

The ABC's

ALPHA JET

BOOKING SYSTEM

CS4227 (Continuation of CS4125)

15147029 - Cailean Finn

15167771 - Stephen King

14160668 - Ben Smith

11098813 - Ali Al-Dobayan

New Work Carried Out

- Interceptor implemented
- Facade implemented
- Builder implemented
- Abstract Factory implemented
- Memento implemented
- Adaptor implemented
- Strategy implemented
- Refactoring
- Testing

Member Contribution for Implementation

15147029 - Cailean Finn	Refactoring, Memento, Builder, Abstract Factory Method, Facade
15167771 - Stephen King	Interceptor, Adaptor, Strategy, Testing
14160668 - Ben Smith	
11098813 - Ali Al-Dobayan	

Table of Contents

New Work Carried Out	1
Member Contribution for Implementation	1
Table of Contents	2
Requirements	4
Scenario:	4
Use Case Diagrams:	5
Use Case Descriptions:	6
Detailed Use Case Descriptions:	9
Quality Attributes:	11
Architectural and Design Patterns	12
Interceptor:	12
Strategy:	12
Adaptor:	12
Abstract Factory Method:	13
Memento:	13
Builder:	13
Facade (Independent Research):	13
System Architecture	14
Package Diagram:	14
Architectural Decisions:	14
Structural and Behavioural Diagrams	15
Class Diagram:	15
Communication Diagram:	15
Code	16
Coding Fragments:	16
Interceptor:	16
Adaptor:	17
Strategy:	18

Facade:	19
Abstract Factory:	19
Memento:	19
Builder:	19
Added Value	19
Refactoring:	19
Evidence of Testing	20
Evaluation	21
References	22

Requirements

Scenario:

Air travel is becoming more and more popular for both leisure and work. However, booking a flight can be complicated and time consuming due to the needlessly complex user interface and the slow to load flight pages, this is a problem we will aim to address with are flight booking system.

Our flight booking system will allow an efficient and easy to use tool for customers and the airline company to use.

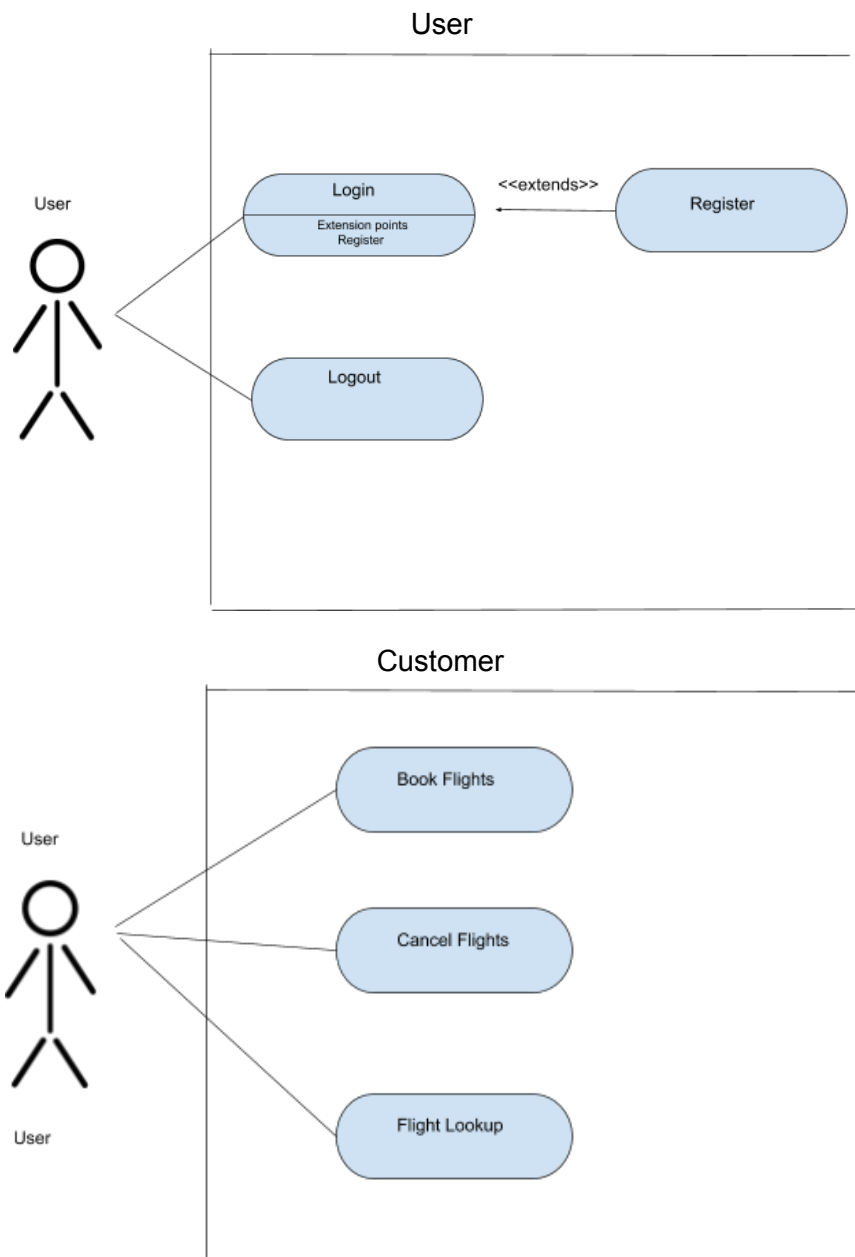
Customers can search for flights easily and quickly by simply selecting a departing airport a destination and the date, then be presented with flights on that date and flights 3 days before and after that date. From there they can book a flight, select which class they wish to fly in and go ahead with the transaction. The flight length(miles) will then be added to the users account details so frequent flyers can get discounts on flights which will be reflected when they next purchase a flight.

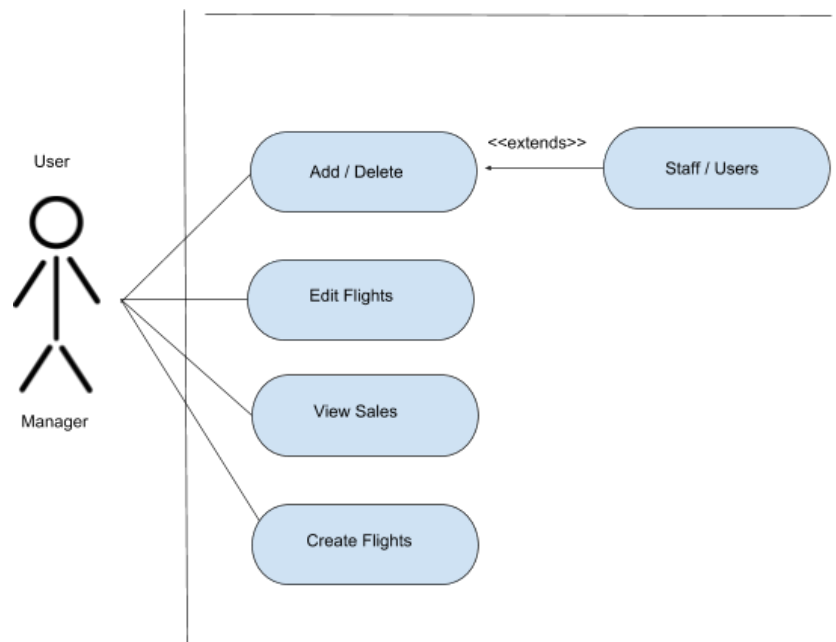
The managers and receptionists can also view all flights and make necessary changes to flights such as price and departure time. Managers will also be to cancel and create flights, selecting the number of seats available and which luxuries like Wi-Fi and inflight entertainment system will be available for that flight.

