance mechanism that can detect and handle errors, ensuring system stability and reliability	None

Person			
Criterion	Satisfied?	Reason(s)	Improvement(s)
Functional Correctness	Yes	Provides correct Person information get and set function, which can correctly add or get name, age, gender and other information.	None

Functional Completeness	Yes	Provides a constructor that meets the needs of the system. Allow the system to read or modify Person information such as age, gender, first name, and second name that needs to be used.	None
Functional Appropriateness	Yes	Person only provides the methods that the system needs to use, but not the extra methods that the system does not need, and all the methods will be used.	None
Time Behaviour	Yes	All the functions provided by Person are available in under O(1) time complexity. It can meet the requirements of the system and there is no room for optimization.	None
Resource Utilization	Yes	Through code reviews and performance testing, the efficiency and optimization of resource usage during functionality execution have been ensured.	None
Capacity	Yes	Parameters in the Person class are stored in the system as variables. Its capacity can fulfill the requirement of storing any Person information.	None
User Error Protection	Yes	To ensure the system's availability, the Person class has designed a series of error alerts for the validation set of the test suite. For example, if the user adds a Person without gender parameter, the class will throw an "Gender cannot be empty" exception message.	None
Accessibility	Yes	Person class can be accessed on our system to anyone who needs to get or set passenger information.	None
Fault Tolerance	Yes	After design and testing, the system has a fault tolerance mechanism that can detect and handle errors, ensuring system stability and reliability.	None

Ticket			
Criterion	Satisfied?	Reason(s)	Improvement(s)
Functional Correctness	Yes	Ticket class can correctly perform the methods of getting specific information, like price, flight, and so on.	None
Functional Completeness	Yes	Provide constructors that meet the needs of the system. Allow the system to read or modify the information needed to purchase a ticket, such as id, flight, and crew information.	None
Functional Appropriateness	No	Some in-class functions are redundant at initialisation time and do not need to be defined	Delete some redundant code for this class
Time Behaviour	Yes	All the functions provided by Ticket are available in under O(1) time complexity. It can meet the requirements of the system and there is no room for optimization.	None
Resource Utilization	Yes	Efficiency and optimisation of resource usage during function execution is ensured through code review and performance testing.	None
Capacity	Yes	The parameters in the ticket class are stored as variables in the system. It has the capacity to fulfill the requirement of storing any ticket information.	None

User Error Protection	No	In the function function that determines the price of an air ticket based on age, some boundary conditions are not well considered	Modified boundary conditions and optimized multiple if statements to plain logic
Accessibility	Yes	Ticket class can be accessed on our system to anyone who needs to get or set ticket information.	None
Fault Tolerance	Yes	Designed and tested, the system has a fault-tolerant mechanism that detects and handles errors to ensure system stability and reliability.	None

TicketCollection			
Criterion	Satisfied?	Reason(s)	Improvement(s)
Functional Correctness	Yes	This class is a good implementation of the query to add and return to the collection of tickets, and can check for duplicate tickets added	None
Functional Completeness	No	A judgment under the null condition was not made for functions in this class that return all tickets.	Add a null judgment
Functional Appropriateness	Yes	TicketCollection only provides the methods that the system needs to use, but not the extra techniques that the system does not need, and all the methods will be used.	None
Time Behaviour	Yes	All the functions provided by Ticket are available under O(n) time complexity. It can meet the requirements of the system but there is still room for optimization.	None
Resource Utilization	Yes	Through code reviews and performance testing, the efficiency and optimization of resource usage during functionality execution have been ensured.	None
Capacity	Yes	The parameters in the TicketCollection class are stored as static variables in the system. It fulfills the requirement of storing any air ticket along with its information.	None
User Error Protection	Yes	Returns errors or non-existent information entered by the user.	None
Accessibility	Yes	TicketCollection class can be accessed on our system to anyone who needs to get or set ticket information.	
Fault Tolerance	Yes	Designed and tested, the system has a fault-tolerant mechanism that detects and handles errors to ensure system stability and reliability	None

