<Assignment 1>

Analysis and Design Document

Student: Moldovan Anda

**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 application is created for National Theatre of Cluj. It two types of users, cashiers who can sell tickets to existing shows, and administrators, who can modify the shows and the cashiers.

# Functional Requirements

The system has the data stored inside a MySQL database. For every operations performed by any types of users on the data, it is modified inside the database model.

The administrator, has the right to do the database operations both on the shows (add shows, modify shows, and delete shows), but also on the cashiers (add, edit or delete a cashier).

It is assumed that the administrators are added to the database manually, and while they are able to identify into the system, they are mot able to view or modify their own information or the other administrator’s information.

The passwords for both administrators and cashiers are encrypted using one-way encryption, more explicitly MD5. This assures that once the user is created, no one in the system can view the actual password.

The administrator can also export an xml file holding the information about the tickets sold to a specific show.

The cashier can sell tickets, edit reservations to a show or delete them (deleting a reservation is approached as a ticket return). Protection measures were taken into account so that the cashier would not be able to sell more tickets that are available for a show.

# 1.3. Non-functional Requirements

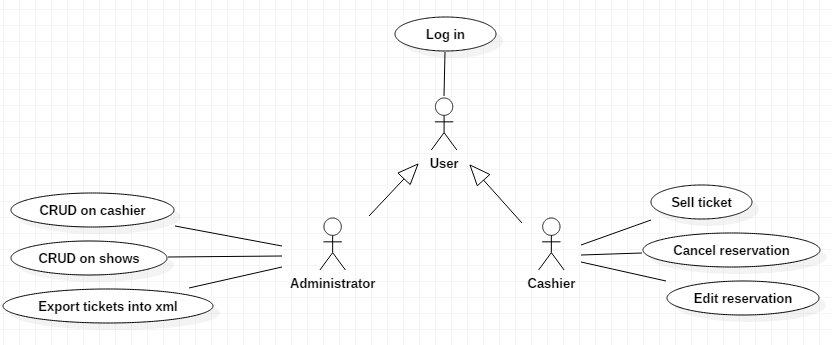
Because working with an external database, the first aspect that was considered was to protect the data integrity and correctness. All the data that any type of user inputs into the system is checked and if it fits some requirements it is them added to the database. Errors may appear if the data is not something that the programmer has predicted as being able to produce a fault. The protection was also ensured at the user interface level by giving the user the possibility to make errors, for example as many fields as possible are drop down lists, tat do not allow input only directly from the database, or for the cashier interface, only the title can be modified, and the row and seat for the ticket.

The scalability is limited to the size of the database the client is able to withhold. Although this can be fixed by implementing features that delete the shows and tickets form the database after they are played.

Unfortunately, any features that do not strictly reflect the application, must be added by a programmer, and cannot be implemented from the user interface.

As previously discussed the security of the application is ensured by password encryption and separating cashier and administrator responsibilities at the top layer and data protection at the bottom layer. As the application will only be deployed locally, the risk of outside intervention is eliminated.

2. Use-Case Model



Use case: CRUD on cashier

Level: user-goal level

Primary actor: Administrator

Main success scenario: operations on cashier table executed successfully

Extensions: failure on operation on cashier table

Use case: CRUD on shows

Level: user-goal level

Primary actor: Administrator

Main success scenario: operations on show table executed successfully

Extensions: failure on operation on show table

Use case: Export tickets into xml

Level: user-goal level

Primary actor: Administrator

Main success scenario: sold tickets for a show successfully exported into a xml file

Extensions: failure on exporting the file

Use case: Log in

Level: user-goal level

Primary actor: User

Main success scenario: successful log in as one of the two existing users

Extensions: failure on log in

Use case: Sell tickets

Level: user-goal level

Primary actor: Cashier

Main success scenario: ticket sold successfully

Extensions: failure on selling the ticket

Use case: Cancel reservation

Level: user-goal level

Primary actor: Cashier

Main success scenario: reservation canceled successfully

Extensions: failure on canceling the reservation

Use case: Edit reservation

Level: user-goal level

Primary actor: Cashier

Main success scenario: reservation edited successfully

Extensions: failure on editing the reservation

3. System Architectural Design

**3.1 Architectural Pattern Description**

The architectural pattern used is layered or more specifically, three-tier.

