



Universitatea TRANSILVANIA din Brașov
Facultatea de Matematică și Informatică
Specializarea Informatică

Lucrare de licență

Autor: Ursta Andrei Alexandru
Coordonator științific: Conf. univ. dr. Deaconu Adrian

Brașov

2012



Universitatea TRANSILVANIA din Braşov
Facultatea de Matematică şi Informatică
Specializarea Informatică

Aplicație cu baze de date de gestionare a unei firme

Autor: Ursta Andrei Alexandru
Coordonator științific: Conf. univ. dr. Deaconu Adrian

Braşov

2012

Cuprins

1. Introducere	1
1.1. Motivație și scop	1
1.2. Scurtă prezentare	2
2. Baze de date relaționale	3
2.1. Normalizarea unei baze de date relaționale	3
2.2. Forme normale	4
2.3. Normalitatea bazei de date a aplicației	8
3. Tehnologii folosite în realizarea aplicației	14
3.1. Utilizare tehnologiilor	14
3.2. SQL server	15
3.3. Limbajul C#	21
3.4. Telerik report	30
4. Prezentarea aplicației	31
4.1. Arhitectura aplicației	31
4.2. Introducerea de date	37
4.3. Adăugarea de facturi	46
4.4. Vizualizarea stocurilor	54
4.5. Efectuarea plăților	61
4.6. Generarea rapoartelor	66
4.7. Utilizatori. Drepturi. Roluri	74
5. Bibliografie	81

Capitolul 1 - Introducere

1.1. Motivație și scop

Motivația alegerii acestei aplicații este susținută de plăcerea de lucra cu limbajul de programare C# și cu baze de date. Ca o părere personală, consider că aceste două tehnologii reprezentate de SQL Server și limbajul de programare C# se află într-o armonie, iar suportul tehnologic oferit de Microsoft, mai exact Visual Studio și SQL Server Management Studio, este admirabil.

O altă motivație a alegerii acestei teme reprezintă oportunitatea de a colabora cu cadre didactice, o părere personală este că experiența acumulată de-a lungul perioadei de muncă pentru licență este remarcabilă și conduce la schimbarea modului de lucru, la un mod mai apropiat de lumea reală.

Scopul aplicației de gestiune este să asigure reflectarea stocurilor pe baza facturilor de intrare și de ieșire, de a stoca toate informațiile cu privire la operațiile efectuate, de a pune la dispoziție utilizatorului toate informațiile necesare pentru a avea o viziune clară asupra stocurilor.

1.2. Scurtă prezentare

Aplicație cu baze de date de gestionare a unei firme - este o aplicație desktop realizată în platforma Microsoft .NET 4.0 folosind limbajul de programare C#. Este structurată pe trei proiecte: unul de business (care conține logica aplicației), unul vizual principal (care oferă interfețele pentru utilizator) și încă unul vizual secundar, care oferă o interfață de autentificare realizată în WPF (Windows Presentation Foundation).

Logica aplicației se bazează pe trei componente mari: stocurile, facturile și plățile. Aceste componente cuprind și alte clase care sunt importante (de ex: Firmele), dar în esență, ele reprezintă fundația aplicației. Toată logica aplicației este apelată din proiectul vizual principal, existând astfel o separare a logicii de interfață.

Aplicația constă la început dintr-o etapă de autentificare, care are rolul de a limita accesul la aplicație doar pentru utilizatorii care dețin date de autentificare. Confirmarea autentificării are ca rezultat încărcarea aplicației principale.

Capitolul 1 - Introducere

Structura aplicației este de tip tree cu noduri și copii. Fiecare nod reprezintă un drept al aplicației, drepturile depind de utilizator.

O funcționalitate a aplicației reprezintă introducerea de date. Pentru început este necesară introducerea de date simple, cum ar fi delegații, produsele, băncile. După ce aplicația este dotată cu suficiente date simple se poate începe adăugarea de date mai complexe, cum ar fi firmele. Firmele au un rol important, deoarece fiecare factură, fie ea de intrare sau de ieșire, se eliberează pe o anumită firmă.

Adăugarea facturilor reprezintă următoarea funcționalitate importantă a aplicației. Facturile sunt cea mai importantă parte a aplicației, de adăugarea facturilor de intrare și de ieșire depinde toată aplicația. Facturile de intrare afectează stocurile, în sensul că toate produsele adăugate pe o factură ajung în stocuri noi, iar facturile de ieșire golesc aceste stocuri. Stocurile, deși nu pot fi afectate direct, reprezintă o componentă importantă, pentru că vizualizarea stocurilor oferă o perspectivă asupra situației produselor.

Rostul facturilor este acela de a obține remunerație de pe urma lor, aceasta este funcționalitatea următoarei componente a aplicației, și anume: Plățile.

Plățile se împart în două categorii: plățile facturilor de intrare și plățile facturilor de ieșire. Plata unei facturi de ieșire, în realitate înseamnă încasarea unei sume de bani pe produsele vândute pe o anumită factură, iar plata unei facturi de intrare înseamnă plățirea produselor de pe o anumită factură. Aplicația oferă acest suport de plătire a facturilor în două moduri: unul manual, adică se poate alege care factură să se plătească și ce sumă să se plătească și unul automat, în care facturile se plătesc, în limita sumei, de la cea mai veche la cea mai nouă.

Aplicația oferă utilizatorului posibilitatea de a tipări, prin intermediul rapoartelor, situații clare, palpabile, a stocurilor, a firmelor care încă mai au datorii. Rapoartele se generează fie pe produse, fie pe firme sau, cum ar fi în cazul stocurilor, fără parametrii.

Ultima funcționalitate, dar una importantă, este reprezentată de administrarea aplicației, care presupune crearea de roluri, adăugarea de drepturi, adăugare de utilizatori, precum și modificarea lor. Din partea de administrare se pot schimba și informații legate de firma utilizatoare o aplicației, precum și schimbarea parolei administratorului.

Capitolul 2

Baze de date relaționale

2.1. Normalizarea unei baze de date relaționale

Pentru a proiecta o bază de date relațională trebuie să avem definită o schemă a acesteia. Schema este formată din relațiile bazei de date. Se determină domeniile pe care vor fi definite relațiile, modul lor de grupare pentru delimitarea fiecărei relații, legăturile logice din interiorul fiecărei relații și dintre relații. Având o schemă a unei relații se definește numele acesteia și mulțimea atributelor acelei relații. Dintr-o schemă a unei relații se definește modul de grupare a datelor și regulile ce vor guverna relația, definind restricțiile datelor și a legăturilor logice dintre date. Legăturile logice dintre relații se definesc prin intermediul domeniilor compatibile care determină attributele comune ale relațiilor. Modul de grupare a domeniilor depinde de complexitatea legăturilor care se definesc între relații. Relațiile ce cuprind prea multe informații specifice pot conduce la probleme redundanță. De exemplu, să presupunem că folosim relația următoare pentru gestionarea vânzărilor unor produse:

Faktură(cod_faktură, număr, serie, data_fakturării, nume_produș, cantitate, preț)

Astfel, pentru fiecare produs care se vinde se generează aceleași date din nou pentru număr, serie și data_fakturării. Deci, relația definită nu este potrivită, este o relație ce provoacă redundanță și anomalii la operații de actualizare. De exemplu dacă vindem un produs în cantități separate pe aceeași factură, la o operație de actualizare nu putem distinge cele două înregistrări. Se poate reduce redundanța și anomaliile generate de aceasta prin definirea unei relații noi și reducerea datelor din prima relație:

Faktură(cod_faktură, număr, serie, data_fakturării, cod_produș, cantitate)

Produș(cod_produș, nume_produș, preț)

Capitolul 2 – Baze de date relaționale

Constatăm că redundanța a fost înlăturată din relația Factură prin definirea relației Produs, care conține informații legate de produs. Această operație a condus la un nou inconvenient, deoarece dacă dorim aflarea numelor tuturor produselor care au fost vândute pe o anumită factură trebuie să cuplăm cele două relații prin intermediul atributului `cod_produs`. Operația de cuplare a două relații (join) este operație destul de costisitoare deoarece presupune realizarea produsului cartezian a celor două relații. În teorie, schema bazei de date se realizează pornind de la identificarea tuturor grupurilor de entități și a proprietăților lor. În practică, determinarea atributelor și proprietăților acestora poate fi de multe ori dificilă. În plus, în definirea unei scheme pentru o bază de date relațională trebuie să ținem cost și de faptul că un număr mic de relații poate provoca redundanță, dar și că un număr prea mare de relații crește costul cererilor către baza de date, deoarece aceste cereri se realizează pe baza operațiilor de cuplare între relații.

În concluzie observăm că apare necesitatea definirii unui procedeu optimizare a schemei bazei de date relaționale, pentru a înlătura redundanțele și anomaliile care apar odată cu acestea, dar și pentru definirea legăturilor care există între grupuri de entități. Acest procedeu se numește *Normalizarea relațiilor din baza de date relațională* și presupune pornirea de la o schemă a bazei de date determinată empiric, după care se aplică o serie de transformări succesive asupra schemelor care o alcătuiesc.

Normalizarea se bazează pe formalizarea legăturilor logice (numite dependențe), ce există între atributele unei relații și pe analiza acestora. Dependențele care se definesc între atributele unei relații au fost împărțite în trei categorii:

- dependențe funcționale;
- dependențe multi-valoare;
- dependențe de uniune;

2.2. Forme normale

Existența dependențelor funcționale este naturală. Puteam lua spre exemplu dependența fiecărui atribut față de cheia primară a relației. Dependențele reprezintă informații suplimentare asociate relației, informații ce nu pot fi demonstrate, dar care pot fi verificate

Capitolul 2 – Baze de date relaționale

prin confruntarea cu realitatea. Dependențele definite pentru schema unei relații pot fi considerate constrângeri de integritate a datelor, iar implementarea lor în baza de date se face prin normalizarea relației respective.

Procesul prin care se ajunge la o schemă optimă a bazei de date, din punctul de vedere al modelării legăturilor existente în cadrul relațiilor și pentru reducerea redundanței reprezintă normalizarea relațiilor din baza de date. Se începe cu identificarea atributelor relației care se normalizează și a dependențelor în care există aceste atribute, în funcție de natura dependențelor care au fost identificate se determină forma normală în care se află relația respectivă. Prin aplicarea unui set de reguli de normalizarea, specifice acelei forme normale, se transformă relația care a fost identificată într-un set de relații echivalente, care aparțin unei forme normale mai superioare. Normalizarea este un proces iterativ, fiecare dependență identificată trebuie să treacă prin aceste faze, pentru a se ajunge la ultima formă normală.

Normalizarea bazei de date presupune satisfacerea condițiilor de formă normală:

- prima formă normală (1NF – First Normal Form);
- a doua formă normală (2NF – Second Normal Form);
- a treia formă normală (3NF – Third Normal Form);

Prima formă normală exclude posibilitatea existenței grupurilor repetitive, cerând ca fiecare atribut într-o bază de date să cuprindă numai o valoare atomică. De asemenea, prima formă normală presupune și ca fiecare înregistrare să fie definită astfel încât să fie identificată în mod unic prin intermediul unei chei primare.

Exemplu de încălcare a primei forme normale:

Persoană	Filme închiriate
Daniel	Titanic, Filantropica, Pinocchio
Ovidiu	Tăcerea mieilor, Pinocchio
Ionel	Filantropica, Cenușăreasa

Tabelul 2.1. Exemplu de tabel ce încalcă prima formă normală

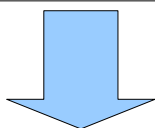
Problema cu tabela exemplificată în **Tabelul 2.1.** este că la interogarea devine foarte încurcată. De exemplu dacă dorim să aflăm care sunt persoanele ce au închiriat filmul ”Titanic” și ”Pinocchio” trebuie să parcurgem fiecare șir, să-l împărțim în subșiruri și să

Capitolul 2 – Baze de date relaționale

vedem dacă apare în vreun subșir titlurile căutate. După care trebuie selectate doar acele înregistrări ce conțin ambele titluri. O soluție de satisfacere a primei forme normale reprezintă introducerea unei noi înregistrări pentru fiecare film.

Astfel, din prima formă a tabelului obținem:

Persoană	Filme închiriate
Daniel	Titanic, Filantropica, Pinocchio
Ovidiu	Tăcerea mieilor, Pinocchio
Ionel	Filantropica, Cenușăreasa



ID	Persoană	Filme închiriate
1	Daniel	Titanic
2	Daniel	Filantropica
3	Daniel	Pinocchio
4	Ovidiu	Tăcerea mieilor
5	Ovidiu	Pinocchio
6	Ionel	Filantropica
7	Ionel	Cenușăreasa

Tabelul 2.2. Transformarea unei tabeli pentru a satisface prima formă normală

A doua formă normală cere ca toate atributele să fie dependente funcțional de totalitatea cheii primare. Dacă unul sau mai multe atribute sunt dependente funcțional numai de o parte a cheii primare, atunci ele trebuie să fie în tabele separate. Dacă o tabelă are o cheie primară formată numai din un atribut, atunci ea este automat în a doua formă normală.

