



SEJ Airline Reservation System

System Development

01.06.2016

Computer Science DAT15 – I

T h e • G r o u p • 5

Lei Xian 26.04.1984

Dana Iliescu 02.10.1996

Felix Leber 02.09.1988

Marius Ailisoaie 08.10.1995

Group Contract

1. The language to use among the group is officially English.
2. Sufficient and up to date communication requires everybody pays the best effort.
3. Everyone should participate the discussion of the project, to contribute on ideas.
4. Distribute works evenly among team members, in the meanwhile taking into each member's personal skill into consideration.
5. No absence without a good reason.
6. It is everyone's duty to read through the requirements of the project, and keep on track if our work fulfills the requirements.
7. It is everyone's duty to help each other to understand the project.

The image shows four handwritten signatures in blue ink on a light-colored background. From left to right, the signatures are: 'Lauri', 'Felix', 'Miguel', and 'Dario'. Each signature is written in a cursive, flowing style.

Table of Contents

Introduction (Felix).....	1
Aim (Felix)	1
Problem formulation (Felix)	1
Section I: ITO Report (all)	1
1. Background Analysis (Lei, Dana)	2
1.1 Company Description (Lei)	2
1.2 Mission (Dana)	2
1.3 Vision (Dana)	3
1.4 Values (Dana)	3
2. Organizational Structure of SEJ (Felix)	3
3. SWOT Analysis (Dana)	4
4. Stakeholder Analysis (Marius)	5
5. Feasibility Study (Lei, Dana)	6
5.1 Operational Feasibility (Lei)	6
5.2 Technical Feasibility (Lei)	7
5.3 Schedule Feasibility (Lei)	8
5.4 Legal and contractual feasibility (Dana)	8
5.5 Political feasibility (Dana)	8
5.6 Economic (Dana)	9
6. Risk Analysis (Felix)	10
Section II: OSCA (Lei, Felix, Dana)	12
Section III: Software Design (Lei, Marius)	15
1. A Brief Introduction (Lei)	15
2. Phase Overview (Lei)	15
2.1 The Inception Phase (Lei)	15
2.2 The Elaboration Phase (Lei)	16
2.3 The Construction Phase (Lei)	17
3. Iteration Description (Lei)	18
3.1 The 1 st iteration - Elaboration Phase (Lei)	18
3.2 The 2 nd iteration – Elaboration Phase (Lei)	22
3.3 The 3 rd iteration to 5 th iteration – Elaboration Phase (Lei)	26

3.4 The 6 th iteration - Construction Phase (Lei)	26
3.5 The 7 th iteration - Construction Phase (Lei)	27
4.Artifacts (all)	27
4.1 A brief introduction (Lei)	27
4.2 Use Cases (all)	27
4.3 Use Case Diagram (Felix)	31
4.4 Domain Model (Marius, Lei)	32
4.5 Class Diagram (Felix, Marius, Lei)	33
4.6 Sequence Diagram: Create Customer (Dana)	34
4.7 System Sequence Diagram (Lei, Dana, Marius)	35
4.8 State Machine: Ticket (Felix)	35
5.Conclusion of following UP (Lei)	36
6.Three Layered Architecture (Lei)	37
7. Git Experience (Lei)	37
8. GRASP (Marius)	38
Section IV: Software Construction (Dana, Felix, Lei)	39
1. Database and EER Diagram (Dana)	39
2. Scope of the Application (Dana)	41
3. A thorough description of the construction of the application and decisions (Dana)	41
3.1 Add Plane & Add Schedule	43
3.2 Search Flight	44
3.3 Search Order	44
3.4 Confirm ticket	45
3.5 Cancel & Refund	45
3.6 Book a ticket	47
3.7 Other features	48
3.8 How can we improve it?	49
4.Exciting Snippet of Code – Handling Seats (Dana)	49
5. What works and what doesn't in our system (Felix)	50
5.1 What works:	50
5.2 What doesn't work:	51
6. Construction of the GUI (Felix)	52
7. SEJ System User Manuel (Felix)	54

7.1 Logging In	54
7.2 Admin section	55
7.3 Adding a new flight schedule	55
7.4 Adding a plane	56
7.5 Customer Services Section	57
7.6 Booking a ticket.....	57
7.7 Searching for an order	58
7.8 Confirming a Booked ticket.....	58
7.9 Cancelling a Booked ticket	58
7.10 Refunding a Confirmed ticket	59
Conclusion (Felix)	60
Bibliography	61

Introduction

Aim

The aim of this project is to build a flight system based on certain requirements.

Problem formulation

The system that we need to build has to be able to handle both administrative and customer service related duties. The administrative duties should include adding planes to the system and scheduling new flights. The customer service part should be able to search for flights and orders, while also being able to handle all the ticket functionality such as canceling, refunding, booking and confirming. All this should be done using Unified Process. The application will also have an option for exporting two tables from the database into CSV files.

Section I: ITO Report

In this section of report, we are mainly focused on answering the following questions:

What is the vision and business case for this project?

Is it feasible?

We made up an airline company named SEJ (Scandinavian Easy Jet) which is a fast growing business, and therefore it has the need to develop a good system that can help the company to handle the requirements. The company has its own IT department, which we imagine we are part of, and have 4 weeks to develop a system based on the old version of the system for the company. Therefore, a lot of setting from here is actually real for our situation.

In this phase we have done the following studies:

1. Background Analysis

1.1 Company Description

1.2 Mission

1.3 Vision

1.4 Values

2. Organizational Structure

3. SWOT Analysis
4. Stakeholder Analysis
5. Feasibility Study
6. Risk Analysis

1. Background Analysis

1.1 Company Description

The Scandinavian Easy Jet (SEJ) airline company is a new, but extremely fast developing company.

Mikkel Jensen and Hans Jørgensen, two friends that have been working for different airline companies for more than 10 years, founded it. They decided to start their own company in 2014 and aim to combine their experience with the desire to offer top quality service to their customers.

The company is rather small, based on its number of employees and fleet of planes. SEJ wants to help all their customers have a good flight experience, with their needs fulfilled. It mainly focuses on two types of services.

Firstly, it offers increased comfort during the trip and a “careless” flight, taking care of everything all the way, from when the customer has bought the ticket until they arrive at their destination. Families with children, travelers and all those who need to go from point A to point B by air will experience a safe, fast and decent-priced flight.

The second type of service focuses on business people. SEJ found the best-developed cities and the cities where there is a rapid growing economy and has flight routes directly to them.

The company understands that business people don’t want to waste much of their time, especially when the trip is long, so they might want to get some of the work done on the way, and they would like to have space to do it in. Thus, the company decided to mainly use business jets for this part of the service, to be able to offer more space for each passenger. This will allow the passengers to feel more comfortable, whether they would like to work or just to relax. The on board service is also much more personal and precise. This line of service also offers the passengers a lobby where they can meet other business elites, which might bring them new opportunities.

SEJ believes in quality over quantity, therefore the company wants to automate processes as much as possible. Therefore, SEJ airlines wants a new airline reservation system, which can handle their business orders faster and more accurate, helps the staff work more efficiently and find out the needs of the customers. This will help the customers have a better trip and service experience. The company hopes the system will not only improve efficiency, but also reduce unnecessary costs on human resource.

1.2 Mission

SEJ’s mission is to help people by: offering them a nice trip to their dream holiday destination or simply by making their businesses flourish.

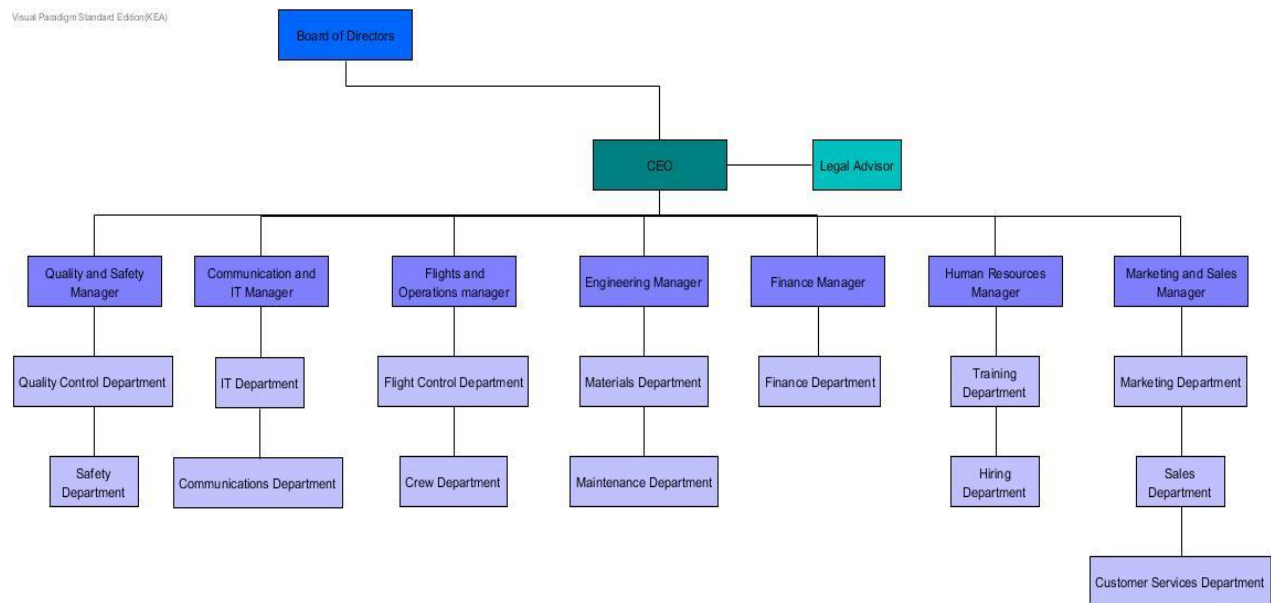
1.3 Vision

We want to be our target group passenger's first choice.

1.4 Values

- Quality: We offer nothing but the best.
- Trust: We deliver what we promise.
- Customer-focused: The customer's experience is what matters to us.
- Innovation: We always try to find new ways to increase customer's comfort.
- Passion: Happy customers means a happy company.

2. Organizational Structure of SEJ



Appendix 1: Organizational Structure Suggestion (page 61)

Board of Directors: the highest in the power structure of the company, the owner of the company. Able to make all important decisions.

CEO: the highest power of the management structure, listens to the board's directions, leading the company's development.

Quality and Safety Department: takes care of the administrative part of things relating to quality and safety. For example, to give the cabin crew on-board safety instructions and to train them so they have the necessary knowledge to cope with potential dangerous happenings/situations during a flight.

Communication and IT Department: in charge of communication among departments, and the company's IT system's performance.

Flights and Operation Department: the responsibility is to take care of everything to do with scheduling flights, flights, arrange things in the airports and on board.

Engineering Department: ensures that all components of the plane are in proper working order. It is their duty to make any repairs if a mechanical issue does arise. Works closely with the Quality and Safety Department.

Finance Department: it is the department's duty to take care of everything related to finances, budgets and so on.

Human Resource Department: it is the department's duty is to find out the company's human resource situation and to fire or to hire employees.

Sales and Marketing Department: the department's duty is to build the company's reputation and image in the market, make advertisements and so on.

3. SWOT Analysis

Strengths

- What differs us from our competitors is that SEJ's founders have years of experience in the airline industry. They have experienced all types of flights, from the lowest class to business class, on all continents, and from many travel agencies. That means they understand the customer and know how to cope with different problems that may occur.
- The airline staff are highly trained, in all our departments. We take staff selection very seriously, so each member is put through a test before becoming part of our team. We offer monthly training to our cabin crew, therefore we are always well-prepared and up-to-date with every change in the airline market.
- We offer special services to business people, using the latest types of jets and making their trip work-friendly, so that they can work and travel in comfort.
- Our technology approach, a strong robust IT system makes it easy for customers to book tickets, and make the company communication efficient and good.
- Safety, speed and comfort is our focus, as suggested by our clean record.

Weaknesses

- Although SEJ offers quality service, the company is still new to the market and not so well-known (in comparison to the bigger competitors).
- Costs for aircraft are high, therefore the company needs huge capital.
- Our business elite focusing service is a new concept, can be difficult to convince the customers.

Opportunities

- As the company expands, we expect that we will be able to enter the more markets all over the world.

- New technology makes it possible to offer faster flights and better service to customers, as well as cost reductions (for example more automated processes on the ground).
- With credibility of the company increasing daily, advertising and many satisfied customers, more clients will be attracted and more capital generated.

Threats

- Bigger competitors from inside and outside Denmark.
- A potential global economic downturn.
- IT system failure.
- Increases in price of fuel.
- Any terrorist attack that might happen in the countries that we operate in.

4. Stakeholder Analysis

Stakeholder	Importance	Influence & power	Interests	Concerns / Negative Impacts
Board of Directors	High Importance	High power, able to decide what project to t or to stop	To have good projects leading to a well growing company economy, earning more and invest more	Not satisfied with the project, decide to cut the project
Customers	High Importance	Low power, they don't influence the project	To have quick and reliable service	Not satisfied with speed and quality of the service
Customer Service Department	Medium Importance	Low power, they use the system but don't influence the requirements	To have an good looking, not boring and easy to use everyday program	The interface is boring, unprofessional and hard to use everyday
Shareholders	Medium Importance	Low power,not able to decide the requirements for the system	Have a prosperous developing company	Financially unsatisfied because the poor quality of the final product
Project team	High Importance	Medium power, they can make suggestions but they have to follow the	Good communication during the process of making the project, making a satisfying	Due to lack of communication starts losing the motivation

		requirements and make a good quality product	product according to the requirements	
Flights and operations manager	High Importance	Medium power, they will only use the system	The system to be reliable and efficient	The system is slow, not reliable, the interface is hard/boring to use everyday
Communication and IT manager	High Importance	High power, able to influence the project	The system runs well, has a good quality, budget and schedule are kept/respected	Budget and schedule are not kept, final result not satisfying

5. Feasibility Study

5.1 Operational Feasibility

SEJ is a new airline company whose business' develops rapidly. It has an increasing need of developing a new system that can handle and ease their daily tasks.

The reason SEJ has gained such quick success, although it has been started just a couple of years ago, is because it has given its customers very good travel experiences with their quick, agile service. Its service differs from the traditional one, which can be redundant and slow. Instead, SEJ has created a way to provide customers with fast, reasonable, and moderate service. For instance, it offers fast refunding in case a customer wants to cancel his flight.

SEJ is also a pioneer in bringing its elite service to a more affordable level for business people. By using a new type of business jet, and by reducing unnecessary services and costs, it is able to offer those business elites an air trip where they can still focus on their work or get a decent rest before their intensive work starts again at the destination.

Since SEJ is getting bigger, its current system is not good enough to handle all facets of this kind of business, for example to handle the information about its fleet, or the different types of refunds to customers. SEJ does not want to employ many people to do the workload, as this might result in human mistakes, so it prefers to use an automatic system instead. This is more in line with their principle: reduce unnecessary employment, more efficient working methods and a good working environment. As a new airline company, it benefits from new technology, and SEJ believes technology brings a better tomorrow.

As a conclusion, the airline reservation system will support the business strategy of the company. Because it also gives a quicker way to work with their daily business orders and it can help SEJ staff to identify their customers' needs. The system is consistent with SEJ's IS strategy.

The reservation system can help the staff work much more efficiently, and the new refunding function of the system is agile and quick, so the customers can get a fast response if a cancellation is made, and get their refund in a much faster and easier way. This gives a feeling of security to the customers, and all customers like to be secure in as many ways as they can. The system will provide a better, faster service, which at the same cost would give SEJ a good chance in the market and prove the fact that they are a trust-worthy company.

Furthermore, SEJ wants to avoid a complicated and redundant company structure, as an airline company could easily get into that. A good system can help keep SEJ's structure simple. The money spared from maintaining a big structure can be used on developing their core business parts and focus on accomplishing their mission.

5.2 Technical Feasibility

Factors:

- **Project size:** medium
- **Number of members on the project team:** 4
- **Project duration time:** 4 weeks
- **Project complexity:** high
- **Developers' familiarity with the type of application and the technology:** medium
- **Number of organizational departments involved:** IT department, HR department, Management group, customer service department (almost all departments are involved)
- **Users' familiarity with the system development process and the domain area:** users will be SEJ staff. As they feel that the existing system is not good enough to handle the daily growing business, there is a need from the user's side for a better system. So the users are willing to participate in building a better system. Management is also willing to invest and make the best effort for the system to be done in a right way.
- **Project structure:** renew the existing system, covering more aspects of the company. Since the company is growing fast, and it has always been very careful with its simple clean structure, the release of the new system will not have a huge impact on the company's structure.
- **Development Group:** SEJ's own IT department is experienced and know their work and tools well. Even though they have developed a similar project earlier and they have the experience and the skills, the schedule is very tight, and this can result in an undesired software product. This would be a risk factor.

In conclusion, the project is medium sized, the IT department is experienced, the technologists are available and both users and management are willing to commit to the development of the system as they know it is going to be beneficial. The users know the interfaces from the earlier system, the improvement of the system is not going to change that a lot, so they will not have much trouble using the new system. Therefore, the overall risk of the project is low.

5.3 Schedule Feasibility

Project timeframe: 4 weeks

	Elaboration			Construction			
Phases	Iteration1	Iteration2	Iteration3	Iteration4	Iteration5	Iteration6	Iteration7
Goal	Finish the critical functions			Rest of the function			
Likelihood*	7	7	7	8	9	8	8

*likelihood: ranges from 0 (impossible) to 10 (no problem of finishing the work).

It is a tight schedule if the system has to be put in use already after 4 weeks. In any case, the worst scenario is the work will not be done by the deadline and this will influence the whole company.

A solution to this can be to keep on evaluating the work process and working with the users to get prompt feedback. If there is any sign showing that it is not possible to finish the project by the deadline, then SEJ should consider hiring a reasonable number of programmers from other IT companies to help complete the system. Luckily, there are several programmers we know from the startup phase of the company which we can count on. So even though the schedule is tight, it is possible.

5.4 Legal and contractual feasibility

As part of the European Union, Denmark must follow some rules specific to this union. SEJ operates under the regulations and restrictions of the EU concerning air travel, which means that they focus on passenger's safety, international trade, tax policy and environment protection.

5.5 Political feasibility

There is already an increasing need within the company that calls for a better, easier, faster system which can cope with the daily fast growing business, and the project is supported by the board of the company, the overall reaction of developing a new system among the employees is positive. The system's interface design is consistent of the old version; thus the employees won't feel they have to change a lot to adapted into the new system. The using manual is simple, concise and easy to use.

As conclusion, the release of the new system will not cause human resource changes, and is generally welcomed by the employees, so it is doable.

5.6 Economic

Firstly, Denmark's growing economy suggests that there are a lot of business people that need to travel all over the world in order to run their businesses. Secondly, statistics show that 12,5% of people's salaries are set aside to holidays. This means that all of our target groups have access to funds in order to travel, whether it is for business purposes or to relax.

SEJ's new system will provide benefits to the company, as shown in the following economic feasibility study that we conducted.

Table 1 – Costs and Benefits for SEJ's new system

Costs		Benefits	
Tangible	Intangible	Tangible	Intangible
Salary	Lower motivation	Error reductions	Higher motivation
Hardware		Increased productivity	More efficiency
Operating Costs		Increased sales	Satisfied customers
Training			
Maintenance			
Consultants			

*numbers are only an example, not authentic

The economic feasibility study is based on the following set of information.

- Estimated total one-time cost is 270,000 (development staff estimate is 6 man-months, average cost for 1 development man-month is 40,000 kr and hardware for the system cost is 30,000 kr, which means $6 \cdot 40,000 + 30,000 = 270,000$ kr).
- The lifetime of the system is expected to be 3 years.
- Maintenance costs are estimated to 10,000 kr/year.
- The discount rate is 4%.
- The new system will reduce man-made errors and costs, and will be faster than the previous one.

Table 2 displays a positive value of the Overall Net Present Value and a value of Return of Investment of 1.01, which means the project is doable and the company will benefit from implementing it. After 3 years, the total costs will be covered and there will be a profit of 302,230 kr.

Table 2 – Costs and Benefits analysis

	year 0	year 1	year 2	year 3	TOTALS
Benefits					
Net benefits		115,000	120,000	122,000	357,000
PV of benefits		110,576	110,946	108,457	329,979
NVP of all benefits		110,576	221,522	329,979	674,723
Costs					
One-time costs	270,000				
Recurring costs		10,000	10,000	10,000	30,000
PV of costs		9,615	9,245	8,889	27,749
NPV of all costs		9,615	18,860	37,364	297,749
Overall NPV					302,230
ROI					1.01

6. Risk Analysis

Ris k ID	Risk	Likeliho od	Impac t	Priorit y	Mitigation (how to avoid)	Monitor	Management
1	Code the flight system in a way which does not match exactly the requirements .	40	80	60	Verify before coding that it fits with the requirements. Code the system in parts, so that only a small part has to be redone if needed.	The system doesn't have the functionality as per the requirements .	Change the system so that it fits the requirements .
2	Fall behind schedule due to needing more time on certain parts.	60	80	70	Follow the plan closely, if needed put a second person on it. Update the schedule as needed to ensure project gets	Project is behind schedule.	Divide workload better so that the project gets back on schedule.

					completed in time.		
3	System does not work properly, has bugs.	40	90	55	Use Unified Process, lots of testing, code in small parts so it's easier to fix bugs.	System does not run as expected	Test system part by part to find and fix bug.
4	Lack of knowledge to make a part of the system.	40	100	70	Good communication between the group members to share their knowledge as needed.	Project is stuck and not getting anywhere.	Work together to figure out a way to code that part of the system.
5	Team members work on the same part, wasting time.	20	80	50	Hold regular meetings to get updated on what is done and what each member will work on next.	Having a part of the same system twice.	Re-assign one of the members to another task.
6	Team member gets sick	20	100	60	The members should be aware of the importance of their work and therefore they should care of their health as much as possible.	Team member is sick and unable to do his part.	Share the sick member's tasks between the other members.

Conclusion:

The project is doable after all the analyzes from all different aspects. It is supported by the company, it fits the current needs of the development of the company, and the company have the people and capital to do it.

Section II: OSCA

Employee Class/Table					
Fields	MySQL Data Type	Java Data Type	Possible Range	Theoretical amount of bits	Consideration
employee_id	int	Int 32 bits (2^{31} to $2^{31} - 1$)	1 – 200,000	18 bits 200,000 = 110000110101000000	Lufthansa is world's largest airline by employee, it has 120,262 employees according to their 2015 report. The number of SEJ's employees probably will also stay in this range
employee_first_name	Varchar (50)	String			
employee_last_name	Varchar (50)	String			
employee_title	Varchar (50)	String			
employee_username	Varchar (50)	String			
employee_password	Varchar (50)	String			

Plane Class / Table					
Fields	MySQL Data Type	Possible Range	Theoretical amount of bits	Java Data Type	Consideration
plane_id	Int	1 - 2000	11 bits 2000 = 11111010000	Short 16 bits (-32768 to 32767)	The world's biggest airline's fleet has 1,494 planes
plane_type	varchar(50)			String	
first_class_seats	int	1 - 20	5 bits 20 = 10100	Byte 8 bits (-128 to 127)	According to wiki, the largest air bus A380-800 with 3 classes has normally 14 first class seats*
business_class_seats	int	1 - 100	7 bits 100 = 1100100	Byte 8 bits (-128 to 127)	According to wiki, the largest air bus A380-800 with 3 classes has 56 business class seats*
economy_class_seats	int	1 – 900	10 bits 900 = 1110000100	Short 16 bits (-32768 to 32767)	According to wiki, the largest air bus A380-800 can carry up to 853 passengers*

* https://en.wikipedia.org/wiki/Airbus_A380#Overview

*Extra information about seats:

http://www.seatguru.com/airlines/Emirates_Airlines/Emirates_Airlines_Airbus_A380.php

Documentation for the Fixed Binary Format File

In the future, the application will have an option to export the tables to binary fixed record size format. The code below is an example of how this could be implemented.

```

import java.util.*;
import java.io.*;

public class CreateFixedBinaryRecord
{
    private static byte[] createRecord(short plane_Id,
                                       String plane_type,
                                       byte first_class_seats,
                                       byte business_class_seats,
                                       short economny_class_seat)
    {
        byte[] rawRecord = new byte[2 + 1 + 50*2 + 1 + 1 + 2];
        rawRecord[0] = (byte) (plane_Id >> 8);
        rawRecord[1] = (byte) (plane_Id & 0xFF);
        rawRecord[2] = (byte) plane_type.length();
        int j = 3;
        for (int i = 0; i < plane_type.length(); i++)
        {
            char ch = plane_type.charAt(i);

            rawRecord[j] = (byte) (ch >> 8);
            j++;

            rawRecord[j] = (byte) (ch & 0xFF);
            j++;
        }

        j = 3 + 50 * 2;
        rawRecord[j] = first_class_seats;
        j++;
        rawRecord[j] = business_class_seats;
        j++;
        rawRecord[j] = (byte) (economny_class_seat >> 8);
        j++;
        rawRecord[j] = (byte) (economny_class_seat & 0xFF);
        j++;
        return rawRecord;
    }

    public static void main(String[] args) throws Exception
    {
        List <Plane> allPlanes = PlaneInfo.selectAllPlanes();
        File sej = new File("sej.txt");
        PrintStream output = new PrintStream(sej);
        byte[] bytes;
        for (int i = 0; i < allPlanes.size(); i++)
        {
            Plane = allPlanes.get(i);
            bytes = createRecord ((short) plane.getPlaneID(),
                                plane.getPlaneType(), (byte) plane.getFirstClassSeatTotal(),
                                                         (byte) plane.getBusinessSeatTotal(), (byte)
                                plane.getCoachSeatTotal());
            output.write(bytes);
        }
    }
}

```

Section III: Software Design

The report consists of:

1. A brief introduction
2. Phase overview (Inception, Elaboration, Construction)
3. Iterations descriptions
4. All Artifacts
*code is covered in the Software construction report
5. Conclusion of following UP
6. 3 layered architecture
7. GIT experience
8. GRASP
9. Appendix: artifacts from each iteration

1. A Brief Introduction

The Unified Process (UP) has 4 phases:

- Inception
- Elaboration
- Construction
- Transition

We have mainly gone through the first 3 phases. The transition phase is not included as this is a school project, so we did not do the deployment.

We have followed Unified Process, from finding requirements by writing the use cases, to using diagrams to help to do the analysis and the business modelling, and always making a new iteration plan when one iteration is completed.

A thorough description of each iteration can be found later in this section.

Because each artifact has a lot of different versions as we kept on developing them in each iteration, it would be unrealistic to include all artifacts generated in each iteration in the iteration description section. Therefore, we have put all of them in the Appendix, and have only chosen a few to use in the iteration description to demonstrate the evolution.

2. Phase Overview

2.1 The Inception Phase

Period: 6 – 10 May

This is a short phase to answer the question from both business and technical perspectives: is the project doable?

Part of the study was done from the business perspective, which is covered in *Section I: ITO* above.

Besides what is covered in the ITO report, other artifacts which are made in this phase are: glossary, vision*, and a brief start of use cases – meaning to figure out the use cases in brief format.

**Vision is included in the ITO report (see page 3)*

Appendix 2: Inception Phase Artifacts (page 63)

2.2 The Elaboration Phase

Period: 11 – 24 May

This phase has 5 iterations. The table (*Figure SWD - 1*) below can give an overview of the 5 iterations we went through, and it will also show what artifacts are generated in which iteration.

The goal of this phase is to figure out the most important use cases, capture the main system requirements, and to tackle the most essential part, or the most problematic part of the construction. We expected to have the core, most significant components done by the end of this phase, and the system architecture should also be stabilized by then.

In the first 2 iterations, we focused mainly on finding the requirements and analyzing and designing the objects. We have also constructed some of the basic necessary components, such as the data types and the classes that provide access to the database. This is to support or to test our understanding of the requirements in practice. But at this moment, the construction is not the biggest focus.

From the 3rd iteration on to the construction phase, we have focused more on the construction. The work we did in the first 2 iterations were useful, and the significant risk factors were found, so now it became more a matter of tackling these. The flight search, book/buy tickets and search order functionality were the main focus from the 3rd until the 5th iteration.

Figure SWD - 1: An overview of the Elaboration Phase with 5 iterations

Elaboration Phase		
Iteration	Period	Main Artifacts*
1	3 days (11 – 13 May)	<ul style="list-style-type: none"> - Some use cases in brief format - Use case diagram - Domain model draft - Draft sequence diagram for “book ticket” - Draft system sequence diagram for “book ticket” - Database draft - The next iteration plan
2	4 days (14 – 17 May)	<ul style="list-style-type: none"> - More use cases in brief, casual - Construction of the application <ul style="list-style-type: none"> ▪ database built ▪ data access layer built, ▪ application layer (data type built) - Updated domain model, SD, SSD, CD - State machine - The next iteration plan
3	4 days (17 – 20 May)	<ul style="list-style-type: none"> - Updated use cases, developed the “book ticket” use case in fully dressed format - Updated diagrams - Construction on all 3 layers - The next iteration plan
4	2 days (21 – 22 May)	<ul style="list-style-type: none"> - More construction on all 3 layers - Updated use cases - Updated domain model, CD, other diagrams
5	3 days (23- 25 May)	<ul style="list-style-type: none"> - More construction on all 3 layers - Final use cases - Updated domain model, CD, other diagrams

*code is not included, because it is covered in the report for Software Construction

2.3 The Construction Phase

Period: 26 – 31 May

In the last phase, we have finished the most significant features of the application. All functionality works. The Elaboration phase had built the foundation for the system, so the goal of the Construction phase was to build the rest of the system. This meant we needed to put all the artifacts together, update

the diagrams, and to construct the final missing features in the system. In other words, the application should be complete and have the functionalities as described in the use cases or the diagrams.

The table (*Figure SWD - 2*) below can give an overview of the iterations and the artifacts generated in each iteration.

It is also in this phase where we test our code, meaning we have integrated our code from different group members, run the application and input data to see if things happen in the way we want. We have fixed a few bugs after the tests.

Figure SWD – 2: An overview of the Construction Phase with 2 iterations

Construction Phase		
Iteration	Period	Main Artifacts
6	3 days (26 – 28 May)	<ul style="list-style-type: none">- Integrated code from all team members- CSS- Updated sequence diagram- User Guide draft for the application
7	3 days (29 – 31 May)	<ul style="list-style-type: none">- Final versions of all artifacts, includes implementation

3. Iteration Description

3.1 The 1st iteration - Elaboration Phase

Period: 11- 13 May

Original Plan:

1. Write use cases in brief format, only the use case “Book Ticket” in casual format
2. Domain model (*Figure SWD - 3*)
3. Class diagram (*Figure SWD - 4*)
4. Sequence diagram for “Book Ticket”
5. System sequence diagram for “Book Ticket”
6. Start to make the database structure
7. Construct some basic classes
8. State machine for “ticket”
9. Iteration plan for the next iteration

Actual Process:

We started with finding some use cases and writing them in brief format. It resulted in 7 – 8 use cases, which we got from the project requirements, and wrote a description in brief format for those use cases. From the use cases, we made a use case diagram to demonstrate a user's interaction with the system, and by using methods such as finding nouns and verbs, a draft of a domain model was made. By using a sequence diagram and state machine, we figured out some possible methods and to which objects they could belong. We discovered that it is very easy to fall into the "water-fall" method, as we were trying to write all use cases and defining all requirements at this step already. However, we soon learnt that there is no need to do that. We tried to keep it in mind and reminded each other about it.

We discussed the use cases and domain model, which helped us understand the assignment. Because to design an airline reservation system is a huge topic, as there are so many things and aspects which can be involved, that the attempt of finding out what information we need in order to build such a system can get overwhelming very quickly. For example, when we were discussing the relationship between the objects in our domain model, we thought that of course a plane flies between 2 cities, thus there could be a plane class, and a schedule class has planes and cities as fields, but wait... a city can have more than one airport, and a schedule can have more than 1 plane, or have 1 plane with more than 1 leg.... This is just an example of how easy it is to get lost in the complication.

From the use cases, we have 3 use cases (*UC#1: Add Plane, UC#2: Schedule Flight, UC#3: Book Ticket*) which we thought the requirements were quite clear. The system's user will be the employee, and it should be able to handle 2 types of requirements: the requirements from the administration such as to add a plane and to schedule a plane, and the requirements from the customer service department such as to book tickets for a customer. We chose to write the use case "Book Ticket" in casual format, to analyze more requirements, as the casual form has more steps or actions which we should describe. At this moment, there were 4 objects in our domain model: passenger, plane, leg and ticket. Each of them have some attributes gotten from the nouns from the use cases. Later we have added employee to the domain model as well.

We decided to transfer those domain model objects to a basic database, each object corresponding to a table in the database, then we could construct part of the code to play with. Making the sequence diagram and system sequence diagram helped us to capture possible methods. Those methods didn't show in our class diagram yet, as what we constructed were mainly the classes to access data in the database, and the data types in the application layer. In addition, we made the second iteration plan when we completed the to-do list at the end of this iteration.

Appendix 3: Artifacts – Iteration 1 - Elaboration Phase (see page 65)

Artifacts examples:

UC # 3: Book Ticket - casual

Primary Actor: Customer Service Department Staff

Main Success Scenario:

1. The staff opens the system.
2. The system asks for the staff's user name, and password.

3. The system verifies the identity of the staff, if it is valid, the staff can log in and use the application.
4. The operation menu appears, and the user chooses the operation "Book Ticket".
5. The system asks for the information of the passenger (passengers).
 - 1) Situation A: The staff knows exactly what information to type in:
 - Type in the required information of the flight:
Airplane number, departure airport, destination airport, departure date time, arrival date time, class, seat number
 - Type in the required information of the customer:
First name, Last name, Gender, Birthday, Nationality, Passport number, Email, Phone number
 - 2) Situation B: the staff searches the flight after the customer's requirements
 - The staff uses the search tool of the system, fills the search conditions
 - The system shows the flights which satisfies the search condition
 - The staff chooses among the flights, clicks the desired one to book it
 - The system asks for the information of the customer after the flight is selected
 - The staff types in the customer's information and confirms
 - A ticket is booked.
6. The system shows message "A ticket is booked; the notification is sent to the customer".
7. Customer's information goes to the database, the ticket's information goes to the database, airplane's seat information is updated accordingly.

Alternative Flow:

1. At any time, if the system fails:
 - The system shows an error message, and restarts
 - Ask for the valid log in, restarts again
2. No available tickets left, no reservation can be done

Figure SWD – 3: Domain Model from the 1st iteration

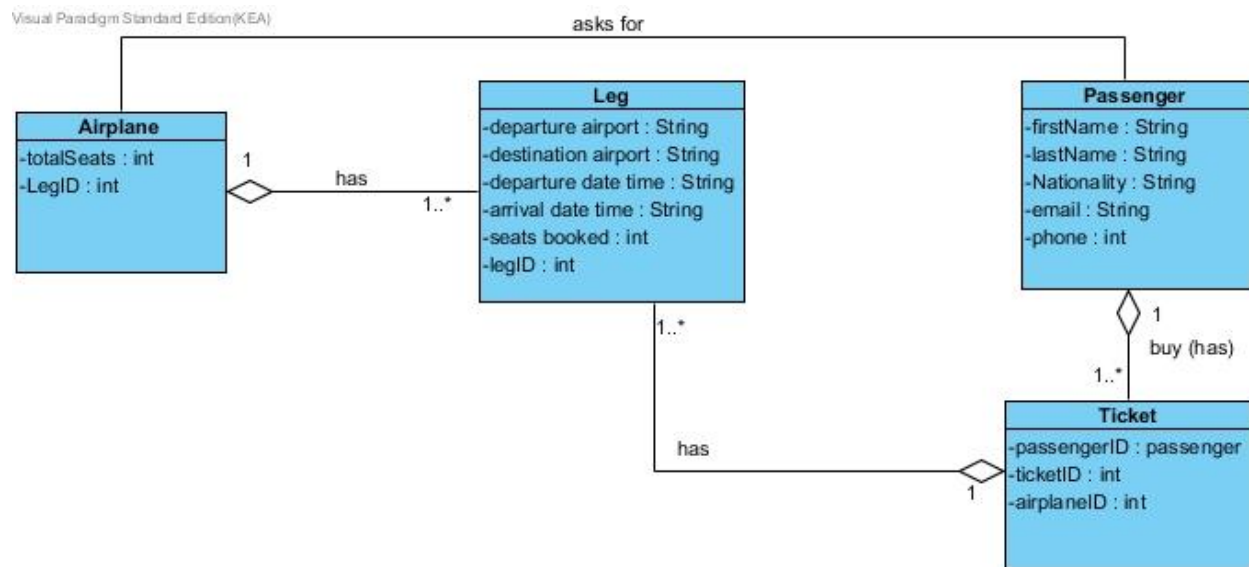
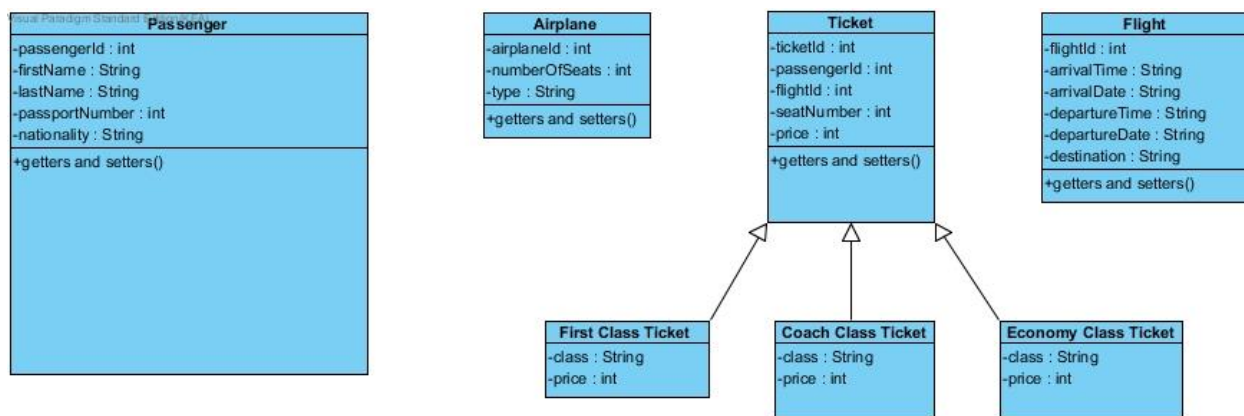


Figure SWD – 4: Class Diagram from the 1st iteration



3.2 The 2nd iteration – Elaboration Phase

Period: 14 - 17 May

Original Plan:

1. Write 2 use cases in casual format mostly, use case “Book Ticket” in fully dressed
2. Update domain model (*Figure SWD - 5*)
3. Updated class diagram (*Figure SWD – 6, Figure SWD – 7*)
4. Update SSD
5. Input data to database
6. Continue to construct the data access layer
7. Continue to construct the application layer (the controller classes as well)
8. Presentation layer: UI construction for “Add plane” scene
9. Iteration plan for the next iteration

Actual Process:

In this iteration, what we wanted to achieve was to implement our design, such as the data access classes, the data type classes, and the classes that had the role as a controller. We needed those classes to do the implementation for the use cases we have. We also wanted to have a better domain model after this iteration.

We already have the basic database structure, and now we have input some data into the tables. The input to the database has led us to a few more questions of our design, such as if we should distinguish between passenger and customer. In reality, there must be a difference between a customer and a passenger – of course, the one who pays for the ticket will be a customer, but if the ticket is not for himself, then he will not be a passenger. But considering the time we have to finish the project, we have decided we will simplify some parts in order to reach a better performance, therefore in our system, we have chosen not to make a difference between a customer and a passenger. The sequence diagram of “Book Ticket” helped us to detect a new object in the domain model, which is “order”.

After constructing partially the classes in the data access layer, and the application layer, they have exposed more questions for us to reconsider, for instance the relationship between ticket and customer, plane and leg, thus we looked at our domain model and added some new objects we thought were necessary. The class diagram was updated accordingly.

Since the use case “Add Plane” was seemingly stable, meaning there was not much to change, we started to make the UI design for this use case.

Appendix 4: Artifacts – Iteration 2 - Elaboration Phase (see page 71)

Artifacts example:

UC # 3: Book a ticket- Fully Dressed

Scope: Book Ticket System

Level: User goal

Primary Actor: Customer Service Department Employee

Stakeholders and interests:

- Employee: wants to have an easy to use system when he wants to book a ticket.
- Customer: wants to book a ticket in a simple way.

Preconditions: The system is opened and ready to process

Postconditions (Success Guarantee):

- A ticket has been booked.
- A record for the ticket is created by the system, so the customers can be informed about the ticket details later.

Main Success Scenario:

1. The employee opens the system and logs in with username and password.
2. The employee gets into the operation menu, chooses the operation "Book Ticket".
3. The staff searches flight after the customer's requirements: departure date, departure place, arrival date, arrival place, ticket price (range from high to low or the opposite) and/or ticket type (class).
4. The system shows the flights that satisfy the search condition(s).
5. The customer tells the staff which flight meets best his requirements and the staff chooses that one.
6. The system asks for customer's information: first name, last name, gender, birthday, age, nationality, passport number, passport expiration date, address, email, phone number.
7. The staff inputs requested information.
8. Customer and flight information is saved and seat information is updated accordingly (the seat becomes booked).
9. The ticket is booked.

Alternative Flow:

1. If the system fails:
 - The user restarts the system
 - Internet connection is checked
 - User asks for IT support
2. If identity is not confirmed:
 - The user checks entered data and write it right
 - Change password if needed
3. If there are no available seats left: reservation cannot be done.
4. If the staff inputs wrong data: the staff edits information and saves the ticket again.

Special Requirements:

- Device from which the system can be used (PC, Mac)

- Platform (Windows, OS X)
- Confirmation from customer when everything is done

Frequency of Occurrence:

- Every time a ticket has to be booked.

