Electric Castle

Student: Alexandru Sabou

**Group: 30239**

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

Casierii Electric Castle au nevoie de o aplicatie desktop pe care sa o foloseasca pentru a vinde bilete la festival.

Festivalul are 4 tipuri de bilete:

* Ziua 1 - pret 200 RON
* Ziua 2 - pret 200 RON
* Ziua 3 - pret 200 RON
* Toate zilele - pret 400 RON

Deoarece spatiul e limitat, in fiecare zi pot fi maxim MAX\_CAPACITY oameni la festival si va trebui restrictionata vanzarea biletelor de acest aspect. Biletele din Ziua x + Biletele Toate zilele < MAX\_CAPACITY

Aplicatia va avea 2 tipuri de utilizatori: casier si administrator. (LOGIN)

Nu e nevoie de partea de User Register, Adminu va adauga casieri.

Administratorul va putea:

* CRUD pe casieri
* Modificarea numarului MAX\_CAPACITY
* Raport cu nr bilete vandute per casier
* [Exerictiu laborator] Raport cu incasarile pe zi + totale bazate pe biletele vandute

Casierul va putea:

* Vanzare de bilete - [BONUS] creat fisier .txt pe disk pentru fiecare bilet, cu detalile biletului.
* Vizionarea istoricului biletelor vandute de catre el ordonate dupa timp+data (ultimele bilete vandute fiind primele)

**Detalii tehnice**

* Se va folosi patternul arhitectural: **Layers**
* Aplicatie desktop folosind JAVA/C#
  + Frameworkuri sugerate:
    - JAVA - Swing
    - C# - Windows Presentation Foundation (WPF)
* Datele vor fi stocate intr-o baza de date
  + MySql/ Sql Server Express/ Oracle Express
* Accesul la baza de date se va face folosind
  + Un pattern de acces de date:
    - <https://martinfowler.com/eaaCatalog/dataMapper.html>
    - <https://martinfowler.com/eaaCatalog/tableDataGateway.html>
  + Frameworks/Libraries
    - JDBC - Java
    - Ado.Net - C#
* Nu trebuie pus accent pe aspectul estetic al interfetei. E important doar sa functioneze bine

# Functional Requirements

* Autentificare utilizatori
* Administratorul va putea:
* CRUD pe casieri
* Modificarea numarului MAX\_CAPACITY
* Raport cu nr bilete vandute per casier
* [Exerictiu laborator] Raport cu incasarile pe zi + totale bazate pe biletele vandute
* Casierul va putea:
* Vanzare de bilete - [BONUS] creat fisier .txt pe disk pentru fiecare bilet, cu detalile biletului.
* Vizionarea istoricului biletelor vandute de catre el ordonate dupa timp+data (ultimele bilete vandute fiind primele)

# Non-functional Requirements

Availability

* + In functie de “uptime”-ul bazei de date, deoarece aplicatia nu poate rula fara o baza de date
  + Deoarece dorim ca aplicatia sa fie una de tip “high availability” baza de date are un uptime de 99.999%

Performance

* Toate ferestrele aplicatiei au un timp de raspuns < 500ms

Security

* Autentificare si autorizatie – autentificarea este posibila prin operatia de logare, iar autorizatia este posibila datorita celor doua tipuri de conturi.
* Prevenirea SQL Injection – este realizata cu ajutorul clasei PreparedStatement.

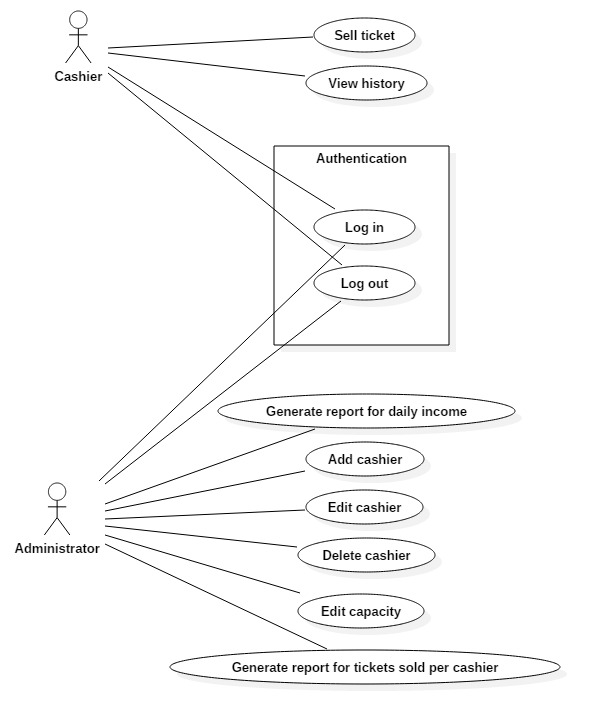
Testability

* Easy to automate – este usor de testat datorita pattern-ului architectural Layers.