For the business logic we will implement a Membership system. The system will offer different discounts for the tier of membership in which the user has. Tiers will consist of Standard, Bronze, Silver and Gold; each offering different benefits to the user and for the higher tier the user is he better benefits he receives while booking flights in the future. For a user to reach a certain tier there will be multiple goals the user will have to achieve to advance to the next tier. Points will diminish after a certain amount of time if the user is not active, as well if the user is late for checking in, he/she will lose points.

This project is a continuation of the project submitted for last semesters module CS4125. We were confident we could work on the already existing application using methods that we were taught in CS4227 to create a more refined system.

Use Case Diagrams:





Use Case Descriptions:

Use Case 1: Log Out

Actor Action	System Response
1 - User selects log out button	2 - System logs the user out and displays log out screen

Use Case 2: User cancels flight

Actor Action	System Response
1 - User selects "Cancel Flight" option 3 - User completes cancellation form	2 - System displays "Cancel Flight" window 4 - System displays successful cancellation window and flight is cancelled
Alternative Route	
1(a) - User cancels cancellation	2(a) - System returns to main window
Non-Functional Requirements	

Performance: successful cancellation window shows, and the flight is immediately cancelled

Use Case 3: User “My flights”

Actor Action	System Response
1 - Users selects “My Flights” option	2 - System displays all of the user flights
Alternative Route	
1(a) - User has no flights booked, so no bookings show	2(a) - System returns to main window as it is unable to open “My Flights” Window
Non-Functional Requirements	

Use Case 4: Staff views flights

Actor Action	System Response
1 - Staff presses “View flights” button	2 - System displays “View flights” window
Alternative Route	
1(a) - “View flights” button not working	2(a) - System does not display any result
Non-Functional Requirements	
<u>Performance:</u> flights list displayed immediately after pressing “View flights” button.	

Use Case 5: Staff edit flights

Actor Action	System Response
1 - Staff presses “Edit flights” button	2 - System displays edit options: <ul style="list-style-type: none"> - Edit time - Edit destination
Alternative Route	
1(a) - Error message displayed that cannot edit flights	2(a) - There is an overlap in either of the edit options
Non-Functional Requirements	

Use Case 6: Staff updates user info

Actor Action	System Response
1 - Staff presses “Edit Users”	2 - Systemes displays user info
Alternative Route	
1(a) - Error Message displayed that cannot open edit users	2(a) - There are no users within the database
Non-Functional Requirements	

Use Case 7: Manager creates flight

Actor Action	System Response
1 - Manager presses "Create flight" button 3 - Select departing airport 4 - Select destination airport 5 - Select other luxury options 7 - Set price for each class 8 - Create flight	2 - System displays "Create flight" window
Alternative Route	
1(a) - "Create flight" button not working	2(a) - System does not display any window and displays error message
Non-Functional Requirements	
<u>Performance:</u> "Create flight" window displayed immediately after pressing "Create flight" button.	

Use Case 8: Manager view sales

Actor Action	System Response
1 - Manager presses "View sales" button 3 - Manager can client on different sales and view information about it	2 - System displays "View sales" window
Alternative Route	
1(a) - "View sales" button not working	2(a) - System does not display any window and displays error message
Non-Functional Requirements	
<u>Performance:</u> sales list displayed immediately after pressing "View sales" button.	

Use Case 9: Manager cancels flight

Actor Action	System Response
1 - Manager opens "Cancel Flight" window and choses a flight to cancel	2 - The flight is then cancelled and will not appear in the flight list
Alternative Route	
1(a) - Manager is unable to cancel the flight and displays an error message	2(a) - Flight has already departed and is unable to delete it from the database
Non-Functional Requirements	

Use Case 10: Manager adds staff

Actor Action	System Response
1 - Manager presses "Add User" and creates a staff account	2 - Account is created as a staff member
Alternative Route	

1(a) - User details are insufficient, error is given : "Insufficient details"	2(a) - Unable to create user so error is given : "Unable to create user"
Non-Functional Requirements	

Use Case 11: Manager deletes staff

Actor Action	System Response
1 - Manager presses "Delete User" while selecting a staff member	2 - Account is deleted
Alternative Route	
1(a) - Manager is unable to delete the user and an error message is shown	2(a) -
Non-Functional Requirements	

Use Case 12: Manager deletes user

Actor Action	System Response
1 - Manager presses "remove" on a users details	2 - System displays "User removed"
Alternative Route	
1(a) - Error message displayed as there was no user selected/multiple users selected	2(a) - Error message displayed if "remove" button is not working
Non-Functional Requirements	

Detailed Use Case Descriptions:

Use Case 1	User Logs In	
Goal in Context	For the user to be logged into the AlphaJet system	
Scope & Level		
Preconditions	User is already registered	
Success End Conditions	User has logged in	
Failed End Conditions	User has not logged in	
Primary, Secondary, Actors	Manager, Staff, User (Customer)	
Trigger	User has started the application	
DESCRIPTION	STEP	ACTION
	1.	Actor is greeted by login screen

	2.	Actor type in username and password
	3.	Presses login button
	4.	Login details are checked against the database
	5.	If the details match the actor is greeted with the dashboard that corresponds to the user type
EXTENSIONS	STEP	BRANCHING ACTION
	4a.	The user's details are found to be incorrect 4a1. User is told details are incorrect and the user can try again
VARIATIONS		

Use Case 2	Customer Books Flight	
Goal in Context	For a customer to successfully book a flight	
Scope & Level		
Preconditions	User is signed in and is a customer	
Success End Conditions	The customer has booked a flight	
Failed End Conditions	The customer has not booked a flight	
Primary, Secondary, Actors	AlphaJet, User (Customer)	
Trigger	User has searched for a flight	
DESCRIPTION	STEP	ACTION
	1.	User clicks on the desired flight they want to book
	2.	User selects the seats they want to buy from first, business and economy
	3.	User selects the amount of baggage he will be bringing
	4.	User selects what amenities they want
	5.	A summary and final price is shown
	6.	User then selects make a payment
	7.	Input card details and re-enters password
	8.	Payment and confirmation are shown, and the flight details are viewed
EXTENSIONS	STEP	BRANCHING ACTION
	2a.	There are no seats left in the desired class 2a1. User selects a different flight or class
	4a.	A user can spend points on amenities 4a1. User checks user points box
	7a.	Password is incorrect 7a1. User is told and can re-enter password
VARIATIONS		

Use Case 3	Create Flight	
Goal in Context	to create a new flight	
Scope & Level		
Preconditions	Signed in as a manager	
Success End Conditions	A new flight has been created	
Failed End Conditions	A new flight has not been created	
Primary, Secondary, Actors	Manager	
Trigger	Manager clicks on create flight from his dashboard	
DESCRIPTION	STEP	ACTION
	1.	Manager presses "Create flight"
	2.	Manager selects departing airport
	3.	Selects destination airport
	4.	Selects date and time of departure
	5.	Selects number of seats available for the flight
	6.	Select amenities available in the flight
	7.	Set price for each class
	8.	Set price for each amenity
	9.	Manager clicks create flight
EXTENSIONS	STEP	BRANCHING ACTION
	9a.	Manager has not filled out everything 9a1. System notified him of which sections have not been filled out
VARIATIONS		

Quality Attributes:

For our project we decided to use Java for multiple reasons. First we are all very familiar with using Java as it has been a core language throughout our years in college studying Computer Systems. It also benefits us while studying System Analysis and

Design as Java is the language of choice in the module. There are also many resources online available in Java to help us while creating the system. Another benefit is Java allows for strong portability.