Figure SWD – 5: Domain Model from the 2nd iteration

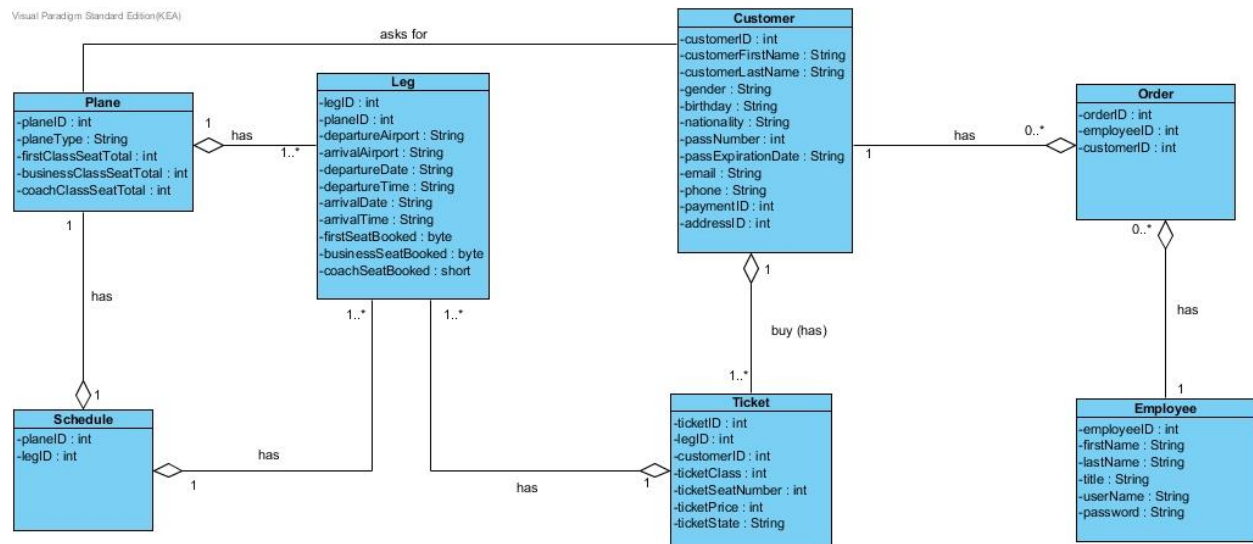


Figure SWD – 6: Class Diagram from the 2nd iteration

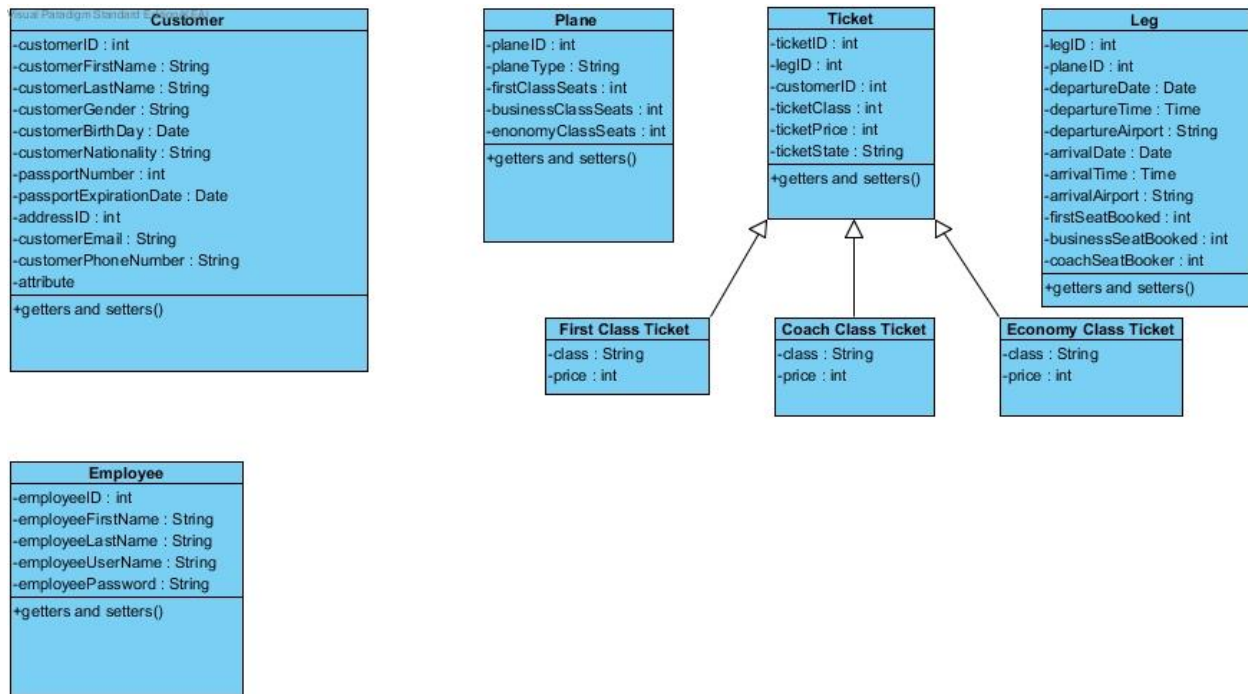
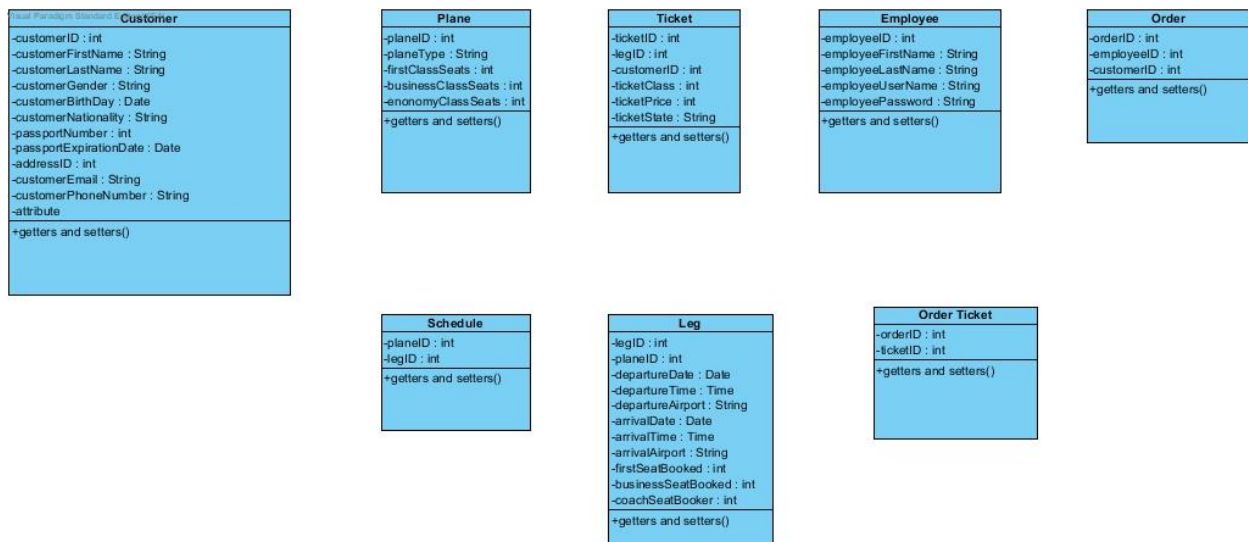


Figure SWD – 7: Class Diagram from the 2nd iteration



3.3 The 3rd iteration to 5th iteration – Elaboration Phase

Original Plan:

1. Construct the most significant requirements from the “book ticket”, “search flight”, “search order” (they were the most difficult ones as we could see, because the data we needed for those functions is from a lot of different classes and tables)
2. Keep the use cases, domain model, class diagram and other diagrams updated always

Actual Process:

After the first 2 iterations, we have captured the main requirements of the system, the domain model had the most objects we wanted to modelling. The requirements of the “Book Ticket”, “Search flight” and “Search order” use cases were the most significant, as a lot of information from different classes and tables in the databases are involved. For example, when we update a ticket, we will have to update the customer table, leg table, plane table, order table and order_ticket table in the database. To make the application work, we need to work on all 3 layers. The state machine of “ticket” has also proposed that a ticket’s status change can lead to a change in the database.

Therefore, the coming 3 iterations processes were similar, as the goal was to tackle the most significant requirements, and to keep on updating all other artifacts, while the main effort was focused on the construction part.

Appendix 5: Artifacts – Iteration 3 - Elaboration Phase (see page 77)

Appendix 6: Artifacts – Iteration 4 - Elaboration Phase (see page 83)

Appendix 7: Artifacts – Iteration 5 - Elaboration Phase (see page 87)

3.4 The 6th iteration - Construction Phase

Period: 26 – 28 May

Original Plan:

1. Finish the rest of the implementation on all layers, fixing bugs
2. Check all artifacts if they are updated

Actual Process:

The goal of this iteration is simple: to integrate the code from the different group members, make sure everything works as expected and finish the construction of the functionalities, and to apply CSS.

After the construction in the Elaboration phase, the main features were done, the domain model was stabilized, and we have found a few bugs in the application, which should be fixed in this iteration. The presentation layer was not fully integrated yet. For example, the log in scene was independent from other UI scenes, because we didn’t want to log in every time we wanted to test the code. This we will do in the last iteration.

We looked over all the artifacts and updated them.

3.5 The 7th iteration - Construction Phase

Period: 29 – 31 May

Original Plan:

1. Everything should get ready, regardless of the artifacts.

Actual Process:

The goal of this iteration is to finish everything. The code should be able to demonstrate the requirements we described in the use cases, and the methods of the classes should be the same in the diagrams (if there is one), and the class diagram should demonstrate all the classes we have in the code.

4. Artifacts

4.1 A brief introduction

Each iteration has either generated some artifacts, such as use cases, domain model, diagrams and code, or caused some artifacts to be changed and updated. What we will present here is those artifacts required in the Software Design requirements. They are the final version we have achieved after several changes and updates throughout the iterations.

For the construction of the system, please see the software construction section for all details (see page 39).

4.2 Use Cases

Here are 7 use cases in total, 4 in brief format, 2 in casual format, and 1 is fully dressed.

UC #1: Add Planes - Brief

Primary Actor: Flights and Operation Department Manager

Main Success Scenario:

1. The manager opens the system and logs in with his username and password.
2. The system verifies the identity of the manager, if it is valid, the staff can log in and use the application.
3. The manager chooses the function “Add Planes” in the menu bar.
4. The system asks for necessary input: plane type, number of first class, business class and economy class seats.
5. The manager types in the information and chooses to save the plane.

UC #2: Schedule Flights - Brief

Primary Actor: Flights and Operation Department Manager

Main Success Scenario:

1. The manager opens the system and logs in as manager with his username and password.
2. The system verifies the identity of the manager, if it is valid, the staff can log in and use the application.
3. The manager chooses the function “schedule flights”.
4. The system asks for the necessary input: airplane number, departure time, date and place (airport), arrival time, date and place (airport) and price of the flight.
5. The manager types in the requested information.
6. The flight is scheduled.

UC # 3: Book ticket- Fully Dressed

Scope: Book Ticket System

Level: User goal

Primary Actor: Customer Service Department Employee

Stakeholders and interests:

- The company owners: want to have a good system which is in line with the company business strategy, making their work easier, keeping the competitive advantages, and reducing unnecessary costs.
- Employee: wants to have an easy to use system when he wants to book a ticket.
- Customer: wants to book a ticket in a simple way.

Preconditions: The system is opened and ready to process, a customer wants to buy ticket.

Postconditions (Success Guarantee):

- A ticket has been booked.

Main Success Scenario:

1. The employee opens the system and logs in with username and password.
2. The employee gets into the operation menu, chooses the operation “Book Ticket”.
3. The staff searches flight after the customer’s requirements: departure place and date and arrival place.
4. The system shows the flights that satisfy the search condition(s).
5. The customer tells the staff which flight meets best his requirements and the staff chooses that one to book.

6. The system asks for customer's information: first name, last name, gender, birthday, age, nationality, passport number, passport expiration date, address, email, phone number and flight type (first class, business or economy).
7. The staff inputs requested information and chooses to save.
8. The ticket is booked.

Alternative Flow:

5. If the system fails:
 - The user restarts the system.
 - Internet connection is checked.
 - User asks for IT support.
6. If user's identity is not confirmed:
 - The user needs to enter username and password again.
 - Change password if it is needed.
 - The system sends an email to the user to recover/verify the account.
7. If there are no available seats left: reservation cannot be done.
8. If the staff inputs wrong data: the staff edits information and saves the ticket again, notifying customer if needed.

Special Requirements:

- Device from which the system can be used (PC, Mac)
- Platform (Windows, OS X)
- Confirmation from customer when everything is done

Frequency of Occurrence:

- Every time a ticket has to be booked.
- Can be many times per day.

UC # 4: Cancel ticket - Brief

Primary Actor: Customer Service Department Employee

Main Success Scenario:

1. The staff logs in to the system with username and password.
2. The system verifies the identity of the staff, if it is valid, the staff can log in and use the application.
3. The staff searches the ticket by searching the order ID or customer's name.
4. The system shows the search results; it is the order that contains the ticket/tickets to be canceled.
5. The staff finds the ticket and selects it.
6. The staff chooses the operation of "Cancel Ticket".
7. The ticket is cancelled.

UC # 5: Search Flight - Brief

Primary Actor: Customer Services staff

Main Success Scenario:

1. The staff logs in to the system with username and password.
2. The system verifies the identity of the staff, if it is valid, the staff can log in and use the application.
3. The staff selects the search function.
4. The system asks for search condition: departure date, departure place, arrival place.
5. The staff inputs the search info.
6. The system displays the data.

UC # 6: Refund ticket - Casual

Primary Actor: Customer Service Department Employee

Main Success Scenario:

1. The staff logs in to the system with username and password.
2. The system verifies the identity of the staff, if it is valid, the staff can log in and use the application.
3. The staff searches the ticket by searching the order ID or the customers' name.
4. The system shows the search results.
5. The staff finds the ticket and selects it.
6. The staff chooses the operation of "Refund Ticket". There will always be full refund for first class and business class. There will be no refund for the economy class if the ticket is canceled less than two weeks before the flight.
7. The ticket is refunded.

Alternative Flow:

1. If the system fails:
 - The user restarts the system.
 - Internet connection is checked.
 - User asks for IT support.
2. If identity is not confirmed:
 - The user needs to enter username and password again.
 - Change password if it is needed.
 - The system sends an email to the user to recover/verify the account.
3. If the wrong amount of money is refunded to the customer:
 - The transaction is cancelled and the right amount is sent to the customer.

UC # 7: Search Order - Casual

Primary Actor: Customer Service Department Employee

Main Success Scenario:

1. The employee opens the system and logs in with username and password.
2. The employee gets into the operation menu, chooses the operation "Search Order".
3. The staff searches order by orderID or customer names.
4. The system shows the tickets that satisfy the search condition(s).
5. The staff can choose ticket and operate on the ticket depends on the customer's requirement: confirm, cancel, or refund.
6. The ticket's status is viewed or changed.

Alternative Flow:

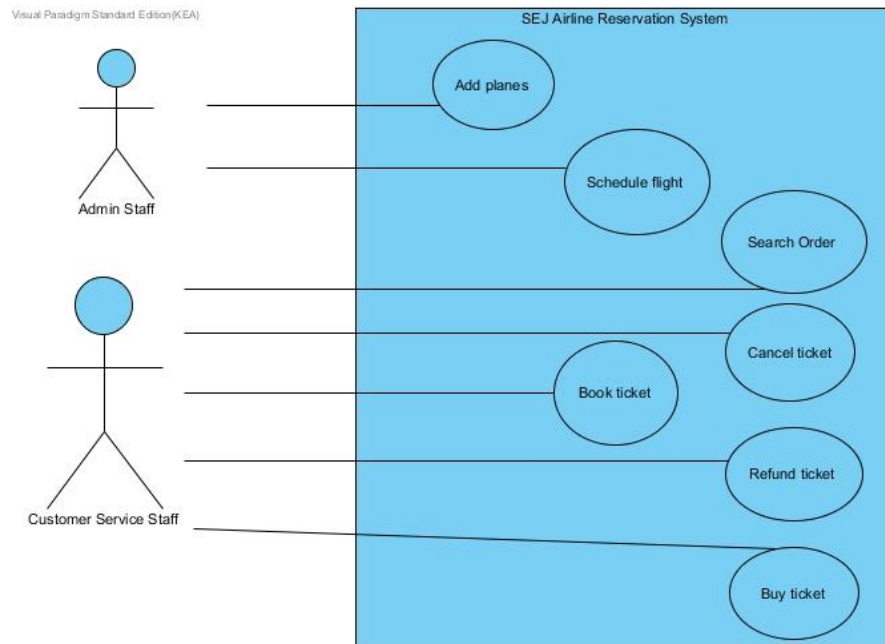
1. If the system fails:
 - The user restarts the system.
 - Internet connection is checked.
 - User asks for IT support.
2. If identity is not confirmed:
 - The user needs to enter username and password again.
 - Change password if it is needed.
 - The system sends an email to the user to recover/verify the account.
3. If the staff inputs wrong data: no result will show and type in again.

4.3 Use Case Diagram

The Use Case Diagram shows the user's interaction with the system.

The user of the system will be the employees of the airline company, so the primary actor of the use case diagram is the staff. But there are 2 types of functions: the first being that the system should be able to allow the company to add planes and schedule flights, this will be done by the administration department staff, and the second is to handle ticket related matters, and these functions will be done by the customer service department staff. That is why there are 2 primary actors in the use case diagram. And they don't have access to each other's responsibility.

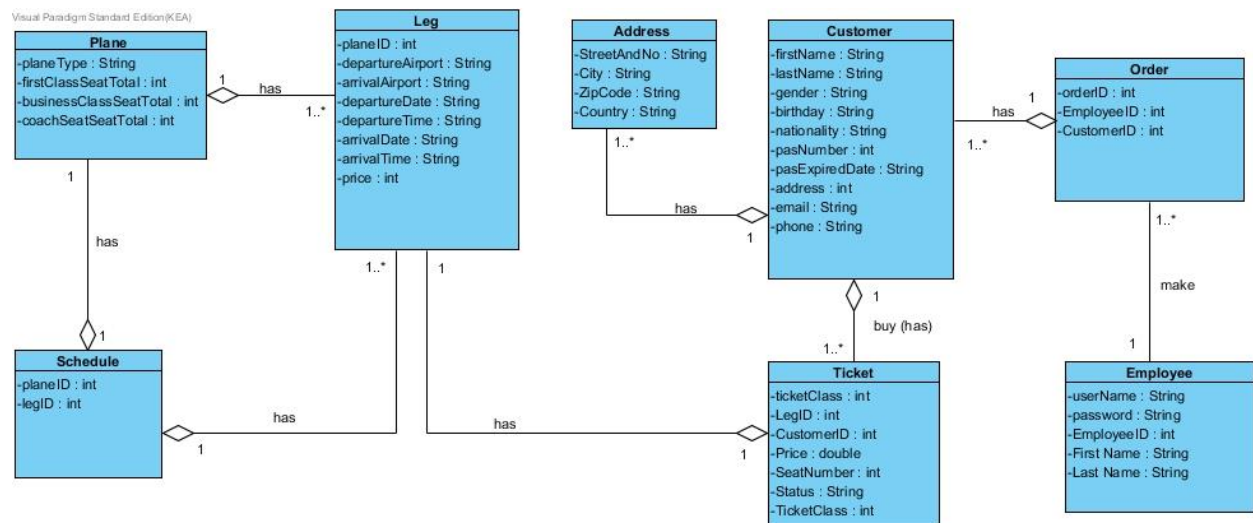
Figure SWD – 8: Use Case Diagram



4.4 Domain Model

The Domain Model is the real world we want to model. The objects present in the domain model have become the data types in the system.

Figure SWD – 9: Domain Model

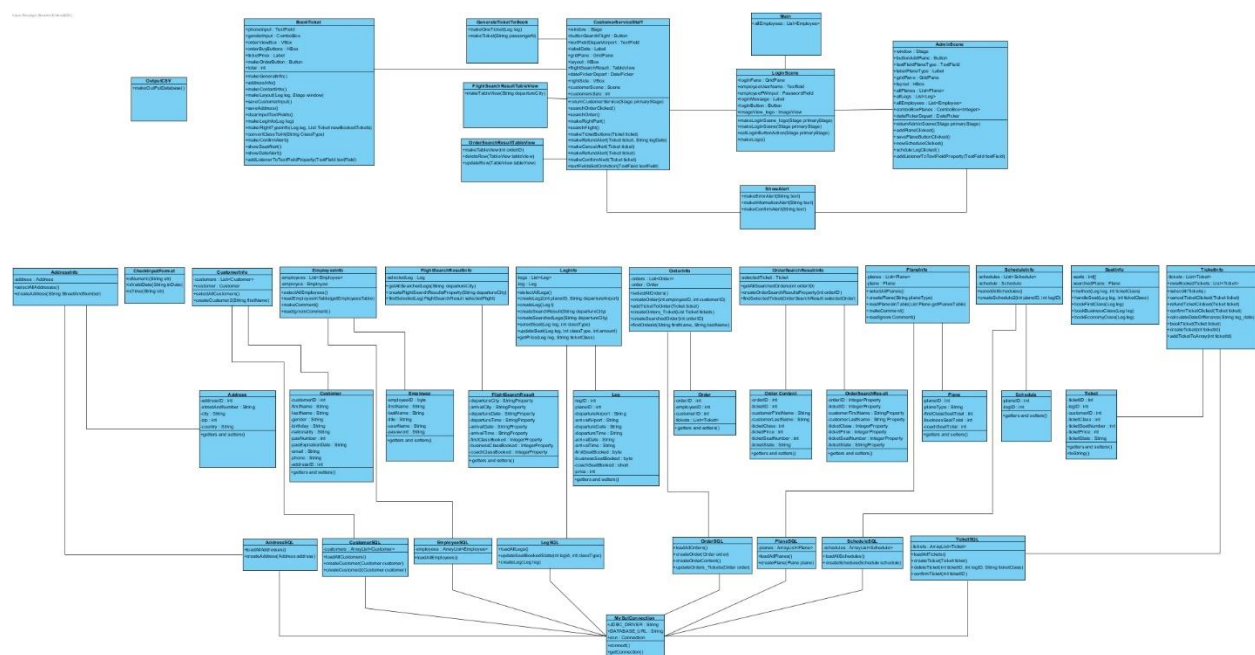


4.5 Class Diagram:

As the Class Diagram suggests, we have applied 3-layer architecture.

The top part is the presentation layer, in this diagram it doesn't connect with the application layer. The reason we have left the connection blank, is because the diagram contains a lot of information already, and drawing all the connections will make the diagram difficult to read. But this layer has got its functionality from the application layer. For the same reason of making the diagram more readable, not all the methods with parameters are specified with what those parameters are, if it is very obvious what parameter it should have.

Figure SWD – 10: Class Diagram



How does it look like in code?

“Employee” object as an example:

```
public class Employee
{
    private byte employeeID;
    private String firstName;
    private String lastName;
    private String title;
    private String userName;
    private String password;
```

```

public Employee(byte employeeID, String firstName, String lastName, String title, String userName,
String password)
{
    this.employeeID = employeeID;
    this.firstName = firstName;
    this.lastName = lastName;
    this.title = title;
    this.userName = userName;
    this.password = password;
}

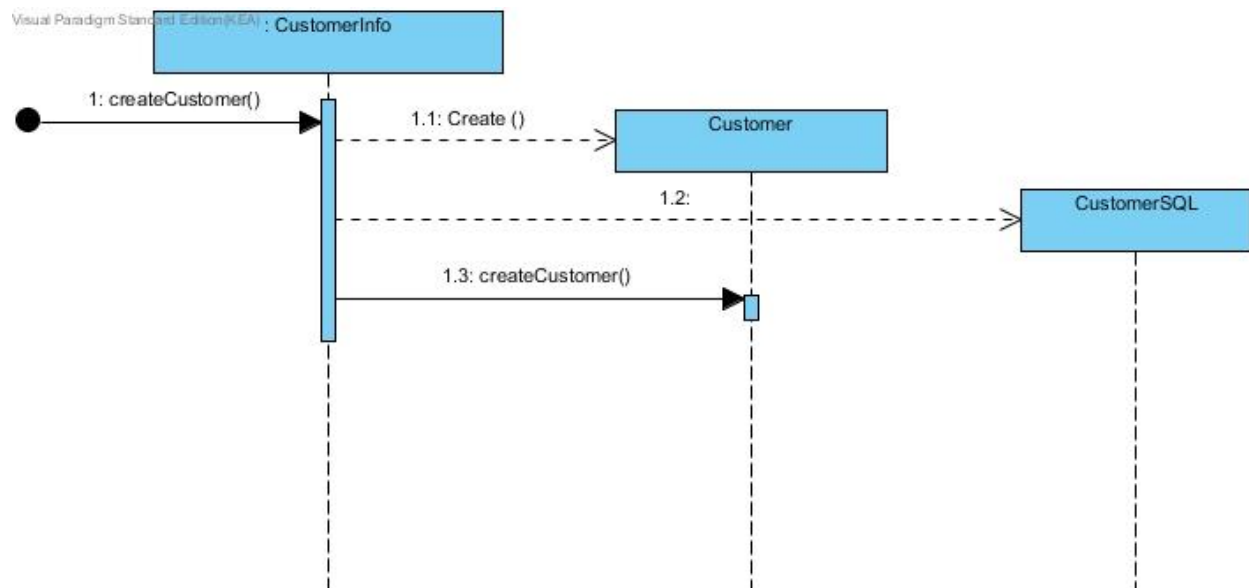
```

4.6 Sequence Diagram: Create Customer

This diagram is about the interaction between objects arranged in time.

In this diagram, it shows that the class called CustomerInfo has a method called createCustomer(), when the method is called, a Customer is created, and in the meanwhile, it calls the createCustomer() method from CustomerSQL class.

Figure SWD – 11: Sequence Diagram: Create Customer



Are the classes in the SD the same as in the domain model? If not, why?

The reason some objects in the Sequence Diagram are not the same as in the Domain Model is that the Domain Model is a business model used to make the representation of the things in real world, whereas the Sequence Diagram is a technical diagram that shows the interaction between objects.

How does it look like in code?

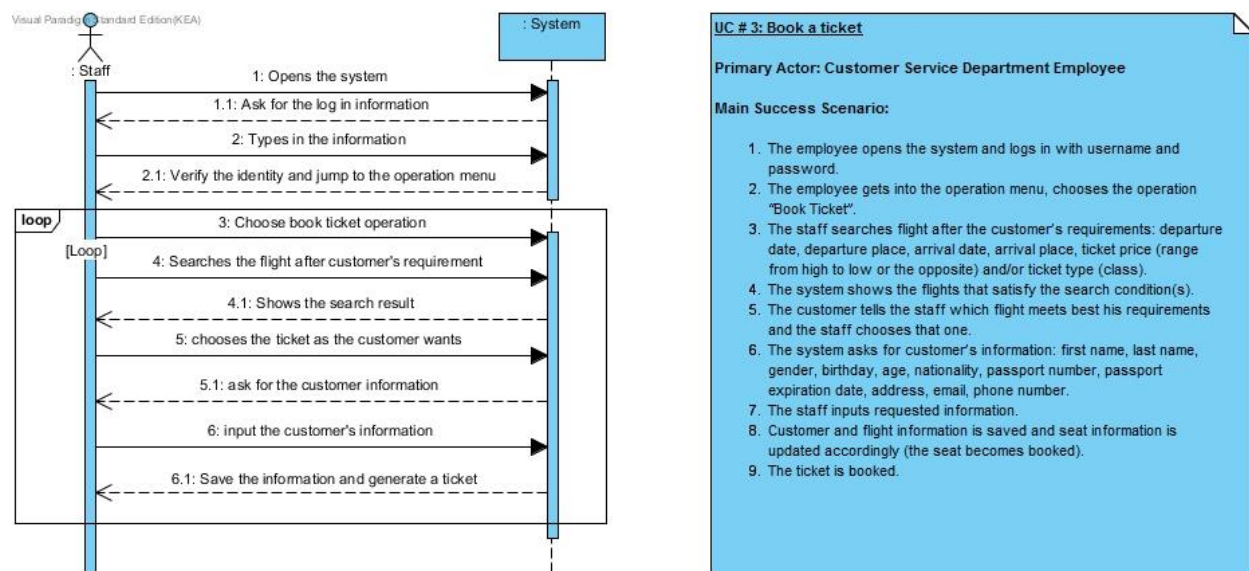
```
public static void createCustomer(String firstName, String lastName, String gender, String
birthday, String nationality, int pasNumber, String pasExpiredDate, int addressID,
String email, String phone)throws Exception
{
    customer = new Customer(firstName, lastName, gender, birthday, nationality, pasNumber,
pasExpiredDate, addressID, email, phone);

    CustomerSQL.createCustomer(customer);
}
```

4.7 System Sequence Diagram:

This diagram shows the particular scenario for the use case “Book Ticket”. It shows the flow of a staff’s interaction with the system.

Figure SWD – 12: System Sequence Diagram

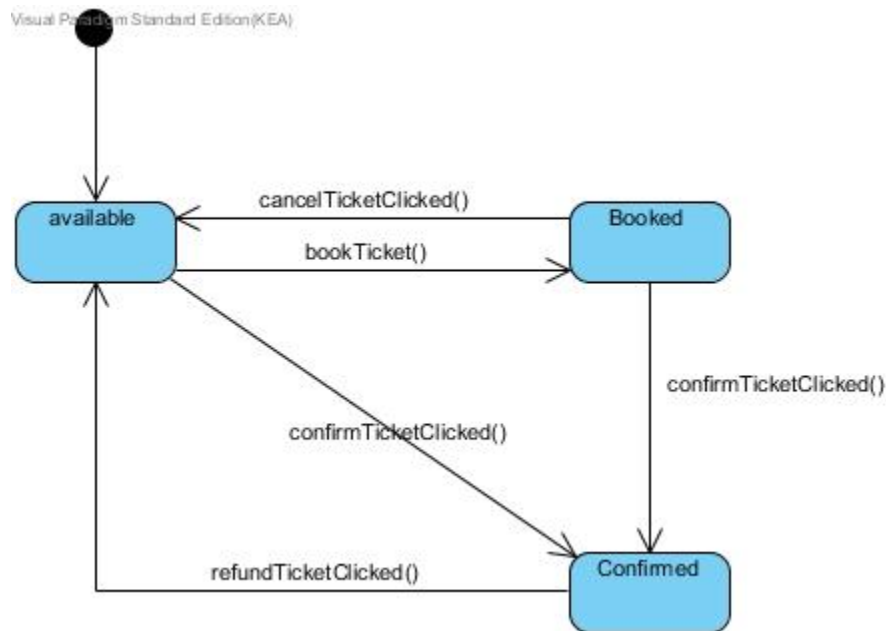


4.8 State Machine: Ticket

The initial state of a ticket is the “available” state. That means a ticket is ready to be booked or to be bought (“confirmed” status). When a ticket is booked through the *bookTicket()* method, the state of the ticket is changed to “booked” or if a ticket is confirmed through the *confirmTicketClicked()* method, its state becomes “confirmed”. Once a ticket is in the “booked” state, it can be canceled through the *cancelTicketClicked()* method and return to the “available” state, or it will be changed to “confirmed”

state through the *confirmTicketClicked()* method. When a ticket is in the “confirmed” state, it can become “available” again through accessing the *refundTicketClicked()* method.

Figure SWD – 13: State Machine “Ticket”



How the methods look like in the code?

```
public static void bookTicket(Ticket ticket) throws Exception
{
    TicketSQL.createTicket(ticket);
    tickets.add(ticket);
    LegInfo.updateSeat(ticket.getLegID(), ticket.getTicketClass());
}
```

5.Conclusion of following UP

We have really enjoyed the process of UP! Comparing to the water-fall method we used in the previous projects, the benefit of Unified Process is very obvious: it allows the fact that not everything will be clear from the beginning. This has reduced the unnecessary guessing of the requirements. Instead, the requirements are found step by step, which is more logical and agile. Each iteration consists of different disciplines from finding requirements, design, modelling and construction, this offers the chance to optimize the code all the time.

Looking back at the whole process, we had first made some small “bricks” according to the understanding we had, then slowly we got the big picture of the application. This method helped us

avoid unrealistic ambitions, thus the outcome is more reasonable. We could imagine, if we were really working for a company, the UP can probably be quite agile and energy-saving, like for example it could save the effort being put into unnecessary implementation because of the wrong understanding of requirements.

The interesting thing is that, as soon as we start to program, our minds very easily get into the “water fall” mind-set, it just happens sometimes, so at these moments we have to remind ourselves to follow UP instead.

6. Three Layered Architecture

We are applying the 3 layered architecture.

The Data Access Layer is where the connection with the database is made, which queries data and stores the data in arraylists.

The Application Layer is the layer where data types are constructed, and it receives data from the Data Access Layer, and does the operations based on those data. This layer consists of data types and controllers (in our code, they are the classes which end with “Info” in their names). Data types are made according to the domain model, but some of them are made due to the pure fabrication pattern, like the “FlightSearchResult” or the “CheckInputFormat” class.

The Presentation Layer is where the UI is made. There is no logical operation in this layer, as that is done in the Application Layer.

7. Git Experience

We didn’t feel the need of using GIT in the first 2 iterations, as we didn’t code that much at that moment, so the old fashioned way of just giving others our java files worked fine enough. But from the 3rd iteration, as our code was getting more complicated, and when we were working on different parts of the same code, we often had the need of rewriting or changing others’ code, so at this point we thought it was a good idea to start using GIT.

We have tried to use it for the 3rd iteration, and we could see the benefit of using it, especially since it keeps track for us of what has been changed by who. But the problem was that we needed time to get used to it and learn how to use it properly. During the iteration, we have not always succeeded in sharing our code in a proper way, and taking in consideration the time we had, we didn’t feel like we could afford to spend a lot of time on figuring it out, and therefore we decided not to use it anymore for this project.

8. GRASP

GRASP stands for General Responsibility Assignment Software Patterns.

1. Information Expert – assign the responsibility to the class that has the information to fulfill it

For example, the MySqlConnection class is responsible to connect the java application with the database. It has all the data needed for this process (fields, methods) and therefore this responsibility was assigned to it, based on the Information Expert pattern.

Another example is the TicketInfo Class. It has all the data and methods needed to work with ticket related processes.

2. Low Coupling – to reduce the amount of required connections between objects

The level of dependencies between objects should be as low as possible. For example, the CustomerInfo class only deals with events related to customers, it doesn't have any connection with the plane class. If the CustomerInfo class is changed, it will not influence other classes such as plane, leg or employee.

3. High Cohesion – a class contains focused and related behavior

Each class has its own responsibility, for example, the classes in the data access layer are only responsible for accessing the database in order to get and write information, the ticket controller class does all the logical operations on ticket, and the data types only have getters and setters and a toString() method.

4. Controller – the first class beyond the UI layer that is responsible for receiving or handling a system operation message

The classes whose names end with "Info" are the controllers in our system, they deal with the system events, and they offer the functionalities to the GUI in the presentation layer.

5. Pure Fabrication

We have a class called "CheckInputFormat" which is there purely to solve the problem of having to verify user input. They are made to achieve low coupling and high cohesion.

Section IV: Software Construction

1. Database and EER Diagram

The creation of the database was an important part in constructing the application. Here we would like to explain some of the decisions that we have made.

In this application, one customer can have one order with many tickets on it that can belong to many customers, but only for one leg. For example: customer Emily would have one order with two tickets, one for her and one for her boyfriend, flying from Copenhagen to Berlin. She is the one that can afterwards confirm the tickets or cancel them. If then they want to travel from Berlin to Paris or back to Copenhagen, they would have to make another order. This way, it is easier for us to handle the customers, the tickets and update the information about the desired leg.

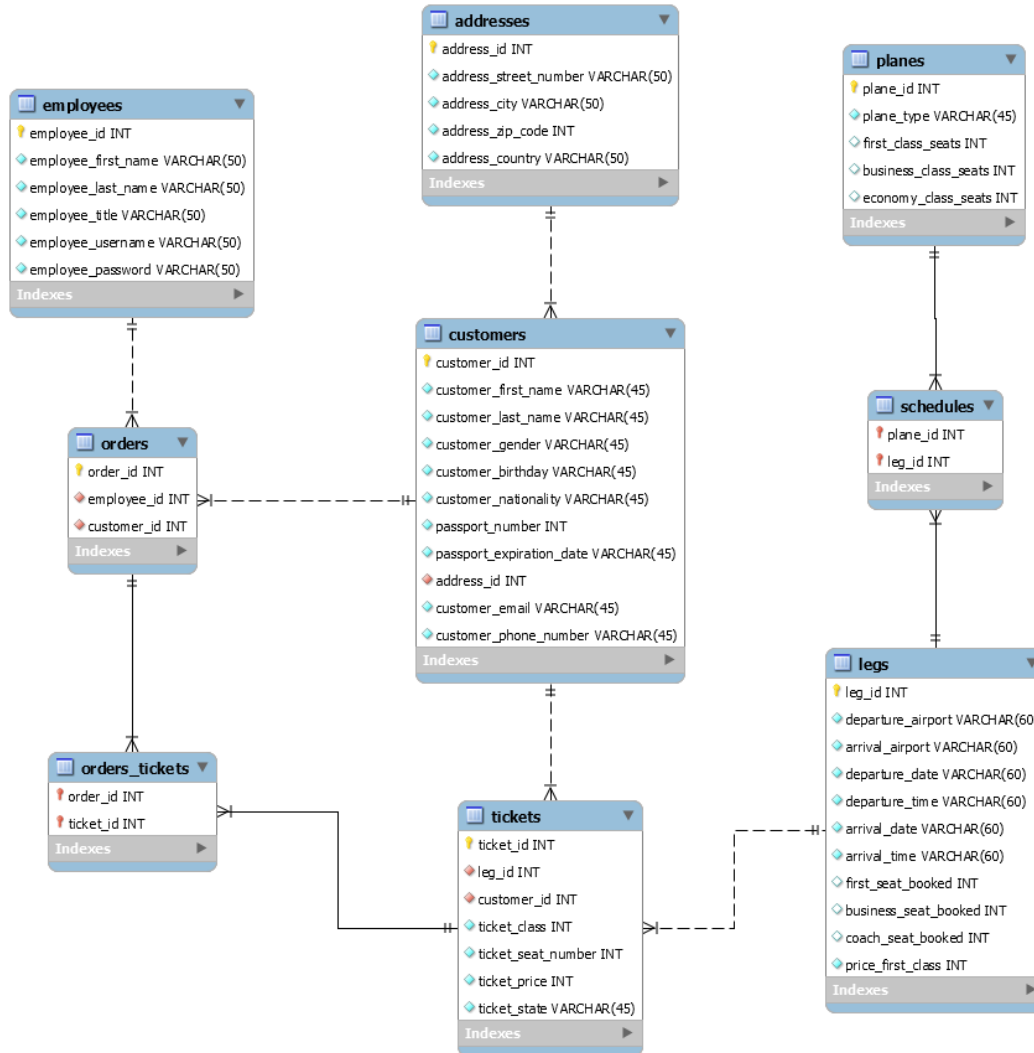
But what is a leg? According to <http://aviation.stackexchange.com>, a leg is “a trip of an aircraft, from take-off to landing”, which is different from the term “flight” that includes also stopovers and/or change of planes.

Each leg (flight) starts with 0 first class, 0 business class and 0 economy class seats booked. These numbers increase by one every time a ticket is booked or bought, depending on the ticket class the customer chooses.

Many legs will form a schedule and each plane will be assigned a schedule. This makes it possible to for instance choose a plane and see its schedule, meaning all the legs that have been assigned to this schedule for this plane. This option is not implemented in the application, as our main priority was to make all functionality that was *required*.

The EER Diagram below shows the structure of our database.

Figure SWC – 14: EER Diagram



The database is in the 3rd Normal Form of Normalization.

1NF – For achieving this we created a new table for addresses, instead of storing all information about an address in one cell in the customers table.

2NF – As can be seen, the non-key columns are dependent on the table's primary key.

3NF – Every non-key column depends only on the primary key.

By normalizing the data, we reduce data redundancy and therefore avoid data modification issues.

The 'orders_tickets' and 'schedules' tables reflect a many-to-many relationship between orders and tickets and between legs and planes. This means that an order can have many tickets and a plane can have many legs.

For example, the 'orders_tickets' table will this, so when we search for order number 3, we will get two tickets.

Figure SWC – 15: the “order_ticket” table in the database

	order_id	ticket_id
▶	1	1
	2	2
	3	3
	3	4
	4	5
★	NULL	NULL

Appendix 8: SQL Script for creating the database (page 93)

2. Scope of the Application

The Software Construction discipline was used mainly in the Elaboration and in the Construction phases of the UP. As in the elaboration phase the workload started with focusing more on the requirements, analysis and design, programming was gradually introduced and it was getting emphasized more from the 3rd iteration of the Elaboration phase to the end of the Construction phase. All classes from the Class Diagram are integrated in the code and there is a clear correlation between Software Construction and Software Design.

3. A thorough description of the construction of the application and decisions

The construction of the application is based on the class diagram and it has a layered architecture. This is because we want to avoid direct access from the GUI to the database and to reduce dependencies between classes, so that when something is changed in one of the layers, the others will not be influenced much.

Figure SWC – 16: Data Access Layer



The classes in the data access layer are the only ones that have direct access to the database and they are divided in the following way: MySqlConnection and SQL Classes.

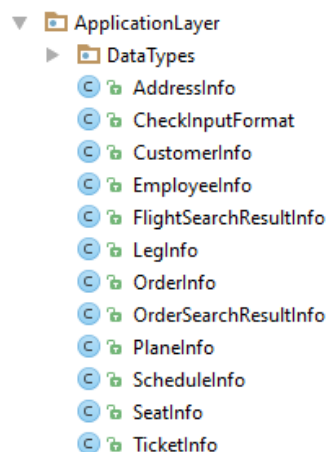
The MySqlConnection class establishes the connection to the database; the name of the database driver, the URL of the DBMS, user id and password are specified here. It contains two methods (connect() and getConnection()) that will be later used in the other classes of the Data Access Layer.

The so-called SQL classes (“EmployeeSQL”, “OrderSQL” etc.) are used for reading from and writing to the database. This is achieved within a few steps:

1. Get the connection from the MySqlConnection class.
2. Prepare the SQL statements through statement objects. These can be “SELECT”, “INSERT”, “DELETE” and “UPDATE”.
3. Access database and retrieve or manipulate data (execution of the statements).
4. Close the connection to DBMS.

We decided that in this particular project it is more reasonable to write or make changes in the database frequently as long as the connection is closed after accessing it. For example, as soon as the ticket is booked, the tables in the database are updated, in order to avoid dirty reads and to prevent loss of data.

Figure SWC – 17: Application Layer

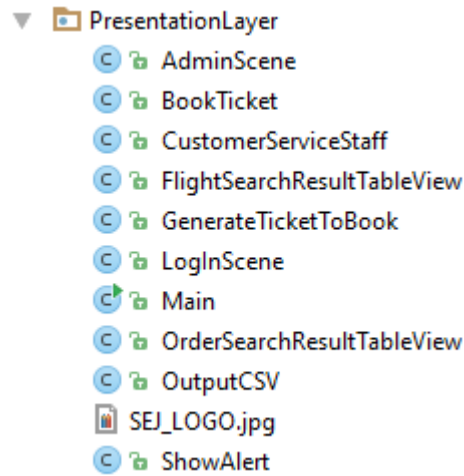


The Application layer is the tier between the database and the UI. It contains the classes in which the logical methods are made and the data types the application operates with: Data Types and Info Classes.

The Data Types package consists of objects we are working with: Customer, Employee, Plane, Ticket etc. They are based on the noun list and each contains fields, getters, setters as well as the toString method. Each table from the database has a corresponding class in code. In addition to that, we created 3 more classes that contain information from more than 1 table: OrderContent, OrderSearchResult and FlightSearchResult. Because the search result will be shown in a TableView, the last two classes have fields and methods that allow working with TableViews. Populating such UI component requires an Observable List, therefore the fields from these classes must be declared as Simple Properties.

The Info classes control the application's functionality, making calculations (for example the getPrice() method in LegInfo) or checking for valid data (isTime, isValidDate, isNumeric methods). They are connected with the surrounding layers, taking information from the presentation layer and sending it down to the database, and the other way around.

Figure SWC – 18: Presentation Layer



The classes in this layer manage the interaction with the user of the application, making it possible to display information or to accept input. Their sole purpose is to translate tasks into something that the user can understand. Some of them take care of only one UI component, for example the FlightSearchResultTableView or GenerateTicket to make a booking. We use only one stage, but multiple scenes. The transition between the scenes is illustrated below.

```

Scene adminScene = AdminScene.returnAdminScene(primaryStage);
primaryStage.setScene(adminScene);
primaryStage.show();
  
```

3.1 Add Plane & Add Schedule

The application is designed to be simple but efficient, and it still meets all requirements. For adding a plane or a schedule, the user has to input information in TextFields. All this data is then sent to the Application layer through method calls with parameters. There the object is created and sent to the Data Access Layer in order to be written in the database.

GUI:

```

PlaneInfo.createPlane(textFieldPlaneType.getText(), Integer.parseInt(textFieldFirstClass.getText()),
    Integer.parseInt(textFieldBusinessClass.getText()), Integer.parseInt(textFieldEconClass.getText()));
  
```


Application Layer:

```
public static void createPlane(String planeType, int firstClassSeats, int businessClassSeats, int econClassSeats)
    throws Exception
{
    plane = new Plane(planeType, firstClassSeats, businessClassSeats, econClassSeats);
    PlaneSQL.createPlane(plane);
}
```

Data Access Layer:

```
public static void createPlane(Plane plane) throws Exception{ //create new plane
    Connection con = MySqlConnection.getConnection();
    Statement st = con.createStatement();
    String sql = ("INSERT INTO planes VALUES (" + plane.getPlaneID() + ", "
        + "\"" + plane.getPlaneType() + "\"" + ", "
        + plane.getFirstClassSeatTotal() + ", "
        + plane.getBusinessSeatTotal() + ", "
        + plane.getCoachSeatTotal() + ")");
    st.executeUpdate(sql);

    con.close();
    st.close();
}
```

3.2 Search Flight

Searching a flight requires input for departure place, arrival place and departure date. The information is passed on to the FlightSearchedResultTableView, then to the FlightSearchResultInfo and finally to the LegInfo class. Here we use a method that loops through the array of all legs and returns a new array with the legs that satisfy the search condition. The array is returned in the FlightSearchResultInfo, where another method wraps information of each leg into a FlightSearchResult object (its fields are Simple Properties) and adds it to the ObservableList used to populate the TableView.

