

ZAir

Claudiu Rediu 266129

Dominika Kubicz 266148

Nikita Roskovs 266900

Tudor Ciobanu 267632

Supervisors:

Ib Havn (IHA)

Joseph Chukwudi Okika (JOOK)

Knud Erik Rasmussen (KERA)

Mona Wendel Andersen (MWA)

ICT Engineering

Semester 2

8.06.2018

[49222 characters]

ICT Engineering

2nd Semester

8th of June

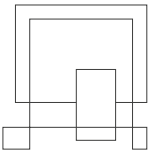


Table of content

Abstract iii

1 Introduction..... 1

2 Requirements 2

 2.1 Functional Requirements..... 4

 2.2 Non-Functional Requirements 11

3 Analysis..... 12

4 Design 13

5 Implementation 26

6 Test 33

 6.1 Test Specifications 33

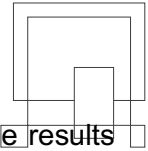
7 Results and Discussion 43

8 Conclusions..... 44

9 Project future 45

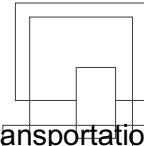
10 Sources of information 46

11 Appendices 1



Abstract

The following report describes the process of creating the ZAir system and the results that have been drawn when the project was finished. The project was completed in five phases, starting with Introduction and continuing with the Requirements and Analysis, followed by Design and finishing with Testing. The introduction phase began with setting the scene for ZAir Airlines and its need for a new system. In the next phase, a set of functional and non-functional requirements for the system were captured and defined. A list of Use Cases was established to describe the system's behaviour. In the Analysis phase, a Domain Model was created to better understand how different concepts are related to each other. The Design phase was about defining the structure of the system. To properly visualize the structure of the system, a Design Class diagram based on the existing Domain Model was created. In the implementation phase, the Design Class diagram was used as a blueprint for the actual code for the system. In the Testing phase, each Use Case was tested via multiple Test Cases and the code has been a subject to multiple JUnit tests. After completing all phases, results were drawn to analyse the success rate of the project and what could be improved in the future.



1 Introduction

According to The World Bank (2016), in the second half of the 20th century air transportation began to grow rapidly. In the period* from 2000 to 2016, the number of passengers carried increased by 120% (1,674 billion to 3,696 billion passengers carried per year). Statistics forecasts an increase in demand for passenger transportation. The current systems are not maintainable as the stream of passengers is growing. In this case, there will be a need for a new system to handle the current growth.

A business with the name of Arkia Israeli Airlines operates scheduled flights, linking several cities from Israel as well as charter flights to some European destinations. Similar to Zair, AIA had problems with managing bookings, as they had a manual management system, which often included issues such as double-booking and the inability to relate outbound and inbound flights of specific passengers. By developing their own tailored management system, called AMSYS, they managed to not only solve their problems, but also introduce custom features, their own security measures, improve their ticket revenue, it being the main source of income, and reduce costs compared to buying a generic system from other companies and as stated by Arkia Israeli Airlines (1988).

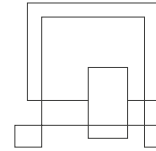
Zair is an upcoming airline business that runs on a system that it uses from the startup. Zair has its head office in Horsens, currently managing direct flights all over Europe and looking forward to expanding to other continents. Some services that it provides are booking flights, seat reservation, providing the cheapest tickets for frequent travelers, account system etc. With the present growth of the market, the current system is lacking in fulfilling the needs of the growing company.

The airline business looks forward to improving their system, so they can expand to other continents and gain a loyal following of customers while keeping its initial features of travelers get their satisfaction and low prices.

All these problems and requests are explored in more detail in the Project Description, which can be accessed in Appendix A.

The aim of the project is to create a system to help ZAir Airlines to store and manage flights more efficiently and allow its customers to book tickets and have quick access to the flights the company provides.

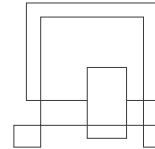
The next sections of the report will outline how the system was developed and describe the phases it has progressed through, starting with requirements capturing, up until testing and drawing results.



2 Requirements

This phase of the system development deals with capturing and defining the functional and non-functional requirements. Based on the project description and following the SMART principle (YourCoach n.d.) and the MoSCoW (Business Analyst Learnings 2013) method, 20 requirements have been set for the project. Thus, it should meet the following conditions: (to prioritize)

1. The administrator should be able to add a flight.
2. The system should be able to store a flight's flight id, origin, destination, date and time of departure and arrival, price, number of tickets.
3. The system should be able to store a customer including his/her customer id, first name, last name, email, telephone number, passport number.
4. The customer should be able to book a flight and choose the seat for it.
5. The system should be able to store tickets including its ticket id, seat number, price.
6. The customer should be able to get a list of cheapest flights and today's flights.
7. The customer should be able to create an account.
8. The customer should be able to use the credentials linked to his account to log in and use the system.
9. The customer should be able to cancel a booking for a flight.
10. The administrator should be able to see all the flights.
11. The administrator should be able to search for flight by flight id.
12. The administrator should be able to remove a flight.
13. The system should be able to store credentials for the customer, which include username and password.
14. The customer should be able to search flights by origin, destination and by a certain time frame.
15. The user should be able to view his current and past bookings.
16. The administrator should be able to see actions performed on the system and the date and time of their occurrence.



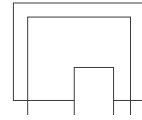
The last 4 requirements are defined and explained in section 2.1.

To better identify the actors and their goals when using the system, an actor-goal list has been created for the project. (Table 1)

Actor	Goals
Administrator	Add flights Remove flights Search flights by their id
Customer	Book tickets for flights Cancel tickets for flights Search for the cheapest flights Search flights by their origin, destination and date of departure/arrival. Search for flights in a certain day. View flight history.

Table 1. Actor-goal list.

The requirements defined in this section can be categorized into two general groups: functional and non-functional. Both types are further examined in the following paragraphs.



2.1 Functional Requirements

Functional requirements can be any specific functionality that define *what* a system is supposed to accomplish (Wikipedia, 2012). As functional requirements describe “behaviors”, they are modelled by/contained into Use Cases. Use cases are text stories, widely used to discover and record requirements (Larman 2004). After a proper analysis of the previously drawn Actor-goal list (Table 1), only two Use Cases have been identified for the ZAir system (Figure 2), but with multiple scenarios of achieving the same goals. Both have been tested using the Boss Test (Larman 2004, ch. 6), EBP Test (Larman 2004, ch. 6) and the Size Test (Larman 2004, ch. 6).

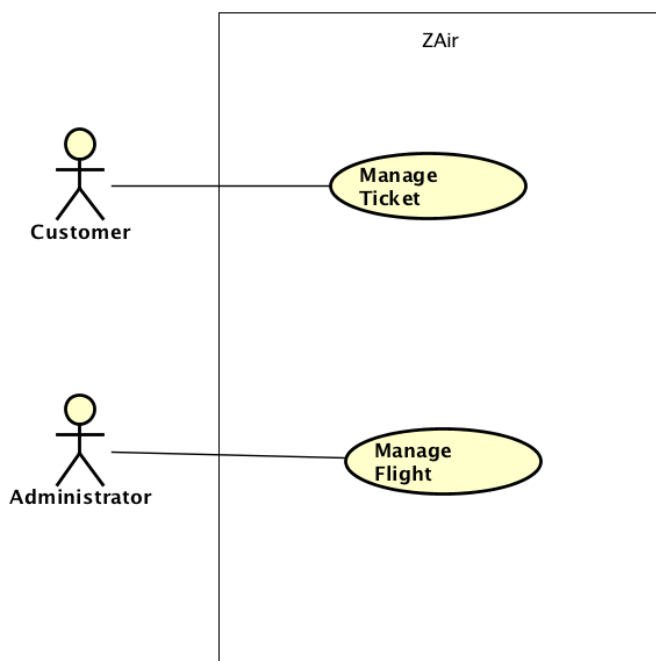
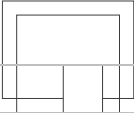


Figure 2. Use Case Diagram

For instance, *Manage Flights* Use Case refers to the Administrator’s ability to add and remove details about the flights that ZAir Airlines provides. The description in Figure 3 offers more details about three out of all the possible scenarios when the Administrator is trying to add details about a new flight. The complete Use Case description can be viewed in Appendix B.



Use Case Name:	Manage Flights
Scope:	ZAir
Level:	User goal
Primary Actor:	Administrator
Pre-conditions:	No preconditions.
Post-conditions:	Details about a flight are either added or removed from the application and the database.
Main Success Scenario:	<p>ADD FLIGHT:</p> <ol style="list-style-type: none"> 1. Administrator chooses origin city, destination city, departure date, arrival date, departure time, arrival time and types in the price. 2. Administrator validates his decision via button press. 3. System displays the message "Successfully added".
Extensions:	<p>ADD FLIGHT SCENARIO:</p> <ol style="list-style-type: none"> 1a. At any time, Administrator cancels the operation; 1b. Destination city is the same as the origin city: <ol style="list-style-type: none"> 1. System notifies the Administrator about the error - "Origin and destination must have different values." and Use Case ends.

Figure 3. *Manage Flights* Use Case description (partial)

The "sunny scenario" occurs when the Administrator, chooses and types in valid values for a flight's details and validates his decision to add the flight details via button press. In response, the system correctly adds the information about the flight and displays a validation

message. The System Sequence Diagram (SSD) in Figure 4 is a visual representation of the interaction between the Administrator and the system in this scenario.

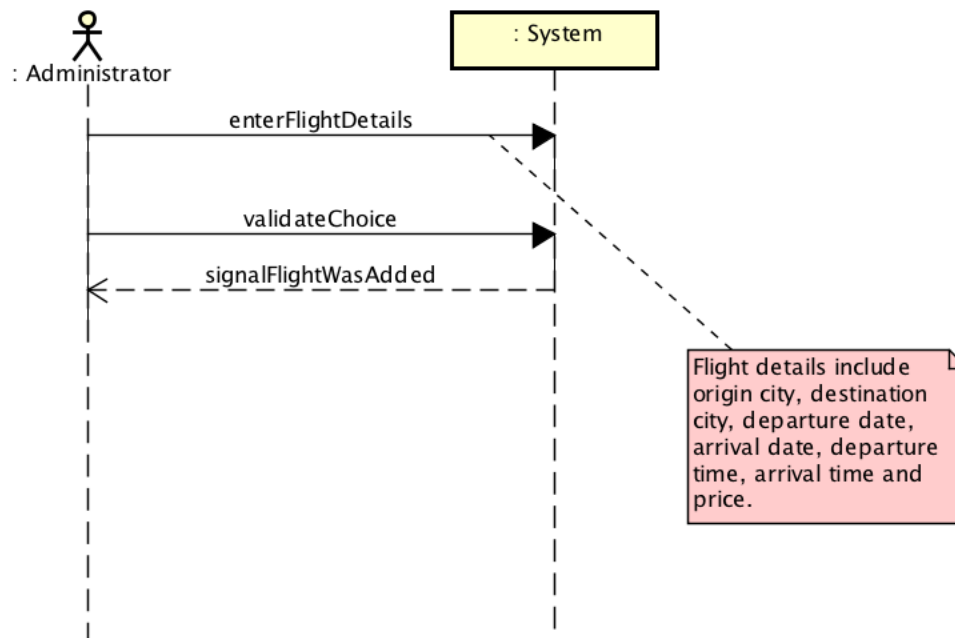
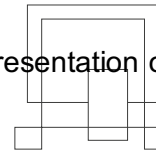


Figure 4. SSD for the main success scenario

In addition to this main success scenario, there is plenty of exception scenarios. For instance, if the Administrator is trying to add a flight with identical origin and destination cities, the system displays/notifies about the error and thus, the Use Case ends. This can be also observed in the SSD for this scenario in Figure 5. All the SSDs can be found in Appendix B.

