Project Report

Airline Reservation System

**Heterogeneous System**

**Names of Students:**

Jan Vasilcenko – 293098

Karrtiigehyen Veerappa - 293076

Nicolas Popal - 279190

Patrik Horny – 293112

**Supervisors:**

Jakob Knop Rasmussen

Ole Ildsgaard Hougaard

**VIA University College**

****

**Number of characters:**

**Software Technology Engineering**

**Semester 3**

**15 November 2020**

Table of Contents

[1 Introduction (Karrtiigehyen and Nicolas Popal) 4](#_Toc56525987)

[1.1 Background Description (Karrtiigehyen and Nicolas Popal) 4](#_Toc56525988)

[1.2 Definition of Purpose (Karrtiigehyen and Nicolas Popal) 5](#_Toc56525989)

[1.3 Delimitations (Karrtiigehyen and Nicolas Popal) 5](#_Toc56525990)

[2 Analysis (Everybody) 6](#_Toc56525991)

[2.1 Requirements (Everybody) 6](#_Toc56525992)

[2.1.1 Functional Requirements (Everybody) 6](#_Toc56525993)

[2.1.2 Non-Functional Requirements (Everybody) 8](#_Toc56525994)

[2.2 Use Case Diagram (Everybody) 9](#_Toc56525995)

[2.3 Use Case Descriptions (Everybody) 10](#_Toc56525996)

[2.4 System Sequence Diagram (Nicolas Popal and Patrik Horny) 12](#_Toc56525997)

[2.5 Domain Model () 16](#_Toc56525998)

[3 Design (Everybody) 17](#_Toc56525999)

[3.1 Architecture (Karrtiigehyen and Nicolas Popal) 17](#_Toc56526000)

[3.1.1 Tiers (Karrtiigehyen and Nicolas Popal) 18](#_Toc56526001)

[3.1.2 Communications Between the Tiers (Karrtiigehyen and Nicolas Popal) 18](#_Toc56526002)

[3.2 Design of Airline System () 19](#_Toc56526003)

[3.3 Database (Karrtiigehyen and Nicolas Popal) 20](#_Toc56526004)

[3.3.1 Conceptual Model (Karrtiigehyen and Nicolas Popal) 20](#_Toc56526005)

[3.3.2 Logical Model (Karrtiigehyen and Nicolas Popal) 24](#_Toc56526006)

[3.4 Technologies Used (Karrtiigehyen and Nicolas Popal) 26](#_Toc56526007)

**List of Diagrams**

[Diagram 1: Use case diagram of airline reservation system 9](#_Toc56526008)

[Diagram 2: System sequence diagram for creating flights 13](#_Toc56526009)

[Diagram 3: System sequence diagram for booking flights 14](#_Toc56526010)

[Diagram 4: Sequence diagram for delaying flights 15](#_Toc56526011)

[Diagram 5: System architecture diagram 17](#_Toc56526012)

[Diagram 6: EE/R diagram of the database 20](#_Toc56526013)

[Diagram 7: Section 1 of the EE/R diagram 21](#_Toc56526014)

[Diagram 8: Section 2 of the EE/R diagram 22](#_Toc56526015)

[Diagram 9: Section 3 of the EE/R diagram 23](#_Toc56526016)

**List of Tables**

[Table 1: Use case description for Create flights 11](#_Toc56526027)

[Table 2: Use case description for Book flights 11](#_Toc56526028)

[Table 3: Use case description for Delay flights 12](#_Toc56526029)

# Introduction (Karrtiigehyen and Nicolas Popal)

## Background Description (Karrtiigehyen and Nicolas Popal)

Aviation has allowed mankind to visit other countries and continents in a matter of hours. Once, aviation was only available to people of power and money, but nowadays, commercial airlines provide flights to the public. Not only that, airlines provide extra services such as hotels, car rentals and insurance to the public as well. Tickets to these flights are mainly booked online through a website.

An airline reservation system (ARS) should be simple to use and easy to understand from the customer’s point of view. Unfortunately, this is not the case. A survey conducted on the satisfaction of customers using ARSs indicate that a big group of customers are more than unhappy with the ARSs that they have used.

Aviation is an unpredictable business due to the scale, weather patterns and most importantly, fuel prices (CAPA, 2017). Due to this, customers can often have unpleasant surprises such as delays and cancellation of flights, and many flight booking websites warn customers when it is too late, which leads to frustration among customers. Refunds can also take a long time to process, which often leads to customers wasting their time contacting the airlines for refunds. This entire process leaves customers confused and irritated (Blessing, Umar and Opeyemi A., 2017).

The people operating the flight booking systems, operators, have a complicated job coordinating and organizing the flights (Aviation Job Search, 2012). The last thing an operator would want to do is input or edit the wrong data in the ARS. So, easing this job with an improved ARS on the operator side will lead to less mistakes and a more efficient workflow.

In a nutshell, customers experiences with the current airline reservation systems are unsavory. ARS should be easily navigable by customers, and simple to use. Poor notification of cancellations and delays of flights to customers leads to low customer satisfaction, which will lead the customers to switch to another ARS or reserve directly at the airport, which is less efficient and more time consuming.

## Definition of Purpose (Karrtiigehyen and Nicolas Popal)

The purpose is to help customers book flight tickets and operators to manage the flight booking system, so that the customers can manage and keep track of the status of their flight tickets, and operators can manage and update flights with ease.

## Delimitations (Karrtiigehyen and Nicolas Popal)

1. We will not include anything regarding insurance.

2. We will not include anything beyond the flight, for example cars and hotels at the destination.

3. We will not include rebooking services.

The methodology used, time schedule, and risk assessment can be found in the Project Description, which can be found in Appendix A.

# Analysis (Everybody)

In the analysis, the problem domain will be further defined by elaborating upon the contents from the previous section. Firstly, the requirements will be explored in the shape of user stories. Then, use case diagram and descriptions are presented to show the different actors and scenarios involved, from which System Sequence Diagrams can be erected. The conclusion of this section will be with the presentation of the domain model of the system, which will be used as the groundwork for the 3rd section, Design.

## Requirements (Everybody)

SMART principles were used as the framework for the requirements, so that the requirements are testable.