TicketSystem			
Criterion	Satisfied?	Reason(s)	Improvement(s)
Functional Correctness	No	<p>Regarding the fact that no direct flights were searched for, but the code to look for connecting flights was not set correctly and was not tested, the <i>chooseTicket()</i> method input the flightID but we need a ticket ID.</p> <p>Some error messages returned by other classes conflict with messages returned by Ticket System itself.</p>	<p>Add test cases for connecting flights and improve the <i>chooseTicket()</i> method.</p> <p>Changing or deleting error messages</p>

Functional Completeness	No	In this test we found that in the TicketSystem code for Assignment1, the code about searching for non-direct flights was not completed and was not tested.	Added code about searching for non-direct flights and tested it.
Functional Appropriateness	No	Some lines of code in the buyTicket function are duplicated and redundant, and can be deleted directly. The reduction in the number of seats after the purchase of a ticket was not tested in the test.	Remove redundant code and add test cases
Time Behaviour	Yes	All the functions provided by Ticket are available under O(n) time complexity. It can meet the requirements of the system but there is still room for optimization.	None
Resource Utilization	Yes	Through code reviews and performance testing, the efficiency and optimization of resource usage during functionality execution have been ensured.	None
Capacity	Yes	Parameters in the TicketSystem class are stored in the system as variables in list format. Its capacity can fulfill the requirement of processing tickets about clients buying.	None
User Error Protection	Yes	The TicketSystemclass has designed a series of error alerts for user error protection. If the user input invalid passenger information when buying tickets, the class will throw an corresponding exception message.	None
Accessibility	Yes	TicketSystem class can be accessed on our system to anyone who needs to choose or buy a ticket.	None
Fault Tolerance	Yes	Designed and tested, the system has a fault-tolerant mechanism that detects and handles errors to ensure system stability and reliability.	None

## 1.2 Identification for improvement in the codebase

### 1.2.1 Flight

1. For the setting of the departure time and arrival time of the flight, there may be a logical error risk that the user enters the arrival time of the flight earlier than the departure time, which is not in line with the actual situation.
2. In addition, besides the logical error risk of users inputting an arrival time earlier than the departure time, there may also be a logical error risk of users inputting the same departure and destination locations for a flight. This situation is also inconsistent with reality because the departure and destination locations of a flight should be different places.
3. In our Assignment1 code, when the *getFlightInfo()* function within the Flight class doesn't find a suitable flight, it will throw an exception, which prevents the *chooseticket()* function within TicketSystem from continuing to look for a connecting flight with transfer.



```

public static Flight getFlightInfo(String city1, String city2) { city1: "SHANGHAI" city2: "BEIJING"
//display the flights where there is a direct flight from city 1 to city2
if(city1 == null||city2 == null){
    throw new IllegalArgumentException("City name cannot be null");
}
if (!city1.matches( regex: "[a-zA-Z\\s]+$"))||!city2.matches( regex: "[a-zA-Z\\s]+$"))
    throw new IllegalArgumentException("City name can only contain letter and space");
for (Flight flight : flights) {
    if(flight.getDepartFrom().equals(city1) && flight.getDepartTo().equals(city2)){ city1: "SHANGHAI" city2: "BEIJING"
        return flight;
    }
}
throw new RuntimeException("No such flight exists");
}

```

## 1.2.2 Ticket

1. For the function of pricing according to age in the Ticket class, the logic was not taken into account in the previous modification, and some special cases and boundary conditions were not tested in the test cases, which led to the successful survival of the hair club in the subsequent mutation test.

```

58 59 public void saleByAge(int age)
59 60 {
60 61     int price = getPrice();
61 62     if(age < 15)
62 63     if(age == 0)
63 64     this.price = price;
64 65     if(age < 15 && age > 0)
65 66     {
66 67         price=(int)price*0.5;//50% sale for children under 15
67 68         this.price=price;