Exemplu care încalcă a doua formă normală:

ID_comandă	ID_produc	nume_produc	cantitate
12	1	cereale	20
13	2	ciocolată	7

Tabelul 2.3. Tabela comandă ce încalcă a doua formă normală

În cazul tabelului "Comandă" exemplificat în **Tabelul 2.3** cheia primară este o cheie

Capitolul 2 – Baze de date relaționale

compusă, în alcătuirea ei intră atât *ID_comandă*, cât și *ID_produc*. Observăm că *nume_produc* nu depinde de *ID_comandă*, ea depinde doar de *ID_comandă*. Pentru a fi în a doua formă normală, tabela "Comandă" trebuie modificată ca în **Tabelul 2.4** de mai jos.

ID_comandă	ID_produc	nume_produc	cantitate
12	1	cereale	20
13	2	ciocolată	7

ID_produc	nume_produc
1	cereale
2	ciocolată

ID_comandă	ID_produc	cantitate
12	1	20
13	2	7

Tabelul 2.4. Transformarea unei tabele pentru a satisface a doua formă normală

A treia formă normală presupune că toate atributele non-cheie trebuie să fie mutual independente. Relația *depinde de cheie* este bazată în întregime de cheie și de nimic altceva.

Piesă	nume_producător	adresă_producător
1245	Boss	București
5487	Huawei	Craiova
9541	Boss	București

Tabela 2.5. Exemplu de tabelă ce încalcă a treia formă normală

Tabela 2.5 poate fi reorganizată ca în **Tabela 2.6** pentru a satisface a treia formă normală.

Piesă	nume_producător
1245	Boss
5487	Huawei
9541	Boss

nume_producător	adresă_producător
Boss	București
Huawei	Craiova

Tabela 2.6. Transformarea Tabelei 2.5 astfel încât să satisfacă a treia formă normală

Capitolul 2 – Baze de date relaționale

2.3. Normalitatea bazei de date a aplicației

Baza de date a aplicației, ilustrată în **Figura 2.1** este organizată astfel: avem facturi, produse, firme, stoc și plăți. Aceste patru componente trebuie organizate astfel încât să permită o evidență clară a produselor din stoc, a plăților, firmelor.

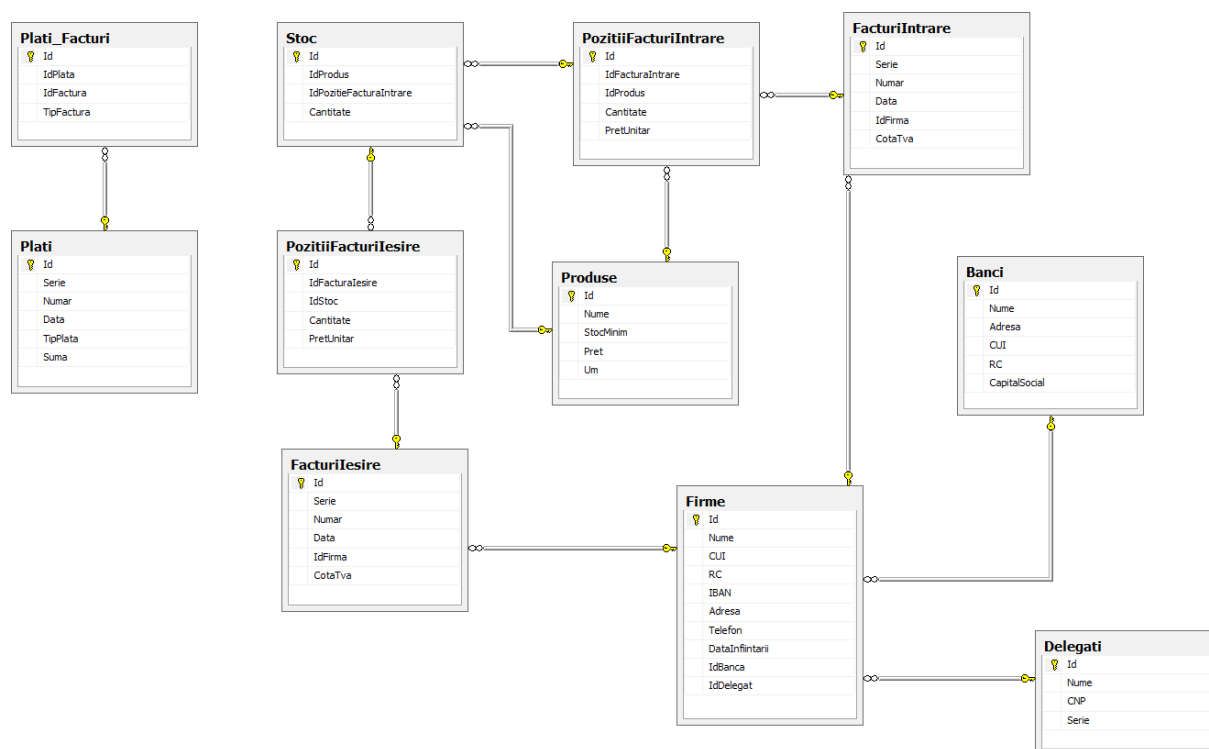


Figura 2.1. Vedere de ansamblu a bazei de date a aplicației

Această structurare a relațiilor din baza de date a condus la evidențierea a trei părți strâns legate între ele.

Prima parte are ca piesă centrală *Firmele* și dependențele sale (cum se vede în **Figura 2.2**). Tabela firmelor depinde de tabelele *Bănci* și *Delegați*, dar de ea depind și facturile.

Tabela *Delegați* din **Figura 2.2** prezintă următoarea relație:

Delegați(Id, Nume, CNP, Serie)

Această relație conține attribute care sunt legate strict de cheia sa. Nu avem grup compus, iar relația este strâns legată de cheie. Această tabelă este în a treia formă normală.

Tabela *Bănci* din **Figura 2.2** are relația:

Capitolul 2 – Baze de date relaționale

Bănci(**Id**, Nume, Adresă, CUI, RC, Capital social)

Această relație nu conține grupuri compuse, toate atributele sunt strâns legate de cheie. Nu avem anomalii, nici redundanță. Această tabelă satisface condițiile celei de-a treia formă normale.

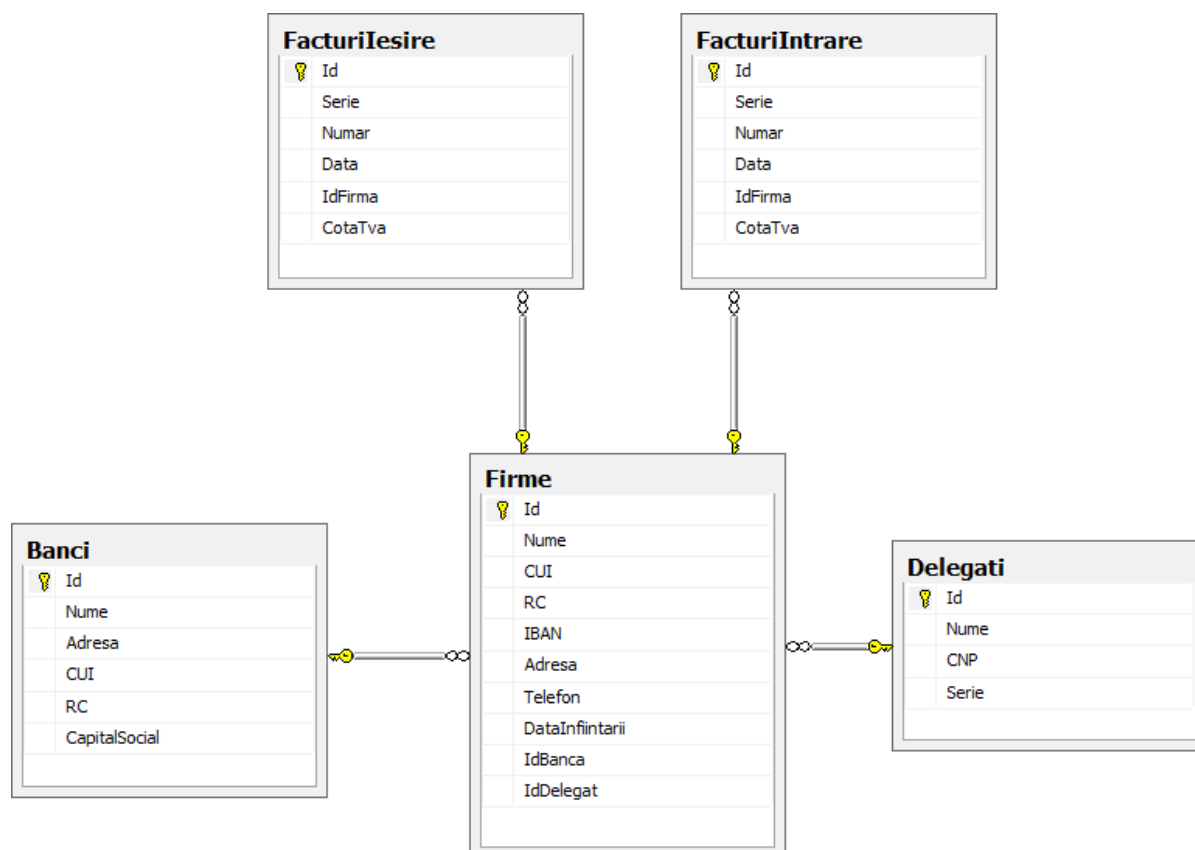


Figura 2.2. Tabela firmelor și dependențele sale

Tabela *Firme* din **Figura 2.2** este caracterizată de următoarea relație:

Firme(**Id**, Nume, CUI, RC, IBAN, Adresă, Telefon, DataÎnființării, *IdBancă*, *IdDelegat*)

Această relație nu conține grupuri compuse, toate atributele sunt strâns legate de cheie. Avem două atribute: *IdBancă* și *IdDelegat*, care sunt necesare pentru operația de cuplare cu tabelele *Bănci* și *Delegați*.

În schema reprezentată în **Figura 2.2** mai avem două tabele pentru reținerea datelor necesare facturilor de intrare. Cele două tabele se bazează pe relația:

Facturi(**Id**, Serie, Număr, Data, *IdFirmă*, CotaTva)

Această relație satisface la rândul ei condițiile formei normale trei, nu conține grupuri

Capitolul 2 – Baze de date relaționale

compuse, iar toate atributele sunt strâns legate de cheie. Atributul *IdFirmă* este un atribut necesar pentru operația de cuplare cu tabela *Firme* din **Figura 2.2**, pentru a afla datele despre firma căreia i s-a eliberat o factură.

A doua parte se concentrează asupra gestionării stocului, a produselor vândute și a produselor cumpărate. Această gestionare este facilitată de două relații importante care descriu produsele ce au ieșit din stoc, cât și cele ce au intrat în stoc. Aceste relații, care se pot vedea în **Figura 2.3**, sunt următoarele:

PozițiiFacturiIntrare(**Id**, *IdFacturăIntrare*, *IdProdus*, Cantitate, PrețUnitar)

PozițiiFacturiIeșire(**Id**, *IdFacturăIeșire*, *IdStoc*, Cantitate, PrețUnitar)

Aceste relații sunt ambele în formă normală trei, având doar două atribute ce descriu cantitatea și prețul unitar al produsul; mai conțin și câte două atribute care sunt necesare pentru operațiile de cuplare cu tabelele *FacturiIntrare*, *FacturiIeșire*, *Produse* și *Stoc*.