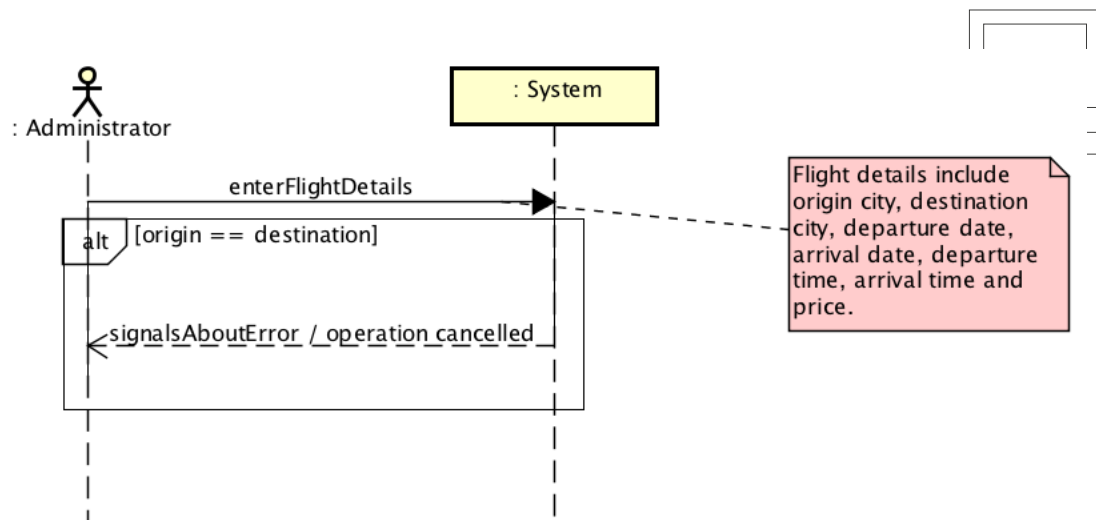
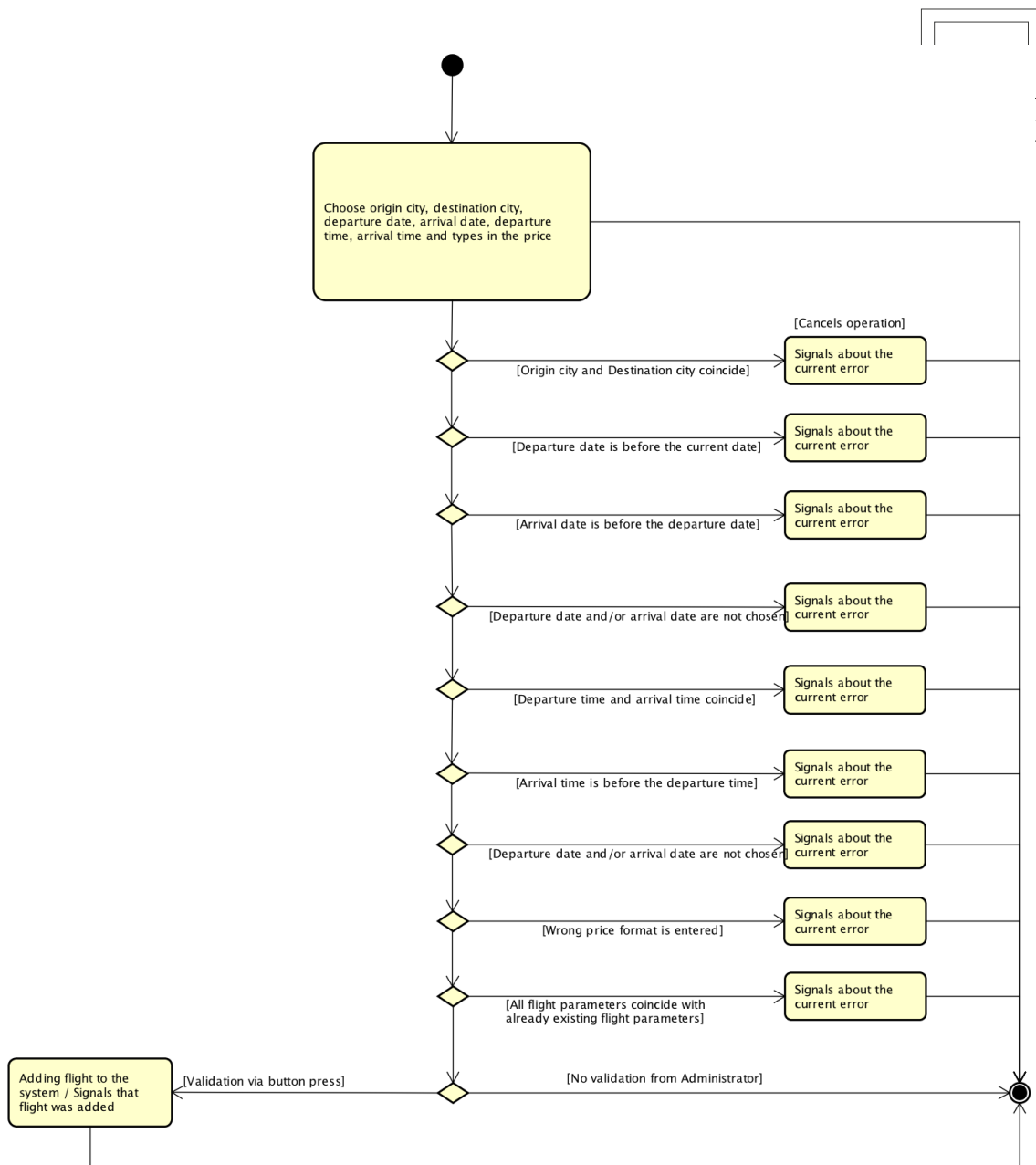
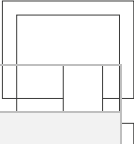


Figure 5. SSD for an alternate flow

An Activity Diagram can model the control flow from one activity to another. The activity diagram in Figure 6 shows the overall flow of control in the *Manage Flights* Use Case.

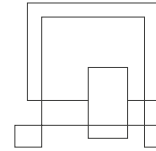
Figure 6. Activity Diagram for *Manage Flights* Use Case

All the activity diagrams can be viewed in Appendix B.



Use Case Name:	Manage Ticket
Scope:	ZAir
Level:	User goal
Primary Actor:	Customer
Pre-conditions:	The user has an account in the system and there are flights stored in the system.
Post-conditions:	The user has booked a flight. The user has removed a booking.
Main Success Scenario:	<p>BOOKING:</p> <ol style="list-style-type: none"> 1. User inserts user name and password and logs in. 2. User chooses to see the cheapest flights via button press. 3. System displays a list of ten cheapest flights. 4. User selects a flight from the list. 5. User confirms his choice via button press. 6. System displays a list of available seats. 7. User selects a seat. 8. The user confirms his choice via button press. 9. The system displays a message "Successfully booked. Thank you!"
Extensions:	<p>BOOKING:</p> <ol style="list-style-type: none"> 6a. The system displays a message "no flight selected". 9a. The system displays the message "You already booked a ticket for this flight!" when you already have that flight booked. 9b. The system displays the message "Flight was removed by the administrator!" when the flight was removed while booking.

Figure 7. *Manage Tickets* Use Case description (partial)



On the other side, the *Manage Tickets* Use Case refers to Customer's ability to book and/or cancel tickets for the flights that ZAir Airline provides. The Use Case description in Figure 7 shows just four possible scenarios that may occur when booking a ticket. The full description can be viewed in Appendix B.

The "sunny scenario" starts with the Customer logging in with existing credentials. The Customer then chooses to see the cheapest flights via button press. In response, the system displays all the cheapest flights available. The Customer then selects the desired flight and confirms his choice. In response, the system displays all available seats. The Customer then chooses a seat and confirms his choice. In the end, the system displays a validation message.

This interaction can be further examined in the System Sequence Diagram in Figure 8.

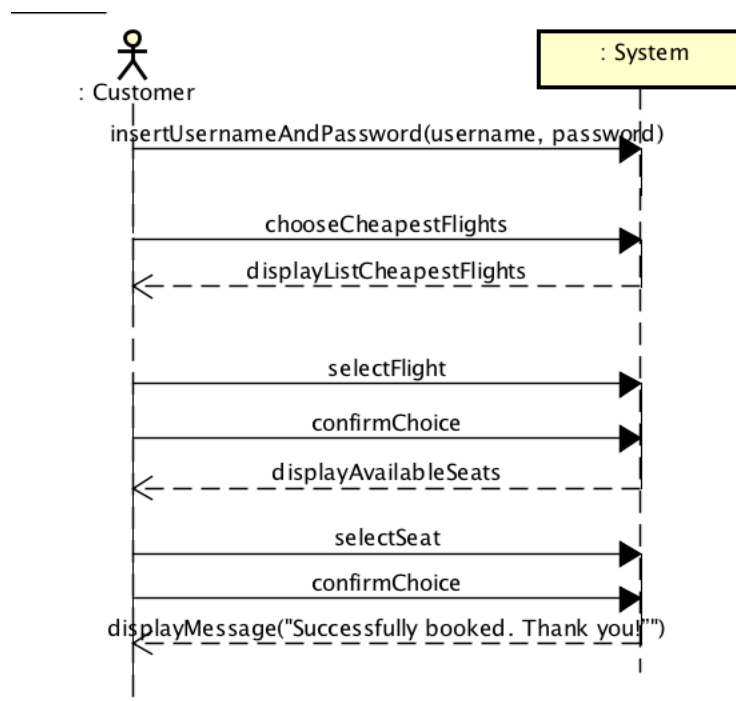
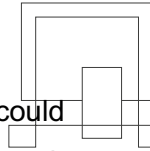


Figure 8. SSD for the main success scenario



In addition to the “sunny scenario”, there is multiple alternative scenarios that could happen in the Use Case. For instance, if a flight is not selected by the Customer, the system displays an error message. Similarly, when the Customer tries to book multiple tickets for one flight, the system displays an error message and the Use Case ends.

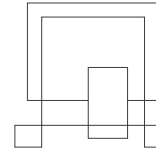
2.2 Non-Functional Requirements

Non-functional requirements or NFR specify the criteria that can be used to judge the operation of a system, rather than specific behaviours.

Using the NFR checklist (Banger, 2014), the non-functional requirements can be categorized and organized in the table below (Figure 9):

NFA	NFR
Implementation	The system must be implemented in Java. The system should not contain hard coded values.
Persistence	The system should use a database.
Compatibility	The system must be compatible with Microsoft Windows 7,8,10, Mac OS.