```

2. There is some redundant code in the constructor and parameters of the ticket class, as a ticket needs to be created successfully with the appropriate information, so the *setXXX()* function is redundant.

```

11 11 public Ticket(int ticket_id, int price, Flight flight, boolean classVip, Passenger passenger)
12 12 {
13 13     //this.passenger=passenger;
13 14     this.passenger=passenger;
14 14     this.setPassenger(passenger);
15 15     //this.flight = flight;
15 16     this.flight = flight;
16 16     this.setFlight(flight);
17 17     //this.classVip = classVip;
18 18     this.setClassVip(classVip);
19 19     //this.status = false;
20 20     this.setTicketStatus(status);
21 21     //this.ticket_id=ticket_id;
22 22     this.setTicket_id(ticket_id);
23 23     this.classVip = classVip;
24 24     this.status = false;
25 25     this.ticket_id=ticket_id;
26 26     this.price = price;
27 27     this.setPrice(price);

```

## 1.2.3 TicketSystem

1. Because it is a team effort, the legality of the location has already been checked when searching for the relevant flights, so there is no need to check the place name again in the Ticket System, hence the redundant code snippet.



```

        flight = flightCollection.getFlightInfo(city1, city2);
    }
    // Check if the city parameters are valid
    // delete below two lines
    // if(city1 == "" || city2 == ""){
    //     throw new IllegalArgumentException("City name cannot miss");
    // }
    // if (!city1.matches("[a-zA-Z\\s]+$") || !city2.matches("[a-zA-Z\\s]+$"))
    //     throw new IllegalArgumentException("City name can only contain letter and space");

```

2. In the Ticket System's ticketing function, the test that the number of seats on a flight should be reduced after a ticket is purchased is not covered, so it needs to be checked.

```

        if (ticket.getClassVip() == true)
        {
            airplane.setBusinessSitsNumber(airplane.getBusinessSitsNumber() - 1);
        } else
        {
            airplane.setEconomySitsNumber(airplane.getEconomySitsNumber() - 1);
        }

    } else {
        throw new IllegalArgumentException("Error Input");
    }

```

3. We found that in the TicketSystem code for Assignment1, the code about searching ticket for non-direct flights was not completed. So we added code about searching for non-direct flights and tested it.

## 1.3 Changes made to improve the quality of the codebase

### 1.3.1 Flight

1. After receiving the arrival and departure time of the flight entered by the user, perform a logical check to capture potential errors. Compare the order of departure time and arrival time.

```

if(stringToTimestamp(dateTo).compareTo(dateFrom) <= 0){

    throw new IllegalArgumentException("The dateFrom must before time of dateTo");

}

```

If the arrival time is earlier than the departure time, the user will be prompted to enter incorrectly and ask for re-enter.

2. Upon receiving the user input for the departure and destination locations of the flight, a logical check is performed to identify potential errors.

```

if (departTo.equals(departFrom)) {

```

```

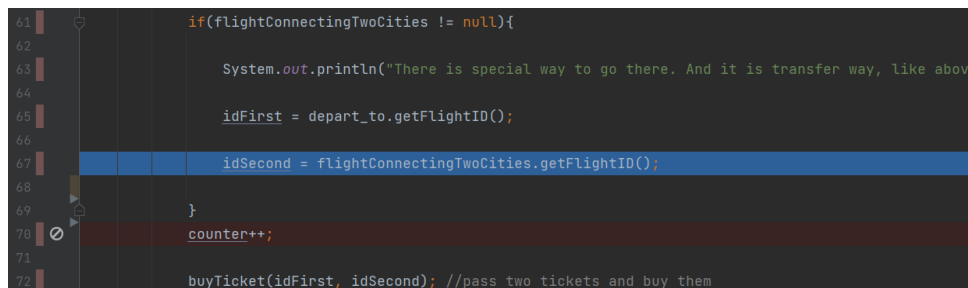
        throw new IllegalArgumentException("Departure and Destination cities
cannot be the same");
    }

```

The departure and destination locations are compared, and if they are found to be the same, the user is prompted with an error message indicating incorrect input and asked to re-enter the information.

3. When the *getFlightInfo()* function within the Flight class doesn't find a suitable flight, the original direct throw exception is replaced with returning a NULL value, allowing subsequent TicketSystem functions to continue to determine whether a connecting flight exists.