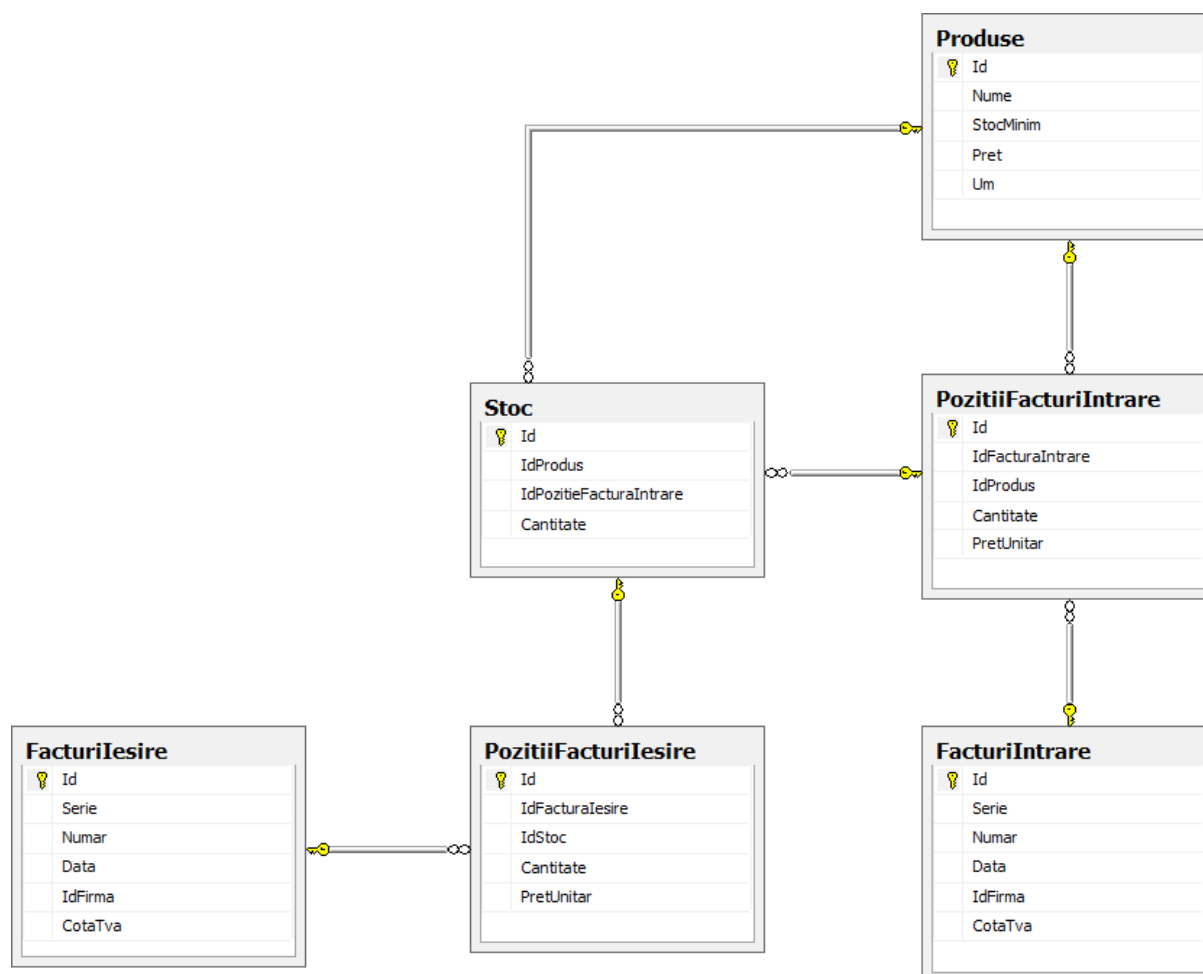


Figura 2.3. Tabelele care se ocupă cu gestionarea stocurilor

Capitolul 2 – Baze de date relaționale

În schema din **Figura 2.3** introducem două tabele noi: *Produse* și *Stoc*. Tabela produselor reține date despre produsele care se comercializează, iar tabela stocurilor ține evidența acestor produse, din interogarea sa putem deduce dacă un produs este disponibil sau nu, iar dacă este disponibil atunci aflăm și cantitatea disponibilă.

Relația tablei *Produse* este:

Produse(**Id**, Nume, StocMinim, Preț, UM)

Această relație este simplă, are patru atribute ce depind exclusiv de cheie, nu prezintă grupuri compuse, ea satisface condițiile formei normale trei.

Tabela *Stoc* este reprezentată de relația:

Stoc(**Id**, *IdProdus*, *IdPozițieFacturăIntrare*, Cantitate)

Relația nu conține decât un singur atribut, care reprezintă cantitatea produsului din stoc. Mai avem două atribute, necesare pentru operațiile de cuplare, care indică produsul și poziția din factura de intrare cu care a fost adăugat produsul respectiv în stoc.

A treia componentă a bazei de date, ilustrată în **Figura 2.4**, se are în vedere detaliile legate de achitarea facturilor. Ea aduce nou o tabelă în care se rețin datele plăților efectuate și o tabelă de legătură, care permite cuplarea plăților cu facturile.

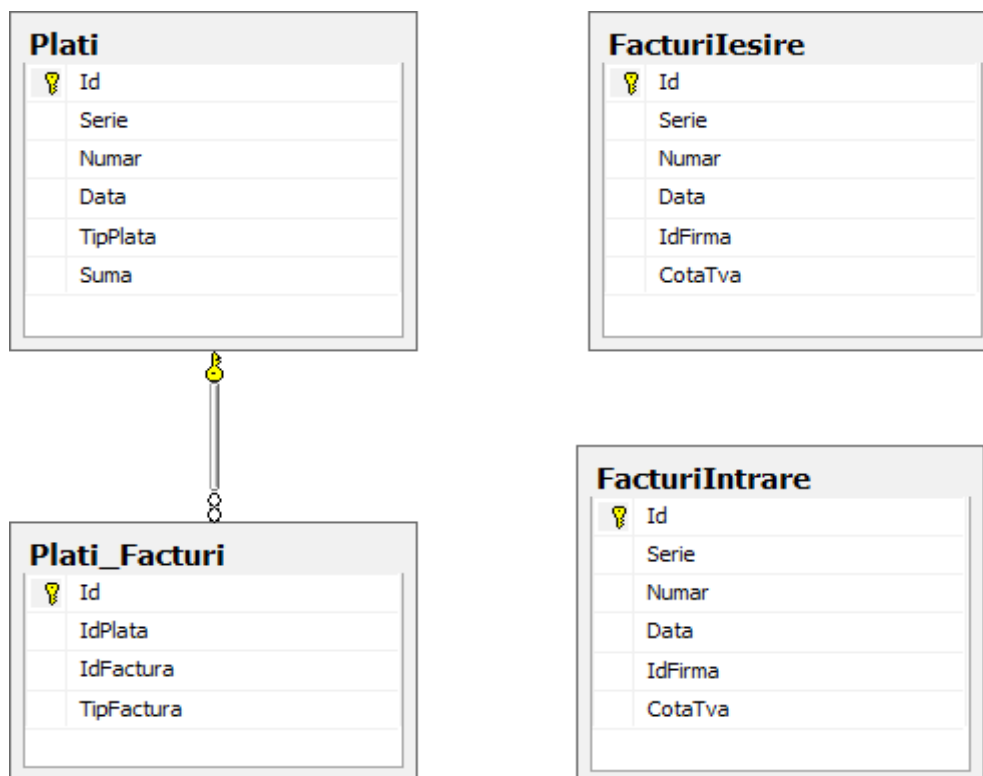


Figura 2.4. Tabelele care se ocupă cu plățile

Capitolul 2 – Baze de date relaționale

Tabela *Plăți* are următoarea relație:

Plăți(**Id**, Serie, Număr, Data, TipPlată, Sumă)

Relația Plăți conține attribute care sunt strict legate de cheie, nu conține grupuri compuse, în concluzie satisface necesitățile pentru a fi în forma normală trei.

Tabela *Plăți_Facturi* este o tabelă intermediară între plăți și facturi, are relația:

Plăți_Facturi(**Id**, *IdPlată*, *IdFactură*, TipFactură)

Această relație conține un sigur atribut care sugerează tipul facturii care este plătită și două attribute necesare pentru operațiile de cuplare cu tabelele *Plăți*, *FacturiIntrare* și *Facturileșire* pentru a putea afla care sunt facturile plătite, facturile neplătite, cât mai trebuie plătit pentru o factură și cât s-a plătit pentru o factură. Toate attributele sunt strâns legate de plăți pe facturi, iar având doar un atribut în cheie, relația *Plăți_Facturi* satisface condițiile pentru forma normală trei.

Aplicația mai conține și o componentă ce adaugă o facilitare de autentificare pentru folosirea aplicației. Această componentă presupune o autentificare înainte de folosirea aplicației, adăugare și modificarea drepturilor pentru fiecare utilizator creat și existența unui utilizator care este administrator pe întreaga aplicație și care nu poate fi șters.

Schema componentei (ilustrată în **Figura 2.5**) are trei tabele necesare pentru utilizatori, roluri și drepturile existente în aplicație. Mai există o tabelă de legătură între roluri și drepturi și o tabelă care nu este dependentă de alte tabele în care se reține parola administratorului și firma curentă a aplicației (necesară la eliberarea de facturi).

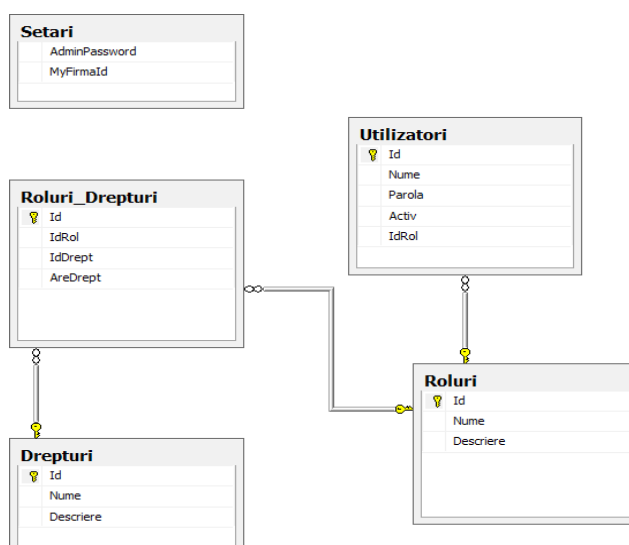


Figura 2.5 Tabelele care se ocupă cu partea de administrare

Capitolul 2 – Baze de date relaționale

Tabela *Utilizatori* este reprezentată de relația:

Utilizatori(**Id**, Nume, Parolă, Activ, *IdRol*)

Această relație se află în formă normală trei, deoarece nu conține grupuri compuse, iar toate atributele sunt strâns legate de cheie. Atributul *IdRol* este necesar pentru operațiile de cuplare cu tabela de roluri.

Tabela *Roluri* este reprezentată de relația:

Roluri(**Id**, Nume, Descriere)

Această tabelă este evident în a treia formă normală deoarece conține două atribute care au legătură exclusiv cu cheia relației.

Tabela *Drepturi* este reprezentată de relația:

Drepturi(**Id**, Nume, Descriere)

Tabela este în a treia formă normală deoarece conține două atribute care au legătură exclusiv cu cheia relației.

Tabela *Roluri_Drepturi* este reprezentată de relația:

Roluri_Drepturi(**Id**, *IdRol*, *IdDrept*, AreDrept)

Tabela este în a treia formă normală deoarece un singur atribut care se află în legătură strânsă cu cheia relației și două atribute necesare pentru cuplarea cu tabelele de roluri și drepturi.

Tabela *Setări* conține două atribute care reprezintă parola curentă a administratorului și firma care este setată ca fiind firma curentă a aplicației.

Capitolul 3

Tehnologii folosite în realizarea aplicației

3.1. Utilizarea tehnologiilor

Aplicația a fost realizată pentru platforma Microsoft .NET. Această platformă a apărut inițial în 2002 și dispune de patru versiuni: .NET 2.0, .NET 3.0, .NET 3.5 și .NET 4.0.

Folosind mediul integrat de dezvoltare Microsoft Visual Studio 2010 și platforma .NET 4.0 această aplicație a fost realizată pentru a rula pe sisteme de tip Windows.

Limbajul de programare folosit este C sharp, un limbaj de programare dezvoltat de Windows. Versiunea folosită este C# 4.0, care a fost lansată în data de 12 Aprilie 2010. Serverul folosit pentru baza de date este SQL Server 2008 împreună cu SQL Server Management Studio. Iar ca generator de rapoarte a fost folosit Telerik Reporting Q1 2012.

Pentru realizarea interfețelor grafice s-a folosit WinForms (Windows Forms) și WPF (Windows Presentation Foundation).

Windows Forms este un API (application programming interface – interfață pentru programare de aplicații) care este inclus în platforma .NET. Acest API se bazează destul de mult pe evenimente, principiul de funcționare fiind următorul: ”Aștept ca utilizatorul să introducă date pentru a face ceva”. Există și o versiune care oferă suport și pentru WinForms 2.0 în sisteme de tip linux, numită Mono.

Windows Presentation Foundation este interfață nouă de programare introdusă din platforma .NET 3.0 care aduce noi lucruri legate de realizarea interfețelor. Acesta folosește DirectX pentru ”desenarea” interfeței. Folosește un limbaj de marcare pentru realizarea interfeței numit XAML (eXtensible Application Markup Language).

SQL Server Management Studio oferă suport pentru crearea tabelor utilizând elemente grafice, dar și prin scrierea de cod SQL. Folosește o conexiune la un server și permite operații de back-up sau restore asupra bazelor de date, care sunt foarte utile. Permite și administrare utilizatorilor pentru server.

Capitolul 3 – Tehnologii folosite în realizarea aplicației

3.2. SQL Server

SQL (Structured Query Language) este un limbaj structura pentru interogarea bazelor de date relaționale. Modul de folosirea în aplicațiile de tip client/server presupune că aplicația client este cea care generează instrucțiunile SQL.

A fost lansat inițial de IBM, după care a fost standardizat pentru prima dată de ANSI, iar apoi ISO.

SQL operează asupra datelor normalizate, fiind un limbaj neprocedural. Conceptele necesare pentru lucrul cu acest limbaj sunt următoarele: tabelă, cheie primară, coloană, rând, index, bază de date relațională etc.

Tabela sau relația este un ansamblu format din n coloane și m rânduri (numite și tupluri/linii) care respectă următoarele condiții minime: valorile aflate la intersecția rândului cu coloane trebuie să fie de nivel elementar, liniile să nu se repete, iar descrierile coloanelor să nu fie repetitive.

Coloana tabelii este formată din valorile pe care le ia atributul respectiv în liniile tabelii.

Rândul/tuplul este format din valorile coloanelor ce se referă la o entitate a tabelii respective.

Accesul la date se face folosind indexarea, deoarece este rapidă. Un *index* poate fi reprezentat de o cheie pe una sau mai multe coloane. Această indexare se face automat, deoarece se pot adăuga sau șterge entități oricând, fără ca datele memorate să fie afectate.