Figure 9. NFR



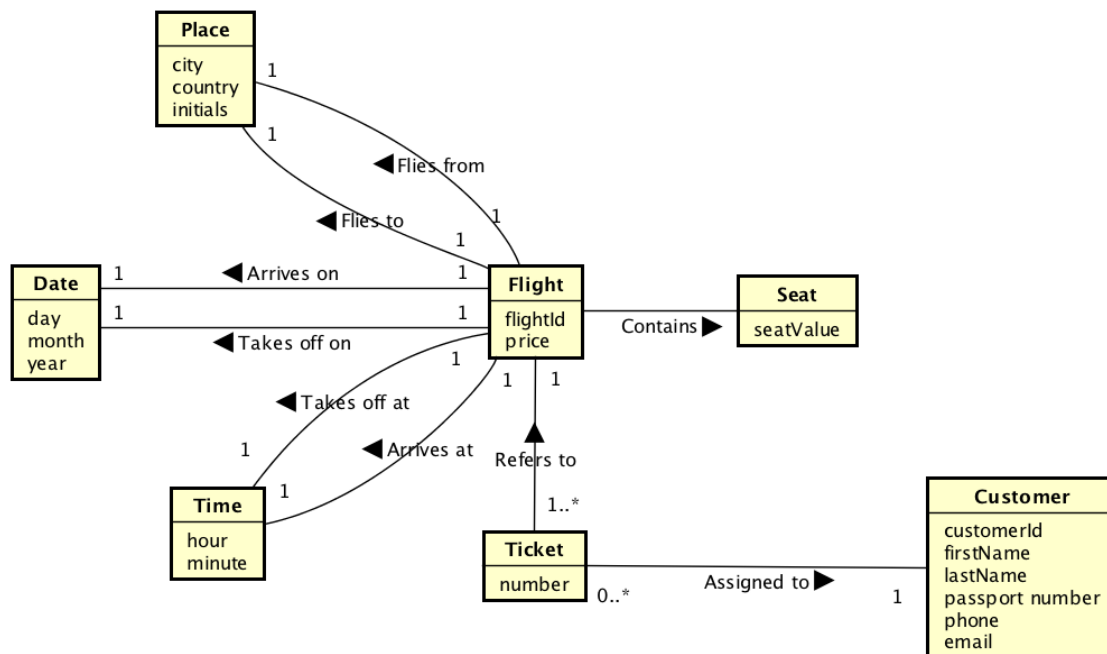
3 Analysis

Before implementing any system, there is need for analysis, to ensure that the area of expertise of the problem and the demands from the stakeholders are correctly understood.

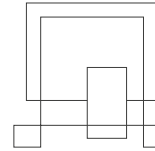
By developing the system, it was expected that the Administrator can manage flights (add or remove details about flights from the system) and that the Customer is able to book and remove tickets, choose specific seats and have access to his/her flight history.

The other point was to make sure that features such as searching for a flight in the specific period, seeing all today's flights or the top cheapest flights offered by a company are available for the Customer to achieve his/her goals. For instance, an account system would be a solution to help managing purchased tickets from the customers side (customer will be able to see his/her full flight history and cancel purchased tickets).

In this section is illustrated the main concepts of the system and ideas of a domain. As all these demands are already contained in Use Cases, some of the terms used in their descriptions have served as an inspiration to better identify candidate conceptual classes. By carefully analysing the background description and all of the Use Case descriptions, the main conceptual classes that shape the structure of the system have been defined. A Domain Model has been developed to show the conceptual structure of the system and relation between its components. (Diagram below)



The Domain Model can also be viewed in Appendix B.



4 Design

With all the analysis work completed in the previous phases, the Design section deals with the actual structure of the software. As stated by (Larman 2004, ch. 34), the design phase has a profound impact on the maintainability and clarity of an object software system, plus on the degree and quality of the reusable components.

ZAir system is based on the RMI architecture, a mechanism that involves remote method invocations. This mechanism allows objects from one Java Virtual Machine (JVM) to be accessed by other objects from another Java Virtual Machine. A visual representation of the mechanism is shown in Figure 10.

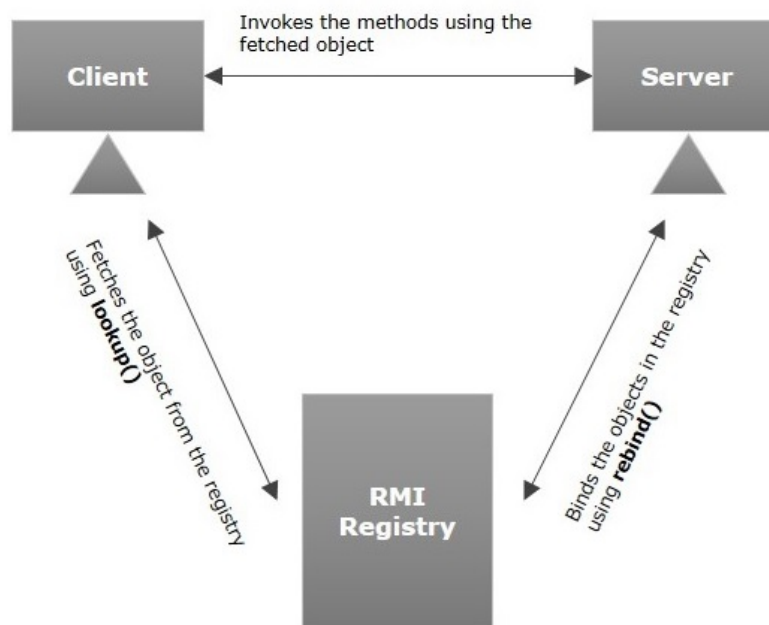
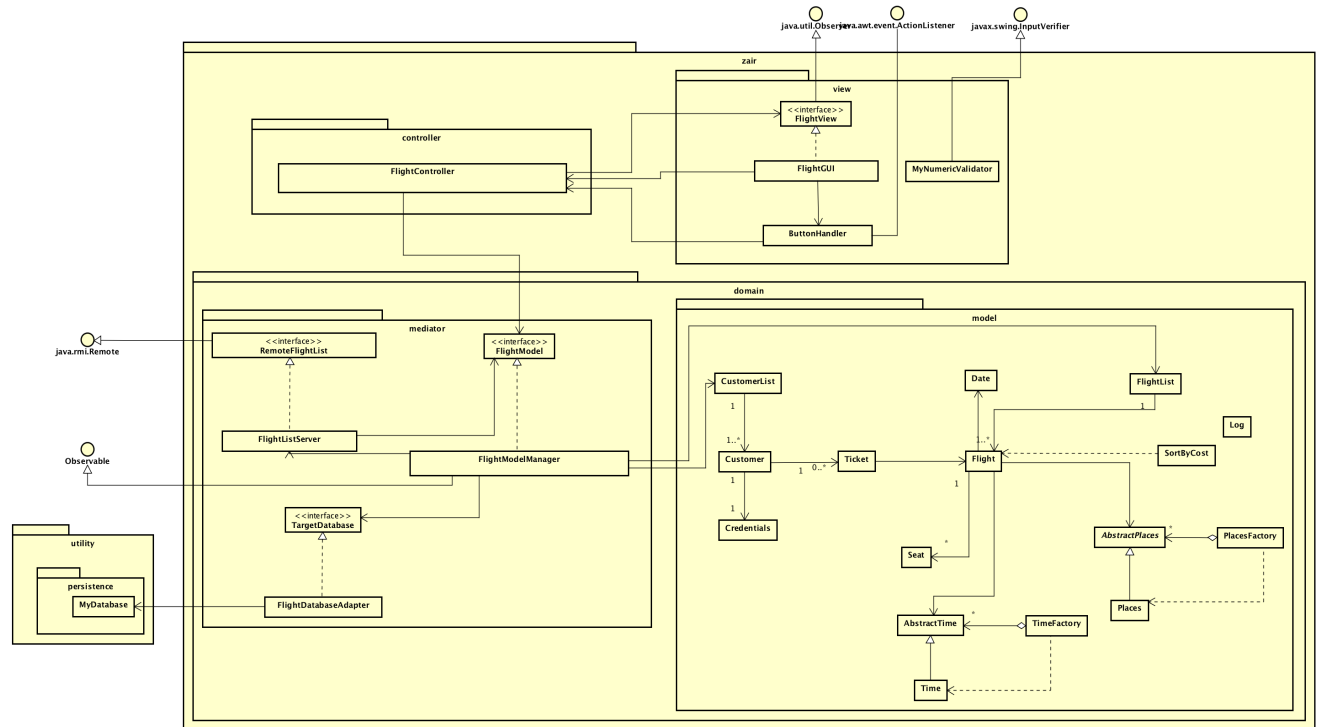


Figure 10. RMI mechanism

The RMI registry is a namespace where all the objects from the Server are placed. Whenever the Client needs to invoke a remote object, it first needs to look up the registry's bind name and then fetch it.

The system was also designed with the Logical Architecture (Larman 2004, ch. 34) in mind, as different responsibilities have been separated using layers, such as the Domain layer, UI layer and the Persistence layer. The structural design pattern used for the system is Model-View-Controller. It was used to separate the UI layer from the Domain layer which contains

Model, View
erver and th
of the Clas



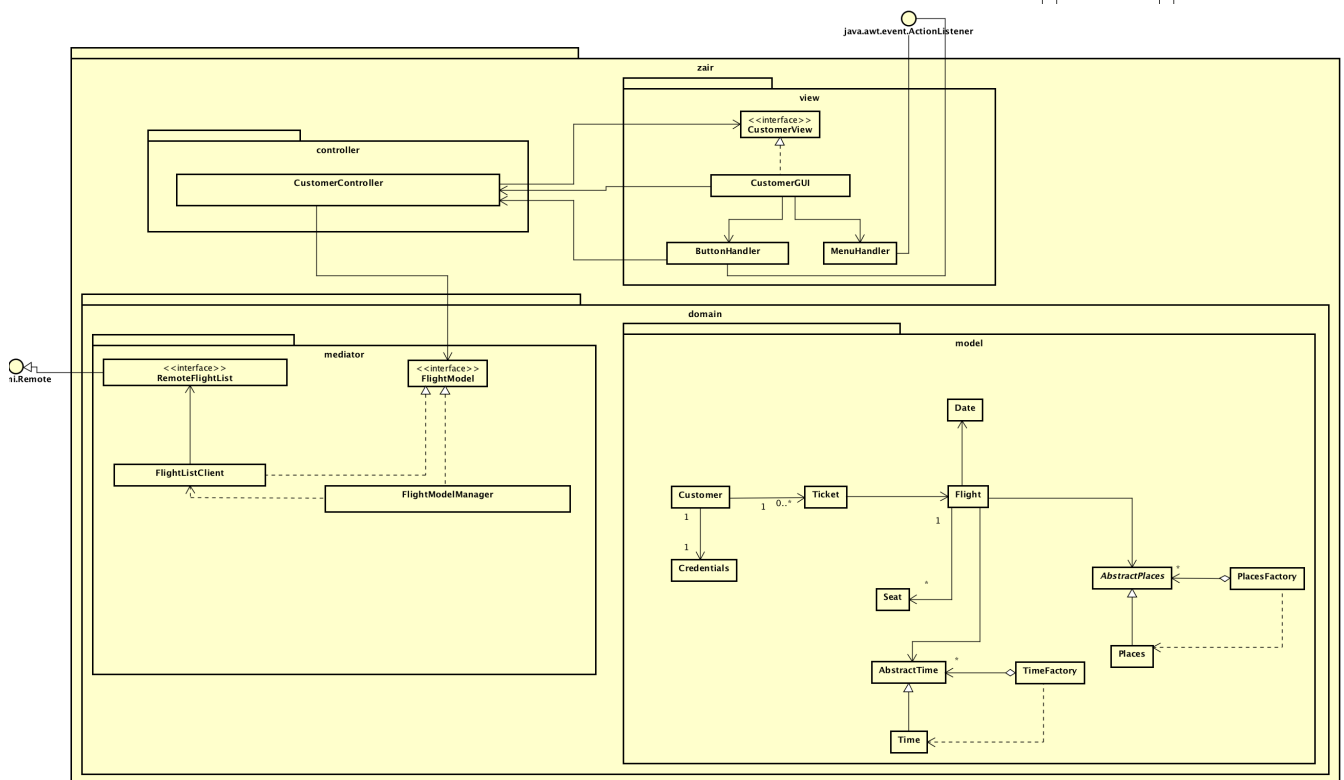


Figure 12. Class diagram for client

On the server side, the Model component (domain package/layer) contains all the application logic to launch the server and the logic necessary to complete the two Use Cases defined in section 2.