3.3 Search Order

One can search an order either by order id or by first name and last name. If one searches by first name and last name, the information from the TextFields is passed on to the OrderInfo class, in which a method finds the order id associated with the order that has the customer id with the specified first name and last name. Then the same method used to search by order id is called. This way, we can achieve reusability of code and therefore reduce redundancy. The logic behind searching an order is similar to the one of searching a flight. The following snippet of code shows this.

```
if(!textFieldOrderID.getText().isEmpty())
    orderSearchResult = OrderSearchResultTableView.makeTableView(Integer.parseInt(textFieldOrderID.getText()));
else
    orderSearchResult = OrderSearchResultTableView.makeTableView
        (OrderInfo.findOrderId(textFieldFirstName.getText(), textFieldLastName.getText()));
```

Operations on tickets: confirm, cancel and refund

After the order search result is shown in the TableView, one can click on a row and by a “setOnClick” method applied to the tableview, it will return the ticket that corresponds to the row. Then, buttons that can operate on that ticket appear according to the ticket state. In case of a booked ticket, confirm and cancel buttons pop up, however, you can only refund a confirmed ticket.

3.4 Confirm ticket

If the confirm ticket button is pressed, firstly an information dialog box appears. Then, through the confirmTicketClicked() method in the TicketInfo class, the confirmTicket method is called in the SQL class which updates the ticket state to “confirmed”. The TableView is also updated.

GUI:

```
TicketInfo.confirmTicket(ticket);
OrderSearchResultTableView.updateRow(orderSearchResult);
```

Application Layer:

```
public static void confirmTicket(Ticket ticket) throws Exception{
    TicketSQL.confirmTicket(ticket.getTicketID());
}
```

Data Access Layer:

```
public static void confirmTicket(int ticketID) throws Exception{
    Connection con = MySqlConnection.getConnection();
    Statement st = con.createStatement();
    String sql = ("UPDATE tickets\n" +
        "\tSET ticket_state = 'confirmed'\n" +
        "\tWHERE ticket_id = " + ticketID);
    st.executeUpdate(sql);

    con.close();
    st.close();
}
```

3.5 Cancel & Refund

When cancelling a ticket, the cancelTicket method in the Application Layer sends the ticket id, the leg id of that ticket and the ticket class (first class, business or economy) as parameters to the Data Access Layer, in which the deleteTicket method will: delete the ticket from the tickets and orders_tickets tables in the database and update the legs table to -1 seats booked for that class of ticket.

As we want our code to be efficient, the refund feature uses the same method call because it will result in the same changes in the database, but has some additional functionality too. First of all, the information dialog box is different, showing the price of the ticket to be refunded. Then, if it is an economy class ticket and the cancellation is done within less than two weeks before the flight, there will be no refund and the user has to choose whether to cancel the ticket or not via a Confirmation Dialog Box.

Data Access Layer:

```
public static void deleteTicket(int ticketID, int legID, String ticketClass) throws Exception{
    Connection con = MySqlConnection.getConnection();
    Statement st = con.createStatement();
    String sql = ("DELETE from tickets\n" +
        "\tWHERE ticket_id = " + ticketID);

    String sql2 = ("DELETE from orders_tickets\n" +
        "\tWHERE ticket_id = " + ticketID + " AND order_id = (SELECT order_id FROM \n" +
        "\t\t\t\t\t\t\t\t\t\t(SELECT order_id FROM orders_tickets WHERE ticket_id = " + ticketID + ") x)");

    String sql3 = ("UPDATE legs \n" +
        "\tSET " + ticketClass + " = " + ticketClass + " - 1\n" +
        "\tWHERE leg_id = " + legID );

    st.executeUpdate(sql);
    st.executeUpdate(sql2);
    st.executeUpdate(sql3);

    con.close();
    st.close();
}
```

Calculating the date difference has been a challenge for us. This is successfully done now in the TicketInfo class in the Application Layer.

The first implementation of this option is presented below. This caused issues because the Date class in the Java Util libraries does not handle daylight savings time.

```
public static long CalculateDateDifference(String leg_date) throws Exception {
    DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
    Date today = new Date();
    dateFormat.format(today);

    Date legDate = dateFormat.parse(leg_date);

    final long MILLIS_PER_DAY = 24 * 3600 * 1000;
    long msDiff= legDate.getTime() - today.getTime();
    long daysDiff = Math.round(msDiff / ((double)MILLIS_PER_DAY));

    return daysDiff;
}
```

Converting Dates to LocalDates and using the Java Time library was the solution. This is the second implementation.

```

public static long calculateDateDifference(String leg_date) throws Exception {
    DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
    Date today = new Date();
    dateFormat.format(today);

    Date legDate = dateFormat.parse(leg_date);

    LocalDate localDateLEG = legDate.toInstant().atZone(ZoneId.systemDefault()).toLocalDate();
    LocalDate localDateTODAY = today.toInstant().atZone(ZoneId.systemDefault()).toLocalDate();

    long days = ChronoUnit.DAYS.between(localDateTODAY, localDateLEG);

    return days;
}

```

3.6 Book a ticket

The flight search will result in a TableView populated with legs. When clicking one of the rows, the book ticket button appears which will lead to another Scene. This will contain information about the selected leg, information about the ticket and the customer input (general information, address and contact info).

By pressing the Save button, the information about the customer including address is saved to the database and the ticket created is added to the newBookedTickets array. The TextFields are cleared in order to allow entering of information about a new customer on the same order. That is because an order with one leg can have multiple tickets that belong to different customers, but an order has only one customer id. In this case, the first customer registered is responsible for the order. The Make Order button is set on action to send, for each ticket of the newBookedTickets array, information to the Application Layer which then will: insert a new ticket in the tickets and in the orders_tickets tables, and update the legs table, changing how many seats have been booked for each class of flight. The Buy Tickets button has the same functionality, but before the information is sent to the Application Layer the tickets' state is changed to "confirmed".

Calculating the price depending on the ticket class

```

//return ticket price for different classes
public static int getPrice(Leg leg, String ticketClass)
{
    int price = 0;
    if(ticketClass.equalsIgnoreCase("Business Class"))
    {
        price = (leg.getPrice() * 85) / 100;
    }
    if(ticketClass.equalsIgnoreCase("Economy Class"))
    {
        price = (leg.getPrice() * 70) / 100;
    }
    if(ticketClass.equalsIgnoreCase("first class"))
    {
        price = leg.getPrice();
    }
    return price;
}

```

In the code, there are some unused methods and constructors. We tried to get some of them to work, but couldn't manage to do it and decided to proceed with the project as the deadline was approaching. We didn't delete them though, because they could be useful for future development of our application, if implemented correctly. Here are some examples:

```
// unused method
// it is meant to avoid adding ticket objects to an array in the Presentation Layer
public static void addTicketToArray(int ticketId, int legId, int customerId, int ticketClass, int seatNr,
                                   int ticketPrice, String ticketState){
    newBookedTickets = new ArrayList<>();
    Ticket ticket = new Ticket(ticketId, legId, customerId, ticketClass, seatNr, ticketPrice, ticketState);
    newBookedTickets.add(ticket);
}

// unused constructor
public Customer(int customerId, String firstName, String lastName, String email, String phone)
{
    this.customerID = customerId;
    this.firstName = firstName;
    this.lastName = lastName;
    this.email = email;
    this.phone = phone;
}
```

3.7 Other features

The application provides help facilities for the user. It checks every time if all the fields are filled out and if the input is in the right format to prevent getting exceptions such as `NullPointerException` or `IllegalArgumentException`. The check is done when pressing a button or by adding a listener to a `TextField`. We created a new class called `CheckInputFormat` in the Application Layer that makes sure the data that the user types in is in the right format. If something goes wrong, an error dialog box will appear informing the user about the problem. The `ShowAlert` class in the Presentation Layer handles this. The feedback is adequate, meaning the user understands what he should do in order to set the situation right. Confirmation alerts also appear, so that the user won't be tempted to try again, thinking that nothing happened the first time, which might lead to unexpected results.

We designed the application to be fast and easy to use. It provides flexible interaction through mouse and keyboard, which is why some of the `TextFields` have an action set when the "enter" key is pressed.

```
textFieldEconClass.setOnKeyPressed(e -> {
    if (e.getCode() == KeyCode.ENTER) {
        savePlaneButtonClicked();
    }
});
```

3.8 How can we improve it?

Future development of the application can be:

1. To divide Customer to 2 classes, customer and passenger. A customer is the one who makes the order, or the airline company can contact, but not necessarily is a passenger.
2. Passenger's personal information can be more complete, for example, one passenger/customer can have more than 1 address.
3. When an order is made, it should be able to handle multiple legs.
4. the user can see a schedule for a plane, meaning all legs that have been assigned to that plane, in a TableView.
5. When a ticket is generated, it would have a random seat number.
6. The user will be able to edit information on tickets, such as changing the name, date or time.
7. The admin could have the option to add an employee and change his information.

4.Exciting Snippet of Code – Handling Seats

The SeatInfo class from the Application Layer handles the seat problem: how many seats does the plane have, how many of them are first/business/economy seats and how many of each kind are booked already for the leg that we want to book the ticket on? The leg object contains the plane id as a field. The getSeat() method begins with looping through the planes array in order to find the one assigned to the specific leg. Once the plane has been found, it creates an array with the fixed size of how many seats there are in the plane. This array holds assignments for all of the 3 classes of flight.

Each element in the array is instantiated with 0, meaning all seats are available. The array is then split into 3 parts depending on the desired ticket class and taking into consideration the amount of seats in the plane and the amount of seats booked already. In each of the bookFirstClass, bookBusinessClass and bookEconomy methods there is a loop iterating the array within a specific range depending on the class and when it finds a free seat it will mark it as booked by putting the value 1 in that index of the array. The seat number returned will be index + 1, as an array starts from 0 and normally we count seats from 1. If there are no free seats, the method will return -1. In this case, an error dialog box appears, asking the user to change the class of the ticket or to change the flight. After booking, the legs table is updated, adding 1 to the booked seats.

```

public static int getSeat(Leg leg, int ticketClass) throws Exception{
    List<Plane> planes = PlaneInfo.selectAllPlanes();

    for(int i = 0; i < planes.size(); i++)
        if(planes.get(i).getPlaneID() == leg.getPlaneID())
            searchedPlane = planes.get(i);
    int maximumSeats = searchedPlane.getFirstClassSeatTotal() +
        searchedPlane.getBusinessSeatTotal() +
        searchedPlane.getCoachSeatTotal();
    seats = new int[maximumSeats];
    for (int i = 0; i < seats.length; i++)
        seats[i] = 0;

    return handleSeats(leg, ticketClass);
}

public static int handleSeats(Leg leg, int ticketClass){
    int seatnumber = 0;
    if (ticketClass == 1) {
        seatnumber = bookFirstClass(leg);
    }
    if (ticketClass == 2) {
        seatnumber = bookBusinessClass(leg);
    }
    if (ticketClass == 3) {
        seatnumber = bookEconomyClass(leg);
    }
    return seatnumber;
}

private static int bookBusinessClass(Leg leg) {
    for (int i = searchedPlane.getFirstClassSeatTotal() + leg.getBusinessSeatBooked();
        i < searchedPlane.getFirstClassSeatTotal() + searchedPlane.getBusinessSeatTotal(); i++) {
        if (seats[i] == 0) {
            seats[i] = 1;
            return i + 1;
        }
    }
    return -1;
}

```

5. What works and what doesn't in our system

5.1 What works:

We have managed to get most things working in our SEJ system. It has a working login system, which will take the user to two different main menus depending on the department they work, one which is the admin section, that has all the admin functionality, and the other which has all customer services related functionality. The system compares login information to what is stored in the database and only grants access if the information matches. If the login is an admin login, it will take the user to the admin section.

The admin section has three functions, all of which work: Adding planes, adding new schedules and making two of our database tables to CSV files as per the OSCA requirements. The system is able to add planes, which are saved in the database. The planes have a type (like Boeing 747) and the number of seats

in each class. These planes can then be used in flight schedules. That's the next Admin function which works well too.

A new schedule can be created and stored in the database. A plane is chosen, and then legs are created. A leg has the departure and arrival information of one flight, meaning from one airport to another. A Schedule can have several legs, or just one. We had a bug where the planes in the comboBox were being added every time the New Schedule button was pressed, but we managed to fix that. It was because every time the New Schedule button was clicked, all the planes in the database would be added again to the array of planes. We solved this by adding an array that is loaded only once when the program starts. The Admin section also has a button which exports the data in the Planes and Employees tables from the database to CSV files.

When one logs in with a Customer Service login, the user goes to the Customer Services menu. The first thing one can do is book a flight. One can search for the flight one wants and then book either one or several tickets on that flight. We made it so that the ticket is automatically assigned a seat number in the chosen class, and if there are no seats free of that class a message will appear saying so. Our system is able to then book or buy all the tickets that were made, and all of this is stored in the database. We ran into several bugs and problems while we were making this part. The biggest was that after we booked or bought a ticket, if we tried to book another one, it would not work and we would get errors. Some parts of our program were based on the size of an array, and this array kept on changing as the program was run, so in some scenarios it would cause errors. We solved this by saving the size of the array when the program started in a variable and used that from there on.

The other part of the Customer Services section is related to existing orders. One can search for orders and cancel, refund or confirm booked tickets. We had a bug when searching for orders, where we would get other orders than the one searched for. This was due to the way orders and tickets were linked in the database, but it was changed and works now.

Our system can do the cancelling, refunding and confirming of tickets with no problems. When a ticket is cancelled, the seat becomes available to be booked again and the ticket is deleted from the database.

5.2 What doesn't work:

Overall, our system has all the functionality that is required, but in some cases it was done in a way which isn't optimal. One such case is that in the CustomerServiceStaff Class we create a Leg object. This class is in the presentation layer, so an object ideally shouldn't be created there. But we needed it as we had to save the data of the flight leg that was selected, so that the data could be displayed in the GUI when a ticket is booked.

We have a similar situation in the BookTicket Class, which is also in the presentation layer, where we create a Ticket object. We need the object there as when a new ticket is made in the GUI, it needs to be added to the Array of Tickets which is gotten when the program starts, as then it will show up in the search results if a new search is made and the program hasn't been restarted. It is also needed to be able to give the next created ticket the next free seat number, as otherwise all the tickets in one order would be given the same seat number.

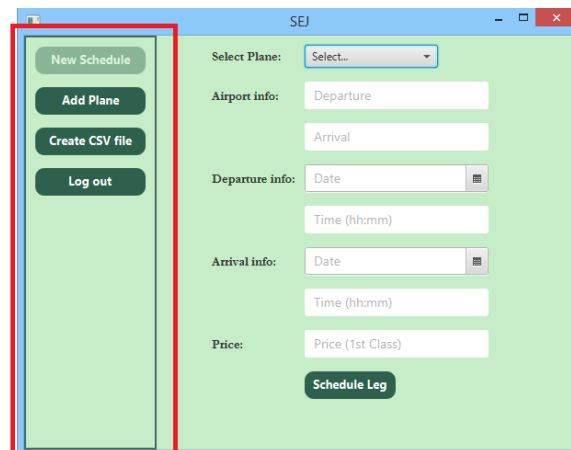
6. Construction of the GUI

We built our GUI according to the Golden Rules and the Gestalt Laws. We made sure the User is in control by making it so that 'Enter' can be pressed in certain textfields, where it makes sense to do so, instead of having to click a button. The user also doesn't really have to remember anything to use the system. For example, if a new schedule needs to be made and assigned to a plane, instead of the user having to know the name of the plane and typing it, we have a combobox with all the planes in the system, so that one can simply be selected.

We designed the GUI so that it is consistent in how it looks. Both the Admin menu and the Customer Services menu share a same style in the way they look, we made sure of this with the CSS. Considering the user of the system will be the employees, and the frequency of the usage will be high, so it should have a design which is neutral, not flashy. We have carefully chosen colors that fit well together. A light green color for the main background, which is neither boring nor annoying to look at in general. The buttons are a darker green, which fits well with the background color, and enriches the visual effect. We chose green as the main color because it is a relaxing color and it also fits with the orange color of the SEJ logo. The buttons are rounded to give the application a gentle soft appearance, and for the same reason we use a different font (Garamond) for the labels.

There are help facilities for the users, which are several alerts are used for information, like informing the user that what he wanted to do was successfully done, or to alert the user that some input is in the wrong format. The alerts are consistent in their appearance and it is clear what each alert is for.

We use the Law of Closure in our menus by keeping the buttons in one box, separated from the rest of the GUI, as can be seen below.



In addition, the Law of Proximity is used, in that the textfields and labels are aligned with equal spacing between them. There is also a flow to the way the textfields, no matter what option you choose, you fill out the textfields from top to bottom. This falls under the Gestalt Law of good Continuation.

The GUI is constructed by having one class per section, meaning 1 class for each for the Admin Section, Customer Services Section and one for the Login. We split up the functionality into multiple methods so it's easier to understand what happens where. For example, if a button is clicked, there is a method that has the functionality for that button, as shown in the below example.

```
buttonSavePlane.setOnAction(e -> savePlaneButtonClicked());
```

```

public static void savePlaneButtonClicked() {
    if(textFieldPlaneType.getText().isEmpty() || textFieldFirstClass.getText().isEmpty() || textFieldBusinessClass.getText().isEmpty() ||
        textFieldEconClass.getText().isEmpty()) {
        Alert alert = new Alert(Alert.AlertType.ERROR);
        alert.setTitle("ERROR");
        alert.setHeaderText(null);
        alert.setContentText("Please fill out all fields.");
        alert.showAndWait();
    } else {
        try {
            PlaneInfo.createPlane(textFieldPlaneType.getText(), Integer.parseInt(textFieldFirstClass.getText()),
                Integer.parseInt(textFieldBusinessClass.getText()), Integer.parseInt(textFieldEconClass.getText()));
            Alert alert = new Alert(Alert.AlertType.INFORMATION);
            alert.setTitle("Information");
            alert.setHeaderText(null);
            alert.setContentText(textFieldPlaneType.getText() + " has been added to the fleet.");
            alert.showAndWait();
        } catch (Exception e1) {
            e1.printStackTrace();
        }
    }

    textFieldPlaneType.clear();
    textFieldFirstClass.clear();
    textFieldBusinessClass.clear();
    textFieldEconClass.clear();
}

```

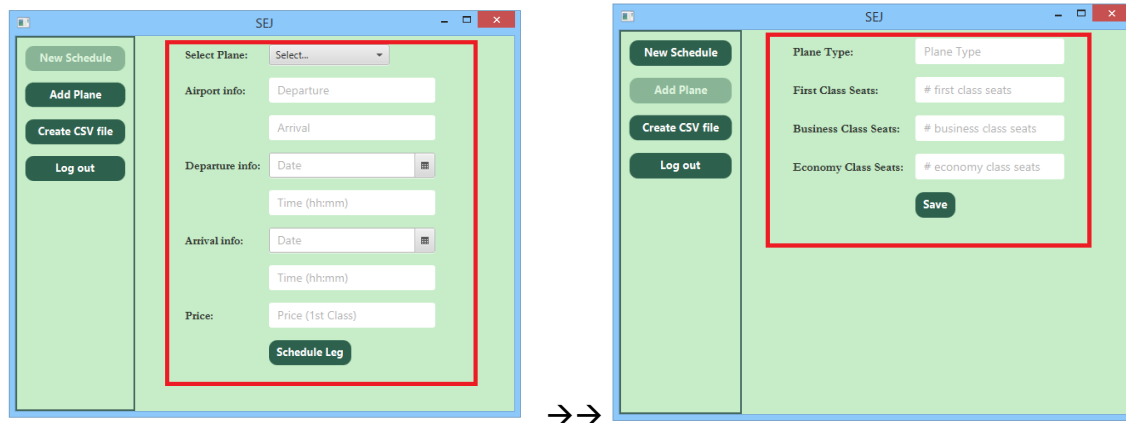
We built our GUI using multiple scenes, but only one stage. The login, Admin and Customer Services section each have their own scene, plus we use a scene for the booking of a ticket, as this has a lot of textfields and labels, so it required its own. The Admin and Customer Services scene have buttons on the left side of the menu, and when these buttons are clicked, the gridpane, which contains all the children needed for the functionality of that button, is generated.

```

gridPane.getChildren().addAll(labelPlaneType, textFieldPlaneType, labelFirstClass, textFieldFirstClass, labelBusinessClass,
    textFieldBusinessClass, labelEconClass, textFieldEconClass, buttonSavePlane);

layout.getChildren().add(gridPane);

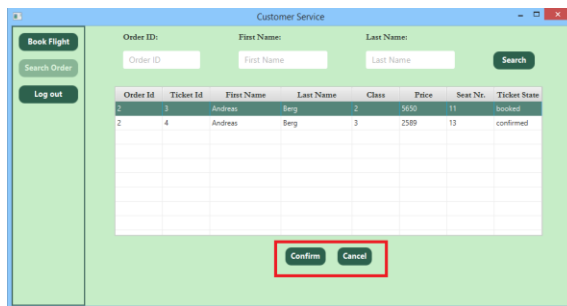
```



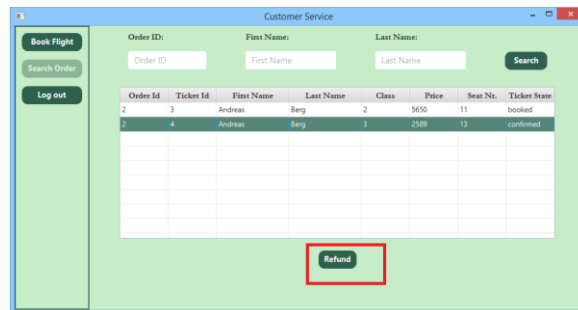
As can be seen above, when one of the buttons is clicked a new scene or stage isn't generated, only the gridpane is changed (the red box is there to show what is changed). We use only one gridpane per scene to avoid redundancy, so we simply remove all the children from the gridpane and add the new ones depending on what button is pressed.

In the Customer Services scene we made it so that one can search for flights or orders. To display the results, we use a tableview. We added buttons for refunding and confirming tickets, but the appropriate buttons only show up when you select one of the tickets, so that there is no confusion on which tickets can be refunded or confirmed. This means that if a ticket is confirmed, a refund button will appear below

the tableview, but not a Confirm button, as this is not needed in this case. You can also see this in the images below.

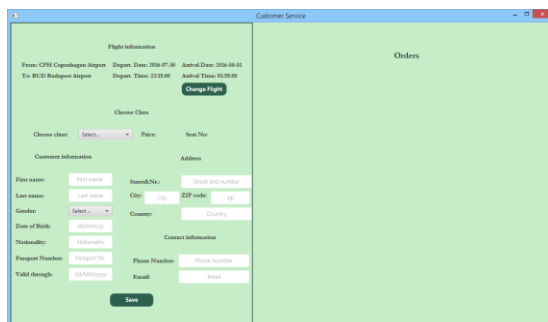


A Booked ticket selected



A Confirmed ticket selected

If a flight is selected and the Book flight button is pressed, it goes to the booking scene. It needed its own scene as it is quite big. Half the scene has all the textfields that need to be filled in with all the customers information, and it also displays the selected flights information. The other half of the scene is blank at this point with the exception of one label. However, when a ticket is saved, it is displayed in this area, and every other ticket created in this same order is added there as well. Once at least one ticket is made, buttons appear to make the order or Buy the tickets right then, and the total price is displayed.



The booking scene with no tickets



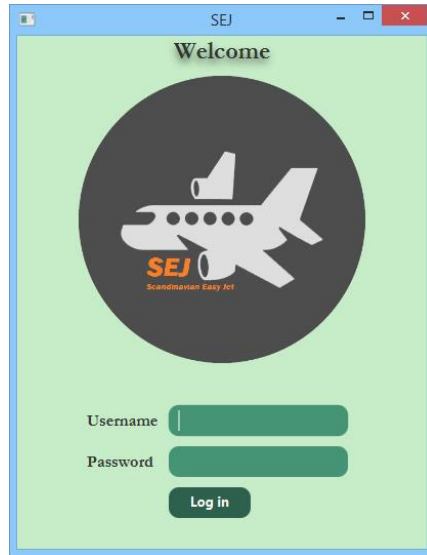
The booking scene with tickets

7. SEJ System User Manuel

7.1 Logging In

Depending on what you want to do, you need to login with either your admin username and password (if you have one) or your Customer Services username and password.

Using your Admin login information will take you to the Admin section, where you will be able to Add Flights, Add new Schedules or create CSV files of the data in the employees and planes databases. The Customer Service section allows you to book, confirm, refund or cancel tickets.



7.2 Admin section

If you logged in with your Admin login information, you will now be in the Admin section. You will have three options: Adding a plane, making a new schedule or making CSV files.

7.3 Adding a new flight schedule

If you want to add a new flight schedule, do the following steps:

1. Click the “New Schedule” button.
2. Select the Plane ID of the plane you wish to add the schedule to from the drop down box.
3. Fill in the Airport info fields. You must start with a three-letter abbreviation, followed by the name of the airport, such as ‘CPH Copenhagen Airport’.
4. Fill out the Departure and Arrival fields. Click on the little icon in the date text field and select a date. In the time field, you must input the time in the format indicated, such as ‘12:15’.
5. Fill out the price field. You must enter the price for a 1st Class ticket, as the prices for the other classes will be calculated based on a percentage of this price.
6. Click the “Schedule Leg” button. The flight leg is now saved. You can keep on adding legs to that plane, or change the flight ID to add legs to a new plane.

Note: all fields must be filled out, otherwise you will get an error message.

The screenshot shows a web application window titled "Admin". On the left is a sidebar with buttons: "New Schedule", "Add Plane", "Create CSV file", and "Log out". The main area contains a form for adding a plane. It starts with a "Select Plane:" dropdown menu. Below that is "Airport info:" with "Departure" and "Arrival" text inputs. Then "Departure info:" with "Date" and "Time (hh:mm)" inputs. Similarly, "Arrival info:" with "Date" and "Time (hh:mm)" inputs. At the bottom is a "Price:" section with a "Price (1st Class)" input and a "Schedule Log" button.

7.4 Adding a plane

If you want to add a plane in the system, do the following steps:

1. Click the “Add plane” button on the right side of the menu.
2. Fill in the Plane Type field, for example ‘Airbus A330-250’.
3. Fill in the Class fields. You must input the number of seats that there are in the plane for that Class. Only numeric characters are accepted in these fields. If a plane does not have a Class, input ‘0’.
4. Click the “Save” button. The plane is now saved in the system.

Note: all fields must be filled out, otherwise you will get an error message.

This screenshot shows the same "Admin" window but with a different form layout for adding a plane. It features a sidebar with "New Schedule", "Add Plane", "Create CSV file", and "Log out" buttons. The main form area has a "Plane Type:" label with a text input field. Below that are three labels: "First Class Seats:", "Business Class Seats:", and "Economy Class Seats:", each followed by a text input field containing a placeholder "#". At the bottom of the form is a "Save" button.

7.5 Customer Services Section

If you logged in with your Customer Services login information, you will get the Customer Services menu. Here you will have two options: Book a flight or search for an order. You can find the all the functions that can be done in this section below.

The screenshot shows a web application window titled "Customer Service". On the left is a sidebar with three buttons: "Book Flight", "Search Order", and "Log out". The main area contains a search form with fields for "From:" (City), "To:" (City), and "Departure date:" (Departure). A "Search" button is to the right. Below the form is a table with columns: "From", "To", "Dep. Date", "Dep. Time", "Arr. Date", and "Arr. Time". The table is currently empty, displaying the message "No content in table".

7.6 Booking a ticket

1. Click the "Book Flight" button.
2. Fill out all of the fields. In the From/To fields you must use the city's abbreviation in order for the system to be able to find available tickets, such as BUD or CPH.
3. Select the flight in which you want to book the ticket and click "Book Ticket". You will now get the Book Ticket section. You can see how it looks below.
4. At the top you will see the flight information of the selected flight. If you picked the wrong one, you can click "Change Flight" to pick another flight.
5. Choose the Class, the price and which seat will be displayed right next to it.
6. Fill in the customer's information, address and contact information. Fill in the data in the format indicated in the text fields.
- Note:** all fields must be filled out, otherwise you will get an error message.
7. Click "Save". The ticket will be shown on the right side of the screen. You can repeat the above steps to add more tickets to the order as needed.
8. When you have made all the tickets needed, click the "Make Order" button (The button will only be there if there is at least one order).

The screenshot shows the "Book Ticket" section of the "Customer Service" application. It is divided into two main panels. The left panel contains the booking form, and the right panel is titled "Orders" and is currently empty.

Flight Information:

From: CPH Copenhagen Airport To: BUD Budapest Airport
Depar. Date: 2016-07-30 Depar. Time: 22:15:00 Arrival Date: 2016-08-01 Arrival Time: 05:15:00
[Change Flight](#)

Choose Class:

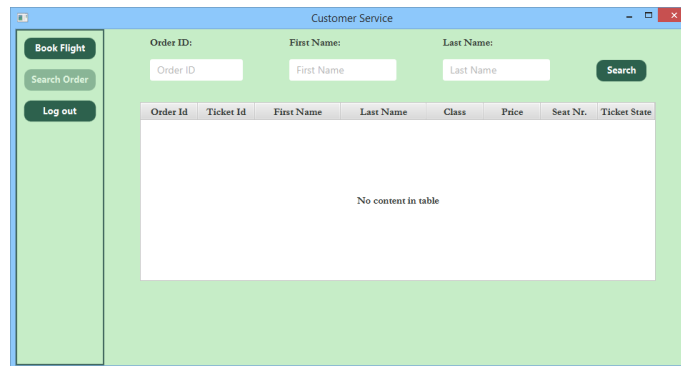
Choose class: Price: Seat No:

Customer Information:

First name: Surname: Email and number:
Last name: City: ZIP code:
Gender: Country:
Date of Birth: Nationality:
Passport Number: Phone Number:
Valid through: Email: Email:
[Save](#)

7.7 Searching for an order

1. Click the “Search Order” button.
2. Fill out the necessary fields. You need to either put the order ID OR the first and last name. Nothing else is accepted.
3. Click “Search”. All the orders that match your input will appear on a table.



The screenshot shows a web application window titled "Customer Service". On the left is a sidebar with three buttons: "Book Flight", "Search Order", and "Log out". The main area contains a search form with three input fields: "Order ID:", "First Name:", and "Last Name:". Below these fields is a "Search" button. Under the search form is a table with the following headers: "Order Id", "Ticket Id", "First Name", "Last Name", "Class", "Price", "Seat Nr.", and "Ticket State". The table body is empty, displaying the message "No content in table".

7.8 Confirming a Booked ticket

1. Click the “Search Order” button and follow the steps of the ‘Searching for an order’ section above, in order to find the ticket.
2. Select the Booked ticket in the table. The Ticket State field should say ‘Booked’.
3. Click the “Confirm” button. A message will appear that the ticket is now confirmed.

7.9 Cancelling a Booked ticket

1. Click the “Search Order” button and follow the steps of the ‘Searching for an order’ section above, in order to find the ticket.
2. Select the Booked ticket which you want to cancel from the table.
3. Click the “Cancel” button. A message will appear asking if you are sure that you want to cancel the ticket. Press “Yes”. The ticket is now cancelled.

Conclusion

In this report, the designing and building of the flight system has been documented. The flight system has all the functionality that was required from our project. It can add planes and schedule flights, which is what we needed it to do for the Administration part of the system. For the Customer Services part, it can search for flights and orders, and book or buy tickets. These tickets can also be refunded if needed. It also has an option to export two of the tables from the database (Employees and Planes) as CSV files.

Overall, the process to build the system was smooth. We used Unified Process, so we built the program over several iterations. Each iteration we went through the disciplines such as requirements, analysis, design and construction. We used use cases to analyze the interaction between an actor and the system, and we made diagrams to help us understand the structure of the program, such as a Class diagram and a Domain Model. The sequence diagram and state machine have helped us to detect methods, understand the relationship between objects in time sequence, or the transformation of the states of an object.

Bibliography:

https://en.wikipedia.org/wiki/Airbus_A380#Overview

http://www.seatguru.com/airlines/Emirates_Airlines/Emirates_Airlines_Airbus_A380.php

<http://aviation.stackexchange.com>

<http://stackoverflow.com>

<http://code.makery.ch/>

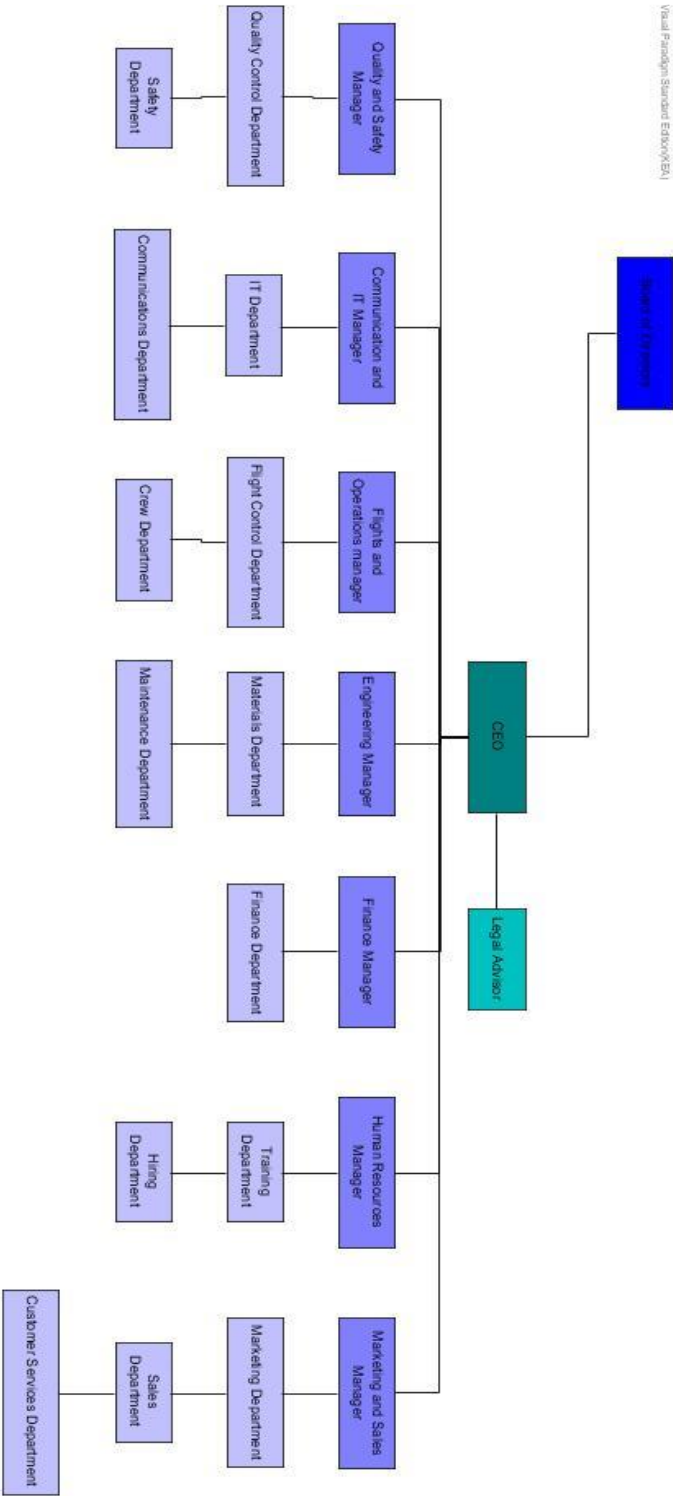
Craig Larman: Applying UML and Patterns, ISBN-13: 007-6092037224

Joel Murach: Murach's MySQL, ISBN 978-1-890774-68-4

Carl Dea: JavaFX 8 - Introduction by Example, ISBN13: 978-1-4302-6460-6

Appendix

Appendix 1: Organization Structure Suggestion



Appendix 2: Inception Phase Artifacts

Glossary

Actor - a role that a user takes when invoking a use case.

Aggregation - A “has-a” relationship (e.g. A car has a driver).

Association - An overall relation between classes.

ArrayList - A dynamic array that can grow as needed.

Attribute - a significant piece of data owned by a Class, often containing values describing each instance of the class.

Class diagram - a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes and the relationships between the classes.

Composition - An “is part of” relationship (strong association, e.g. Engine is part of Car).

Dependency - a dependency exists between two defined elements if a change to the definition of one would result in a change to the other.

Domain Model - A description of things in the real world.

Design pattern - In software engineering, a design pattern is a general repeatable solution to a commonly occurring problem in software design.

Diagram - a visual representation of a subset of features of a UML Model.

Event - when it occurs on an Object it may cause a Transition in a State machine diagram.

GRASP (General Responsibility Assignment Software Patterns) - consist of guidelines for assigning responsibility to classes and objects in object-oriented design.

Inheritance - A way to form new classes based on existing classes taking on their attributes and behavior.

Inheritance (in UML) - An “is-a” relationship (e.g. Employee is a person)

Interaction Diagram - diagram that is used to describe some type of interactions among the different elements in the model.

Lifeline - indicates a participating Object or Part in a Sequence diagram. The Lifeline may show activation, Object creation, and Object deletion.

Message - a signal from one object (or similar entity) to another, often with parameters.

Object - a runtime instance of a Class.

Package - A package is a collection or grouping of related classes or of classes with related functionality.

Package diagram in the Unified Modeling Language - depicts the dependencies between the packages that make up a model.

Object Oriented Programming (OOP) - A programming language model organized around objects rather than "actions" and data rather than logic

Return - a reply that may be issued from a Method following a Message.

Role - description of the part played in an Association by one of the Classes in the Association.

Scenario - a narrative describing foreseeable interactions.

Sequence diagram - describes the Messages sent between a number of participating Objects in a Scenario.

System Sequence Diagram (SSD) - Sequence Diagram that shows, for a particular scenario of a use case, the events that external actors generate, their order, and inter-system events.

State machine diagram - describes the lifetime behaviour of a single Object in terms of in which State it exists and the Transition between those States.

Use Case - Text Document with step by step instructions.

Use Case Diagram - Graphical representation of a Use Case.

Unified Process (UP) - An iterative and incremental software development process framework.

Type - the options are: an elementary Value type such as integer, string, date, or Boolean or a Reference type defined in a Class.

Transition - movement from one State to another in a State machine diagram.

Workflow - Set of sequential steps which must be done to get a job done.

Use cases

UC #1: Add planes

Primary Actor: Flights and Operation Manager

Main Success Scenario:

6. The manager opens the system and logs in as manager
7. The system verifies the identity of the manager
8. Successfully logged in, the manager chooses the function "add planes" in the menu bar
9. The system asks for the input of the plane information which needs to be added
10. The manager types in the information and choose to save
11. Plane(s) is(are) added

UC #2: Add classes

Primary Actor: Flights and Operation Manager

Main Success Scenario:

1. The manager opens the system and logs in as manager
2. The system verifies the identity of the manager
3. Successfully logged in, the manager chooses the function “add classes” in the menu bar
4. The system asks for the input of the information
5. The manager types in the information and choose to save
6. Classes are added

UC #3: Schedule flights

Primary Actor: Flights and Operation Manager

Main Success Scenario:

7. The manager opens the system and logs in as manager
8. The system verifies the identity of the manager
9. Successfully logged in, the manager chooses the function “schedule flights” in the menu bar
10. The system asks for the necessary input, fx choose a date and time for the flight
11. The manager types in the information needed
12. The flight is scheduled

Appendix 3: Artifacts – Iteration 1 - Elaboration Phase

Use Cases

UC #1: Add planes

Primary Actor: Flights and Operation Manager

Main Success Scenario:

1. The manager opens the system and logs in as manager
2. The system verifies the identity of the manager
3. Successfully logged in, the manager chooses the function “add planes” in the menu bar
4. The system asks for the input of the plane information which needs to be added
5. The manager types in the information and choose to save
6. Plane(s) is(are) added

UC #2: Add classes

Primary Actor: Flights and Operation Manager

Main Success Scenario:

7. The manager opens the system and logs in as manager
8. The system verifies the identity of the manager
9. Successfully logged in, the manager chooses the function “add classes” in the menu bar
10. The system asks for the input of the information
11. The manager types in the information and choose to save
12. Classes are added

UC #3: Schedule flights

Primary Actor: Flights and Operation Manager

Main Success Scenario:

1. The manager opens the system and logs in as manager
2. The system verifies the identity of the manager
3. Successfully logged in, the manager chooses the function “schedule flights” in the menu bar
4. The system asks for the necessary input, fx choose a date and time for the flight
5. The manager types in the information needed
6. The flight is scheduled

UC # 4: Book flight

Primary Actor: Sales and Marketing manager

Main Success Scenario:

1. The staff logs in to the system.
2. The system verifies the identity of the staff.
3. The staff goes to the book flight section of the system.
4. The system asks for input of flight data.
5. The sales staff inputs the customer’s information.
6. The flight is booked and saved in the database.
7. The staff closes the system.

UC # 5: Cancel flight

Primary Actor: Customer Services staff

Main Success Scenario:

1. The staff logs in to the system.
2. The system verifies the identity of the staff.

3. The staff finds the customer in the system.
4. The staff cancels the customer's ticket in the system.
5. The seat is marked as free in the system.
6. The staff logs out.

UC # 6: Search schedule

Primary Actor: Customer Services staff

Main Success Scenario:

7. The staff logs in to the system.
8. The system verifies the identity of the staff.
9. The staff selects the search function.
10. The system asks for the info to be searched.
11. The staff inputs the search info (for example flight dates, destination etc)
12. The system displays the data.
13. The staff logs off.

UC # 7: Refund ticket

Primary Actor: Customer Services staff

Main Success Scenario:

1. The staff logs in to the system.
2. The system verifies the identity of the staff.
3. The staff selects the tickets option from the menu.
4. The system returns the ticket options.
5. The staff selects the Refund ticket option.
6. The staff searches for the ticket id.
7. The staff selects the Refund ticket option.
8. The system refunds the ticket
9. The staff logs out of the system.