The components are organized into three horizontal layers, each layer performing a specific role within the application. The three layers used in developing this application are: data access, service and presentation, each one of them having specific roles and responsibilities.

Data Access Layer

Handles a connection to the database, creating it and closing it, as well as a class for each table in the database to store data and the repository class, that contains instructions specific to each class, or CRUD operations that are performed to the corresponding table inside the database.

Service Layer

Creates the services the interface will need, based on the basic queries in the repository for each class. Also, it has the responsibility of ensuring that the data has the correct form before saving it to the database, and as a feature it contains the class that creates an xml file and populates it with sold tickets for a show on request, from the administrator.

User Interface Layer

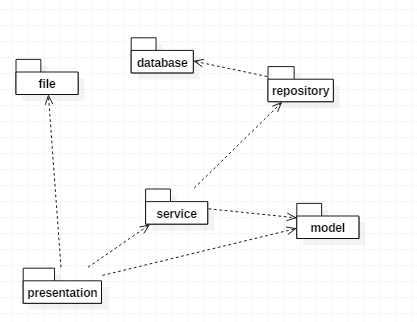
As the names says, it creates the user interface and deals with different user requests, by using the service layer to complete them.

As a general rule, for a layered architecture, the communication is done in only one way and only to the layer bellow, for instance the user interface communicates with the service layer to service requests and the service layer to the data access layer in order to use the interface. Any other form of communication between layers is not permitted.

The advantage of the layered architecture is that it creates the so called layers of independence, meaning that changes to one layer do not affect the others, and as a result the application is more loosely coupled than it would be otherwise.

**3.2 Diagrams**

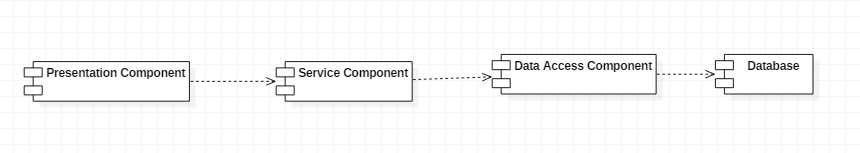
Package Diagram

Because of the architectural pattern, the whole system gravitates around the service, and model package, members of the service layer.

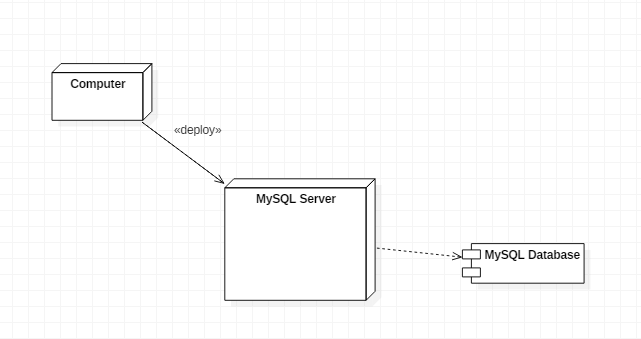
They constitute the middle layer of the system, so every request made to the database goes through it.

The presentation package is the top package in the system, so it uses service layer to do the desired actions.

