Assignment 2

Analysis and Design Document

Student: Dulău Marius-Cristian

**Group: 30235**

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

Opera Națională din Cluj are nevoie de o aplicație pentru gestiunea spectacolelor și a biletelor care se pun în vânzare pentru spectacole.

Accesul la aplicație se face pe baza de login. Vor exista două tipuri de utilizatori: casieri și administratori.

Administratorii vor putea să gestioneze lista de spectacole (operații CRUD). Pentru fiecare spectacol se vor salva în baza de date: genul (opera, opereta, balet), titlul, regia, distribuția, data premierei, număr de bilete care se pun ân vânzare. Administratorii vor putea să creeze conturi pentru casieri (nume, username, parolă).

Casierul va putea după login să adauge în sistem bilete care s-au cumpărat pentru un anumit spectacol. Fiecare bilet va conține: spectacolul, rând, număr. Un casier va putea să vadă lista cu toate biletele adăugate pentru un anumit spectacol. Periodic, casierii pot exporta lista de bilete vândute pentru un anumit spectacol (în format csv sau xml).

# Functional Requirements

Administratorul va putea efectua următoarele operații:

* CRUD pe lista spectacolelor;
* creare conturi casieri.

Casierii vor putea efectua următoarele operații:

* adăugare bilete în sistem;
* vizualizare cu bilete vândute;
* exportare listă bilete.

# Non-functional Requirements

Sistemul va verifica sa nu se depășească numărul de bilete puse in vânzare pentru acel spectacol si va verifica sa nu mai fi fost bilete vândute pe același rând si număr. Pentru a implementa funcționalitatea de export cu ajutorul design pattern-ul Strategy. La runtime, utilizatorul poate alege formatul dorit pentru export. Se folosește Factory pattern pentru a crea obiectul specific cu care se instanțiază contextul din Strategy pattern.

Constrângeri:

* Datele vor fi salvate intr-o baza de date;
* Se va folosi pattern-ul arhitectural MVC la proiectarea si organizarea aplicatiei;
* Parolele vor fi salvate criptat in baza de date;
* Se va folosi o biblioteca ORM.

2. Use-Case Model

**Use case:** Management bilete

**Level:** user-goal level

**Primary actor:** Casier

**Main success scenario:** Casierul accesează aplicația prin intermediul contului creat de administrator. El introduce datele contului în fereastra de login. Dacă datele sunt corecte, va putea adăuga bilete în sistem.

**Extensions:** După autentificare casierul poate vizualiza biletele vândute pentru fiecare spectacol. Când vizualizează biletele pentru un spectacol nu are dreptul de a le modifica datele. Periodic, un casier poate exporta lista de bilete vândute pentru un anumit spectacol (în format csv sau xml).

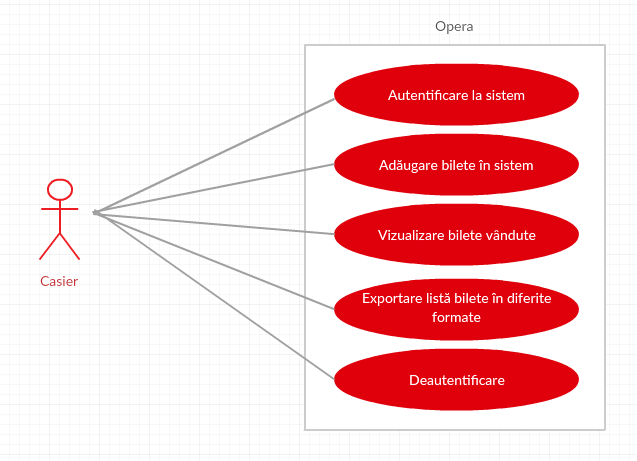


Figura 1. Use case casier

**Use case:** Administrare aplicație

**Level:** user-goal level

**Primary actor:** Administrator

**Main success scenario:** administratorul accesează aplicația prin intermediul unui cont standard. El introduce datele contului în fereastra de login. Dacă datele sunt corecte, va putea efectua modificări asupra conținutului bazei de date.

**Extensions:** După autentificare administratorul poate vizualiza spectacolele, distribuția pentru un spectacol și casierii. De asemenea, poate să modifice datele spectacolelor, a actorilor și a casierilor, să adauge spectacole noi, să adauge/modifice conturile casierilor, să adauge/modifice distribuția la un spectacol.