Cheia primară este un atribut care nu permite valori duplicate, el este unic pe coloană, deci fiecare linie se identifică în mod unic.

Baza de date relațională este un ansamblu de tabele normalizate, grupate în jurul unui subiect, în principiu, bine definit. Într-o bază de date relațională, entitățile și legăturile sunt transpuse în tabele.

SQL Server este un sistem de gestionare a bazelor de date relaționale produs de către Microsoft. Principalele limbaje de interogare sunt MS-SQL și T-SQL. Este cel mai răspândit limbaj pentru bazele de date.

T-SQL (Transact- SQL) este o implementare a standardului SQL-92 (standard ISO

Capitolul 3 – Tehnologii folosite în realizarea aplicației

pentru SQL), dar mai facilitează și unele extensii.

Principalele tipuri de dată din SQL Server sunt:

Tip de dată	Capacitate
bigint	8 octeți
int	4 octeți
smallint	2 octeți
tinyint	1 octet
decimal, cu precizie între 1-38 zecimale	5 - 17 octeți
bit	1 octet
date	3 octeți
datetime	8 octeți
varchar(lungime)	max $2^{31}-1$ octeți
varbinary	max $2^{31}-1$ octeți
image	max $2^{31}-1$ octeți

Tabelul 3.1. *Principalele tipuri de dată în SQL Server*

Pentru a crea o bază de date în SQL Server trebuie într-un query nou să scriem următoarea secvență de cod:

```
CREATE DATABASE <numele bazei de date>
```

Această secvență provoacă apariția unei baze de date noi.

Pentru a adăuga o nouă tabelă utilizăm următoarea secvență de cod:

```
CREATE TABLE <numele tabelii>
```

```
(  
    nume_coloană1 tip_de_dată,  
    nume_coloană2 tip_de_dată,  
    nume_coloană3 tip_de_dată,  
    .....  
)
```

Capitolul 3 – Tehnologii folosite în realizarea aplicației

Unde *tip_de_dată* este un tip de dată din **Tabelul 3.1**, iar *nume_coloană* este numele atributului.

Adăugarea unui index se face folosind comanda CREATE INDEX:

```
CREATE UNIQUE INDEX nume_index ON nume_tabelă (nume_coloană)
```

Este de preferat să adăugăm și proprietatea de unicitate a indexului deoarece el trebuie să existe o singură dată într-o tabelă pentru se evita eventualele anomalii care pot să apară. Acest index va permite căutarea rapidă a datelor fără a fi nevoie să citim toate datele din tabelă.

Tabela mai are nevoie și de o cheie primară, este foarte importantă pentru indentificarea unei entități într-o tabelă cu multe date. Adăugare cheii primare trebuie menționată în comanda de creare a tabelului. Astfel comanda pentru crearea unei tabeli ce conține și o cheie primară devine:

```
CREATE TABLE <numele tabelii>
(
    coloană_primară INT PRIMARY KEY IDENTITY
    nume_coloană2 tip_de_dată,
    nume_coloană3 tip_de_dată,
    .....
)
```

O cheie primară trebuie să fie de tip întreg și trebuie setată ca fiind auto-incrementabilă prin adăugarea declarației IDENTITY (aceasta asigură și unicitatea cheii), iar adăugarea de *PRIMARY KEY* specifică faptul că acest atribut este cheie primară.

Mai trebuie menționat și faptul că pentru a efectua operații de cuplare între tabele trebuie să creăm și un atribut numit *FOREIGN KEY*, cheie străină. Acest se realizează tot în declarația de creare a tabelului.

Capitolul 3 – Tehnologii folosite în realizarea aplicației

```
CREATE TABLE <numele tabelii>
(
    coloană_primară INT PRIMARY KEY IDENTITY,
    nume_coloană2 tip_de_dată,
    nume_coloană3 tip_de_dată,
    cheie_străină INT FOREIGN KEY REFERENCES tabelă(cheie_primară)
    .....
)
```

Cheia străină este specificată printr-o referință către cheia primară a altei table. Această dependență impune ca orice valoare a atributului care este cheia străină, să existe în tabela cu care se află în relație ca și cheia primară.

Vom exemplifica crearea unei baze de date *Gestiune* și a unei table *Produse* ce are ca cheia primară un atribut numit *ID*.

```
CREATE DATABASE Gestiune
```

```
CREATE TABLE PRODUS
(
    ID INT PRIMARY KEY IDENTITY,
    NUME_PRODUS VARCHAR(50),
    PRET_PRODUS DECIMAL
)
```

Continuăm exemplul prin adăugarea unei noi table STOC care va conține și un atribut de tip numeric ce va fi cheia străină cu referință către produs pentru a putea reține cantitatea din fiecare produs.

```
CREATE TABLE STOC
(
    ID INT NOT NULL PRIMARY KEY,
    CANTITATE DECIMAL,
    ID_PRODUS INT FOREIGN KEY REFERENCES PRODUS(ID)
)
```

Capitolul 3 – Tehnologii folosite în realizarea aplicației

Pentru adăuarea, modificarea și ștergerea datelor dintr-o tabelă SQL Server ne pune la dispoziție *Procedurile Stocate*, acestea reprezintă bucăți de cod ce sunt salvate în baza de date, iar la dorința utilizatorului ele pot fi apelate. Procedurile stocate permit și parametrării pentru codul ce îl execută. Există mult prea faimoasele operații CRUD (Create, Read, Update și Delete), adică creare, citire, modificare și ștergere, care sunt necesare pentru fiecare tabelă, mai puțin operația de ștergere, care în general nu este recomandată, rolul bazei de date fiind să stocheze date.

Operația de creare într-o bază de date se face folosind declarația INSERT, această declarație inserează un rând nou în tabelă.

```
INSERT INTO nume_tabelă (colana1, colana2, ...) VALUES (valoare1, valoare2, ...)
```

Exemplu de creare pe tabela produs creată anterior:

```
INSERT INTO  
PRODUS(NUME_PRODUS, PRET_PRODUS)  
VALUES('Zahar', 10)
```

Trebuie să menționăm faptul că nu mai trebuie să precizăm o valoare pentru atributul ID, acesta având specificatorul *IDENTITY* are proprietatea ca la inserarea datelor se adaugă automat, valoarea nouă fiind valoare veche plus o unitate.

Exemplu de citire a datelor din tabela PRODUS:

```
SELECT ID, NUME_PRODUS,PRET_PRODUS FROM PRODUS  
sau  
SELECT * FROM PRODUS
```

Aceste două variante de citire a datelor dintr-o tabelă sunt relativ la fel, dar a doua variantă nu reflectă în mod clar datele care ne sunt aduse din tabelă.

Pentru exemplificarea operației de modificare într-o tabelă este bine să introducem și două concepte noi, cel de procedură stocată și cel de parametru. O procedură stocată se

Capitolul 3 – Tehnologii folosite în realizarea aplicației

crează folosind declarația *CREATE PROCEDURE*, iar parametrii se specifică folosind semnul ”@”.

Exemplu de procedură stocată ce modifică un rând din tabela produs folosind ca parametru id-ul rândului ce trebuie modificat, numele nou al produsului și prețul nou al produsului.

```
CREATE PROCEDURE Modifica_produc  
@id int,  
@nume_nou varchar(50),  
@pret_nou decimal  
AS  
BEGIN  
UPDATE PRODUS SET  
NUME_PRODUS=@nume_nou,  
PRET_PRODUS=@pret_nou  
WHERE ID=@id  
END
```

Pentru ștergerea datelor dintr-o tabelă se poate face o procedură stocată, dar de data asta nu va mai fii nevoie decât de id-ul rândului pe care vrem să-l ștergem.

```
CREATE PROCEDURE Sterge_produc  
@id int  
AS  
BEGIN  
DELETE FROM PRODUS WHERE ID=@id  
END
```

Declarațiile necesare pentru operațiile de tip CRUD sunt: *SELECT*, *INSERT*, *UPDATE* și *DELETE*.

Capitolul 3 – Tehnologii folosite în realizarea aplicației

3.3. Limbajul C#

Limbajul C# este un limbaj imperativ, obiect-orientat. Este asemănător din punct de vedere sintactic cu Java și C++, astfel trecerea de la unul dintre limbaje la celălalt nu este foarte abruptă.

Orice program scris în C# conține o serie de elemente, exemplu:

```
using System;
class HelloWorld
{
    public static void Main()
    {
        Console.WriteLine("Hello world!");
    }
}
```

Prima linie *using System;* este o directivă care specifică faptul că se vor folosi clasele care sunt incluse în spațiul de nume *System*. Spațiul de nume este o colecție de tipuri sau de alte spații de nume care pot fi folosite într-un program. În spațiul de nume *System* se află și spațiul de nume *Console*.

Un program trebuie să conțină și o metodă *Main* care returnează o valoare întreagă; această metodă se execută la pornirea programului, reprezintă punctul de intrare în aplicație, poate să preia parametrii din linie de comandă și trebuie să returneze o valoare. Modificarea programului pentru a accepta parametrii din linie de comandă este următoarea:

```
public static void Main(String[] args)
{
    for( int i = 0; i < args.Length; i++)
    {
        Console.WriteLine(args[i]);
    }
}
```


Capitolul 3 – Tehnologii folosite în realizarea aplicației

C# este un limbaj de programare orientat pe obiecte(object oriented programming) respectând toate principiile: încapsularea, supraîncărcarea, moștenirea, polimorfism. Toate componentele limbajului sunt într-un fel sau altul, asociate noțiunii de clasă. Chiar și tipurile primitive: byte, int sau bool sunt clase sigilate derivate din clasa ValueType din spațiul de nume System. Pentru a evita unele tehnici de programare periculoase, limbajul oferă tipuri speciale cum ar fi: interfețe și delegări. Versiunii 2.0 a limbajului i s-a adăugat un nou tip: clasele generice

Sintaxa de declarare a unei clase este următoarea:

```
[atribut] [modificator] class [nume_clasă] [: clasă_de_bază]
{
    corp_clasă
}
```

Clasa de bază reprezintă clasa care este părintele clasei curente, clasa curentă moștenind toate calitățile acelei clase.

Atributele oferă o metodă puternică de a asocia informații declarative cu cod C#.

Clasele pot conține constante, câmpuri, metode, proprietăți, evenimente, indexatori, operatori, constructori, destructori, tipuri imbricate. Modificatorii de acces oferă un nivel de protecție care controlează gradul de acces la componentele clasei. Modificatorii sunt ilustrați în **Tabelul 3.2** împreună cu semnificațiile lor.

Accesor	Semnificație
public	Acces nelimitat
protected	Accesul este limitat la clasa conținătoare și la cele care derivă din ea
internal	Accesul este limitat la assembly-ul curent
protected internal	Acces limitat la assembly-ul curent sau la clasele derivate din ea
private	Acces limitat doar la clasa conținătoare, acesta este și modificatorul implicit

Tabelul 3.2. *Modificatorii de acces*

Capitolul 3 – Tehnologii folosite în realizarea aplicației

Exemplu în care se folosesc de modificatorii de acces:

```
public class A
{
    public string Nume;

    private int cantitate;
    public int Cantitatea
    {
        get
        {
            return cantitate;
        }
        set
        {
            cantitate = value;
        }
    }
    public A( int cantitate )
    {
        this.cantitate = cantitate;
    }
    public void AfișeazăTest(string text)
    {
        Console.WriteLine("Test " + text);
    }
    protected void AfișeazăTest2(string text)
    {
        Console.WriteLine("Test2 " + text);
    }
}
```

Capitolul 3 – Tehnologii folosite în realizarea aplicației

```
public class B : A
{
    public B(string nume)
    {
        this.Nume = nume;
    }
    public void SchimbăCantitate(int cantitate)
    {
        // this.cantitate = cantitate; // nu funcționează din cauza gradului de acces
        AfișeazăTest2(cantitate.ToString()); // funcționează
    }
    private class b
    {        // clasă în clasă sau clasa imbricată    }
}
class Test
{
    static void Main()
    {
        A obiectA = new A(20);
        B obiectB = new B("TestB");
        // B.b obiectBb = new B.b(); // nu funcționează datorită nivelului de acces
        A.AfișeazăTest("Test");
        // A.AfișeazăTest2("Test2"); // nu funcționează
        B.SchimbăCantitate(20); // funcționează
        Mesaj();
    }
    static void Mesaj()
    {
        Console.WriteLine("Bye");
    }
}
```

Capitolul 3 – Tehnologii folosite în realizarea aplicației

Exemplificăm următoarea ierarhie de clase

```
public abstract class Mașină { }  
public class Dacia: Mașină { }  
public sealed class Trabant: Mașină { }
```

Modificatorul *abstract* este folosit pentru a desemna faptul că nu se pot obține obiecte din clasa Mașină, practic este o clasă ce nu poate fi instanțiată ci numai derivată, iar modificatorul *sealed* a fost folosit pentru a desemna faptul că nu se mai pot obține clase derivate din clasa Trabant (de exemplu, subclasa TrabantNou).

Constructorul este o metodă ce se execută la instanțierea unei clase, ea poate primi diverși parametrii, o clasa poate avea mai mulți constructori.

