<Plane Tickets>

Analysis and Design Document

Student: Prata Mirela

**Group: 30233**

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Description** | **Author** |
| 25/04/2019 | 1.0 | Iteration 1.1 | Prata Mirela |
| 9/05/2019 | 2.0 | Iteration 1.2 | Prata Mirela |
| 30/05/2019 | 3.0 | Iteration 2 | Prata Mirela |
|  |  |  |  |

Table of Contents

I. Project Specification 4

II. Elaboration – Iteration 1.1 4

1. Domain Model 4

2. Architectural Design 4

2.1 Conceptual Architecture 4

2.2 Package Design 4

2.3 Component and Deployment Diagrams 4

III. Elaboration – Iteration 1.2 4

1. Design Model 4

1.1 Dynamic Behavior 4

1.2 Class Design 4

2. Data Model 4

3. Unit Testing 4

IV. Elaboration – Iteration 2 4

1. Architectural Design Refinement 4

2. Design Model Refinement 4

V. Construction and Transition 5

1. System Testing 5

2. Future improvements 5

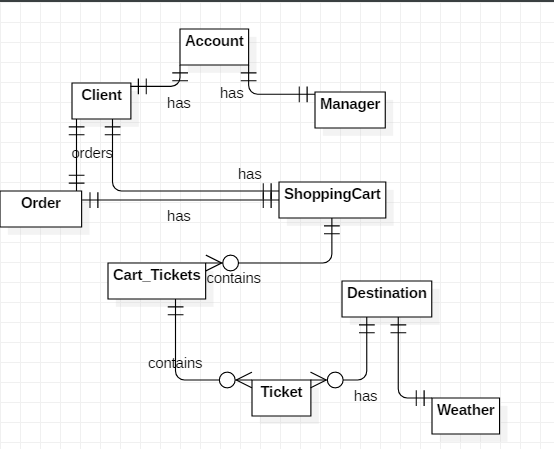
VI. Bibliography 5

# Project Specification

# The application is an online platform that helps the users in purchasing plane tickets by offering the list of vacation offers. The main purpose of the application is to give the user the most affordable and good deals to reduce as much as possible the search time. The system allows the user to see the description of each destination offer and to add plane tickets to the chart and buy them. Also, each destination specifies the weather at the time and the forecast for the next days. The system is developed with a user-friendly GUI, users have to first login into the platform to view the offers and destination.

# Elaboration – Iteration 1.1

# Domain Model



# Architectural Design

## Conceptual Architecture

**Model View Controller Architecture**

MVC Pattern stands for Model-View-Controller Pattern. This pattern is used to separate application's concerns.

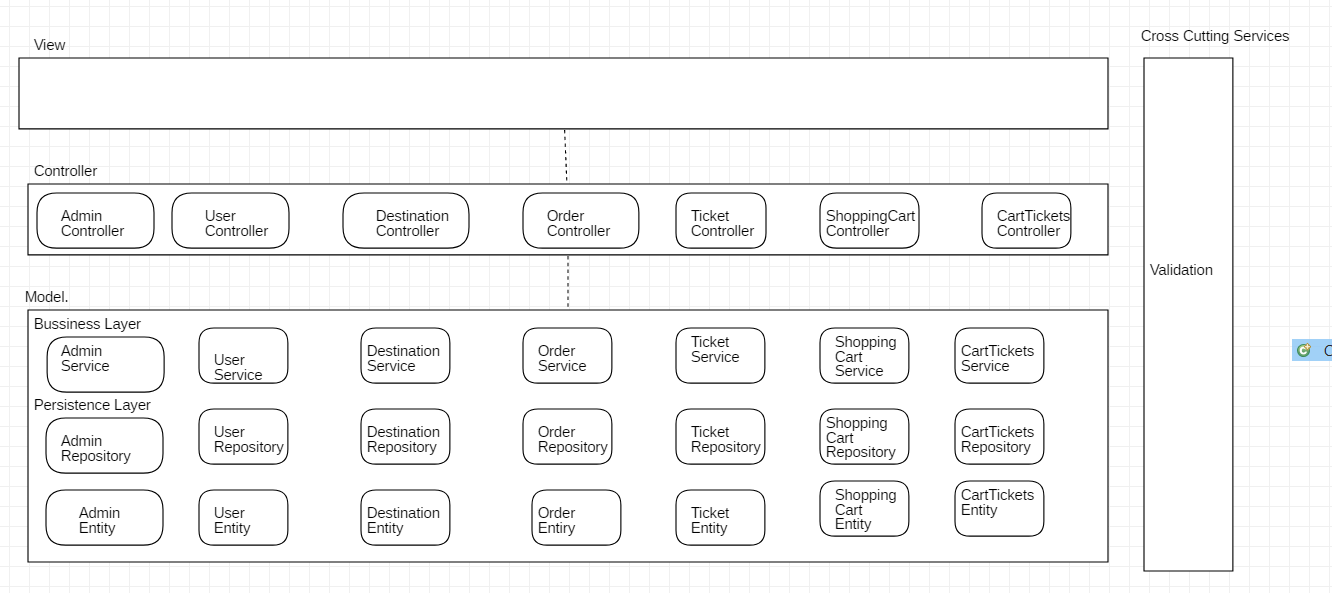
**Model** - Model represents an object or JAVA POJO carrying data. It can also have logic to update controller if its data changes.