As ZAir Airlines provides flights only to the same locations and at the same scheduled hours, the obvious choice was to use the Flyweight Pattern, to save the system from creating the same type of objects all the time. “Simply put, the flyweight pattern is based on a factory which recycles created objects by storing them after creation. Each time an object is requested, the factory looks up the object in order to check if it’s already been created. If it has, the existing object is returned – otherwise, a new one is created, stored and then returned.” (Baeldung, 2018). This mechanism can be better understood by analysing Figure 13.

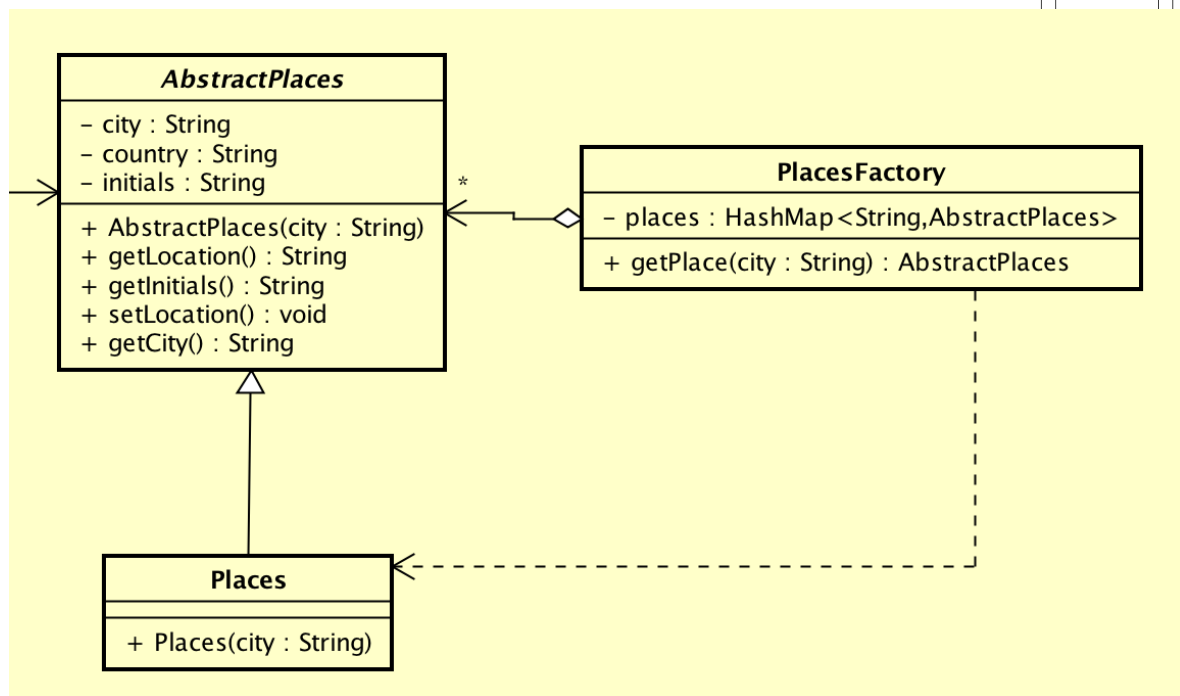


Figure 13. Flyweight pattern

For instance, if a Places object that has city attribute set to Copenhagen has already been created, the factory will return it, otherwise the factory will create a new one, store and return it. The same principle applies for the objects of the type Time.

As stated in the user requirements, the Administrator must be notified when an operation is performed on the Model, including the date and time of the occurrence.

As the Model communicates with the View via the Controller, there is no direct connection between them, which means MVC is not the method to achieve that. The solution was to use an Observer Pattern between the Model and View and a Singleton (Log class) that returns the timestamp at a certain moment. The Observer Pattern is consisted from an Observer and a Subject. The Observer is notified when a change is made on the Subject. In the system, the FlightModelManager class is the Subject, as it has access to all the operations that can be performed on the Model and the FlightView is the View component interface. Therefore, the FlightModelManager class extends the Observable class from Java API, while FlightView extends the Observer class from Java API. An UML representation of the pattern can be viewed in Figure 14 and 15.

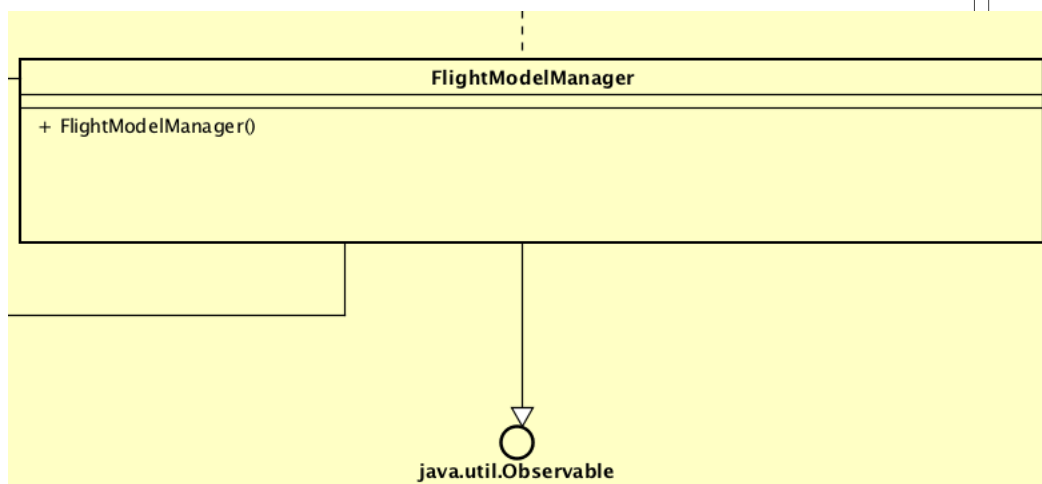


Figure 14. Observer Pattern (1)

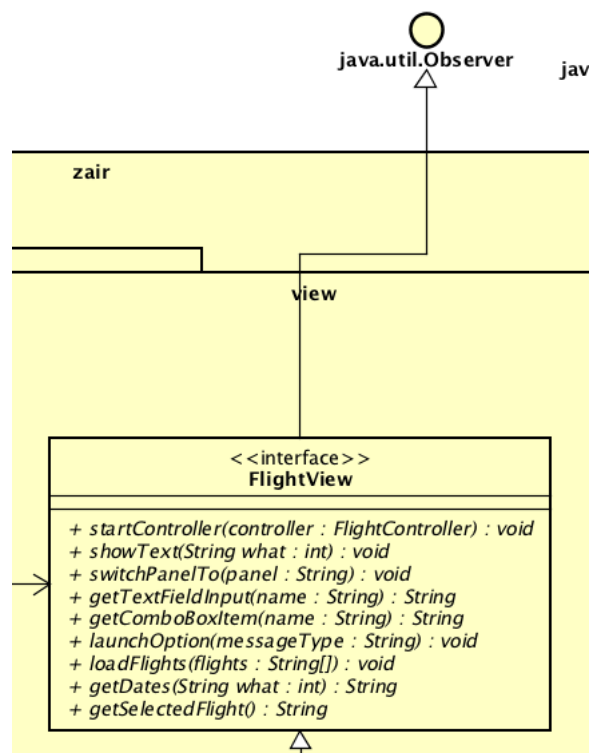
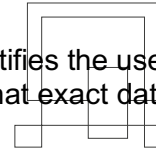


Figure 15. Observer Pattern (2)

As soon as a change is made in the Model, the FlightModelManager class notifies the user interface, which immediately displays what change has been made and at what exact date and time.



Part of the solution was to also use the Singleton Pattern. It makes sure that a class can have only one instance at a time. The Log class in Figure 16 is a Singleton. Because it has a private constructor, it restricts the creation of more than one instance at a time. As the methods from this class are used often in the system, it made more sense to have a global point to access them, rather than create multiple objects which share the same state.

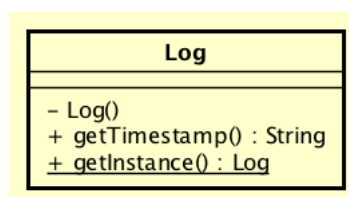


Figure 16. Log class

The SortByCost class is Comparator class, that compares two Flight objects by their price and returns the truth value of the comparison. This can be observed in the Class Diagram in Figure 17.

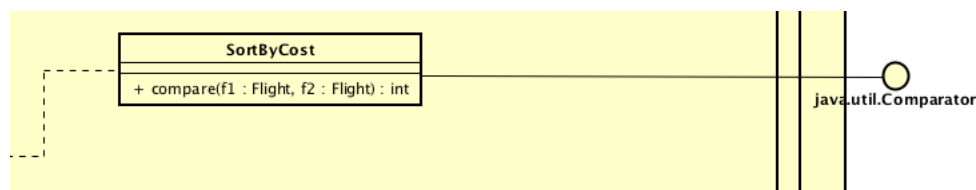


Figure 17. SortByCost class

While the zair.domain.model package contains all the business logic, the FlightModel and FlightModelManager classes from the zair.domain.mediator package serve as a façade to the actual model that keeps its state. Both classes contain all the main operations that could be performed on the model. Therefore, this ensures loose coupling between the model and the other components, meaning that other components know about the operations the model can perform but have no knowledge about the model classes themselves.

Additionally, the FlightModelManager class has an instance of both the CustomerList and FlightList classes, which are collections of Customer objects and Flight objects respectively, that perform operations on a multitude of objects at a time. All the methods in FlightModelManager class are delegating to the work to these two instances.

The RemoteFlightList class is a remote interface that declares all the operations that can be invoked from a remote JVM. The FlightListServer class makes sure the necessary objects are uploaded to the RMI registry and binds it to a name. As shown in Figure 5, this class implements the remote interface and has an instance of the Model

interface, meaning that the remote methods call operations on the server-side Model.

On the server side, the View component consists of an interface that ensures loose coupling and contains all the operations that could be performed on the UI, a Swing GUI class implementing it and an Action Listener class, which makes sure that the actions performed on the buttons are registered and there is a proper response. The UML representation of the View package and its classes is shown in Figure 18. The ButtonHandler class has an instance of the Controller because whenever a button is pressed it just delegates to the Controller to perform the required operations. The MyNumeric class is a Singleton and implements InputVerifier interface and checks if a JTextField object contains a number. If it doesn't contain a number, it returns an error message.