Tipurile de valori întregi sunt cele mai simple tipuri de valoare pentru computer pentru a stoca. Fiecare valoare va mapa pe un model special de biți. Singura problemă este că permite stocarea valorilor dintr-un anumit interval. Cu cât este mai mare valoarea pe care vrem să o reprezentăm cu atât numărul de biți trebuie să fie mai mare. C# oferă o varietate de tipuri de întregi, în funcție de intervalul de valori pe care în care se dorește să se stocheze.

sbyte	8 octeți	-128 până la 127
byte	8 octeți	0 până la 255
short	16 octeți	-32768 până la 32767
ushort	16 octeți	0 până la 65535
int	32 octeți	-2147483648 până la 2147483647
uint	32 octeți	0 până la 4294967295
long	64 octeți	-9223372036854775808 până la 9223372036854775807
ulong	64 octeți	0 până la 18446744073709551615
char	16 octeți	0 până la 65535

Tabelul 3.3. Tipurile întregi de dată

Capitolul 3 – Tehnologii folosite în realizarea aplicației

"Real" este un termen generic pentru numere care nu sunt numere întregi. Ele au un parte zecimală și o parte fracționară. În funcție de valorile zecimalelor, plutește în jurul numărului, de aici numele de *float* (plutire). Tipul standard *float* are un interval între 1,5E-45 și 3,4E48 cu o precizie de doar 7 zecimale. Dacă dorim mai multă precizie putem folosi tipul *double* care are o precizie de 15 zecimale, iar intervalul este de la 5.0E-324 până la 1.7E308. Mai este un tip, *decimal*, care are cea mai mare precizie 28 de zecimale.

Tipurile literale sunt *char* și *string*, care pot reține orice șiruri de caractere. Iar tipul rămas nemenționat încă este tipul *boolean* care are două valori de adevărat și fals (true / false).

Colecțiile generice au apărut împreună cu clasele generice. Listele, notate List<T> sunt colecții generice care sunt utile pentru manipularea șirurilor de date de același tip.

Exemplu:

```
List<int> intList = new List<int>();  
intList.Add(30);  
intList.Add(6);  
intList.Add(35);
```

Dacă încercăm să introducem un alt tip de dată decât int atunci va rezulta o eroare, lista nu permite decât tipul int.

Tratarea excepțiilor este o măsură de siguranță, C# oferă posibilitatea de a face față oricărei situații excepționale sau neprevăzute, care apare în timpul execuției unui program. Pentru a manipula excepțiile, limbajul folosește try, catch și finally, acestea fiind cuvinte cheie pentru a testa reușita anumitor acțiuni, pentru a stopa eșecul (failure) și pentru a șterge memoria. Excepțiile pot fi generate de către Common Language Runtime (CLR), de către alte biblioteci, sau din codul aplicației folosind cuvântul cheie throw.

Exemplu:

```
// împărțirea a 0
```

```
int ImparteLa(int nr, int imp)  
{
```

Capitolul 3 – Tehnologii folosite în realizarea aplicației

```
    return nr / imp;  
}
```

Acest cod nu este sigur, dacă se trimite ca parametru orice număr și zero atunci el va genera o excepție, care dacă nu este capturată în nicio porțiune de cod atunci toată aplicația va eșua. Pentru a face față oricăror numere transmise acelei metode trebuie să introducem un bloc try - catch în metodă. Metoda rescrisă va arăta astfel:

```
int ImparteLa(int nr, int imp)  
{  
    try  
    {  
        int rezultat = nr / imp;  
        return rezultat;  
    }  
    catch(Exception)  
    {  
        return 0;  
    }  
}
```

În acest caz, orice excepție provocată de împărțirea celor două numere va provoca returnarea rezultatului zero.

Excepțiile mai pot fi provocate și intenționat folosind *throw* putem arunca excepții de oriunde, chiar din blocul de *catch*. În exemplul de mai sus, dacă metoda *ImparteLa* se află într-un bloc try-catch în metoda care o apelează atunci putem face verificarea înainte și aruncăm o excepție dacă *imp* este zero.

```
int ImparteLa(int nr, int imp)  
{  
    if(imp == 0)
```

Capitolul 3 – Tehnologii folosite în realizarea aplicației

```
{
    throw new Exception("Împărțire la zero!");
}
return nr / imp;
}

void functieX()
{
    try
    {
        int rez = ImparteLa(5, 0);
    }
    catch(Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}
```

La apelul *functieX()* se intră în funcția *ImparteLa*, *imp* fiind egal cu zero se intră pe ramura if-ului și se aruncă o excepție cu mesajul: "Împărțire la zero!", această excepție este capturată în blocul de catch al funcției *functieX* și se afișează în consolă mesajul excepției.

Clasele care permit lucrul cu baza de date SQL se află în spațiul de nume *System.Data.SqlClient*. Pentru persistență avem nevoie de trei clase: *SqlConnection*, *SqlCommand* și *SqlDataReader*.

SqlConnection este folosit pentru a deschide o conexiune cu serverul SQL pentru a accesa baza de date. Primește ca parametru un string de conexiune care conține informații despre conexiune: se specifică serverul, username-ul parola și baza de date cu care se va lucra. Stringul de conexiune arată astfel:

```
"Data Source=ADI-PC\SQLEXPRESS;Initial Catalog=GestiuneFirma;Persist Security
Info=True;User ID=sa;Password=1q2w3e4r;"
```

Capitolul 3 – Tehnologii folosite în realizarea aplicației

SqlCommand este folosită pentru a apela o procedură stocată deja existentă, prin metoda *AddWithValue* putem adăuga parametrii necesari pentru procedură:

```
SqlConnection con = new SqlConnection(connectionString); // specificare conexiunii
SqlCommand cmd = new SqlCommand(ProceduraStocata, con);
cmd.CommandType = System.Data.CommandType.StoredProcedure;
con.Open(); // se deschide conexiunea cu baza de date
SqlDataReader reader = cmd.ExecuteReader(); // execută procedura stocată din baza de date
while(reader.Read()) // reader face citirea fiecărui rând din interogarea rezultată
{
    obiect = reader[0],
    .....
    // transformarea datelor citite din baza de date în echivalentul lor din C#
    .....
}
reader.Close();
con.Close();
```

Reader-ul și conexiunea trebuie închise după ce s-a încheiat lucrul cu baza de date, pentru că altfel pot duce la rezultate neașteptat, chiar și excepții. Deschiderea unei conexiuni de mai multe ori duce la apariția unei excepții. Cea mai sigură metodă de a folosi conexiune este cu blocul *using*.

```
using(SqlConnection con = new SqlConnection(connectionString))
{
    // aici se face persistența
}
```

Folosirea blocului *using* provoacă distrugerea obiectului folosit la părăsirea blocului, asta presupune automat și închiderea conexiunii cu baza de date.

Capitolul 3 – Tehnologii folosite în realizarea aplicației

3.4. Telerik Report

Telerik este o compania care produce o multitudine de produse soft pentru .NET. Telerik Reporting este unul dintre produsele soft produse de Telerik pentru generarea rapoartelor folosind platforma .NET în Microsoft Visual Studio. Telerik Reporting este disponibil pentru platforme de tip cloud, web și desktop (Azure, Silverlight, WPF, ASP.NET și Windows Forms). Din aceste categorii, generatorul de rapoarte folosit în aplicație este de tip desktop, mai exact pentru Windows Forms.

Acest generator de rapoarte folosește sursele de date (datasource) pentru a-și genera conținutul, astfel putem lega la un raport o listă de obiecte, pe care le putem dispune convenabil pe raport, iar ulterior raportul știe să-și autogenereze conținutul în funcție de obiectele din listă.

De exemplu, fie clasa Persoana:

```
class Persoana
{
    public string Nume;
    public int Varsta;
}
```

Dacă legăm la raport o colecție de Persoane atunci, prin adăugarea unui control pe raport, putem accesa și afișa proprietățile clasei direct din generatorul de raport (**Figura 3.1**). Acest generator are un avantaj foarte mare deoarece permite accesare claselor imbricate, spre deosebire de generatorul clasic de rapoarte de la Microsoft. Accesare datelor din sursa de date atașată raportului se face folosind cuvântul *Fields*, în cazul persoane accesarea numelui pe raport se face astfel: "`=Fields.Nume`".

U.M.	Cantitate
[=Fields.StocObject.ProdusObject.Um]	[=Fields.Cantitate]

Figura 3.1. Exemplu de introducere date în generatorul de rapoarte Telerik

Capitolul 4 - Prezentarea aplicației

4.1. Arhitectura aplicației

Aplicația este alcătuită din două proiecte mari și unul mai mic care oferă o interfață în WPF (Windows Presentation Foundation) pentru autentificarea utilizatorului (**Figura 4.1**).

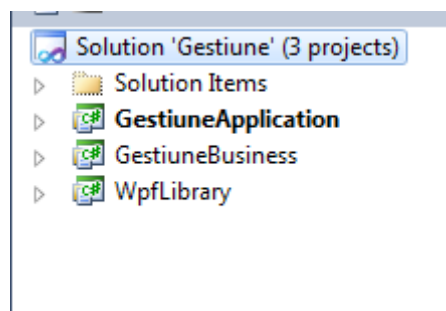


Figura 4.1. Proiectele aplicației

Proiectul care conține toată logica aplicației se numește *GestiuneBusiness* și conține toate clasele necesare pentru lucrul cu baza de date și obiectele sinonime din baza de date.

Fiecare tabelă are asociată câte o clasă în proiectul *GestiuneBusiness*, fiecare astfel de clasă are aceleași proprietăți ca și tabela pe care o reprezintă, dar mai moștenește clasa abstractă *GestiuneObject* al cărei cod este următorul:

```
public abstract class GestiuneObject : IDataPersistence
{
    public int ID { get; set; }

    public abstract string NumeCompact { get; }

    public abstract PersistenceResult Save();

    public bool Contains(string text)
    {
        // verifică dacă atributele clasei moștenitoare conține textul 'text'
    }
}
```

Capitolul 4 - Prezentarea aplicației

```
public string GetErrorString()
{
    // returnează un text în care specifică dacă attributele au valoare sau nu
}

public abstract List<DBObject> PropertiesNamesWithValues { get; }

protected const string StringSaveSuccess = "Salvare efectuada cu succes!";

protected const string StringSaveFail = "Au aparut erori in timpul salvarii! Verificati
datele!";
}
```

ID reprezintă cheia primară a fiecărei tabele, ea se moștenește în fiecare clasă care extinde clasa *GestiuneObject*.

PropertiesNamesWithValues este o proprietate abstractă care lasă clasa moștenitoare să decidă care este lista atributelor ei. De exemplu, în cazul unui delegat, această proprietate va returna Nume, CNP și Serie.

Toate aceste clase inspirate după tabele din baza de date se salvează și se actualizează prin intermediul claselor derivate din clasa abstractă *GestiuneDataHelper*:

```
internal abstract class GestiuneDataHelper
{
    protected string selectAllStoredProcedureName = "";

    protected string insertStoredProcedureName = "";

    protected string updateStoredProcedureName = "";

    private const string StringProcedureFail = "Stored procedure failed: ";
}
```

Capitolul 4 - Prezentarea aplicației

```
private const string StringDatabaseFail = "Connection to database failed.";

public List<GestiuneObject> GetAll()
{
    // metodă care se leagă la baza de date și aduce toate înregistrările dintr-o tabelă
}

public int Create(List<DbObject> dbObjects)
{
    // metodă care se leagă la baza de date și salvează o înregistrare dintr-o tabelă
}

public void Update(List<DbObject> dbObjects, int id)
{
    // metodă care se leagă la baza de date și actualizează o înregistrare dintr-o tabelă
}

protected abstract GestiuneObject ToPocoObject(SqlDataReader reader);
}
```

Clasele care derivează din *GestiuneDataHelper* se ocupă cu persistența bazei de date și transformarea entităților citite în clase sinonime și sunt singleton deoarece instanțierea lor de mai multe ori nu ar avea sens, este de preferabil să existe o singură instanță a fiecărei clase *DataHelper* pe întreaga aplicație. Pentru fiecare clasă sinonimă cu o tabelă din baza de date există o clasă de tip *DataHelper*.

Metoda *protected abstract GestiuneObject ToPocoObject* este abstractă deoarece lasă clasa moștenitoare să transforme datele citite din baza de date în obiectul care se dorește a fi creat.

Lucrul cu clasele data helper se întâmplă exclusiv în clasele care derivă *GestiuneObject*, ele sunt complet transparente în afara proiectului. Acest proiect utilizează

Capitolul 4 - Prezentarea aplicației

încărcarea datelor înainte de a fi folosite în niște liste, care rețin, ca un cache, toate datele. Această abordare reduce semnificativ numărul apelurilor claselor care se ocupă cu citirea datelor din baza de date și transformarea lor în obiecte, mai exact apelul claselor *DataHelper*. Această încărcare are loc o singură dată după prima autentificare a unui utilizator prin apelul funcției *GetAll()* care încarcă cache-ul cu datele din baza de date. Iar pe măsură ce obiectele se salvează cu succes ele se adaugă automat în acest cache.

