Test Plan:

## I. Unit and component Test:

We deployed unit test for all Java classes except those classes for responding front-end requests in the server-side by using JUnit.

1. Equivalence testing: e.g. Airport.java JUnit test class: AirportTest.java

In the unit testing for Airport Object, we deployed Equivalence testing as the main approach. Considering all the time zones in the U.S, we partitions all the airports in the current databases into several classes by their geolocation and time zone informations. We choose one specific input from each group and verify if our setTimeZone() can correctly retrieve timezone information and match them with corresponding instances.

|  |  |  |
| --- | --- | --- |
| Equivalence Class | Value for Location Input | Expected Results |
| Airports in EDT time zone | (JFK) 40.641518f, -73.77816f | Name: EDT Offset: -14400l DST used: True |
| Airports in CDT time zone | (MEM) 35.042343f, -89.97922f | Name: CDT Offset: -18000l DST used: True |
| Airports in MDT time zone | (DEN) 39.866373f, -104.67377f | Name: MDT Offset: -21600l DST used: True |
| Airports in PDT time zone | (LAX) 33.94443f, -118.408356f | Name: PDT Offset: -25200l DST used: True |
| Airports in AKDT time zone | (ANC) 61.176033f, -149.99008f | Name: AKDT Offset: -28800l DST used: True |
| Airports in HST time zone | (HNL) 21.324808f, -157.92519f | Name: HST Offset: -36000l DST used: False |
| Airports in MST time zone | (PHX) 33.43755f, -112.0078f | Name: MST Offset: -25200l DST used: False |

The test results are all failed at the first time because of the OVER-QUERY LIMIT of Google Map API. However, after we used another API called timezoneDB, problem was solved, and all test cases passed.

Log files and regarding screen shots are provided for presenting details.

2. Boundary testing: e.g. LocalTime.java JUnit Test class: LocalTimeTest.java

LocalTime is an object inheritance from Date from Java API. We decided to use Boundary teasing, because there are a lot of boundaries and “edge” cases in the time system. As a data object with regarding utility methods as well, LocalTime should be proved that it can handle both the common cases and corner cases. The testGetDSTOffset() is a good example in which we want to verify if getDSTOffset() can get the correct DST offset based on the input Date instance.

|  |  |  |  |
| --- | --- | --- | --- |
| Equivalence Class | Boundary type | String Value of Date instance Input | Expected Results |
| Time in DST used (U.S) | Up boundary | "2015 03 08 Sun 02:00 GMT" | DST Offset: 0l |
| Time in DST used (U.S) | Down boundary | “2014 11 02 Sun 01:59 GMT” | DST Offset: 0l |
| Time in DST not used (U.S) | Up boundary | "2015 03 08 Sun 01:59 GMT" | DST Offset: -3600l |
| Time in DST not used (U.S) | Down Boundary | “2014 11 02 Sun 02:00 GMT” | DST Offset: -3600l |

\*we assume DST is used because of the current operating time period is in DST time used range

The test results showed that this method of LocalTime unit can return correct value both in the common cases and the corner cases.

3. Path testing e.g. Airplane.java JUnit test class: AirplaneTest.java

We adopted path testing as the approach to test this unit, because the structure is simple enough for testing every paths inside this component. The test goal in this unit is to verify if setModel() and getModel() can set and return a specific Manufacturer and Model for an instance of Airplane. Also, we want to verify if setSeats() and getSeatNumber() can set and get a specific First Class seat number and Coach seat number;

|  |  |
| --- | --- |
| AirplaneTest | |
| testSetModel()  **Tested method:** setModel(); getModel(); | Test approach: Path testing  Test Case:  *I. Precondition:*  Input two Strings standing for Manufacturer name and Model for setModel();  Input a boolean variable to get a model or a manufacturer;  *II. Specific inputs and results:*  case1: case2:  Input: Input: "Boeing", "747"  Expected result: “”, “” (not null) Expected result: "Boeing", "747"  JUnit test results: Pass 1/1, Failure 0. |
| testSetSeats()  **Tested method:** setSeats();  getSeatNumber(); | Test approach: Path testing  Test Case:  *I. Precondition:*  Input two ints standing for First Class seat number and Coach seat number for setSeats();  Input a boolean variable to get a First Class seat number or Coach seat number;  *II. Specific inputs and results:*  Input: setSeats(20, 50); getSeatNumber(true); getSeatNumber(false);  Expected result: "20", "50"  JUnit test results: Pass 1/1, Failure 0. |