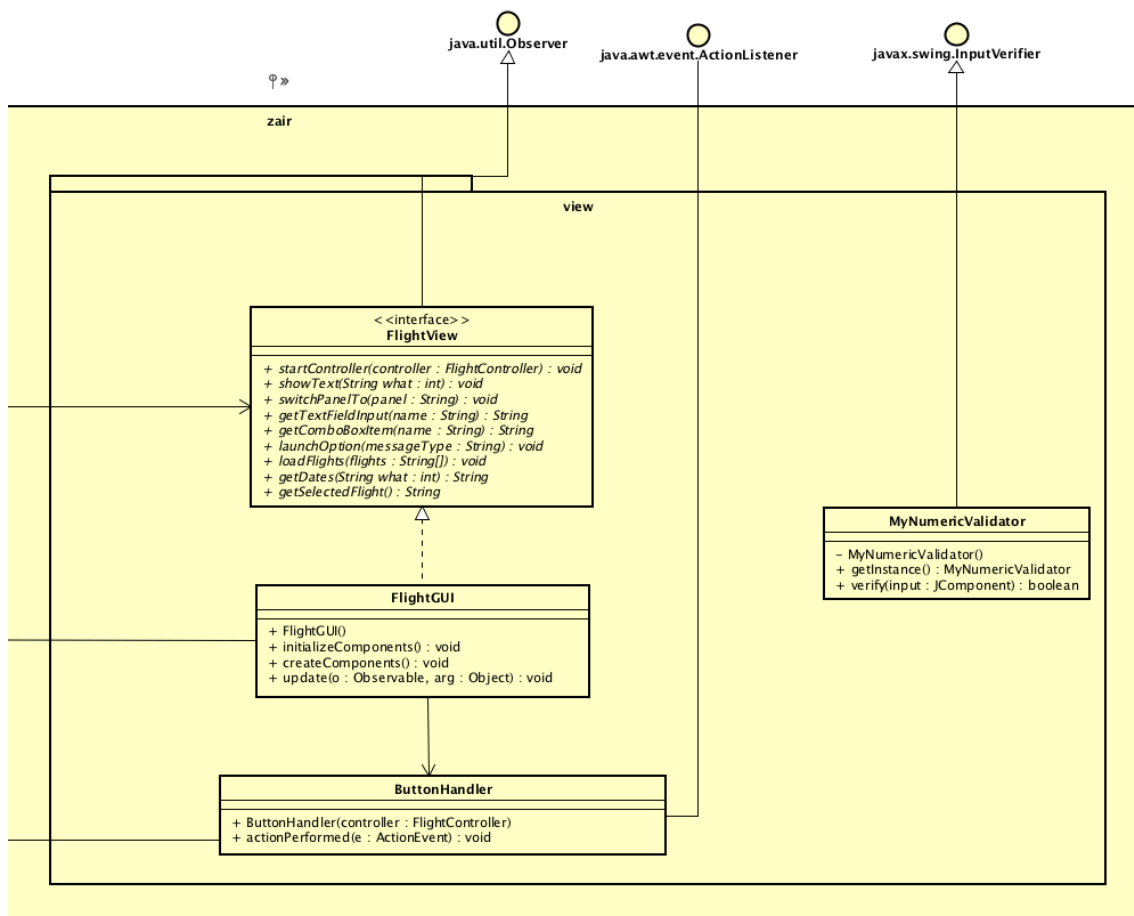


Figure 18. View Component

On the server side, the Controller component consists of the FlightController class, which controls the interactions between the View and the Model. In this case it has instance of the Model and View interfaces, thus having access to the operations available on both the View and the Model. As shown in Figure 19, it has the execute() method, which takes the input from View, creates the objects required, then calls the necessary methods from the Model and eventually generates a response for the View.

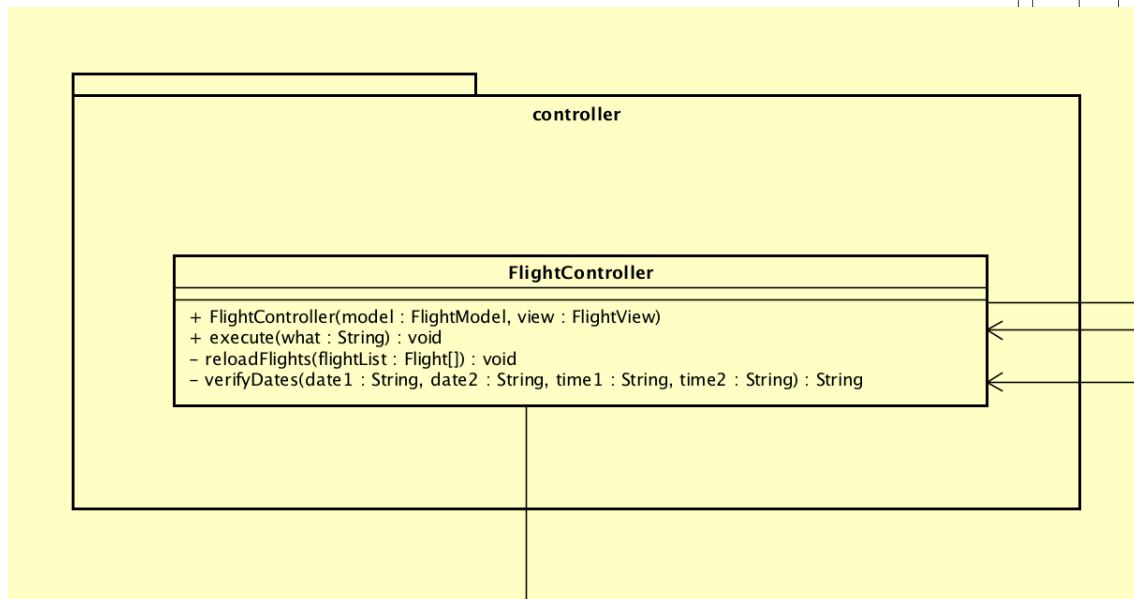
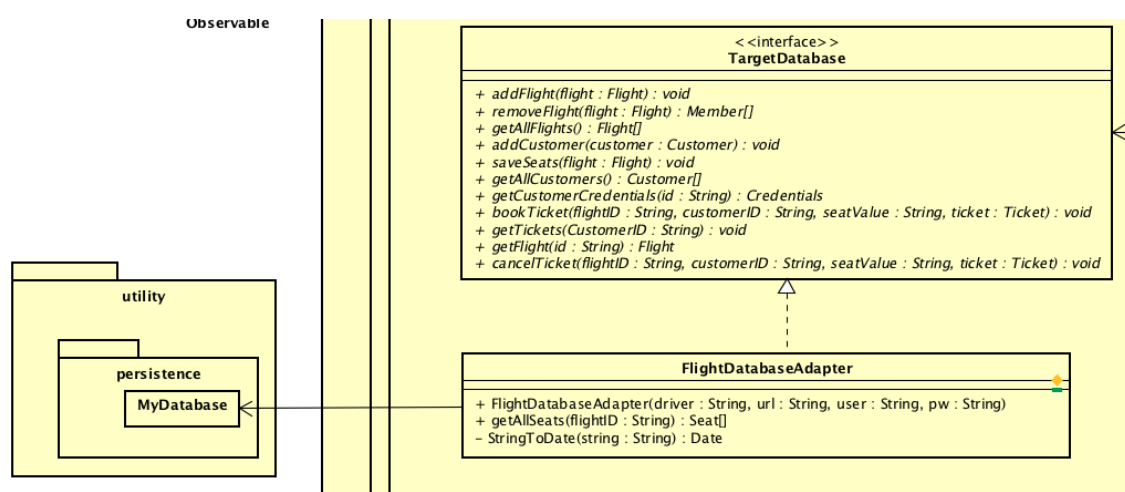


Figure 19. Controller Component

The persistence for the system is taken care of by a database. In the system, the database access is taken care of using an Adapter design pattern, as shown in Figure 20. The Adapter design pattern allows two incompatible classes/interfaces to work together. In Figure 20, the MyDatabase class is the adaptee, a general class to access any database and the FlightDatabaseAdapter class is the adapter, a more specific class that connects to a specific database. The TargetDatabase interface specifies which methods related to the Model could be performed. As the Model has only an instance of this interface, the Model has no direct relation to the database, ensuring that the persistence layer is separated from the domain layer. The FlightList and CustomerList instances in the FlightModelManager constructor are instantiated by calling the getAllFlights() and getAllCustomers() respectively from the FlightDatabaseAdapter. The FlightDatabaseAdapter class then delegates the methods to the MyDatabase class.



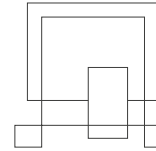


Figure 20. Adapter pattern

The database which the system connects to has the following business rules:

- The database will contain flights, tickets, customers, seats and customer credentials.
- Flights will be consisted of flightID, origin, destination, dateOfDeparture, dateOfArrival, timeOfDeparture, timeOfArrival, nrOfTickets, price.
- Each flight will have seats (seatPo, seatValue, flightID).
- Each flight will be linked with several tickets.
- Customers will be consisted of customerID, fName, lName, email, passportNo, phone.
- Tickets will be consisted of ticketID, seatPo, customerID, flightID.
- Each customer will have attached to it customer credentials (customerID, password).

The EER diagram for the database is shown in Figure 21.

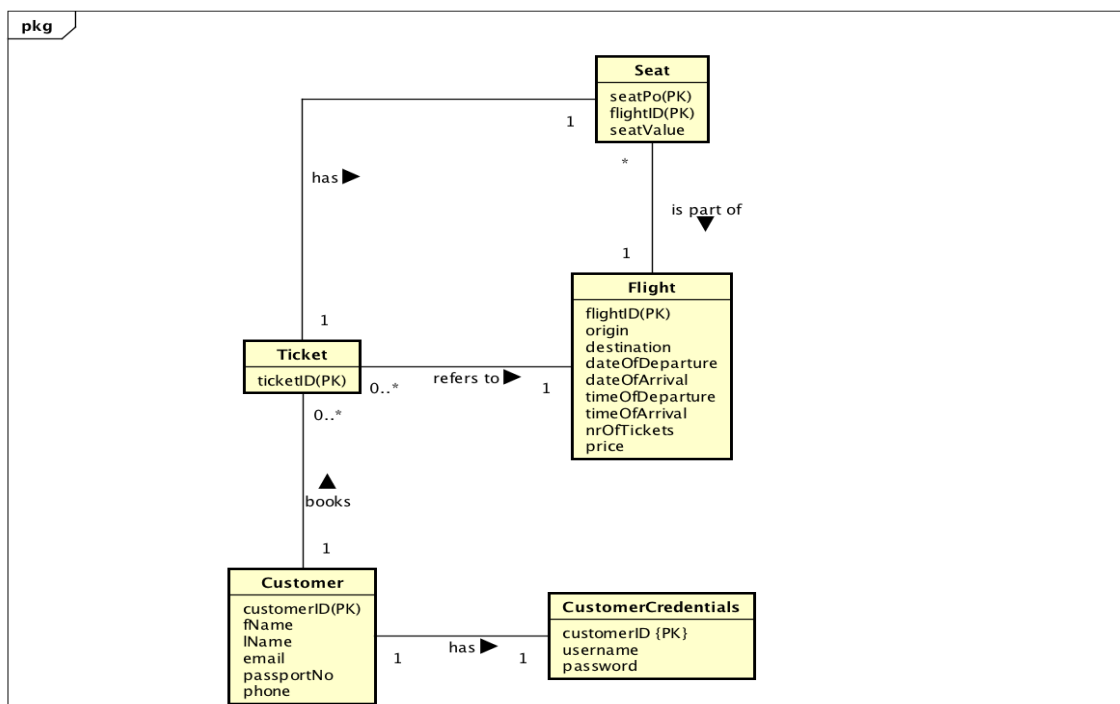


Figure 21. EER diagram

After the process of normalization, the logical model for the database is represented in the table below:

Flight(flightID, origin, destination, dateOfDeparture, dateOfArrival, timeOfDeparture, timeOfArrival, nrOfTickets, nrOfTicketsLeft, price) Primary Key flightID
Ticket(ticketID, flightID, customerID, seatPo) Primary Key ticketID Foreign Key flightID references Flight(flightID) Foreign Key customerID references Customer(customerID) Foreign Key seatPo references Seat(seatPo)
Customer(customerID, fName, lName, email, passportNo, phone) Primary Key customerID
CustomerCredentials(customerID, password) Primary Key customerID Foreign Key customerID references Customer(customerID)
Seat(seatPo, flightID, seatValue) Primary Key seatPo, flightID Foreign Key flightID references Flight(flightID)

The SQL code for the database is present in Appendix E.