Usability

* Aplicatia este foarte usor de folosit avand o interfata intuitiva.Utilizatorul nu trebuie sa se gandeasca la ce repercusiuni ar putea avea actiunea pe care o va face, deoarece sistemul nu ii va da voie sa faca actiuni daunatoare aplicatiei.

2. Use-Case Model



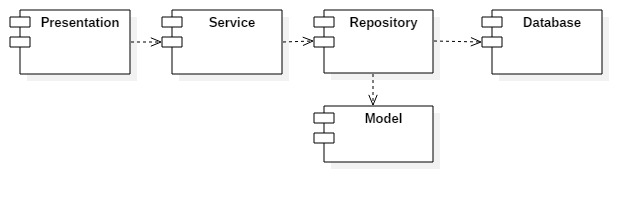
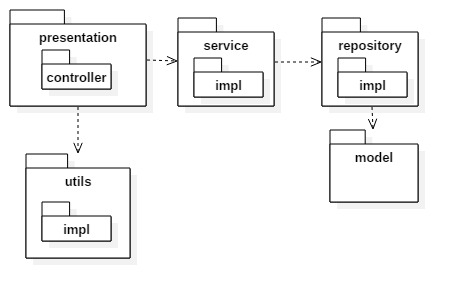
3. System Architectural Design

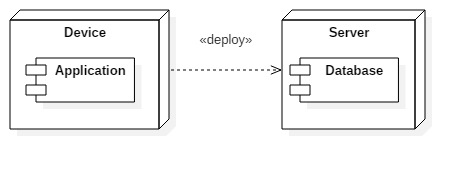
**3.1 Architectural Pattern Description**

Arhitectura Layers este cel mai cunoscut sablon arhitectural, cunoscut si sub numele de “n-tier architecture pattern”. Este sablonul standard pentru majoritatea aplicatiilor Java, astfel fiind bine cunoscut de majoritatea arhitectilor si dezvoltatorilor de aplicatii. Acest sablon nu impune un anumit numar de nivele, insa de regula exista 4 nivele standard:

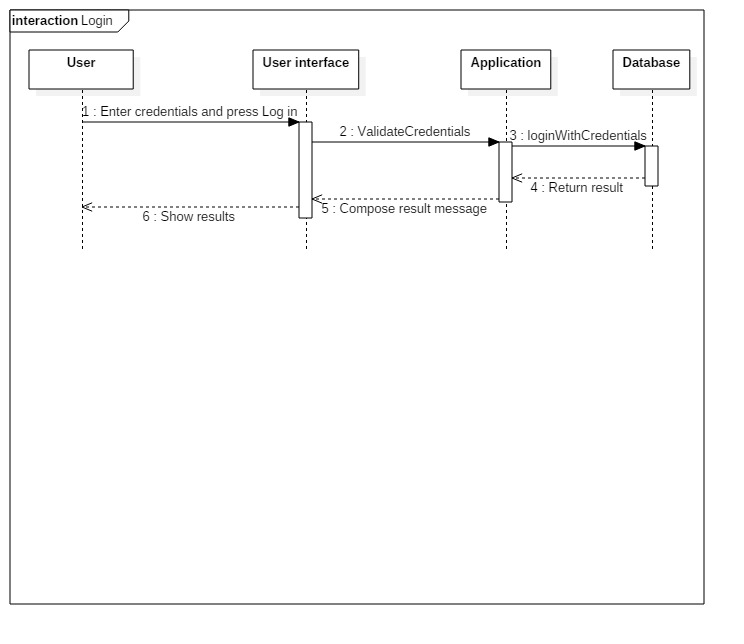
* Presentation Layer
* Business Logic Layer
* Data Access Layer
* Database Layer