UC # 8: Buy flight

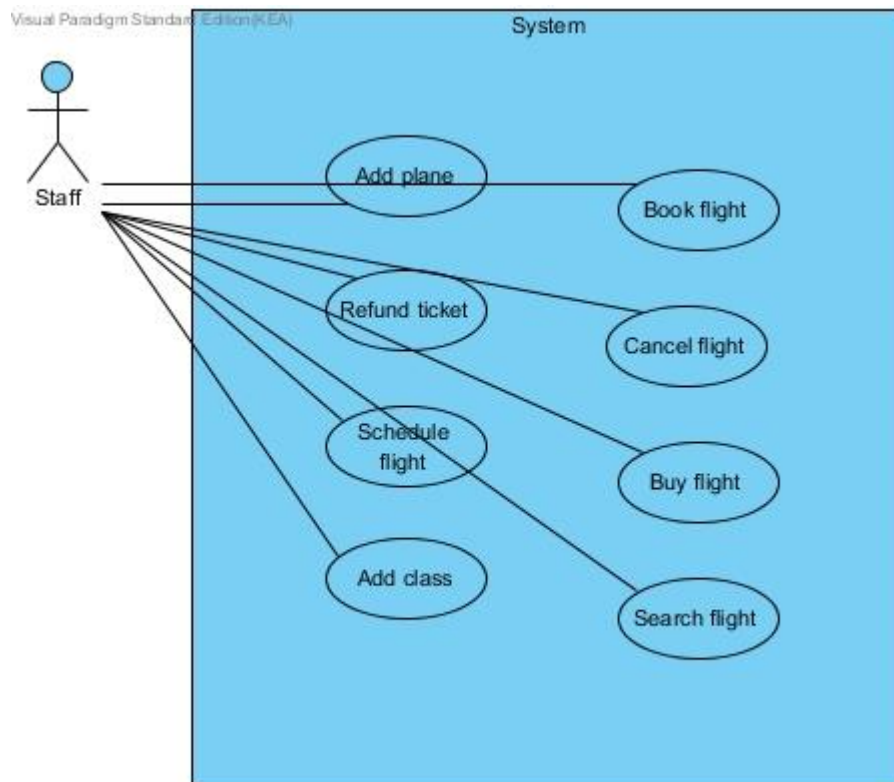
Primary Actor: Customer Services staff

Main Success Scenario:

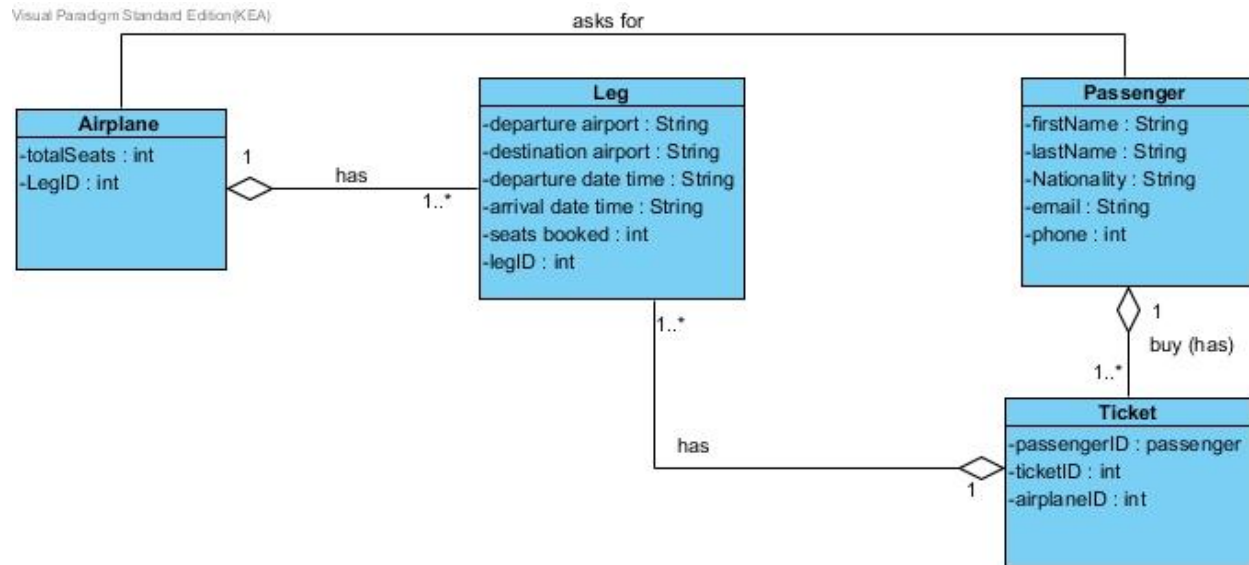
1. The staff logs in to the system.
2. The system verifies the identity of the staff.

3. Staff selects the tickets option.
4. System asks for flight data input.
5. The staff inputs the flight the customer wants and finds it.
6. The system requests customer info input.
7. The staff inputs the requested data.
8. The system returns a receipt for the ticket.
9. The staff logs off.

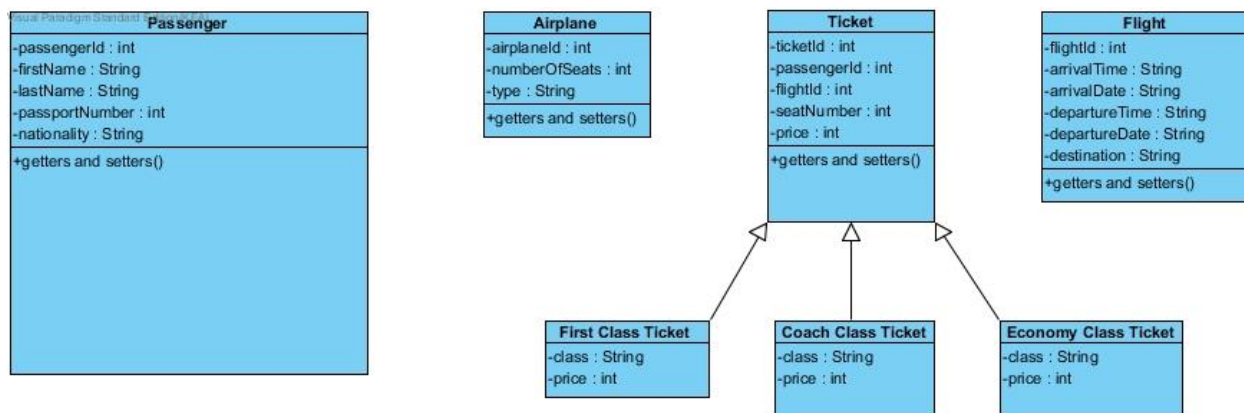
Use Case Diagram:



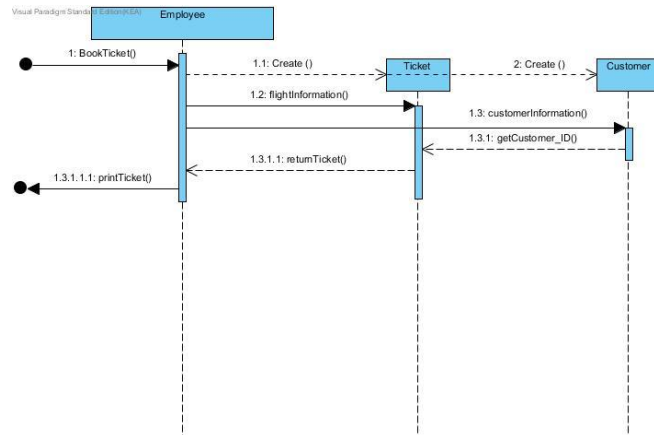
Domain Model:



Class Diagram:



Sequence Diagram for “Book Ticket”:



UC # 3: Book a ticket – Casual, it will become fully dressed

Primary Actor: Customer Service Department Employee

Main Success Scenario:

1. The employee opens the system and logs in with username and password.
2. The employee gets into the operation menu, chooses the operation “Book Ticket”.
3. The system asks for the information of the passenger and flight.

Situation A: The employee types in:

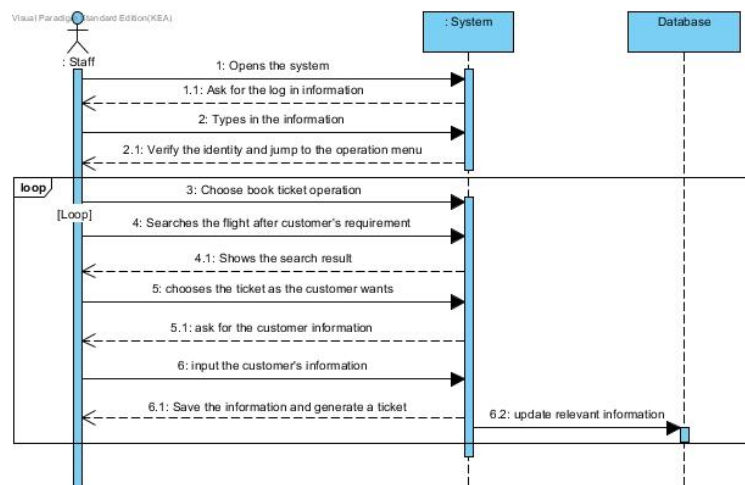
- required information for the flight: airplane number, departure airport, destination airport, departure date and time, arrival date and time, class, seat number
- required information of the customer: first name, last name, gender, birthday, nationality, passport number, email, phone number.

Situation B: The employee searches the flight after the customer's requirements

- The employee uses the search tool of the system, fills the search conditions
- The system shows the flights which satisfy the search condition
- The employee chooses among the flights, clicks the selected one, and chooses to book it
- The system asks for the information of the customer after the flight is chosen
- The employee types in the customer's information

4. Customer and flight information is saved and seat information is updated accordingly (the seat becomes booked or confirmed).
5. The ticket is booked.

System Sequence Diagram for “Book Ticket”:



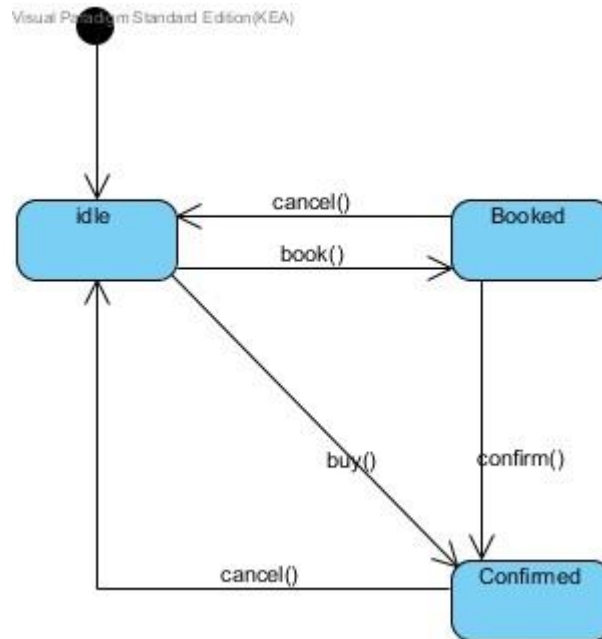
UC # 3: Book a ticket

Primary Actor: Customer Service Department Employee

Main Success Scenario:

1. The employee opens the system and logs in with username and password.
2. The employee gets into the operation menu, chooses the operation “Book Ticket”.
3. The staff searches flight after the customer's requirements: departure date, departure place, arrival date, arrival place, ticket price (range from high to low or the opposite) and/or ticket type (class).
4. The system shows the flights that satisfy the search condition(s).
5. The customer tells the staff which flight meets best his requirements and the staff chooses that one.
6. The system asks for customer's information: first name, last name, gender, birthday, age, nationality, passport number, passport expiration date, address, email, phone number.
7. The staff inputs requested information.
8. Customer and flight information is saved and seat information is updated accordingly (the seat becomes booked).
9. The ticket is booked.

State Machine for “Ticket”:



Appendix 4: Artifacts – Iteration 2- Elaboration Phase

Use Cases

UC #1: Add Planes

Primary Actor: Flights and Operation Manager

Main Success Scenario:

1. The manager opens the system and logs in with username and password.
2. The system verifies the identity of the manager, if it is valid, the staff can log in and use the application.
3. The manager chooses the function “Add Planes” in the menu bar.
4. The system asks for necessary input: airplane ID, maximum number of seats, plane type and the legs/flights assigned to that airplane. The legs can be added at a later time as well.
5. The manager types in the information and chooses to save.
6. Plane(s) is(are) added.

UC #2: Schedule Flights

Primary Actor: Flights and Operation Manager

Main Success Scenario:

1. The manager opens the system and logs in as manager with username and password.
2. The system verifies the identity of the manager, if it is valid, the staff can log in and use the application.
3. The manager chooses the function “schedule flights”.
4. The system asks for the necessary input: flight number, departure time, date and place (airport), arrival time, date and place(airport), and classes that are on the flight (first class, business and/or economy).
5. The manager types in the requested information.
6. The flight is scheduled.

UC # 3: Book Ticket

Primary Actor: Customer Service Department Staff

Main Success Scenario:

1. The staff opens the system.
2. The system asks for the staff’s user name, and password.
3. The system verifies the identity of the staff, if it is valid, the staff can log in and use the application.
4. The staff gets into the operation menu, chooses the operation “Book Ticket”.
5. The system asks for the information of the passenger (passengers).
 - 3) Situation A: The staff knows exactly what information to type in:
 - Type in the required information of the flight:
Airplane number, departure airport, destination airport, departure date time, arrival date time, class, seat number
 - Type in the required information of the customer:
First name, Last name, Gender, Birthday, Nationality, Passport number, Email, Phone number
 - 4) Situation B: the staff searches the flight after the customer’s requirements
 - The staff uses the search tool of the system, fills the search conditions
 - The system shows the flights which satisfies the search condition
 - The staff chooses among the flights, clicks the selected one, and chooses to book it
 - The system asks for the information of the customer after the flight is chosen
 - The staff types in the customer’s information and confirm
 - A ticket is booked.
6. The system shows message “A ticket is booked; the notification is sent to the customer”.
7. Customer’s information goes to the database, the ticket’s information goes to the database, airplane’s seat information is updated accordingly.

Alternative Flow:

1. At any time, if the system fails:
 - The system shows an error message, and restarts
 - Ask for the valid log in, restarts again
2. No available tickets left, no reservation can be done

UC # 4: Cancel ticket

Primary Actor: Customer Services staff

Main Success Scenario:

1. The staff logs in to the system with username and password.
2. The system verifies the identity of the staff, if it is valid, the staff can log in and use the application.
3. The staff selects the ticket options from the menu.
4. The system shows the ticket operation window.
5. The staff searches ticket by ticket ID or customer name.
6. The staff chooses the operation of "Cancel Ticket". The ticket is cancelled.
7. The seat is marked as free in the system and database.

UC # 5: Search Schedule

Primary Actor: Customer Services staff

Main Success Scenario:

1. The staff logs in to the system with username and password.
2. The system verifies the identity of the staff, if it is valid, the staff can log in and use the application.
3. The staff selects the search function.
4. The system asks for search condition: departure date, departure place, arrival date, arrival place, ticket price (range from high to low or the opposite) and/or ticket type (class).
5. The staff inputs the search info.
6. The system displays the data.

UC # 6: Refund ticket

Primary Actor: Customer Services staff

Main Success Scenario:

1. The staff logs in to the system with username and password.

2. The system verifies the identity of the staff, if it is valid, the staff can log in and use the application.
3. The staff gets into the operation menu, chooses the operation “Buy Ticket”.
4. The staff selects the “Refund Ticket” option.
5. The system refunds the ticket, based on which type of ticket it is.
First class: 100% refund
Business class: 85% refund
Economy: 70% refund, and only if they are cancelled at least two weeks before the flight.
6. Once the ticket is refunded, the seat is marked as free in the system and the database again.

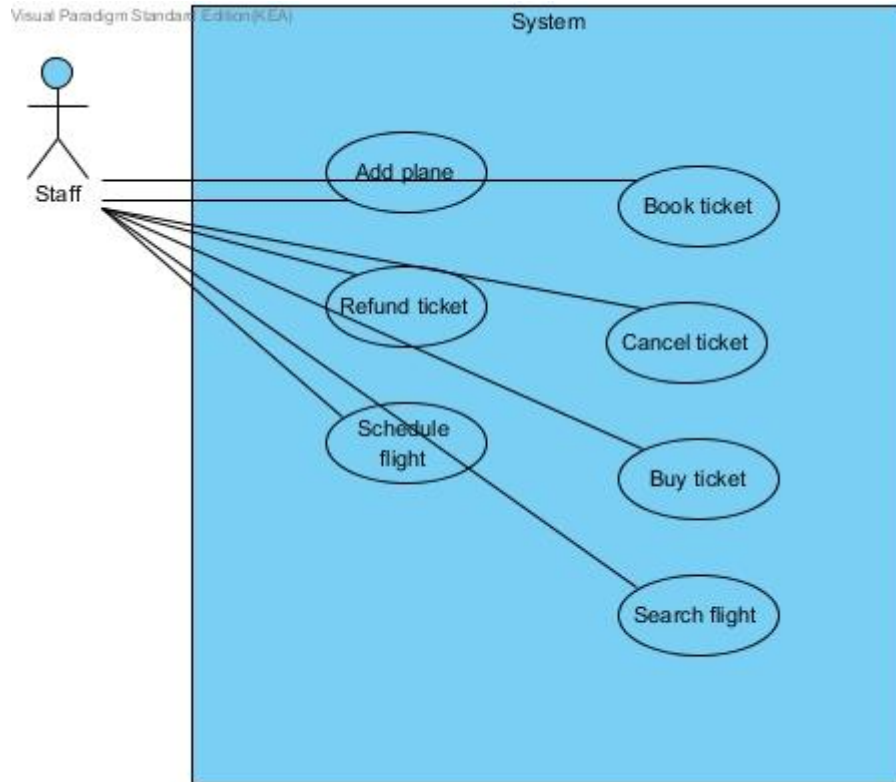
UC # 7: Buy Ticket

Primary Actor: Customer Services staff

Main Success Scenario:

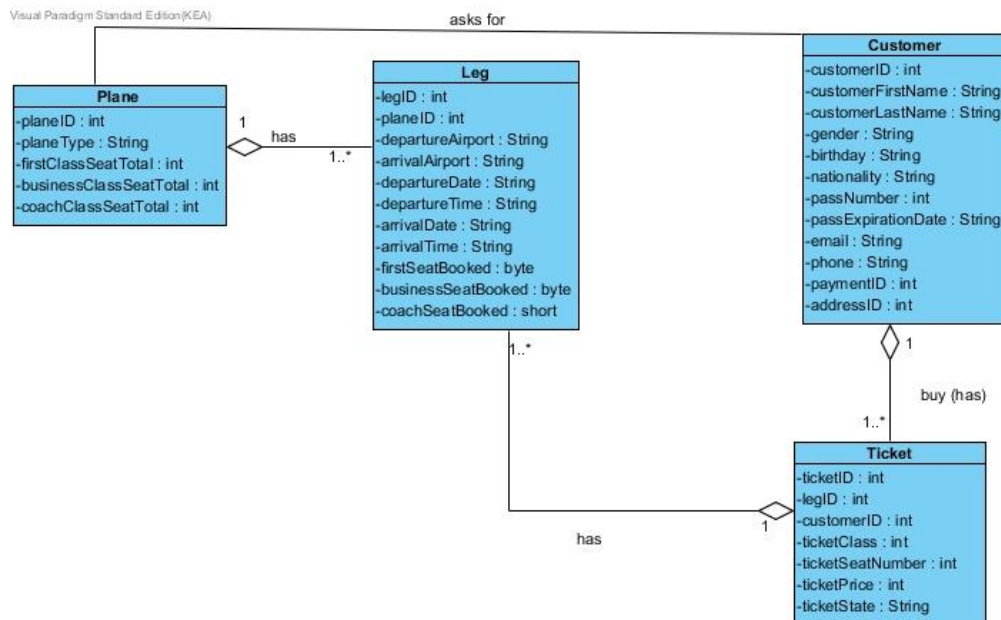
1. The staff logs in to the system with username and password.
2. The system verifies the identity of the staff, if it is valid, the staff can log in and use the application.
3. The staff gets into the operation menu, chooses the operation “Buy Ticket”.
4. Procedure is mainly the same as “Book Ticket” at this step.
5. Furthermore, the system asks for customer payment information: card number, expiration date and security number.
6. The staff inputs the requested data.
7. The system returns a receipt for the ticket.
8. The ticket becomes confirmed.

Use Case Diagram:

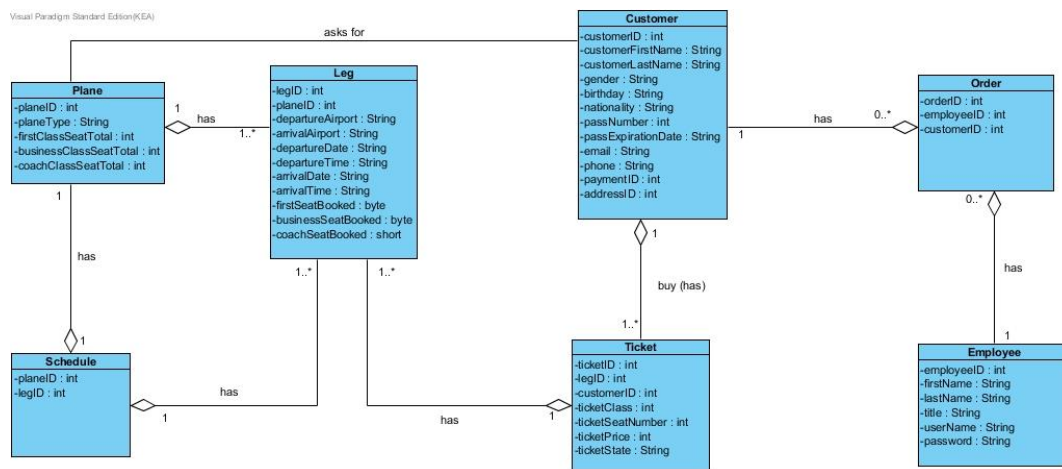


Domain Model:

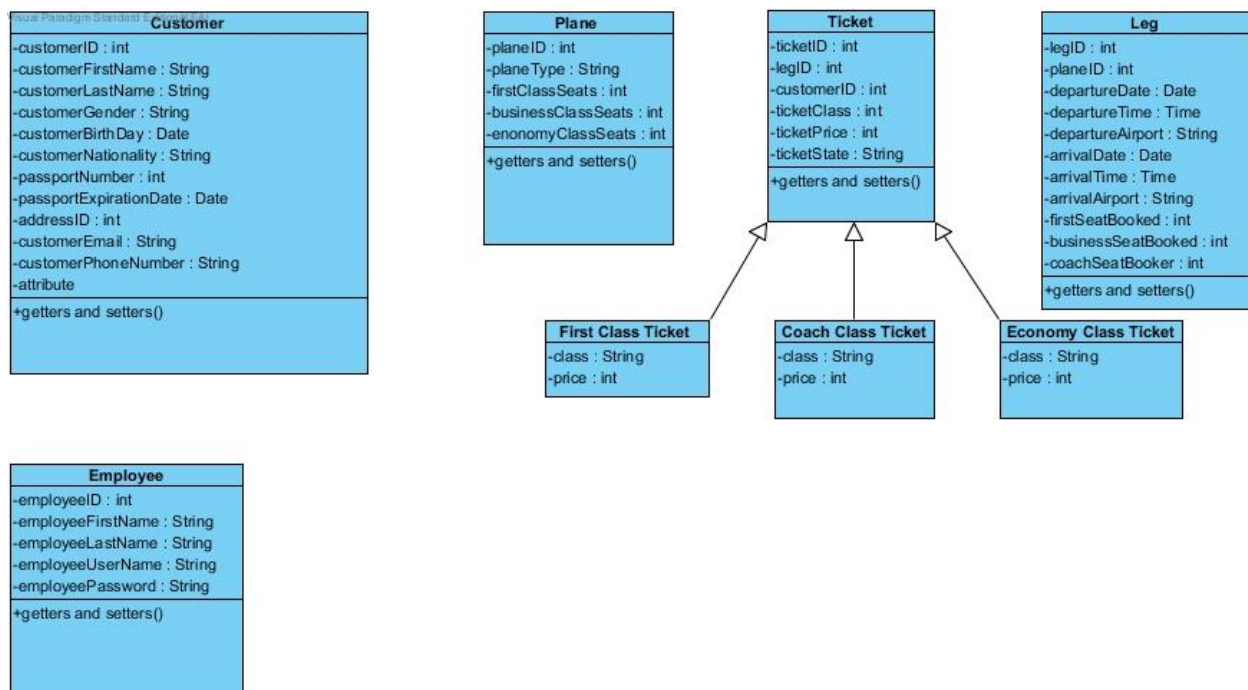
Updated after the 1st iteration



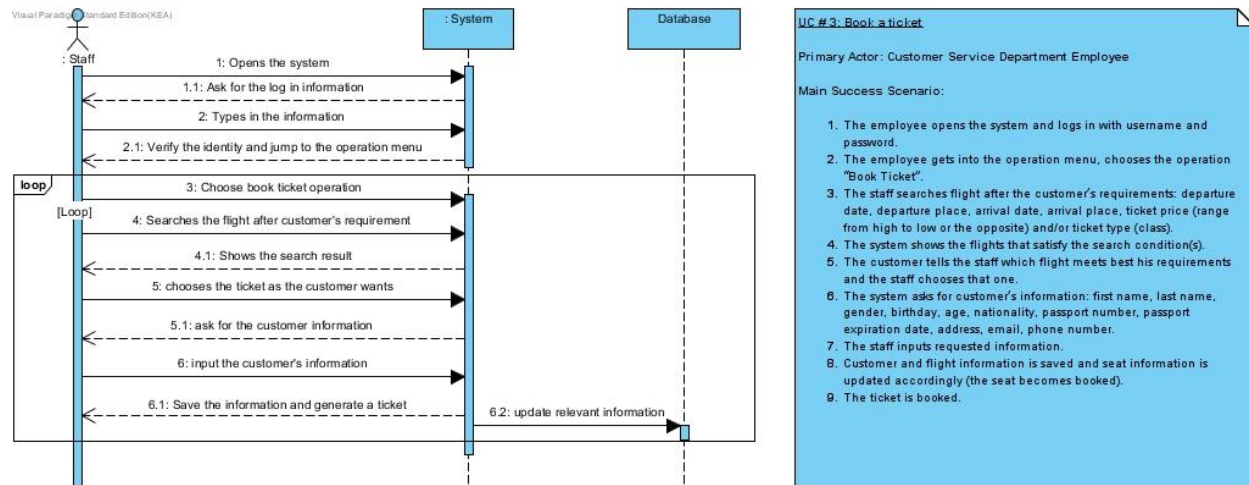
Updated at the end of the 2nd iteration



Class Diagram:



System Sequence Diagram:



Appendix 5: Artifacts – Iteration 3 - Elaboration Phase

Use Cases

UC #1: Add Planes - Brief

Primary Actor: Flights and Operation Manager

Main Success Scenario:

1. The manager opens the system and logs in with his username and password.
2. The system verifies the identity of the manager, if it is valid, the staff can log in and use the application.
3. The manager chooses the function "Add Planes" in the menu bar.
4. The system asks for necessary input: plane type, number of first class, business class and economy class seats.
5. The manager types in the information and chooses to save the plane.
6. The plane is saved to the database.

UC #2: Schedule Flights - Brief

Primary Actor: Flights and Operation Manager

Main Success Scenario:

1. The manager opens the system and logs in as manager with his username and password.
2. The system verifies the identity of the manager, if it is valid, the staff can log in and use the application.
3. The manager chooses the function “schedule flights”.
4. The system asks for the necessary input: airplane number, departure time, date and place (airport), arrival time, date and place(airport).
5. The manager types in the requested information.
6. The flight is scheduled.

UC # 3: Book a ticket- Fully Dressed

Scope: Book Ticket System

Level: User goal

Primary Actor: Customer Service Department Employee

Stakeholders and interests:

- Employee: wants to have an easy to use system when he wants to book a ticket.
- Customer: wants to book a ticket in a simple way.

Preconditions: The system is opened and ready to process

Postconditions (Success Guarantee):

- A ticket has been booked.
- A record for the ticket is created by the system, so the customers can be informed about the ticket details later.

Main Success Scenario:

1. The employee opens the system and logs in with username and password.
2. The employee gets into the operation menu, chooses the operation “Book Ticket”.
3. The staff searches flight after the customer’s requirements: departure date, departure place, arrival date, arrival place, ticket price (range from high to low or the opposite) and/or ticket type (class).
4. The system shows the flights that satisfy the search condition(s).
5. The customer tells the staff which flight meets best his requirements and the staff chooses that one.
6. The system asks for customer’s information: first name, last name, gender, birthday, age, nationality, passport number, passport expiration date, address, email, phone number.
7. The staff inputs requested information.
8. Customer and flight information is saved and seat information is updated accordingly (the seat becomes booked).
9. The ticket is booked.

Alternative Flow:

1. If the system fails:
 - The user restarts the system
 - Internet connection is checked
 - User asks for IT support
2. If identity is not confirmed:
 - The user checks entered data and write it right
 - Change password if needed
3. If there are no available seats left: reservation cannot be done.
4. If the staff inputs wrong data: the staff edits information and saves the ticket again.

Special Requirements:

- Device from which the system can be used (PC, Mac, Tablet, Phone)
- Platform (Windows, OS X, Android, iOS)
- Confirmation from customer when everything is done

Frequency of Occurrence:

- Every time a ticket has to be booked.

UC # 4: Cancel ticket - Brief**Primary Actor: Customer Services staff****Main Success Scenario:**

1. The staff logs in to the system with username and password.
2. The system verifies the identity of the staff, if it is valid, the staff can log in and use the application.
3. The staff selects the ticket options from the menu.
4. The system shows the ticket options.
5. The staff searches ticket by ticket ID or customer name.
6. The staff chooses the operation of “Cancel Ticket”. The ticket is cancelled.
7. The seat is marked as free in the system and database.

UC # 5: Search Schedule - Brief**Primary Actor: Customer Services staff****Main Success Scenario:**

1. The staff logs in to the system with username and password.
2. The system verifies the identity of the staff, if it is valid, the staff can log in and use the application.

3. The staff selects the search function.
4. The system asks for search condition: departure date, departure place, arrival date, arrival place, ticket price (range from high to low or the opposite) and/or ticket type (class).
5. The staff inputs the search info.
6. The system displays the data.

UC # 6: Refund ticket - Casual

Primary Actor: Customer Services staff

Main Success Scenario:

1. The staff logs in to the system with username and password.
2. The system verifies the identity of the staff, if it is valid, the staff can log in and use the application.
3. The staff gets into the operation menu, chooses the ticket options.
4. The staff searches the ticket by ticket ID or customer name.
5. The staff selects the “Refund Ticket” option.
6. The system refunds the ticket, based on which type of ticket it is.
First class: 100% refund
Business class: 85% refund
Economy: 70% refund, and only if they are cancelled at least two weeks before the flight.
7. Once the ticket is refunded, the seat is marked as free in the system and the database again.

Alternative Flow:

1. If the system fails:
 - The user restarts the system
 - Internet connection is checked
 - User asks for IT support
2. If identity is not confirmed:
 - The user checks entered data and write it right
 - Change password if needed
3. If the wrong amount of money is refunded to the customer:
 - The transaction is cancelled and the right amount is sent to the customer.

UC # 7: Buy Ticket - Casual

Primary Actor: Customer Services staff

Main Success Scenario:

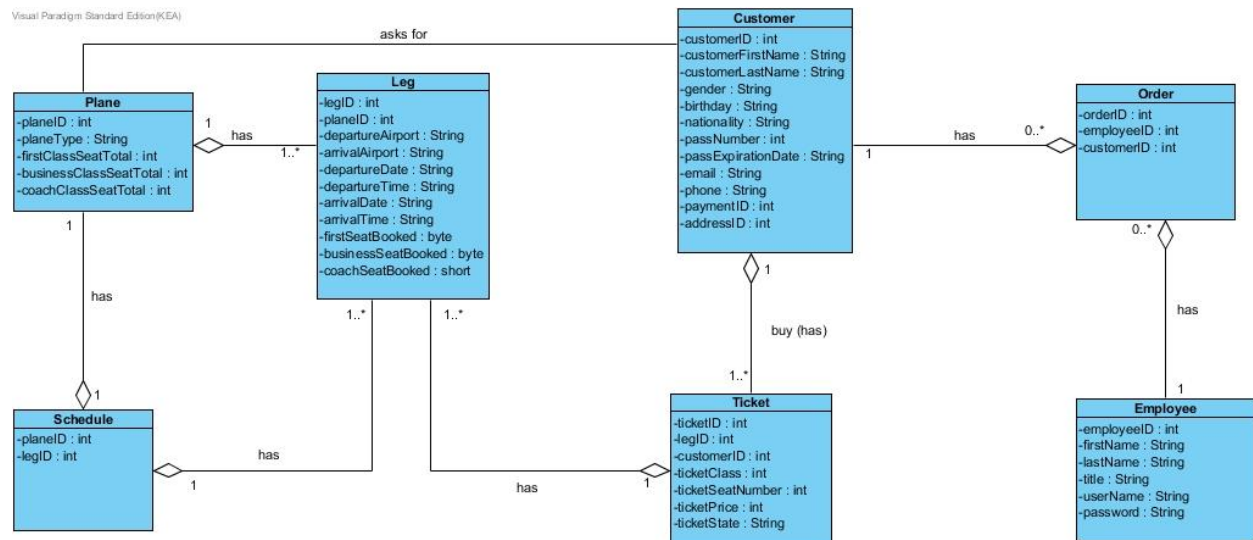
1. The staff logs in to the system with username and password.
2. The system verifies the identity of the staff, if it is valid, the staff can log in and use the application.
3. The staff gets into the operation menu, chooses the operation “Buy Ticket”.

4. Procedure is mainly the same as “Book Ticket” at this step.
5. Furthermore, the system asks for customer payment information: card number, expiration date and security number.
6. The staff inputs the requested data.
7. The system returns a receipt for the ticket.
8. The ticket becomes confirmed.

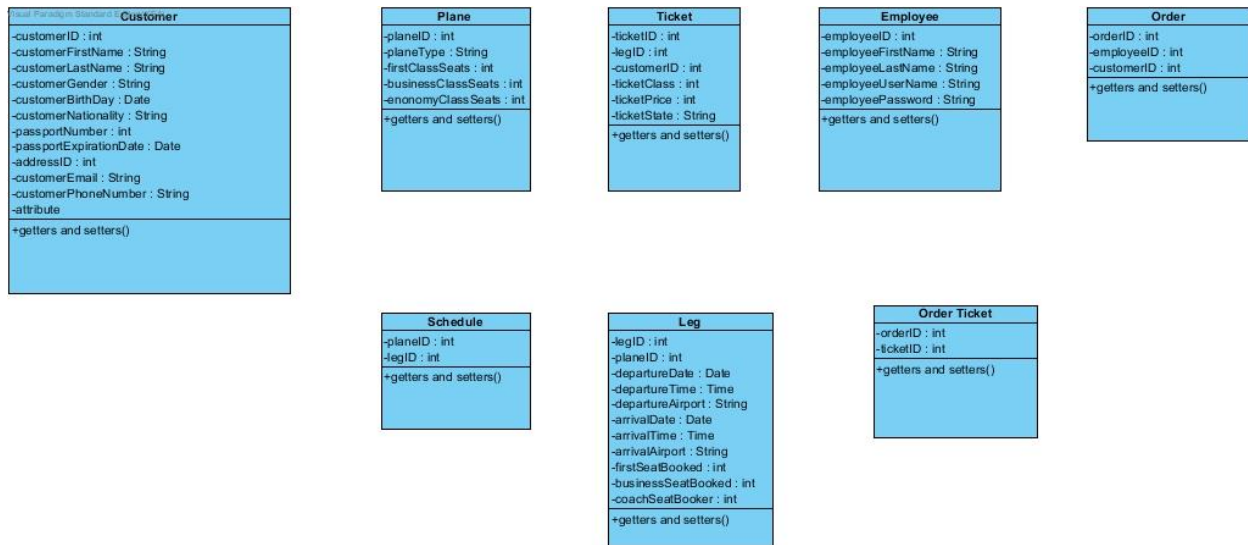
Alternative Flow:

1. If the system fails:
 - The user restarts the system
 - Internet connection is checked
 - User asks for IT support
2. If identity is not confirmed:
 - The user checks entered data and write it right
 - Change password if needed
3. If there are no available seats left: the ticket cannot be bought.
4. If the staff inputs wrong data: the staff edits information and saves the ticket again.

Domain Model:



Class Diagram:



Appendix 6: Artifacts – Iteration 4 - Elaboration Phase

Use Cases

UC #1: Add Planes - Brief

Primary Actor: Flights and Operation Manager

Main Success Scenario:

1. The manager opens the system and logs in with his username and password.
2. The system verifies the identity of the manager, if it is valid, the staff can log in and use the application.
3. The manager chooses the function “Add Planes” in the menu bar.
4. The system asks for necessary input: plane type, number of first class, business class and economy class seats.
5. The manager types in the information and chooses to save the plane.
6. The plane is saved to the database.

UC #2: Schedule Flights - Brief

Primary Actor: Flights and Operation Manager

Main Success Scenario:

1. The manager opens the system and logs in as manager with his username and password.
2. The system verifies the identity of the manager, if it is valid, the staff can log in and use the application.
3. The manager chooses the function “schedule flights”.
4. The system asks for the necessary input: airplane number, departure time, date and place (airport), arrival time, date and place(airport).
5. The manager types in the requested information.
6. The flight is scheduled.

UC # 3: Book a ticket- Fully Dressed

Scope: Book Ticket System

Level: User goal

Primary Actor: Customer Service Department Employee

Stakeholders and interests:

- Employee: wants to have an easy to use system when he wants to book a ticket.
- Customer: wants to book a ticket in a simple way.

Preconditions: The system is opened and ready to process

Postconditions (Success Guarantee):

- A ticket has been booked.
- A record for the ticket is created by the system, so the customers can be informed about the ticket details later.

Main Success Scenario:

1. The employee opens the system and logs in with username and password.
2. The employee gets into the operation menu, chooses the operation "Book Ticket".
3. The staff searches flight after the customer's requirements: departure date, departure place, arrival date, arrival place, ticket price (range from high to low or the opposite) and/or ticket type (class).
4. The system shows the flights that satisfy the search condition(s).
5. The customer tells the staff which flight meets best his requirements and the staff chooses that one.
6. The system asks for customer's information: first name, last name, gender, birthday, age, nationality, passport number, passport expiration date, address, email, phone number.
7. The staff inputs requested information.
8. Customer and flight information is saved and seat information is updated accordingly (the seat becomes booked).
9. The ticket is booked.

Alternative Flow:

1. If the system fails:
 - The user restarts the system
 - Internet connection is checked
 - User asks for IT support
2. If identity is not confirmed:
 - The user checks entered data and write it right
 - Change password if needed
3. If there are no available seats left: reservation cannot be done.
4. If the staff inputs wrong data: the staff edits information and saves the ticket again.

Special Requirements:

- Device from which the system can be used (PC, Mac, Tablet, Phone)
- Platform (Windows, OS X, Android, iOS)
- Confirmation from customer when everything is done

Frequency of Occurrence:

- Every time a ticket has to be booked.

UC # 4: Cancel ticket - Brief

Primary Actor: Customer Services staff

Main Success Scenario:

1. The staff logs in to the system with username and password.
2. The system verifies the identity of the staff, if it is valid, the staff can log in and use the application.
3. The staff selects the ticket options from the menu.
4. The system shows the ticket options.
5. The staff searches ticket by ticket ID or customer name.
6. The staff chooses the operation of "Cancel Ticket". The ticket is cancelled.
7. The seat is marked as free in the system and database.

UC # 5: Search Schedule - Brief

Primary Actor: Customer Services staff

Main Success Scenario:

1. The staff logs in to the system with username and password.
2. The system verifies the identity of the staff, if it is valid, the staff can log in and use the application.
3. The staff selects the search function.
4. The system asks for search condition: departure date, departure place, arrival date, arrival place, ticket price (range from high to low or the opposite) and/or ticket type (class).
5. The staff inputs the search info.
6. The system displays the data.

UC # 6: Refund ticket - Casual

Primary Actor: Customer Services staff

Main Success Scenario:

1. The staff logs in to the system with username and password.
2. The system verifies the identity of the staff, if it is valid, the staff can log in and use the application.
3. The staff gets into the operation menu, chooses the ticket options.
4. The staff searches the ticket by ticket ID or customer name.
5. The staff selects the "Refund Ticket" option.
6. The system refunds the ticket, based on which type of ticket it is.

First class: 100% refund

Business class: 85% refund

Economy: 70% refund, and only if they are cancelled at least two weeks before the flight.

7. Once the ticket is refunded, the seat is marked as free in the system and the database again.

Alternative Flow:

1. If the system fails:
 - The user restarts the system
 - Internet connection is checked
 - User asks for IT support
2. If identity is not confirmed:
 - The user checks entered data and write it right
 - Change password if needed
3. If the wrong amount of money is refunded to the customer:
 - The transaction is cancelled and the right amount is sent to the customer.

UC # 7: Buy Ticket - Casual

Primary Actor: Customer Services staff

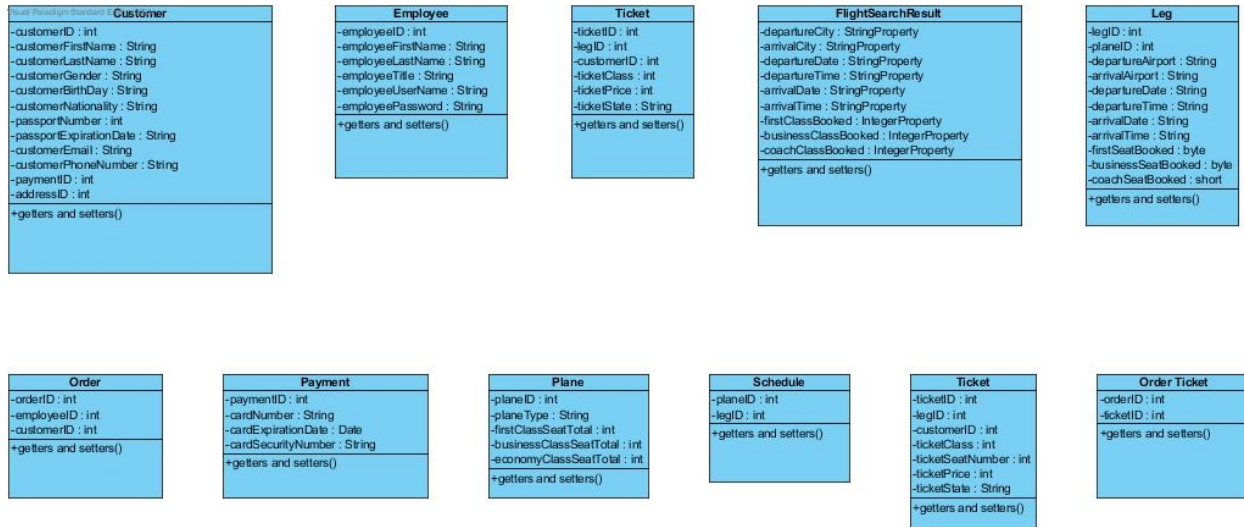
Main Success Scenario:

1. The staff logs in to the system with username and password.
2. The system verifies the identity of the staff, if it is valid, the staff can log in and use the application.
3. The staff gets into the operation menu, chooses the operation "Buy Ticket".
4. Procedure is mainly the same as "Book Ticket" at this step.
5. Furthermore, the system asks for customer payment information: card number, expiration date and security number.
6. The staff inputs the requested data.
7. The system returns a receipt for the ticket.
8. The ticket becomes confirmed.

Alternative Flow:

1. If the system fails:
 - The user restarts the system
 - Internet connection is checked
 - User asks for IT support
2. If identity is not confirmed:
 - The user checks entered data and write it right
 - Change password if needed
3. If there are no available seats left: the ticket cannot be bought.
4. If the staff inputs wrong data: the staff edits information and saves the ticket again.

Class Diagram:



Appendix 7: Artifacts – Iteration 5 - Elaboration Phase

Use Cases

UC #1: Add Planes - Brief

Primary Actor: Flights and Operation Department Manager

Main Success Scenario:

1. The manager opens the system and logs in with his username and password.
2. The system verifies the identity of the manager, if it is valid, the staff can log in and use the application.
3. The manager chooses the function “Add Planes” in the menu bar.
4. The system asks for necessary input: plane type, number of first class, business class and economy class seats.
5. The manager types in the information and chooses to save the plane.
6. The plane is added and saved to the database.

UC #2: Schedule Flights - Brief

Primary Actor: Flights and Operation Department Manager

Main Success Scenario:

1. The manager opens the system and logs in as manager with his username and password.
2. The system verifies the identity of the manager, if it is valid, the staff can log in and use the application.
3. The manager chooses the function “schedule flights”.
4. The system asks for the necessary input: airplane number, departure time, date and place (airport), arrival time, date and place(airport).
5. The manager types in the requested information.
6. The flight is scheduled.

UC # 3: Book ticket- Fully Dressed

Scope: Book Ticket System

Level: User goal

Primary Actor: Customer Service Department Employee

Stakeholders and interests:

- The company owners: want to have a good system which is in line with the company business strategy, making their work easier, keeping the competitive advantages, and reducing unnecessary costs.
- Employee: wants to have an easy to use system when he wants to book a ticket.
- Customer: wants to book a ticket in a simple way.

Preconditions: The system is opened and ready to process, a customer wants to buy ticket.

Postconditions (Success Guarantee):

- A ticket has been booked.
- A record for the ticket is created by the system, so does a customer. Related information is updated in the system.

