Football ticketing app

Analysis and Design Document

Student: Ștefan Ciuprina

**Group:30434**

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Description** | **Author** |
| <dd/mmm/yy> | <x.x> | <details> | <name> |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

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 purpose of this project is to design and implement a football ticketing app, as the name suggest. There are two types of users: the administrator (the football club responsible of creating football match events, specifying the place and the date of time of the game, as well as the cost and the total number of tickets) and the cashier (the one responsible for selling the tickets provided by the football club).

# Elaboration – Iteration 1.1

# Domain Model

To build our domain model, we first have to define the entities. Since it is a ticketing app for football matches, we must certainly have the following entities:

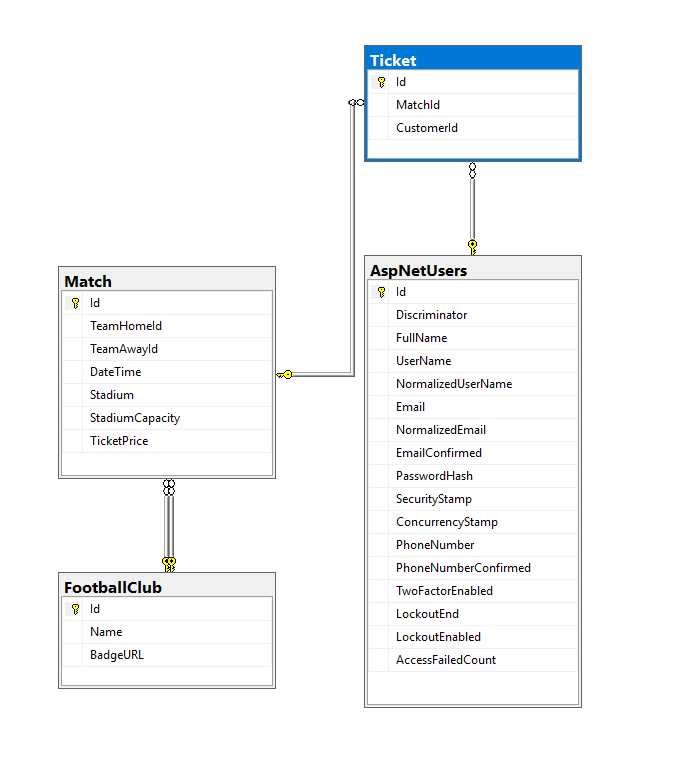
* Football team
* Football match (the event)
* Ticket

Along with these base entities, we may add:

* Stadium – the place where the event will take place
* League – the league the teams are playing in

There are also some relationships we have to establish between the entities mentioned above.

* Football team *has many* Football matches (composition)
* Football match *has a* Stadium (aggregation)
* Football match *has many* Tickets (composition)
* League *has many* Football teams (aggregation)



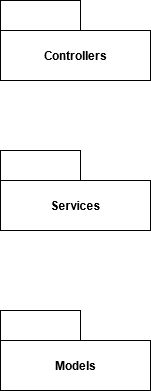
# Architectural Design

## Conceptual Architecture

This project is designed using Layered architecture, separating the project into three parts: data access, business logic and presentation. Using this approach, the UI and the Server are clearly separated (the presentation separated from the business and data access), and the backend is better organized.

## Package Design

The packages of this project will Layered Architecture specific, with controllers for managing the http requests to communicate with the UI.