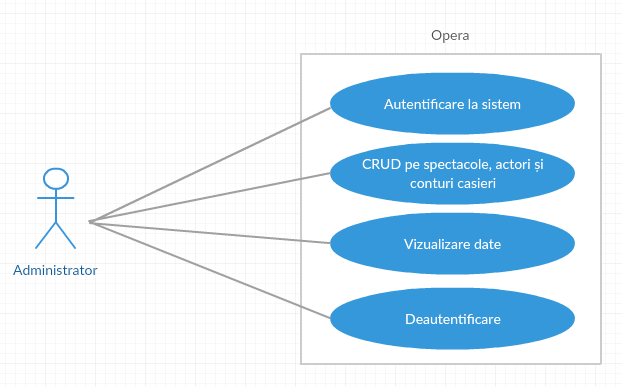


Figura 2. Use case administrator

3. System Architectural Design

**3.1 Architectural Pattern Description**

MVC (Model-View-Controller) este un pattern architectural folosit pentru a dezvolta interfețe utilizator care impart aplicația în 3 părți. Design-ul MVC decuplează cele 3 componente permițând reutilizarea codului și dezvoltare paralelă. Pattern-ul MVC este cel mai des întâlnit în aplicațiile web.

Cele 3 componente ale pattern-ului sunt modelul, view-ul și controller-ul. Modelul încapsulează datele și funcționalitatea de bază. View-ul afișează informații utilizatorului. View obține datele de la model. Fiecare view are asociat un controller. Controller-ul se ocupă de preluarea datelor de la utilizator și transmiterea acestora la model și de la model înapoi în interfața grafică.



Figura 3. Exemplu arhitectură MVC

Consrângeri:

* aceleași informații sunt prezentate diferit în ferestre diferite;
* afișarea și comportamentul aplicației trebuie să reflecte manipularea datelor;
* modificările interfeței cu utilizatorul ar trebui să fie ușor și chiar posibile la runtime;
* portarea interfeței cu utilizatorul nu ar trebui să afecteze codul din interiorul aplicației.

Modelul încapsulează și manipulează datele care urmează a fi afișate în interfață. Modelul nu are idee cum vor fi afișate datele, nici nu interacțiunează cu utilizatorului.

View-ul este o redare vizuală sub formă grafică sau text a datelor conținute în model. View-ul este independent de model. Când se modifică datele din model, toate interfețele grafice (view) sunt actualizate.

Controlerul manipulează datele introduse de utilizator. De multe ori, un controler “ascultă” evenimentele de la mouse și de la tastură.

Avantaje:

* Dezvoltare paralelă;
* Coeziune înaltă
* Cuplare redusă
* Ușor de modificat

Dezavantajul este reprezentat prin navigarea prin codul aplicației.

În Strategy pattern, comportamentul unei clase poate fi modificat la runtime. Acest design pattern este un pattern comportamental. Pattern-ul înglobează obiecte care reprezintă diferite strategii și un obiect context al cărui comportament variază în funcție de obiectul său de strategie. Se creează o interfață de strategie care definește o acțiune și clase de strategie concrete care implementeze interfața Strategy.

Factory method are integrat design pattern-ul Strategy.