### Functional Requirements (Everybody)

**Critical priority**

1. As a customer, I want to book flights, so that I can fly to my destination.
2. As a customer, I want reserve seats on flights, so that I can sit on the favoured seat in the flight.
3. As a customer, I want to view available flights, so that I can book my preferred flights.
4. As a customer, I want to choose the date of the flights, so that I can reach my destination in my desired date.
5. As a customer, I want to be notified when my booked flights get delayed or cancelled, so that I will be aware of it.
6. As a customer, I want to be able to view the status of my booked flights, so that I will know the status of my booked flights.
7. As a customer, I want to register myself as a customer, so that I can use the airline reservation system.
8. As a customer, I want to be able to log in as a customer, so that I can access the airline reservation system.
9. As a customer, I want to view my booked flights, so that I know which flights I have booked.
10. As a customer, I want to cancel a booked flight, so that the flight is not booked to me anymore.
11. As a customer, I want to get a code from booked flights, so that I can use that information to check in at the airport.
12. As an operator, I want to create flights, so that the customers can view them.
13. As an operator, I want to edit the flight information on existing flights, so that the customers can have the latest flight information.
14. As an operator, I want to cancel flights, so that the cancelled flights will be removed and customers who have booked the cancelled flights will be notified.
15. As an operator, I want to log in as an operator, so that I can access the airline reservation system.

**High priority**

1. As a customer, I want to be able to refund booked flights, so that I can refund flights that are cancelled.
2. As a customer, I want to search for specific flights by their destination, so that I can narrow my flight search.
3. As a customer, I want to view the details of the flights such as the destination, time taken and date, so that I know the details of the flight I am about to book.
4. As a customer, I want the option to buy additional luggage, so that I can bring more luggage to the flight.
5. As an operator, I want to delay flights, so that customers will be able to check whether their flights are delayed.
6. As an operator, I want to view information about the customers, so that I can view which flights are booked by who.

**Low priority**

1. As a customer, I want to edit my user info, so that it is updated.
2. As a customer, I want to view my name when I log in, so that I can identify if I have logged into my account.
3. As a customer, I want to view the previous flights that I have taken, so that I can look up which flights I have taken.

### Non-Functional Requirements (Everybody)

1. The airline reservation system should be a heterogeneous system, using Java and C#.
2. The airline reservation system should be a distributed system.
3. The airline reservation system should use sockets and web services.
4. Customers cannot modify the flight information.
5. Customers cannot view other users’ information.
6. The system must be usability tested by end users.
7. The airline reservation system should work on separate computers over the internet.
8. There will only one type of seat offered by the system to customers.

## Use Case Diagram (Everybody)



Diagram 1: Use case diagram of airline reservation system

As seen from Diagram 1, there are three actors – *Visitor, Customer* and *Operator*. When a user has not performed login or register, the actor is then a *Visitor.* Once the *Visitor* has logged in or registered, the actor is then either *Customer* or an *Operator* based on their account type*.*

*Customer* can book flights, which are not fully booked. While booking flights, *Customer* can reserve their seat position, provided that the chosen seat is not already reserved by another *Customer*. The *Customer* can view available flights to book and view already booked flights. The *Customer* can also cancel flights that are booked, and edit their account information. The *Customer* will receive notifications if any of their booked flights are delayed or cancelled.

The relationship between the *Customer* and *Operator* is generalized, which means that the *Operator* has all of the use cases of *Customer,* while its own use cases. The additional use cases of *Operator* are create flights, cancel flights, edit flight information, view customer information and delay flights.

## Use Case Descriptions (Everybody)

The use case descriptions show how each scenario will play out. The use case descriptions are based on the use case diagram that was just shown.

The use case descriptions below are for Book Flights, Create Flights and Delay Flights. These three cases will serve as the main examples throughout this report, and will be referred to frequently. These use cases are chosen because the book flights use case is from the *Customer’s* point of view, and the create flights use case is from the *Operator’s* point of view. The delay flights use case involves both actors, where the use case is initiated by the *Operator,* and the *Customer* is affected by it via a notification.

Rest of the use case descriptions can be found in Appendix B.

**Create flights**

|  |  |
| --- | --- |
| **Use case** | Create flights |
| **Summary** | The operator creates a new flight |
| **Actor** | Operator |
| **Precondition** | Operator must have all details about new flight |
| **Postcondition** | The flight is created |
| **Main scenarios** | 1. The operator chooses to create new flight  2. The operator needs to insert information about flight  3. The system validates data of the new flight  4. The flight is created and stored in the database |
| **Alternative scenarios** | 2a The flight information is invalid   1. System informs operator, that information is invalid |

Table 1: Use case description for Create flights

**Book flights**

|  |  |
| --- | --- |
| **Use case** | Book flight |
| **Summary** | The customer can book a flight |
| **Actor** | Customer |
| **Precondition** | Customer must be logged into their account and the flight must have available seats |
| **Postcondition** | The chosen flight is booked |
| **Main scenarios** | 1. The customer chooses the flight that they want to book.  2. The system shows details regarding the flight.  3. The customer books seats.  4. The customer selects what type of luggage they want to bring.  5. The customer purchases the ticket.  6. The system generates an unique code for the ticket purchase. |
| **Alternative scenarios** | At any time during step 1 customer cancels  1. The use case ends  3a The customer does not want to reserve desired seats  1. System chooses a random seat for the customer. |

Table 2: Use case description for Book flights

**Delay flights**

|  |  |
| --- | --- |
| **Use case** | Delay flights |
| **Summary** | The operator delays a chosen flight |
| **Actor** | Operator |
| **Precondition** | There is at least one created flight in database |
| **Postcondition** | The departure and arrival time for the flight is updated, and notification is sent to customers who booked this flight |
| **Main scenarios** | 1. The operator chooses a flight to delay  2. A message asking to confirm the delay  2. The flight is delayed  3. Customers get notification about delay |
| **Alternative scenarios** | At any time during step 1 and 2 operator cancels  1. The use case ends |

Table 3: Use case description for Delay flights

## System Sequence Diagram (Nicolas Popal and Patrik Horny)

The examples shown here are for the System Sequence Diagrams of book flights, create flights and delay flights. All the system sequence diagrams can be found in Appendix C.

