National Theater Application

Student: Andrei Baraian

**Group: 30431**

Table of Contents

1. Requirements Analysis 3

1.1 Assignment Specification 3

1.2 Functional Requirements 3

1.3 Non-functional Requirements 3

2. Use-Case Model 3

3. System Architectural Design 3

4. UML Sequence Diagrams 3

5. Class Design 3

6. Data Model 3

7. System Testing 3

8. Bibliography 3

1. Requirements Analysis

# Assignment Specification

The assignment requested to implement an application for the National Theater of Cluj. The application has two types of users, an admin and cashiers, both having to provide a username and password for authentication. They can perform various operations on the shows and tickets for sale.

# Functional Requirements

The application should have two types of users (a cashier user represented and an administrator) which must provide a username and a password to use the application.

The administrator user can perform the following operations: CRUD on cashiers’ information, CRUD on the list of shows that are performed at the theater, keep track of the Genre (Opera, Ballet), Title, Distribution list, date of the show and the number of tickets per show. From time to time he can export all the tickets that were sold for a certain show (either in a csv or xml file).

The cashier can perform the following operations: sell tickets to a show. A ticket should hold information about the seat row and seat number. The system should notify the cashier that the number of tickets per show was not exceeded. A cashier can see all the tickets that were sold for a show, cancel a reservation or edit the seat.

The data has to be stored in a database, so I chose MySQL since it has a really easy integration with Java. Regarding architecture, I used the Layers architectural pattern.

# Non-functional Requirements

*Availability*

The application should be available for most of the time after it is up on the market. The maintenance time should be reduced as much as possible. This can be enforced by having a good design which can easily respond to new requests or solve bugs.

*Security*

The security of the app consists of having different access rights for the administrator and normal user (cashier in our case). This is implemented by offering different functionalities based on the login user. Another security feature is to store the password in an encrypted form, using a one-way encryption algorithm. I have used SHA-256, provided by the MessageDigest class in Java. Message digests are secure one-way hash functions that take arbitrary-sized data and output a fixed-length hash value.

*Testability*

I provided unit tests for the limit of number of tickets that can be sold for a show and also for the correctness of the encryption algorithm.

*Usability*

The application is easy to use by anyone who first interacts with it. The messages, the button messages and the layout are pretty self-explanatory and there is no ambiguity regarding the operations that each user can do.

*Design Constraints*

The implementation language is Java. The database is MySQL. For the UI, I have used the JavaFX, which tends to replace the old-fashioned Swing.

The architecture of the project is Layer Architecture. The software is divided in 4 layers: Presentation, Service, Repository, Unit Tests. The Service and Repository layers have each their own model.

2. Use-Case Model

Use case: CRUD on DB tables

Level: user-goal-level

Primary actor: admin

Main success scenario: successful CRUD operations

Extensions: failed CRUD operations

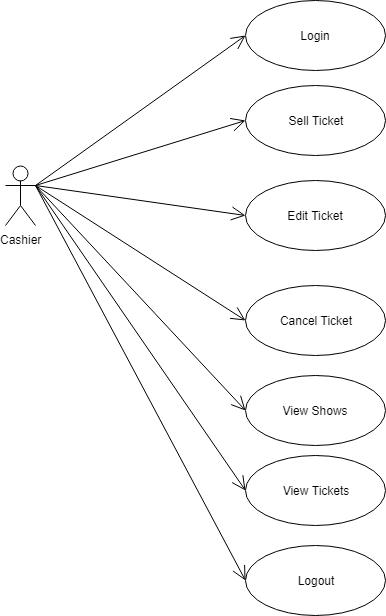
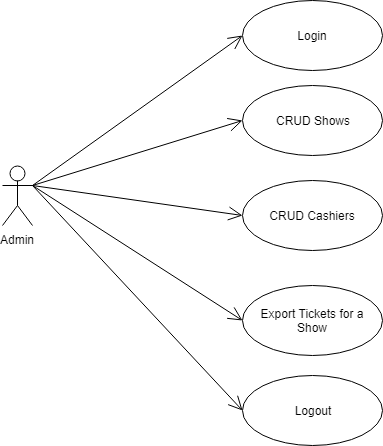
Use case: Sell ticket

Level: user-goal-level

Primary actor: cashier

Main success scenario: successfully reserve a seat and emit a ticket

Extensions: fail to reserve a seat, not enough tickets



3. System Architectural Design

**3.1 Architectural Pattern Description**

The architectural pattern used was Layers pattern. For this, the application was divided into 3 main components: Presentation, Service, Repository. Service and Repository, each have their own models which can be mapped one to the other.