**3.2 Diagrams**

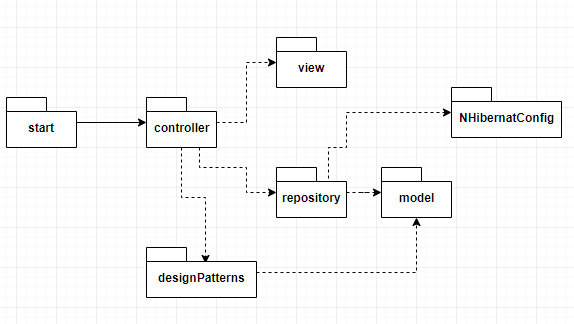


Figura 4. Diagramă de pachete

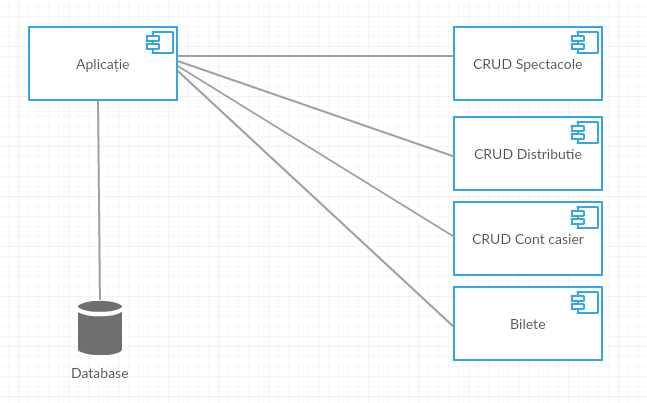


Figura 5. Diagrama de componente

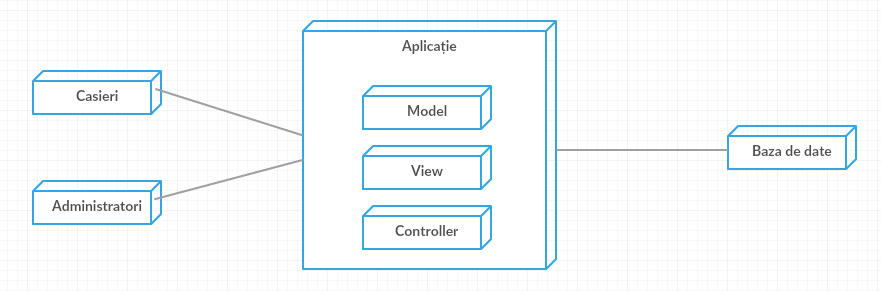


Figura 6. Diagrama de deployment

4. UML Sequence Diagrams

Un scenario relevant este adăugarea de bilete în sistem. Casierul trebuie să se autentifice. După autentificare, el trebuie să selecteze un loc disponibil în sală.

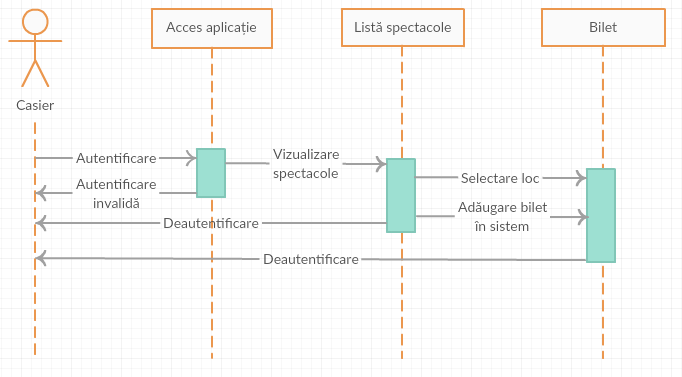


Figura 7. Diagramă de secvență

5. Class Design

**5.1 Design Patterns Description**

Design pattern-urile utilizate în cadrul acestui proiect sunt MVC, Strategy și Factory method. MVC este un pattern arhitectural folosit în separarea interfeței grafice de modelele cu care se lucrează. Pentru a trimite date din interfață la model se folosește un controler. Strategy este un design pattern în care se crează o interfață cu o metodă care definește o acțiune, iar apoi clasele ce implementează această interfață descriu acțiunea concret. Factory method este un pattern care conține o clasă cu o metodă ce returnează un obiect de un anumit tip. Acest pattern are integrat design pattern-ul Strategy.

**5.2 UML Class Diagram**

Pachetul model are următoarele clase: User, Admin, Casier, Bilet, Spectacol, Distributie, DistributieOp și DistributieBalet. Clasele din acest pachet reprezintă entitățile principale în care sunt stocate datele. Pachetul repository conține 2 clase: UserRepository și SpectacolRepositroy. Aceste clase folosesc metodele din NHibernate pentru CRUD, preluare blocuri de date și trimiterea lor la controler pentru a le afișa în interfața grafică.

Pachetul view conține clase ce contruiesc interfața cu utilizatorul. Pachetul controller conține clase pentru fiecare clasă din pachetul view pentru a prelua și a apela metode de afișare a datelor. Un pachet important este designPattern în care se află structura la Factory method și la Strategy.

