ASSIGNMENT A1

Student: Tatomir 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

Use JAVA/C# API to design and implement an application for the National Theater of Cluj. 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:

1. CRUD on cashiers’ information.
2. CRUD on the list of shows that are performed at the theater. Keep track of the Genre (Opera, Ballet), Title, Distribution list (a long string is enough), Date of the show and the Number of tickets per show.
3. 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:

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

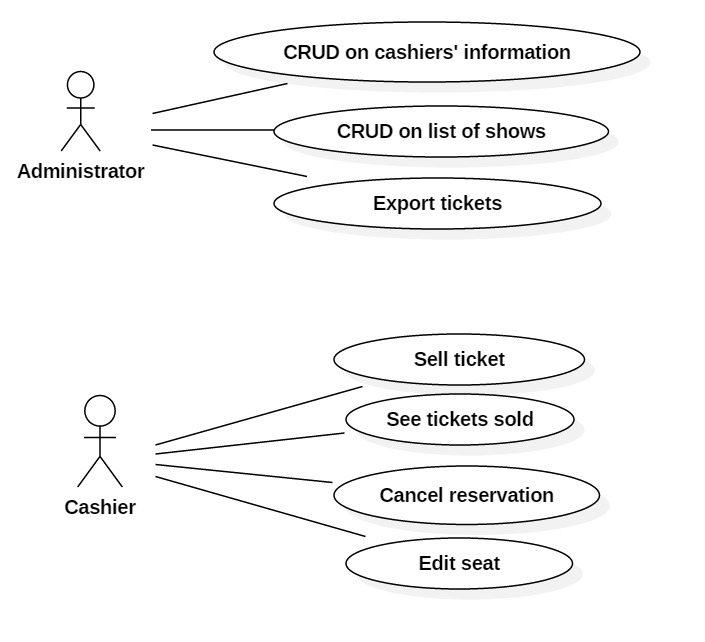
# Functional Requirements

1. The data should be stored in a database.
2. The Layers architectural pattern should be used.
3. Passwords should be encrypted when stored to the database with a one-way encryption algorithm.
4. Provide unit tests for the number of tickets for show exceeded validation and the encryption algorithm.
5. Export to csv/xml using Factory Method.

# Non-functional Requirements

1. The application should be easy to use, by both the administrator and the cashier
2. The application should be secure, storing the passwords of the users encrypted.
3. Extensibility
4. Maintainability

2. Use-Case Model



*Use case: Export tickets*

*Level: User-level*

*Primary actor: Administrator*

***Success guarantee: A file in the selected format, containing the correct information is* created *and stored***

*Main success scenario:*

1. The actor inputs its username and password
2. The actor logs in
3. The actor selects “Manage shows”
4. The actor selects a show from the table
5. The actor inputs a file name in the “filename” text field
6. The actor clicks the “Export CSV” or “Export XML” button
7. A file containing information about the tickets sold, in the desired format is saved.

*Extensions:*

1. The actor inputs an incorrect username or password. Occurs at step 1.
2. A dialog appears warning the user that the username or password is invalid
3. The actor closes the window
4. The actor inputs his credentials correctly

Return to 2.

3. System Architectural Design

* 1. **Architectural Pattern Description**

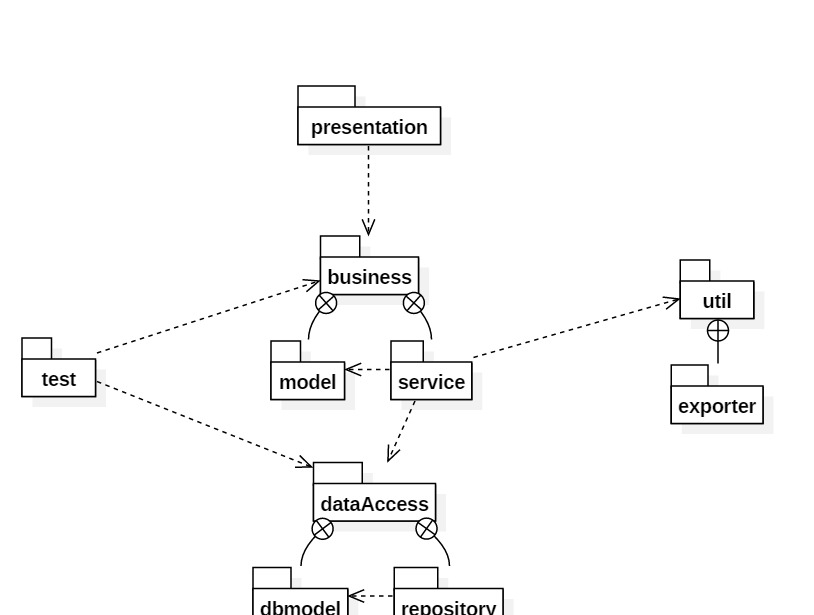
The pattern used in developing the application is the Layers Architectural Pattern. This pattern involves separating the functionality of the application into layers based on their functionality. Each layer only interacts with the layer directly below it and it accesses it through interfaces. Typically, the layers are the presentation, the business/service and database access layer.