**View** - View represents the visualization of the data that model contains.

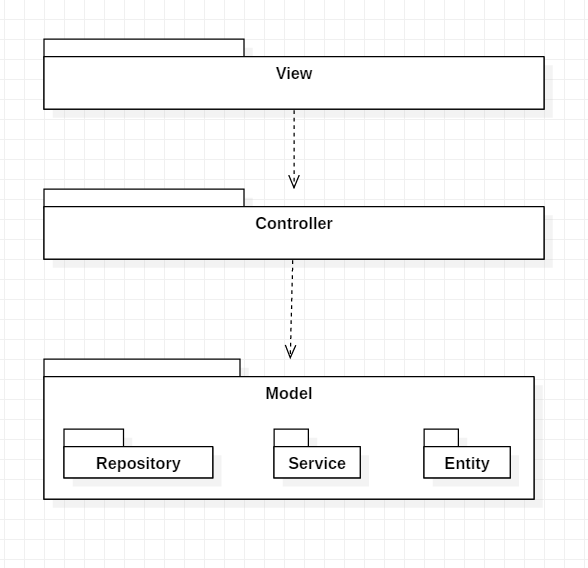
**Controller** - Controller acts on both model and view. It controls the data flow into model object and updates the view whenever data changes. It keeps view and model separate.

The application also has a mini layered architectural pattern. One of the powerful features of the layered architecture pattern is the separation of concerns among components. Components within a specific layer deal only with logic that pertains to that layer. Business layer**.** This layer implements the core functionality of the system, and encapsulates the relevant business logic. It generally consists of components, some of which may expose service interfaces that other callers can use. Data layer. This layer provides access to data hosted within the boundaries of the system, and data exposed by other networked systems; perhaps accessed through services. The data layer exposes generic interfaces that the components in the business layer can consume.

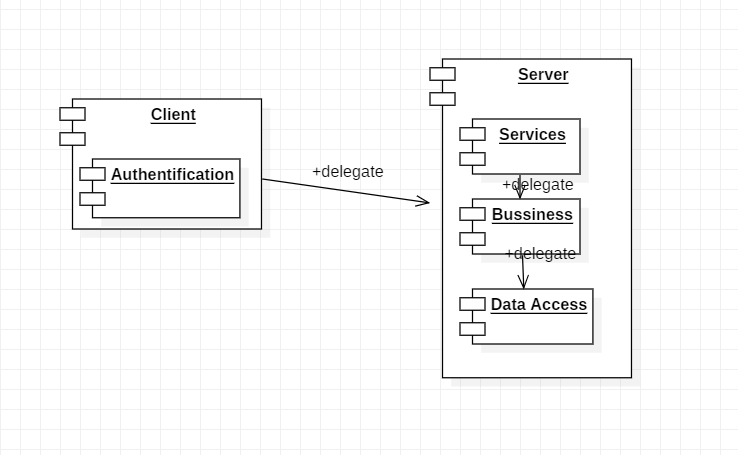
**Observer Design Pattern -** Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. Every time a ticket is added to the shopping cart, the total price of the cart is updated to its value. Also, every time a ticket is removed from the cart, or the shopping cart is bought, the total price of the cart is updated.