### 1.3.2 FlightCollection



```

61  if(flightConnectingTwoCities != null){
62
63      System.out.println("There is special way to go there. And it is transfer way, like above");
64
65      idFirst = depart_to.getFlightID();
66
67      idSecond = flightConnectingTwoCities.getFlightID();
68
69  }
70  counter++;
71
72  buyTicket(idFirst, idSecond); //pass two tickets and buy them

```

1. In the case where a connecting flight is required, the corresponding code part in TicketSystem class inputs the IDs of the two connecting flights directly into the *buyTicket()* function instead of the ticket's ID, which leads to an error. We repaired it.

### 1.3.3 Ticket

1. Changes have been made to the constructor of the Ticket class and the *SalebyAge()* function has been logically changed and optimized to make it easy to understand.

```

public Ticket(int ticket_id, int price, Flight flight, boolean classVip,
Passenger passenger)
{
    this.passenger=passenger;
    this.setPassenger (passenger) ;
    this.flight = flight;
    this.setFlight (flight) ;
    this.classVip = classVip;
    this.status = false;
}

```

```
this.ticket_id = ticket_id;

this.price = price;
}
```

```
public void saleByAge(int age) {
    int price = getPrice();
    if (age < 15) {
        price -= (int) price * 0.5; //50% sale for children under 15
        this.price = price;
    } else if (age >= 60) {
        this.price = 0; //100% sale for elder people
    } else
        this.price = price;
}
```

### 1.3.4 TicketSystem

1. Comment out the Exception snippet in the class after the place name selection.
2. Added code about searching for non-direct flights.

## 2. Mutation Testing via PIT

### 2.1 Test suite quality based on mutation testing

The Pit test coverage result based on the code designed in Assignment 1:

# Pit Test Coverage Report

## Package Summary

### Codebase

Number of Classes	Line Coverage	Mutation Coverage	Test Strength
8	78% <div><div></div><div></div><div></div></div> 397/508	66% <div><div></div><div></div><div></div></div> 206/311	84% <div><div></div><div></div><div></div></div> 206/246

### Breakdown by Class

Name	Line Coverage	Mutation Coverage	Test Strength
<a href="#">Airplane.java</a>	100% <div><div></div><div></div><div></div></div> 49/49	100% <div><div></div><div></div><div></div></div> 33/33	100% <div><div></div><div></div><div></div></div> 33/33
<a href="#">Flight.java</a>	98% <div><div></div><div></div><div></div></div> 61/62	98% <div><div></div><div></div><div></div></div> 41/42	98% <div><div></div><div></div><div></div></div> 41/42
<a href="#">FlightCollection.java</a>	95% <div><div></div><div></div><div></div></div> 38/40	96% <div><div></div><div></div><div></div></div> 22/23	96% <div><div></div><div></div><div></div></div> 22/23
<a href="#">Passenger.java</a>	100% <div><div></div><div></div><div></div></div> 52/52	100% <div><div></div><div></div><div></div></div> 31/31	100% <div><div></div><div></div><div></div></div> 31/31
<a href="#">Person.java</a>	100% <div><div></div><div></div><div></div></div> 34/34	100% <div><div></div><div></div><div></div></div> 23/23	100% <div><div></div><div></div><div></div></div> 23/23
<a href="#">Ticket.java</a>	100% <div><div></div><div></div><div></div></div> 54/54	82% <div><div></div><div></div><div></div></div> 27/33	82% <div><div></div><div></div><div></div></div> 27/33
<a href="#">TicketCollection.java</a>	89% <div><div></div><div></div><div></div></div> 24/27	70% <div><div></div><div></div><div></div></div> 7/10	78% <div><div></div><div></div><div></div></div> 7/9
<a href="#">TicketSystem.java</a>	45% <div><div></div><div></div><div></div></div> 85/190	19% <div><div></div><div></div><div></div></div> 22/116	42% <div><div></div><div></div><div></div></div> 22/52

Based on the mutation testing results, it can be observed that the Airplane, Passenger, Person, and Ticket classes performed well, achieving 100% coverage in the mutation testing. Additionally, the Flight and FlightCollection, as well as the TicketCollection classes, fell slightly short of achieving 100% coverage and required improvement. The TicketSystem exhibited poor performance in the mutation testing, indicating a need to enhance the test cases based on the PIT test report.