As the client side of the system is built on the same design pattern – MVC, the structure is very similar. However, there is some changes, mainly in the zair.domain.mediator package and the View component.

As shown in Figure 22, the FlightListClient class has an instance of the same remote interface used on the server side. In this case, the local representative for the objects residing on the server is the FlightListClient class. A proxy pattern was used between the FlightListClient class and the FlightModelManager, so that the FlightListClient is just a placeholder that just deals with all the RMI network logic. This can be observed in Figure 22.

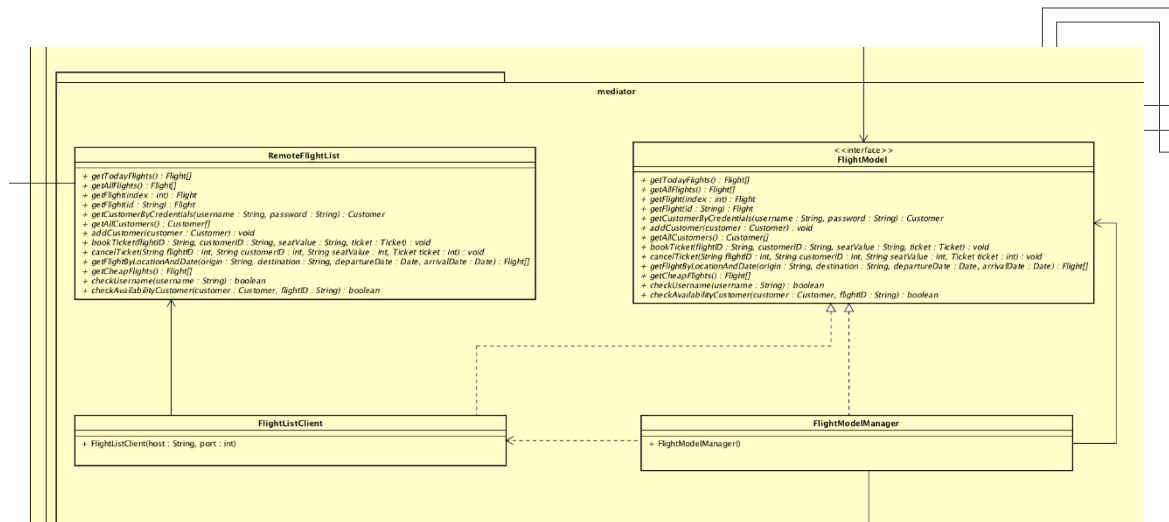


Figure 22. Proxy pattern (Client side)

On the client side, the View component also has the MenuHandler class which is also an ActionListener, but that registers the actions performed on the menu items of the user interface. This can be observed in Figure 23.

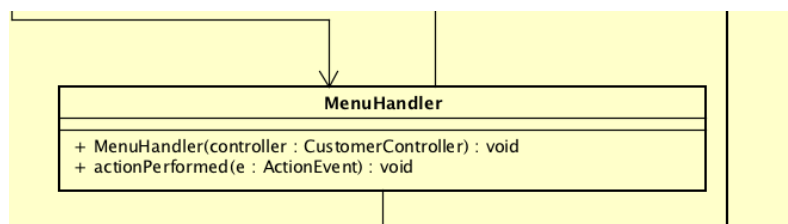


Figure 23. MenuHandler class

The dynamic between the objects in the system can be observed in a Sequence diagram. It shows the flow and the way different objects interact with each other.

The Sequence diagram in Figure 24 is based on the *Manage Tickets* Use Case, specifically the booking scenario, on the client side. The Customer presses on the “Cheap flights” button on the user interface. This calls the actionPerformed method in the ButtonHandler class. This method calls the execute method in the Controller. The Controller gets the list of all the cheapest flights from the Model by calling the getCheapFlights() method. The Model gets this list from the server Model via the remote interface. If the list of cheap flights is empty, the user interface displays an error message. If the list contains any flights, the Controller sends the list to the View by calling the loadFlights() method with the list as an argument and the View loads everything in a table to be seen by the Customer. With all this done, the Customer presses on the “Book” button on the user interface. This again calls the actionPerformed method in the ButtonHandler class, which respectively calls the execute

method in the Controller. The controller checks if any flight from the table with the flights on the user interface was selected by calling the `isRowSelected()` method on the View. If a flight was selected, the Controller gets the its flightId by calling the `getSelectedFlight()` method from the View.

Using the flightId, the Controller calls the `getFlight` method from the model and gets the specific Flight object. The object is used to call the `loadSpecificFlight` method from the view. By calling this method, all the details about the flight are loaded on the user interface. If, initially, no flight was selected, by calling the `launchOption` on the View the user interface displays an error and no other actions are done. Once all the details are displayed, the Customer presses on the “Confirm” button. This calls the `actionPerformed` method in the `ButtonHandler` class. The `actionPerformed` method calls the `execute` method in the Controller. The controller checks if any seat from the table with the seats on the user interface was selected by calling the `isRowSelected()` method on the View. If a seat was selected, the Controller gets the flightId of the flight by calling the `getSelectedFlight()` method from the View. Using the flightId, the Controller calls the `getFlight` method from the model and gets the specific Flight object. By calling the `valueOf()` method on the View, the Controller gets the number of the chosen seat. A Ticket object is then created the seat number and the Flight object from the Model. The Controller then calls the `getLoggedInCustomer()` method which returns the Customer object from the Model that has the specific credentials that have been used to log in the system. The customerId of the Customer object, the flightId of the Flight object, the Ticket object and the seat number are then used as arguments to call the `book` method on the Model. The user interface displays a validation message and the Customer object is used for the `loadCustomer` method so that the user interface updates the information about the customer. If no seats would be selected, UI would display a failure message.

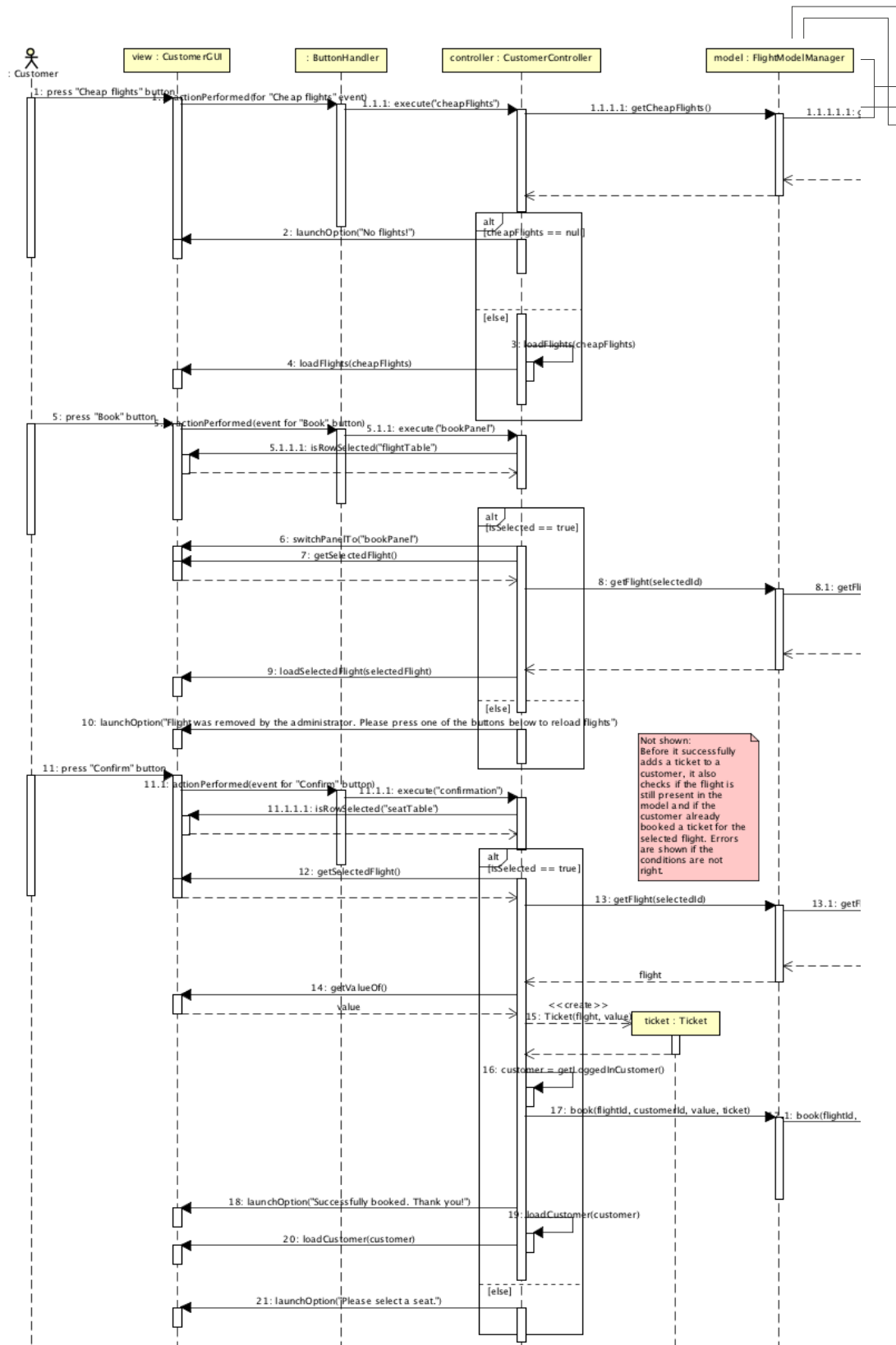
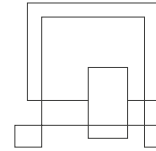


Figure 24. Sequence Diagram for booking a ticket (simplified version)



This Sequence Diagram represents only the client side of the operations. All the methods called on the client Model respectively call other methods on the server Model. All the completed Sequence Diagrams can be viewed in Appendix D.

5 Implementation

The implementation phase uses the blueprints from the Design phase and integrates them into an executable system.