Component Diagram

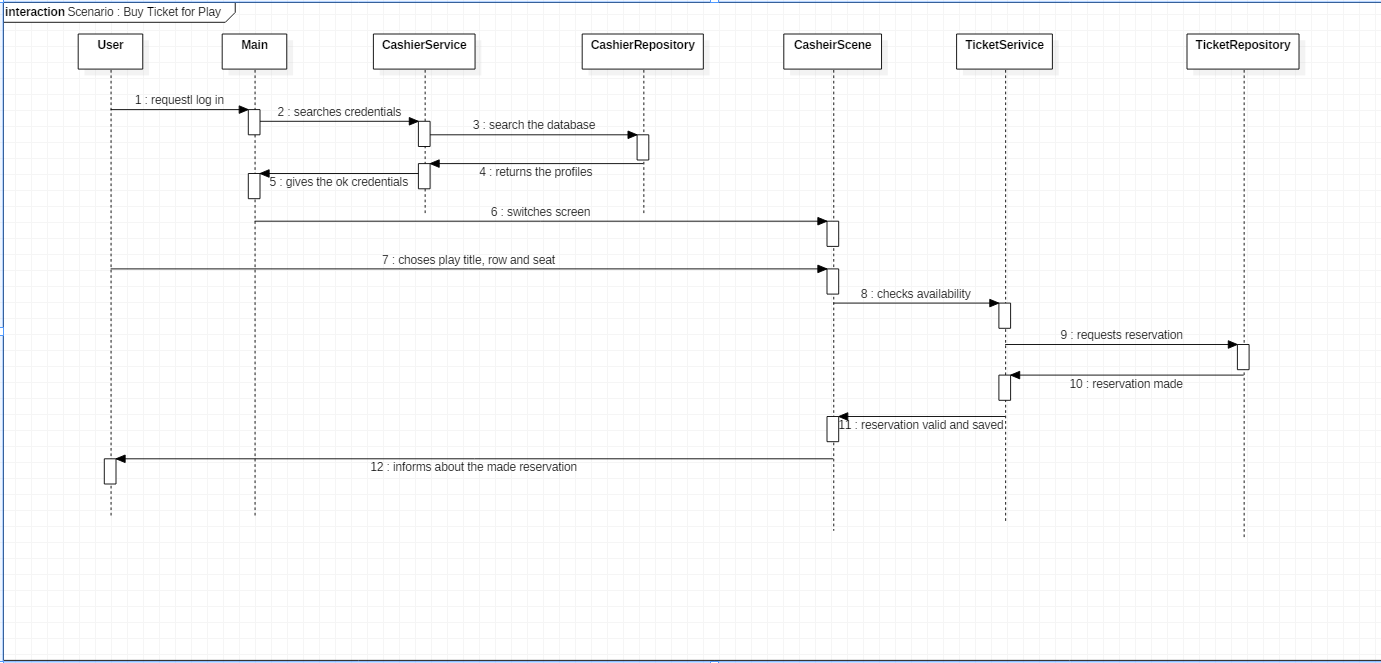


Deployment Diagram



Because the system is used as a desktop application, it requires that the database is stored locally, and the database server is generated locally, in this cast, through the use of xampp MySQL database server.

4. UML Sequence Diagrams



Chosen Scenario : Cashier wants to sell a ticket.

As seen, in the diagram above, the cashier first must log in, then choose a show to sell the ticket to, and select a row and seat. The ticket service, checks the validation of the ticket, if the row and sear is available for that paly, and if everything checks, the seat is booked and the cashier is notified about this in the user interface by the means of a message and the reservation should appear in the list on the interface window.