System sequence diagram (SSD) is based on use case description. In SSD the system is treated as a black box, which means that the inner workings of the system is not displayed or known. For an example, the system sequence diagrams used for booking, creating, and delaying flights in the airline reservation system shows the input and output events of the system.

**Create flights**

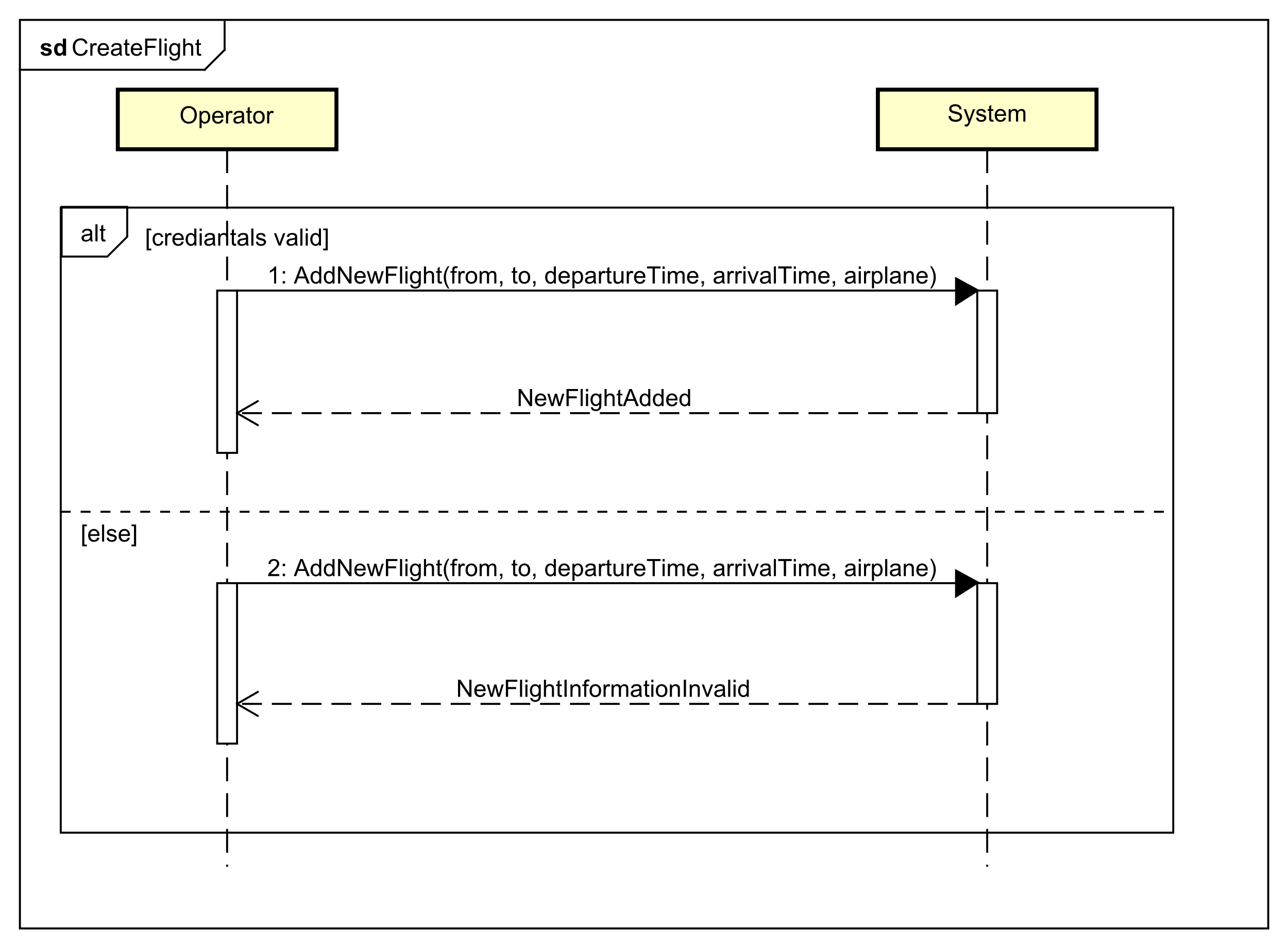


Diagram 2: System sequence diagram for creating flights

This system sequence diagram is from the *Operator’s* point of view. There are two lifelines, *Operator* and *System.*

If the credentials of the *Operator* are correct, then the *Operator* can create flights with parameters origin airport, destination airport, departure and arrival times, dates, and the airplane. Else, the *Operator* would be informed that a new flight cannot be created.