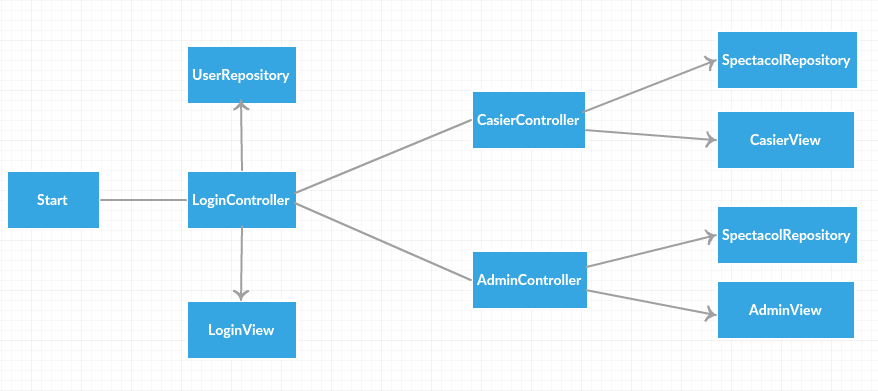


Figura 8. Diagrama UML proiect

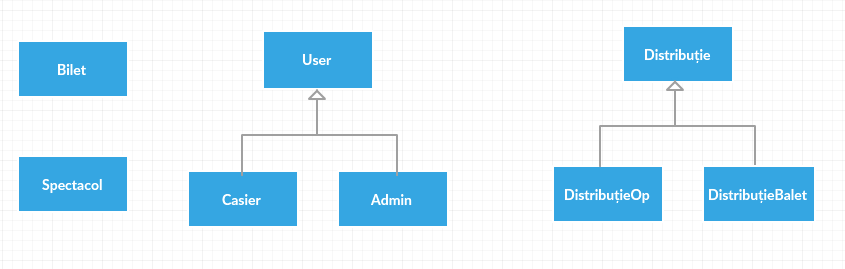


Figura 9. Conținutul pachetului model

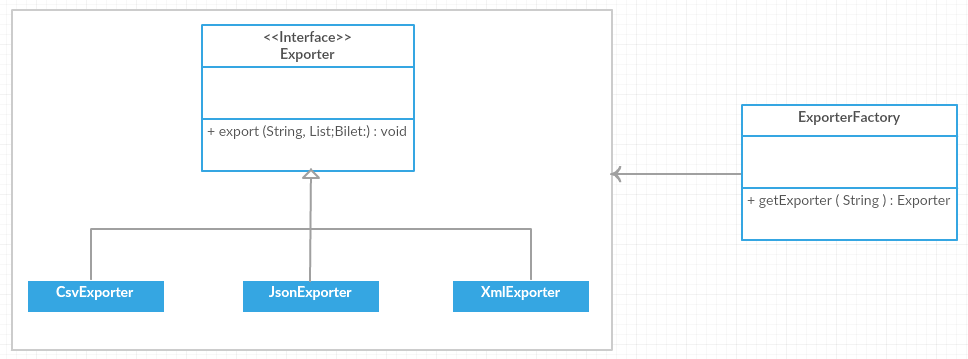


Figura 10. Diagrama UML pachet designPatterns

6. Data Model

Majoritatea claselor din pachetul model se mapează pe tabelul pe care îl reprezintă. Pentru a eficientiza operațiile efectuate pe baza de date din interiorul aplicației s-a utilizat biblioteca NHibernate. Cu această bibliotecă s-a mapat o clasă pe un tabel în mai pușine linii de cod față de metoda clasică, în care se construia un string, care apoi era executat pe structura bazei de date. O clasă are ca variabile de instanță câmpurile tabelului pe care îl reprezintă. Pentru a avea acces la facilitățile bibliotecii NHibernate, a trebuit creat un fișier de configurare al acestuia. Pentru ca NHibernate să poată mapa clasele pe tabele s-au creat fișiere .xml care conțin maparea clasei pe structura tabelului pe care îl reprezintă. Un alt aspect important este că o clasă trabbuie să aibă proprietăți (gettere și settere) virtuale (pot fi suprascrise).

În continuare, vom exemplifica modelul de date pe spectacol.

Clasa Spectacol din pachetul model are 6 variabile de instanță, iar tabela spectacol din baza de date are 6 câmpuri.

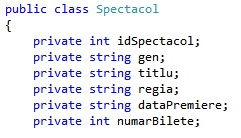


Figura 11. Variabile instanță clasa Spectacol

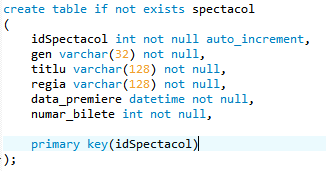


Figura 12. Tabelul corespunzător clasei Spectacol

Fișerul .xml prin care clasa se mapează pe tabel arată în felul următor:



Figura 13. Fișierul .xml pentru maparea clasei Spectacol pe tabela din BD folosind NHibernate

Fiecare clasă se mapează pe un tabel din baza de date cu ajutorul framework-ului NHibernate.

