**WRITTEN REPORT**

Problem Description

This project is based on a hotel booking website. An example of such websites can be trivago, expedia or tripcentral. Within these websites, one of the things you can do is view different hotels and compare their deals that they offer. This project follows similar principle of viewing different deals for different destinations. A customer interacts with our system when he/she chooses to go on vacation in the 5 different countries; Chile, Italy, Barbados, India, or Japan. Each country has a list of different deals, and each deal has its own airline, rating, duration of stay, and the price of the deal. After choosing the country and stating the number of passengers expected to travel, the customer might receive a list of deals. The major problem for the customer or the user is the list of deals displayed are random and difficult to follow or compare. To simplify and make the system more user-friendly, a great solution is to sort the list of deals to be displayed by an orderly fashion.   
  
Before displaying the list of deals, the user or customer is asked to choose one of the four elements or properties (airline, rating, duration of stay, cost) to sort the list by. The next step for the customer is to choose how the list to be sorted (high to low, or low to high) with the one of the properties of the deal they chose before. The system now is more user-friendly and has an organized structure because the list of deals for that country will be displayed based on the customer’s specifications. This concept of sorting out the results by high to low or low to high strongly resembles many real-world examples.

Functional Requirements

*Case 1*

* Name of case: UserInterface
* Participating actors: person
* Entry conditions: None. There are no entry conditions. Interface opens right away when the program runs.
* Flow of events:

1. The customer can choose a country
2. The customer can view flight information on the chosen country.
3. The customer can sort the details, depending on the category they choose.
4. The customer can sort the information from low to high or high to low.
5. Exit the class and return back to main menu

* Exit conditions: Enter ‘0’ to go to the main menu. Then enter “yes” to exit the program.
* Quality Requirements: None

*Case 2*

* Name of case: Country
* Participating actors: person
* Entry conditions: Chosen country
* Flow of events:

1. Depending on the chosen country an array of information of the flights is stored into an object.
2. When the person decides to view the information, all the values are converted into a formatted toString.

* Exit conditions: No exit conditions
* Quality Requirements: None

Design

The primary design pattern used in this project is the bridge pattern. The bridge pattern in this project decouples abstraction from its implementation so the abstraction(s) and implementation(s) vary independently. To resemble the real-life example of a viewing different deals on hotels/vacation deals, the customer can sort the results by high to low or low to high. This pattern, in other words, provides two layers of abstractions. Referring to the UML class diagram, the following classes express the abstractions/refined abstractions: sorter, sortActivity, sortDuration, sortRating, and sortPrice. The implementor is <<interface>>sorter and the concrete implementors are sortHighToLow and sortLowToHigh. The abstractions and the implementors vary independently because any of the abstractions (airline, duration, rating, price) can implement any of the concrete implementors (high to low, low to high) without affecting any other components. The abstraction and implementations are placed in separate hierarchies and is therefore beneficial to this project because another property of a flight deal can be added and can use any of the concrete implementors. Another implementor or method of sorting can also be added independently and can be used by any of the abstractions or properties of a flight deal.

Testing *Flight Class:*

|  |  |  |
| --- | --- | --- |
| **Test case ID** | **Test Case Description** | **White box/ Black box** |
| 01: testGetPrice(); | An object type flight was created with test parameters. The instance test variable price is passed in the test parameter as 1222.99 so the result when using instance.getPrice() in this situation should be 1222.99. This test will be successful if 1299.99 is obtained. | Black box |
| 02: testSetPrice(); | A instance test variable price is created and initialized as 1222.99. Then an object type flight was created with test parameters. The instance test variable price is passed to the customer class with the method setPrice().This test will be successful if the instance test variable price and instance.getPrice() are equal. | Black box |
| 03: testGetDuration(); | An object type flight was created with test parameters. The instance test variable duration is passed in the test parameter as 4.0 so the result when using instance.getDuration() in this situation should be 4.0. This test will be successful if 4.0 is obtained. | Black box |
| 04: testSetDuration(); | A instance test variable duration is created and initialized as 4.0. Then an object type flight was created with test parameters. The instance test variable duration is passed to the customer class with the method setDuration().This test will be successful if the instance test variable duration and instance.getDuration() are equal. | Black box |
| 05: testGetRating(); | An object type flight was created with test parameters. The instance test variable rating is passed in the test parameter as 4.5 so the result when using instance.getRating() in this situation should be 4.5. This test will be successful if 4.5 is obtained. | Black box |
| 06: testSetRating(); | A instance test variable rating is created and initialized as 4.5. Then an object type flight was created with test parameters. The instance test variable rating is passed to the customer class with the method setRating().This test will be successful if the instance test variable rating and instance.getRating() are equal. | Black box |
| 07: testGetAirlLne(); | An object type flight was created with test parameters. The instance test variable airline is passed in the test parameter  as “AirCanada” so the result when using instance.getAirLine() in this situation should be “AirCanada”. This test will be successful if “AirCanada” is obtained. | Black box |
| 08: testSetAirLine(); | A instance test variable airline is created and initialized as “AirCanada”. Then an object type flight was created with test parameters. The instance test variable airline is passed to the customer class with the method setAirLine().This test will be successful if the instance test variable airline and instance.getAirLine() are equal. | Black box |
| 09: testToString(); | An object type flight was created with test parameters. The instance test variable expResults stored the test value. In this case the expResults is “Airline: AirCanada: Rating: 4.5, Duration of your stay: 4.0 days, Price: $1222.99”. This test will be successful if expResults and toString values are equal. | Black box |

*Country Class*:

|  |  |  |
| --- | --- | --- |
| **Test case ID** | **Test Case Description** | **White box/ Black box** |
| 10: testAddFlight(); | An object type country was created. Four test parameters: price, duration, rating and airline were initialized and the instance test variables were passed to the country class with the method addFlight().This test will be successful if the instance test variables were accurately added. This can be checked by comparing the tostring output. | Black box |