**Book flights**

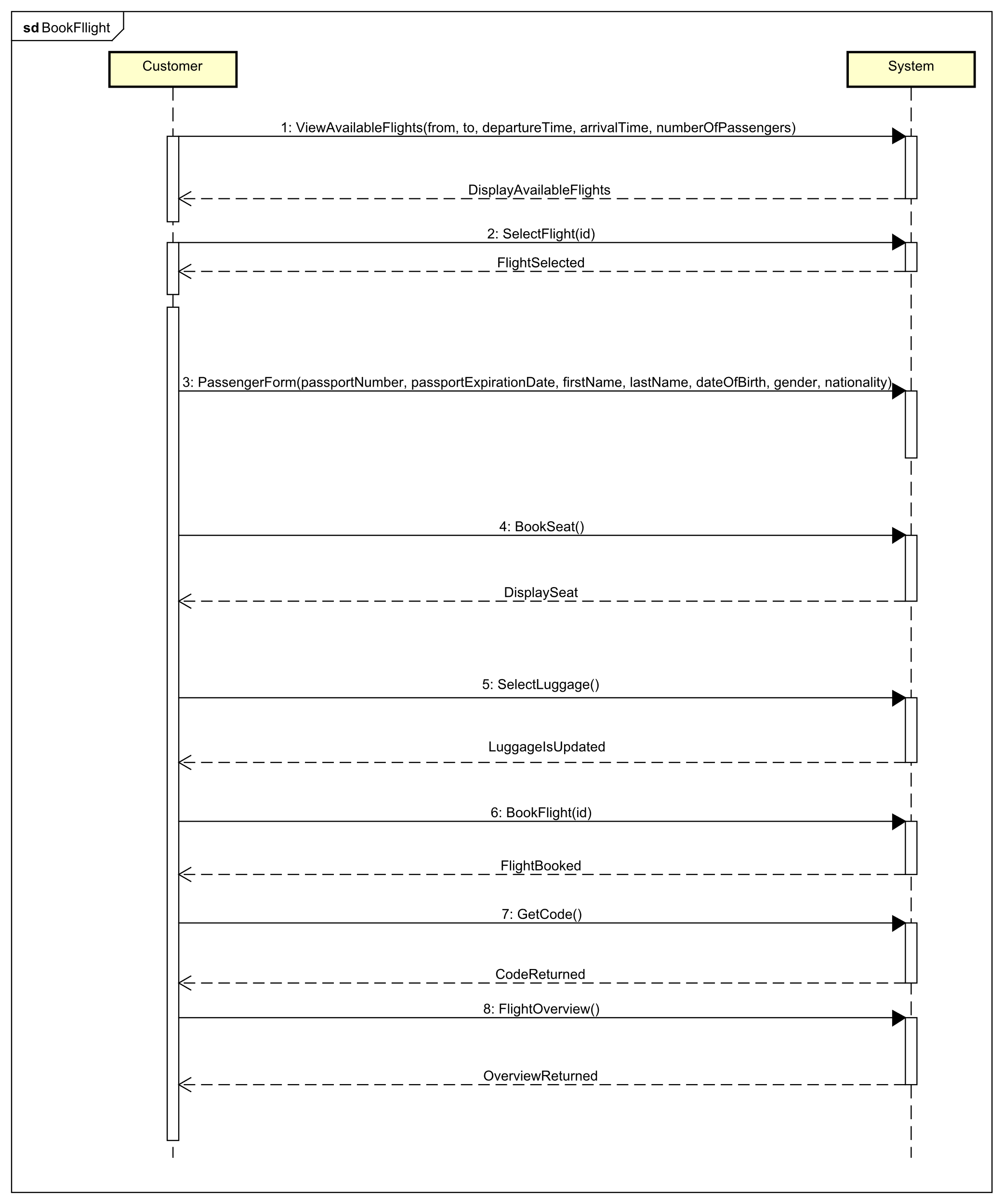


Diagram 3: System sequence diagram for booking flights

In this sequence diagram, the actor is *Customer,* and the lifelines are *Customer* and *System.*

At first, the *Customer* will be able to view available flights based on the *Customer’s* input, origin airport, destination airport, departure and arrival times, date, and the number of passengers. The number of passengers is important for the booking of multiple tickets for more than one person. The *System* will display available flights corresponding to the *Customer’s* preferences. The *Customer* will select a flight to be booked, and the flight details will be shown to the *Customer.* Then, the *Customer* will have to fill out forms on the information of passengers. Then, the *Customer* chooses the seats and luggage for each passenger. Finally, the *Customer* can book the flight if there are no invalid statements, and the overview of the flight is shown back to the *Customer.*

**Delay flights**

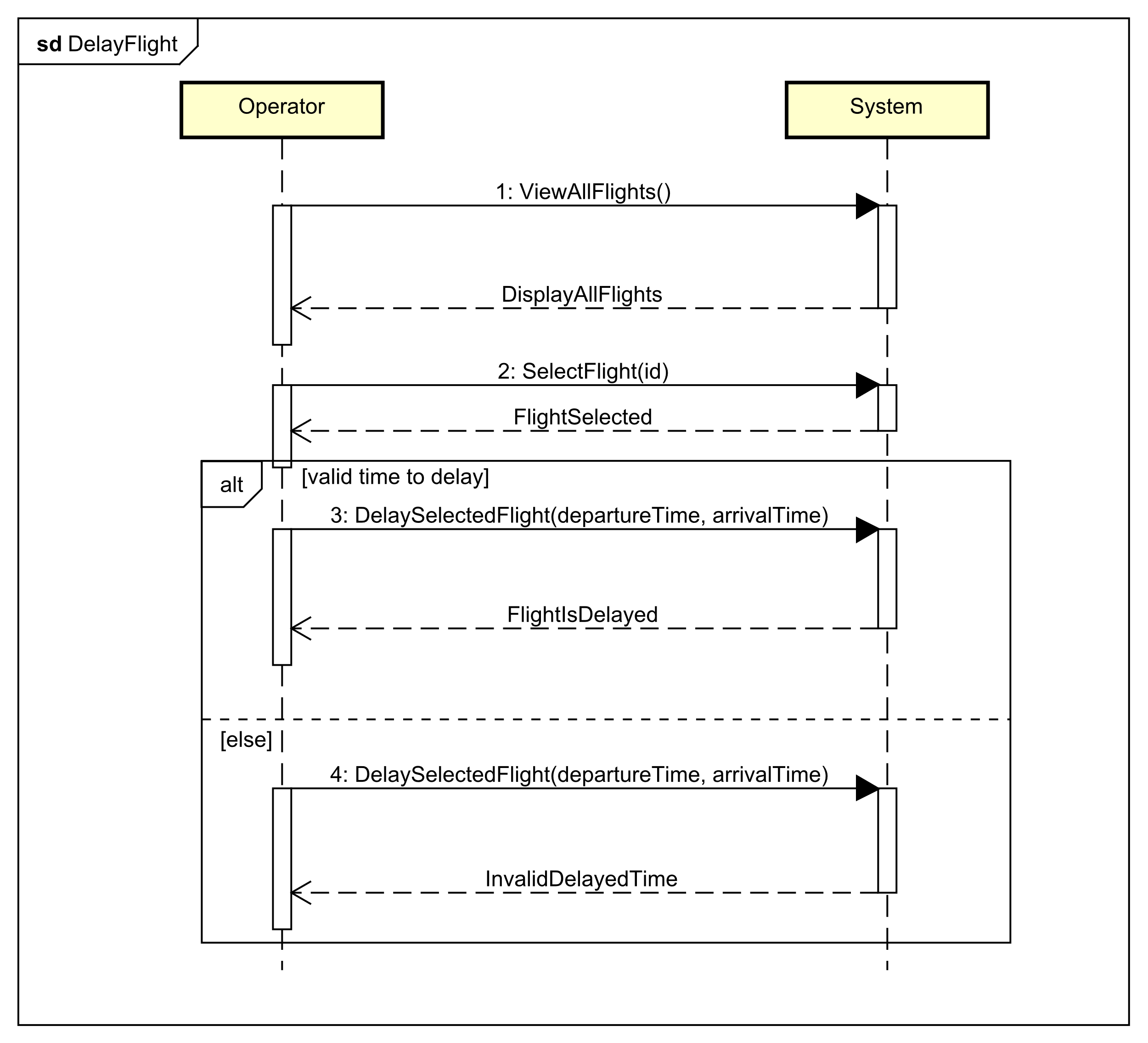


Diagram 4: Sequence diagram for delaying flights

This sequence diagram is initiated by the operator, and there are two lifelines, which are *Operator* and *System.*