The Pit test coverage result based on the code designed in Assignment 2:

# Pit Test Coverage Report

## Package Summary

### Codebase

Number of Classes	Line Coverage	Mutation Coverage	Test Strength
8	99% <div><div></div><div></div><div></div></div> 496/501	99% <div><div></div><div></div><div></div></div> 296/300	99% <div><div></div><div></div><div></div></div> 296/299

### Breakdown by Class

Name	Line Coverage	Mutation Coverage	Test Strength
<a href="#">Airplane.java</a>	100% <div><div></div><div></div><div></div></div> 49/49	100% <div><div></div><div></div><div></div></div> 33/33	100% <div><div></div><div></div><div></div></div> 33/33
<a href="#">Flight.java</a>	100% <div><div></div><div></div><div></div></div> 66/66	100% <div><div></div><div></div><div></div></div> 44/44	100% <div><div></div><div></div><div></div></div> 44/44
<a href="#">FlightCollection.java</a>	100% <div><div></div><div></div><div></div></div> 39/39	100% <div><div></div><div></div><div></div></div> 22/22	100% <div><div></div><div></div><div></div></div> 22/22
<a href="#">Passenger.java</a>	100% <div><div></div><div></div><div></div></div> 52/52	100% <div><div></div><div></div><div></div></div> 31/31	100% <div><div></div><div></div><div></div></div> 31/31
<a href="#">Person.java</a>	100% <div><div></div><div></div><div></div></div> 34/34	100% <div><div></div><div></div><div></div></div> 23/23	100% <div><div></div><div></div><div></div></div> 23/23
<a href="#">Ticket.java</a>	100% <div><div></div><div></div><div></div></div> 54/54	100% <div><div></div><div></div><div></div></div> 26/26	100% <div><div></div><div></div><div></div></div> 26/26
<a href="#">TicketCollection.java</a>	100% <div><div></div><div></div><div></div></div> 28/28	100% <div><div></div><div></div><div></div></div> 9/9	100% <div><div></div><div></div><div></div></div> 9/9
<a href="#">TicketSystem.java</a>	97% <div><div></div><div></div><div></div></div> 174/179	96% <div><div></div><div></div><div></div></div> 108/112	97% <div><div></div><div></div><div></div></div> 108/111

Report generated by [PIT](#) 1.11.6

We based on the information provided by the mutation test on the code snippet, most of the information is because of the code used in the `System.out.println` function, which for the mutation test to remove is able to survive, we queried the information and learned that in the normal process of writing code should be used in the logging means to avoid the use of `println` function to detect. After the modification, we can ensure that almost all the classes of the code coverage of 100%, mutation test coverage is also as far as possible to achieve 100%, the whole test intensity is very considerable.

## 2.2 Identification for improvement in the test suite

### 2.2.1 Airplane

- To address the issue of "Replaced integer addition with subtraction → KILLED" in the line of code `int total = businessSitsNumber + economySitsNumber + crewSitsNumber`, it need to introduce test cases that trigger an error before the mutation and ensure no error occurs after the mutation.

### 2.2.2 Flight

- Based on the analysis of the PIT test report, test cases should be developed to target the uncovered code lines and branches. These newly created test cases aim to achieve full coverage of the identified code lines.
- In order to address the issue of "Changed conditional boundary → SURVIVED," test cases need to be added that specifically cover the vicinity of boundary values. This approach ensures the detection of mutations caused by the `CONDITIONALS_BOUNDARY` mutation operator.

### 2.2.3 FlightCollection

- Based on the analysis of the PIT test report, test cases will be developed to target the uncovered code lines and branches. These test cases aim to achieve full coverage of the identified code segments. The objective is to ensure that no code will be left untested.