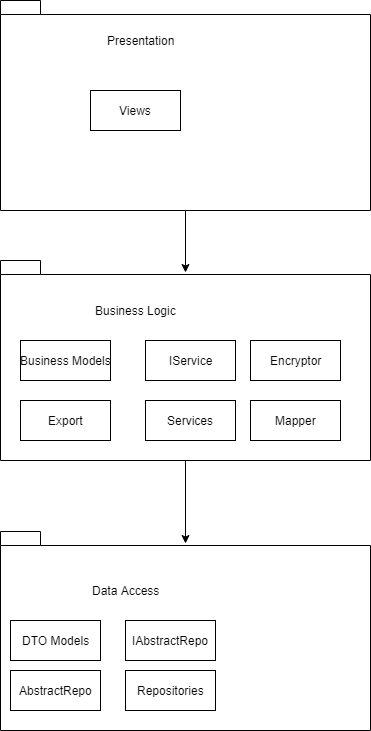
**3.2 Diagrams**

*System conceptual architecture*

**

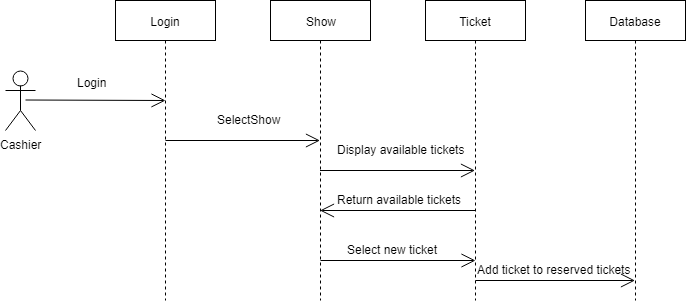
I used the Layers architectural pattern, which consists of having 4 main layers: Presentation, Business Logic, Data Access and Database. In the Presentation layer I have the views and models of the business logic which are rendered in windows. In the Business Logic, there are the services which are used by the application to call methods to the Data Access layer. Here, the presentation model is mapped to the database object. In the Data Access layer, all the queries to the DB are created and executed. To avoid code duplication, I have a created an abstract repository which performs all the CRUD operations, since they are the same for all objects. The other repositories just extend this abstract class and infer their type at run time.

*Package diagram*

**

4. UML Sequence Diagrams

Sequence diagram for cashier selling a ticket.

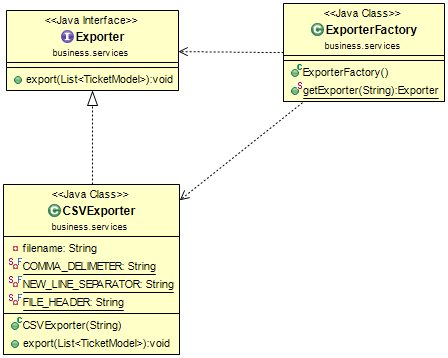


5. Class Design

**5.1 Design Patterns Description**

For exporting the tickets for a particular show, I have used the Factory method design pattern, which is a creational design pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created. It prevents clients from knowing the concrete classes used in the instantiation process. Minimizes cost of modification on the client side when concrete classes change and delegates creation to subclasses and localizes the creation process.

**5.2 UML Class Diagram**



6. Data Model

The data models used in the application are Admin, Cashier, Show, and Ticket. The Admin and Cashier models share the same description: username, password, first name and last name. The show is characterized by title, category, distribution list, date, available tickets and remaining tickets. The Ticket entity is described by the seat row, seat column and the id of the show.

7. System Testing

For testing, I used the JUnit library from Java. I made the unit tests on the limit of tickets that can be sold for a show and for the correctness of the encryption algorithm. If the number of remaining tickets for a show is equal to 0, it means that no more tickets can be sold for that show.

8. Bibliography

<https://www.uml-diagrams.org/package-diagrams.html>

<https://www.youtube.com/watch?v=FLkOX4Eez6o>

<https://www.tutorialspoint.com/junit/junit_test_framework.htm>

<https://dzone.com/articles/java-the-factory-pattern>