## Package Design



## Component and Deployment Diagrams



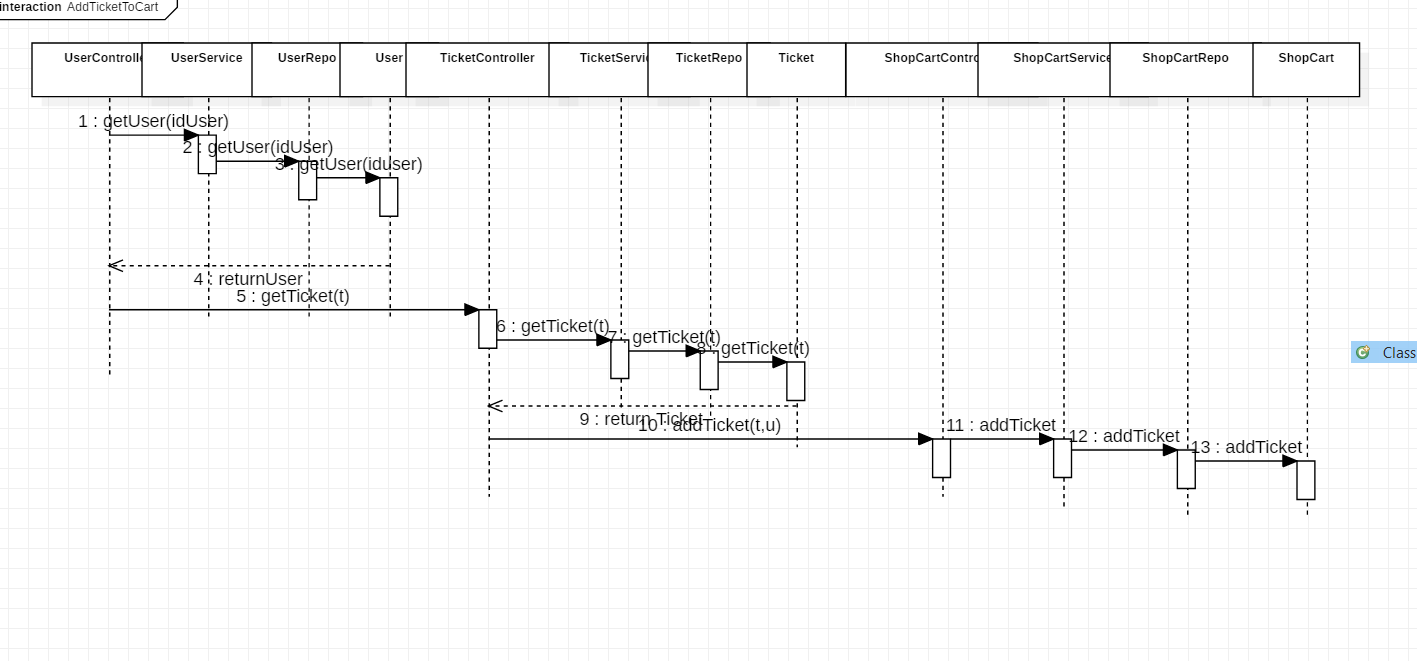
# 

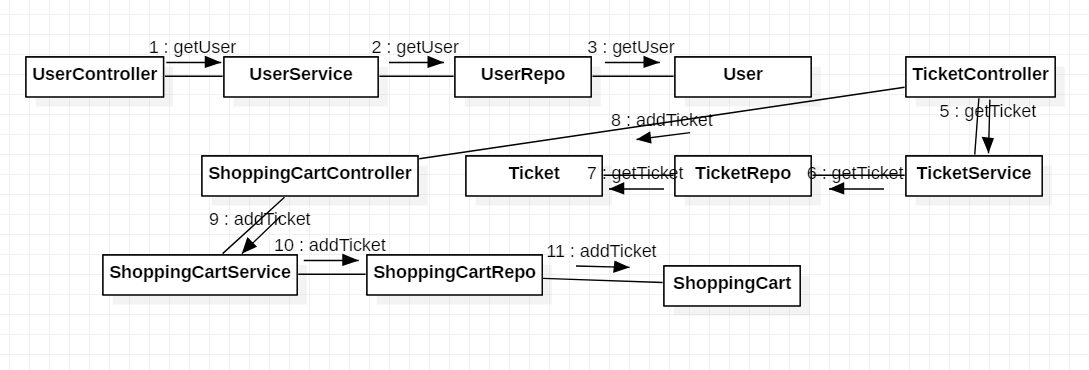
# Elaboration – Iteration 1.2

# Design Model

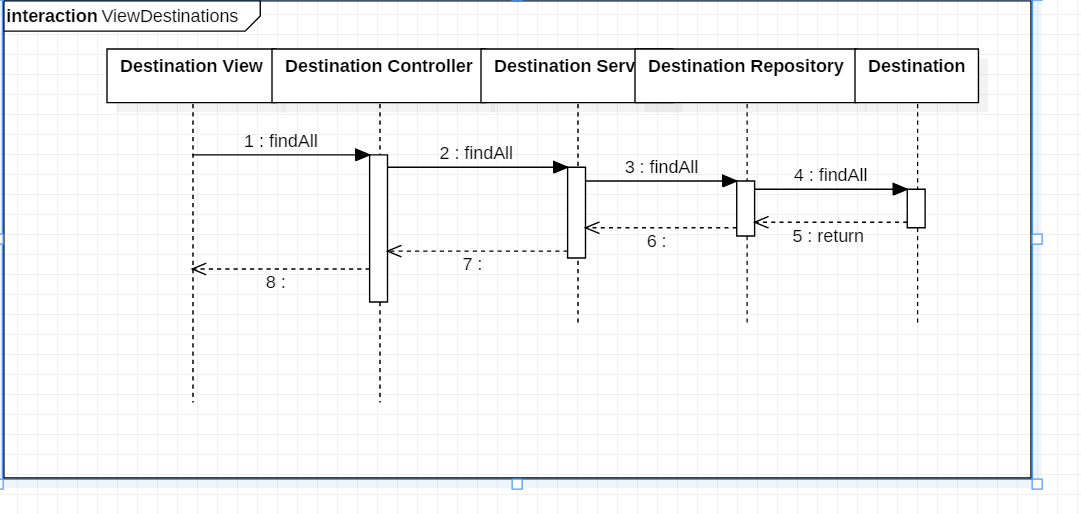
## Dynamic Behavior

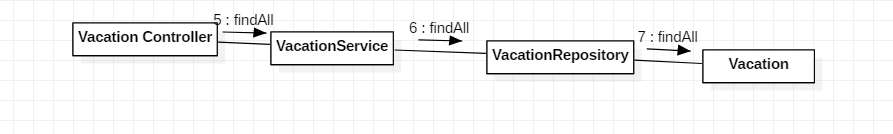
1. Add Ticket to Cart



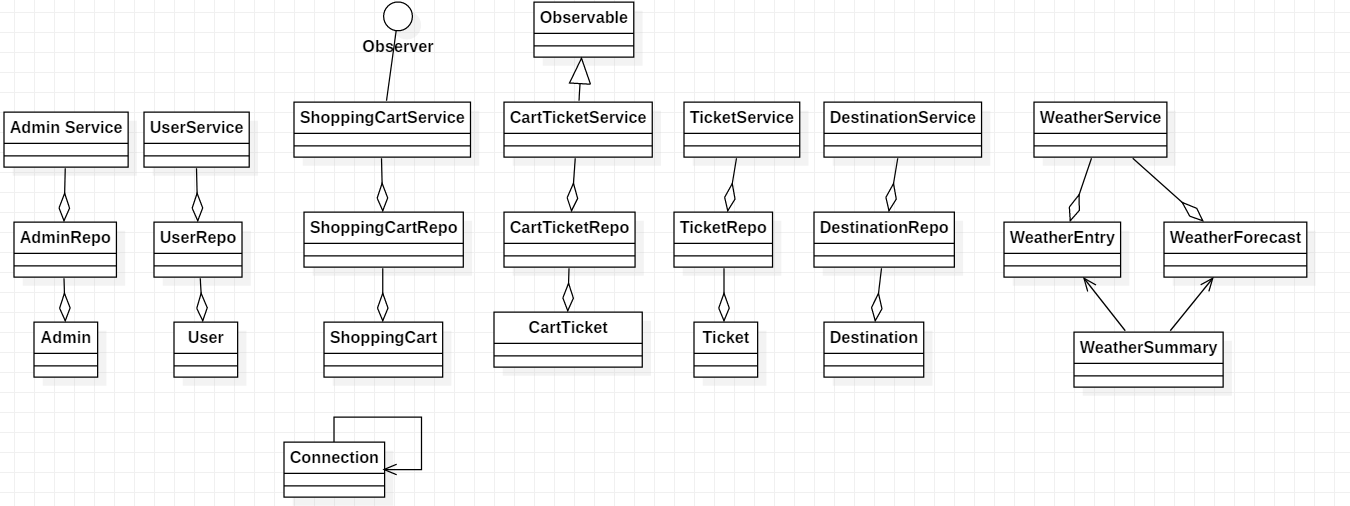


1. View Destinations

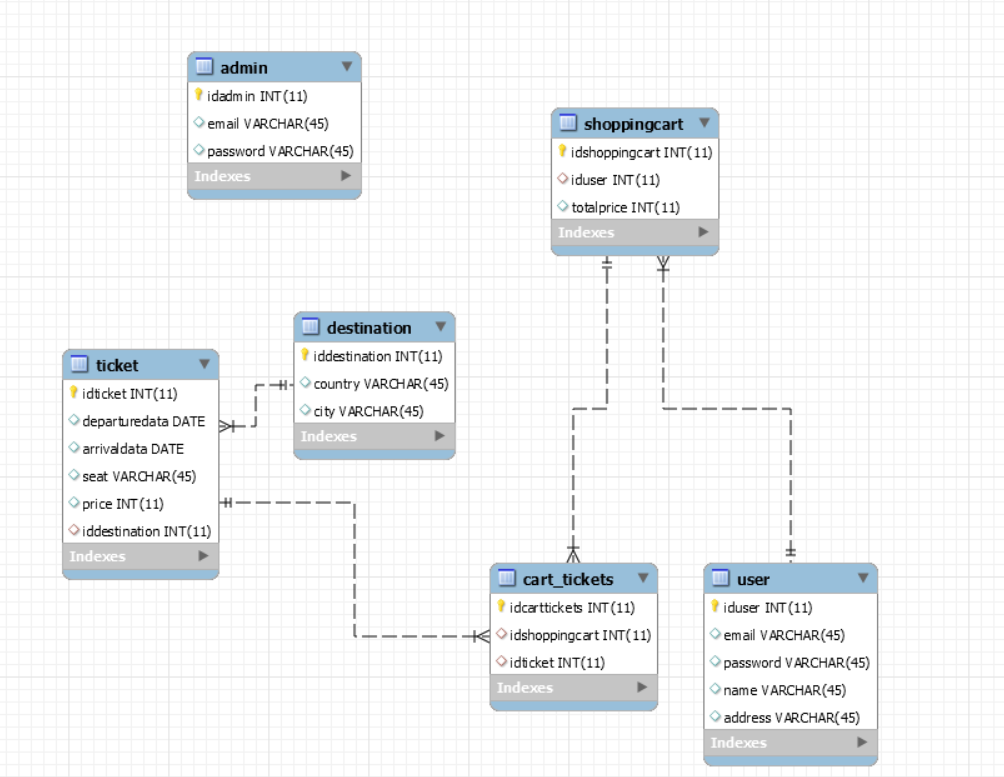




## Class Design



# Data Model



# Unit Testing

Junit4 is used for testing the services of an user. (CRUD operations for all the scenarious).

# Elaboration – Iteration 2

# Architectural Design Refinement

In the conceptual architecture I modified the diagram by adding a weather service, controller and a package for the weather classes: weather, weather forecast and weather summary. Also, in the deployment diagram I added the service for weather (<https://openweathermap.org/weather-conditions>).

Other than that, the architecture of the project remains the same.

# Design Model Refinement

## The class diagram was extended by adding the Weather classes. The domain model was completely modified by adding relationships (1:1,1:n) instead of aggregation, as it was on the beginning.

# Construction and Transition

# System Testing

The application was tested using the client-side feature. The user can log in as a regular user or as an admin and interacts with the application. Therefore, every action from the following was tested: log in, to create a new user, view profile, modify a profile, to create a shopping cart, to buy the shopping cart, remove an item from a shopping cart, to view the destinations with the weather, to buy a ticket. As an admin: to create a user, to modify a user, to insert, modify and delete a destination, ticket or user.

The system has passed all the tests and the behavior was as expected.

# Future improvements

The future improvements: the application security can be implemented using Spring Security, the user can have a wish list. The project can also be migrated to a non-relational database.

# Bibliography

<https://spring.io/guides/gs/spring-boot/>

<https://openweathermap.org/weather-conditions>

<https://sourcemaking.com/design_patterns/observer>

<https://www.thymeleaf.org/>

<https://www.thymeleaf.org/doc/tutorials/3.0/thymeleafspring.html>