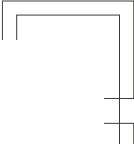
While the Sequence diagram in Figure 24 shows the dynamic interaction between different objects in the booking scenario for the *Manage Tickets* Use Case, in the following lines, some of the code to achieve the removing ticket scenario is going to be examined.

When the Customer presses on the “Cancel ticket” button in the user interface to remove a ticket for a flight, the ButtonHandler class register the action performed on the button in the actionPerformed() method and delegates the work to the controller by calling the execute() method on it with the “cancelTicket” string token as an argument. This can be observed in Figure 25.

```
else if (((JButton) e.getSource()).getText().startsWith("Cancel"))
{
    controller.execute("cancelTicket");
}
```

Figure 25. actionPerformed method in ButtonHandler

In the controller, the execute method contains a switch statement. By calling the execute method with the “cancelTicket” string token as an argument, the controller executes the code that belongs to the “cancelTicket” case. This can be observed in Figure 26.



```

case "cancelTicket":
    if (view.isRowSelected("ticketTable"))
    {
        if (view.launchYesNoOption("Are you sure you want to cancel the ticket?").equals("yes"))
        {
            String[] parts = view.getTicket().split("<");
            String flightID = parts[0];
            String seatValue = parts[1];
            String customerID = getLoggedInCustomer().getId();
            for (int i = 0; i < getLoggedInCustomer().getTickets().length; i++)
            {
                if (getLoggedInCustomer().getTickets()[i].getFlight().getId().equals(flightID))
                {
                    model.cancelTicket(flightID, customerID, seatValue, getLoggedInCustomer().getTickets()[i]);
                }
            }
            loadCustomer(getLoggedInCustomer());
        }
        else
        {
            view.launchOption("Please select a ticket to remove!");
        }
    }
    break;

```

Figure 26. Confirmation case in the execute method

First, the controller checks if a ticket has been selected from the ticket table from the user interface by calling the `isRowSelected` method in the view with the name of the respective table – “ticketTable”.

```

public boolean isRowSelected(String name)
{
    boolean valid = true;
    for (int i = 0; i < tables.length; i++)
    {
        if (tables[i].getName().equals(name))
        {
            valid = !tables[i].getSelectionModel().isSelectionEmpty();
        }
    }

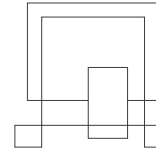
    return valid;
}

```

Figure 27. IsRowSelected method

As shown in Figure 27, the `isRowSelected` method checks if any table in the View has the specified name and then returns a boolean value stating if any row in the table has been selected or not.

The `launchYesNoOption` method is then called on the View, which just launches a *Yes or No* JOptionPane with a certain message and returns the chosen value by the Customer, as shown in Figure 28.



```
@Override
public String launchYesNoOption(String messageType)
{
    String output = "";
    int dialogResult = JOptionPane.showConfirmDialog(null, messageType);
    if(dialogResult == JOptionPane.YES_OPTION){
        output = "yes";
    }
    return output;
}
```

Figure 28. launchYesNoOption method

If the response is affirmative, the `getTicket()` method is then called on the View. This method returns the first and the sixth column of the ticket table from the UI. These columns coincide with the `flightId` and `seat number` values for a ticket. As shown in Figure 29, the method gets the two column values separated by a "<" sign and returns it as a string.

```
public String getTicket()
{
    String output = (String) (tables[2].getValueAt(tables[2].getSelectedRow(), 0) + "<"
    + tables[2].getValueAt(tables[2].getSelectedRow(), 6));
    return output;
}
```

Figure 29. `getTicket()` method in the View

In the Controller, this string is split into two parts, one containing the `flightId` and one containing the `seat value`. The `customerId` is then obtained from the `getId()` method called on `getLoggedInCustomer()` method. The `getLoggedInCustomer()` method returns the Customer object from the Model that matches the credentials used for logging in.

A loop is then used to find the ticket with the `flightId` and `seat number` in the Model and then the `cancelTicket` method is called with the `flightId`, `customerId`, `seat value` and matching Ticket object as arguments. The View is then updated so that the UI doesn't show the ticket anymore via the `loadCustomer` method. This can be observed in Figure 26.

As the Model on the client side is just a local representative of the server Model, when the `cancelTicket` method is called in the `FlightModelManager` class, it delegates the work to the Proxy class, which is the `FlightListClient`. `FlightListClient` calls the `cancelTicket` remotely, because it has an instance of the remote interface. Figure 30, 31, 32 and 33 represent exactly how this mechanism works.

```

public class FlightModelManager implements FlightModel
{
    private FlightModel list;
    private static final String HOST = Init.getInstance().getIp();
    private static final int PORT = Init.getInstance().getPort();

    public FlightModelManager() throws IOException {
        try
        {
            list = new FlightListClient(HOST, PORT);
        }
        catch (NotBoundException e)
        {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}

```

Figure 30. FlightModelManager class on the client side

```

@Override
public void cancelTicket(String flightID, String customerID,
    String seatValue, Ticket ticket)
{
    list.cancelTicket(flightID, customerID, seatValue, ticket);
}

```

Figure 31. cancelTicket method in the FlightModelManager class

```

public class FlightListClient implements FlightModel {

    private RemoteFlightList list;

    public FlightListClient(String host, int port) throws IOException, NotBoundException {
        list = (RemoteFlightList) Naming.lookup("rmi://" + host + ":" + port + "/Flight");
    }
}

```

Figure 32. FlightListClient class (Proxy class)


```

@Override
public void cancelTicket(String flightID, String customerID,
    String seatValue, Ticket ticket)
{
    try
    {
        list.cancelTicket(flightID, customerID, seatValue, ticket);
    }
    catch (RemoteException e)
    {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}

```

Figure 33. cancelTicket method in FlightListClient class

Once the cancelTicket method is called in the FlightListClient class, it remotely calls the cancelTicket method in the FlightModelManager class on the server. This class has an instance of the FlightList class, CustomerList class and TargetDatabase class, as shown in Figure 34.

```

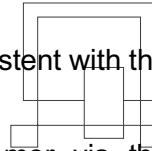
public class FlightModelManager extends Observable implements FlightModel
{
    private FlightList list;
    private CustomerList customerList;
    private FlightListServer server;
    private TargetDatabase databaseConnection;

    public FlightModelManager() {
        databaseConnection = new FlightDatabaseAdapter(InitServer.getInstance().getDriver(),
            InitServer.getInstance().getURL(), InitServer.getInstance().getUser(), InitServer.getInstance().getPw());
        FlightList flightList = new FlightList();
        for (int i = 0; i < databaseConnection.getAllFlights().length; i++)
        {
            flightList.addFlight(databaseConnection.getAllFlights()[i]);
        }
        list = flightList;
        CustomerList customerList = new CustomerList();
        for (int i = 0; i < databaseConnection.getAllCustomers().length; i++)
        {
            customerList.addCustomer(databaseConnection.getAllCustomers()[i]);
        }
        list = flightList;
        this.customerList = customerList;
        server = new FlightListServer(this);
    }
}

```

Figure 34. FlightModelManager on the server side

As shown in Figure 34, the FlightList and CustomerList instances are loaded with the information from the database. This means that the FlightModelManager class must remove



the ticket both from the Model and the database, so that the Model stays consistent with the database.

Thus, in the `cancelTicket` method, it removes the ticket from the customer via the `removeTicket` method called on the `CustomerList` instance, makes the seat available again, for the flight via the `cancelTicket` method called on the `FlightList` instance and removes the information about the ticket in the database using the `cancelTicket` method on the database adapter class. This can be observed in Figure 35.

```
@Override
public void cancelTicket(String flightID, String customerID,
    String seatValue, Ticket ticket)
{
    list.cancelTicket(flightID, seatValue);
    customerList.removeTicket(customerID, flightID);
    databaseConnection.cancelTicket(flightID, customerID, seatValue, ticket);
    notifyObservers(Log.getInstance().getTimestamp() + " Ticket " + ticket.getTicketID() + " canceled by Customer " + customerID + "");
}
```

Figure 35. `cancelTicket` method in `FlightModelManager` class on the server side

The `cancelTicket` method in the database adapter creates two SQL statements in form of two strings and use them to call the update method in the `MyDatabase` class from the Persistence layer. This is shown in Figure 36.

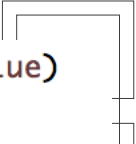
```
@Override
public void cancelTicket(String flightID, String customerID,
    String seatValue, Ticket ticket)
{
    String statement = "DELETE FROM Ticket WHERE flightID = '" + flightID + "' AND customerID = '" + customerID + "';";

    String statement1 = "UPDATE Seat SET seatValue = false WHERE flightID = '" + flightID
        + "' AND " + "seatPo = '" + seatValue + "';";

    try
    {
        database.update(statement, null);
        database.update(statement1, null);
    }
    catch (SQLException e)
    {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
```

Figure 36. `cancelTicket` method in the database adapter

The `cancelTicket` method in the `FlightList` class finds the `Flight` object with the given `flightId` and makes the seat with the specified number available again. This is shown in Figure 37.



```
public void cancelTicket(String flightId, String seatValue)
{
    for (int i = 0; i < flights.size(); i++)
    {
        if (flights.get(i).getId().equals(flightId))
        {
            flights.get(i).setSeatUnoccupied(seatValue);
        }
    }
}
```

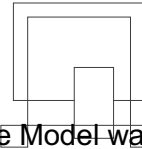
Figure 37. cancelTicket method in the FlightList class

The removeTicket method in the CustomerList class finds the Customer object with the given customerId and removes the Ticket object related to it. This is shown in Figure 38.

```
public void removeTicket(String customerId, String flightId)
{
    for (int i = 0; i < customers.size(); i++)
    {
        if (customers.get(i).getId().equals(customerId))
        {
            customers.get(i).removeTicket(flightId);
        }
    }
}
```

Figure 38. removeTicket method in the CustomerList class

All the source code for the system can be found in Appendix E, including the Jar Files.



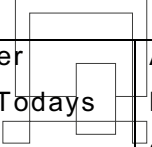
6 Test

The system was both tested using the Black-Box and White-Box approach. The Model was tested using JUnit tests while the entire actual system has been tested using Test Cases. The full JUnit tests can be found in Appendix F.

6.1 Test Specifications

Test Cases have been created to verify that the Use Cases and the requirements have been fulfilled. The following table displays all the test specifications:

Test Case	Test Case Description	Actor	Precondition	Expected Result	Steps	Result
1.	Booking the cheapest flight with flights in the system.	Customer	To be logged in & to have flights in the system.	A flight has been booked and a ticket linked to the user.	1. The user presses "Cheapest flights" 2. System displays a list of ten cheapest flights. 3. User selects a flight from the list. 4. User presses "Book". 5. The system displays a list of available seats. 6. The user chooses a seat. 7. The user presses "Confirm".	A flight has been booked and a ticket linked to the user.

2.	Booking a flight that is today.	Customer	To be logged in & to have flights in the system.	A flight has been booked and a ticket linked to the user.	 <ol style="list-style-type: none"> 1. The user presses "Today's flights". 2. System displays a list of flights that are today. 3. User selects a flight from the list. 4. User presses "Book". 5. The system displays a list of available seats. 6. The user chooses a seat. 7. The user presses "Confirm". 	A flight has been booked and a ticket linked to the user.
3.	Booking a flight that has a certain origin, destination, date of departure and arrival.	Customer	To be logged in & to have flights in the system.	A flight has been booked and a ticket linked to the user.	<ol style="list-style-type: none"> 1. The user selects an origin, destination, date of departure and date of arrival. 2. The user presses the button "Search". 3. User selects a flight from the list. 4. User presses "Book". 	A flight has been booked and a ticket linked to the user.