### 2.2.4 Passenger

- To effectively handle the "changed conditional boundary → SURVIVED" issue, it needs to introduce test cases that focus on the neighboring values of boundary conditions. This strategy aims to detect and address mutations triggered by the `CONDITIONALS_BOUNDARY` mutation operator, thereby enhancing the overall reliability and effectiveness of the testing process.

### 2.2.5 Ticket

- To effectively handle the "changed conditional boundary → SURVIVED" issue, it needs to introduce test cases that focus on the neighboring values of boundary conditions. This strategy aims to detect and address mutations triggered by the CONDITIONALS\_BOUNDARY mutation operator, thereby enhancing the overall reliability and effectiveness of the testing process. I add three more cases that suits the boundary of the code I wrote.

```
passenger.setAge(14); // boundary value
ticket.setPrice(1000);
assertEquals( expected: 500, ticket.getPrice());
passenger.setAge(15); // boundary value
ticket.setPrice(1000);
assertEquals( expected: 1000, ticket.getPrice());

passenger.setAge(59); // boundary value
ticket.setPrice(1000);
assertEquals( expected: 1000, ticket.getPrice());
```

### 2.2.6 TicketCollection

- Based on the analysis of the PIT test report, the Line Coverage for TicketCollection is only 89% and 30% of the Mutation is not covered. We found that this is mainly due to the fact that the part of the code about adding duplicate tickets is not tested completely and the test method is flawed. We will make improvements in this assignment.

### 2.2.7 TicketSystem

```
32 // Check if the city parameters are valid
33 2 if(city1 == "" || city2 == ""){
34     throw new IllegalArgumentException("City name cannot miss");
35 }
36 2 if (!city1.matches("[a-zA-Z\\s]+$") || !city2.matches("[a-zA-Z\\s]+$"))
37     throw new IllegalArgumentException("City name can only contain letter and space");
```

- In our Assignment1's TicketSystem class, the *setFlight()* function is called with the city name entered by the user as a parameter first, and the legality of the city name is verified later, resulting in an exception being thrown by the Flight class when the user enters an incorrect city name, and the detection function of the TicketSystem class is not called.
- To effectively handle the “Replaced integer subtraction with addition - NO COVERAGE” and “Removed call to Codebase/Airplane::setbusinesssitshumber - NO COVERAGE” issue, two additional test cases are needed to see if the number of seats on a flight decreases after a successful ticket purchase.

```

assertEquals(29, TicketCollection.tickets.get(0).getFlight().getAirplane().getBusinessSitsNumber());

//Two additional cases for checking whether the seats are decrease.

assertEquals(99, TicketCollection.tickets.get(1).getFlight().getAirplane().getEconomySitsNumber());

```

- In our Assignment1's TicketSystem class, the code about searching for non-direct flights was not fully tested, so we supplemented the test with the code about non-direct flights following the test code about direct flights, using the *assertEquals()* function for all the information about the passenger, the airplane, and the ticket.

## 2.3 Benefits of mutation testing over simply running test cases

Mutation testing is designed to detect defects in the code by introducing intentional changes and evaluating the effectiveness of test cases in identifying these changes. This method uncovers weaknesses or gaps in the test suite that may not be apparent when only running test cases.

Mutation testing measures the ability of the test suite to detect and identify different types of code changes. This evaluation helps identify areas of the code that are not adequately covered by the tests and guides improvements in the test suite.

Additionally, mutation testing verifies the effectiveness of individual test cases by assessing whether they can distinguish between correct and faulty program behavior. This analysis helps identify weak or redundant test cases that may not significantly contribute to the overall test coverage.

By identifying code areas that are not well covered by the test suite, mutation testing highlights potential code quality issues. It encourages developers to write more comprehensive tests, leading to improved code quality and reducing the likelihood of undetected defects.