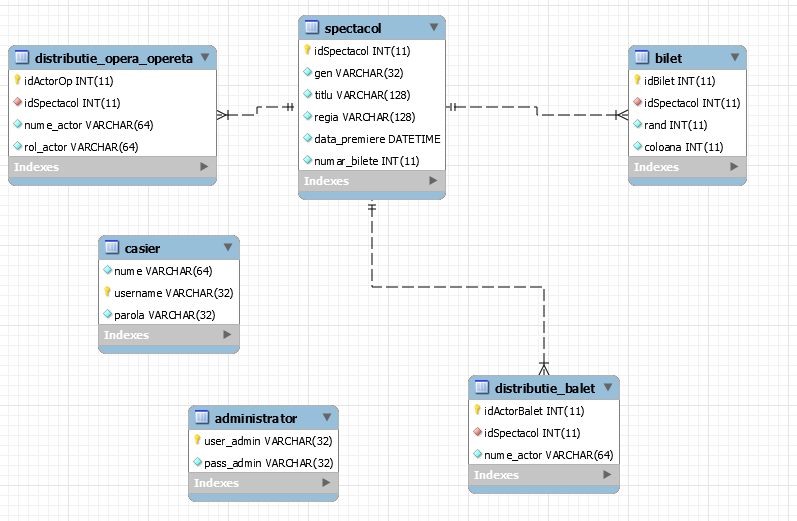


Figura 14. Diagrama bazei de date

7. System Testing

Pentru acest assignment s-a ales metoda test unit. S-au efectuat 10 teste care să verifice consistență datelor. Descrierea testelor este următoarea:

Test 1: creare cont casier

Test 2: modificare cont casier

Test 3: ștergere cont casier

Test 4: creare spectacol

Test 5: modificare date spectacol

Test 6: creare actor pentru spectacolul de la testul 4

Test 7: modificare date actor

Test 8: ștergere actor

Test 9: adăugare bilet pentru spectacolul de la testul 4

Test 10: ștergere spectacol

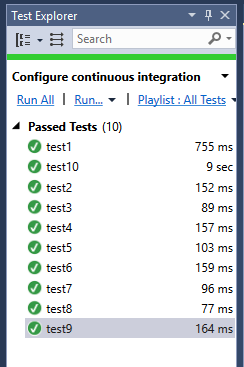


Figura 15. Rezultatele testelor

Proiectul fiind scris în limbajul C# s-a utilizat NUnit. Structura unui test arată în felul următor:

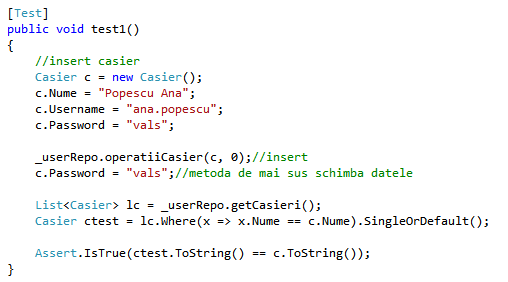


Figura 16. Metodă de test

8. Bibliography

<http://users.utcluj.ro/~dinso/PS2018/Lectures/> - curs 3

<https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller>

<https://www.tutorialspoint.com/design_pattern/strategy_pattern.htm>

<https://www.tutorialspoint.com/design_pattern/factory_pattern.htm>

<http://geekswithblogs.net/pariam/archive/2006/07/26/86352.aspx>

<http://www.imkrisna.com/blog/2010/05/nhibernate-for-mysql-c-source-code/>

<http://www.kode-blog.com/nhibernate-tutorial-csharp>

<https://www.youtube.com/watch?v=FkmFI736wMU>

<https://social.msdn.microsoft.com/Forums/vstudio/en-US/af41549d-cfe3-4a54-8c23-e6d6db86d57e/how-to-convert-list-object-to-xml-?forum=csharpgeneral>

<https://stackoverflow.com/questions/1890093/converting-a-generic-list-to-a-csv-string>

<https://stackoverflow.com/questions/16921652/how-to-write-a-json-file-in-c>

<https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-nunit>