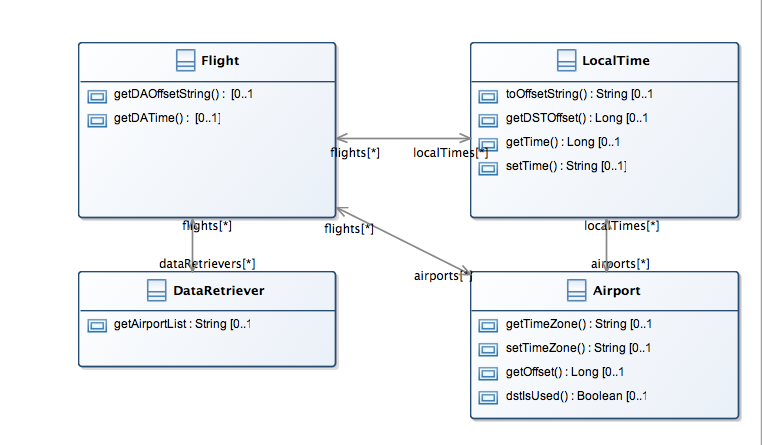
The final test results are as same as we expected.

4. State-based testing: e.g. BackEndInterface.java

5. Polymorphism testing: e.g. Flight.java

During the unit testing of object Flight, Polymorphism testing is adopted, due to the multiplicity of related methods called by certain methods inside object Flight. We want to both get the original value retrieved from databases and get the value converted, modified or added during the process of searching flights. A very good example is testGetDAOffsetString(). When getDAOffsetString() is called, a series of methods will also be called, the following figure can show the relations of this process.

|  |  |  |  |
| --- | --- | --- | --- |
| Bound Methods | Value for Input | Bound Methods | Value for Input |
| Flight.setDATime() | "2015 May 10 05:04 GMT", "2015 May 10 20:44 GMT" | Airport.getOffset() |  |
| Flight.getDAOffsetString() | 1 | Airport.dstIsUsed() |  |
| Flight.getDATime() | 1 | LocalTime.getDSTOffset() | Data date (same time as dep\_time + offset) |
| DataRetriever.getAirportList() |  | LocalTime.setTime() | Long this.getTime() + 1000\*offset + 1000\*dstoffset |
| LocalTime.toOffsetString() | Airport dep\_port | LocalTime.toGMTString() |  |
| LocalTime.getTime() |  |  |  |
| Expected Result | "10 May 01:04 EDT" | | |

By testing this method of unit Flight, we can verify if other units’ bound methods and paths are able to work well. The results of JUnit test case showed the same output string as we expected.

6. Pairwise Testing

In the unit testing of Object SearchResults, we deployed pairwise testing for one equivalence input class to verify if a strategy that extend query airports range can improve the probability to get more flight plans. We input all the possible combinations of two airports from the database as the depart and arrival airports in the situation that searching the 1 stop flight plans in the day 2015\_05\_11. All the input airport combina-tions that the unit cannot find flights are printed with a counter.

|  |  |  |
| --- | --- | --- |
|  | Input for depart and arrival airports, date code, stopover number, window time | Test Results |
| before modification | All combination of airports, “2015\_05\_11”,1, 0l | No Flight: 268 |
| after modification | All combination of airports, “2015\_05\_11”,1, 0l | No Flight: 80 |

Test details can be found in the test log file, screen shots and JUnit test case: SearchResultsPairwise-Testing.java. We found that by modifying the core algorithm by adding query airports range we can improve the probability by sacrificing a little time efficiency.

III. System Testing:

1. User Interface — Usability Test:

2. Performance:

3. Compatibility

4. Exceptions

5. Scalability

6. Installation

7. System Behavior

8. Prioritized Test Case

9. Valid inputs

10. Invalid Input

11. Unexpected Inputs

12. Unexpected Environment