In this case, the application is structured into:

* Presentation layer
* Business layer containing classes for the business logic of the application, structured into the model and service packages
* Data access layer containing classes for establishing the connection with and accessing the database, structured into the dbmodel and repository packages

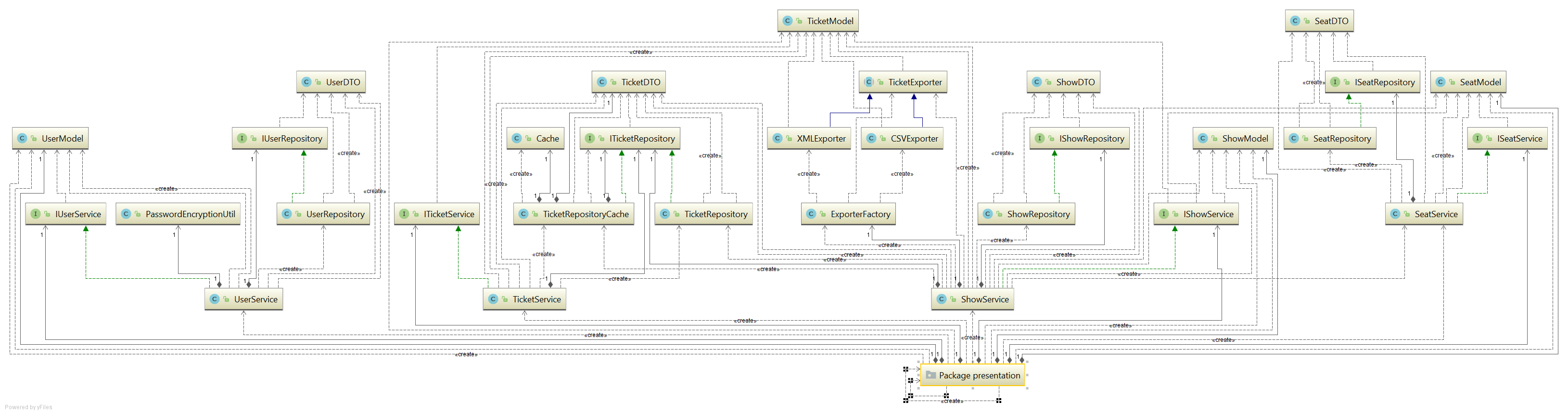
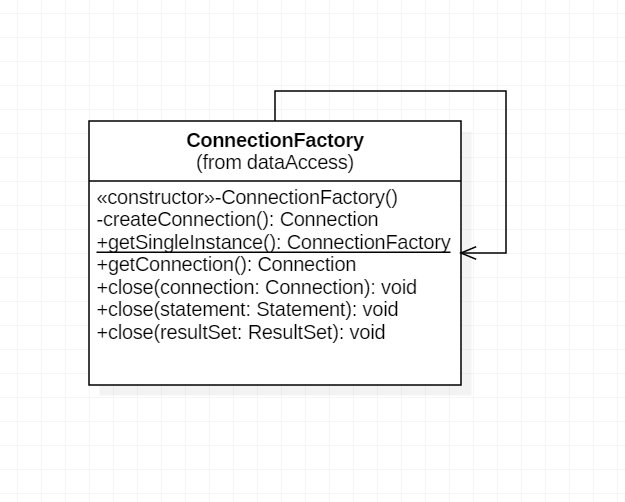
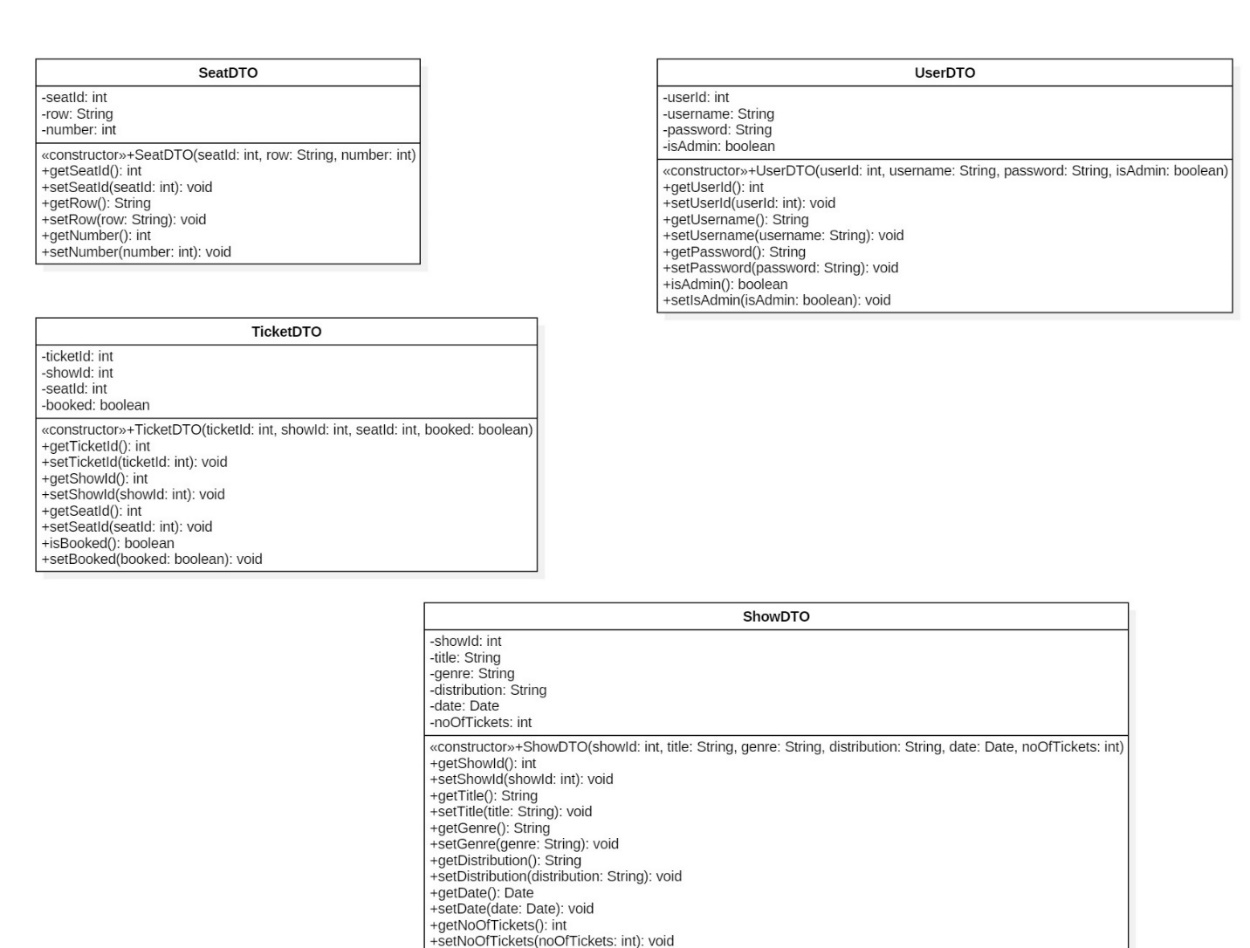
Besides the above packages, the application also contains packages util (for exporting data into csv and xml formats) and test, for testing various functionalities of the application.