For Security and Usability reasons we implemented a role base system, which allows us to separate functions by permissions.

We used “MySQL JDBC Driver” to allow us to access MySQL databases within Java. The decision was made to use this driver as the majority of the team was already very familiar with how the library worked. It provides queries and methods for updating data in the database and is considered to be very low-level and prone to very little errors. The driver is a pure Java implementation of the MySQL Protocol and is independent of the MySQL Client libraries.

Readability was also another key attribute discussed as we wanted to create code with high-level readability so when one of the team needs to extend the functionality of the code even further. We stuck with similar naming conventions throughout the project and have clear comments describing the process of each class/method.

For the GUI we used JFrame as it provided us with the tools to create simple and user-friendly GUI to give users the best experience possible as it allows anyone to be able to access all functionalities of the program with ease.

Architectural and Design Patterns

Interceptor:

The Interceptor is an architectural pattern. It is a software design pattern that is used when software systems or frameworks want to offer a way to change or augment, their usual processing cycle.

In our case, during Login or Registration if an exception error has occurs, a new Interceptor is created and dispatched, the Interceptor will then print out the level of error that had occurred e.g. Severe, the message Error Detected, the type of error that has occurred and the time that the error had occurred. The Interceptor will also log this error on the database in the loginlogger database.

Strategy:

The Strategy design pattern is a behavioural pattern. It allows for similar algorithms to be defined and encapsulated in their own classes. The algorithm to be used for a particular purpose may then be selected at run-time according to your requirements.

In our case, we used the Strategy design pattern to multiple the points that the user gains once they book a flight based off the level Gold, Silver, Bronze or Standard that they have obtained through gathering points. Once the user books a flight, a new instance of points is created which finds the level of the user and finds the correct class multiplier that refers to the user's level. If the user is Gold the points will be multiplied by 4, if the user is Silver the points will be multiplier by 3, if the user is Bronze the points will be multiplier by 2 and the if the user is Silver the points will be multiplier by 1.

Adaptor:

The Adaptor is a structural design pattern. The Adaptor pattern allows two incompatible types to communicate, where one class relies upon a specific interface that is not implemented by another call, the Adaptor acts as a translator between the two types.

We implemented this pattern in our case to differentiate the different size/weights of a Mid Size Jet, a Light Jet and a Heavy Jet. This pattern is called when creating the airlines for the Flights, a new AAdaptor is called and this in turn either sets the baggage to 30 if it is medium or if it is a Light Jet or Heavy Jet, it will call a new ConnectorAdaptor and we call the correct method of planeType() for the name entered in.

Abstract Factory Method:

The abstract factory method design pattern is a creational design pattern. It allows for the creation of product families without specifying their concrete class. This is done by defining an interface for creating all the distinct products but the creation of the products are left to the concrete factory classes, where each factory corresponds to a product variety.

The abstract factory method was implemented within the project for the creation of different airplanes and their flight attributes such as the maximum baggage capacity and the flights price for each class.

Memento:

The memento design pattern falls under the behavioural design pattern category. The memento is used for the restoration of a state of an object. Three actor classes are necessary for the correct implementation of the design pattern, these are the Memento, Originator and CareTaker. The Memento stores the state of the object that needs to be restored. The Originator creates and sets states and the CareTaker objects responsibility is to restore the object to the desired state from the Memento.

In our case, the design pattern was implemented to restore the state of the options selected within the Booking Menu when a booking is cancelled, so the customer can view the cost of the booking post discount on the Membership Menu.

Builder:

The builder design pattern is a creational design pattern. It enables for the creation/building of complex objects step by step, with the final step returning the object that has been built.

For the implementation of the builder design pattern in our project, it was used for the creation of Booking objects. A Booking Builder class was created to pass in parameters for a certain booking and then the build() method within this class would return a object of type Booking.

Facade (Independent Research):

The last design pattern implemented was the Facade. The Facade was the one additional pattern researched that was not covered within the CS4125 and CS4227 lectures. The design pattern is a Gang of Four design pattern and is categorised as a structural design pattern. The main purpose of the Facade is to hide the complexities of a system while providing a simplified interface for the client, so they can access the system.

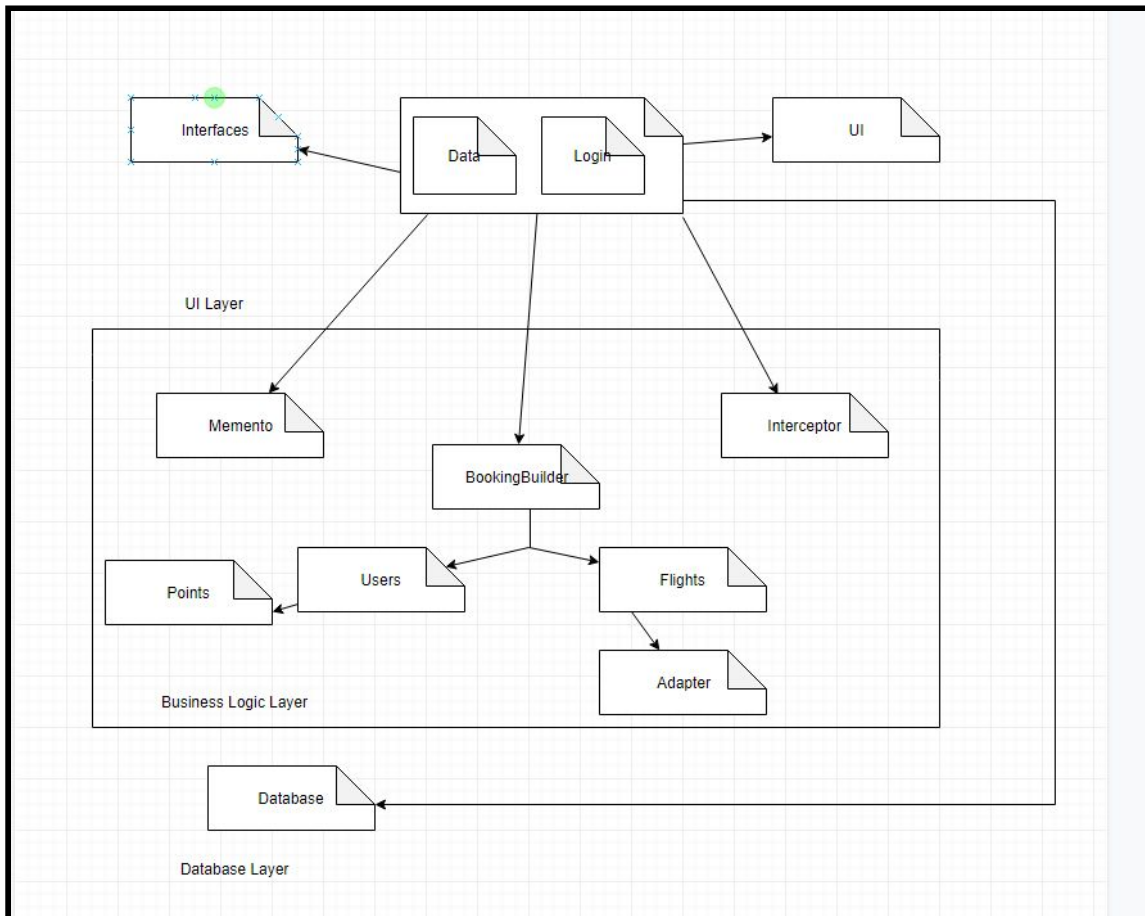
This enables the software library to be understood and tested more easily, seeing as the Facade has very convenient methods for related tasks. It also reduces dependencies on outside code on the workings inside the library, allowing for more flexibility when developing the system.