Main Success Scenario:

1. The employee opens the system and logs in using his/her username and password.
2. The employee gets into the operation menu, chooses the operation “Book Ticket”.
3. The staff searches flight after the customer’s requirements: departure place, arrival place, departure date.
4. The system shows the flights that satisfy the search condition(s).
5. The customer tells the staff which flight meets best his requirements and the staff chooses that one.
6. The staff chooses the ticket and to book for the customer, the system asks for the customer’s information: first name, last name, gender, birthday, age, nationality, passport number, passport expiration date, address, email, phone number.
7. The staff inputs requested information.
8. Customer and flight information is saved and the seat information is updated accordingly (the seat becomes booked).

9. The ticket is booked.

Alternative Flow:

1. If the system fails:
 - The user restarts the system.
 - Internet connection is checked.
 - User asks for IT support.
2. If user's identity is not confirmed:
 - The user needs to enter username and password again.
 - Change password if it is needed.
 - The system sends an email to the user to recover/verify the account.
3. If there are no available seats left: reservation cannot be done.
4. If the staff inputs wrong data: the staff edits information and saves the ticket again, notifying customer if needed.

Special Requirements:

- Device from which the system can be used (PC, Mac, Tablet)
- Platform (Windows, OS X)
- Confirmation from customer when everything is done

Frequency of Occurrence:

- Every time a ticket has to be booked.
- Can be many times per day.

UC # 4: Cancel ticket - Brief

Primary Actor: Customer Service Department Employee

Main Success Scenario:

1. The staff logs in to the system with username and password.
2. The system verifies the identity of the staff, if it is valid, the staff can log in and use the application.
3. The staff searches the ticket from the customer service menu by searching order .
4. The system shows the search results, meaning an order contains ticket/tickets.
5. The staff finds the ticket and clicks it.
6. The staff chooses the operation of "Cancel Ticket". The ticket is cancelled.
7. The seat is marked as free in the system and database.

UC # 5: Refund ticket - Casual

Primary Actor: Customer Service Department Employee

Main Success Scenario:

1. The staff logs in to the system with username and password.
2. The system verifies the identity of the staff, if it is valid, the staff can log in and use the application.
3. The staff first needs to find the ticket by searching order, customer name.
4. The staff selects the “refund” option on the ticket.
5. The system refunds the ticket, based on which type of ticket it is.
First class: 100% refund
Business class: 85% refund
Economy: 70% refund, and only if they are cancelled at least two weeks before the flight.
6. Once the ticket is refunded, the seat is marked as free in the system and the database again.

Alternative Flow:

1. If the system fails:
 - The user restarts the system.
 - Internet connection is checked.
 - User asks for IT support.
2. If identity is not confirmed:
 - The user needs to enter username and password again.
 - Change password if it is needed.
 - The system sends an email to the user to recover/verify the account.
3. If the wrong amount of money is refunded to the customer:
 - The transaction is cancelled and the right amount is sent to the customer.

UC # 6: Buy Ticket - Casual

Primary Actor: Customer Service Department Employee

Main Success Scenario:

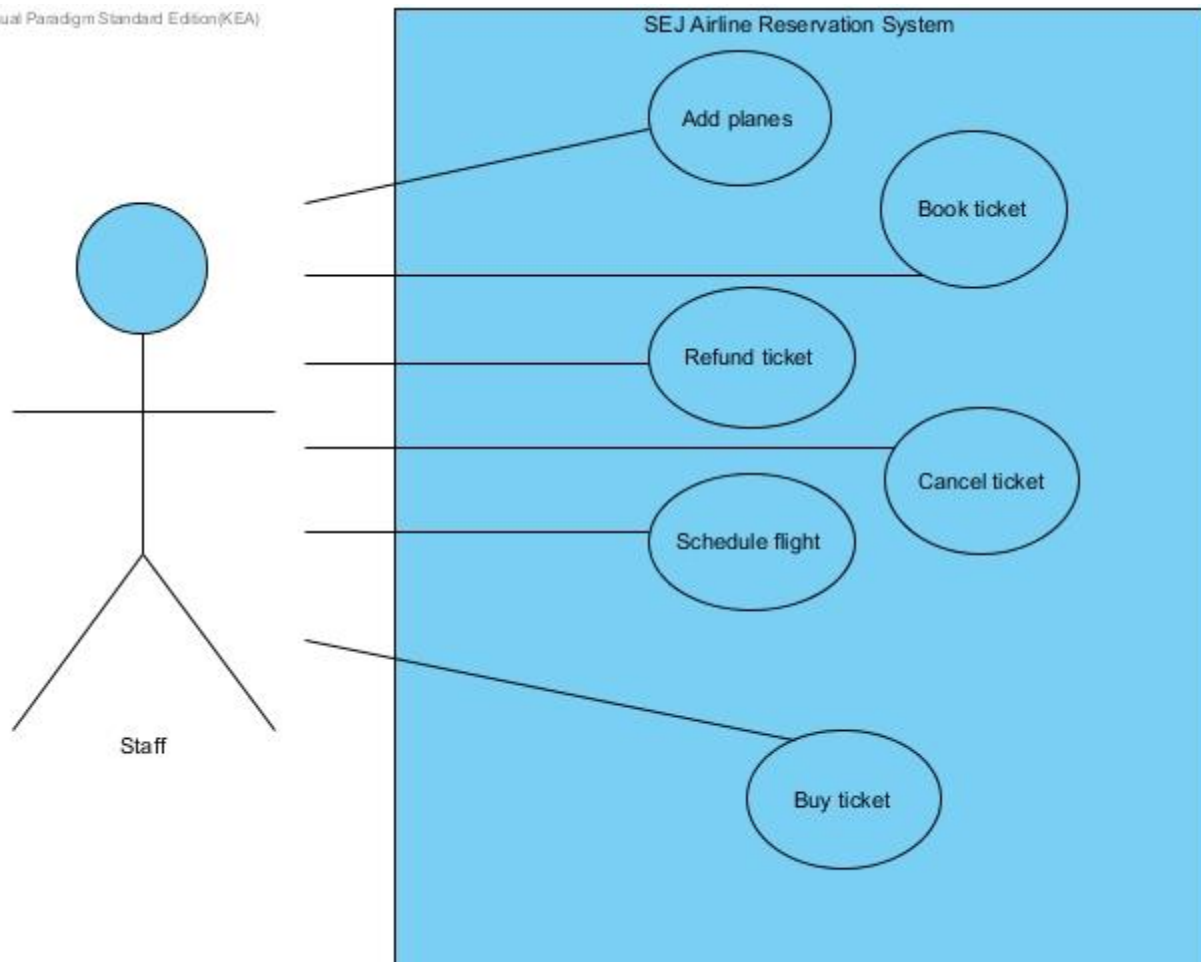
1. The staff logs in to the system with username and password.
2. The system verifies the identity of the staff, if it is valid, the staff can log in and use the application.
3. The staff gets into the operation menu, chooses the operation “Book Ticket”.
4. Procedure is mainly the same as “Book Ticket” at this step, meaning the staff has to first search the ticket according to the customer’s needs, and finds the ticket that satisfy the customer.
5. Furthermore, the system asks for customer payment information: card number, expiration date and security number.
6. The staff inputs the requested data.
7. The system returns a receipt for the ticket.
8. The ticket becomes confirmed, related information is now changed in the system.

Alternative Flow:

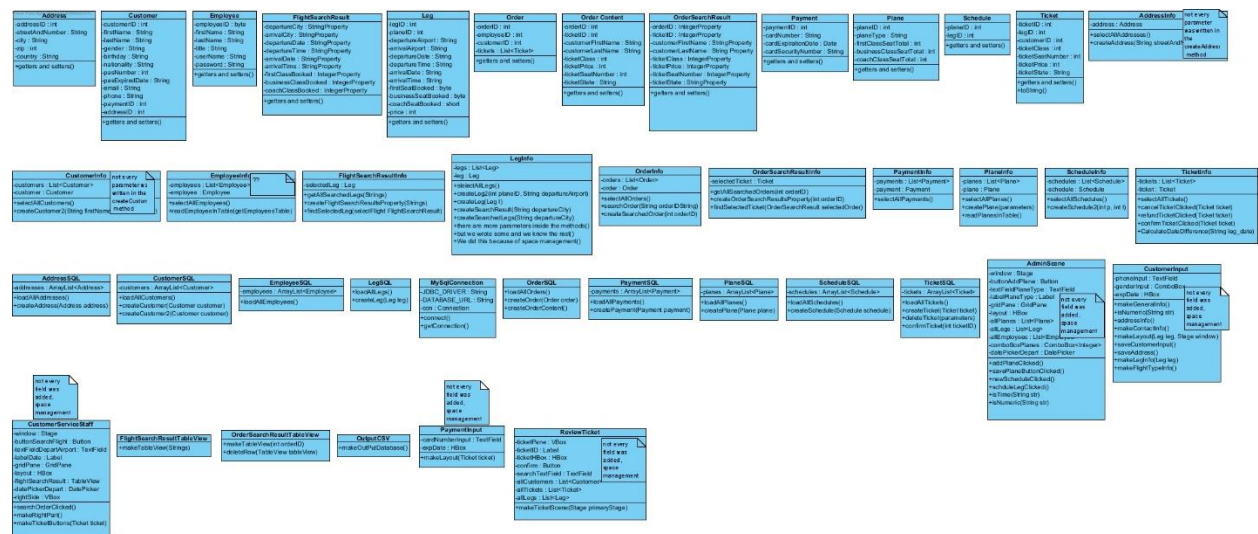
1. If the system fails:
 - The user restarts the system.
 - Internet connection is checked.
 - User asks for IT support.
2. If identity is not confirmed:
 - The user needs to enter username and password again.
 - Change password if it is needed.
 - The system sends an email to the user to recover/verify the account.
3. If there are no available seats left: the ticket cannot be bought.
4. If the staff inputs wrong data: the staff edits information and saves the ticket again.

Use Case Diagram:

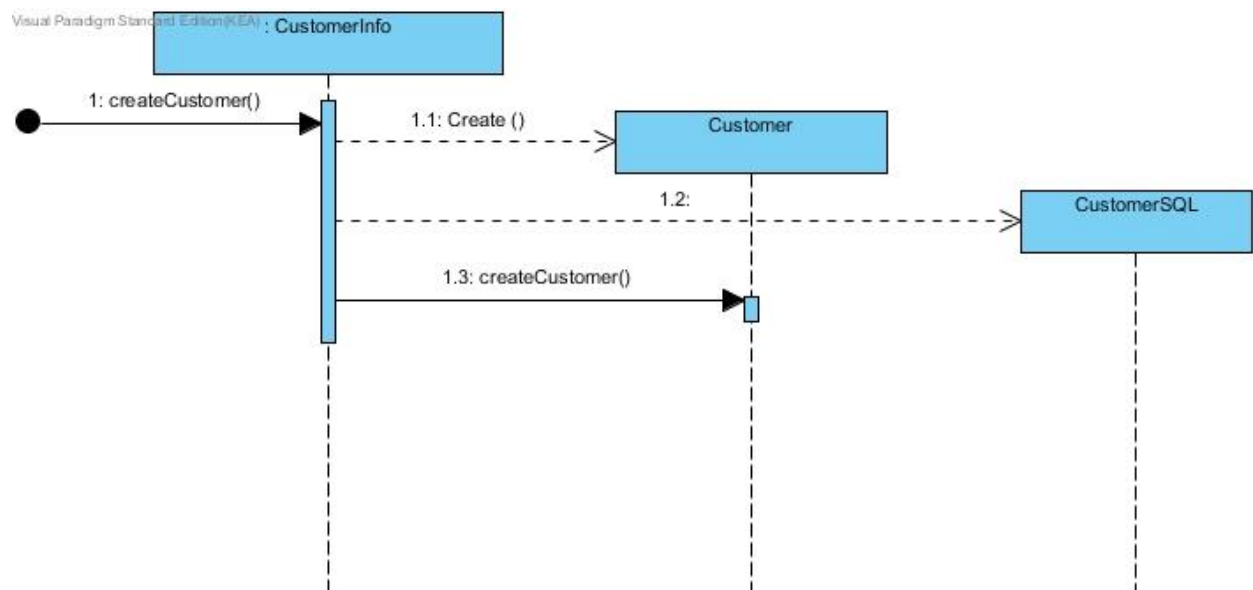
Visual Paradigm Standard Edition(KEA)



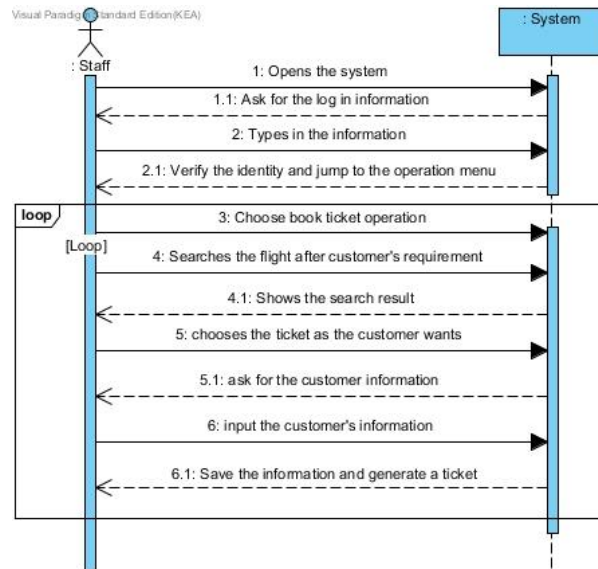
Class Diagram:



Sequence Diagram for “Create Customer”:



System Sequence Diagram:



UC # 3: Book a ticket

Primary Actor: Customer Service Department Employee

Main Success Scenario:

1. The employee opens the system and logs in with username and password.
2. The employee gets into the operation menu, chooses the operation "Book Ticket".
3. The staff searches flight after the customer's requirements: departure date, departure place, arrival date, arrival place, ticket price (range from high to low or the opposite) and/or ticket type (class).
4. The system shows the flights that satisfy the search condition(s).
5. The customer tells the staff which flight meets best his requirements and the staff chooses that one.
6. The system asks for customer's information: first name, last name, gender, birthday, age, nationality, passport number, passport expiration date, address, email, phone number.
7. The staff inputs requested information.
8. Customer and flight information is saved and seat information is updated accordingly (the seat becomes booked).
9. The ticket is booked.

Appendix 8: SQL Script for creating the database

#DDL: create SEJ database

DROP DATABASE IF EXISTS SEJ;

CREATE DATABASE SEJ;

USE SEJ;

create table 'addresses'

CREATE TABLE addresses

(

address_id	INT	PRIMARY KEY	NOT
NULL UNIQUE	AUTO_INCREMENT,		

address_street_number	VARCHAR(50)	NOT NULL,
-----------------------	-------------	-----------

```

address_city          VARCHAR(50)          NOT NULL,
address_zip_code      INT                  NOT NULL,
address_country       VARCHAR(50)          NOT NULL
);

```

create table 'customers'payments

CREATE TABLE customers

```

(
    customer_id        INT                  PRIMARY KEY      NOT NULL
    UNIQUE             AUTO_INCREMENT,
    customer_first_name VARCHAR(45) NOT NULL,
    customer_last_name  VARCHAR(45) NOT NULL,
    customer_gender     VARCHAR(45) NOT NULL,
    customer_birthday   VARCHAR(45) NOT NULL,
    customer_nationality VARCHAR(45) NOT NULL,
    passport_number     INT                  NOT NULL,
    passport_expiration_date VARCHAR(45) NOT NULL,
    address_id          INT                  NOT NULL,
    customer_email      VARCHAR(45) NOT NULL,
    customer_phone_number VARCHAR(45) NOT NULL,

```

CONSTRAINT customers_fk_addresses

FOREIGN KEY (address_id)

REFERENCES addresses (address_id)

```
);
```

```
# create table 'employees'
```

```
CREATE TABLE employees
```

```
(
```

```
    employee_id          INT          PRIMARY KEY          NOT NULL          UNIQUE  
    AUTO_INCREMENT,
```

```
    employee_first_name  VARCHAR(50)          NOT NULL,
```

```
    employee_last_name   VARCHAR(50)          NOT NULL,
```

```
    employee_title       VARCHAR(50)  NOT NULL,
```

```
    employee_username    VARCHAR(50)          NOT NULL          UNIQUE,
```

```
    employee_password    VARCHAR(50)          NOT NULL          UNIQUE
```

```
);
```

```
# create table 'planes'
```

```
CREATE TABLE planes
```

```
(
```

```
    plane_id            INT          PRIMARY KEY  
    NOT NULL          UNIQUE          AUTO_INCREMENT,
```

```
    plane_type          VARCHAR(45)          NOT NULL,
```

```
    first_class_seats   INT,
```

```
    business_class_seats INT,
```

```
    economy_class_seats INT
```

```
);
```

```
# create table 'legs'
```

```
CREATE TABLE legs
```

```
(
    leg_id            INT            PRIMARY KEY      NOT
    NULL             UNIQUE          AUTO_INCREMENT,
    departure_airport VARCHAR(60)     NOT NULL,
    arrival_airport   VARCHAR(60)     NOT NULL,
    departure_date     VARCHAR(60)     NOT NULL,
    departure_time     VARCHAR(60)     NOT NULL,
    arrival_date       VARCHAR(60)     NOT NULL,
    arrival_time       VARCHAR(60)     NOT NULL,
    first_seat_booked  INT              DEFAULT 0,
    business_seat_booked INT            DEFAULT 0,
    coach_seat_booked  INT              DEFAULT 0,
    price_first_class  INT              NOT NULL
);
```

```
# create table 'tickets'
```

```
CREATE TABLE tickets
```

```
(
    ticket_id         INT            PRIMARY KEY      NOT NULL
    UNIQUE            AUTO_INCREMENT,
    leg_id            INT            NOT NULL,
    customer_id       INT            NOT NULL,
    ticket_class       INT            NOT NULL,
```

```

ticket_seat_number INT NOT NULL,

ticket_price INT NOT NULL,

ticket_state VARCHAR(45) NOT NULL,

CONSTRAINT tickets_fk_legs

FOREIGN KEY (leg_id)

REFERENCES legs (leg_id),

CONSTRAINT tickets_fk_customers

FOREIGN KEY (customer_id)

REFERENCES customers (customer_id)

);

```

create table 'schedule'

```

CREATE TABLE schedules (

plane_id INT REFERENCES planes (plane_id) ON UPDATE CASCADE ON
DELETE CASCADE,

leg_id INT REFERENCES legs (leg_id) ON UPDATE CASCADE,

CONSTRAINT schedules_pkey PRIMARY KEY (plane_id, leg_id)

);

```

create table 'orders'

```

CREATE TABLE orders

(

order_id INT PRIMARY KEY NOT NULL UNIQUE
AUTO_INCREMENT,

employee_id INT NOT NULL,

```

```

customer_id          INT      NOT NULL,

CONSTRAINT orders_fk_employees

        FOREIGN KEY (employee_id)

        REFERENCES employees (employee_id),

CONSTRAINT orders_fk_customers

        FOREIGN KEY (customer_id)

        REFERENCES customers (customer_id)

);

# create table 'schedule'

CREATE TABLE orders_tickets (

    order_id          INT          REFERENCES   orders (order_id) ON UPDATE CASCADE ON
DELETE CASCADE,

    ticket_id         INT          REFERENCES   tickets (ticket_id) ON UPDATE CASCADE,

    CONSTRAINT orders_tickets_pkey PRIMARY KEY (order_id, ticket_id)

);

INSERT INTO addresses VALUES

        (DEFAULT, 'Gronjordsvej 1', 'Copenhagen', '2300', 'Denmark'),

        (DEFAULT, 'Amagerbrogade 151', 'Copenhagen', '2300', 'Denmark'),

        (DEFAULT, 'Amagerfælledvej 135', 'Copenhagen', '2300', 'Denmark'),

        (DEFAULT, 'Frederiksborggade 11', 'Copenhagen', '1360', 'Denmark'),

        (DEFAULT, 'Falkoner Alle 75', 'Frederiksberg', '2000', 'Denmark');

```

INSERT INTO customers VALUES

(DEFAULT, 'Martin', 'Lauren', 'Male', '08/10/1993', 'American', 45973175, '01/01/2020', 1, 'mar_lauren@yahoo.com', '71459637'),

(DEFAULT, 'Victor', 'Jensen', 'Male', '25/05/1990', 'Danish', 97314608, '01/03/2019', 2, 'victor_jen90@gmail.com', '51521305'),

(DEFAULT, 'Andreas', 'Berg', 'Male', '12/06/1985', 'Norwegian', 93479214, '01/05/2018', 3, 'berg.andreas@yahoo.com', '31614665'),

(DEFAULT, 'Emily', 'Haugen', 'Female', '12/09/1987', 'Danish', 65491279, '15/10/2020', 4, 'emily_haugen@gmail.com', '13513279');

INSERT INTO employees VALUES

(DEFAULT, 'John', 'Conan', 'admin', 'Johnny', 'conan1'),

(DEFAULT, 'Andrea', 'Beam', 'admin', 'Andy', 'beam1'),

(DEFAULT, 'Mach', 'Esch', 'customer_service', 'Mac', 'esch1'),

(DEFAULT, 'Garret', 'Fendley', 'customer_service', 'Gary', 'fen1'),

(DEFAULT, 'Ashlie', 'Harper', 'customer_service', 'Ashy', 'harper1');

INSERT INTO planes VALUES

(DEFAULT, 'Airbus A330-200', 12, 42, 183),

(DEFAULT, 'Boeing 777-200', 18, 49, 236),

(DEFAULT, 'Boeing 777-300', 18, 42, 420),

(DEFAULT, 'Airbus A340-300', 46, 28, 171),

(DEFAULT, 'Airbus A330-300', 34, 35, 195);

INSERT INTO legs VALUES

(DEFAULT, 'CPH Copenhagen Airport', 'OTP Bucharest Henri Coandă International Airport', '08/10/2016', '10:21', '08/10/2016', '12:33', 10, 25, 150, 1500),

(DEFAULT, 'CPH Copenhagen Airport', 'BER Berlin Tegel Airport', '01/06/2016', '06:00', '01/06/2016', '07:15', 12, 40, 201, 1000),

(DEFAULT, 'CPH Copenhagen Airport', 'VIE Vienna International Airport', '29/10/2016', '17:05', '29/10/2016', '18:55', 6, 30, 200, 1200),

(DEFAULT, 'CPH Copenhagen Airport', 'GOR Gorna Oryahovitsa Airport', '13/08/2016', '18:30', '13/08/2016', '21:00', 15, 28, 170, 1500),

(DEFAULT, 'CPH Copenhagen Airport', 'BUD Budapest Airport', '30/07/2016', '12:12', '30/07/2016', '14:45', 13, 28, 195, 1300),

(DEFAULT, 'CPH Copenhagen Airport', 'BUD Budapest Airport', '20/07/2016', '22:15', '20/07/2016', '01:55', 13, 28, 199, 1300),

(DEFAULT, 'CPH Copenhagen Airport', 'BUD Budapest Airport', '30/07/2016', '15:30', '30/07/2016', '19:45', 10, 20, 200, 1300),

(DEFAULT, 'CPH Copenhagen Airport', 'FRA Frankfurt Airport', '12/10/2016', '12:45', '12/10/2016', '13:50', 12, 26, 179, 1200),

(DEFAULT, 'CPH Copenhagen Airport', 'MUC Munich Airport', '15/09/2016', '06:15', '15/09/2016', '08:20', 4, 24, 124, 1200),

(DEFAULT, 'CPH Copenhagen Airport', 'MMA Montpellier–Méditerranée Airport', '10/10/2016', '18:50', '10/10/2016', '21:00', 8, 22, 118, 2000),

(DEFAULT, 'CPH Copenhagen Airport', 'OTP Bucharest Henri Coandă International Airport', '18/12/2016', '09:55', '18/12/2016', '12:15', 8, 16, 155, 1500),

(DEFAULT, 'CPH Copenhagen Airport', 'ORL Orly Airport', '24/12/2016', '12:25', '24/12/2016', '15:00', 12, 26, 100, 1600),

(DEFAULT, 'CPH Copenhagen Airport', 'BRU Brussels Airport', '03/07/2016', '05:48', '03/07/2016', '07:15', 12, 26, 123, 1500),

(DEFAULT, 'CPH Copenhagen Airport', 'BHX Birmingham International Airport', '19/08/2016', '17:35', '19/08/2016', '19:00', 16, 24, 180, 1600),

(DEFAULT, 'CPH Copenhagen Airport', 'BRS Bristol Airport', '15/06/2016', '18:00', '15/06/2016', '19:35', 12, 18, 199, 1600),

(DEFAULT, 'CPH Copenhagen Airport', 'EMA East Midlands Airport', '30/06/2016', '20:30', '30/06/2016', '22:00', 14, 28, 185, 1600),

(DEFAULT, 'CPH Copenhagen Airport', 'LTN Luton Airport', '09/08/2016', '08:15', '09/08/2016', '09:55', 8, 16, 231, 1800),

(DEFAULT, 'CPH Copenhagen Airport', 'MAN Manchester Airport', '10/10/2016', '10:35', '10/10/2016', '12:15', 12, 22, 178, 1500),

(DEFAULT, 'ABZ Aberdeen Airport', 'BFS Belfast International Airport', '21/11/2016', '13:37', '21/11/2016', '15:00', 16, 26, 199, 1000);

INSERT INTO tickets VALUES

(DEFAULT, 1, 1, 1, 199, 8320, 'booked'),

(DEFAULT, 2, 2, 1, 97, 8600, 'booked'),

(DEFAULT, 3, 3, 2, 11, 5650, 'booked'),

(DEFAULT, 4, 3, 3, 13, 2589, 'confirmed'),

(DEFAULT, 4, 4, 3, 14, 2589, 'confirmed');

plane id, leg id

INSERT INTO schedules VALUES

(1, 1),

(2, 2),

(3, 3),

(4, 4),

(5, 5),

(2, 6),

(3, 7),

(5, 8),

(2, 9),

```
(1, 10),  
(4, 11),  
(4, 12),  
(1, 13),  
(2, 14),  
(3, 15),  
(5, 16),  
(2, 17),  
    (1, 18),  
(3, 19);
```

order id, employee id, customer id

INSERT INTO orders VALUES

```
(DEFAULT, 2, 1),  
(DEFAULT, 2, 2),  
(DEFAULT, 4, 3),  
(DEFAULT, 1, 4);
```

order id, ticket id

INSERT INTO orders_tickets VALUES

```
(1, 1),  
    (2, 2),  
(3, 3),  
(3, 4),
```

(4, 5);

Appendix 9: The commented source code

// Address Data Type - created by Dana

```
package SEJ.ApplicationLayer.DataTypes;

public class Address {
    private int addressID;
    private String streetAndNumber;
    private String city;
    private int zip;
    private String country;

    public Address(int addressID, String streetAndNumber, String city, int zip,
String country) {
        this.addressID = addressID;
        this.streetAndNumber = streetAndNumber;
        this.city = city;
        this.zip = zip;
        this.country = country;
    }

    public Address(String streetAndNumber, String city, int zip, String country)
{
        this.streetAndNumber = streetAndNumber;
        this.city = city;
        this.zip = zip;
        this.country = country;
    }

    public int getAddressID() {
        return addressID;
    }
    public void setAddressID(int addressID) {
        this.addressID = addressID;
    }
    public String getCountry() {
        return country;
    }
    public void setCountry(String country) {
        this.country = country;
    }
    public String getCity() {
        return city;
    }
    public void setCity(String city) {
        this.city = city;
    }
    public int getZip() {
        return zip;
    }
    public void setZip(int zip) {
        this.zip = zip;
    }
}
```

```

    }
    public String getStreetAndNumber() {
        return streetAndNumber;
    }
    public void setStreetAndNumber(String streetAndNumber) {
        this.streetAndNumber = streetAndNumber;
    }

    @Override
    public String toString() {
        return "Address{" +
            "addressID=" + addressID +
            ", streetAndNumber=" + streetAndNumber + '\'' +
            ", city=" + city + '\'' +
            ", zip=" + zip +
            ", country=" + country + '\'' +
            '}';
    }
}

```

```

// Customer Data Type - Created by Dana

package SEJ.ApplicationLayer.DataTypes;

public class Customer
{
    private int customerID;
    private String firstName;
    private String lastName;
    private String gender;
    private String birthday;
    private String nationality;
    private int passNumber;
    private String passExpirationDate;
    private String email;
    private String phone;
    private int addressID;

    // unused constructor
    public Customer(int customerID, String firstName, String lastName, String
email, String phone)
    {
        this.customerID = customerID;
        this.firstName = firstName;
        this.lastName = lastName;
        this.email = email;
        this.phone = phone;
    }

    // to construct a customer without customer ID, used for writing in database
    // in the CustomersSQL when we write a new Customer the customer ID is
    'DEFAULT'
    public Customer(String firstName, String lastName, String gender, String
birthday, String nationality, int pasNumber, String pasExpirationDate, int
addressID, String email, String phone)
    {
        this.firstName = firstName;
        this.lastName = lastName;
        this.gender = gender;
        this.birthday = birthday;
        this.nationality = nationality;
        this.passNumber = pasNumber;
        this.passExpirationDate = pasExpirationDate;
        this.addressID = addressID;
        this.email = email;
        this.phone = phone;
    }

    //to construct a customer with Customer ID
    public Customer(int customerID, String firstName, String lastName, String
gender, String birthday, String nationality, int passNumber, String
pasExpiredDate, int addressID, String email, String phone)
    {
        this.customerID = customerID;
        this.firstName = firstName;
        this.lastName = lastName;
        this.gender = gender;
        this.birthday = birthday;
        this.nationality = nationality;
        this.passNumber = passNumber;
    }
}

```

```

        this.passExpirationDate = pasExpiredDate;
        this.email = email;
        this.phone = phone;
        this.addressID = addressID;
    }

    public int getCustomerID()
    {
        return customerID;
    }
    public void setCustomerID(int customerID)
    {
        this.customerID = customerID;
    }
    public String getFirstName()
    {
        return firstName;
    }
    public void setFirstName(String firstName)
    {
        this.firstName = firstName;
    }
    public String getLastName()
    {
        return lastName;
    }
    public void setLastName(String lastName)
    {
        this.lastName = lastName;
    }
    public String getGender()
    {
        return gender;
    }
    public void setGender(String gender)
    {
        this.gender = gender;
    }
    public String getBirthday()
    {
        return birthday;
    }
    public void setBirthday(String birthday)
    {
        this.birthday = birthday;
    }
    public void setPassExpirationDate(String passExpirationDate)
    {
        this.passExpirationDate = passExpirationDate;
    }
    public int getAddressID()
    {
        return addressID;
    }
    public void setAddressID(int addressID)
    {
        this.addressID = addressID;
    }
    public String getNationality()
    {

```

```

        return nationality;
    }
    public void setNationality(String nationality)
    {
        this.nationality = nationality;
    }
    public int getPasNumber()
    {
        return passNumber;
    }
    public void setPasNumber(int pasNumber)
    {
        this.passNumber = pasNumber;
    }
    public String getPassExpirationDate()
    {
        return passExpirationDate;
    }
    public String getEmail()
    {
        return email;
    }
    public void setEmail(String email)
    {
        this.email = email;
    }
    public String getPhone()
    {
        return phone;
    }
    public void setPhone(String phone)
    {
        this.phone = phone;
    }

    @Override
    public String toString()
    {
        return "Customer{" +
            "customerID='" + customerID + '\'' +
            ", firstName='" + firstName + '\'' +
            ", lastName='" + lastName + '\'' +
            ", gender='" + gender + '\'' +
            ", birthday=" + birthday +
            ", nationality='" + nationality + '\'' +
            ", pasNumber='" + passNumber + '\'' +
            ", passExpirationDate=" + passExpirationDate +
            ", email='" + email + '\'' +
            ", phone='" + phone + '\'' +
            ", address='" + addressID + '\'' +
            '}';
    }
}

```



```

// Employee Data Type - Created by Lei

package SEJ.ApplicationLayer.DataTypes;

public class Employee
{
    private byte employeeID;
    private String firstName;
    private String lastName;
    private String title;
    private String userName;
    private String password;

    // constructor
    public Employee(byte employeeID, String firstName, String lastName, String
title, String userName, String password)
    {
        this.employeeID = employeeID;
        this.firstName = firstName;
        this.lastName = lastName;
        this.title = title;
        this.userName = userName;
        this.password = password;
    }

    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }
    public String getFirstName()
    {
        return firstName;
    }
    public void setFirstName(String firstName)
    {
        this.firstName = firstName;
    }
    public String getLastName()
    {
        return lastName;
    }
    public void setLastName(String lastName)
    {
        this.lastName = lastName;
    }
    public int getEmployeeID()
    {
        return employeeID;
    }
    public void setEmployeeID(byte employeeID)
    {
        this.employeeID = employeeID;
    }
    public String getUserName()
    {
        return userName;
    }
    public void setUserName(String userName)

```

```

    {
        this.userName = userName;
    }
    public String getPassword()
    {
        return password;
    }
    public void setPassword(String password)
    {
        this.password = password;
    }

    @Override
    public String toString()
    {
        return employeeID + "," + firstName + "," + lastName + "," + userName +
        "," + password + "\n";
    }
}

```

```

// FlightSearchResult Data Type - Created by Lei
// Used in searching a flight

package SEJ.ApplicationLayer.DataTypes;

import javafx.beans.property.IntegerProperty;
import javafx.beans.property.SimpleIntegerProperty;
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;

public class FlightSearchResult
{
    private StringProperty departureCity;
    private StringProperty arrivalCity;
    private StringProperty departureDate;
    private StringProperty departureTime;
    private StringProperty arrivalDate;
    private StringProperty arrivalTime;
    private IntegerProperty firstClassBooked;
    private IntegerProperty businessClassBooked;
    private IntegerProperty coachClassBooked;

    public FlightSearchResult(String departureCity, String arrivalCity, String
departureDate, String departureTime, String arrivalDate, String arrivalTime)
    {
        this.departureCity = new SimpleStringProperty(departureCity);
        this.arrivalCity = new SimpleStringProperty(arrivalCity);
        this.departureDate = new SimpleStringProperty(departureDate);
        this.departureTime = new SimpleStringProperty(departureTime);
        this.arrivalDate = new SimpleStringProperty(arrivalDate);
        this.arrivalTime = new SimpleStringProperty(arrivalTime);
        this.firstClassBooked = new SimpleIntegerProperty(0);
        this.businessClassBooked = new SimpleIntegerProperty(0);
        this.coachClassBooked = new SimpleIntegerProperty(0);
    }

    public String getDepartureCity()
    {
        return departureCity.get();
    }
    public StringProperty departureCityProperty()
    {
        return departureCity;
    }
    public void setDepartureCity(String departureCity)
    {
        this.departureCity.set(departureCity);
    }
    public String getArrivalCity()
    {
        return arrivalCity.get();
    }
    public StringProperty arrivalCityProperty()
    {
        return arrivalCity;
    }
    public void setArrivalCity(String arrivalCity)
    {
        this.arrivalCity.set(arrivalCity);
    }
}

```

```

public String getDepartureTime()
{
    return departureTime.get();
}
public StringProperty departureTimeProperty()
{
    return departureTime;
}
public void setDepartureTime(String departureTime)
{
    this.departureTime.set(departureTime);
}
public String getArrivalTime()
{
    return arrivalTime.get();
}
public StringProperty arrivalTimeProperty()
{
    return arrivalTime;
}
public void setArrivalTime(String arrivalTime)
{
    this.arrivalTime.set(arrivalTime);
}
public String getDepartureDate()
{
    return departureDate.get();
}
public StringProperty departureDateProperty()
{
    return departureDate;
}
public void setDepartureDate(String departureDate)
{
    this.departureDate.set(departureDate);
}
public String getArrivalDate()
{
    return arrivalDate.get();
}
public StringProperty arrivalDateProperty()
{
    return arrivalDate;
}
public void setArrivalDate(String arrivalDate)
{
    this.arrivalDate.set(arrivalDate);
}
public int getFirstClassBooked()
{
    return firstClassBooked.get();
}
public IntegerProperty firstClassBookedProperty()
{
    return firstClassBooked;
}
public void setFirstClassBooked(int firstClassBooked)
{
    this.firstClassBooked.set(firstClassBooked);
}

```

```

public int getBusinessClassBooked()
{
    return businessClassBooked.get();
}
public IntegerProperty businessClassBookedProperty()
{
    return businessClassBooked;
}
public void setBusinessClassBooked(int businessClassBooked)
{
    this.businessClassBooked.set(businessClassBooked);
}
public int getCoachClassBooked()
{
    return coachClassBooked.get();
}
public IntegerProperty coachClassBookedProperty()
{
    return coachClassBooked;
}
public void setCoachClassBooked(int coachClassBooked)
{
    this.coachClassBooked.set(coachClassBooked);
}

@Override
public String toString() {
    return "FlightSearchResult{" +
        "departureCity=" + departureCity +
        ", arrivalCity=" + arrivalCity +
        ", departureDate=" + departureDate +
        ", departureTime=" + departureTime +
        ", arrivalDate=" + arrivalDate +
        ", arrivalTime=" + arrivalTime +
        ", firstClassBooked=" + firstClassBooked +
        ", businessClassBooked=" + businessClassBooked +
        ", coachClassBooked=" + coachClassBooked +
        '}';
}
}

```

```

// Leg Data Type - Created by Lei

package SEJ.ApplicationLayer.DataTypes;

public class Leg
{
    private int legID;
    private String departureAirport;
    private String arrivalAirport;
    private String departureDate;
    private String departureTime;
    private String arrivalDate;
    private String arrivalTime;
    private byte firstSeatBooked;
    private byte businessSeatBooked;
    private short coachSeatBooked;
    private int price;

    // constructor with leg id, without specifying how many seats are booked for
    // each leg
    // used when creating a new schedule
    public Leg(int legID, String departureAirport, String arrivalAirport, String
departureDate, String departureTime, String arrivalDate, String arrivalTime, int
price)
    {
        this.legID = legID;
        this.departureAirport = departureAirport;
        this.arrivalAirport = arrivalAirport;
        this.departureDate = departureDate;
        this.departureTime = departureTime;
        this.arrivalDate = arrivalDate;
        this.arrivalTime = arrivalTime;
        this.price = price;
    }

    // constructor with all fields
    // used when reading from the database
    public Leg(int legID, String departureAirport, String arrivalAirport, String
departureDate, String departureTime, String arrivalDate, String arrivalTime, byte
firstSeatBooked, byte businessSeatBooked, short coachSeatBooked, int price)
    {
        this.legID = legID;
        this.departureAirport = departureAirport;
        this.arrivalAirport = arrivalAirport;
        this.departureDate = departureDate;
        this.departureTime = departureTime;
        this.arrivalDate = arrivalDate;
        this.arrivalTime = arrivalTime;
        this.firstSeatBooked = firstSeatBooked;
        this.businessSeatBooked = businessSeatBooked;
        this.coachSeatBooked = coachSeatBooked;
        this.price = price;
    }

    // constructor without leg id
    // used when creating a new schedule
    public Leg(String departureAirport, String arrivalAirport, String
departureDate,
                String departureTime, String arrivalDate, String arrivalTime, int
price)

```

```

{
    this.departureAirport = departureAirport;
    this.arrivalAirport = arrivalAirport;
    this.departureDate = departureDate;
    this.departureTime = departureTime;
    this.arrivalDate = arrivalDate;
    this.arrivalTime = arrivalTime;
    this.price = price;
}

public int getLegID()
{
    return legID;
}
public void setLegID(int legID)
{
    this.legID = legID;
}
public String getDepartureAirport()
{
    return departureAirport;
}
public void setDepartureAirport(String departureAirport)
{
    this.departureAirport = departureAirport;
}
public String getArrivalAirport()
{
    return arrivalAirport;
}
public void setArrivalAirport(String arrivalAirport)
{
    this.arrivalAirport = arrivalAirport;
}
public String getDepartureDate()
{
    return departureDate;
}
public void setDepartureDate(String departureDate)
{
    this.departureDate = departureDate;
}
public String getDepartureTime()
{
    return departureTime;
}
public void setDepartureTime(String departureTime)
{
    this.departureTime = departureTime;
}
public String getArrivalDate()
{
    return arrivalDate;
}
public void setArrivalDate(String arrivalDate)
{
    this.arrivalDate = arrivalDate;
}
public String getArrivalTime()
{

```

```

        return arrivalTime;
    }
    public void setArrivalTime(String arrivalTime)
    {
        this.arrivalTime = arrivalTime;
    }
    public byte getFirstSeatBooked()
    {
        return firstSeatBooked;
    }
    public void setFirstSeatBooked(byte firstSeatBooked)
    {
        this.firstSeatBooked = firstSeatBooked;
    }
    public byte getBusinessSeatBooked()
    {
        return businessSeatBooked;
    }
    public void setBusinessSeatBooked(byte businessSeatBooked)
    {
        this.businessSeatBooked = businessSeatBooked;
    }
    public short getCoachSeatBooked()
    {
        return coachSeatBooked;
    }
    public void setCoachSeatBooked(short coachSeatBooked)
    {
        this.coachSeatBooked = coachSeatBooked;
    }
    public int getPrice() {
        return price;
    }
    public void setPrice(int price) {
        this.price = price;
    }
}

@Override
public String toString() {
    return "Leg{" +
        "legID=" + legID +
        ", departureAirport='" + departureAirport + '\'' +
        ", arrivalAirport='" + arrivalAirport + '\'' +
        ", departureDate='" + departureDate + '\'' +
        ", departureTime='" + departureTime + '\'' +
        ", arrivalDate='" + arrivalDate + '\'' +
        ", arrivalTime='" + arrivalTime + '\'' +
        ", firstSeatBooked=" + firstSeatBooked +
        ", businessSeatBooked=" + businessSeatBooked +
        ", coachSeatBooked=" + coachSeatBooked +
        ", price=" + price +
        '}';
}
}

```



```

// Order Data Type - Created by Dana

package SEJ.ApplicationLayer.DataTypes;

import java.util.*;

public class Order
{
    private int orderID;
    private int employeeID;
    private int customerID;
    static List<Ticket> tickets = new ArrayList<>();

    // used when creating an order for the orders table
    public Order(int orderID, int employeeID, int customerID) {
        this.orderID = orderID;
        this.employeeID = employeeID;
        this.customerID = customerID;
    }

    // constructor with employee id and customer id
    public Order(int employeeID, int customerID) {
        this.employeeID = employeeID;
        this.customerID = customerID;
    }

    // unused constructor
    public Order()
    {
    }

    // constructor used for the orders_tickets table
    public Order(int orderID, List<Ticket> tickets)
    {
        this.orderID = orderID;
        this.tickets = tickets;
    }

    public static void addTicketToOrder(Ticket ticket){
        tickets.add(ticket);
    }

    public int getOrderID() {
        return orderID;
    }
    public void setOrderID(int orderID) {
        this.orderID = orderID;
    }
    public int getEmployeeID() {
        return employeeID;
    }
    public void setEmployeeID(int employeeID) {
        this.employeeID = employeeID;
    }
    public int getCustomerID() {
        return customerID;
    }
    public void setCustomerID(int customerID) {
        this.customerID = customerID;
    }
}

```

```
    public List<Ticket> getTickets() {  
        return tickets;  
    }  
    public void setTickets(List<Ticket> tickets) {  
        this.tickets = tickets;  
    }  
}
```

```

// OrderContent Data Type - Created by Lei

// Used for searching an order
package SEJ.ApplicationLayer.DataTypes;

public class OrderContent
{
    private int orderID;
    private int ticketID;
    private String customerFirstName;
    private String customerLastName;
    private int ticketClass;
    private int ticketPrice;
    private int ticketSeatNumber;
    private String ticketState;

    public OrderContent(int orderID, int ticketID, String customerFirstName,
String customerLastName, int ticketClass, int ticketPrice, int ticketSeatNumber,
String ticketState)
    {
        this.orderID = orderID;
        this.ticketID = ticketID;
        this.customerFirstName = customerFirstName;
        this.customerLastName = customerLastName;
        this.ticketClass = ticketClass;
        this.ticketPrice = ticketPrice;
        this.ticketSeatNumber = ticketSeatNumber;
        this.ticketState = ticketState;
    }

    public int getOrderID()
    {
        return orderID;
    }
    public void setOrderID(int orderID)
    {
        this.orderID = orderID;
    }
    public int getTicketID() {
        return ticketID;
    }
    public void setTicketID(int ticketID) {
        this.ticketID = ticketID;
    }
    public String getCustomerFirstName()
    {
        return customerFirstName;
    }
    public void setCustomerFirstName(String customerFirstName)
    {
        this.customerFirstName = customerFirstName;
    }
    public String getCustomerLastName()
    {
        return customerLastName;
    }
    public void setCustomerLastName(String customerLastName)
    {
        this.customerLastName = customerLastName;
    }
}