					<p>5. The system displays a list of available seats.</p> <p>6. The user chooses a seat.</p> <p>7. The user presses "Confirm".</p>	
4.	Booking the cheapest flights while there are no flights.	Customer	To be logged in & to not have flights in the system	The system displays a message "no flights"	1. The user presses "Cheapest flights"	The system displays a message "no flights"
5.	Booking a flight from today while there are no flights.	Customer	To be logged in & to not have flights in the system	The system displays a message "no flights for today"	1. The user presses "Today's flights"	The system displays a message "no flights for today"
6.	Booking a flight that has a certain origin, destination, date of departure and arrival while there are no flights.	Customer	To be logged in & to not have flights in the system	The system displays a message "no flights"	<p>1. The user selects an origin, destination, date of departure and date of arrival.</p> <p>2. The user presses the button "Search".</p>	The system displays a message "no flights"
7.	Booking a flight that has a certain origin, destination, date of departure arrival while not selecting the date of	Customer	To be logged in.	The system displays a message "Please use the date choosers to select date"	1. The user presses "Search"	The system displays a message "Please use the date choosers to select date"

	departure and date of arrival.					
8.	Booking a flight that has been already booked.	Customer	To be logged in & be booking a selected flight.	The system displays the message "You already booked a ticket for this flight!"	1. The system displays a list of available seats. 2. The user chooses a seat. 3. The user presses "Confirm".	The system displays the message "You already booked a ticket for this flight!"
9.	Booking a flight that has been already deleted.	Customer	To be logged in & be booking a selected flight that has been deleted from the system.	The system displays the message "Flight was removed by the administrator!"	1. The system displays a list of available seats. 2. The user chooses a seat. 3. The user presses "Confirm".	The system displays the message "Flight was removed by the administrator!"
10	Booking a flight that has not been selected.	Customer	To be logged in & having searched for flights.	The system displays the message "Please select a flight to book!"	1. The user presses "Cheapest flights" 2. System displays a list of ten cheapest flights. 3. User presses "Book".	The system displays the message "Please select a flight to book!"
11.	Removing a booked ticket.	Customer	To be logged in & have a flight booked.	Ticket removed.	1. The user presses "My profile" and then "View profile" 2. User selects a ticket from "My flight history"	The ticket has been removed.

					3. User presses "Cancel ticket" 4. The system displays a message "Are you sure you want to cancel the ticket?" 5. The user presses "Yes"	
12.	Removing a ticket while not selecting one.	Customer	To be logged in & have a flight booked.	The system displays the message "Please select a flight to remove!"	1. The user presses "My profile" and then "View profile" 2. User presses "Cancel ticket"	The system displays the message "Please select a flight to remove!"
13.	Adding a flight.	Administrator	No precondition	The system displays the message "Successfully added" and a flight is added.	1. Administrator chooses origin city, destination city, departure date, arrival date, departure time, arrival time and types in the price. 2. Administrator validates his choice.	The system displays the message "Successfully added" and a flight is added.
14.	Removing a flight without searching.	Administrator	Having flights in the system	System displays the message "Successfully removed" and the	1. System displays a list of all flights; 2. Administrator selects the	System displays the message "Successfully removed" and

				flight is removed.	specific flight to remove; 3. Administrator validates his choice.	the flight is removed.
15.	Removing a flight that has been searched for.	Administrator	Having flights in the system	System displays the message "Successfully removed".	1. Administrator types in the flightID. 2. System displays the flight with that flightID. 3. Administrator selects the flight. 4. Administrator validates his choice.	System displays the message "Successfully removed".
16.	Adding a flight when the destination city is the same as the arrival city.	Administrator	No precondition	System displays the message "Origin and destination must have different values! Make sure the price field is completed as well".	1. Administrator chooses origin city, destination city, departure date, arrival date, departure time, arrival time and types in the price. 2. Administrator validates his choice.	System displays the message "Origin and destination must have different values! Make sure the price field is completed as well".
17.	Adding a flight when the departure date is before the current date.	Administrator	No precondition	System displays the message "Departure date should	1. Administrator chooses origin city, destination city, departure date, arrival date,	System displays the message "Departure date should not

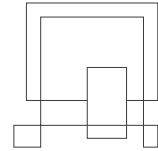
				not be before today".	departure time, arrival time and types in the price. 2. Administrator validates his choice.	be before today".
18.	Adding a flight when the arrival date is before the departure date.	Administrator	No precondition	System displays the message "Arrival date cannot be before the departure date!".	1. Administrator chooses origin city, destination city, departure date, arrival date, departure time, arrival time and types in the price. 2. Administrator validates his choice.	System displays the message "Arrival date cannot be before the departure date!".
19.	Adding a flight when there is no arrival date or departure date chosen.	Administrator	No precondition	System displays the message "Please use the date choosers to select the date".	1. Administrator chooses origin city, destination city, departure time, arrival time and types in the price. 2. Administrator validates his choice.	System displays the message "Please use the date choosers to select the date".
20.	Adding a flight when departure time and arrival	Administrator	No precondition	System displays the message	1. Administrator chooses origin city, destination	System displays the message

	date coincide, while the departure date and arrival date coincide.			"Arrival time cannot have the same value as the departure time!".	city, departure date, arrival date, departure time, arrival time and types in the price. 2. Administrator validates his choice.	"Arrival time cannot have the same value as the departure time!".
21.	Adding a flight when arrival time is before the departure time, while the departure date and arrival date coincide.	Administrator	No precondition	System displays the message "Arrival time cannot be before the departure time!".	1. Administrator chooses origin city, destination city, departure date, arrival date, departure time, arrival time and types in the price. 2. Administrator validates his choice.	System displays the message "Arrival time cannot be before the departure time!".
22.	Adding a flight when the introduced price value is not made exclusively of numbers.	Administrator	No precondition	System displays the message "Price has to be a number! No other characters are allowed.".	1. Administrator chooses origin city, destination city, departure date, arrival date, departure time, arrival time and types in the price. 2. Administrator validates his choice.	System displays the message "Price has to be a number! No other characters are allowed.".

23.	Adding a flight that is already in the system.	Administrator	No precondition	System displays the message "This flight already exists in the system."	1. Administrator chooses origin city, destination city, departure date, arrival date, departure time, arrival time and types in the price. 2. Administrator validates his choice.	System displays the message "This flight already exists in the system."
24.	Removing a flight while not selecting one.	Administrator	Having flights in the system	System displays the message "Please select a flight to delete!"	1. Administrator types in the flightID. 2. System displays the flight with that flightID. 3. Administrator validates his choice.	System displays the message "Please select a flight to delete!"
25.	Removing a flight that cannot be found.	Administrator	Having flights in the system	System displays the message "No flight with the id has been found".	1. Administrator types in the flightID.	System displays the message "No flight with the id has been found".
26.	Removing a found flight while not selecting one.	Administrator	Having flights in the system	System displays the message "Please	1. Administrator types in the flightID.	System displays the message "Please select

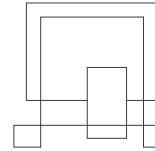
				select a flight to delete!".	2. System displays the flight with that flightID. 3. Administrator validates his choice.	a flight to delete!".
--	--	--	--	------------------------------	---	-----------------------

7 Results and Discussion



All the defined Use Cases have been implemented successfully. We managed to fulfil all assigned tasks, whilst also following the Design diagrams. Testing proved that the system meets all of the defined requirements.

The system was developed and delivered in time. Team evaluates the result as satisfactory.



8 Conclusions

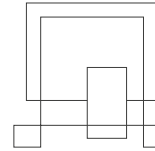
As stated in the introduction part, the purpose of the project was aimed at helping Zair to manage their bookings in a more efficient manner and creating an attractive interface. Team conducted the analysis and it resulted in 20 requirements (16 functional requirements, 4 non-functional requirements).

Based on the requirements team defined main conceptual classes and made a domain model which represents the overall structure and main concept of the system. Domain model was used as a basis for the Design Class Diagram which describes specifically the structure of the system, and relations and dependencies between classes used in implementing the actual system.

Design part includes requirement-based Use Case description and all Use Case related diagrams that represent not only general behaviour of the system but also show detailed sequencing inside the system including interaction between systems logical parts. Besides the basic flows, sequencing of alternative scenarios is also shown in the appended diagrams.

Testing topic clearly describes all cases to be tested. By testing the system team makes sure that results of all scenarios match the expected result.

Altogether, all functional and non-functional requirements have been met. The system works flawlessly and accomplishes everything that it has to do. The system was designed and implemented as efficient as possible.



9 Project future

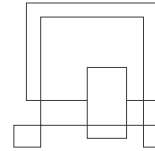
All requirements have been met and all standard alternative scenarios were considered, however there more improvements and features that could be implemented in the system.

The first step in developing the system could be improving the 'Personal Account' part. Right now, system does not allow to edit personal information once customer entered it when registering. Frequently customers personal data (such as last name, phone number, passport number) can be changed and it should match the one that is stored in the system.

Moreover, inability to change a password is insecure. And inability to recover the password once customer has forgotten it can cause storing inactive profiles in the system as well as multiple profiles with the same personal data.

Besides, there are more features to be implemented in the process of booking a ticket by a customer. Currently the system possesses only standart seats, without having business or premium class tickets and also system does not consider customers to have luggage.

Also the customer loyalty program could be implemented in our system such as the customers frequently booking tickets for the flights would get extra discounts.



10 Sources of information

Banger, D., 2014. A Basic Non-Functional Requirements Checklist « Thoughts from the Systems front line.... Available at: <https://dalbanger.wordpress.com/2014/01/08/a-basic-non-functional-requirements-checklist/> [Accessed January 31, 2017].

Business Analyst Learnings, 2013. MoSCoW : Requirements Prioritization Technique — Business Analyst Learnings. , pp.1–5. Available at: <https://businessanalystlearnings.com/ba-techniques/2013/3/5/moscow-technique-requirements-prioritization> [Accessed January 31, 2017].

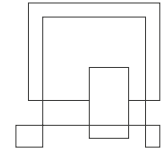
Larman, C., 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*,

YourCoach, S.M.A.R.T. goal setting | SMART | Coaching tools | YourCoach Gent. Available at: <http://www.yourcoach.be/en/coaching-tools/smart-goal-setting.php> [Accessed August 19, 2017].

Wikipedia, 2012, *Functional requirement* [online]. Available at: <https://en.wikipedia.org/wiki/Functional_requirement> [Accessed 5 June 2018]

Baeldung, 2018, Flyweight Pattern in Java [online]. Available at <<http://www.baeldung.com/java-flyweight>> [Accessed 5 June 2018]

Thomas C., 2010. *Database Systems, A practical Approach to Design, Implementation, and Management*.Gr



11 Appendices

Appendix A: Project Description.

Appendix B: Use Cases, Use Cases descriptions, SSDs, Activity diagrams, Domain Model.

Appendix C: Design Class diagrams.

Appendix D: Sequence Diagrams

Appendix E: Source code + Jar Files.

Appendix F: JUnit tests.

Appendix G: User guide.

Appendix H: Javadoc.