For our implementation of the Facade, it was used for the registration and login operation. This would allow for the verification of a login. Checking the syntax of the details entered within each field, which was done through the LoginEmailCheck class and LoginPassCheck class. It was also used to verify the creation of a new user through the registration operation. Once again the RegNameCheck class and RegContactCheck class were responsible for checking the syntax.

The Check objects for both Login and Registration were created within the two Facade classes, LoginFacade and RegisterFacade. These classes made calls to the database that allowed us to verify the users email and password, along with providing the parameters for the creation of a user.

System Architecture

Package Diagram:



Architectural Decisions:

When designing the architecture for our system, we decided on splitting it into 3 components. The User Interface, Business Logic and Data layer. The architectural pattern we plan on using is the Model-View-Controller as it is commonly used when developing applications with user interfaces. The UI layer would be the view, the business layer would be the controller and the database would be the model.

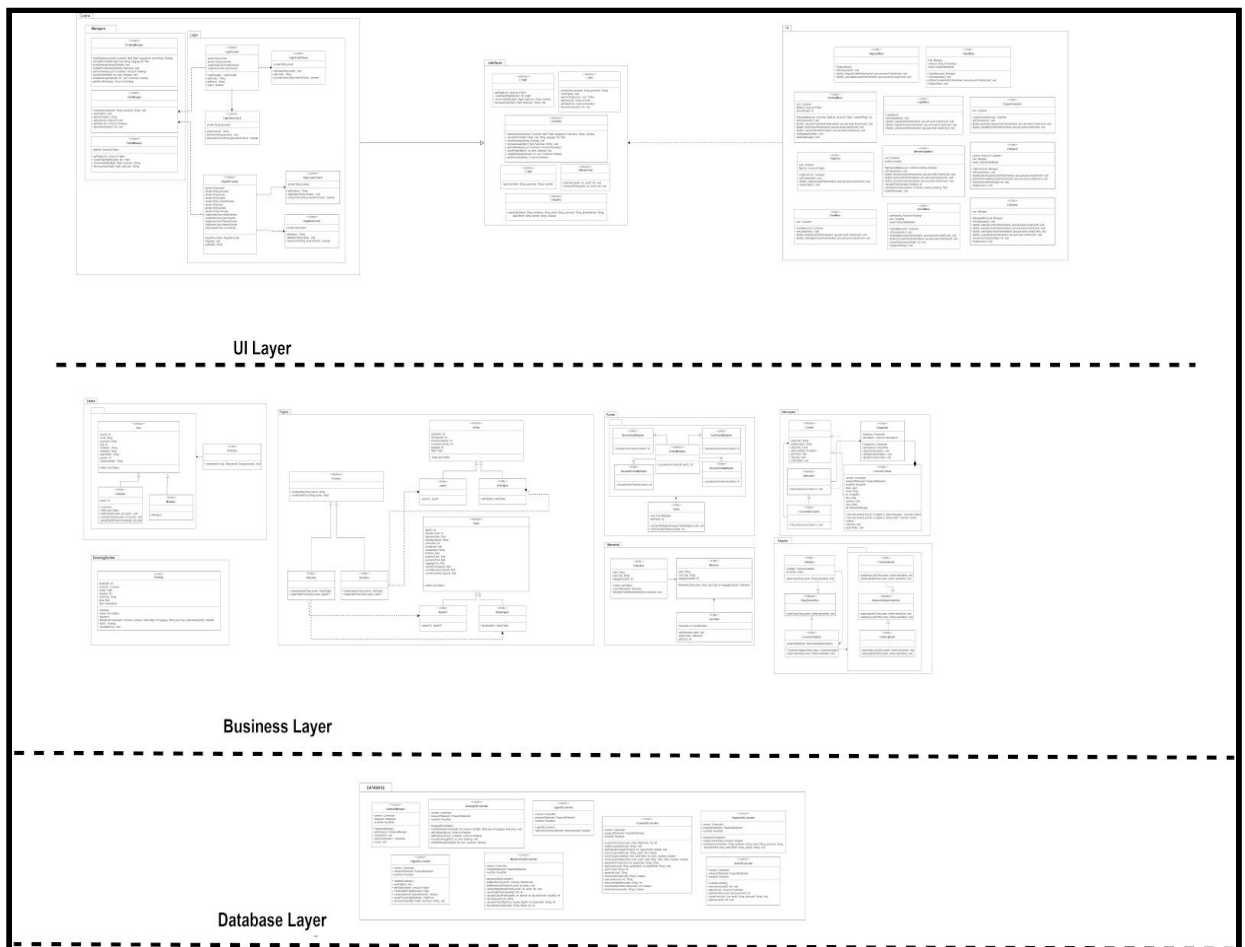
This pattern allows us to separate the UI layer from the Business layer, so if we needed to make any changes to the UI layer it would have no effect on the business logic and would only require swapping the UI code.

The Business Logic layer is the core of our system, which contains all the main processes such as **FlightManager** and **Booking**. This layer allows for information to be exchanged from the UI layer to the Data layer, which maintains the business logic of the system.

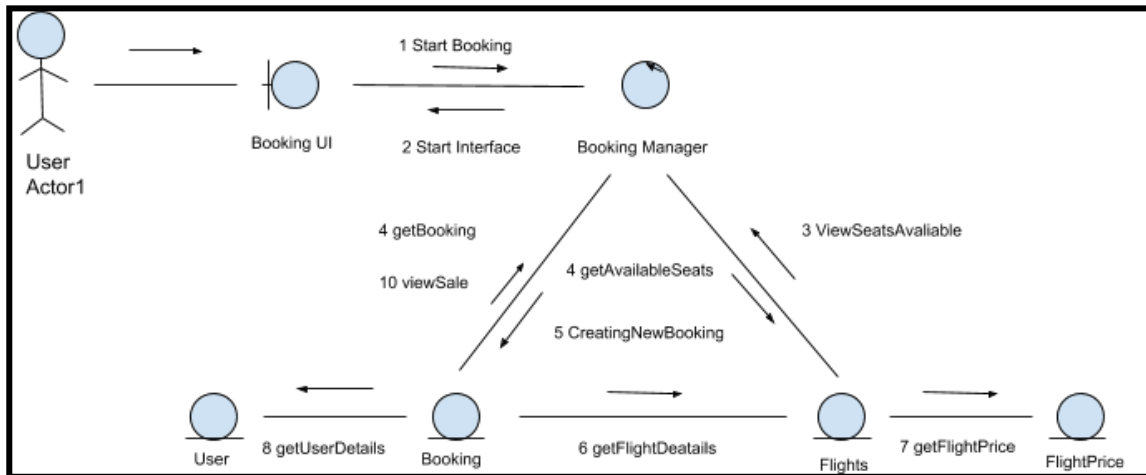
Finally, the Data layer communicates with the business logic layer by sending data to and from it, then storing the data received from the business logic layer.