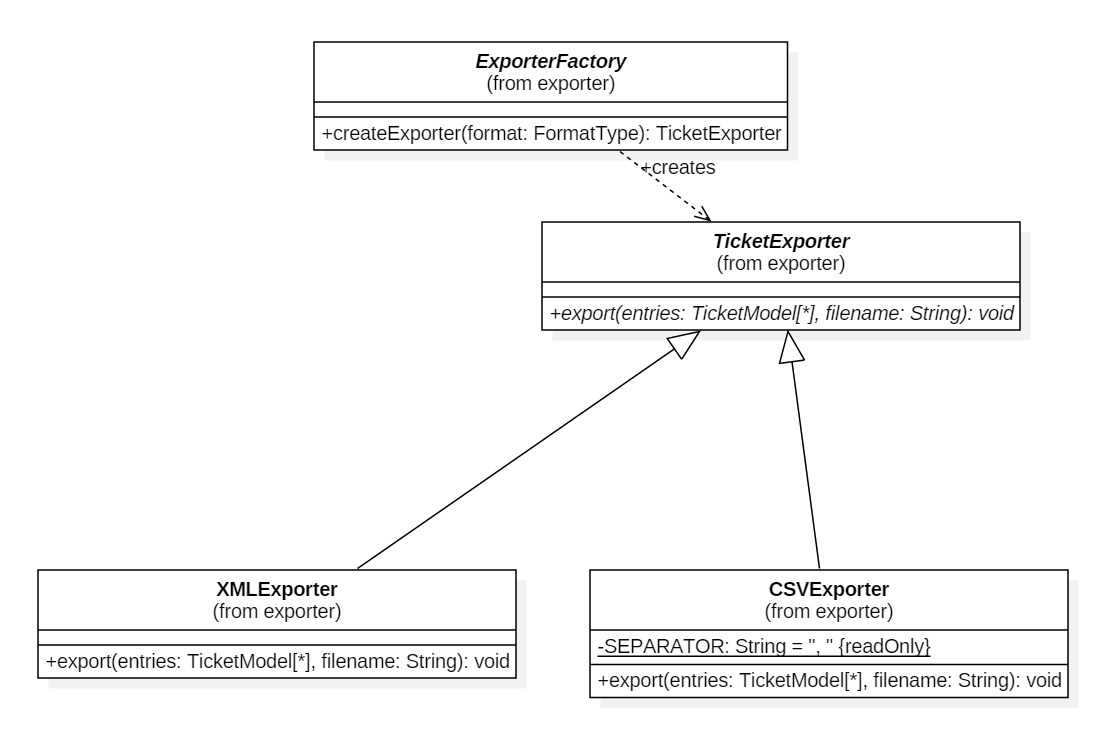
* 1. **Diagrams**
* **Package diagram:**