Firstly, the *Operator* views all the flights and will choose which flight to delay. If the chosen flight is an upcoming flight (not a flight that is before the current date), then the *Operator* can input the new departure and arrival time, and it will be updated in the system. Else, if the date of the chosen flight is before the current date, and the new delayed date of the flight is before the current date, then the system will throw an error.

## Domain Model ()

Text

# Design (Everybody)

In this section, the software will be designed more in-depth. The topics that will be discussed are the architecture of the system, design patterns, technologies used, and UI design choices. The outcome of this section will contain the necessary knowledge and diagrams that will be used for the implementation of the system.

## Architecture (Karrtiigehyen and Nicolas Popal)

The architecture chosen for the airline reservation system is the three-tier architecture. The three-tier architecture is a client-server architecture pattern where the presentation, application and data tiers are separated. This has many benefits. For example, there is a separation of concerns. This reduces coupling and dependencies, while improving cohesion and increasing reusability of code. Another advantage to using the three-tier architecture is that any tier can be replaced or modified independently.

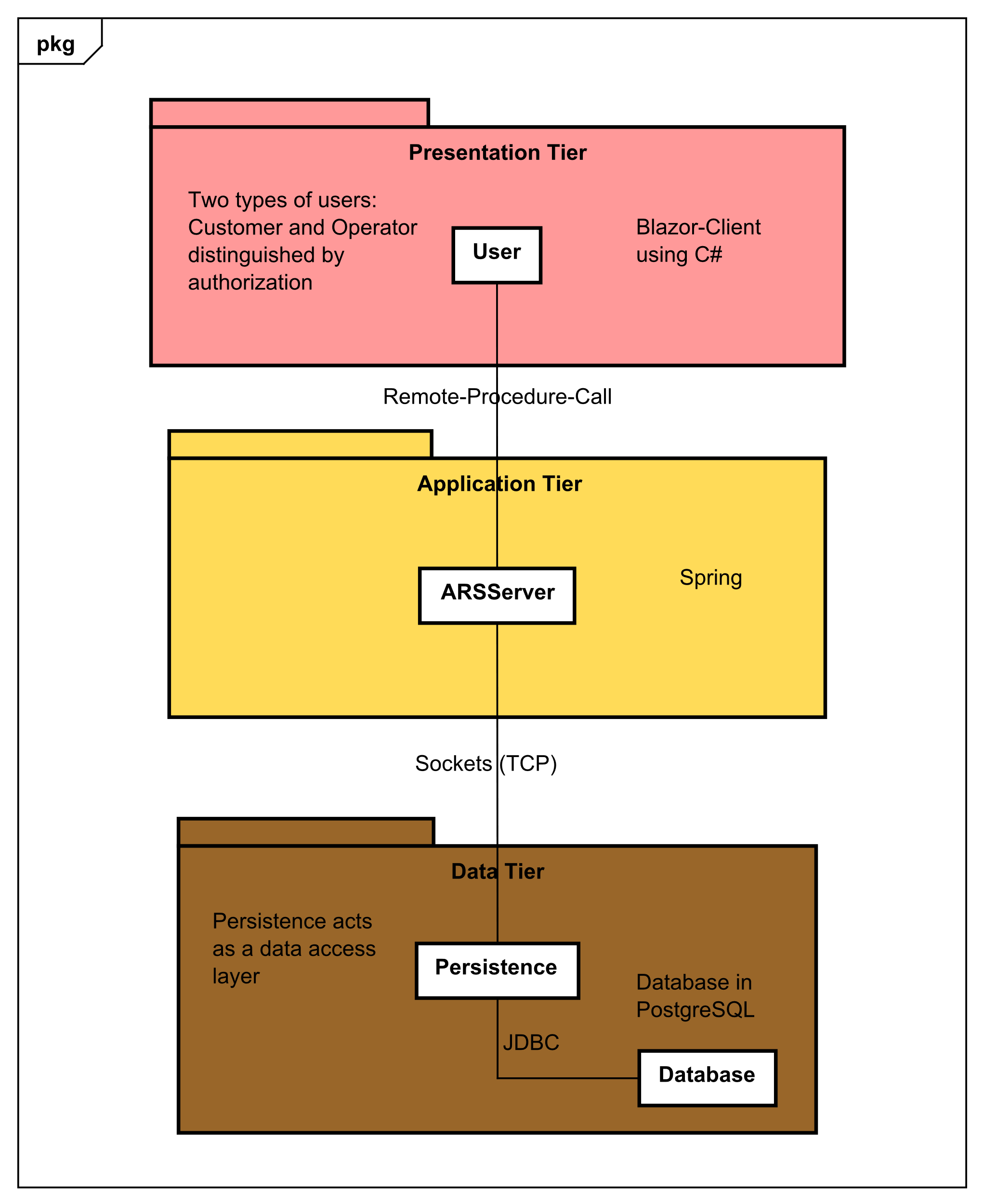


Diagram 5: System architecture diagram

### Tiers (Karrtiigehyen and Nicolas Popal)

In the presentation tier, the topmost level of the application, displays information regarding services such as booking flights, viewing available flights, and creating flights. The result will be displayed on a web page in the form of a GUI. This is achieved by using *Blazor*, namely *Blazor WebAssembly*. The reason for choosing to work with *Blazor* is that it offers more flexibility and options compared to other GUIs. The entire presentation tier is programmed in *C#.* As shown in the use case diagram in the Analysis section, the two actors, *Customer* and *Operator.* These actors are distinguished by authorization in the presentation tier. For more on how this is achieved, look in the Implementation section.