Proiectul aplicației principale se numește *GestiuneApplication*. Rolul ei este să apeleze funcționalitățile proiectului de business și să le dispună corespunzător pentru utilizator. Singurele clase care sunt complet dependente de acest proiect sunt cele pentru realizarea rapoartelor.

Cel de-al treilea proiect: *WpfLibrary* conține o interfață grafică pentru autentificarea utilizatorului. Această interfață este realizată folosind limbajul de marcare XAML:

```
<Border CornerRadius="10" BorderThickness="3" Background="White" Padding="20"
Margin="4" BorderBrush="Black">
    <Border.Effect>
        <DropShadowEffect Color="Gray" Opacity=".50" ShadowDepth="6"/>
    </Border.Effect>
    <Grid Background="White"
        FocusManager.FocusedElement="{Binding ElementName=usernameTbox}">
        <Label Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="2" FontSize="24"
            HorizontalAlignment="Center"
            VerticalAlignment="Center">Introduceti datele pentru autentificare</Label>
        <Label Grid.Row="1"
            Grid.Column="0" FontSize="20" HorizontalAlignment="Right"
            VerticalAlignment="Center">Utilizator</Label>
        <Label Grid.Row="2"
            Grid.Column="0" FontSize="20" HorizontalAlignment="Right"
            VerticalAlignment="Center">Parola</Label>
        <TextBox Grid.Row="1"
            Grid.Column="1" Margin="10" Name="usernameTbox"
```

Capitolul 4 - Prezentarea aplicației

```
        ToolTip="Introduceți numele utilizatorului" TabIndex="0"/>
    <PasswordBox Grid.Row="2"
        Grid.Column="1" Margin="10" Name="pwdBox"
        ToolTip="Introduceți parola utilizatorului" TabIndex="1" />
    <Button Grid.Column="0" Grid.Row="3" Margin="10" Name="exitBtn"
IsCancel="True"
        Content="Iesire" Click="exitBtn_Click" TabIndex="3">
    <Button.Effect>
        <DropShadowEffect Color="Gray" Opacity=".50" ShadowDepth="8" />
    </Button.Effect>
</Button>
<Button Grid.Column="1" Grid.Row="3" HorizontalAlignment="Right"
    Width="80" Margin="10" Name="loginBtn"
    Content="Intrare"
        Click="loginBtn_Click"
        IsDefault="True"
        IsCancel="False"
        TabIndex="2">
    <Button.Effect>
        <DropShadowEffect Color="Gray" Opacity=".50" ShadowDepth="8" />
    </Button.Effect>
</Button>
</Grid>
</Border>
```

Această clasă conține și un eveniment care este generat în momentul în care se face click pe butonul de *Intrare* al interfeței (**Figura 4.2**).

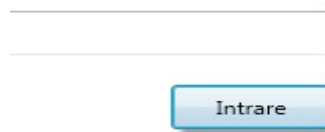


Figura 4.2. Butonul de intrare de pe interfața de autentificare

Capitolul 4 - Prezentarea aplicației

Declararea evenimentului pentru generarea acestuia se face declarând mai întâi un delegat, după care evenimentul va fi de tipul acelui delegat, iar generarea acestuia se face apelând *OnLoginClick* cu parametrii corespunzători:

```
public delegate void ChangedEventHandler(string username, string password);
```

```
public event ChangedEventHandler OnLoginClick;
```

```
private void loginBtn_Click(object sender, RoutedEventArgs e)
{
    OnLoginClick(usernameTbox.Text, pwdBox.Password);
    pwdBox.Password = string.Empty;
}
```

Evenimentul semnalizează faptul că s-a făcut click pe buton și trimite numele utilizatorului și parola metodei care este înregistrată la acest eveniment. De exemplu în forma principală din proiectul *GestiuneApplication*:

```
private void loginWindow_OnLoginClick(string username, string password)
{
    // validarea utilizatorului
}
```

Înregistrare unei metode la un eveniment se face prin intermediul unui delegat:

```
loginWindow.OnLoginClick += new ChangedEventHandler(loginWindow_OnLoginClick);
```

Prin înregistrarea metodei *loginWindow_OnLoginClick* la evenimentul de click pe butonul de intrare se asigură faptul că de fiecare dată când un utilizator va da click pe butonul de intrare se va apela metoda *loginWindow_OnLoginClick*, din proiectul principal, pentru a se face validarea acestuia.

Capitolul 4

Prezentarea aplicației

4.2 Introducerea de date

Funcționalitățile aplicației sunt organizate într-o structură de tip tree care este permanent vizibilă în partea stângă a aplicației.

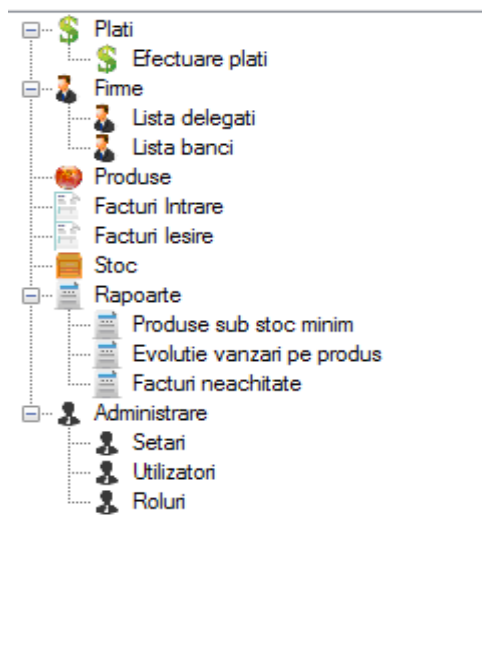


Figura 4.3. Tree-ul aplicației

Pentru folosirea aplicației trebuie introduse datele de bază, acestea sunt: băncile, delegații, firmele și produsele. Toate funcționalitățile aplicației se bazează pe aceste patru tipuri de dată. Fiecare clasă care are un echivalent în baza de date extinde clasa `GestiuneObject`, o clasă abstractă care conține metode pentru verificarea erorilor și o listă care reține numele fiecărei proprietăți. Această listă nu este implementată, este abstractă deoarece fiecare clasă ce derivă din ea trebuie să își enumere proprietățile.

Capitolul 4 - Prezentarea aplicației

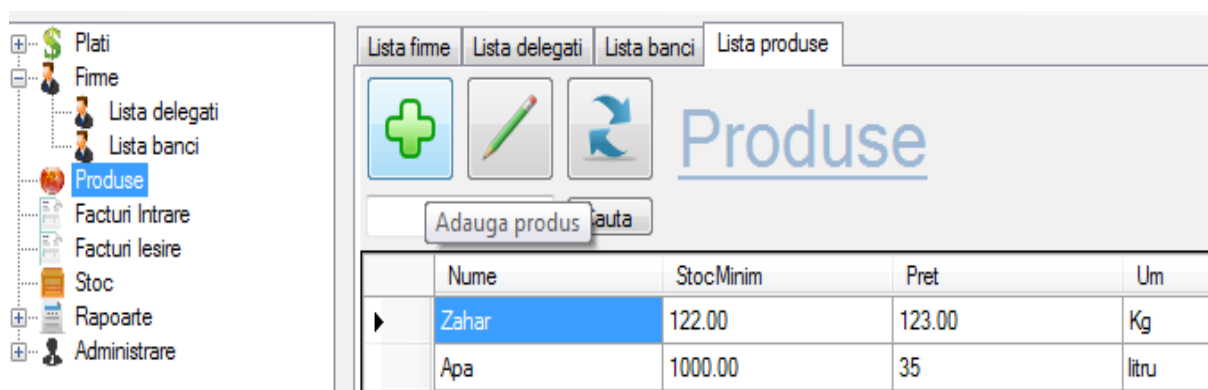


Figura 4.4. Deschiderea listei de produse

Pentru adăugarea unui produs trebuie să deschidem tabul cu lista produselor: din tree-ul din partea dreaptă (**Figura 4.3**) deschidem Produse și toate produsele sunt enumerate într-un grid, după care facem click pe butonul cu plus verde din **Figura 4.4**, în general în toată aplicația semnificația acestui buton este de adăugare, completăm datele necesare pentru un produs: numele, stocul minim, prețul și unitatea de măsură a produsului și apăsăm butonul de salvare din **Figura 4.5**.

Adauga produs

Nume

Stoc minim Pret

Unitate de masura

Iesire Salvare

Figura 4.5. Completarea datelor pentru adăugarea unui produs

La salvare se apelează codul funcției Create, care se ocupă de persistența bazei de date, mai exact operația de inserare:

```
public int Create(List<DBObject> dbObjects)
{
```

Capitolul 4 - Prezentarea aplicației

```
try
{
    using (SqlConnection con = ConnectionHelper.Connection)
    {
        try
        {
            using (SqlCommand cmd =
                    new SqlCommand(insertStoredProcedureName, con))
            {
                const string ReferenceId = "@ReferenceID";
                con.Open();
                cmd.CommandType = System.Data.CommandType.StoredProcedure;
                cmd.Parameters.Add(ReferenceId, System.Data.SqlDbType.Int).Direction =
System.Data.ParameterDirection.ReturnValue;

                foreach (var dboObject in dbObjects)
                {
                    cmd.Parameters.AddWithValue(dboObject.Name, dboObject.Value);
                }

                cmd.ExecuteReader();

                var id = cmd.Parameters["@ReferenceID"].Value;
                return (int)id;
            }
        }
        catch (Exception ex)
        {
            throw new Exception(StringProcedureFail + insertStoredProcedureName, ex);
        }
    }
}
```

Capitolul 4 - Prezentarea aplicației

```
}  
catch (Exception ex)  
{  
    throw new Exception(StringDatabaseFail, ex);  
}  
}
```

Această funcție deschide conexiunea cu baza de date și apelează procedura stocată din baza de date care se ocupă cu inserarea de produse. Funcția primește ca parametru o listă de dbObjects care este o colecție de informații despre proprietățile unui produs. Se încearcă salvarea produsului, dacă operația se efectuează cu succes atunci se returnează id-ul înregistrării din baza de date, dacă nu, atunci se aruncă o excepție cu un mesaj corespunzător. Procedura stocată ce adaugă produse în baza de date este următoarea:

```
CREATE Procedure [dbo].[sp_Produse_Insert]  
    @Nume nvarchar(50),  
    @StocMinim decimal(18,2),  
    @Pret decimal(18,2),  
    @Um nvarchar(50)  
  
As  
Begin  
    Insert Into Produse  
        ([Nume],[StocMinim],[Pret],[Um])  
    Values  
        (@Nume,@StocMinim,@Pret,@Um)  
  
    Declare @ReferenceID int  
    Select @ReferenceID = @@IDENTITY  
  
    Return @ReferenceID  
  
End
```

Capitolul 4 - Prezentarea aplicației

Adăugarea unei bănci se face deschizând din tree *Firmele* după care deschidem *Listă Bănci* ce se expandează din *Firme*, ca în **Figura 4.6**. Se deschide tabul cu lista băncilor de unde putem adăuga o nouă bancă sau să modificăm o bancă existentă.



Figura 4.6. Deschiderea listei cu bănci

Pentru adăugarea unei bănci trebuie să completăm informațiile necesare despre o bancă: nume, adresă, CUI (cod unic de identificare), RC (registru comerț) și capitalul social într-o formă ilustrată în **Figura 4.7**.

Adauga banca

Nume

Adresa

CUI RC

Capital social

Iesire Salvare

Figura 4.7. Completarea datelor pentru salvarea unei bănci

La salvarea unei bănci se apelează aceeași funcție Create, dar în acest caz lista dbObjects va conține informații despre proprietățile băncii. Iar procedura stocată care se ocupa cu salvarea entității în baza de date este:

```
CREATE Procedure [dbo].[sp_Banci_Insert]
    @Nume varchar(50),
    @Adresa varchar(50),
    @CUI varchar(50),
```

Capitolul 4 - Prezentarea aplicației

@RC varchar(50),

@CapitalSocial decimal(18,2)

As

Begin

Insert Into Banci

([Nume],[Adresa],[CUI],[RC],[CapitalSocial])

Values

(@Nume,@Adresa,@CUI,@RC,@CapitalSocial)

Declare @ReferenceID int

Select @ReferenceID = @@IDENTITY

Return @ReferenceID

End

Adăugarea unui delegat presupune deschiderea listei de delegați din **Figura 4.8** și apelarea formei de adăugare a unui delegat.

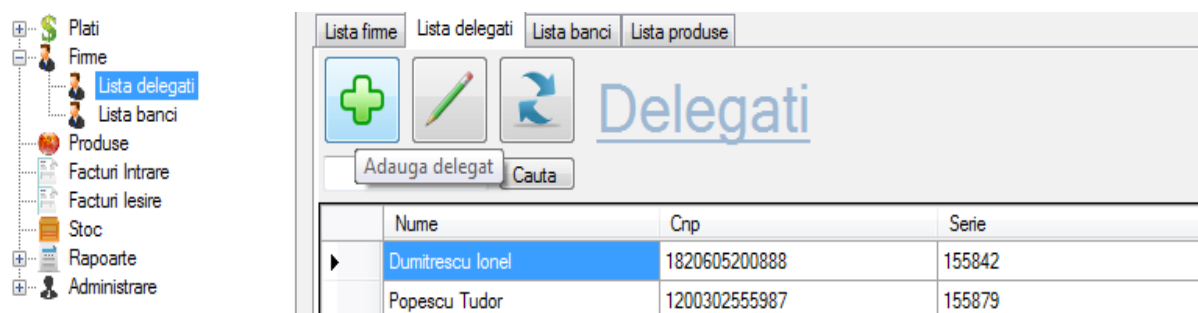


Figura 4.8. Deschiderea listei cu delegați

Butonul cu plus deschide o formă similară cu cea pentru produs și bancă ca în **Figura 4.9**. Datele necesare pentru adăugarea unui delegat fiind: numele, CNP (codul numeric personal) și seria.