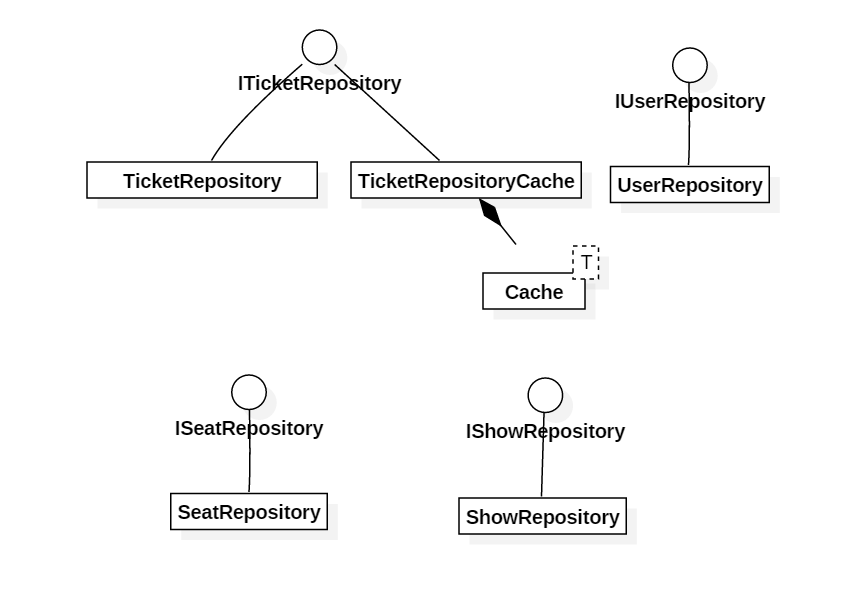
5. Class Design

**5.1 Design Patterns Description**

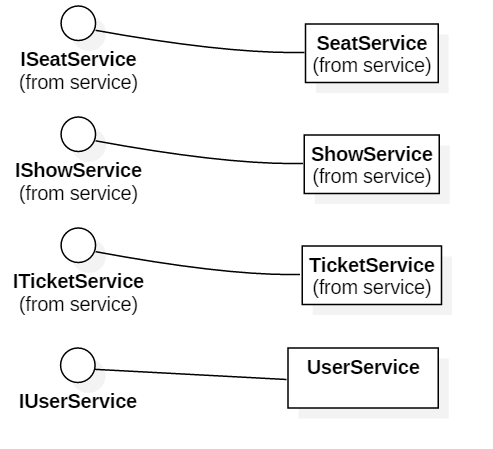
* **Singleton pattern -** a design pattern that restricts the instantiation of a class to one object. This is achieved by making the constructor of the class private and defining a public static method for getting the only instance of the class
* **Factory Method pattern –** define interface for creating objects, but subclasses decide which class to instantiate
  1. **UML Class Diagram**
* Connection factory – used for creating the connection to the database, implements singleton pattern in order to have a single connection in the whole application
* Classes in the dbmodel package, modelling the data transfer objects (object that carries data between processes), one class corresponding to each entity in the database
* Classes in the exporter package, implementing the Factory design pattern:
* The creation of the appropriate TicketExporter object is delegated to the ExporterFactory class, according to the type of the file to be exported (csv or xml), modelled with the FormatType enum.



The classes in the repository package: for each entity there is a Repository class which implements the corresponding interface and which is responsible for executing queries and getting data through DTOs

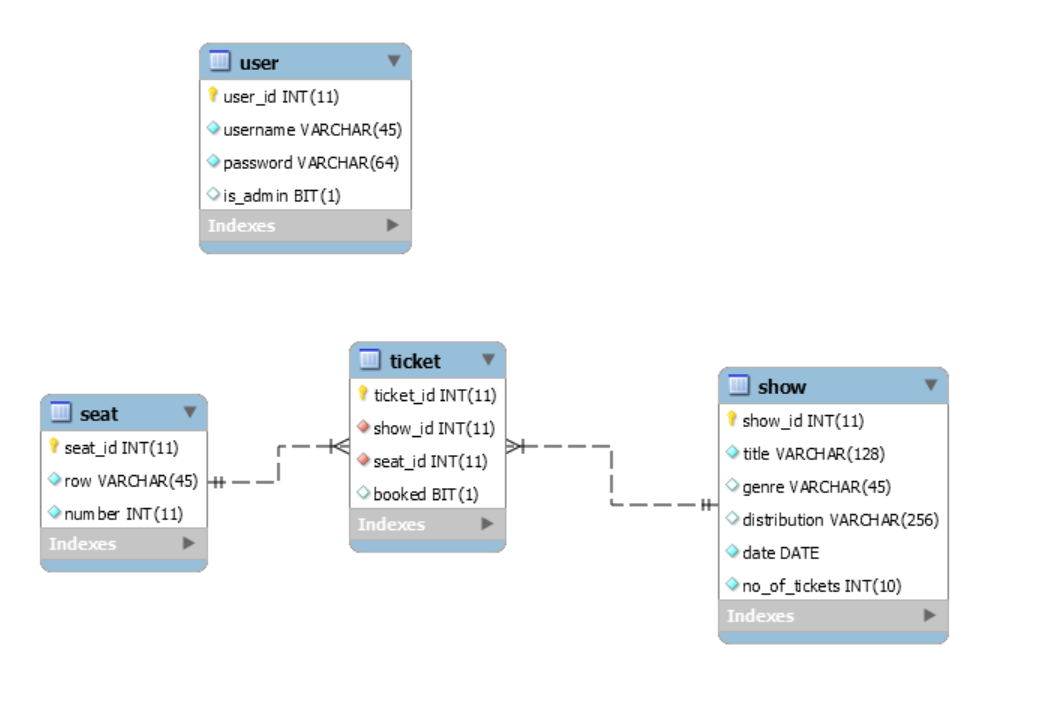


Each class in the service package implements the corresponding interface. The layer above, presentation, accesses the business layer through these interfaces. The classes in this package use the classes in the model package for modelling data.



6. Data Model

The following entities were used to model the application domain. Throughout the application, in each layer there is a class implementing an interface for the functionality associated to each model.



7. System Testing

For testing the system unit tests were designed for the following purposes:

- Testing the connection with the database

- Testing the correctness of the encryption algorithm, by comparing the results provided by the application with results obtained from an external source

- Testing whether the system notifies the cashier when he attempts to sell a ticket and there are no more tickets available for that show. In order to test only the correctness of the desired functionality, the access to the database is simulated by building a collection of DTO objects that mock the data stored in the database.

- Testing the functions of the ShowRepository class

8. Bibliography

* <https://sourcemaking.com/design_patterns/factory_method>
* https://en.wikipedia.org/wiki/Factory\_method\_pattern
* <https://docs.oracle.com/javafx/2/ui_controls/table-view.htm>
* https://www.mkyong.com/java/how-to-create-xml-file-in-java-dom