The application tier holds the business rules of the system. This tier is also the middleware of the system. This is a component-based middleware instead of an object-based middleware. This is so that the dependencies will be made explicit and will provide a more complete contract for system construction. The middleware is create using the *Spring Boot* framework. *Spring Boot* is used since it gives a flexible way to database transactions and *XML* configurations. It also manages *REST* endpoints. The middleware is programmed in *Java*.

The data tier houses the persistence layer and the database. The persistence layer consists of the Data Access Objects (DAO), and data can be stored and retrieved from the database. The persistence layer is programmed in *Java*, while the *PostgreSQL* is used for the database. This tier depends on no other tiers, while the application tier is dependent on the data tier, and the presentation tier is dependent on the application tier. For more on the *DAO*, look in the subsubsubsection Persistence Layer.

### Communications Between the Tiers (Karrtiigehyen and Nicolas Popal)

The communication used between the presentation tier and application tier is Remote Procedure Call. WHY? Alternatives

Sockets are used for the communication between the middleware (application tier) and the persistence layer in the data tier. The socket protocol used for this is *TCP* since *TCP* ensures that there will be a lossless and reliable data transmission, which is needed for this system. *UDP* would be incompatible since reliability of the data transmission is of a higher importance than performance.

The persistence layer in the data tier can perform *CRUD* operations on the database via *JDBC*.

## Design of Airline System ()

A lot of text

## Database (Karrtiigehyen and Nicolas Popal)

### Conceptual Model (Karrtiigehyen and Nicolas Popal)

The conceptual model is heavily influenced by the domain model.

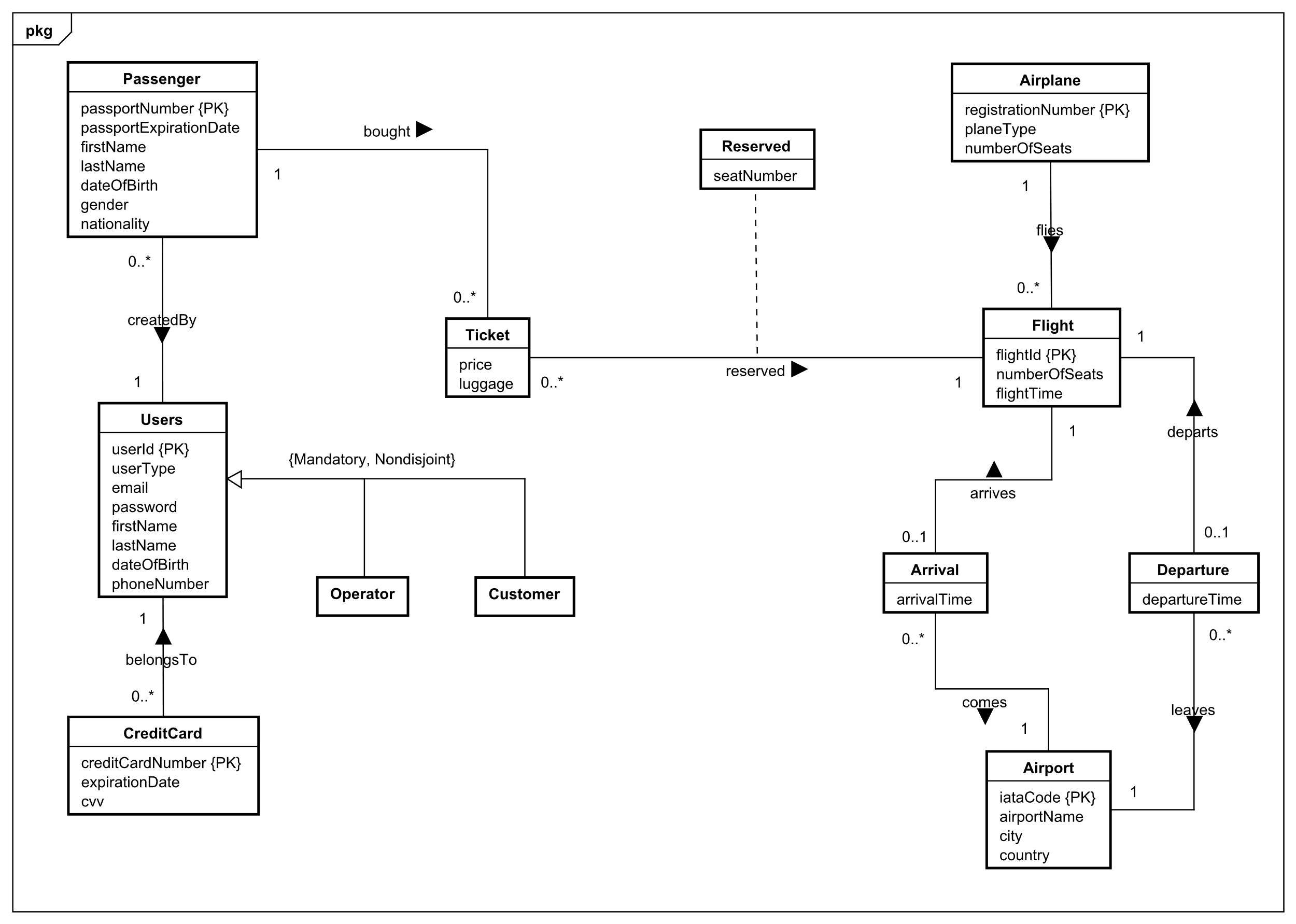


Diagram 6: EE/R diagram of the database

Entity/Relationship modelling models a business situation by describing the relevant entities and their relationships. For example, *Passenger* and *Ticket* are entities, and *boughtBy* describes the relationship between the entities.

This conceptual model is in third normal form. Normalization is a technique for producing a set of relations with desirable properties, given the data requirements of an enterprise (Connolly and Begg, 2015). It is in third normal form since it satisfies the first and second normal form, and no non-primary key attribute are transitively dependent on the primary key.

To better explain the conceptual model, it will be separated into three sections - Section 1, Section 2 and Section 3.