5. Class Design

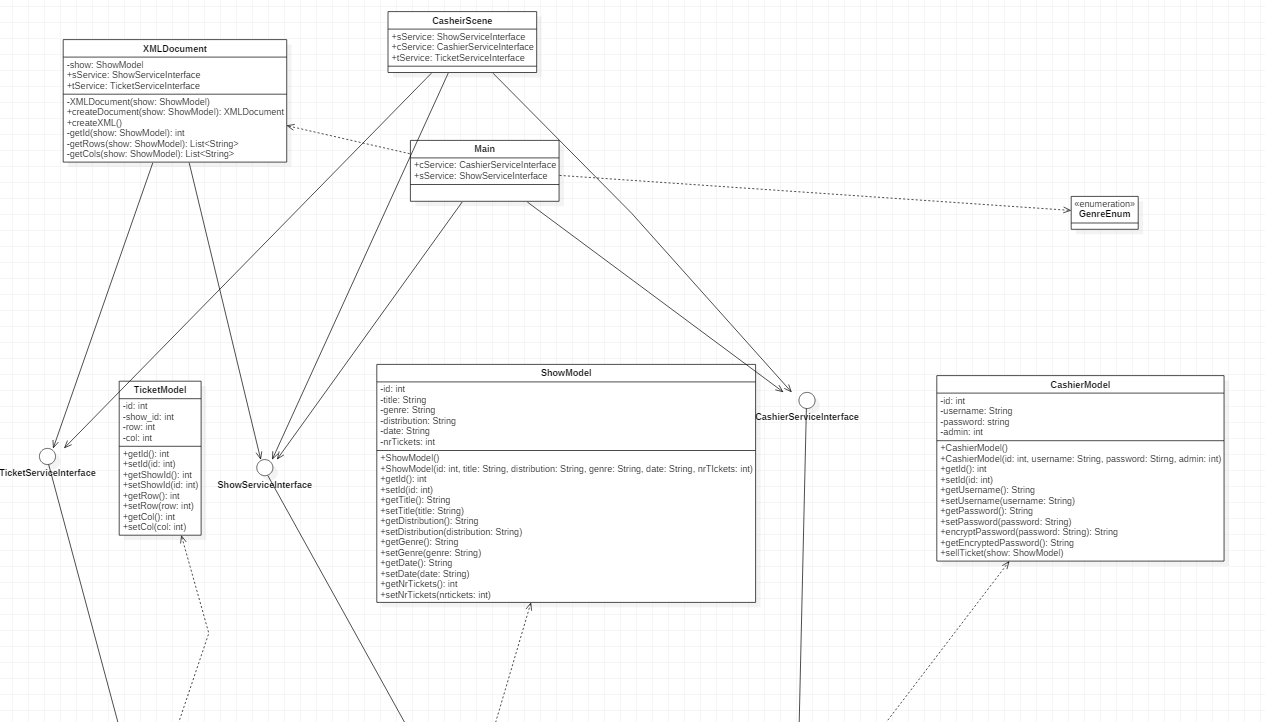
**5.1 Design Patterns Description**

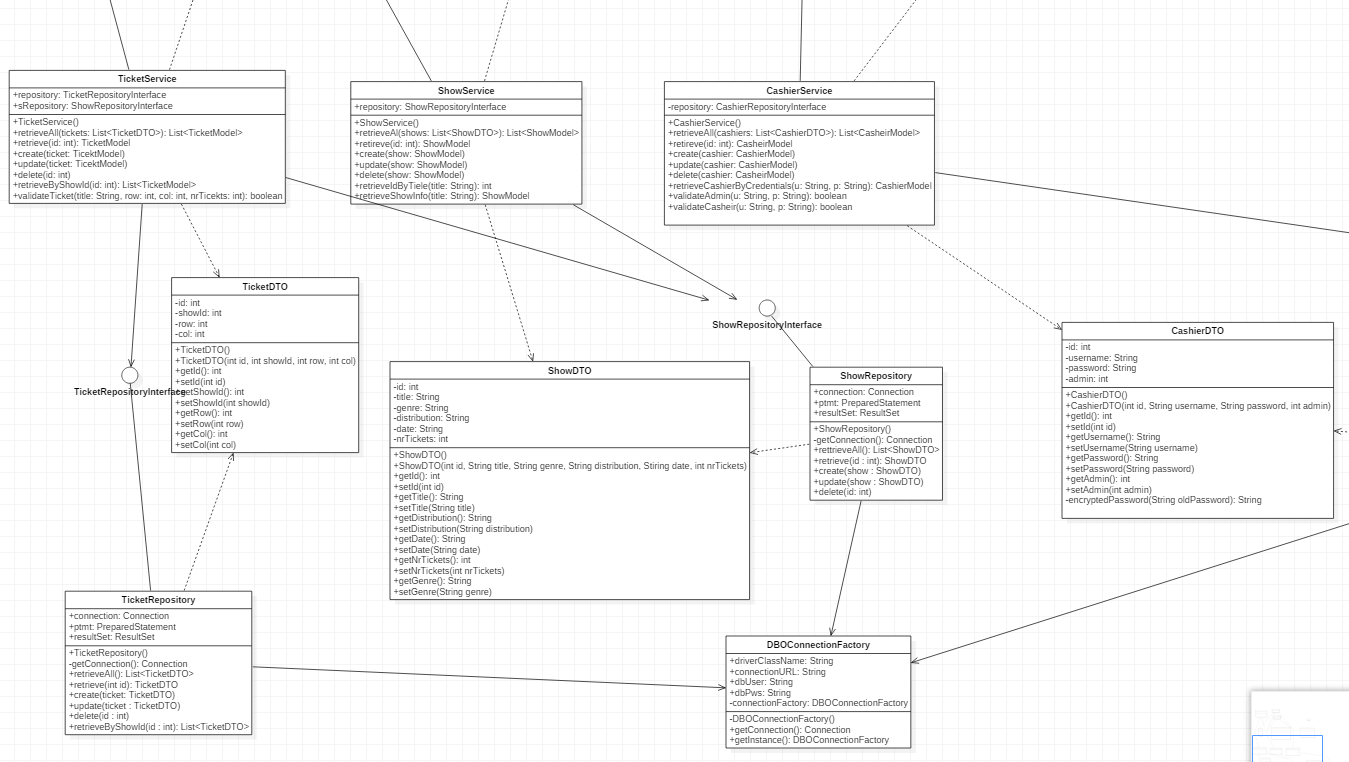
Factory Method

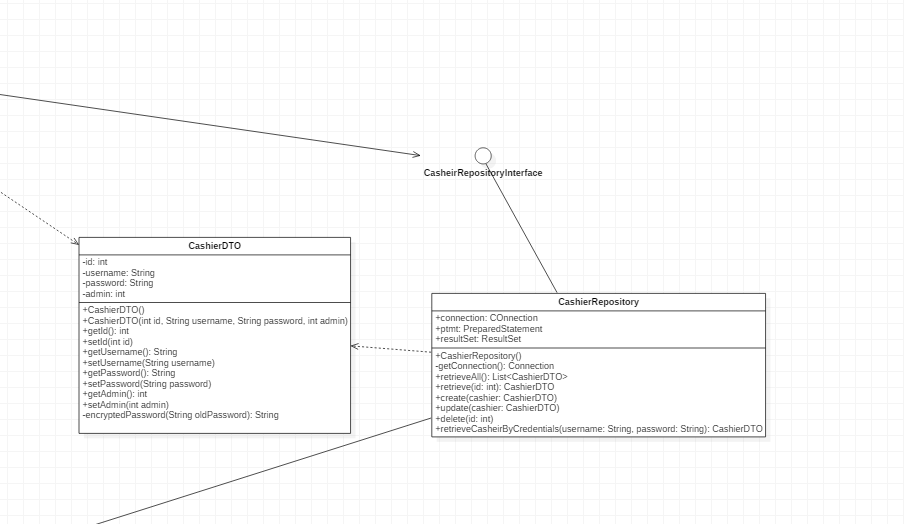
Allows the creation of objects without having to deal with the constructor of that class. The constructor itself is hidden from the user, and the object is instantiated by the use of a static method.

The design was used for both the database connection, and the xml file document.

**5.2 UML Class Diagram**

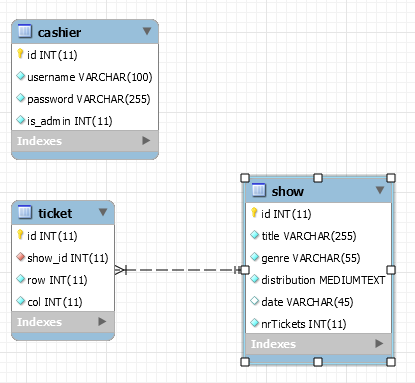






A presented in the class diagram, the communication between the layers happens only in one direction, and by the use of interfaces.

6. Data Model

**As the design shows, the

Database contains three tables: show, cashier and ticket.

Cashier is the most straight forward, as it is only use when a user logs in or when they are added into the system or edited by an administrator.

Inside the table, the difference between the cashiers and the administrators is made by a field is\_admin that is 0 for cashiers and 1 for administrators.

The table show has a 1 to many relationship with the ticket show, so for each ticket sold for a show a new row containing the relevant information is stored into the ticket table.

7. System Testing

Each component of the system was tested locally as it was developed, to ensure that whatever errors may arise in the future, the base layers of the application were working.

More thoroughly tested trough unit testing, were the password encoding and ticket selling mechanism for a show.

For the password, the encoded one coming from the database, needed to be matched with a password coming, finally, from the user input. The way to test the match is to compare the encoded password to the one that is given as input, after the last one is run through the encoding mechanism. This follows the assumption that a string that is run through the MD5 encoding algorithm, will always have the same value, and no other string will be able to be encoded with that value.

The test for the ticket selling, is much more straight forward. It ensures that once the number of available tickets reaches 0, however many times the cashier wants to sell a ticket for that show it will not happen. Visually, the cashier is informed about this.

8. Bibliography