```

```

    }
    public int getTicketClass()
    {
        return ticketClass;
    }
    public void setTicketClass(int ticketClass)
    {
        this.ticketClass = ticketClass;
    }
    public int getTicketPrice()
    {
        return ticketPrice;
    }
    public void setTicketPrice(int ticketPrice)
    {
        this.ticketPrice = ticketPrice;
    }
    public int getTicketSeatNumber()
    {
        return ticketSeatNumber;
    }
    public void setTicketSeatNumber(int ticketSeatNumber)
    {
        this.ticketSeatNumber = ticketSeatNumber;
    }
    public String getTicketState()
    {
        return ticketState;
    }
    public void setTicketState(String ticketState)
    {
        this.ticketState = ticketState;
    }

    @Override
    public String toString()
    {
        return "OrderContent{" +
            "orderID=" + orderID +
            ", customerFirstName='" + customerFirstName + '\'' +
            ", customerLastName='" + customerLastName + '\'' +
            ", ticketClass=" + ticketClass +
            ", ticketPrice=" + ticketPrice +
            ", ticketSeatNumber=" + ticketSeatNumber +
            ", ticketState='" + ticketState + '\'' +
            '}';
    }
}

```

// OrderSearchResult Data Type - Created by Lei

```
package SEJ.ApplicationLayer.DataTypes;

import javafx.beans.property.IntegerProperty;
import javafx.beans.property.SimpleIntegerProperty;
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;

public class OrderSearchResult
{
    private IntegerProperty orderID;
    private IntegerProperty ticketID;
    private StringProperty customerFirstName;
    private StringProperty customerLastName;
    private IntegerProperty ticketClass;
    private IntegerProperty ticketPrice;
    private IntegerProperty ticketSeatNumber;
    private StringProperty ticketState;

    public OrderSearchResult(int orderID, int ticketID, String customerFirstName,
String customerLastName, int ticketClass, int ticketPrice, int ticketSeatNumber,
String ticketState)
    {
        this.orderID = new SimpleIntegerProperty(orderID);
        this.ticketID = new SimpleIntegerProperty(ticketID);
        this.customerFirstName = new SimpleStringProperty(customerFirstName);
        this.customerLastName = new SimpleStringProperty(customerLastName);
        this.ticketClass = new SimpleIntegerProperty(ticketClass);
        this.ticketPrice = new SimpleIntegerProperty(ticketPrice);
        this.ticketSeatNumber = new SimpleIntegerProperty(ticketSeatNumber);
        this.ticketState = new SimpleStringProperty(ticketState);
    }

    public int getOrderID()
    {
        return orderID.get();
    }
    public IntegerProperty orderIDProperty()
    {
        return orderID;
    }
    public void setOrderID(int orderID)
    {
        this.orderID.set(orderID);
    }
    public int getTicketID() {
        return ticketID.get();
    }
    public IntegerProperty ticketIDProperty() {
        return ticketID;
    }
    public void setTicketID(int ticketID) {
        this.ticketID.set(ticketID);
    }
    public StringProperty customerLastNameProperty() {
        return customerLastName;
    }
    public String getCustomerFirstName()
```

```

    {
        return customerFirstName.get();
    }
    public StringProperty customerFirstNameProperty()
    {
        return customerFirstName;
    }
    public void setCustomerFirstName(String customerFirstName)
    {
        this.customerFirstName.set(customerFirstName);
    }
    public String getCustomerLastName()
    {
        return customerLastName.get();
    }
    public StringProperty customerLasttNameProperty()
    {
        return customerLastName;
    }
    public void setCustomerLastName(String customerLasttName)
    {
        this.customerLastName.set(customerLasttName);
    }
    public int getTicketClass()
    {
        return ticketClass.get();
    }
    public IntegerProperty ticketClassProperty()
    {
        return ticketClass;
    }
    public void setTicketClass(int ticketClass)
    {
        this.ticketClass.set(ticketClass);
    }
    public int getTicketPrice()
    {
        return ticketPrice.get();
    }
    public IntegerProperty ticketPriceProperty()
    {
        return ticketPrice;
    }
    public void setTicketPrice(int ticketPrice)
    {
        this.ticketPrice.set(ticketPrice);
    }
    public int getTicketSeatNumber()
    {
        return ticketSeatNumber.get();
    }
    public IntegerProperty ticketSeatNumberProperty()
    {
        return ticketSeatNumber;
    }
    public void setTicketSeatNumber(int ticketSeatNumber)
    {
        this.ticketSeatNumber.set(ticketSeatNumber);
    }
    public String getTicketState()

```

```

    {
        return ticketState.get();
    }
    public StringProperty ticketStateProperty()
    {
        return ticketState;
    }
    public void setTicketState(String ticketState)
    {
        this.ticketState.set(ticketState);
    }

    @Override
    public String toString() {
        return "OrderSearchResult{" +
            "orderId=" + orderId +
            ", customerFirstName=" + customerFirstName +
            ", customerLastName=" + customerLastName +
            ", ticketClass=" + ticketClass +
            ", ticketPrice=" + ticketPrice +
            ", ticketSeatNumber=" + ticketSeatNumber +
            ", ticketState=" + ticketState +
            '}';
    }
}

```

```

// Plane Data Type - Created by Lei

package SEJ.ApplicationLayer.DataTypes;

public class Plane
{
    private int planeID;
    private String planeType;
    private int firstClassSeatTotal;
    private int businessSeatTotal;
    private int coachSeatTotal;

    // constructor with all fields
    public Plane(int planeID, String planeType, int firstClassSeatTotal, int
businessSeatTotal, int coachSeatTotal)
    {
        this.planeID = planeID;
        this.planeType = planeType;
        this.firstClassSeatTotal = firstClassSeatTotal;
        this.businessSeatTotal = businessSeatTotal;
        this.coachSeatTotal = coachSeatTotal;
    }

    // constructor without plane id
    public Plane(String planeType, int firstClassSeatTotal, int
businessSeatTotal, int coachSeatTotal)
    {
        this.planeType = planeType;
        this.firstClassSeatTotal = firstClassSeatTotal;
        this.businessSeatTotal = businessSeatTotal;
        this.coachSeatTotal = coachSeatTotal;
    }

    public Plane()
    {
    }

    public int getPlaneID()
    {
        return planeID;
    }
    public void setPlaneID(int planeID)
    {
        this.planeID = planeID;
    }
    public String getPlaneType()
    {
        return planeType;
    }
    public void setPlaneType(String planeType)
    {
        this.planeType = planeType;
    }
    public int getFirstClassSeatTotal()
    {
        return firstClassSeatTotal;
    }
    public void setFirstClassSeatTotal(int firstClassSeatTotal)
    {

```



```

        this.firstClassSeatTotal = firstClassSeatTotal;
    }
    public int getBusinessSeatTotal()
    {
        return businessSeatTotal;
    }
    public void setBusinessSeatTotal(int businessSeatTotal)
    {
        this.businessSeatTotal = businessSeatTotal;
    }
    public int getCoachSeatTotal()
    {
        return coachSeatTotal;
    }
    public void setCoachSeatTotal(int coachSeatTotal)
    {
        this.coachSeatTotal = coachSeatTotal;
    }

    @Override
    public String toString()
    {
        return planeID + "," + planeType + "," + firstClassSeatTotal + "," +
businessSeatTotal + "," + coachSeatTotal + "\n";
    }
}

```

```
// Schedule Data Type - Created by Lei

package SEJ.ApplicationLayer.DataTypes;

public class Schedule
{
    int planeID;
    int legID;

    public Schedule(int planeID, int legID)
    {
        this.planeID = planeID;
        this.legID = legID;
    }

    public int getPlaneID() {
        return planeID;
    }
    public void setPlaneID(int planeID) {
        this.planeID = planeID;
    }
    public int getLegID() {
        return legID;
    }
    public void setLegID(int legID) {
        this.legID = legID;
    }
}
```

// Ticket Data Type - Created by Lei

package SEJ.ApplicationLayer.DataTypes;

public class Ticket

```
{
    private int ticketID;
    private int legID;
    private int customerID;
    private int ticketClass;
    private int ticketSeatNumber;
    private int ticketPrice;
    private String ticketState;

    public Ticket(int ticketID, int legID, int customerID, int ticketClass,
                  int ticketSeatNumber, int ticketPrice, String ticketState)
    {
        this.ticketID = ticketID;
        this.legID = legID;
        this.customerID = customerID;
        this.ticketClass = ticketClass;
        this.ticketSeatNumber = ticketSeatNumber;
        this.ticketPrice = ticketPrice;
        this.ticketState = ticketState;
    }

    public Ticket() {}

    public int getTicketID()
    {
        return ticketID;
    }
    public void setTicketID(int ticketID)
    {
        this.ticketID = ticketID;
    }
    public int getLegID()
    {
        return legID;
    }
    public void setLegID(int legID)
    {
        this.legID = legID;
    }
    public int getCustomerID()
    {
        return customerID;
    }
    public void setCustomerID(int customerID)
    {
        this.customerID = customerID;
    }
    public int getTicketClass()
    {
        return ticketClass;
    }
    public void setTicketClass(int ticketClass)
    {
        this.ticketClass = ticketClass;
    }
}
```

```

    }
    public int getTicketSeatNumber()
    {
        return ticketSeatNumber;
    }
    public void setTicketSeatNumber(int ticketSeatNumber)
    {
        this.ticketSeatNumber = ticketSeatNumber;
    }
    public int getTicketPrice()
    {
        return ticketPrice;
    }
    public void setTicketPrice(int ticketPrice)
    {
        this.ticketPrice = ticketPrice;
    }
    public String getTicketState()
    {
        return ticketState;
    }
    public void setTicketState(String ticketState)
    {
        this.ticketState = ticketState;
    }

    @Override
    public String toString()
    {
        return "Ticket{" +
            "ticketID=" + ticketID +
            ", legID=" + legID +
            ", customerID=" + customerID +
            ", ticketClass=" + ticketClass +
            ", ticketSeatNumber=" + ticketSeatNumber +
            ", ticketPrice=" + ticketPrice +
            ", ticketState=" + ticketState + '\'' +
            '}';
    }
}

```

```

// AddressInfo Class - Created by Lei

package SEJ.ApplicationLayer;

import SEJ.ApplicationLayer.DataTypes.Address;
import SEJ.DataAccessLayer.AddressSQL;
import java.util.List;

public class AddressInfo {
    static Address address;

    // gets all addresses from the Data Access Layer
    public static List<Address> selectAllAddresses() throws Exception {
        List<Address> addresses;
        addresses = AddressSQL.loadAllAddresses();
        return addresses;
    }

    // sends address to the Data Access Layer
    public static void createAddress(String streetAndNumber, String city, int
zip, String country) throws Exception {
        address = new Address(streetAndNumber, city, zip, country);
        AddressSQL.createAddress(address);
    }
}

```

```

// CheckInputFormat Class - Created by Lei
// Used for checking if the information typed in by user is in the right format

package SEJ.ApplicationLayer;

import java.text.ParseException;
import java.text.SimpleDateFormat;

public class CheckInputFormat
{
    /* The following method is copied from http://stackoverflow.com
    * http://stackoverflow.com/questions/1102891/how-to-check-if-a-string-is-
    numeric-in-java*/
    public static boolean isNumeric(String str) {
        for (char c : str.toCharArray()) {
            if (!Character.isDigit(c))
                return false;
        }
        return true;
    }
    /* The following method is copied from http://www.java2s.com
    http://www.java2s.com/Tutorial/Java/0120__Development/CheckifaStringisavalidddate.h
    tm*/
    public static boolean isValidDate(String inDate) {
        SimpleDateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy");
        dateFormat.setLenient(false);
        try {
            dateFormat.parse(inDate.trim());
        } catch (ParseException pe) {
            return false;
        }
        return true;
    }

    // method created by Dana
    public static boolean isTime(String str){
        if(str.length() != 5)
            return false;
        if(!Character.isDigit(str.charAt(0)) || !Character.isDigit(str.charAt(1))
        || !Character.isDigit(str.charAt(3))
        || !Character.isDigit(str.charAt(4)))
            return false;
        if(str.charAt(2) != ':')
            return false;
        return true;
    }
}

```

```

// CustomerInfo Class - Created by Marius

package SEJ.ApplicationLayer;

import SEJ.ApplicationLayer.DataTypes.Customer;
import SEJ.DataAccessLayer.CustomerSQL;
import java.util.List;

public class CustomerInfo{
    private static List<Customer> customers;
    static Customer customer;

    // gets all customers from the Data Access Layer
    public static List<Customer> selectAllCustomers() throws Exception
    {
        customers = CustomerSQL.loadAllCustomers();
        return customers;
    }

    // sends customer to the Data Access Layer
    public static void createCustomer2(String firstName, String lastName, String
gender, String birthday, String nationality, int pasNumber, String
pasExpirationDate, int addressID, String email, String phone) throws Exception {
        customer = new Customer(firstName, lastName, gender, birthday,
nationality, pasNumber, pasExpirationDate, addressID, email, phone);
        CustomerSQL.createCustomer(customer);
    }
}

```

// EmployeeInfo Class - Created by Marius

```
package SEJ.ApplicationLayer;

import SEJ.ApplicationLayer.DataTypes.Employee;
import SEJ.DataAccessLayer.EmployeeSQL;
import java.io.File;
import java.io.FileOutputStream;
import java.io.PrintStream;
import java.util.List;
import java.util.Scanner;

public class EmployeeInfo{
    private static List<Employee> employees;

    // gets all employees from the Data Access Layer
    public static List<Employee> selectAllEmployees() throws Exception
    {
        employees = EmployeeSQL.loadAllEmployees();
        return employees;
    }

    // OSCA part
    // loops through the array of all employees, writes them in CSV file
    public static void readEmployeesInTable(List<Employee> getEmployeesTable)
    throws Exception
    {
        FileOutputStream fos = new FileOutputStream("employees.CSV", false);
        fos.write(makeComment().getBytes());
        for(Employee e : getEmployeesTable)
        {
            fos.write(e.toString().getBytes());
        }
        fos.close();
    }

    // adds a comment at the beginning of CSV file
    public static String makeComment(){
        String comment = "/*\n" +
            "        Employees Table\n" +
            "id, fname, lname, username, password\n" +
            "*/\n";
        return comment;
    }

    // ignores comment from the CSV file
    // writes data without comment in the employeeOutput.txt file
    public static void readIgnoreComment() throws Exception {
        Scanner input = new Scanner(new File("employees.CSV"));
        PrintStream output = new PrintStream (new File("employeeOutput.txt"));
        boolean flag = false;
        while (input.hasNextLine() ) {
            String line = input.nextLine();
            if(line.startsWith("/*"))
                flag = true;
            if(line.endsWith("*/")) {
                flag = false;
            }
            Scanner lineScan = new Scanner(line);
        }
    }
}
```



```
        while (lineScan.hasNextLine() && flag == false) {  
            if (line.startsWith("*/"))  
                break;  
            else  
                output.println(lineScan.nextLine());  
        }  
    }  
}
```

```

// FlightSearchResultInfo Class - Created by Lei

package SEJ.ApplicationLayer;

import SEJ.ApplicationLayer.DataTypes.FlightSearchResult;
import SEJ.ApplicationLayer.DataTypes.Leg;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import java.util.List;

public class FlightSearchResultInfo
{
    static Leg selectedLeg;

    // gets all legs that satisfy the search condition
    private static List<Leg> getAllSearchedLegs(String departureCity, String
arrivalCity, String departureDate) throws Exception
    {
        List<Leg> allLSearchedLegs = LegInfo.createSearchedLegs(departureCity,
arrivalCity, departureDate);
        return allLSearchedLegs;
    }

    // creates Observable List with FlightSearchResult objects
    // takes information from each leg and creates a FlightSearchResult object
    (that has StringProperties as fields)
    // returns the allFlightResults Observable List that will be used in the
    TableView
    public static ObservableList<FlightSearchResult>
createFlightSearchResultsProperty(String departureCity, String arrivalCity, String
departureDate) throws Exception
    {
        List<Leg> allLSearchedLegs = getAllSearchedLegs(departureCity,
arrivalCity, departureDate);

        ObservableList<FlightSearchResult> allFlightResults =
FXCollections.observableArrayList();

        for(Leg leg: allLSearchedLegs)
        {
            String dep_city = leg.getDepartureAirport();
            String arr_city = leg.getArrivalAirport();
            String dep_Date = leg.getDepartureDate().toString();
            String dep_Time = leg.getDepartureTime().toString();
            String arr_date = leg.getArrivalDate().toString();
            String arr_Time = leg.getArrivalTime().toString();
            FlightSearchResult flightSearchResult = new
FlightSearchResult(dep_city, arr_city, dep_Date, dep_Time,
arr_date, arr_Time);
            allFlightResults.add(flightSearchResult);
        }
        return allFlightResults;
    }
}

```

```

// finds the selected leg when the user clicks on one row in the TableView
    public static Leg findSelectedLeg(FlightSearchResult selectedFlight) throws
Exception
    {
        List<Leg> allLegs = LegInfo.selectAllLegs();
        for (int i = 0; i < allLegs.size(); i++){

            if(selectedFlight.getDepartureCity().equals(allLegs.get(i).getDepartureAirport())
            && selectedFlight.getDepartureDate().equals(allLegs.get(i).getDepartureDate()) &&
            selectedFlight.getDepartureTime().equals(allLegs.get(i).getDepartureTime()))
                selectedLeg = allLegs.get(i);
            }
            return selectedLeg;
        }
    }
}

```

```

// LegInfo Class - created by Felix

package SEJ.ApplicationLayer;

import SEJ.ApplicationLayer.DataTypes.Leg;
import SEJ.DataAccessLayer.LegSQL;
import java.util.ArrayList;
import java.util.List;

public class LegInfo {
    private static List<Leg> legs;
    static Leg leg;

    // gets all legs from the Data Access Layer
    public static List<Leg> selectAllLegs() throws Exception
    {
        legs = LegSQL.loadAllLegs();
        return legs;
    }

    // sends leg to the Data Access Layer
    // adds it to the array
    public static void createLeg(String departureAirport, String arrivalAirport,
String departureDate, String departureTime, String arrivalDate, String
arrivalTime, int price) throws Exception {
        leg = new Leg(departureAirport, arrivalAirport, departureDate,
departureTime, arrivalDate, arrivalTime, price);
        LegSQL.createLeg(leg);
        legs.add(new Leg(legs.size(), departureAirport, arrivalAirport,
departureDate, departureTime, arrivalDate, arrivalTime, price));
    }

    //returns an array of legs that satisfy the search requirements
    public static List<Leg> createSearchedLegs(String departureCity, String
arrivalCity, String departureDate) throws Exception
    {
        List<Leg> searchedLegs = new ArrayList<>();
        List<Leg> localLegs = selectAllLegs();
        for(int i = 0; i < localLegs.size(); i++)
        {
            //legs that fits search condition is added to a list
            String leg_departureCity =
localLegs.get(i).getDepartureAirport().substring(0,3);
            String leg_arrivalCity =
localLegs.get(i).getArrivalAirport().substring(0,3);
            String leg_departureDate =
localLegs.get(i).getDepartureDate().toString();
            if(departureCity.equalsIgnoreCase(leg_departureCity) &&
arrivalCity.equalsIgnoreCase(leg_arrivalCity)
                && departureDate.equalsIgnoreCase(leg_departureDate))
            {
                searchedLegs.add(localLegs.get(i));
            }
        }
        return searchedLegs;
    }
}

```

```

// sends necessary information (leg id, class type) to the Data Access Layer
// this information will be used for updating the legs table after a booking
is done
public static void updateSeat(int legId, int classType) throws Exception
{
    LegSQL.updateSeatBookedState(legId, classType);
}

// returns ticket price for different classes - created by Lei
public static int getPrice(Leg leg, String ticketClass)
{
    int price = 0;
    if(ticketClass.equalsIgnoreCase("Business Class"))
    {
        price = (leg.getPrice() * 85) / 100;
    }
    if(ticketClass.equalsIgnoreCase("Economy Class"))
    {
        price = (leg.getPrice()* 70) / 100;
    }
    if(ticketClass.equalsIgnoreCase("first class"))
    {
        price = leg.getPrice();
    }
    return price;
}
}

```

```

// OrderInfo Class - created by Marius

package SEJ.ApplicationLayer;

import SEJ.ApplicationLayer.DataTypes.Customer;
import SEJ.ApplicationLayer.DataTypes.Order;
import SEJ.ApplicationLayer.DataTypes.OrderContent;
import SEJ.ApplicationLayer.DataTypes.Ticket;
import SEJ.DataAccessLayer.OrderSQL;
import java.util.ArrayList;
import java.util.List;

public class OrderInfo {
    private static List<Order> orders;
    static Order order;

    // gets all orders from the Data Access Layer
    public static List<Order> selectAllOrders() throws Exception
    {
        orders = OrderSQL.loadAllOrders();
        return orders;
    }

    // sends order to the Data Access Layer
    // adds it to the array
    public static void createOrder(int employeeID, int customerID) throws
Exception
    {
        Order order = new Order(employeeID, customerID);
        OrderSQL.createOrder(order);
        orders.add(new Order(orders.size()+1, employeeID, customerID));
    }

    // add a ticket to an order - used for orders_tickets table
    public static void addTicketToOrder(Ticket ticket) throws Exception
    {
        order.addTicketToOrder(ticket);
    }

    // sends information about the orders_tickets table to the Data Access Layer
    public static void createOrders_Tickets(List<Ticket> tickets) throws
Exception
    {
        Order order = new Order(selectAllOrders().size()+1, tickets);
        OrderSQL.updateOrders_Tickets(order);
    }

    // creates an array with OrderContent objects - created by Lei
    // adds all orders that have the same order id as the one in the parameter
    // returns array
    public static List<OrderContent> createSearchedOrder(int orderID) throws
Exception
    {
        List<OrderContent> allOrders = OrderSQL.createOrderContent();
        List<OrderContent> searchedOrder = new ArrayList<>();
        for(int i = 0; i < allOrders.size(); i++)
        {
            if(orderID == allOrders.get(i).getOrderID())
            {

```

```

        searchedOrder.add(allOrders.get(i));
    }
    }
    return searchedOrder;
}

// finds the order id based on the customer's first and last name - created
by Dana
public static int findOrderId(String firstName, String lastName) throws
Exception
{
    List<Customer> allCustomers = CustomerInfo.selectAllCustomers();
    List<Order> allOrders = OrderInfo.selectAllOrders();
    int customerID = 0;
    int orderID = 0;

    // first, it finds the customer id the has the specified first and last
name
    for(int i = 0; i < allCustomers.size(); i++){
        if(firstName.equalsIgnoreCase(allCustomers.get(i).getFirstName()) &&
lastName.equalsIgnoreCase(allCustomers.get(i).getLastName()))
            customerID = allCustomers.get(i).getCustomerID();
    }

    // finds the order id that has the specified customer id
    for(int i = 0; i < allOrders.size(); i++){
        if(customerID == allOrders.get(i).getCustomerID())
            orderID = allOrders.get(i).getOrderID();
    }
    return orderID;
}
}

```

```

// OrderSearchResultInfo Class - Created by Lei

package SEJ.ApplicationLayer;

import SEJ.ApplicationLayer.DataTypes.*;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import java.util.List;

public class OrderSearchResultInfo
{
    static Ticket selectedTicket;

    // gets all orders that satisfy that have the specified order id
    private static List<OrderContent> getAllSearchedOrders(int orderId) throws
Exception
    {
        List<OrderContent> allSearchedOrders =
OrderInfo.createSearchedOrder(orderId);
        return allSearchedOrders;
    }

    // creates Observable List with OrderSearchResult objects
    // takes information from each OrderContent object and creates an
OrderSearchResult object (that has StringProperties as fields)
    // returns the allOrdersResults Observable List that will be used in the
TableView
    public static ObservableList<OrderSearchResult>
createOrderSearchResultsProperty(int orderId) throws Exception{
        List<OrderContent> allSearchedOrders = getAllSearchedOrders(orderId);

        ObservableList<OrderSearchResult> allOrdersResults =
FXCollections.observableArrayList();

        for(OrderContent orderContent: allSearchedOrders)
        {
            int orderID = orderContent.getOrderID();
            int ticketID = orderContent.getTicketID();
            String customerFirstName = orderContent.getCustomerFirstName();
            String customerLastName = orderContent.getCustomerLastName();
            int ticketClass = orderContent.getTicketClass();
            int ticketPrice = orderContent.getTicketPrice();
            int ticketSeatNumber = orderContent.getTicketSeatNumber();
            String ticketState = orderContent.getTicketState();
            OrderSearchResult orderSearchResult = new OrderSearchResult(orderID,
ticketID, customerFirstName, customerLastName, ticketClass, ticketPrice,
ticketSeatNumber, ticketState);
            allOrdersResults.add(orderSearchResult);
        }
        return allOrdersResults;
    }

    // each row from the TableView corresponds to an OrderSearchResult object
    // this method returns the ticket that is contained by that
OrderSearchResult
    // created by Dana
    public static Ticket findSelectedTicket(OrderSearchResult selectedOrder)
throws Exception{
        if(selectedOrder != null) {

```



```

List<Ticket> allTickets = TicketInfo.selectAllTickets();

    //loops through the tickets array and finds the selected ticket
    for (int i = 0; i < allTickets.size(); i++) {
        if (selectedOrder.getTicketID() ==
allTickets.get(i).getTicketID())
            selectedTicket = allTickets.get(i);
        }
    return selectedTicket;
    }
else
    return selectedTicket = new Ticket();
}
}

```

```

// OrderSearchResultInfo Class - Created by Lei

package SEJ.ApplicationLayer;

import SEJ.ApplicationLayer.DataTypes.*;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import java.util.List;

public class OrderSearchResultInfo
{
    static Ticket selectedTicket;

    // gets all orders that satisfy that have the specified order id
    private static List<OrderContent> getAllSearchedOrders(int orderId) throws
Exception
    {
        List<OrderContent> allSearchedOrders =
OrderInfo.createSearchedOrder(orderId);
        return allSearchedOrders;
    }

    // creates Observable List with OrderSearchResult objects
    // takes information from each OrderContent object and creates an
OrderSearchResult object (that has StringProperties as fields)
    // returns the allOrdersResults Observable List that will be used in the
TableView
    public static ObservableList<OrderSearchResult>
createOrderSearchResultsProperty(int orderId) throws Exception{
        List<OrderContent> allSearchedOrders = getAllSearchedOrders(orderId);

        ObservableList<OrderSearchResult> allOrdersResults =
FXCollections.observableArrayList();

        for(OrderContent orderContent: allSearchedOrders)
        {
            int orderID = orderContent.getOrderID();
            int ticketID = orderContent.getTicketID();
            String customerFirstName = orderContent.getCustomerFirstName();
            String customerLastName = orderContent.getCustomerLastName();
            int ticketClass = orderContent.getTicketClass();
            int ticketPrice = orderContent.getTicketPrice();
            int ticketSeatNumber = orderContent.getTicketSeatNumber();
            String ticketState = orderContent.getTicketState();
            OrderSearchResult orderSearchResult = new OrderSearchResult(orderID,
ticketID, customerFirstName, customerLastName, ticketClass, ticketPrice,
ticketSeatNumber, ticketState);
            allOrdersResults.add(orderSearchResult);
        }
        return allOrdersResults;
    }

    // each row from the TableView corresponds to an OrderSearchResult object
    // this method returns the ticket that is contained by that
OrderSearchResult
    // created by Dana
    public static Ticket findSelectedTicket(OrderSearchResult selectedOrder)
throws Exception{
        if(selectedOrder != null) {

```

```

        List<Ticket> allTickets = TicketInfo.selectAllTickets();
        //loops through the tickets array and finds the selected ticket
        for (int i = 0; i < allTickets.size(); i++) {
            if (selectedOrder.getTicketID() ==
allTickets.get(i).getTicketID())
                selectedTicket = allTickets.get(i);
            }
        return selectedTicket;
    }
    else
        return selectedTicket = new Ticket();
}
}

```

```

// ScheduleInfo Class - created by Marius

package SEJ.ApplicationLayer;

import SEJ.ApplicationLayer.DataTypes.Schedule;
import SEJ.DataAccessLayer.ScheduleSQL;
import java.util.List;

public class ScheduleInfo {
    private static List<Schedule> schedules;
    static Schedule schedule;

    // unused method
    // gets all schedules from the Data Access Layer
    public static List<Schedule> selectAllSchedules() throws Exception
    {
        schedules = ScheduleSQL.loadAllSchedules();
        return schedules;
    }

    // sends schedule to the Data Access Layer
    public static void createSchedule(int planeID, int legID) throws Exception{
        schedule = new Schedule(planeID, legID);
        ScheduleSQL.createSchedule(schedule);
    }
}

```

```

// SeatInfo Class - created by Dana

// used to handle seats
package SEJ.ApplicationLayer;

import SEJ.ApplicationLayer.DataTypes.Leg;
import SEJ.ApplicationLayer.DataTypes.Plane;
import SEJ.ApplicationLayer.DataTypes.Schedule;
import SEJ.DataAccessLayer.ScheduleSQL;
import java.util.ArrayList;
import java.util.List;

public class SeatInfo {
    static int[] seats;
    static Plane searchedPlane = new Plane();

    public static int method(Leg leg, int ticketClass) throws Exception{
        List<Plane> planes = PlaneInfo.selectAllPlanes();

        // loops through the planes array, finds the plane corresponding to that
leg
        for(int i = 0; i < planes.size(); i++)
            if(planes.get(i).getPlaneID() ==
ScheduleSQL.getSearchedPlane(leg.getLegID())) // the plane is found by executing
an SQL
                // query
                searchedPlane = planes.get(i); // in the Data Access Layer
                // check ScheduleSQL class for
details

        int maximumSeats = searchedPlane.getFirstClassSeatTotal() +
searchedPlane.getBusinessSeatTotal() + searchedPlane.getCoachSeatTotal();

        seats = new int[maximumSeats]; // the length of the array is set to how
many seats the plane has in total

        for (int i = 0; i < seats.length; i++)
            seats[i] = 0; // each slot of the array is
instantiated with 0, meaning all seats are available

        return handleSeats(leg, ticketClass);
    }

    // calls the corresponding method to the ticket class
    // returns the seat number
    public static int handleSeats(Leg leg, int ticketClass){
        int seatNumber = 0;
        if (ticketClass == 1) {
            seatNumber = bookFirstClass(leg);
        }
        if (ticketClass == 2) {
            seatNumber = bookBusinessClass(leg);
        }
        if (ticketClass == 3) {
            seatNumber = bookEconomyClass(leg);
        }
        return seatNumber;
    }
}

```

```

        // if it is a first class
        // the counting is between how many first class tickets are already booked
        and the maximum amount of first class seats
        // when an available seat is found, it is set to 1, meaning it becomes booked
        // it returns index + 1 as seat number (taking into consideration that
        counting in an array starts from 0)
        // if there are no available seats, returns -1
        private static int bookFirstClass(Leg leg) {
            for (int i = leg.getFirstSeatBooked(); i <
searchedPlane.getFirstClassSeatTotal(); i++) {
                if (seats[i] == 0) {
                    seats[i] = 1;
                    return i + 1;
                }
            }
            return -1;
        }

        // if it is a business class
        // the counting is between total first class + how many business seats are
        already booked
        // and the first class seats + business class in total
        // it returns index + 1 as seat number (taking into consideration that
        counting in an array starts from 0)
        // if there are no available seats, returns -1
        private static int bookBusinessClass(Leg leg) {
            for (int i = searchedPlane.getFirstClassSeatTotal() +
leg.getBusinessSeatBooked();
                i < searchedPlane.getFirstClassSeatTotal() +
searchedPlane.getBusinessSeatTotal(); i++) {
                if (seats[i] == 0) {
                    seats[i] = 1;
                    return i + 1;
                }
            }
            return -1;
        }

        // if it is an economy class
        // the counting is between total first class + total business class + how
        many economy seats are already booked
        // and the total number of seats in the plane
        // it returns index + 1 as seat number (taking into consideration that
        counting in an array starts from 0)
        // if there are no available seats, returns -1
        private static int bookEconomyClass(Leg leg) {
            for (int i = searchedPlane.getFirstClassSeatTotal() +
searchedPlane.getBusinessSeatTotal() + leg.getCoachSeatBooked();
                i < seats.length; i++)
            {
                if (seats[i] == 0) {
                    seats[i] = 1;
                    return i + 1;
                }
            }
            return -1;
        }
    }
}

```


// TicketInfo Class - created by Marius

```
package SEJ.ApplicationLayer;

import SEJ.ApplicationLayer.DataTypes.Leg;
import SEJ.ApplicationLayer.DataTypes.Ticket;
import SEJ.DataAccessLayer.TicketSQL;
import SEJ.PresentationLayer.CustomerServiceStaff;
import javafx.scene.control.Alert;
import javafx.scene.control.ButtonType;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.time.LocalDate;
import java.time.ZoneId;
import java.time.temporal.ChronoUnit;
import java.time.temporal.Temporal;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.Optional;

public class TicketInfo {
    private static List<Ticket> tickets;
    static List<Ticket> newBookedTickets = new ArrayList<>();

    // gets all tickets from the Data Access Layer
    public static List<Ticket> selectAllTickets() throws Exception
    {
        tickets = TicketSQL.loadAllTickets();
        return tickets;
    }

    // converts ticket class from int to String
    // sends necessary information to the Data Access Layer for updating the
tables
    // created by Dana
    public static void cancelTicketClicked(Ticket ticket) throws Exception{
        int ticket_id = ticket.getTicketID();
        int leg_id = ticket.getLegID();
        int ticketClass = ticket.getTicketClass();
        String leg_column_to_update = "";
        if (ticketClass == 1)
            leg_column_to_update = "first_seat_booked";
        if (ticketClass == 2)
            leg_column_to_update = "business_seat_booked";
        if (ticketClass == 3)
            leg_column_to_update = "coach_seat_booked";
        TicketSQL.deleteTicket(ticket_id, leg_id, leg_column_to_update);
    }

    // returns the date of the leg on which the ticket is booked
    // used for calculating how many days there are until the departure date
    // created by Felix
    public static String refundTicketClicked(Ticket ticket) throws Exception {
        int leg_id = ticket.getLegID();
        List<Leg> allLegs = LegInfo.selectAllLegs();
        String leg_date = "";
        // loops through the legs array and finds the leg date
        for (int i = 0; i < allLegs.size(); i++) {
```



```

        if (leg_id == allLegs.get(i).getLegID()) {
            leg_date = allLegs.get(i).getDepartureDate();
        }
    }
    return leg_date;
}

// sends necessary information to the Data Access Layer (the ticket id) for
// updating tables
public static void confirmTicketClicked(Ticket ticket) throws Exception{
    TicketsSQL.confirmTicket(ticket.getTicketID());
}

// calculates the date difference between the present date and the departure
// date of a leg
public static long calculateDateDifference(String leg_date) throws Exception
{
    DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
    Date today = new Date();
    dateFormat.format(today);
    Date legDate = dateFormat.parse(leg_date);
    LocalDate localDateLEG =
legDate.toInstant().atZone(ZoneId.systemDefault()).toLocalDate();
    LocalDate localDateTODAY =
today.toInstant().atZone(ZoneId.systemDefault()).toLocalDate();
    long days = ChronoUnit.DAYS.between(localDateTODAY, localDateLEG);
    return days;
}

// sends ticket to the Data Access Layer
// adds it to the array
// sends information for updating the seats
// created by Lei
public static void bookTicket(Ticket ticket) throws Exception
{
    TicketsSQL.createTicket(ticket);
    tickets.add(ticket);
    LegInfo.updateSeat(ticket.getLegID(), ticket.getTicketClass());
}

// unused method
// it is meant to avoid creating of Ticket objects in the Presentation Layer
public static void createTicket(int ticketId, int legId, int customerId, int
ticketClass, int seatNr, int ticketPrice, String ticketState) throws Exception{
    Ticket ticket = new Ticket(ticketId, legId, customerId, ticketClass,
seatNr, ticketPrice, ticketState);
    TicketsSQL.createTicket(ticket);
}

// unused method
// it is meant to avoid adding ticket objects to an array in the
// Presentation Layer
public static void addTicketToArray(int ticketId, int legId, int customerId,
int ticketClass, int seatNr, int ticketPrice, String ticketState){
    newBookedTickets = new ArrayList<>();
    Ticket ticket = new Ticket(ticketId, legId, customerId, ticketClass,
seatNr, ticketPrice, ticketState);
    newBookedTickets.add(ticket);
}
}

```


// Class to read from and write to the 'addresses' table in the database - created by Dana

```
package SEJ.DataAccessLayer;

import SEJ.ApplicationLayer.DataTypes.Address;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;

public class AddressSQL {

    // receive data from database, put in array
    public static List<Address> loadAllAddresses() throws Exception {
        List<Address> addresses = new ArrayList<>();
        Connection con = MySqlConnection.getConnection();
        Statement st = con.createStatement();
        String sql = ("SELECT * FROM addresses");
        ResultSet rs = st.executeQuery(sql);
        while (rs.next()) {
            int address_id = rs.getInt("address_id");
            String street_number = rs.getString("address_street_number");
            String address_city = rs.getString("address_city");
            int address_zip_code = rs.getInt("address_zip_code");
            String address_country = rs.getString("address_country");
            addresses.add(new Address(address_id, street_number, address_city,
address_zip_code, address_country));
        }
        con.close();
        rs.close();
        st.close();
        return addresses;
    }

    //create new address and add it to addresses table
    public static void createAddress(Address address) throws Exception{
        Connection con = MySqlConnection.getConnection();
        Statement st = con.createStatement();
        String sql = ("INSERT INTO addresses VALUES (DEFAULT, "
            + "\"" + address.getStreetAndNumber() + "\"" + ", "
            + "\"" + address.getCity() + "\"" + ", "
            + address.getZip() + ", "
            + "\"" + address.getCountry() + "\"" + ")");
        st.executeUpdate(sql);
        con.close();
        st.close();
    }
}
```

// Class to read from and write to the 'customers' table in the database - created by Dana

```
package SEJ.DataAccessLayer;

import SEJ.ApplicationLayer.DataTypes.Customer;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;

public class CustomerSQL {

    private static ArrayList<Customer> customers = new ArrayList<Customer>();

    //receive data, put in array
    public static List<Customer> loadAllCustomers() throws Exception {
        Connection con = MySqlConnection.getConnection();
        Statement st = con.createStatement();
        String sql = ("SELECT * FROM customers");
        ResultSet rs = st.executeQuery(sql);
        while (rs.next()) {
            int customer_id = rs.getInt("customer_id");
            String customer_first_name = rs.getString("customer_first_name");
            String customer_last_name = rs.getString("customer_last_name");
            String customer_gender = rs.getString("customer_gender");
            String customer_birthday = rs.getString("customer_birthday");
            String customer_nationality = rs.getString("customer_nationality");
            int pasNumber = rs.getInt("passport_number");
            String passExpirationDate = rs.getString("passport_expiration_date");
            int address_id = rs.getInt("address_id");
            String customer_email = rs.getString("customer_email");
            String customer_phone = rs.getString("customer_phone_number");
            customers.add(new Customer(customer_id, customer_first_name,
customer_last_name, customer_gender,
            customer_birthday, customer_nationality, pasNumber,
            passExpirationDate, address_id, customer_email,
            customer_phone));
        }
        con.close();
        rs.close();
        st.close();
        return customers;
    }

    //create new customer and add it to customers table
    public static void createCustomer(Customer customer) throws Exception{
        Connection con = MySqlConnection.getConnection();
        Statement st = con.createStatement();
        String sql = ("INSERT INTO customers VALUES (DEFAULT" + ", "
            + "\"" + customer.getFirstName() + "\"" + ", "
            + "\"" + customer.getLastName() + "\"" + ", "
            + "\"" + customer.getGender() + "\"" + ", "
            + "\"" + customer.getBirthday() + "\"" + ", "
            + "\"" + customer.getNationality() + "\"" + ", "
            + customer.getPasNumber() + ", "
            + "\"" + customer.getPassExpirationDate() + "\"" + ", "
            + customer.getAddressID() + ", "

```

```
        + "\"" + customer.getEmail() + "\" + ", "
        + "\"" + customer.getPhone() + "\" + ")";
    st.executeUpdate(sql);
    con.close();
    st.close();
}
}
```

// Class to write from and to the 'employees' table in the database - created by Dana

```
package SEJ.DataAccessLayer;

import SEJ.ApplicationLayer.DataTypes.Employee;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;

public class EmployeesSQL {
    private static ArrayList<Employee> employees = new ArrayList<Employee>();

    // receive data from database, put in array
    public static List<Employee> loadAllEmployees() throws Exception {
        Connection con = MySqlConnection.getConnection();
        Statement st = con.createStatement();
        String sql = ("SELECT * FROM employees");
        ResultSet rs = st.executeQuery(sql);
        while (rs.next()) {
            byte employee_id = rs.getBytes("employee_id");
            String employee_first_name = rs.getString("employee_first_name");
            String employee_last_name = rs.getString("employee_last_name");
            String employee_title = rs.getString("employee_title");
            String employee_username = rs.getString("employee_username");
            String employee_password = rs.getString("employee_password");
            employees.add(new Employee(employee_id, employee_first_name,
employee_last_name, employee_title,
                                employee_username, employee_password));
        }
        con.close();
        rs.close();
        st.close();
        return employees;
    }
}
```

// Class to write from and to the 'legs' table in the database - created by Dana

```
package SEJ.DataAccessLayer;

import SEJ.ApplicationLayer.DataTypes.Leg;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;

public class LegSQL {

    //receive data, put in array
    public static List<Leg> loadAllLegs() throws Exception {
        List<Leg> legs = new ArrayList<>();
        Connection con = MySqlConnection.getConnection();
        Statement st = con.createStatement();
        String sql = ("SELECT * FROM legs");
        ResultSet rs = st.executeQuery(sql);
        while (rs.next()) {
            int leg_id = rs.getInt("leg_id");
            String departure_airport = rs.getString("departure_airport");
            String arrival_airport = rs.getString("arrival_airport");
            String departure_date = rs.getString("departure_date");
            String arrival_date = rs.getString("arrival_date");
            String departure_time = rs.getString("departure_time");
            String arrival_time = rs.getString("arrival_time");
            byte first_seat_booked = (byte) rs.getInt("first_seat_booked");
            byte business_seat_booked = (byte) rs.getInt("business_seat_booked");
            short coach_seat_booked = (short) rs.getInt("coach_seat_booked");
            int price = rs.getInt("price_first_class");
            legs.add(new Leg(leg_id, departure_airport, arrival_airport,
departure_date, departure_time, arrival_date,
                                arrival_time, first_seat_booked,
business_seat_booked, coach_seat_booked, price));
        }
        con.close();
        rs.close();
        st.close();
        return legs;
    }

    // method used to increse number of seats booked for a specific class
    public static void updateSeatBookedState(int legId, int classType) throws
Exception
    {
        Connection con = MySqlConnection.getConnection();
        Statement st = con.createStatement();
        String sql = "";
        if(classType == 1)
        {
            sql = ("UPDATE legs \n"
                + "SET first_seat_booked" + " = first_seat_booked + 1 "
                + "WHERE leg_id = " + legId);
        }
        if(classType == 2 )
        {
            sql = ("UPDATE legs \n"
```

```

        + "SET business_seat_booked" + " = business_seat_booked + 1"
        + "WHERE leg_id = " + legId);    }
if(classType == 3)
{
    sql = ("UPDATE legs \n"
        + "SET coach_seat_booked" + " = coach_seat_booked + 1 "
        + "WHERE leg_id = " + legId);
}
st.executeUpdate(sql);
con.close();
st.close();
}

//create new leg and add it to legs table
public static void createLeg(Leg leg) throws Exception{
    Connection con = MySqlConnection.getConnection();
    Statement st = con.createStatement();
    String sql = ("INSERT INTO legs VALUES (" + leg.getLegID() + ", "
        + "\"" + leg.getDepartureAirport() + "\"" + ", "
        + "\"" + leg.getArrivalAirport() + "\"" + ", "
        + "\"" + leg.getDepartureDate() + "\"" + ", "
        + "\"" + leg.getDepartureTime() + "\"" + ", "
        + "\"" + leg.getArrivalDate() + "\"" + ", "
        + "\"" + leg.getArrivalTime() + "\"" + ", "
        + leg.getFirstSeatBooked() + ", "
        + leg.getBusinessSeatBooked() + ", "
        + leg.getCoachSeatBooked() + ", "
        + leg.getPrice()+ ")");
    st.executeUpdate(sql);
    con.close();
    st.close();
}
}