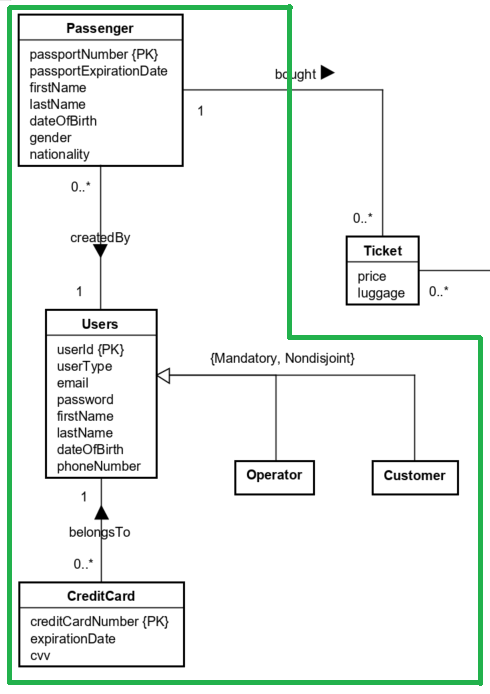


Diagram 7: Section 1 of the EE/R diagram

In Section 1, the strong entities present are the *CreditCard, Users* and *Passenger* entities. The *Users* entity holds the account information of users that use the system, while the *CreditCard* entity holds the credit card information of the users. The relationship between these two strong entities is called *belongsTo* and is a one-to-many relationship. The reasoning behind the one-to-many relationship is that one user can store multiple credit cards in the system and choose which card they prefer to use. The extended entity/relationship modelling is used between *Users*, *Customer* and *Operator*. *Users* acts as the superclass to the subclasses *Customer* and *Operator.* The participation constraint is mandatory and nondisjoint, which means it has a single relation with one or more discriminators to distinguish the type of each tuple (Connolly and Begg, 2015).

The other strong entity in this section is the *Passenger* entity. The *Passengers* entity has the information of passengers, with the *passportNumber* as the primary key The *Passenger* entity is created for the handling of creating multiple tickets by one customer. For example, if one customer decides to buy tickets for 3 people for one flight, each of the 3 people will be regarded as individual passengers, with their own *passportNumber,* names and so on. The relationship, *createdBy*, describes the one-to-many relationship, since multiple passengers can be created by a user, but a passenger can only be created by one user.

The Section 2 primarily deals with the handling of passengers who buy tickets.

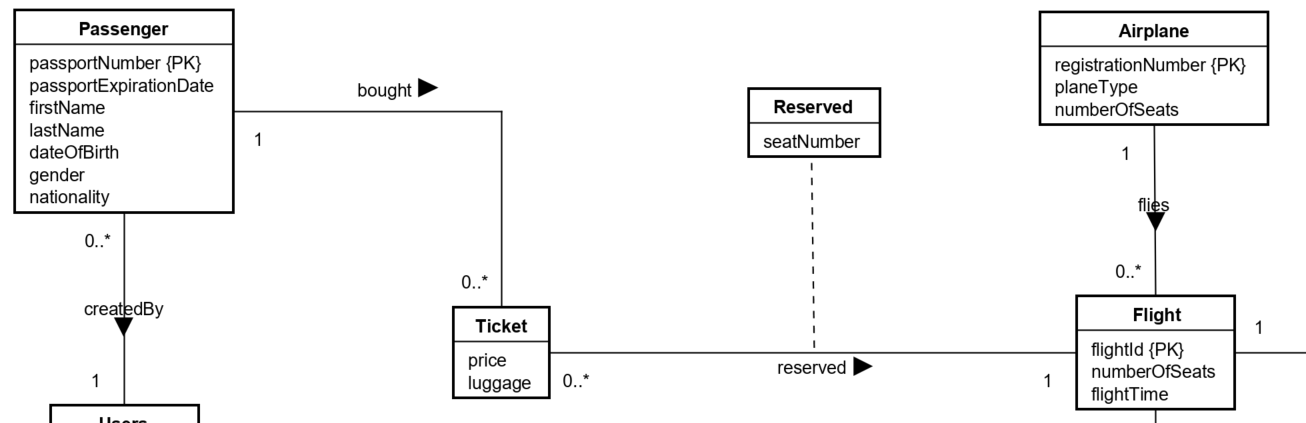


Diagram 8: Section 2 of the EE/R diagram

The new strong entities in this are *Airplane* and *Flight.* The *Airplane* entity describes the airplane, *Flight* describes the flight information. The relationship between *Airplane* and *Flight*, flies, is a one-to-many relationship because a flight can only have one airplane flying and an airplane can fly zero to many flights.

The *Ticket* is a weak entity. *Ticket* is a weak entity becausepassengers’ *passportNumber* can be stored alongside *flightId* from the *Flight* entity. The *passportNumber* and *flightId* will act as the primary keys, which then prevents passengers with the same *passportNumber* to buy more than one ticket for the same flight. *Ticket* and *Passenger* are related by the relationship *boughtBy*, which is a one-to-many relationship. It is a one-to-many relationship since a passenger can buy multiple tickets, but a ticket must belong to only one passenger.

How the reservation of seats is handled can be seen in this section as well. Each airplane has a fixed number of seats. The attribute, *numberOfSeats,* in the *Flight* entity shows how many seats are left that are unreserved in the airplane. The relationship, *reserved,* describes the reservation of seats. The attribute, *seatNumber,* indicates the seat number belonging to a particular flight reserved by a passenger. As seats are reserved, the attribute *numberOfSeats* in *Flight* should decrease.

In section 3 are 3 strong entities and 2 weak entities. Strong entities present are *Airplane*, *Flight* and *Airport* and weak entities are *Departure* and *Arrival*.