**3.2 Diagrams**





4. UML Sequence Diagrams

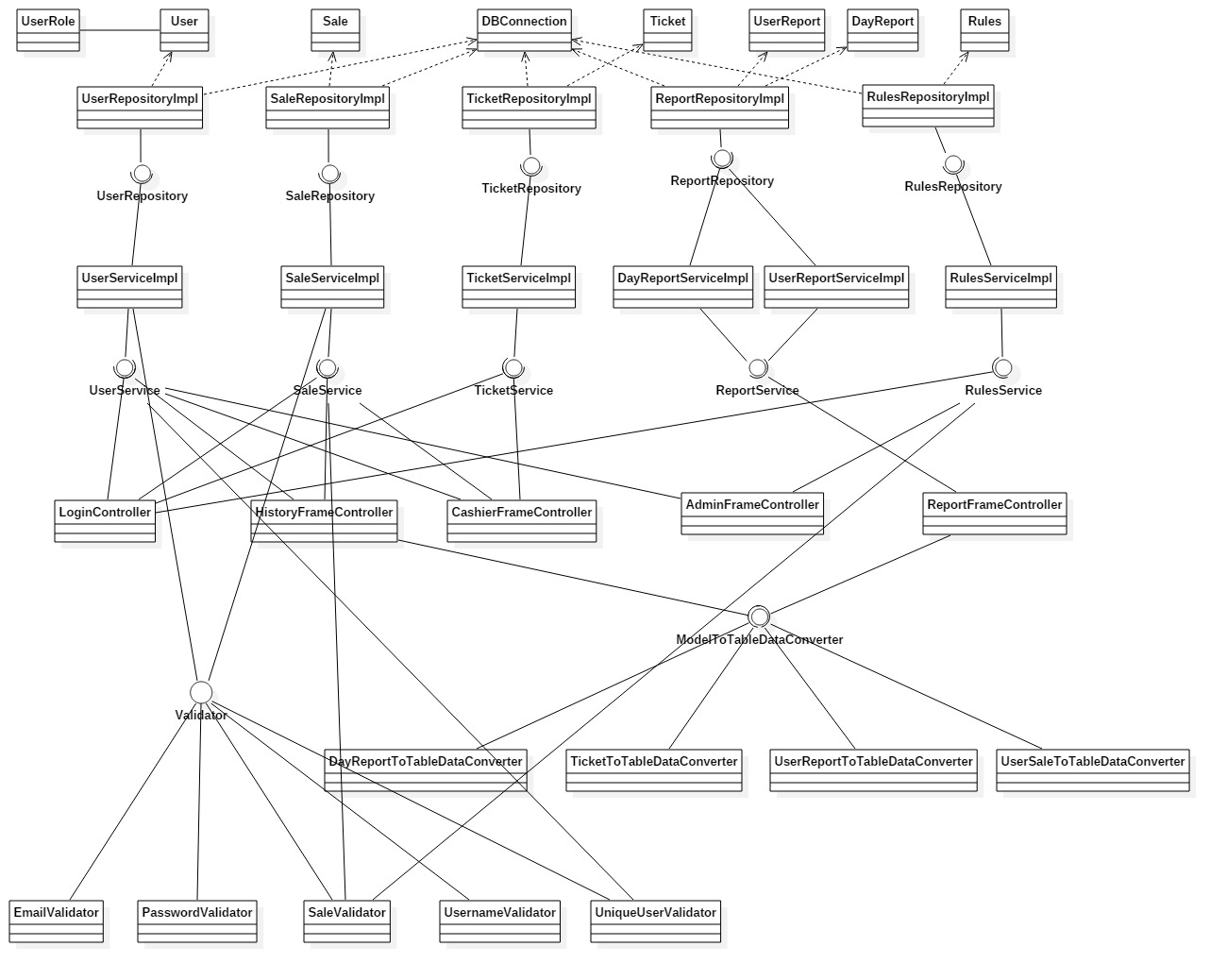
**

5. Class Design

**5.1 Design Patterns Description**

* Factory pattern – folosit pentru crearea obiectelor fara sa dezvalui clientului cum s-au creat obiectele respective si se refera obiectul nou creat folosind o interfata comuna.
* Dependency injection – o clasa accepta obiectele de care are nevoie printr-un “injector” in loc sa creeze obiectele direct.
* Singleton – se asigura ca o clasa are doar o instanta si ofera acces global catre aceasta.

**5.2 UML Class Diagram**

**

6. Data Model

User

* Id
* Username
* Password
* Email
* UserRole – rolul utilizatorului: Cashier/Admin

Ticket

* Id
* Name – tipul biletului
* Price

Sale

* Id
* User – casierul care vinde biletul
* Ticket – tipul de bilet vandut
* Date – data si ora la care s-a vandut biletul

DayReport

* Date – data raportului
* Total – incasarile pe data respectiva

UserReport

* User – utilizatorul pentru care s-a realizat raportul
* Tickets – numarul de bilete vandute

7. System Testing

Testare manuala, dupa fiecare implementare s-a realizat o testare.

8. Bibliography

<https://en.wikipedia.org/wiki/Software_design_pattern>

Diferite pagini de pe <https://stackoverflow.com/>