**

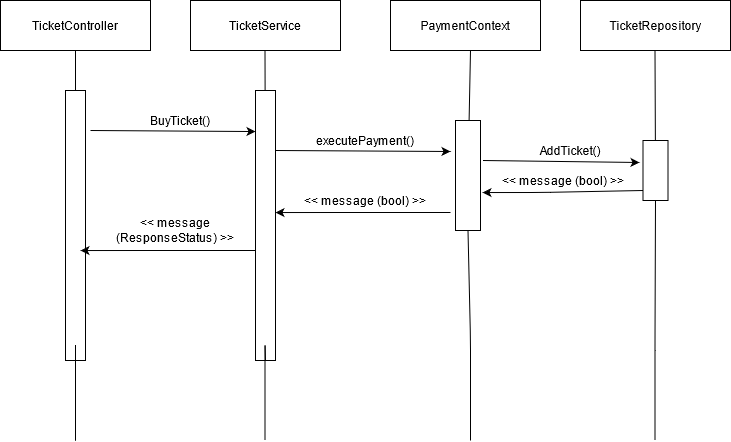
## Component and Deployment Diagrams

# 

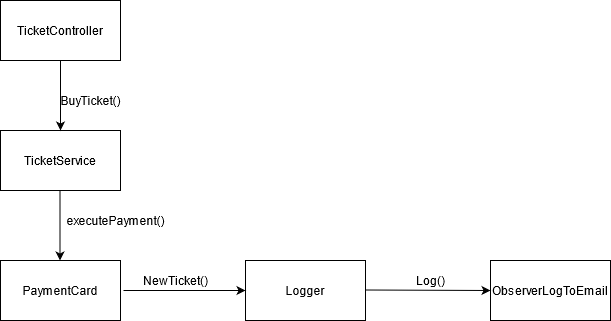
# Elaboration – Iteration 1.2

# Design Model

## Dynamic Behavior



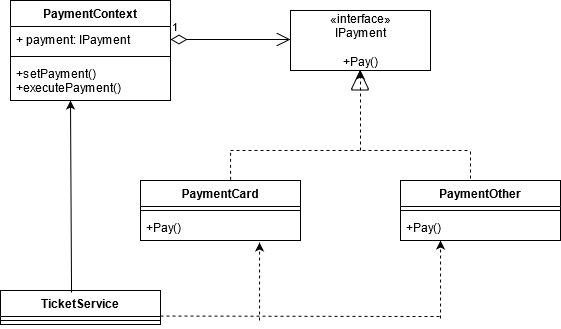
## 



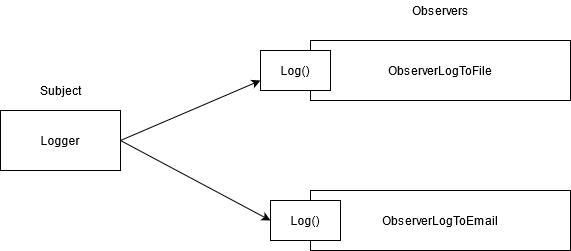
## Class Design

This project makes use of 2 famous design patterns, namely the Strategy Design Pattern and the Observer Design Pattern.

The Strategy Design Pattern is used when deciding which form of payment should be used. There are 3 options, but 2 of them are grouped in two, as follows: card payment, or other (bank transfer or cash). If the first option is used, the card details are verified and the ticket is purchased instantly, while if the other payment is chosen, only a reservation is being made, requesting the user to complete the specified form of payment he/she choose (either transfer the amount or go to a cash register and pay cash). The UML diagram illustrating this pattern can be seen in the picture below.

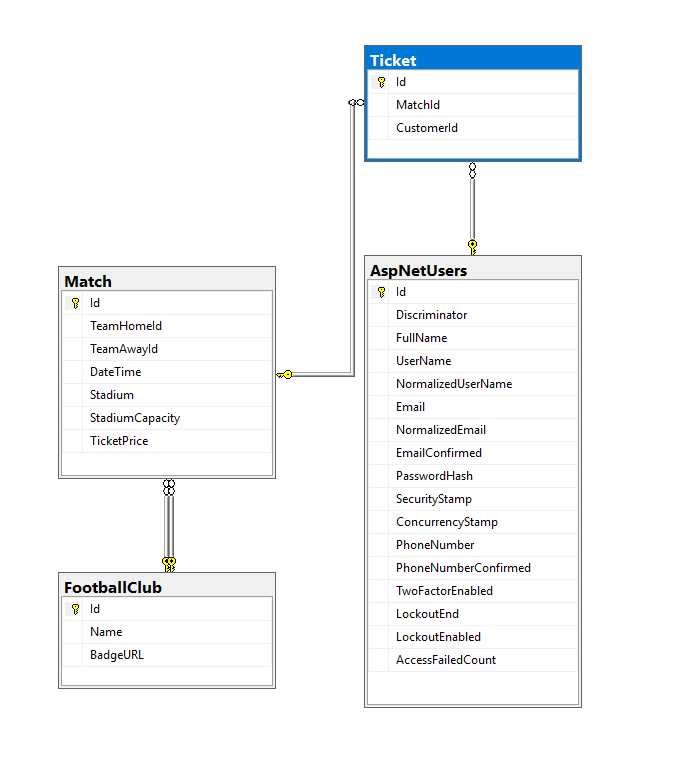
**

The other design pattern used in this project is the Observer Design Pattern. There are 2 observers present: one that writes the data (the tickets) to a log event file and one that sends the event via email. The two subscribers are implemented as separate observers and the notification as a subject. A diagram of this implementation can be seen below:

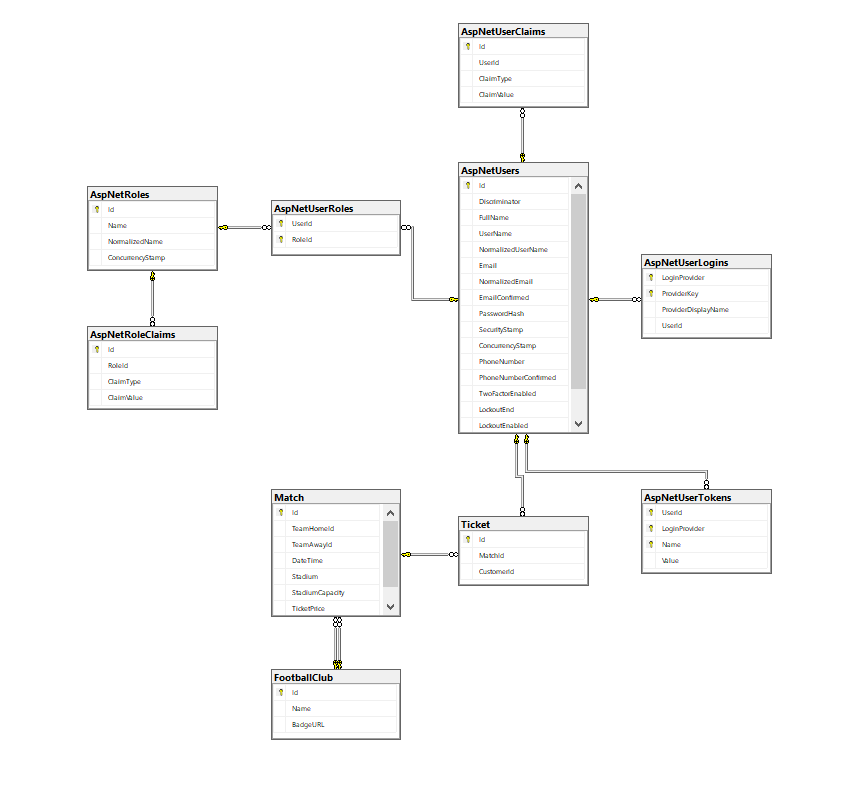


# Data Model

# Main tables + Users



All tables:



# Unit Testing

A unit test has been made for checking the DbSet of FootballClub of the FootballTicketingContext.

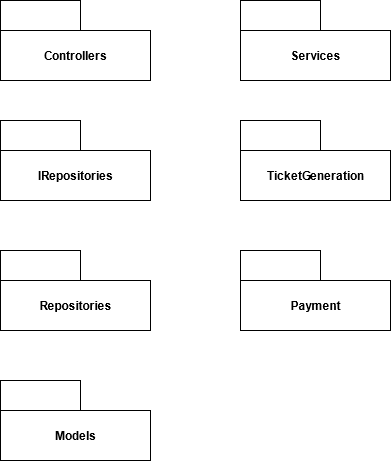
# Elaboration – Iteration 2

# Architectural Design Refinement

No refinement has been made. The project uses Layered Architecture, having Controllers, Services and Repositories.

# Design Model Refinement

The final version of the packages are as shown in the diagram below.

****

# Construction and Transition

# System Testing

The system has been tested manually using Swagger.

# Future improvements

Future improvements may consist of creating a UI for the APIs of this project, making it easier for the user to use the application and have a good experience with it.

# Bibliography

Observer Pattern: <https://www.srinadimpalli.com/2019/04/observer-pattern-in-net/>

Strategy Pattern: <https://refactoring.guru/design-patterns/strategy>