Capitolul 4 - Prezentarea aplicației

Figura 4.9. Completarea datelor pentru adăugarea unui delegat

Salvarea delegatului se face prin apelul funcției Create, care la rândul ei apelează procedura stocată de inserare în tabela delegaților:

```
CREATE Procedure [dbo].[sp_Delegati_Insert]
```

```
    @Nume varchar(50),
```

```
    @CNP varchar(50),
```

```
    @Serie varchar(50)
```

```
As
```

```
Begin
```

```
    Insert Into Delegati
```

```
        ([Nume],[CNP],[Serie])
```

```
    Values
```

```
        (@Nume,@CNP,@Serie)
```

```
    Declare @ReferenceID int
```

```
    Select @ReferenceID = @@IDENTITY
```

```
    Return @ReferenceID
```

```
End
```

Firma este o relație ce are două chei străine: delegatul și banca. Pentru a putea adăuga

Capitolul 4 - Prezentarea aplicației

o firmă trebuie mai întâi să adăugăm cel puțin o bancă și un delegat. Din tree se deschide lista de firme (**Figura 4.10**) și se selectează adăugarea unei firme.

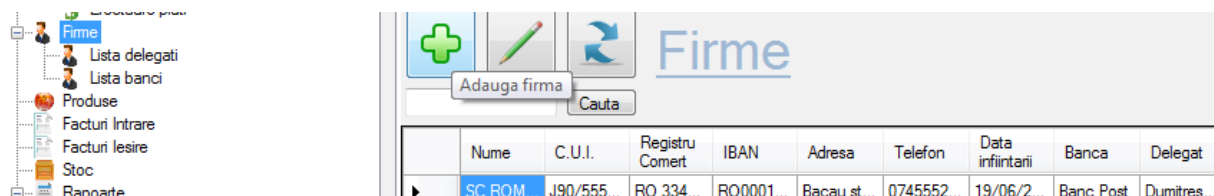


Figura 4.10. Deschiderea listei cu firme

Pentru adăugarea unei firme trebuie să completăm numele, CUI, IBAN, RC, adresa firmei, numărul de telefon și data înființării. Delegații și băncile sunt încărcate în controale de tip ComboBox, astfel se poate selecta fiecare, dar firma nu va reține decât id-ul lor (**Figura 4.11**).

Figura 4.11. Completarea datelor pentru adăugarea unei firme

La salvarea unei firme se apelează aceeași funcție Create, dar în acest caz lista dbObjects va conține informațiile completate despre firmă. Salvarea unei firme este posibilă doar dacă se selectează un delegat și o bancă, acestea fiind chei străine în tabela firmelor. Procedura stocată nu poate să salveze o firmă dacă aceasta nu are asociată niciun delegat sau nicio bancă. Procedura de inserare a unei firme conține următoarea secvență de

Capitolul 4 - Prezentarea aplicației

cod SQL:

```
CREATE Procedure [dbo].[sp_Firme_Insert]
    @Nume nvarchar(50),
    @CUI nvarchar(50),
    @RC nvarchar(50),
    @IBAN nvarchar(50),
    @Adresa nvarchar(50),
    @Telefon nvarchar(50),
    @DataInfiintarii datetime,
    @IdBanca int,
    @IdDelegat int
As
Begin
    Insert Into Firme
        ([Nume],[CUI],[RC],[IBAN],[Adresa],[Telefon],
        [DataInfiintarii],[IdBanca],[IdDelegat])
    Values
        (@Nume,@CUI,@RC,@IBAN,@Adresa,@Telefon,
        @DataInfiintarii,@IdBanca,@IdDelegat)

    Declare @ReferenceID int
    Select @ReferenceID = @@IDENTITY

    Return @ReferenceID
End
```


Capitolul 4 - Prezentarea aplicației

4.3. Introducerea de facturi

Adăugarea unei facturi de intrare se face prin selectarea facturilor de intrare din tree (Figura 4.12).

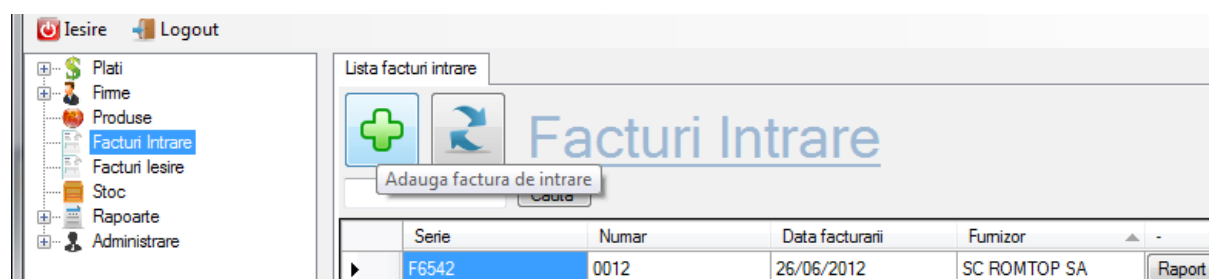


Figura 4.12. Deschiderea listei cu facturile de intrare

Datele necesare pentru o factură de intrare sunt seria, numărul, data facturării și cota TVA, dar pentru a putea adăuga o factură de intrare este nevoie să avem cel puțin o înregistrare pentru firme, respectiv pentru produse (Figura 4.13).

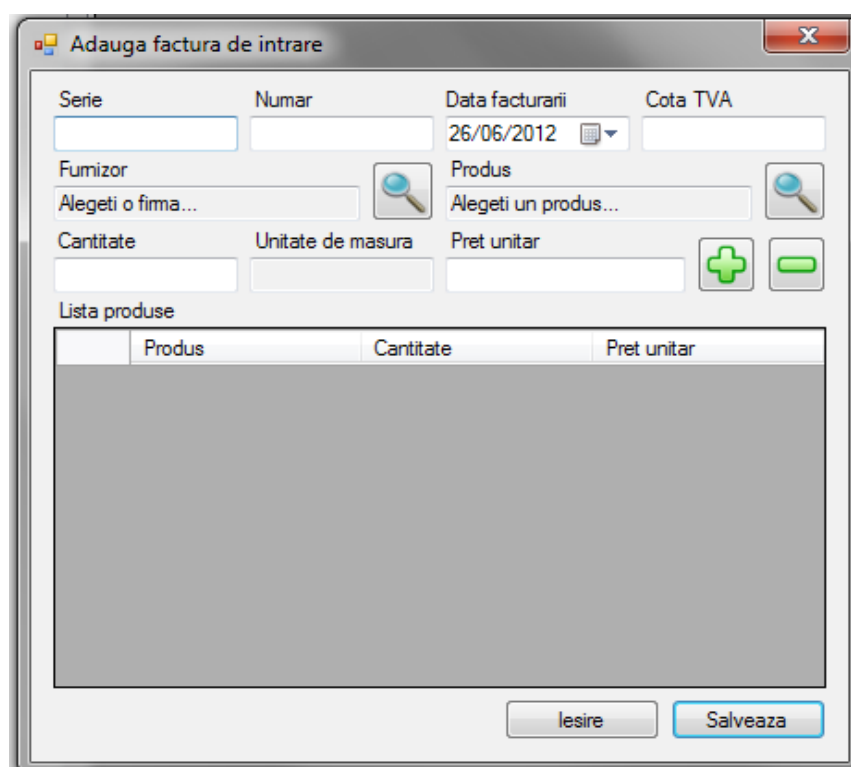


Figura 4.13. Completarea datelor pentru adăugarea unei facturi de intrare

Capitolul 4 - Prezentarea aplicației

Salvarea unei facturi de intrare înseamnă că se creează automat noi stocuri cu cantitățile de produse specificate și cu prețul lor unitar. Procedura stocată ce este apelată din funcția Create este următoarea secvență de cod:

```
Create Procedure [dbo].[sp_FacturiIntrare_Insert]
    @Serie varchar(50),
    @Numar varchar(50),
    @Data datetime,
    @IdFirma int,
    @CotaTva decimal(18,2)
As
Begin
    Insert Into FacturiIntrare
        ([Serie],[Numar],[Data],[IdFirma],[CotaTva])
    Values
        (@Serie,@Numar,@Data,@IdFirma,@CotaTva)

    Declare @ReferenceID int
    Select @ReferenceID = @@IDENTITY

    Return @ReferenceID
End
```

La sfârșitul execuției procedurii stocate se returnează id-ul noii facturi. Legăturile dintre stocuri și factură se face printr-o tabelă intermediară, utilă în momentul în care se dorește aflare furnizorului. Inserarea înregistrării care reprezintă această legătură se face folosind o procedura stocată:

```
CREATE Procedure [dbo].[sp_PozitiiFacturiIntrare_Insert]
    @IdFacturaIntrare int,
    @IdProdus int,
```

Capitolul 4 - Prezentarea aplicației

@Cantitate decimal(18,2),

@PretUnitar decimal(18,2)

As

Begin

Insert Into PozitiiFacturiIntrare

([IdFacturaIntrare],[IdProdus],[Cantitate],[PretUnitar])


Values

((@IdFacturaIntrare,@IdProdus,@Cantitate,@PretUnitar)

Declare @ReferenceID int

Select @ReferenceID = @@IDENTITY

Return @ReferenceID

Adăugarea unui produs și selectarea unui furnizor pe factura de intrare se face prin click pe butonul , care deschide un dialog ce permite căutarea prin toate produsele sau prin toate firmele. (Figura 4.14)

Salvarea stocurilor cu produse se produce odată cu salvarea facturii, prin apelul procedurii:

CREATE Procedure [dbo].[sp_Stoc_Insert]

@IdProdus int,

@IdPozitieFacturaIntrare int,

@Cantitate decimal(18,2)

As

Begin

Insert Into Stoc

([IdProdus],[IdPozitieFacturaIntrare],[Cantitate])

Values

((@IdProdus,@IdPozitieFacturaIntrare,@Cantitate)

Capitolul 4 - Prezentarea aplicației

Declare @ReferenceID int



Select @ReferenceID = @@IDENTITY

Return @ReferenceID

End



Figura 4.14. Formă modală ce permite căutare datelor

După introducerea cantității și a prețului unitar pentru produsul ales acesta se adaugă pe factură folosind butonul , astfel putem adăuga oricâte produse vrem. Iar ștergerea lor, în cazul în care este nevoie de această operație se face folosind butonul . Salvarea facturii înseamnă și inserarea de stocuri noi, stocurile nu se actualizează, ele sunt permanent create, iar cele care se golesc de-a lungul timpului rămân așa pentru o eventuală evidență dacă va fi nevoie.

Adăugare unei facturi de ieșire se face prin deschiderea tabului de facturi ieșire din tree (**Figura 4.15**). Procedura stocată ce se ocupă cu salvarea unei facturi de ieșire este următoarea:

```
ALTER Procedure [dbo].[sp_PozitiiFacturiIesire_Insert]
```

```
    @IdFacturaIesire int,
```

```
    @IdStoc int,
```

Capitolul 4 - Prezentarea aplicației

@Cantitate decimal(18,2),

@PretUnitar decimal(18,2)

As

Begin

Insert Into PozitiiFacturiIesire

([IdFacturaIesire],[IdStoc],[Cantitate],[PretUnitar])

Values

(@IdFacturaIesire,@IdStoc,@Cantitate,@PretUnitar)

Declare @ReferenceID int

Select @ReferenceID = @@IDENTITY

Return @ReferenceID

End

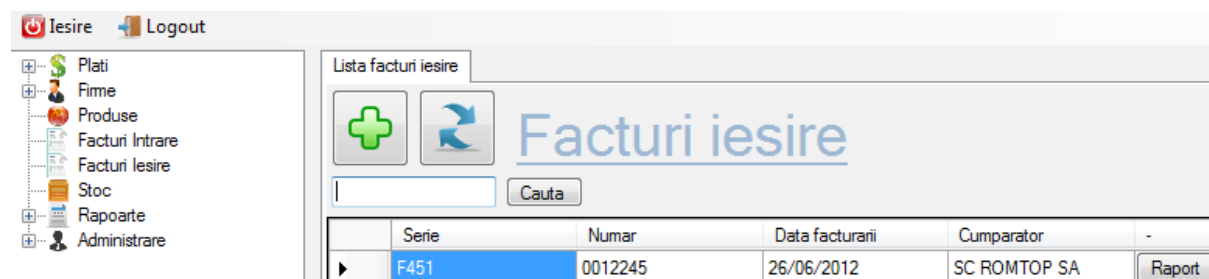



Figura 4.15. Deschiderea listei cu facturile de ieșire

Datele necesare pentru o factură de ieșire sunt seria, numărul, data facturării și cota TVA, dar pentru a putea adăuga o factură de ieșire este nevoie să avem cel puțin o înregistrare pentru firme, respectiv pentru produse (**Figura 4.16**).