Structural and Behavioural Diagrams

Class Diagram:



Communication Diagram:



Code

Coding Fragments:

Interceptor:

ConcreteContext.java:

```
@Override
public void Service() {
    logger l = Logger.getLogger(ConcreteContext.class.getName());
    String lnSkip = "\n";
    l.log(this.getLevel(), lnSkip + (this.getMessage() + lnSkip + this.getException().getClass().getSimpleName() + lnSkip + this.getTime()));
}

@Override
public void putInTable()
{
    db = DatabaseManager.getInstance();
    try {
        db.connectDB();
    } catch (Exception ex) {
        msg = "Error Detected";
        logger.getLogger(ConcreteContext.class.getName()).log(Level.SEVERE, msg, ex);
    }
    try {
        this.connect = db.returnConnection();
    } catch (Exception ex) {
        msg = "Error Detected";
        logger.getLogger(ConcreteContext.class.getName()).log(Level.SEVERE, msg, ex);
    }

    try {
        preparedStatement = connect.prepareStatement("insert into abcs.loginlogger values (?, ?, ?, ?)");
        preparedStatement.setString(1, this.getLevel().getClass().getName());
        preparedStatement.setString(2, this.getMessage());
        preparedStatement.setString(3, this.getException().getClass().getSimpleName());
        preparedStatement.setDate(4, newTime);
        preparedStatement.executeUpdate();
    } catch (SQLException ex) {
        msg = "Error Detected";
        logger.getLogger(ConcreteContext.class.getName()).log(Level.SEVERE, msg, ex);
    }
}
```

In the snippet above, the ConcreteContext is for an instance of the Interceptor with the level of severity of the error, the exception itself and the message "Error Detected". The methods shown above are then entered and the parameters above are shown to the users.

Dispatcher.java:

```

* @author user1
*/
public class Dispatcher {

    private static Dispatcher dispatcher;
    ArrayList<Interceptor> interceptors = new ArrayList<Interceptor>();

    private void Dispatcher(){
        interceptors = new ArrayList();
    }

    public static Dispatcher getInstance(){
        if(dispatcher == null){
            dispatcher = new Dispatcher();
            return dispatcher;
        }
        else{
            return dispatcher;
        }
    }

    public synchronized void register(Interceptor i){
        interceptors.add(i);
    }

    public synchronized void unregister(Interceptor i){
        interceptors.remove(i);
    }

    public void dispatch(Context item){
        ArrayList interceptorList;

        synchronized(this){
            interceptorList = (ArrayList) this.interceptors.clone();
        }

        for (int j = 0; j < interceptorList.size(); j++)
        {
            Interceptor interceptor = (Interceptor) interceptorList.get(j);
            interceptor.ConsumeService(item);
        }
    }
}

```

In the snippet above, once an Interceptor is initiated, it adds the Interceptor to an arraylist of Interceptors in the Dispatcher class.

The Interceptor Pattern allows for easy modifiability and reusability as the Exception will change based on the error and can be easily be reused throughout the project using try and catch blocks.

Adaptor:

AAdaptor.java:

```

package Adapter;

import Flights.Airline;

/**
 *
 * @author Stephen
 */
public class AAdapter implements AdapterInterface {
    ConnectorAdapter cAdapter;
    Airline thisAirline;

    @Override
    public void planeType(String name, Airline newAirline) {
        if(name.equalsIgnoreCase("Mid size Jet")){
            newAirline.setBaggage(30);
        }

        else if(name.equalsIgnoreCase("Light Jet") || name.equalsIgnoreCase("Heavy Jet")){
            cAdapter = new ConnectorAdapter(name);
            cAdapter.planeType(name, newAirline);
        }

        else{
            System.out.println("Invalid PlaneType: " + name + " format not supported");
        }
    }
}

```

In the snippet above, the AAdaptor class is instantiated with a name and an Airline. The corresponding method is then called based off the name that was entered. If it is a Mid

Size Jet, the weight is set to 30, if it is a Light Jet or Heavy Jet. The ConnectorAdaptor is called with a name and an Airline.

ConnectAdaptor.java:

```
public class ConnectorAdaptor implements AdaptorInterface {

    AdvancedAdaptorInterface advancedInterface;

    public ConnectorAdaptor(String name){
        if(name.equalsIgnoreCase("Light Jet")){
            advancedInterface = new CreateLightJet();

        }else if (name.equalsIgnoreCase("Heavy Jet")){
            advancedInterface = new CreateHeavyJet();
        }
    }

    @Override
    public void planeType(String name, Airline newAirline) {
        if(name.equalsIgnoreCase("Light Jet")){
            advancedInterface.planeLightJet(name, newAirline);
        }
        else if(name.equalsIgnoreCase("Heavy Jet")){
            advancedInterface.planeHeavyJet(name, newAirline);
        }
    }
}
```

In the snippet above, the ConnectAdaptor class is instantiated with a name and an Airline. The corresponding method is then called based off the name that was entered. If Light Jet is entered, the CreateLightJet class is called and the planeLightJet method in the CreateLightJet class is called. If Heavy Jet is entered, the CreateLightJet class is called and the planeLightJet method in the CreateHeavyJet class is called.

The Adaptor pattern allows for extensibility as other similar object to the shown above in the snippets could be created for example a Business Jet, the pattern also provides adaptability in allowing the program to adapt to which object is applicable to the name entered in.

Strategy:

Points.java:

```

package Strategy;

/**
 *
 * @author Stephen
 */
public class Points {
    private PointsMultiplier multi;
    private int newPoints;
    //this can be set at runtime by the application preferences
    public void setPointsMultiplierStrategy(PointsMultiplier multi) {
        this.multi = multi;
    }

    //use the strategy
    public int FindCalculatePoints(int points) {
        newPoints = multi.calculateUserPoints(points);
        System.out.println(newPoints);
        return newPoints;
    }
}

```

In the snippet above, the Points class is instantiated and the findCalculateUserPoints method is called which figures out what level the user is and finds the corresponding method for the user level.

GoldPointsMultiplier.java:

```

/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package Strategy;

/**
 *
 * @author Stephen
 */
public class GoldPointsMultiplier implements PointsMultiplier {
    public int calculateUserPoints(int points) {
        int totalPoints;
        totalPoints = points *4;
        return totalPoints;
    }
}

```

In the snippet above, the GoldPointsMultiplier is called if the user level is of Gold level and the points that we gained when booking a flight are quadrupled due to the user's level.

This Strategy pattern allows for extensibility, as more multipliers could be added when other tiers are added.