## 2.4 Changes made to improve the quality of the test suite

### 2.4.1 Airplane

- To ensure that an error occurs before the mutation and no error occurs after the mutation, the following code snippet can be added:

```

Throwable exception = assertThrows(IllegalArgumentException.class, ()
-> { airplane = new Airplane(1, "Boeing 737", 200, 100, 50);});

Assertions.assertEquals("Seat number must be in the range [1, 300].",
    exception.getMessage());

```



By incorporating this code, any errors caused by the mutation introduced by the MATH mutation operator can be detected and resolved, thus effectively eliminating the potential issues associated with the mutation.

### 2.4.2 Flight

- Adding a test for the empty constructor of the Flight class, which increased test coverage.
- Compared to Assignment 1, where only the departure time format of Flight was tested, adding a test for the arrival time format of Flight to increase test coverage.
- Adding a test for the boundary condition of FlightId, which should be a positive number, which successfully kills the mutation caused by the CONDITIONALS\_BOUNDARY mutation operator.

### 2.4.3 FlightCollection

- Compared to Assignment 1, where only testing for retrieving flight information based on a single city was performed, additional tests have been added to compare retrieving flight information based on two cities. These new tests also cover scenarios where both city inputs are empty. By incorporating these tests, the overall test coverage has been increased.

### 2.4.4 Passenger

- To further enhance the robustness of the codebase, tests were introduced to validate the boundary condition of "Passport number should not be more than 9 characters long." Specifically, test cases were designed to cover scenarios where the Passport number was set to both 9 and 8 characters in length. These tests successfully detected and addressed mutations caused by the CONDITIONALS\_BOUNDARY mutation operator, ensuring that the system enforces the specified constraint on Passport number length.

### 2.4.5 Ticket

- The original code was modified using mutation testing to make the logic smoother and to find some bugs.

### 2.4.6 TicketCollection

- In our previous test of TicketCollection, in the Adding Tickets section, we only tested the use case of adding two identical tickets at a time, and we were missing the test case of adding a ticket that already existed in the system. So the Line Coverage was only 89%, after additional testing the Line Coverage increased to 96%.

### TicketCollection.java

```
1 package Codebase;
2
3 import java.util.ArrayList;
4
5 public class TicketCollection {
6
7     public static ArrayList<Ticket> tickets = new ArrayList<>();
8
9     public static ArrayList<Ticket> getTickets() {
10         return tickets;
11     }
12
13     public static void addTickets(ArrayList<Ticket> tickets_db) {
14         // Check whether add exist ticket
15         ArrayList<Integer> ticketTemplst = new ArrayList<Integer>();
16         for(Ticket tempTicket: tickets_db) {
17             if (tempTicket != null) {
18                 if (ticketTemplst.contains(tempTicket.getTicket_id()))
19                     throw new IllegalArgumentException( "ID:" + tempTicket.getTicket_id() + " ticket was add twice");
20                 else {
21                     ticketTemplst.add(tempTicket.getTicket_id());
22                     1. addTickets : negated conditional -> NO_COVERAGE
23                     if (tempTicket.getTicket_id() == tempTicket.getTicket_id())
24                         throw new IllegalArgumentException( "ID:" + tempTicket.getTicket_id() + " ticket is ");
25                 }
26             }
27         }
28     }
29 }
```

- We also found an error in mutation at line 23, this is because in our original test, only a simple *try()....catch()* structure was used to catch the duplicate ticket adding error, when PIT used a negated conditional mutation, no exception would be caught and the test case would also pass. So I added the *assertThrows()* function to satisfy this mutation, and the Mutation Coverage upgraded by 10%.

## 2.4.7 TicketSystem

- We have adjusted the function in the TicketSystem class that validates the legitimacy of the city name before calling *setFlight()*, so that when the user enters the wrong city name, it will be detected by TicketSystem first and throw an exception.
- To make sure the users' inputs are correctly set as ticket information, we followed the test code about direct flights, using the *assertEqual()* function for all the information about the passenger, the airplane, and the ticket.
- We supplemented our testing with non-direct flights by setting up two flights as dummy data and using the *Mock* package to simulate user input and ultimately validate the information for the two tickets purchased. Make TicketSystem's test line coverage up from <60% to 97%.
- Code is more concise, removing redundant or duplicate code.