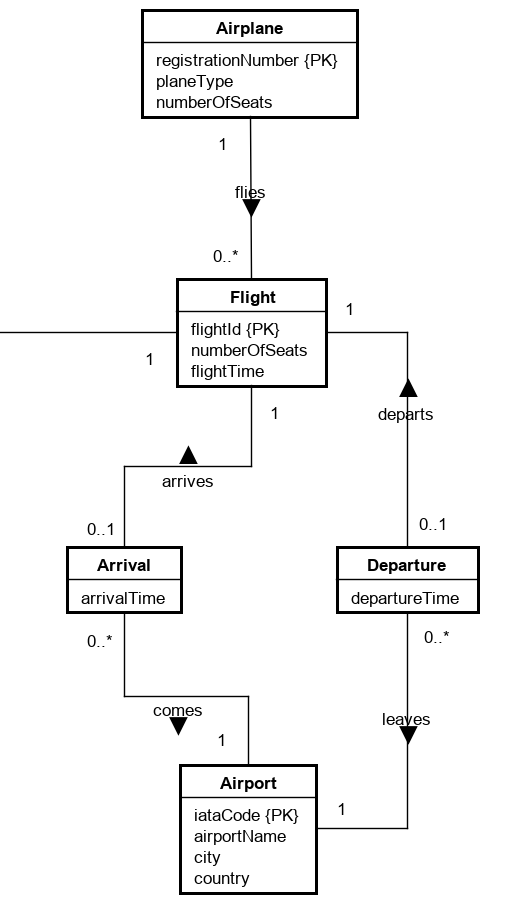


Diagram 9: Section 3 of the EE/R diagram

The *Flight* entity holds the information of flights that use the system, while *Airplane* holds information about airplanes. The relationship between these strong entities is called *flies*, and it is a one-to-many relationship. The reason, why it is a one-to-many relationship, is because *Flight* can store only one airplane (as the one flight will be done with one airplane) and *Airplane* can store multiple flights (as the airplane can be used for many flights). One-to-many relationship is also used between *Airport*-*Arrival* and *Airport*-*Departure.*

*Arrival* and *Departure* are weak entities, because there is no need for primary keys. There is a one-to-one relationship between strong entity *Flight* and weak entities *Arrival* and *Departure*. These one-to-one relationships are called *arrives* and *departs*. This relationship is used, because each flight can have only one departure and one arrival, and that Arrival/Departure can belong to only one *Flight*.

### Logical Model (Karrtiigehyen and Nicolas Popal)

The logical model will be used as the basis for the creation of the physical database. Deriving relations for logical data model was done by following the following steps (Connolly and Begg, 2015):

1. strong entity types
2. weak entity types
3. one-to-many (1:\*) binary relationship types
4. one-to-one (1:1) binary relationship types
5. one-to-one (1:1) recursive relationship types
6. superclass/subclass relationship types
7. many-to-many (\*:\*) binary relationship types
8. complex relationship types
9. multi-valued attributes

An example for each of the steps will be showcased, but since there were no one-to-one (1:1) recursive relationship types, many-to-many (\*:\*) binary relationship types, complex relationship types, and multi-valued attributes in the conceptual model, steps 5, 7, 8, and 9 are skipped.

Only one example for each step will be shown.

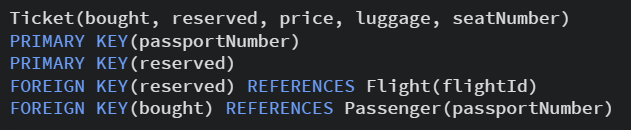
**Step 1:** Strong entity types.



An entity *Users* has attributes *userId, userType, email, password, firstName, lastName, dateOfBirth, phoneNumber,* with the *userId* being the primary key.

**Step 2:** Weak entity types AND **Step 3:** One-to-many (1:\*) binary relationship types

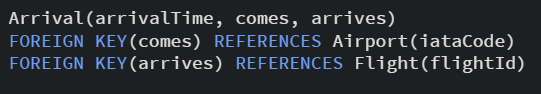
An interesting weak entity is the *Ticket.*



The *Ticket* entity only had the price. But since its relationships with the *Passenger* and *Flight* entities are one-to-many, the *Passenger* and *Flight* entities are designated as the parent entities (which is Step 3). This makes the *Ticket* the child entity. This means that the *fligthId* and *passportNumber* will be foreign keys in the *Ticket* entity. The relationship attribute, *seatNumber,* is also added as an attribute.

**Step 4:** One-to-one (1:1) binary relationship types

The relationship between *Flight* and *Arrival* is a one-to-one type.



The *arrivalTime* will still be an attribute, but the *flightId* will be added as a foreign key. The other foreign showed in the code above comes from the one-to-many relationship with *Airport.*

**Step 6:** Superclass/subclass relationship types



Since the constraint is mandatory and nondisjoint, just one table, *Users,* will be needed.

## Technologies Used (Karrtiigehyen and Nicolas Popal)

The programming languages used are *Java* and *C#.* This mixture of programming languages leads to the system being heterogeneous. *C#* is used in *Blazor*, alongside *HTML* and *CSS* for construction of the web application. Alternatively, JavaFX could have been used for constructing a GUI, but since building a heterogeneous system using *Java* and *C#* was a requirement, *Blazor* was chosen for the presentation tier. *Java* is used in the middleware (application tier) and in the persistence layer (data tier). In the middleware, *Java* is using the *Spring Boot* framework.

RPC

Sockets are used to establish a connection between the middleware and the persistence layer. The protocol used was TCP, since it is lossless and reliable, meaning if a segment is dropped, TCP will resend it. While TCP might not be as fast as UDP, a reliable data transfer is more important than better performance in this system. Alternatively, RMI could have been used instead of sockets, but sockets are chosen since they are part of the requirement.

As the requirement was that the system needs to use a database, *PostgreSQL* is used. *PostgreSQL* is an open-source object-relational database system (PostgreSQL, 2020). All the relevant data is stored and can be fetched in the database.

The persistence layer in the data tier and database must be connected as well. For this connection *PostgreSQL JDBC Driver API* is used. It is an open-source *JDBC* driver written in pure java (PostgreSQL JDBC, 2020). It is necessary to use this to establish a connection between database and persistence layer, so that the DAOs will function.

In the testing part, Mockito and JUnit 5 are used. Mockito framework is used because then system is not dependant on the database existence. Mockito can simulate database existence without having to access database or create data in there. JUnit is a testing tool used for testing class methods in order to check the functionality of the code. It is usually used in parts of the code that involve logic to verify code integrity.

The last used technology is the Maven repository. Maven is an automation tool used in Java to help with managing file structure, tools, and plugins. It was decided to use Maven during Server testing as one of our testing tools (Mockito) required many dependencies that were difficult to manage without a proper tool.