Facade:

LoginFacade.java:

```

/**
 *
 * @author Cailean
 */
public class LoginFacade {

    private String email;
    private String password;

    LoginEmailCheck emailChecker;
    LoginPassCheck passChecker;
    String message;
    String[] test;

    public LoginFacade(String newEmail, String newPass){
        email = newEmail;
        password = newPass;
        passChecker = new LoginPassCheck();
        emailChecker = new LoginEmailCheck();
        passChecker.setPassword(password);
        emailChecker.setEmail(email);
    }

    public String getEmail(){ return email;}
    public String getPass(){return password;}

    public boolean login() throws Exception{
        test = new String[3];
        if(emailChecker.accountActive(getEmail()) && passChecker.passwordCheck(getPass())){
            LoginDBController db = new LoginDBController();
            I_User userMan = new UserManager();

            if(db.loginCheck(email, password)){
                try{
                    userMan.createUser(email, password);
                    return true;
                }
            }

            catch(Exception e)
            {
                message = "Error Detected";
                Interceptor in = new ConcreteInterceptor();
                Dispatcher.getInstance().register(in);
                Dispatcher.getInstance().dispatch(new ConcreteContext(Level.SEVERE, e, message, email));
            }
        }
    }
}

```

LoginEmailCheck.java:

```

/*
 *
 * @author Cailean
 */
public class LoginEmailCheck {

    private String email;

    public void setEmail(String email){this.email = email;}

    public String getEmail(){ return email;}

    public boolean accountActive(String emailToCheck){

        if(emailToCheck.equals(getEmail()) ){
            String emailRegex = "^[a-zA-Z0-9_+&*-]+(?:\\.|"+
            "[a-zA-Z0-9_+&*-])*@" +
            "(?:[a-zA-Z0-9-]+\\.|[a-z]" +
            "A-Z){2,7}$";

            Pattern pattern = Pattern.compile(emailRegex);
            Matcher matcher = pattern.matcher(email);
            if(!matcher.matches()){
                return false;
            }
            return true;
        }

        return false;
    }

}
}

```

The code fragment snippets above show how the Facade design pattern was implemented into our system, hiding the complexities of the system. When a LoginFacade object is created the email and password that was entered by the user is passed in and one object of LoginEmailCheck and LoginPassCheck are created and the email and password values are set in the respective objects. Then the login()

method can be run, checking both the email and password entered, returning either true or false. If true is returned, the users login details are valid and is logged in.

Using this pattern makes the library highly readable and as I stated above, gives more flexibility when developing the system.

Memento:

Memento.java:

```
/**
 *
 * @author Cailean
 */
public class Memento {
    private String state;
    private String classType;
    private int baggageQuantity;

    public Memento(String state, String classType, int baggageQuantity){
        this.state = state;
        this.classType = classType;
        this.baggageQuantity = baggageQuantity;
    }

    public String getState(){
        return state;
    }

    public String getClassType(){
        return classType;
    }

    public int getBaggageQuantity(){
        return baggageQuantity;
    }
}
```

CareTaker.java


```

/**
 *
 * @author Cailean
 */
public class CareTaker {
    private List<Memento> mementoList = new ArrayList<Memento>();

    public void add(Memento state){
        mementoList.add(state);
    }

    public Memento get(int index){
        return mementoList.get(index);
    }

    public int getSize(){
        return (mementoList.size() - 1);
    }
}

```

Originator.java:

```

/**
 *
 * @author Cailean
 */
public class Originator {
    private String state;
    private String classType;
    private int baggageQuantity;

    public void setState(String state){
        this.state = state;
    }

    public String getState(){
        return state;
    }

    public void setClassType(String classType){
        this.classType = classType;
    }

    public String getClassType(){
        return this.classType;
    }

    public void setBaggageQ(int baggageQ){
        this.baggageQuantity = baggageQ;
    }

    public int getBaggageQ(){
        return this.baggageQuantity;
    }

    public Memento saveStateToMemento(){
        return new Memento(state, classType, baggageQuantity);
    }

    public void getStateFromMemento(Memento memento){
        state = memento.getState();
        classType = memento.getClassType();
        baggageQuantity = memento.getBaggageQuantity();
    }
}

```

The code snippets above, show the use of the Memento design pattern in our system. The Memento is used to store the values of classType and baggageQuantity. These

values and the state are stored once the user continues to the MembershipMenu. So, if the user cancels the Booking at the MembershipMenu, that state will be restored.

Builder:
Booking.java

```

/**
 *
 * @author Cailean
 */
public class Booking {

    private int bookingID;
    private Customer customer;
    private Airline flight;
    private int baggage;
    private String classType;
    private float price;
    private Date bookingDate;

    public Booking(){

    }

    public int getBookingID() {
        return bookingID;
    }

    public Customer getCustomer() {
        return customer;
    }

    public Airline getFlight() {
        return flight;
    }

    public String getClassType() {
        return classType;
    }

    public int getBaggage() {
        return baggage;
    }

    public float getPrice() {
        return price;
    }
    public void setPrice(float price) {
        this.price = price;
    }

    public Date getBookingDate() {
        return bookingDate;
    }

    private Booking(BBuilder builder) {

        this.bookingID = builder.bookingID;
        this.customer = builder.customer;
        this.flight = builder.flight;
        this.baggage = builder.baggage;
        this.classType = builder.classType;
        this.price = builder.price;
        this.bookingDate = builder.bookingDate;

    }
}

```

```

public static class BBuilder{

    private int bookingID;
    private Customer customer;
    private Airline flight;
    private int baggage;
    private String classType;
    private float price;
    private Date bookingDate;

    public BBuilder(int bookingID, Customer customer, Airline flight, int baggage, String classType, Date bookingDate){

        this.bookingID = bookingID;
        this.customer = customer;
        this.flight = flight;
        this.baggage = baggage;
        this.classType = classType;
        this.price = price;
        this.bookingDate = bookingDate;
        calculatePrice();

    }

    public Booking build(){
        return new Booking(this);
    }
}

```

The builder design pattern implementation is shown above in the snippets. The builder is used for the creation of Booking objects. Using the builder makes it quite easy to maintain the code if the number of fields required to build the object are more than four. It provides more robustness to the system, as only one object is available to the client.

Added Value

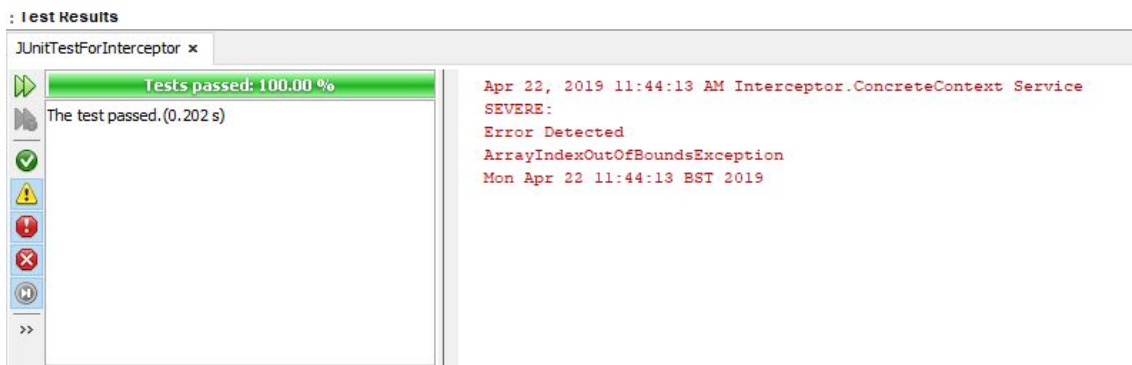
Refactoring:

Evidence of Testing

JUnitTestForInterceptor.java:

This JUnit Test was used to ensure that the Interceptor method was successfully passing as expected when an error occurs.

```
public class JUnitTestForInterceptor {  
    public JUnitTestForInterceptor() {  
    }  
  
    @Test  
    public void testInterceptor(){  
        String[] test = new String[3];  
        String message = "";  
  
        try{  
            System.out.print(test[4]);  
        }  
        catch(Exception e)  
        {  
            message = "Error Detected";  
            Interceptor in = new ConcreteInterceptor();  
            Dispatcher.getInstance().register(in);  
            Dispatcher.getInstance().dispatch(new ConcreteContext(Level.SEVERE, e, message));  
        }  
    }  
}
```



JUnitTestForStrategy.java:

This JUnit Test was used to ensure that the Strategy method was successfully passing as expected to add multipliers when obtaining books when booking flights.

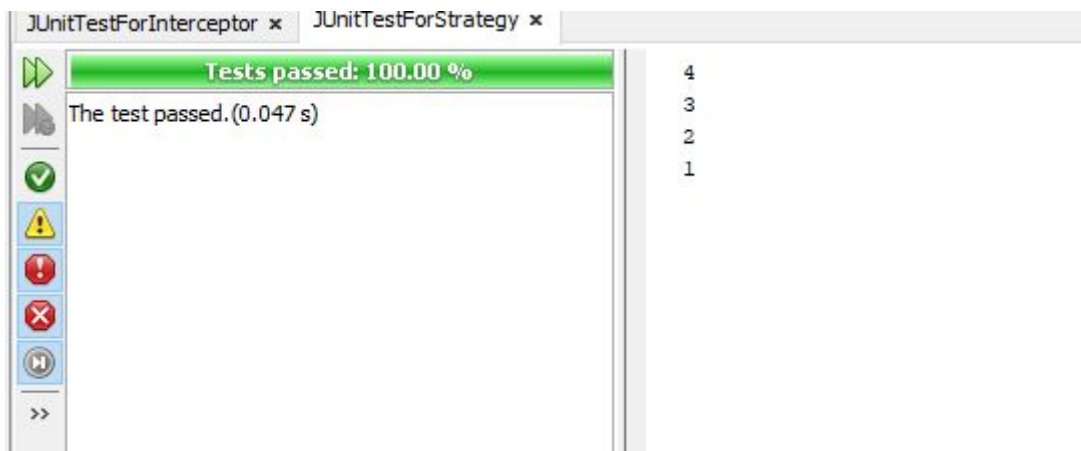
```

public class JUnitTestForStrategy {

    public JUnitTestForStrategy() {
    }

    @Test
    public void testStrategy(){
        int points = 1;
        Points ctx = new Points();
        ctx.setPointsMultiplierStrategy(new GoldPointsMultiplier());
        ctx.FindCalculatePoints(points);
        ctx.setPointsMultiplierStrategy(new SilverPointsMultiplier());
        ctx.FindCalculatePoints(points);
        ctx.setPointsMultiplierStrategy(new BronzePointsMultiplier());
        ctx.FindCalculatePoints(points);
        ctx.setPointsMultiplierStrategy(new StandardPointsMultiplier());
        ctx.FindCalculatePoints(points);
    }
}

```



JUnitTestForAdaptor.java:

This JUnit Test was used to ensure that the Adaptor method was successfully passing as expected to determine what weight is applicable to either Light Jets, Heavy Jets and Mid Size Jets.

```

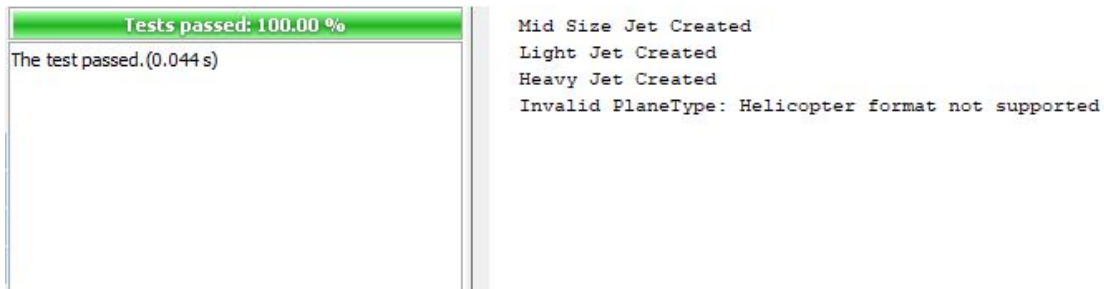
public class JUnitTestForAdaptor {

    public JUnitTestForAdaptor() {
    }

    @Test
    public void testAdapter(){
        AAdaptor planePicker = new AAdaptor();
        Airline newAirline = new Airline();
        Flight newFlight = new Flight();
        newAirline.setFlight(newFlight);

        planePicker.planeType("Mid Size jet", newAirline);
        planePicker.planeType("Light Jet", newAirline);
        planePicker.planeType("Heavy Jet", newAirline);
        planePicker.planeType("Helicopter", newAirline);
    }
}

```

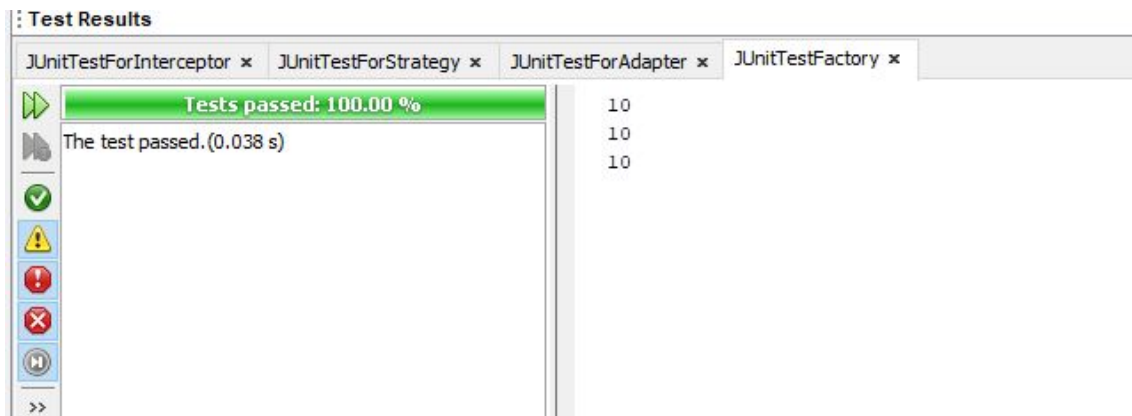


JUnitTestFactory.java:

This JUnit Test was used to ensure that the Abstract Factory method was successfully passing as expected to ensure that the Alpha airline is created correctly.

```
public class JUnitTestFactory {
    public JUnitTestFactory() {
    }

    @Test
    public void TestFactory()
    {
        FFactory flight = new AFactory();
        AlphaFlights airline = (AlphaFlights) flight.createAirplane("Alpha");
        AlphaFP flightPrice = (AlphaFP) flight.createFlightPrice("Alpha");
        airline.setFlight(flightPrice);
        int airlineNumber = 10;
        airline.setAirplaneID(airlineNumber);
        String arrivalAirport = null;
        airline.getFlight().setArrivalAirport(arrivalAirport);
        String departureAirport = null;
        airline.getFlight().setDepartureAirport(departureAirport);
        Time arrivalTime = null;
        airline.getFlight().setArrivalTime(arrivalTime);
        Time departureTime = null;
        airline.getFlight().setDepartureTime(departureTime);
        Date departureDate = null;
        airline.getFlight().setDepartureDate(departureDate);
        Date arrivalDate = null;
        airline.getFlight().setArrivalDate(arrivalDate);
        int flightID = 10;
        airline.getFlight().setFlightID(flightID);
        int currentFirst = 10;
        airline.getFlight().setCurrentFirstCapacity(currentFirst);
        int currentBusiness = 10;
        airline.getFlight().setCurrentBusinessCapacity(currentBusiness);
        int currentEconomic = 10;
        airline.getFlight().setCurrentEconomicCapacity(currentEconomic);
        System.out.println(airline.getAirplaneID());
        System.out.println(airline.getFlight().getFlightID());
        System.out.println(airline.getFlight().getCurrentBusinessCapacity());
    }
}
```



JUnitForBuilder.java:

This JUnit Test was used to ensure that the Builder method was successfully passing as expected to ensure that the Bookings for the system works properly.

```
public class JUnitTestForBuilder {

    private int bookingID;
    private int flightNumber;
    private String classType;
    private int baggage;
    private Customer user;
    private Airline flight;

    public JUnitTestForBuilder() {
    }

    @Test
    public void TestBuilder() throws Exception{
        flightNumber = 1;
        baggage = 1;
        classType = "First Class";
        bookingID = 9;









        FlightDBController db1 = new FlightDBController();
        Calendar cal = Calendar.getInstance();
        Date date=cal.getTime();

        flight = db1.createFlight(flightNumber);
        user = new Customer();

        Booking b = new Booking.BBuilder(bookingID, user, flight, baggage, classType, date).build();
        System.out.println(b.getBookingID());
        System.out.println(b.getBaggage());
        System.out.println(b.getBookingDate());
    }
}
```


Test Results

JUnitTestForBuilder x



Tests passed: 100.00 %

The test passed.(0.184 s)

1

9

1

Mon Apr 22 14:22:37 BST 2019

Evaluation

Overview

Overall this project went smoothly, the start was strong, developing the concept ideas for the project what it would do, what our individual roles would be, the language, database and primary use cases all went well, and we were unanimous in our decisions. However, when it came to the system architecture and analysis time class diagrams we struggled, we failed to understand the importance of designing a good system from the outset. We didn't know how much time it would save us and how much easier it would make implementing and understanding the project. Once we created usable and informed architecture design and time class diagrams their power became apparent. We could approach the project like professionals instead of the usual undergraduate technique of picking a starting point and seeing how it develops. After having decided and created quality diagrams the implementation went well, and we were able to create functional and expandable software.

Architecture Diagrams

Our architecture diagrams remained almost exactly the same, with only minor changes through the implementation. We used a simple n-tier architecture based on the MVC pattern. Due to the limited changes to the architecture and packages this helped implementation process enormously because, we didn't have to change our way of thinking about the code structure. This saved us a lot of time and helped our understanding of the system and the design patterns that would be implemented.

Analysis Class Diagrams

Since the original time class diagram was created without design patterns in mind the final diagram has changed quite a bit. With the design patterns and additional business logical the final diagram is far more substantial and complex. If we were to do the project again we would have decided on the design patterns beforehand, so we wouldn't deviate so much, which would have saved us a lot of time. It would too have helped us grasp and understand the necessary complexity of the project and helped to improve our planning for key quality attributes.

Design Patterns

When we first started to plan the project little thought was given to the design patterns that would eventually be used. The only one we knew for certain would be used was the Model-View-Controller but that was a given. The first design pattern that was implemented was the decorator design pattern to deal with the booking price, because this was the first pattern we implemented we had quite some trouble implementing it to start. We found it hard to understand how it would be used and even what its benefits are. The second design pattern that was implemented was the prototype on recommendation from the lab sessions. This particular pattern deals with the different classes that can be booked. The third design pattern used is the singleton and it manages the database.

If we were to do this project again we would implement the state design pattern as well as it would increase the polymorphic behaviour of our system and increase its ability to support additional behaviour. The design pattern would be incredibly useful for us with the business logic we have were the points that are generated and spent vary depending on the level of the user.

When we first started the project, we were sceptical about the usefulness of using the various design patterns but throughout the implementation the power of the design patterns became increasingly clear. It was a bit of a eureka moment when we realised that changing big chunks of code in one section doesn't break another. This realisation allowed the increased modularisation of tasks which was highly valuable when we were not able to meet up face to face.

When first implementing the Strategy, Adaptor and Interceptor Design Patterns, I found it difficult to find a suitable place to implement the Interceptor or Strategy pattern. I found it easy to find a suitable place to add the Adaptor pattern to create different types of Airlines. After discussion with my project partners I found the best way to implement the Strategy pattern would be to have a multiplier for each tier that would increase the user's points based on which tier they are in. The Interceptor was the hardest to implement due to its complexity, although it was not the most applicable way to implement the pattern, it allows the user or developers to see a clear reason why an error has occurred.

If we were to do this project again, I would find a better way to try to implement the Interceptor design pattern, I would try to implement this design pattern to keep track of logging in and logging out times for users, security authorization or I would implement this design pattern to keep track of payment details which could be card details or paypal.

If I were to implement the Adaptor design pattern again, I would try to return a new type of Airplane such as Light Jet, Mid Size Jet or Heavy Jet instead of creating an Airplane separately and getting the Adaptor to return the baggage of the Light Jet, Mid Size Jet or Heavy Jet.

I believe that the Strategy pattern was applied was the most applicable, other alternatives that the Strategy pattern could have applied to was to save the information about the users in files or compressing files that could be uploaded by a user. However, I do not believe this approach would be applicable for this program.

Implementation Criticisms

Once the architecture diagrams and time class diagrams were created we could get on the project and implementing our chosen design patterns. The implementation started off relatively slowly as we were unsure of ourselves, since we had never worked together it took awhile for us to find our strengths and weaknesses within the group but quickly we got to grasp with the project and started to make progress. We had clear goals defined for each meetup and we were able to achieve those milestones. The biggest setbacks as already mentioned were the creation of the design patterns because we had never worked with them before.

The system has incorporated most of the features we wanted it to. The customer points system is sufficiently complex and we learned alot about system design, design patterns and how to implement such a system of which we had no prior experience.

Had we had time we would have also liked to add SonarQube to this project to inspect the quality of code we have written, which would have no doubt improved the quality of our code and would be another opportunity to learn.

References

A series of informative videos on design patterns by Derek Banas

https://www.youtube.com/watch?v=vNHpsC5ng_E&list=PLF206E906175C7E07

A page describing the implementation of Gang Of Four Design Patterns

<http://www.blackwasp.co.uk/gofpatterns.aspx>