```



```

// Class to connect to the database - created by Dana

package SEJ.DataAccessLayer;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class MySqlConnection {
    static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
    static final String DATABASE_URL = "jdbc:mysql://localhost:3306/sej";
    static Connection con;

    // Establishing the connection
    public static Connection connect() throws SQLException {
        try{
            con = null;
            Class.forName (JDBC_DRIVER);
        }catch(ClassNotFoundException cnfe){
            System.err.println("Error: "+cnfe.getMessage());
        }

        // in the url we have to tell which account and password to use
        con = DriverManager.getConnection(DATABASE_URL, "root",
"numeletrandafirului2");
        return con;
    }

    // returns the open connection
    // used in the SQL classes
    public static Connection getConnection() throws SQLException,
ClassNotFoundException{
        if(con !=null && !con.isClosed())
            return con;
        connect();
        return con;
    }
}

```

// Class to write from and to the 'orders' table in the database - created by Dana

```
package SEJ.DataAccessLayer;

import SEJ.ApplicationLayer.DataTypes.Order;
import SEJ.ApplicationLayer.DataTypes.OrderContent;
import SEJ.ApplicationLayer.DataTypes.Ticket;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;

public class OrderSQL {

    // receive data from database, put in array
    public static List<Order> loadAllOrders() throws Exception {
        Connection con = MySqlConnection.getConnection();
        Statement st = con.createStatement();
        String sql = ("SELECT * FROM orders");
        ResultSet rs = st.executeQuery(sql);
        ArrayList<Order> orders = new ArrayList<Order>();
        while (rs.next()) {
            int order_id = rs.getInt("order_id");
            int employee_id = rs.getInt("employee_id");
            int customer_id = rs.getInt("customer_id");
            orders.add(new Order(order_id, employee_id, customer_id));
        }
        con.close();
        rs.close();
        st.close();
        return orders;
    }

    //create new order and add it to orders table
    public static void createOrder(Order order) throws Exception{
        Connection con = MySqlConnection.getConnection();
        Statement st = con.createStatement();
        String sql = ("INSERT INTO orders VALUES (DEFAULT, "
            + order.getEmployeeID() + ", "
            + order.getCustomerID() + ")");
        st.executeUpdate(sql);
        con.close();
        st.close();
    }

    // method used for searching an order - created by Lei
    public static List<OrderContent> createOrderContent() throws Exception
    {
        List<OrderContent> allOrderContent = new ArrayList<>();
        Connection con = MySqlConnection.getConnection();
        Statement st = con.createStatement();
        String sql = "SELECT order_id, t.ticket_id, customer_first_name,
customer_last_name, t.ticket_class, t.ticket_price, t.ticket_seat_number,
t.ticket_state\n" +
            "FROM orders_tickets ot \n" +
            "JOIN tickets t\n" +
            "ON ot.ticket_id = t.ticket_id\n" +
            "JOIN customers c \n" +
```

```

        "ON t.customer_id = c.customer_id;";
ResultSet rs = st.executeQuery(sql);
while(rs.next())
{
    int orderID = rs.getInt("order_id");
    int ticketID = rs.getInt("ticket_id");
    String customerFirstName = rs.getString("customer_first_name");
    String customerLastName = rs.getString("customer_last_name");
    int ticketClass = rs.getInt("ticket_class");
    int ticketPrice = rs.getInt("ticket_price");
    int ticketSeatNumber = rs.getInt("ticket_seat_number");
    String ticketState = rs.getString("ticket_state");
    OrderContent orderContent = new OrderContent(orderID, ticketID,
customerFirstName, customerLastName,
        ticketClass, ticketPrice, ticketSeatNumber, ticketState);
    allOrderContent.add(orderContent);
}
return allOrderContent;
}

// updating the orders_tickets table
public static void updateOrders_Tickets(Order order) throws Exception{
    Connection con = MySqlConnection.getConnection();
    Statement st = con.createStatement();
    for(int i = 0; i < order.getTickets().size(); i++) {
        String sql = ("INSERT INTO orders_tickets VALUES (" +
order.getOrderID() + ", "
        + order.getTickets().get(i).getTicketID() + ") ");
        st.executeUpdate(sql);
    }
    con.close();
    st.close();
}
}

```

// Class to write from and to the 'planes' table in the database - created by Dana

```
package SEJ.DataAccessLayer;

import SEJ.ApplicationLayer.DataTypes.Plane;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;

public class PlaneSQL {
    private static ArrayList<Plane> planes = new ArrayList<Plane>();

    // receive data from database, put in array
    public static List<Plane> loadAllPlanes() throws Exception {
        Connection con = MySqlConnection.getConnection();
        Statement st = con.createStatement();
        String sql = ("SELECT * FROM planes");
        ResultSet rs = st.executeQuery(sql);
        while (rs.next()) {
            int plane_id = rs.getInt("plane_id");
            String plane_type = rs.getString("plane_type");
            int first_class_seats = rs.getInt("first_class_seats");
            int business_class_seats = rs.getInt("business_class_seats");
            int economy_class_seats = rs.getInt("economy_class_seats");
            planes.add(new Plane(plane_id, plane_type, first_class_seats,
business_class_seats, economy_class_seats));
        }
        con.close();
        rs.close();
        st.close();
        return planes;
    }

    //create new plane and add it to planes table
    public static void createPlane(Plane plane) throws Exception{
        Connection con = MySqlConnection.getConnection();
        Statement st = con.createStatement();
        String sql = ("INSERT INTO planes VALUES (" + plane.getPlaneID() + ", "
            + "\"" + plane.getPlaneType() + "\"" + ", "
            + plane.getFirstClassSeatTotal() + ", "
            + plane.getBusinessSeatTotal() + ", "
            + plane.getCoachSeatTotal() + ")");
        st.executeUpdate(sql);
        con.close();
        st.close();
    }
}
```

// Class to write from and to the 'schedules' table in the database - created by Dana

```
package SEJ.DataAccessLayer;

import SEJ.ApplicationLayer.DataTypes.Schedule;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;

public class SchedulesSQL {
    private static ArrayList<Schedule> schedules = new ArrayList<Schedule>();

    // receive data from database, put in array
    public static List<Schedule> loadAllSchedules() throws Exception {
        Connection con = MySqlConnection.getConnection();
        Statement st = con.createStatement();
        String sql = ("SELECT * FROM schedules");
        ResultSet rs = st.executeQuery(sql);
        while (rs.next()) {
            int plane_id = rs.getInt("plane_id");
            int leg_id = rs.getInt("leg_id");
            schedules.add(new Schedule(plane_id, leg_id));
        }
        con.close();
        rs.close();
        st.close();
        return schedules;
    }

    //create new schedule and add it to schedules table
    public static void createSchedule(Schedule schedule) throws Exception{
        Connection con = MySqlConnection.getConnection();
        Statement st = con.createStatement();
        String sql = ("INSERT INTO schedules VALUES (" + schedule.getPlaneID() +
", " + schedule.getLegID() + ")");
        st.executeUpdate(sql);
        con.close();
        st.close();
    }

    // returns a the plane id that has the leg id specified in the schedules table
    // used for handling seats and booking a ticket
    public static int getSearchedPlane(int legId) throws Exception{
        int plane_id = 0;
        Connection con = MySqlConnection.getConnection();
        Statement st = con.createStatement();
        String sql = ("SELECT plane_id\n" +
            "FROM schedules\n" +
            "WHERE leg_id = " + legId);
        ResultSet rs = st.executeQuery(sql);
        while (rs.next()) {
            plane_id = rs.getInt("plane_id");
        }
        con.close();
        rs.close();
    }
}
```

```
        st.close();  
        return plane_id;  
    }  
}
```

// Class to write from and to the 'tickets' table in the database - created by Dana

```
package SEJ.DataAccessLayer;

import SEJ.ApplicationLayer.DataTypes.Ticket;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;

public class TicketSQL {
    private static ArrayList<Ticket> tickets = new ArrayList<Ticket>();

    // receive data from database, put in array
    public static List<Ticket> loadAllTickets() throws Exception {
        Connection con = MySqlConnection.getConnection();
        Statement st = con.createStatement();
        String sql = ("SELECT * FROM tickets");
        ResultSet rs = st.executeQuery(sql);
        while (rs.next()) {
            int ticket_id = rs.getInt("ticket_id");
            int leg_id = rs.getInt("leg_id");
            int customer_id = rs.getInt("customer_id");
            int ticket_class = rs.getInt("ticket_class");
            int ticket_seat_number = rs.getInt("ticket_seat_number");
            int ticket_price = rs.getInt("ticket_price");
            String ticket_state = rs.getString("ticket_state");
            tickets.add(new Ticket(ticket_id, leg_id, customer_id, ticket_class,
ticket_seat_number, ticket_price, ticket_state));
        }
        con.close();
        rs.close();
        st.close();
        return tickets;
    }

    //create new ticket and add it to tickets table
    public static void createTicket(Ticket ticket) throws Exception{
        Connection con = MySqlConnection.getConnection();
        Statement st = con.createStatement();
        String sql = ("INSERT INTO tickets VALUES (DEFAULT, "
            + ticket.getLegID() + ", "
            + ticket.getCustomerID() + ", "
            + ticket.getTicketClass() + ", "
            + ticket.getTicketSeatNumber() + ", "
            + ticket.getTicketPrice() + ", "
            + "\"\" + ticket.getTicketState() + "\"\" + ")");
        st.executeUpdate(sql);
        con.close();
        st.close();
    }

    // updates the tables when a ticket is deleted
    public static void deleteTicket(int ticketID, int legID, String ticketClass)
throws Exception{
        Connection con = MySqlConnection.getConnection();
        Statement st = con.createStatement();
```

```

        String sql = ("DELETE from tickets\n" +
            "WHERE ticket_id = " + ticketID);
        String sql2 = ("DELETE from orders_tickets\n" +
            "WHERE ticket_id = " + ticketID + " AND order_id = (SELECT order_id FROM
\n" +
            "(SELECT order_id FROM orders_tickets WHERE ticket_id = " +
ticketID + ") x)");
        String sql3 = ("UPDATE legs \n" +
            "SET " + ticketClass + " = " + ticketClass + " - 1\n" +
            "WHERE leg_id = " + legID );
        st.executeUpdate(sql);
        st.executeUpdate(sql2);
        st.executeUpdate(sql3);
        con.close();
        st.close();
    }

    // updates the ticket state from 'booked' to 'confirmed' in the tickets table
    public static void confirmTicket(int ticketID) throws Exception{
        Connection con = MySqlConnection.getConnection();
        Statement st = con.createStatement();
        String sql = ("UPDATE tickets\n" +
            "SET ticket_state = 'confirmed'\n" +
            "WHERE ticket_id = " + ticketID);
        st.executeUpdate(sql);
        con.close();
        st.close();
    }
}

```



```
// AdminScene Class - created by Felix
```

```
package SEJ.PresentationLayer;

import SEJ.ApplicationLayer.*;
import SEJ.ApplicationLayer.DataTypes.Employee;
import SEJ.ApplicationLayer.DataTypes.Leg;
import SEJ.ApplicationLayer.DataTypes.Plane;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.input.KeyCode;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
import javafx.util.StringConverter;
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.util.ArrayList;
import java.util.List;

public class AdminScene {
    static Stage window;
    static Button buttonAddPlane, buttonSavePlane, buttonNewSchedule,
buttonNewLeg, buttonCSV, buttonLogOut;
    static TextField textFieldPlaneType, textFieldFirstClass,
textFieldBusinessClass, textFieldEconClass, textFieldDepartAirport,
textFieldArrivalAirport,
        textFieldDepartureTime, textFieldArrivalTime, textFieldPrice;
    static Label labelPlaneType, labelFirstClass, labelBusinessClass,
labelEconClass, labelSelectPlane, labelAirport, labelDeparture, labelArrival,
labelPrice;
    static GridPane gridPane;
    static HBox layout, hBoxNewFlightButtons;
    static List<Plane> allPlanes = new ArrayList<>();
    static List<Leg> allLegs = new ArrayList<>();
    static List<Employee> allEmployees = new ArrayList<>();
    static ComboBox<String> comboBoxPlanes;
    static DatePicker datePickerDepart, datePickerArrival;

    public static Scene returnAdminScene(Stage primaryStage) throws Exception{
        window = primaryStage;
        allPlanes = Main.allPlanes;
        allEmployees = Main.allEmployees;
        allLegs = Main.allLegs;
        gridPane = new GridPane();
        gridPane.setPadding(new Insets(10,10,10,10));
        gridPane.setHgap(10);
        gridPane.setVgap(15);
        gridPane.setId("gridPane");

        // The buttons on the left side are created, set on action and added to a
        VBox
        buttonNewSchedule = new Button("New Schedule");
        buttonNewSchedule.setMinWidth(120);
        buttonNewSchedule.setMaxWidth(120);
        buttonNewSchedule.setOnAction((e -> {
            try {
```

```

        newScheduleClicked();
    } catch (Exception e1) {
        e1.printStackTrace();
    }
});
buttonAddPlane = new Button("Add Plane");
buttonAddPlane.setMinWidth(120);
buttonAddPlane.setMaxWidth(120);
buttonAddPlane.setOnAction(e -> {
    try {
        addPlaneClicked();
    } catch (Exception e1) {
        e1.printStackTrace();
    }
});
buttonCSV = new Button();
buttonCSV = OutputCSV.makeOutputDatabase();
buttonCSV.setMinWidth(120);
buttonCSV.setMaxWidth(120);
buttonLogOut = new Button("Log out");
buttonLogOut.setMinWidth(120);
buttonLogOut.setMaxWidth(120);

// logs out, goes back to the login scene
buttonLogOut.setOnAction(e -> {
    try{
        VBox login = LogInScene.makeLogInScene_logo(window);
        Scene scene = new Scene(login, 400, 500);
        scene.getStylesheets().add("CSS.css");
        window.setScene(scene);
    }
    catch (Exception e2){
        e2.printStackTrace();
    }
});
VBox vBoxButtons = new VBox(15);
vBoxButtons.setPadding(new Insets(10,10,10,10));
vBoxButtons.setId("vBox");
vBoxButtons.getChildren().addAll(buttonNewSchedule, buttonAddPlane,
buttonCSV, buttonLogOut);

// the layout is a HBox, containing a VBox (the buttons on the left) and
a GridPane (right side)
layout = new HBox(50);
layout.getChildren().addAll(vBoxButtons);

// the empty new schedule GridPane pops up directly
// that is because we think scheduling a new leg is the most frequent
operation of the admin
newScheduleClicked();
Scene adminScene = new Scene(layout, 600, 450);
adminScene.getStylesheets().add("CustomerCSS");
return adminScene;
}

// when the add plane button is clicked
// the gridpane will contain information about the plane to be added( Labels,
TextFields)
public static void addPlaneClicked() throws Exception {
    buttonAddPlane.setDisable(true);

```

```

        buttonNewSchedule.setDisable(false);
        gridPane.getChildren().clear();
        layout.getChildren().remove(gridPane);
        labelPlaneType = new Label("Plane Type: ");
        gridPane.setConstraints(labelPlaneType, 0, 0);
        textFieldPlaneType = new TextField();
        textFieldPlaneType.setPromptText("Plane Type");
        gridPane.setConstraints(textFieldPlaneType, 1, 0);
        labelFirstClass = new Label("First Class Seats: ");
        gridPane.setConstraints(labelFirstClass, 0, 1);
        textFieldFirstClass = new TextField();
        textFieldFirstClass.setPromptText("# first class seats");
        gridPane.setConstraints(textFieldFirstClass, 1, 1);
        addListenerToTextFieldProperty(textFieldFirstClass);
        labelBusinessClass = new Label("Business Class Seats: ");
        gridPane.setConstraints(labelBusinessClass, 0, 2);
        textFieldBusinessClass = new TextField();
        textFieldBusinessClass.setPromptText("# business class seats");
        gridPane.setConstraints(textFieldBusinessClass, 1, 2);
        addListenerToTextFieldProperty(textFieldBusinessClass);
        labelEconClass = new Label("Economy Class Seats: ");
        gridPane.setConstraints(labelEconClass, 0, 3);
        textFieldEconClass = new TextField();
        textFieldEconClass.setPromptText("# economy class seats");
        gridPane.setConstraints(textFieldEconClass, 1, 3);
        textFieldEconClass.textProperty().addListener((observable, oldValue,
newValue) -> {
            if (!CheckInputFormat.isNumeric(textFieldEconClass.getText())) {
                String alertText = "please enter numeric value"; // checking for
the right input
                ShowAlert.makeErrorAlert(alertText);
                textFieldEconClass.clear();
            }
        });
        textFieldEconClass.setOnKeyPressed(e -> {
            if (e.getCode() == KeyCode.ENTER) {
                savePlaneButtonClicked(); // on enter pressed it saves the
plane
            }
        });
        buttonSavePlane = new Button("Save");
        gridPane.setConstraints(buttonSavePlane, 1, 4);
        buttonSavePlane.setOnAction(e -> savePlaneButtonClicked());
        gridPane.getChildren().addAll(labelPlaneType, textFieldPlaneType,
labelFirstClass, textFieldFirstClass, labelBusinessClass,
textFieldBusinessClass, labelEconClass, textFieldEconClass,
buttonSavePlane);
        layout.getChildren().add(gridPane);
    }

    public static void savePlaneButtonClicked() {
        if(textFieldPlaneType.getText().isEmpty() ||
textFieldFirstClass.getText().isEmpty() ||
textFieldBusinessClass.getText().isEmpty() ||
textFieldEconClass.getText().isEmpty()) {
            String alertText = "Please fill out all fields.";
            ShowAlert.makeErrorAlert(alertText);
        } else {
            // the information is collected from the TextFields and sent to the
Application Layer for a plane to be created there

```

```

        try {
            PlaneInfo.createPlane(textFieldPlaneType.getText(),
Integer.parseInt(textFieldFirstClass.getText()),
Integer.parseInt(textFieldBusinessClass.getText()),
Integer.parseInt(textFieldEconClass.getText()));
            String alertText = textFieldPlaneType.getText() + " has been
added to the fleet.";
            ShowAlert.makeInformationAlert(alertText);    // confirms adding
a new plane
        } catch (Exception e1) {
            e1.printStackTrace();
        }
    }
    textFieldPlaneType.clear();    // clearing the TextFields
    textFieldFirstClass.clear();
    textFieldBusinessClass.clear();
    textFieldEconClass.clear();
}

public static void newScheduleClicked() throws Exception {
    // it creates the Labels, TextFields and DatePickers for new schedule
    gridPane.getChildren().clear();
    layout.getChildren().remove(gridPane);
    buttonNewSchedule.setDisable(true);
    buttonAddPlane.setDisable(false);
    labelSelectPlane = new Label("Select Plane:");
    gridPane.setConstraints(labelSelectPlane, 0, 0);
    // ComboBox with the plane id and plane type
    comboBoxPlanes = new ComboBox<>();
    comboBoxPlanes.setPromptText("Select...");
    for(int i = 0; i < allPlanes.size(); i++) {
        comboBoxPlanes.getItems().addAll(allPlanes.get(i).getPlaneID() + " "
+ allPlanes.get(i).getPlaneType());
    }
    gridPane.setConstraints(comboBoxPlanes, 1, 0);
    labelAirport = new Label("Airport info:");
    gridPane.setConstraints(labelAirport, 0, 1);
    textFieldDepartAirport = new TextField();
    textFieldDepartAirport.setPromptText("Departure");
    gridPane.setConstraints(textFieldDepartAirport, 1, 1);
    textFieldArrivalAirport = new TextField();
    textFieldArrivalAirport.setPromptText("Arrival");
    gridPane.setConstraints(textFieldArrivalAirport, 1, 2);
    labelDeparture = new Label("Departure info:");
    gridPane.setConstraints(labelDeparture, 0, 3);
    datePickerDepart = new DatePicker();
    datePickerDepart.setPromptText("Date");
    changeDatePickerFormat(datePickerDepart);
    gridPane.setConstraints(datePickerDepart, 1, 3);
    textFieldDepartureTime = new TextField();
    textFieldDepartureTime.setPromptText("Time (hh:mm)");
    gridPane.setConstraints(textFieldDepartureTime, 1, 4);
    labelArrival = new Label("Arrival info:");
    gridPane.setConstraints(labelArrival, 0, 5);
    datePickerArrival = new DatePicker();
    datePickerArrival.setPromptText("Date");
    changeDatePickerFormat(datePickerArrival);
    gridPane.setConstraints(datePickerArrival, 1, 5);
    textFieldArrivalTime = new TextField();
    textFieldArrivalTime.setPromptText("Time (hh:mm)");
}

```

```

gridPane.setConstraints(textFieldArrivalTime, 1, 6);
labelPrice = new Label("Price (kr):");
gridPane.setConstraints(labelPrice, 0, 7);
// the price has to be inputed for each leg for the first class
// the price for the other classes will be calculated according to this
price
textFieldPrice = new TextField();
textFieldPrice.setPromptText("Price (1st Class)");
gridPane.setConstraints(textFieldPrice, 1, 7);
textFieldPrice.setOnKeyPressed(e -> {
    if (e.getCode() == KeyCode.ENTER) {
        scheduleLegClicked();
    }
});
addListenerToTextFieldProperty(textFieldPrice);
buttonNewLeg = new Button("Schedule Leg");
buttonNewLeg.setOnAction(e -> scheduleLegClicked());
hBoxNewFlightButtons = new HBox(15);
hBoxNewFlightButtons.getChildren().addAll(buttonNewLeg);
gridPane.setConstraints(hBoxNewFlightButtons, 1, 8);
// adding all children to the gridpane
gridPane.getChildren().addAll(labelSelectPlane, comboBoxPlanes,
labelAirport, textFieldDepartAirport, textFieldArrivalAirport,
labelDeparture, datePickerDepart, textFieldDepartureTime,
labelArrival, datePickerArrival, textFieldArrivalTime,
labelPrice, textFieldPrice, hBoxNewFlightButtons);
// adding the gridpane to the layout
layout.getChildren().addAll(gridPane);
}

public static void scheduleLegClicked() {
    if (comboBoxPlanes.getValue() == null ||
textFieldDepartAirport.getText().isEmpty() ||
textFieldArrivalTime.getText().isEmpty() ||
datePickerDepart.getValue() == null ||
textFieldDepartureTime.getText().isEmpty() || datePickerArrival.getValue() ==
null ||
textFieldArrivalAirport.getText().isEmpty()) {
        String alertText = "Please fill out all fields";
        ShowAlert.makeErrorAlert(alertText); // checks if
all fields are filled out
    }
    else if (!CheckInputFormat.isTime(textFieldDepartureTime.getText())
|| !CheckInputFormat.isTime(textFieldArrivalTime.getText())) {
        String alertText = "Wrong time format.";
        ShowAlert.makeErrorAlert(alertText); // checks for
correct time format
    } else {
        try {
            // the information is taken from the TextFields, ComboBoxes and
DatePickers and sent to the Application Layer
LegInfo.createLeg(textFieldDepartAirport.getText(),
textFieldArrivalAirport.getText(), datePickerDepart.getEditor().getText(),
textFieldDepartureTime.getText(),
datePickerArrival.getEditor().getText(), textFieldArrivalTime.getText(),
Integer.parseInt(textFieldPrice.getText()));

ScheduleInfo.createSchedule(Integer.parseInt(comboBoxPlanes.getValue().substring(0
,1)), allLegs.size());

```

```

        String alertText = "The new schedule has been added to the
system.";
        ShowAlert.makeInformationAlert(alertText);
    } catch (Exception e1) {
        e1.printStackTrace();
    }
}

textFieldDepartAirport.clear(); // clears all TextFields, DatePickers
textFieldArrivalAirport.clear();
datePickerDepart.setValue(null);
datePickerDepart.getEditor().clear();
textFieldDepartureTime.clear();
datePickerArrival.setValue(null);
datePickerArrival.getEditor().clear();
textFieldArrivalTime.clear();
textFieldPrice.clear();
}

// adds a listener to the TextFields that should have numeric input
public static void addListenerToTextFieldProperty(TextField textField)
{
    textField.textProperty().addListener((observable, oldValue, newValue) ->
{
    if (!CheckInputFormat.isNumeric(textField.getText())) {
        String alertText = "Please enter numeric value.";
        ShowAlert.makeErrorAlert(alertText);
        textField.clear();
    }
});
}

/* The following method is copied from
http://code.makery.ch/blog/javafx-8-date-picker/
*/
// it changes the default format of 'yyyy-mm-dd' to 'dd/MM/yyyy'
public static void changeDatePickerFormat(DatePicker datePicker){
    String pattern = "dd/MM/yyyy";
    datePicker.setConverter(new StringConverter<LocalDate>() {
        DateTimeFormatter dateFormatter =
DateTimeFormatter.ofPattern(pattern);
        @Override
        public String toString(LocalDate date) {
            if (date != null) {
                return dateFormatter.format(date);
            } else {
                return "";
            }
        }
        @Override
        public LocalDate fromString(String string) {
            if (string != null && !string.isEmpty()) {
                return LocalDate.parse(string, dateFormatter);
            } else {
                return null;
            }
        }
    });
}
}
}

```


// BookTicket Class - created by Dana, Lei, Felix

```
package SEJ.PresentationLayer;

import SEJ.ApplicationLayer.*;
import SEJ.ApplicationLayer.DataTypes.*;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
import java.util.ArrayList;
import java.util.List;

public class BookTicket {
    static TextField phoneInput;
    static TextField emailInput;
    static TextField lastNameInput;
    static TextField firstNameInput;
    static ComboBox genderInput;
    static TextField birthdayInput;
    static TextField nationalityInput;
    static TextField passportNumberInput;
    static TextField streetAndNumberInput;
    static TextField cityInput;
    static TextField zipcodeInput;
    static TextField countryInput;
    static TextField expDate;
    static ComboBox comboBoxClass;
    static VBox orderViewBox;
    static HBox orderBuyButtons;
    static Label ticketPrice;
    static Label seatNo;
    static Label labelOrdes;
    static Label totalPrice;
    static Button makeOrderButton;
    static Button buyButton;
    static int total;

    // GridPane with Labels, Textfields, ComboBox
    // General information about the customer
    public static GridPane makeGeneralInfo(){
        //GridPane with 10px padding around edge
        GridPane grid = new GridPane();
        grid.setPadding(new Insets(10, 10, 10, 10));
        grid.setVgap(8);
        grid.setHgap(10);

        //First Name Label + TextField
        Label firstNameLabel = new Label("First name: ");
        grid.setConstraints(firstNameLabel, 0, 0);
        firstNameInput = new TextField();
        firstNameInput.setId("customerInputTextfield");
        firstNameInput.setPromptText("First name ");
        firstNameInput.setAlignment(Pos.CENTER);
        firstNameInput.setMaxWidth(100);
    }
}
```



```

grid.setConstraints(firstNameInput, 1, 0);

//Last Name Label + TextField
Label lastNameLabel = new Label("Last name: ");
grid.setConstraints(lastNameLabel, 0, 1);
lastNameInput = new TextField();
lastNameInput.setId("customerInputTextfield");
lastNameInput.setPromptText("Last name");
lastNameInput.setAlignment(Pos.CENTER);
lastNameInput.setMaxWidth(100);
grid.setConstraints(lastNameInput, 1, 1);

//Gender Label + ComboBox
Label gender = new Label("Gender:");
grid.setConstraints(gender, 0, 2);
genderInput = new ComboBox();
genderInput.setMinWidth(100);
genderInput.setPromptText("Select ...");
genderInput.getItems().addAll("Male", "Female");
grid.setConstraints(genderInput, 1, 2);

// Birthday Label + TextField
Label birthday = new Label("Date of Birth:");
grid.setConstraints(birthday, 0, 3);
birthdayInput = new TextField();
birthdayInput.setId("customerInputTextfield");
birthdayInput.setPromptText("dd/MM/yyyy");
birthdayInput.setAlignment(Pos.CENTER);
birthdayInput.setMaxWidth(100);
grid.setConstraints(birthdayInput, 1, 3);

// Nationality Label + TextField
Label nationality = new Label("Nationality:");
grid.setConstraints(nationality, 0, 4);
nationalityInput = new TextField();
nationalityInput.setId("customerInputTextfield");
nationalityInput.setPromptText("Nationality");
nationalityInput.setAlignment(Pos.CENTER);
nationalityInput.setMaxWidth(100);
grid.setConstraints(nationalityInput, 1, 4);

// Passport Number Label + TextField
Label passportNumber = new Label("Passport Number: ");
grid.setConstraints(passportNumber, 0, 5);
passportNumberInput = new TextField();
passportNumberInput.setId("customerInputTextfield");
passportNumberInput.setPromptText("Passport Nr.");
passportNumberInput.setAlignment(Pos.CENTER);
passportNumberInput.setMaxWidth(100);
grid.setConstraints(passportNumberInput, 1, 5);
addListenerToTextFieldProperty(passportNumberInput);

// Passport Expiration Date Label + TextField
Label pasExpDate = new Label("Valid through: ");
grid.setConstraints(pasExpDate, 0, 6);
expDate = new TextField();
expDate.setId("customerInputTextfield");
expDate.setPromptText("dd/MM/yyyy");
expDate.setAlignment(Pos.CENTER);
expDate.setMaxWidth(100);

```

```

        grid.setConstraints(expDate, 1, 6);
        grid.setAlignment(Pos.CENTER);
        grid.getChildren().addAll(firstNameLabel, firstNameInput, lastNameLabel,
        lastNameInput, gender, genderInput, birthday, birthdayInput, nationality,
        nationalityInput, passportNumber, passportNumberInput, pasExpDate, expDate);
        return grid;
    }

    // GridPane with Labels and Textfields
    // Address information
    public static GridPane addressInfo(){
        //GridPane with 10px padding around edge
        GridPane grid = new GridPane();
        grid.setPadding(new Insets(1, 1, 1, 1));
        grid.setVgap(8);
        grid.setHgap(5);

        // Street and Number Label + TextField
        Label street = new Label("Street&Nr.");
        grid.setConstraints(street, 0, 0);
        streetAndNumberInput = new TextField();
        streetAndNumberInput.setId("customerInputTextfield");
        streetAndNumberInput.setPromptText("Street and number");
        streetAndNumberInput.setAlignment(Pos.CENTER);
        grid.setConstraints(streetAndNumberInput, 1, 0);

        // City HBox
        HBox cityHbox = new HBox(2);
        Label city = new Label("City: ");
        cityInput = new TextField();
        cityInput.setId("customerInputTextfield");
        cityInput.setPromptText("City");
        cityInput.setAlignment(Pos.CENTER);
        cityInput.setMinWidth(80);
        cityInput.setMaxWidth(80);
        cityHbox.getChildren().addAll(city, cityInput);
        grid.setConstraints(cityHbox, 0, 1);

        // Zip code HBox
        HBox zipHbox = new HBox(5);
        Label zip = new Label("ZIP code: ");
        zipcodeInput = new TextField();
        zipcodeInput.setId("customerInputTextfield");
        zipcodeInput.setPromptText("zip");
        zipcodeInput.setAlignment(Pos.CENTER);
        zipcodeInput.setMinWidth(90);
        zipcodeInput.setMaxWidth(90);
        zipHbox.getChildren().addAll(zip, zipcodeInput);
        grid.setConstraints(zipHbox, 1, 1);
        addListenerToTextFieldProperty(zipcodeInput);
        Label country = new Label("Country: ");
        grid.setConstraints(country, 0, 2);
        countryInput = new TextField();
        countryInput.setId("customerInputTextfield");
        countryInput.setPromptText("Country");
        countryInput.setAlignment(Pos.CENTER);
        grid.setConstraints(countryInput, 1, 2);
        grid.getChildren().addAll(street, streetAndNumberInput, cityHbox,
        zipHbox, country, countryInput);
        grid.setAlignment(Pos.CENTER);
    }

```

```

        return grid;
    }

    // GridPane with Labels, TextFields
    // Contact information
    public static GridPane makeContactInfo(){
        //GridPane with 10px padding around edge
        GridPane grid = new GridPane();
        grid.setPadding(new Insets(1, 1, 1, 1));
        grid.setVgap(8);
        grid.setHgap(10);

        //Telephone Number Label + TextField
        Label phoneLabel = new Label("Phone Number:");
        grid.setConstraints(phoneLabel, 0, 0);
        phoneInput = new TextField();
        phoneInput.setId("customerInputTextfield");
        phoneInput.setPromptText("Phone Number");
        phoneInput.setAlignment(Pos.CENTER);
        grid.setConstraints(phoneInput, 1, 0);
        addListenerToTextFieldProperty(phoneInput);

        //Email Label + TextField
        Label emailLabel = new Label("Email:");
        grid.setConstraints(emailLabel, 0, 1);
        emailInput = new TextField();
        emailInput.setId("customerInputTextfield");
        emailInput.setPromptText("Email");
        emailInput.setAlignment(Pos.CENTER);
        grid.setConstraints(emailInput, 1, 1);
        grid.getChildren().addAll(phoneLabel, phoneInput, emailLabel,
emailInput);
        grid.setAlignment(Pos.CENTER);
        return grid;
    }

    // This is the layout of the left part of the scene - a VBox
    // it contains the leg information, the ticket information and the customer
input
    public static void makeLayout(Leg leg, Stage window) throws Exception {
        List<Ticket> newBookedTickets = new ArrayList<>();
        total = 0;

        VBox layout = new VBox(20);
        HBox hBox = new HBox(30);
        HBox layout1 = new HBox(30);

        Label customerInfo = new Label("Customer information ");
        Label contactInfo = new Label("Contact information ");
        Label addressInfo = new Label("Address");
        Label flightInfo = new Label("Flight information");
        Label flightType = new Label("Choose Class");

        VBox vbox2 = new VBox(5);
        vbox2.getChildren().addAll(flightInfo, makeLegInfo(leg));
        vbox2.setAlignment(Pos.CENTER);

        VBox vbox = new VBox(20);
        vbox.getChildren().addAll(customerInfo, makeGeneralInfo());
    }

```

```

vBox.setAlignment(Pos.CENTER);

VBox vBox1 = new VBox(30);
vBox1.getChildren().addAll(addressInfo, addressInfo(), contactInfo,
makeContactInfo());
vBox1.setAlignment(Pos.CENTER);

hBox.getChildren().addAll(vBox, vBox1);
hBox.setAlignment(Pos.CENTER);

VBox vBox3 = new VBox(20);
vBox3.getChildren().addAll(flightType, makeFlightTypeInfo(leg,
newBookedTickets));
vBox3.setAlignment(Pos.CENTER);

// generate order with tickets - the right side
orderViewBox = new VBox(10);
orderViewBox.setAlignment(Pos.CENTER_LEFT);

HBox saveAndNewCustomerBox = new HBox(20);

GridPane showTicketPane = new GridPane();
showTicketPane.setPrefSize(650, 540);
showTicketPane.setHgap(30);

Button saveButton = new Button("Save");
saveButton.setPrefWidth(100);

// save button is set on action
saveButton.setOnAction(e -> {
    if (genderInput.getValue() == null ||
firstNameInput.getText().isEmpty() || lastNameInput.getText().isEmpty()
    || birthdayInput.getText().isEmpty() ||
nationalityInput.getText().isEmpty()
    || passportNumberInput.getText().isEmpty() ||
expDate.getText().isEmpty()
    || streetAndNumberInput.getText().isEmpty() ||
cityInput.getText().isEmpty()
    || zipcodeInput.getText().isEmpty() ||
countryInput.getText().isEmpty()
    || comboBoxClass.getValue() == null ||
emailInput.getText().isEmpty()
    || phoneInput.getText().isEmpty()) {
        String alertText = "Please fill out all fields.";
        ShowAlert.makeErrorAlert(alertText); // checks if all
fields are filled out
    }
    else {
        if (!CheckInputFormat.isValidDate(expDate.getText())
|| !CheckInputFormat.isValidDate(birthdayInput.getText())) {
            showDateAlert(); // checks if the date (birthday,
passport expiration date) are in the right format
        } else
        try {
            orderViewBox.getChildren().removeAll(totalPrice,
orderBuyButtons);

            // saves the address and the customer input
            saveAddress();
            saveCustomerInput();

```

```

// we need the ticket class, the seat number and the
ticket price
        int classType =
convertClassToInt(comboBoxClass.getValue().toString());
        int seatNo = SeatInfo.method(leg, classType) +
newBookedTickets.size();
        int ticketPrice = LegInfo.getPrice(leg,
comboBoxClass.getValue().toString());
        // shows total price
        total = total + ticketPrice;
        totalPrice.setText("TOTAL: " + Integer.toString(total) +
" kr");

        // creates a ticket with all this information
        // it will be added to the newBookedTickets array
        Ticket ticket = new
Ticket(CustomerServiceStaff.ticketsSize + 1, leg.getLegID(),
CustomerServiceStaff.customersSize + 1,
        classType, seatNo, ticketPrice, "booked");
        CustomerServiceStaff.ticketsSize += 1;
        CustomerServiceStaff.customersSize += 1;
        newBookedTickets.add(ticket);

// generates ticket and adds it to the right side (under the 'Orders' Label)
        HBox ticketsViewBox =
GenerateTicketToBook.makeOneTicket(leg);
        showTicketPane.setConstraints(ticketsViewBox, 0,
newBookedTickets.size() - 1);
        showTicketPane.getChildren().add(ticketsViewBox);
        // all TextFields are cleared in case there is another
customer with on the same flight

// and on the same order
        clearInputTextFields();
        orderViewBox.getChildren().addAll(totalPrice,
orderBuyButtons);
    } catch (Exception e1) {
        e1.printStackTrace();
    }
    });
    saveAndNewCustomerBox.getChildren().addAll(saveButton);
    saveAndNewCustomerBox.setAlignment(Pos.CENTER);

    buyButton = new Button("Buy Tickets");
    buyButton.setPrefWidth(100);
    buyButton.setOnAction(e ->
    {
        try
        {
            // Send information about each ticket to the Application Layer
            // They will be sent to the database and the tables will be
updated
            for(Ticket ticket: newBookedTickets)
            {
                ticket.setTicketState("confirmed"); // the state of the
tickets is changed to 'confirmed'
                TicketInfo.bookTicket(ticket); // because the
tickets are bought directly
                OrderInfo.addTicketToOrder(ticket);
            }
        }
    }

```

```

    } catch (Exception e1) {
        e1.printStackTrace();
    }
    try
    {
        // creates a new order
        // information will be sent in the orders_tickets table as well
        OrderInfo.createOrders_Tickets(newBookedTickets);
        OrderInfo.createOrder(LogInScene.employeeID,
CustomerServiceStaff.customersSize - newBookedTickets.size() + 1);
    }
    catch (Exception e1) {
        e1.printStackTrace();
    }
    try {
        makeConfirmAlert();          // informs user that the tickets have
been bought
    } catch (Exception e2){
        e2.printStackTrace();
    }
    newBookedTickets.clear();
});
makeOrderButton = new Button("Make order");
makeOrderButton.setPrefWidth(100);
makeOrderButton.setOnAction(e ->
{
    try
    {
        // Send information about each ticket to the Application Layer
        // They will be sent to the database and the tables will be
updated
        for(Ticket ticket: newBookedTickets)
        {
            TicketInfo.bookTicket(ticket);
            OrderInfo.addTicketToOrder(ticket);
        }
    } catch (Exception e1) {
        e1.printStackTrace();
    }
    try
    {
        // creates a new order
        // information will be sent in the orders_tickets table as well
        OrderInfo.createOrders_Tickets(newBookedTickets);
        OrderInfo.createOrder(LogInScene.employeeID,
CustomerServiceStaff.customersSize - newBookedTickets.size() + 1);
    } catch (Exception e1) {
        e1.printStackTrace();
    }
    try {
        makeConfirmAlert();          // informs the user that the tickets
have been booked
    } catch (Exception e2){
        e2.printStackTrace();
    }
    newBookedTickets.clear();
    showTicketPane.getChildren().clear();
    orderViewBox.getChildren().removeAll(totalPrice, makeOrderButton);
});
orderBuyButtons = new HBox(20);
orderBuyButtons.getChildren().addAll(makeOrderButton, buyButton);
orderBuyButtons.setAlignment(Pos.CENTER);
labelOrdes = new Label("Orders");

```

```

        labelOrdes.setId("labelOrders");
        totalPrice = new Label();
        orderViewBox.getChildren().addAll(labelOrdes, showTicketPane);
        orderViewBox.setAlignment(Pos.CENTER);
        layout.getChildren().addAll(vBox2, vBox3, hBox, saveAndNewCustomerBox);
        layout.setAlignment(Pos.CENTER);
        layout.setId("vBox");

        // everything is added to the layout
        layout1.getChildren().addAll(layout, orderViewBox);
        Scene scene = new Scene(layout1, 1250, 700);
        scene.getStylesheets().add("CustomerCSS");
        window.setScene(scene);
        window.show();
    }

    // collects information from the TextFields and sends it to the Application
    Layer
    public static void saveCustomerInput() throws Exception{
        List<Address> allAddresses = AddressInfo.selectAllAddresses();

        CustomerInfo.createCustomer2(firstNameInput.getText(),
        lastNameInput.getText(), genderInput.getValue().toString(),
        birthdayInput.getText(), nationalityInput.getText(),
        Integer.parseInt(passportNumberInput.getText()), expDate.getText(),
        allAddresses.size(), emailInput.getText(), phoneInput.getText());
    }

    // collects information from the TextFields and sends it to the Application
    Layer
    public static void saveAddress() throws Exception{
        AddressInfo.createAddress(streetAndNumberInput.getText(),
        cityInput.getText(), Integer.parseInt(zipcodeInput.getText()),
        countryInput.getText());
    }

    // clears all TextFields, ComboBoxes
    public static void clearInputTextFields()
    {
        //clears the selected flight input
        comboBoxClass.setValue("Select...");
        seatNo.setText("");
        ticketPrice.setText("");

        //clear customer input
        firstNameInput.clear();
        lastNameInput.clear();
        genderInput.setValue(null);
        birthdayInput.clear();
        nationalityInput.clear();
        passportNumberInput.clear();
        expDate.clear();
        emailInput.clear();
        phoneInput.clear();

        //clear address input
        streetAndNumberInput.clear();
        cityInput.clear();
        zipcodeInput.clear();
    }

```

```

        countryInput.clear();
    }

    // the upper part of the scene
    // the information about the leg
    // contains also the change flight button that leads back to searching a
    flight
    public static GridPane makeLegInfo(Leg leg){
        GridPane gridPaneLegInfo = new GridPane();
        gridPaneLegInfo.setPadding(new Insets(20, 10, 10, 10));
        gridPaneLegInfo.setVgap(8);
        gridPaneLegInfo.setHgap(15);
        Label labelDepartAirport = new Label("From: " +
leg.getDepartureAirport());
        Label labelArrivalAirport = new Label("To: " + leg.getArrivalAirport());
        Label labelDepartDate = new Label("Depart. Date: " +
leg.getDepartureDate());
        Label labelDepartTime = new Label("Depart. Time: " +
leg.getDepartureTime());
        Label labelArrivalDate = new Label("Arrival Date: " +
leg.getArrivalDate());
        Label labelArrivalTime = new Label("Arrival Time: " +
leg.getArrivalTime());
        Button buttonChange = new Button("Change Flight");
        buttonChange.setOnAction(e -> {

CustomerServiceStaff.window.setScene(CustomerServiceStaff.customerScene);
        });
        gridPaneLegInfo.setConstraints(labelDepartAirport, 0, 0);
        gridPaneLegInfo.setConstraints(labelArrivalAirport, 0, 1);
        gridPaneLegInfo.setConstraints(labelDepartDate, 1, 0);
        gridPaneLegInfo.setConstraints(labelDepartTime, 1, 1);
        gridPaneLegInfo.setConstraints(labelArrivalDate, 2, 0);
        gridPaneLegInfo.setConstraints(labelArrivalTime, 2, 1);
        gridPaneLegInfo.setConstraints(buttonChange, 2, 2);
        gridPaneLegInfo.getChildren().addAll(labelDepartAirport,
labelArrivalAirport, labelDepartDate, labelDepartTime,
labelArrivalDate, labelArrivalTime,
buttonChange);
        gridPaneLegInfo.setAlignment(Pos.CENTER);
        return gridPaneLegInfo;
    }

    // information about the ticket type
    // ComboBox for selecting a ticket class: first class, business or economy
    // the seat number and the ticket price
    public static GridPane makeFlightTypeInfo(Leg leg, List<Ticket>
newBookedTickets){
        GridPane gridFlightTypeInfo = new GridPane();
        gridFlightTypeInfo.setPadding(new Insets(10, 10, 10, 10));
        gridFlightTypeInfo.setVgap(8);
        gridFlightTypeInfo.setHgap(20);
        Label labelClass = new Label("Choose class: ");
        Label price = new Label("Price: ");
        ticketPrice = new Label(" ");
        Label seat = new Label("Seat No: ");
        seatNo = new Label(" ");
        comboBoxClass = new ComboBox();
        comboBoxClass.setPromptText("Select...");

```



```

        comboBoxClass.getItems().addAll("First Class", "Business Class", "Economy
Class");

        // when a ticket class is chosen, the seat number is generate accordingly
        and also the ticket price
        comboBoxClass.setOnAction(e ->
        {    try {
            //show price and seat
            int classType =
convertClassToInt(comboBoxClass.getValue().toString());
            int seat_Number = SeatInfo.method(leg, classType) +
newBookedTickets.size();
            ticketPrice.setText(LegInfo.getPrice(leg,
comboBoxClass.getValue().toString()) + "kr");
            seatNo.setText("" + seat_Number);
        } catch (Exception e2) {
            e2.printStackTrace();
        }
    });
    seatNo.textProperty().addListener((observable, oldValue, newValue) -> {
        if(seatNo.getText().contains("-1"))
            showSeatAlert();                // if there are no availble
seats, it will show alert
    });
    gridFlightTypeInfo.setConstraints(labelClass, 0, 0);
    gridFlightTypeInfo.setConstraints(comboBoxClass, 1, 0);
    gridFlightTypeInfo.setConstraints(price, 2, 0 );
    gridFlightTypeInfo.setConstraints(ticketPrice, 3, 0);
    gridFlightTypeInfo.setConstraints(seat, 4, 0);
    gridFlightTypeInfo.setConstraints(seatNo, 5, 0);
    gridFlightTypeInfo.getChildren().addAll(labelClass, comboBoxClass, price,
ticketPrice, seat, seatNo);
    gridFlightTypeInfo.setAlignment(Pos.CENTER);
    return gridFlightTypeInfo;
}

// converts the String from the ComboBox to the related int value
public static int convertClassToInt(String classType)
{
    int customerClassType = 0;
    if(classType.equalsIgnoreCase("First Class"))
    {
        customerClassType = 1;
    }
    else if (classType.equalsIgnoreCase("Business Class"))
    {
        customerClassType = 2;
    }
    else if (classType.equalsIgnoreCase("Economy Class"))
    {
        customerClassType = 3;
    }
    return customerClassType;
}

// after the tickets are booked/bought, it shows information message and it
goes back to the flight search scene
public static void makeConfirmAlert() throws Exception{
    Alert confirmAlert = new Alert(Alert.AlertType.INFORMATION);
    confirmAlert.setTitle("Information");
    confirmAlert.setHeaderText(null);

```

```

        confirmAlert.setContentText("The order has been successfully placed.");
        confirmAlert.showAndWait();
        CustomerServiceStaff.window.setScene(CustomerServiceStaff.customerScene);
    }

    // shows error alert if all seats are booked
    private static void showSeatAlert()
    {
        String alertText = "There are no available seats for this class. Please
choose another class or change flight.";
        ShowAlert.makeErrorAlert(alertText);
        seatNo.setText("");
        ticketPrice.setText("");
    }

    // shows error alert if the date format is wrong
    // clears the TextField with wrong format input
    public static void showDateAlert() {
        String alertText = "Wrong date format.";
        ShowAlert.makeErrorAlert(alertText);
        if (!CheckInputFormat.isValidDate(birthdayInput.getText()))
            birthdayInput.clear();
        if (!CheckInputFormat.isValidDate(expDate.getText()))
            expDate.clear();
    }

    // adds a listener to the TextFields that should have numeric input
    public static void addListenerToTextFieldProperty(TextField textField)
    {
        textField.textProperty().addListener((observable, oldValue, newValue) ->
{
            if (!CheckInputFormat.isNumeric(textField.getText())) {
                String alertText = "Please enter numeric value.";
                ShowAlert.makeErrorAlert(alertText);
                textField.clear();
            }
        });
    }
}

```

```

// CustomerServiceStaff Class - created by Lei, Dana, Felix
// handles searching an order, searching a flight and it is connected to booking a
ticket

package SEJ.PresentationLayer;

import SEJ.ApplicationLayer.*;
import SEJ.ApplicationLayer.DataTypes.*;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.input.KeyCode;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
import java.util.List;
import java.util.Optional;

public class CustomerServiceStaff {
    static Stage window;
    static Button buttonSearchFlight, buttonSearchOrder, book, buttonLogOut;
    static Button buttonSearch;
    static TextField textFieldDepartAirport, textFieldArrivalAirport,
textFieldOrderID, textFieldFirstName, textFieldLastName;
    static Label labelDate, labelDeparturePlace, labelArrivalPlace, labelOrderID,
labelFirstName, labelLastName;
    static GridPane gridPane;
    static HBox layout, ticketButtons;
    static TableView flightSearchResult, orderSearchResult;
    static DatePicker datePickerDepart;
    static VBox rightSide;
    static Scene customerScene;
    static int customersSize;
    static int ticketsSize;

    public static Scene returnCustomerService (Stage primaryStage) throws
Exception {
        List<Customer> customers = CustomerInfo.selectAllCustomers();
        List<Ticket> tickets = TicketInfo.selectAllTickets();

        customersSize = customers.size();
        ticketsSize = tickets.size();

        window = primaryStage;
        window.setTitle("Customer Service");

        gridPane = new GridPane();
        gridPane.setPadding(new Insets(10, 10, 10, 10));
        gridPane.setHgap(65);
        gridPane.setVgap(15);

        // The buttons on the left side are created, set on action and added to a
VBox
        buttonSearchFlight = new Button("Book Flight");
        buttonSearchFlight.setMinWidth(100);

```

```

        buttonSearchFlight.setMaxWidth(100);
        buttonSearchFlight.setOnAction(e -> {
            try {
                layout.getChildren().remove(rightSide);
                layout.getChildren().add(makeRightPart());
            } catch (Exception e1) {
                e1.printStackTrace();
            }
        });

        buttonSearchOrder = new Button("Search Order");
        buttonSearchOrder.setMinWidth(100);
        buttonSearchOrder.setMaxWidth(100);
        buttonSearchOrder.setOnAction((e -> {
            try {
                layout.getChildren().remove(searchOrderClicked());
                layout.getChildren().addAll(searchOrderClicked());
            } catch (Exception e1) {
                e1.printStackTrace();
            }
        }));

        buttonLogOut = new Button("Log out");
        buttonLogOut.setMinWidth(100);
        buttonLogOut.setMaxWidth(100);
        buttonLogOut.setOnAction(e -> {
            try{
                VBox login = LogInScene.makeLogInScene_logo(window);
                Scene scene = new Scene(login, 400, 500);
                scene.getStylesheets().add("CSS.css");
                window.setScene(scene);
            }
            catch (Exception e2){
                e2.printStackTrace();
            }
        });

        VBox vBoxButtons = new VBox(15);
        vBoxButtons.setPadding(new Insets(10,10,10,10));
        vBoxButtons.setId("vBox");
        vBoxButtons.getChildren().addAll(buttonSearchFlight, buttonSearchOrder,
        buttonLogOut);

        // the layout is a HBox, containing a VBox (the buttons on the left) and
        the right part which is also a VBox

        layout = new HBox(50);
        layout.getChildren().addAll(vBoxButtons, makeRightPart());

        customerScene = new Scene(layout, 943, 470);
        customerScene.getStylesheets().add("CustomerCSS");
        return customerScene;
    }

    // VBox with GridPane (containing Labels, TextFields and Search Button for
    order) and TableView
    public static VBox searchOrderClicked() throws Exception{
        buttonSearchFlight.setDisable(false);
        buttonSearchOrder.setDisable(true);
    }

```

```

        layout.getChildren().remove(gridPane);
        layout.getChildren().remove(rightSide);
        rightSide.getChildren().clear();
        gridPane.getChildren().clear();
        labelOrderID = new Label("Order ID: ");
        gridPane.setConstraints(labelOrderID, 0, 0);

        textFieldOrderID = new TextField();
        textFieldOrderID.setPromptText("Order ID");
        textFieldOrderID.setMaxWidth(130);
        gridPane.setConstraints(textFieldOrderID, 0, 1);
        textFieldOrderID.setOnAction(textFieldOrderID);

        labelFirstName = new Label("First Name: ");
        gridPane.setConstraints(labelFirstName, 1, 0);

        textFieldFirstName = new TextField();
        textFieldFirstName.setPromptText("First Name");
        textFieldFirstName.setMaxWidth(150);
        gridPane.setConstraints(textFieldFirstName, 1, 1);

        labelLastName = new Label("Last Name: ");
        gridPane.setConstraints(labelLastName, 2, 0);

        textFieldLastName = new TextField();
        textFieldLastName.setPromptText("Last Name");
        textFieldLastName.setMaxWidth(150);
        gridPane.setConstraints(textFieldLastName, 2, 1);
        textFieldLastName.setOnAction(textFieldLastName);

        buttonSearch.setOnAction(e -> { try {
            searchOrder();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        });

        gridPane.setConstraints(buttonSearch, 3, 1);
        gridPane.setAlignment(Pos.CENTER);
        gridPane.getChildren().addAll(labelOrderID, textFieldOrderID,
labelFirstName,           textFieldFirstName, labelLastName, textFieldLastName,
buttonSearch);

// starts with TableView with no content, so that it won't be a blank space there
        orderSearchResult = OrderSearchResultTableView.makeTableView(0);

        rightSide.getChildren().addAll(gridPane, orderSearchResult);

        return rightSide;
    }

    public static void searchOrder() throws Exception {
        rightSide.getChildren().remove(orderSearchResult);
        rightSide.getChildren().remove(ticketButtons);

        // if the TextField with the order ID is not empty, it searches by order
        ID
        if(!textFieldOrderID.getText().isEmpty())

```

```

        orderSearchResult =
OrderSearchResultTableView.makeTableView(Integer.parseInt(textFieldOrderID.getText()
()));

// else, if first name and last name are typed in, it searches by name
    else
        if(!textFieldFirstName.getText().isEmpty()
&& !textFieldLastName.getText().isEmpty())
            orderSearchResult =
OrderSearchResultTableView.makeTableView(OrderInfo.findOrderId //returns order id
(textFieldFirstName.getText(),
textFieldLastName.getText()));
        // else, it displays error alert with message
    else{
        String textAlert = "Enter only order ID or first name and last name";
        ShowAlert.makeErrorAlert(textAlert);
    }

    // when a row is selected, it finds the ticket corresponding
    orderSearchResult.setOnMouseClicked(ev -> {
        try{
            if(orderSearchResult.getSelectionModel().getSelectedItem() !=
null) {
                rightSide.getChildren().removeAll(ticketButtons);
                Ticket ticket =
OrderSearchResultInfo.findSelectedTicket((OrderSearchResult)
orderSearchResult.getSelectionModel().getSelectedItem());
                ticketButtons = makeTicketButtons(ticket); // it displays
buttons according to the ticket found
                rightSide.getChildren().addAll(ticketButtons);
            }
        }
        catch (Exception ex){
            ex.printStackTrace();
        }
    });

    rightSide.getChildren().add(orderSearchResult);
    textFieldOrderID.clear();
    textFieldFirstName.clear();
    textFieldLastName.clear();
}

// VBox with GridPane (TextFields, Labels, DatePicker and search Button for
leg) and TableView
public static VBox makeRightPart() throws Exception{
    buttonSearchFlight.setDisable(true);
    buttonSearchOrder.setDisable(false);
    gridPane.getChildren().clear();

    layout.getChildren().remove(gridPane);
    layout.getChildren().remove(rightSide);

    rightSide = new VBox(20);
    rightSide.setAlignment(Pos.TOP_CENTER);

```

```

labelDeparturePlace = new Label("From: ");
gridPane.setConstraints(labelDeparturePlace, 0, 0);

textFieldDepartAirport = new TextField();
textFieldDepartAirport.setPromptText("City");
textFieldDepartAirport.setMaxLength(150);
gridPane.setConstraints(textFieldDepartAirport, 0, 1);

labelArrivalPlace = new Label("To: ");
gridPane.setConstraints(labelArrivalPlace, 1, 0);

textFieldArrivalAirport= new TextField();
textFieldArrivalAirport.setPromptText("City");
textFieldArrivalAirport.setMaxLength(150);
gridPane.setConstraints(textFieldArrivalAirport, 1, 1);

labelDate = new Label("Departure date: ");
gridPane.setConstraints(labelDate, 2, 0);

datePickerDepart = new DatePicker();
datePickerDepart.setPromptText("Departure");
datePickerDepart.setMaxLength(130);
AdminScene.changeDatePickerFormat(datePickerDepart);
gridPane.setConstraints(datePickerDepart, 2, 1);

buttonSearch = new Button("Search");
buttonSearch.setMinWidth(70);
gridPane.setConstraints(buttonSearch, 3, 1);
buttonSearch.setOnAction(e -> {
    if(textFieldDepartAirport.getText().isEmpty() ||
    textFieldArrivalAirport.getText().isEmpty() ||
        datePickerDepart.getValue() == null){
        String alertText = "Please fill out all fields.";
        showAlert.makeErrorAlert(alertText); // displays
error if fields are not filled out
    }else
        try {
            searchFlight(); // searches flight
        }
        catch (Exception e2){
            System.out.println(e2.getMessage());
        }
    }
});

// starts with TableView with no content, so that it won't be a blank
space there
flightSearchResult = FlightSearchResultTableView.makeTableView("", "",
"");

gridPane.getChildren().addAll(labelDeparturePlace,
textFieldDepartAirport, labelArrivalPlace, textFieldArrivalAirport, labelDate,
datePickerDepart, buttonSearch);
gridPane.setAlignment(Pos.CENTER);
gridPane.setId("gridPane");
rightSide.getChildren().addAll(gridPane, flightSearchResult);

return rightSide;
}

public static void searchFlight() throws Exception {

```

```

        rightSide.getChildren().removeAll(flightSearchResult, book);
        // the TableView with all legs that satisfy the search condition
        flightSearchResult =
FlightSearchResultTableView.makeTableView(textFieldDepartAirport.getText(),
        textFieldArrivalAirport.getText(),
datePickerDepart.getEditor().getText());

        book = new Button("Book Ticket");

        //when a row is clicked, the book ticket button appears
        flightSearchResult.setOnMouseClicked(event -> {
            rightSide.getChildren().remove(book);
            // the book button makes the transition to the booking scene
            book.setOnAction(eve -> {
                try {
                    Leg leg = FlightSearchResultInfo.findSelectedLeg
((FlightSearchResult) flightSearchResult.getSelectionModel().getSelectedItem());
                    BookTicket.makeLayout(leg, window);
                }
                catch (Exception exception){
                    exception.printStackTrace();
                }
            });
            rightSide.getChildren().add(book);
        });
        rightSide.getChildren().addAll(flightSearchResult);
    }

    // returns the buttons according to the ticket state
    public static HBox makeTicketButtons(Ticket ticket) {
        HBox buttons = new HBox(20);

        Button buttonConfirm = new Button("Confirm");
        Button buttonRefund = new Button("Refund");
        Button buttonCancel = new Button("Cancel");

        buttonRefund.setOnAction(e -> { try {
            makeRefundAlert(ticket, TicketInfo.refundTicketClicked(ticket));
            buttons.getChildren().remove(buttonRefund);
        } catch (Exception exception) {
            exception.printStackTrace();
        }
        });

        buttonCancel.setOnAction(e -> { try {
            makeCancelAlert(ticket);
            buttons.getChildren().remove(buttonCancel);
            buttons.getChildren().remove(buttonConfirm);
        }
        catch (Exception exception){
            exception.printStackTrace();
        }
        });
        buttonConfirm.setOnAction(e -> {
            try {
                makeConfirmAlert(ticket);
                buttons.getChildren().remove(buttonConfirm);
                buttons.getChildren().remove(buttonCancel);
            }
            catch (Exception e1){

```



```

        e1.printStackTrace();
    }
});

// for a booked ticket, it returns confirm and cancel buttons
if(ticket.getTicketState().equalsIgnoreCase("booked"))
    buttons.getChildren().addAll(buttonConfirm, buttonCancel);
// for a confirmed ticket, it returns a refund button
if(ticket.getTicketState().equalsIgnoreCase("confirmed"))
    buttons.getChildren().add(buttonRefund);

buttons.setAlignment(Pos.CENTER);
return buttons;
}

// handles the refund option
public static void makeRefundAlert(Ticket ticket, String legDate) throws
Exception{
    // if it is an economy class
    if(ticket.getTicketClass() == 3) {
        // and there is less than 2 weeks before the flight
        if (TicketInfo.calculateDateDifference(legDate) <= 14) {
            String alertText = "There is less than 2 weeks to departure date,
" +
                "so there will be no refund, but the ticket will be
canceled. " +
                "Are you sure you want to cancel the ticket?";
            Alert confirmAlert = ShowAlert.makeConfirmAlert(alertText);

            Optional<ButtonType> result = confirmAlert.showAndWait();
            // if the user chooses to cancel although there will be no refund
            if (result.get() == ButtonType.OK) {
                TicketInfo.cancelTicketClicked(ticket);          // ticket is
cancelled
                OrderSearchResultTableView.deleteRow(orderSearchResult); //
TableView updated
            }
            // the user can cancel the refunding - the ticket won't be
deleted, nothing happens
            if (result.get() == ButtonType.CANCEL) {
                confirmAlert.close();
            }
        }
        // if there is more than 2 weeks before the flight
        // the ticket is refunded
        else {
            makeRefundAlert(ticket);
        }
    }
    // if it is a first class or a business class ticket
    // the ticket is refunded
    else
    {
        makeRefundAlert(ticket);
    }
}

// the user has to choose whether to cancel or not
public static void makeCancelAlert(Ticket ticket) throws Exception{
    String alertText = "Are you sure you want to cancel the ticket?";

```

```

Alert cancelAlert = showAlert.makeConfirmAlert(alertText);
ButtonType buttonTypeYes = new ButtonType("Yes");
ButtonType buttonTypeNo = new ButtonType("No");

cancelAlert.getButtonTypes().setAll(buttonTypeYes, buttonTypeNo);

// if the user clicks yes - the ticket is cancelled and the TableView is
Optional<ButtonType> result = cancelAlert.showAndWait();
if (result.get() == buttonTypeYes) {
    TicketInfo.cancelTicketClicked(ticket);
    OrderSearchResultTableView.deleteRow(orderSearchResult);
}
// if the user clicks no - nothing happens
if (result.get() == buttonTypeNo) {
    cancelAlert.close();
}
}
// shows message that the ticket will be refunded
// the ticket is deleted
// updates TableView
public static void makeRefundAlert(Ticket ticket) throws Exception
{
    String alertText = "The ticket has been refunded. "
        + ticket.getTicketPrice() + " kr will be transferred to the
customer's account within 30 days.";
    showAlert.makeInformationAlert(alertText);
    TicketInfo.cancelTicketClicked(ticket);
    OrderSearchResultTableView.deleteRow(orderSearchResult);
}

// shows message that the ticket has been confirmed
// sends information to the Application Layer - the ticket
// updates TableView
public static void makeConfirmAlert(Ticket ticket) throws Exception
{
    String alertText = "The ticket is now confirmed.";
    showAlert.makeInformationAlert(alertText);
    TicketInfo.confirmTicketClicked(ticket);
    OrderSearchResultTableView.updateRow(orderSearchResult);
}

// on the order ID and the last name TextFields the user can press enter and
search order
public static void textFieldSetOnAction(TextField textField)
{
    textField.setOnKeyPressed(e -> {
        if (e.getCode() == KeyCode.ENTER) try {
            searchOrder();
        }
        catch (Exception e1) {
            e1.printStackTrace();
        }
    });
}
}

```

```

// FlightSearchResultTableView Class - created by Lei

package SEJ.PresentationLayer;

import SEJ.ApplicationLayer.DataTypes.FlightSearchResult;
import SEJ.ApplicationLayer.FlightSearchResultInfo;
import javafx.collections.ObservableList;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableView;
import javafx.scene.control.cell.PropertyValueFactory;
import javafx.scene.control.cell.TextFieldTableCell;

public class FlightSearchResultTableView
{
    // creates the necessary columns
    // returns the TableView with legs that match the search condition
    public static TableView makeTableView(String departureCity,
                                           String arrivalCity,
                                           String departureDate) throws Exception
    {
        TableView<FlightSearchResult> flightSearchTableView = new TableView<>();
        TableColumn<FlightSearchResult, String> column_departureCity,
column_arrivalCity, column_departureDate, column_departureTime,
column_arrivalDate, column_arrivalTime;

        //create an Observable list of all flights that fits the search condition
        ObservableList<FlightSearchResult> allFlightSearchResults =
FlightSearchResultInfo.createFlightSearchResultsProperty(departureCity,
arrivalCity, departureDate);
        flightSearchTableView.setItems(allFlightSearchResults);

        //departure city column
        column_departureCity = new TableColumn<>("From");
        column_departureCity.setMinWidth(160);
        column_departureCity.setCellValueFactory(new
PropertyValueFactory<>("departureCity"));

column_departureCity.setCellFactory(TextFieldTableCell.<FlightSearchResult>forTableC
olumn());

        //arrival city column
        column_arrivalCity = new TableColumn<>("To");
        column_arrivalCity.setMinWidth(160);
        column_arrivalCity.setCellValueFactory(new
PropertyValueFactory<>("arrivalCity"));

column_arrivalCity.setCellFactory(TextFieldTableCell.<FlightSearchResult>forTableC
olumn());

        //departure date column
        column_departureDate = new TableColumn<>("Dep. Date");
        column_departureDate.setMinWidth(100);
        column_departureDate.setCellValueFactory(new
PropertyValueFactory<>("departureDate"));

column_departureDate.setCellFactory(TextFieldTableCell.<FlightSearchResult>forTableC
olumn());
    }
}

```

```

        //departure time column
        column_departureTime = new TableColumn<>("Dep. Time");
        column_departureTime.setMinWidth(100);
        column_departureTime.setCellValueFactory(new
PropertyValueFactory<>("departureTime"));

column_departureTime.setCellFactory(TextFieldTableCell.<FlightSearchResult>forTableC
olumn());

        //arrival date column
        column_arrivalDate = new TableColumn<>("Arr. Date");
        column_arrivalDate.setMinWidth(100);
        column_arrivalDate.setCellValueFactory(new
PropertyValueFactory<>("arrivalDate"));

column_arrivalDate.setCellFactory(TextFieldTableCell.<FlightSearchResult>forTableC
olumn());

        //arrival time column
        column_arrivalTime = new TableColumn<>("Arr. Time");
        column_arrivalTime.setMinWidth(100);
        column_arrivalTime.setCellValueFactory(new
PropertyValueFactory<>("arrivalTime"));

column_arrivalTime.setCellFactory(TextFieldTableCell.<FlightSearchResult>forTableC
olumn());

        flightSearchTableView.getColumns().addAll(column_departureCity,
column_arrivalCity,
            column_departureDate, column_departureTime, column_arrivalDate,
column_arrivalTime);
        flightSearchTableView.setMaxHeight(250);

        return flightSearchTableView;
    }
}

```

```

// GenerateTicketToBook Class - created by Lei

// used when booking tickets (it will return a GridPane with ticket information
that appears under 'Orders')

package SEJ.PresentationLayer;

import SEJ.ApplicationLayer.DataTypes.Leg;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.Hbox;

public class GenerateTicketToBook
{
    public static HBox makeOneTicket(Leg leg)
        throws Exception
    {
        // generates tickets according to the size of the array
        String passengerN = BookTicket.firstNameInput.getText() + " " +
BookTicket.lastNameInput.getText();
        String seat_No = "" + BookTicket.seatNo.getText();
        String depAirp = leg.getDepartureAirport();
        String arrAirp = leg.getArrivalAirport();
        String depDateTime = leg.getDepartureDate() + " " +
leg.getDepartureTime();
        String arrDateTime = leg.getArrivalDate() + " " + leg.getArrivalTime();
        String price = BookTicket.ticketPrice.getText();

        HBox orderBox = new HBox(15);
        orderBox.setPrefSize(650, 130);
        orderBox.setPadding(new Insets(0, 10, 10, 10));

        GridPane ticketOrder = makeTicket(passengerN, seat_No, depAirp, arrAirp,
depDateTime, arrDateTime, price);
        ticketOrder.setAlignment(Pos.CENTER_RIGHT);

        orderBox.setId("vBox");

        orderBox.getChildren().add(ticketOrder);
        orderBox.setAlignment(Pos.CENTER);

        return orderBox;
    }

    public static GridPane makeTicket(String passengerN, String seat_No, String
depAirp, String arrAirp, String depDateTime, String arrDateTime, String
ticketPrice) throws Exception
    {
        //the main layout for a ticket
        GridPane ticketPane = new GridPane();
        ticketPane.setPrefSize(600, 110);
        ticketPane.setHgap(20);
        ticketPane.setVgap(10);

        //passenger info
        Label passenger = new Label("Passenger: ");

```

```

Label passenger_info = new Label(passengerN);

// seat and seat_info labels
Label seat = new Label("Seat: ");
Label seat_info = new Label(seat_No);

// from and from_info labels
Label depAirport = new Label("From: ");
Label depAirport_info = new Label(depAirp);

// to label and to_info label
Label arrAirport = new Label("To: ");
Label arrAirport_info = new Label(arrAirp);

// departure and departure_info labels
Label depTime = new Label("Dep.: ");
Label depTime_info = new Label(depDateTime);

// arrival and arrival_info labels
Label arrTime = new Label("Arr.: ");
Label arrTime_info = new Label(arrDateTime);

// price and price_info labels
Label price = new Label("Price: ");
Label price_info = new Label(ticketPrice);

ticketPane.setConstraints(passenger, 0, 0);
ticketPane.setConstraints(passenger_info, 1, 0);
ticketPane.setConstraints(depAirport, 0, 1);
ticketPane.setConstraints(depAirport_info, 1, 1);
ticketPane.setConstraints(arrAirport, 2, 1);
ticketPane.setConstraints(arrAirport_info, 3, 1);
ticketPane.setConstraints(seat, 4, 1);
ticketPane.setConstraints(seat_info, 5, 1);
ticketPane.setConstraints(price, 4, 2);
ticketPane.setConstraints(price_info, 5, 2);
ticketPane.setConstraints(depTime, 0, 2);
ticketPane.setConstraints(depTime_info, 1, 2);
ticketPane.setConstraints(arrTime, 2, 2);
ticketPane.setConstraints(arrTime_info, 3, 2);

ticketPane.getChildren().addAll(passenger, passenger_info, depAirport,
depAirport_info, arrAirport, arrAirport_info, seat, seat_info, price, price_info,
depTime, depTime_info, arrTime, arrTime_info);

return ticketPane;
}
}

```

// LoginScene Class - created by Marius

```
package SEJ.PresentationLayer;

import SEJ.ApplicationLayer.DataTypes.Employee;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.PasswordField;
import javafx.scene.control.TextField;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.input.KeyCode;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class LogInScene {
    static private GridPane logInPane;
    static private TextField employeeUsername;
    static private PasswordField employeePWInput;
    static private Label logInMessage, welcomeToSystem;
    static private Button logInButton;
    static private ImageView imageView_logo;
    static int employeeID;

    // returns a VBox containing the login info
    public static VBox makeLogInScene_logo(Stage primaryStage)
    {
        logInPane = makeLogInScene(primaryStage);
        imageView_logo = makeLogo();
        welcomeToSystem = new Label("Welcome");
        welcomeToSystem.setId("welcomeLabel");
        Label label= new Label();
        VBox logInScene_pic = new VBox(10);
        logInScene_pic.getChildren().addAll(label, welcomeToSystem,
        imageView_logo, logInPane);
        logInScene_pic.setAlignment(Pos.CENTER);

        return logInScene_pic;
    }

    // creates the login using a GridPane
    // it contains Labels and TextFields
    public static GridPane makeLogInScene(Stage primaryStage)
    {
        logInPane = new GridPane();
        logInPane.setMinWidth(250);
        logInPane.setMinHeight(200);
        logInPane.setHgap(10);
        logInPane.setVgap(10);
        Label employeeName_label = new Label("Username");
        employeeUsername = new TextField();
        Label password = new Label("Password");
        employeePWInput = new PasswordField();
        employeePWInput.setOnKeyPressed(e -> {
            if(e.getCode() == KeyCode.ENTER) {
                setLogInButtonAction(primaryStage);
            }
        });
    }
}
```

```

    }
});
loginMessage = new Label();
loginButton = new Button("Log in");
loginButton.setMinWidth(80);
loginButton.setOnAction(e -> setLoginButtonAction(primaryStage));
loginPane.add(employeeName_label, 0, 0);
loginPane.add(employeeUsername, 1, 0);
loginPane.add(password, 0, 1);
loginPane.add(employeePWInput, 1, 1);
loginPane.add(loginButton, 1, 2);
loginPane.add(loginMessage, 1, 3, 2, 3);
loginPane.setAlignment(Pos.CENTER);

return loginPane;
}

// when the login button is pressed, the scene is changed to either admin
scene or customer service scene
// depending on the username and password the user typed in
public static void setLoginButtonAction(Stage primaryStage)
{
    {
        String checkUsername = employeeUsername.getText();
        String checkPassword = employeePWInput.getText();
        try
        {
            int employeesSize = Main.allEmployees.size();
            // loops through the employees array, finds the one that matches
            username and password
            for(int i = 0; i < employeesSize; i++)
            {
                Employee employee = Main.allEmployees.get(i);
                String employeeUsername = employee.getUserName();
                String employeePW = employee.getPassword();
                String employeeTitle = employee.getTitle();
                employeeID = employee.getEmployeeID();
                // if the title of the employee is 'admin', the scene is
                changed to the admin scene
                if (checkPassword.equalsIgnoreCase(employeePW) &&
                checkUsername.equalsIgnoreCase(employeeUsername) &&
                employeeTitle.equalsIgnoreCase("admin"))
                {
                    Scene adminScene =
AdminScene.returnAdminScene(primaryStage);
                    primaryStage.setScene(adminScene);
                    primaryStage.show();
                    break;
                    // if the title of the employee is 'customer service',
                    the scene is changed to the customer service scene
                }else if (checkPassword.equalsIgnoreCase(employeePW) &&
                checkUsername.equalsIgnoreCase(employeeUsername) &&
                employeeTitle.equalsIgnoreCase("customer_service"))
                {
                    Scene adminScene =
CustomerServiceStaff.returnCustomerService(primaryStage);
                    primaryStage.setScene(adminScene);
                    primaryStage.show();
                    break;
                }
            }
        }
    }
}

```



```

        // if the username and password do not match with any of the
employee's, displays message
        else
        {
            loginMessage.setText("Sorry, user does not exist.");
        }
    }
} catch (Exception e1)
{
    e1.printStackTrace();
}
}

// the logo
public static ImageView makeLogo()
{
    Image logoImage = new
Image(LogInScene.class.getResourceAsStream("SEJ_LOGO.jpg"));
    logoImage.getClass().getResourceAsStream("SEJ_LOGO.jpg");
    ImageView imageView_logo = new ImageView(logoImage);
    imageView_logo.setFitWidth(280);
    imageView_logo.setPreserveRatio(true);
    imageView_logo.setSmooth(true);

    return imageView_logo;
}
}

```

```

// Main Class - created by Marius

package SEJ.PresentationLayer;

import SEJ.ApplicationLayer.DataTypes.Employee;
import SEJ.ApplicationLayer.DataTypes.Leg;
import SEJ.ApplicationLayer.DataTypes.Plane;
import SEJ.ApplicationLayer.EmployeeInfo;
import SEJ.ApplicationLayer.LegInfo;
import SEJ.ApplicationLayer.PlaneInfo;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
import java.util.List;

// starts with loading information into arrays
// it leads to the login scene
public class Main extends Application {
    static List<Employee> allEmployees;
    static List<Plane> allPlanes;
    static List<Leg> allLegs;

    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) throws Exception {
        allEmployees = EmployeeInfo.selectAllEmployees();
        allPlanes = PlaneInfo.selectAllPlanes();
        allLegs = LegInfo.selectAllLegs();

        primaryStage.setTitle("SEJ");

        // VBox with the login scene
        VBox logInScene_pic = LogInScene.makeLogInScene_logo(primaryStage);

        Scene scene = new Scene(logInScene_pic, 400, 500);
        scene.getStylesheets().add("CSS.css");

        primaryStage.setScene(scene);
        primaryStage.show();
    }
}

```

```

// OrderSearchResultTableView - created by Felix, Dana

package SEJ.PresentationLayer;

import SEJ.ApplicationLayer.DataTypes.OrderSearchResult;
import SEJ.ApplicationLayer.OrderSearchResultInfo;
import javafx.collections.ObservableList;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableView;
import javafx.scene.control.cell.PropertyValueFactory;
import javafx.scene.control.cell.TextFieldTableCell;
import javafx.util.converter.IntegerStringConverter;

public class OrderSearchResultTableView {
    // creates the necessary columns
    // returns the TableView with orders searched
    public static TableView makeTableView(int orderId) throws Exception
    {
        TableView<OrderSearchResult> orderSearchTableView = new TableView<>();
        TableColumn<OrderSearchResult, String> column_firstName,
column_lastName, column_ticketState;
        TableColumn<OrderSearchResult, Integer> column_orderId, column_ticketId,
column_ticketClass, column_ticketPrice, column_ticketSeatNr;

        //create an Observable list of all orders that fit the search condition
        ObservableList<OrderSearchResult> allOrderSearchResults =
OrderSearchResultInfo.createOrderSearchResultsProperty(orderId);
        orderSearchTableView.setItems(allOrderSearchResults);

        //order ID column
        column_orderId = new TableColumn<>("Order Id");
        column_orderId.setMinWidth(80);
        column_orderId.setMaxWidth(80);
        column_orderId.setCellValueFactory(new
PropertyValueFactory<>("orderId"));
        column_orderId.setCellFactory(TextFieldTableCell.forTableColumn(new
IntegerStringConverter()));

        //ticket ID column
        column_ticketId = new TableColumn<>("Ticket Id");
        column_ticketId.setMinWidth(80);
        column_ticketId.setMaxWidth(80);
        column_ticketId.setCellValueFactory(new
PropertyValueFactory<>("ticketID"));
        column_ticketId.setCellFactory(TextFieldTableCell.forTableColumn(new
IntegerStringConverter()));

        //first name column
        column_firstName = new TableColumn<>("First Name");
        column_firstName.setMinWidth(120);
        column_firstName.setMaxWidth(120);
        column_firstName.setCellValueFactory(new
PropertyValueFactory<>("customerFirstName"));
        column_firstName.setCellFactory(TextFieldTableCell.forTableColumn());

        //last name column
        column_lastName = new TableColumn<>("Last Name");
        column_lastName.setMinWidth(120);
        column_lastName.setMaxWidth(120);
    }
}

```

```

        column_lastName.setCellValueFactory(new
PropertyValuFactory<>("customerLastName"));

column_lastName.setCellFactory(TextFiledTableCell.<OrderSearchResult>forTableColumn
n());

        //ticket class column
column_ticketClass = new TableColumn<>("Class");
column_ticketClass.setMinWidth(80);
column_ticketClass.setMaxWidth(80);
column_ticketClass.setCellValueFactory(new
PropertyValuFactory<>("ticketClass"));
column_ticketClass.setCellFactory(TextFiledTableCell.forTableColumn(new
IntegerStringConverter()));

        //ticket price column
column_ticketPrice = new TableColumn<>("Price");
column_ticketPrice.setMinWidth(80);
column_ticketPrice.setMaxWidth(80);
column_ticketPrice.setCellValueFactory(new
PropertyValuFactory<>("ticketPrice"));
column_ticketPrice.setCellFactory(TextFiledTableCell.forTableColumn(new
IntegerStringConverter()));

        //ticket seat number column
column_ticketSeatNr = new TableColumn<>("Seat Nr.");
column_ticketSeatNr.setMinWidth(80);
column_ticketSeatNr.setMaxWidth(80);
column_ticketSeatNr.setCellValueFactory(new
PropertyValuFactory<>("ticketSeatNumber"));
column_ticketSeatNr.setCellFactory(TextFiledTableCell.forTableColumn(new
IntegerStringConverter()));

        //ticket state
column_ticketState = new TableColumn<>("Ticket State");
column_ticketState.setMinWidth(80);
column_ticketState.setMaxWidth(80);
column_ticketState.setCellValueFactory(new
PropertyValuFactory<>("ticketState"));

column_ticketState.setCellFactory(TextFiledTableCell.<OrderSearchResult>forTableCo
lumn());
        orderSearchTableView.getColumns().addAll(column_orderId, column_ticketId,
column_firstName,
        column_lastName, column_ticketClass, column_ticketPrice,
column_ticketSeatNr, column_ticketState);
        orderSearchTableView.setMaxHeight(250);

        return orderSearchTableView;
    }

    // when a ticket is cancelled or refunded, it is removed from the Observable
List and consequently from the TableView
    public static void deleteRow(TableView tableView){
        try {
            ObservableList<OrderSearchResult> allOrderResults =
tableView.getItems();
            ObservableList<OrderSearchResult> readOnlyItems =
tableView.getSelectionModel().getSelectedItem();
            //Removes all selected elements in the table

```

```

        readOnlyItems.stream().forEach((item) -> {
            allOrderResults.remove(item);
        });

        //Clear the selection
        tableView.getSelectionModel().clearSelection();
    }catch (Exception exception){
        exception.printStackTrace();
    }
}

// when a ticket is confirmed, the tableview is updated
public static void updateRow(Tableview tableView){
    try {
        ObservableList<OrderSearchResult> allOrderResults =
tableView.getItems();
        ObservableList<OrderSearchResult> readOnlyItems =
tableView.getSelectionModel().getSelectedItem();

        // finds the ticket chosen and changes its state to "confirmed"
        for(int i = 0; i < allOrderResults.size(); i++){
            if (allOrderResults.get(i) == readOnlyItems.get(0))

allOrderResults.get(0).ticketStateProperty().setValue("confirmed");
        }

        //Clear the selection
        tableView.getSelectionModel().clearSelection();
    }catch (Exception exception){
        exception.printStackTrace();
    }
}
}

```

```

// OutputCSV Class - created by Felix

// OSCA part
// creates the CSV files (with the comment) and the output files (without the
comment)

package SEJ.PresentationLayer;

import SEJ.ApplicationLayer.EmployeeInfo;
import SEJ.ApplicationLayer.PlaneInfo;
import javafx.scene.control.Button;

public class OutputCSV
{
    public static Button makeOutputDatabase()
    {
        Button outputDatabase = new Button("Create CSV file");
        outputDatabase.setOnAction(e -> {
            try
            {
                EmployeeInfo.readEmployeesInTable(AdminScene.allEmployees);
                EmployeeInfo.readIgnoreComment();
                PlaneInfo.readPlanesInTable(AdminScene.allPlanes);
                PlaneInfo.readIgnoreComment();

                String alertText = "The files have been created.";
                ShowAlert.makeInformationAlert(alertText);
            } catch (Exception e1)
            {
                e1.printStackTrace();
            }
        });
        return outputDatabase;
    }
}

```



```

// showAlert class - created by Lei

// Manages alerts - has three types of alerts with String parameters which will
// become the Content Text of the alert
package SEJ.PresentationLayer;

import javafx.scene.control.Alert;

public class showAlert
{
    public static void makeErrorAlert(String text)
    {
        Alert alert = new Alert(Alert.AlertType.ERROR);
        alert.setTitle("ERROR");
        alert.setHeaderText(null);
        alert.setContentText(text);
        alert.showAndWait();
    }

    public static void makeInformationAlert(String text)
    {
        Alert refundAlert = new Alert(Alert.AlertType.INFORMATION);
        refundAlert.setTitle("Information");
        refundAlert.setHeaderText(null);
        refundAlert.setContentText(text);
        refundAlert.showAndWait();
    }

    public static Alert makeConfirmAlert(String text)
    {
        Alert confirmAlert = new Alert(Alert.AlertType.CONFIRMATION);
        confirmAlert.setTitle("Confirmation");
        confirmAlert.setHeaderText(null);
        confirmAlert.setContentText(text);
        return confirmAlert;
    }
}

```


Appendix 10: The CSS files

CSS.css

```
.root {  
  
    -fx-background-color: #C6EDC7;  
}  
  
.button {  
    -fx-background-color: #2D614D;  
    -fx-text-fill: #FFFFFF;  
    -fx-font-size: 13px;  
    -fx-font-weight: bold;  
    -fx-background-radius: 10;  
}  
  
.text-field {  
    -fx-background-color: #459475;  
    -fx-text-fill: #FFFFFF;  
    -fx-font-size: 14px;  
    -fx-background-radius: 10;  
}  
  
.label {  
    -fx-font: 16px "Garamond";  
    -fx-text-fill: #313732;  
    -fx-font-weight: bold;  
}  
  
#vBox {  
    -fx-border-color: #007099;  
    -fx-border-width: 2px;  
}  
  
#gridPane{  
    -fx-border-color: #007099;  
    -fx-border-width: 2px;  
}  
  
#welcomeLabel{  
    -fx-font: 24px "Garamond";  
    -fx-text-fill: #313732;  
    -fx-font-weight: bold;  
    -fx-effect: dropshadow(three-pass-box, rgba(0,0,0,0.5), 10, 0.0, 0, 5);  
}
```

CustomerCSS.css

```
.root {

    -fx-background-color: #C6EDC7;
}

.button {
    -fx-background-color: #2D614D;
    -fx-text-fill: #FFFFFF;
    -fx-font-size: 13px;
    -fx-font-weight: bold;
    -fx-background-radius: 10;
}

.text-field {
    -fx-background-color: #FFFFFF;
    -fx-text-fill: black;
    -fx-font-size: 14px;
}

.label {
    -fx-font: 14px "Garamond";
    -fx-text-fill: #313732;
    -fx-font-weight: bold;
}

.table-view:focused {
    -fx-background-color: transparent, -fx-box-border, -fx-control-inner-
background;
}

.table-row-cell{
    -fx-background-color: -fx-table-cell-border-color, white;
    -fx-background-insets: 0, 0 0 1 0;
    -fx-padding: 0.0em;
}

.table-row-cell:selected {
    -fx-background-color: #558579;
    -fx-background-insets: 0;
    -fx-background-radius: 1;
}

#customerInputTextfield {
    -fx-background-color: #FFFFFF;
    -fx-text-fill: black;
    -fx-font-size: 13px;
}

#vBox {
    -fx-border-color: #43695F;
    -fx-border-width: 2px;
}

#labelOrders {
    -fx-font: 20px "Garamond";
    -fx-text-fill: #313732;
    -fx-font-weight: bold;
}
```