Adăugarea unui produs și selectarea unui cumpărător pe factura de ieșire se face la fel ca pentru o factură de intrare, folosind butonul .

Capitolul 4 - Prezentarea aplicației

Adauga factura de iesire

Serie: Numar: Data facturarii: 26/06/2012 Cota TVA:

Cumparator: Alegeti o firma... Produs: Alegeti un produs...

Cantitate disponibila: Cantitate: Pret unitar: + -

Lista produse

Produs	Cantitate	Pret unitar
--------	-----------	-------------

Iesire Salveaza

Figura 4.16. Completarea datelor pentru adăugarea unei facturi de ieșire

După ce se alege un produs se actualizează automat cantitatea disponibilă, se introduce cantitatea ce se vinde, care nu poate fi mai mare decât cantitatea disponibilă și se folosește butonul pentru a adăuga produsul pe factura de ieșire (**Figura 4.17**). Dacă este cazul produsele pot fi scoase folosind butonul .

Cantitatea disponibilă reprezintă o sumă a tuturor cantităților a stocurilor ce au ca și cheie străină produsul ales. Cantitatea disponibilă este calculată folosind LINQ astfel:

```
public decimal CantitateDisponibila()
{
    return (from s in Stoc.GetAll()
            where s.IdProdus == this.ID
            group s by s.NumeProdus into s_nou
            select s_nou.Sum(p => p.Cantitate)).FirstOrDefault();
```

Capitolul 4 - Prezentarea aplicației

}

Stocurile sunt grupate după nume, după care se face o sumă a cantităților din gruparea rezultată.

Salvarea unei facturi de ieșire actualizează automat stocurile, scăzând din ele cantitatea ce se vinde pe factura de ieșire. Dacă se vinde o cantitate mare, iar aceasta trebuie obținută din mai multe stocuri, atunci actualizarea stocurilor se face în ordinea introducerii lor. Modul de lucru cu stocurile este unul de tip FIFO (first in, first out - primul intrat, primul ieșit), adică metoda de actualizarea stocurilor este similară cu metoda în care operează o structură de date de tip *Coadă*. Procedura stocată ce actualizează stocurile este una de tip *Update*:

```
ALTER Procedure [dbo].[sp_Stoc_Update]
    @Id int,
    @IdProdus int,
    @IdPozitieFacturaIntrare int,
    @Cantitate decimal(18,2)
As
Begin
    Update Stoc
    Set
        [IdProdus] = @IdProdus,
        [IdPozitieFacturaIntrare] = @IdPozitieFacturaIntrare,
        [Cantitate] = @Cantitate
    Where
        [Id] = @Id
End
```

Capitolul 4 - Prezentarea aplicației

Serie	Numar	Data facturarii	Cota TVA
F887	00145	26/06/2012	19

Cumparator	Produs
SC ROMTOP SA	Apa

Cantitate disponibila	Cantitate	Pret unitar
1238.96	15	35.00

Produs	Cantitate	Pret unitar
Apa	15.00	35.00

Figura 4.17. Adăugarea unui produs în listă

Legătura dintre stocurile din care au fost scoate produsele și facturile de ieșire pe care au fost vândute este reținută într-o tabelă intermediară, care este transparentă pentru utilizator, dar devine foarte utilă în cazul în care firma care utilizează această aplicație trebuie să ofere informații despre produsele vândute, de exemplu numirea furnizorului pentru un anumit produs vândut dintr-un anumit stoc.

Adăugarea unei entități produce apariția ei automată în tabelul aferent fiecărui tab pentru gestiunea datelor. Există și posibilitatea reîncărcării datelor la comandă folosind butonul încadrat în **Figura 4.18**.

Nume	C.U.I.	Registru Comert	IBAN
SC ROMTOP SA	J90/555/1960	RO 334055	RO0001RBNC123339...

Figura 4.18. Lista de date și butonul de reîncărcare manuală

Capitolul 4 - Prezentarea aplicației

4.4. Vizualizarea stocurilor

Posibilitatea de a vizualiza stocurile reprezintă o necesitate importantă pentru a putea afla în cantități sunt disponibile produsele. Dacă se cunosc cantitățile atunci se poate deduce care sunt produsele care sunt în cantitate mică pentru a se semnaliza necesitatea lor.

Adăugarea de produse în stoc nu se face direct, aceste produse trebuie să provină de undeva, nu este posibil pur și simplu să introducem o cantitate dintr-un anumit produs pe stoc. Crearea stocurilor se face prin odată cu introducerea unei facturi de intrare, care se ocupă cu introducerea de stocuri noi. Operația de adăugare de stoc (**Figura 4.19**) este executată de funcția Create, procedura stocată care este apelată din această funcție este:

```
CREATE Procedure [dbo].[sp_Stoc_Insert]
    @IdProdus int,
    @IdPozitieFacturaIntrare int,
    @Cantitate decimal(18,2)
As
Begin
    Insert Into Stoc
        ([IdProdus],[IdPozitieFacturaIntrare],[Cantitate])
    Values
        (@IdProdus,@IdPozitieFacturaIntrare,@Cantitate)

    Declare @ReferenceID int
    Select @ReferenceID = @@IDENTITY

    Return @ReferenceID
End
```

Practic poziția factură de intrare este o cheie străină care folosește la operația de cuplare cu facturile de intrare.

Capitolul 4 - Prezentarea aplicației

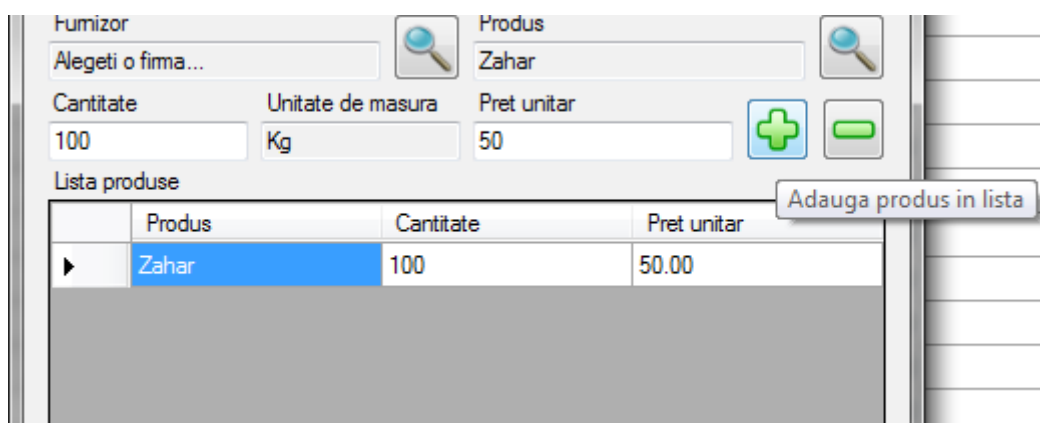


Figura 4.19 Adăugarea unui produs în lista de stocuri

Vânzarea produselor de pe stoc este iarăși o operație indirectă, stocurile nu se pot goli pur și simplu, se scade din stocuri prin intermediul unei facturi de ieșire. Adăugarea unui produs în limita cantității disponibile în lista de produse a unei facturi de ieșire conduce, la salvarea facturii de ieșire, la scăderea cantității de produse vândute din stocurile actuale se apelează funcția Update:

```
public void Update(List<DBObject> dbObjects, int id)
{
    try
    {
        using (SqlConnection con = ConnectionHelper.Connection)
        {
            try
            {
                using (SqlCommand cmd =
                    new SqlCommand(updateStoredProcedureName, con))
                {
                    con.Open();
                    cmd.CommandType = System.Data.CommandType.StoredProcedure;
                    if (id != 0) cmd.Parameters.AddWithValue("@Id", id);
                    foreach (var dboObject in dbObjects)
```

Capitolul 4 - Prezentarea aplicației

```
        {
            cmd.Parameters.AddWithValue(dboObject.Name, dboObject.Value);
        }
        cmd.ExecuteNonQuery();
        con.Close();
    }
}
catch (Exception ex)
{
    throw new Exception(StringProcedureFail + updateStoredProcedureName, ex);
}
}
}
catch (Exception ex)
{
    throw new Exception(StringDatabaseFail, ex);
}
}
```

Această funcție apelează procedura stocată pentru update din baza de date, care actualizează datele din stocul din care s-au vândut produsele. Mai importantă este procedura stocată care salvează legătura stocului cu tabela facturilor de ieșire:

```
CREATE Procedure [dbo].[sp_PozitiiFacturiIesire_Insert]
```

```
    @IdFacturaIesire int,
```

```
    @IdStoc int,
```

```
    @Cantitate decimal(18,2),
```

```
    @PretUnitar decimal(18,2)
```

```
As
```

```
Begin
```

```
    Insert Into PozitiiFacturiIesire
```

Capitolul 4 - Prezentarea aplicației

```
        ([IdFacturaIesire],[IdStoc],[Cantitate],[PretUnitar])
Values
        (@IdFacturaIesire,@IdStoc,@Cantitate,@PretUnitar)

Declare @ReferenceID int
Select @ReferenceID = @@IDENTITY

Return @ReferenceID

End
```

Poziția facturii de ieșire reține stocul din care s-au vândut produsele, factura asociată acestei vânzări, cantitatea produsului vândut și prețul unitar pe produs. Iar printr-o interogare ce implică două operații de cuplare cu tabela facturilor de ieșire și cu tabela stocurilor se poate afla ușor stocul din care s-au vândut produsele pentru o anumită factură.

Funcția de salvare a unei facturi de ieșire primește ca parametru o listă de poziții pentru factura de ieșire, pentru fiecare poziție în parte se actualizează stocul aferent poziție respective. Salvarea este o tranzacție, deoarece nu se permite salvarea doar a unei părți dintr-o factură. Se încercă salvarea întregii facturi, dacă nu se reușește se dă un mesaj corespunzător și se face rollback.

```
public PersistenceResult Save(List<PozitieFacturaIesire> pozitiiList)
{
    var result = new PersistenceResult();
    try
    {
        if (pozitiiList.Count == 0)
        {
            return new PersistenceResult
            {
                Status = Enums.StatusEnum.Errors,
```

Capitolul 4 - Prezentarea aplicației

```
        Message = "Nu ati adaugat niciun produs in factura!"
    };
}
using (TransactionScope scope = new TransactionScope())
{
    var facturaPR = this.Save(); // salvez factura
    if (facturaPR.Status == Enums.StatusEnum.Errors)
    {
        throw new Exception(facturaPR.Message, facturaPR.ExceptionOccurred);
    }
    var facturaId = this.ID;
    foreach (var item in pozitiiList) // se adaugă fiecare poziție și se face update la
stoc
    {
        var stocPR = item.StocObject.Save();
        if (stocPR.Status == Enums.StatusEnum.Errors)
        {
            throw new Exception(stocPR.Message, stocPR.ExceptionOccurred);
        }
        item.IdFacturaIesire = facturaId;
        var pozitiePR = item.Save();
        if (pozitiePR.Status == Enums.StatusEnum.Errors)
        {
            throw new Exception(pozitiePR.Message, pozitiePR.ExceptionOccurred);
        }
    }

    scope.Complete();
}
result.Message = StringSaveSuccess;
result.Status = Enums.StatusEnum.Saved;
```

Capitolul 4 - Prezentarea aplicației

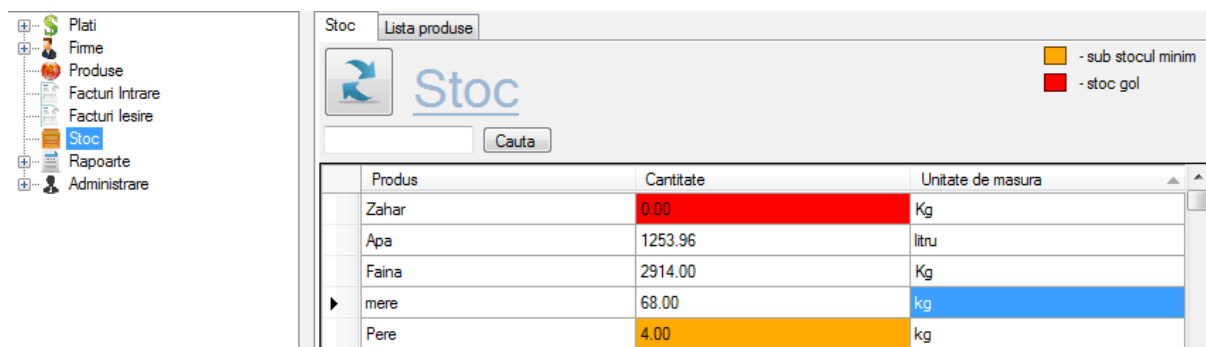
```
    }  
    catch (Exception ex)  
    {  
        result.Message = StringSaveFail;  
        result.Status = Enums.StatusEnum.Errors;  
        result.ExceptionOccurred = ex;  
    }  
    return result;  
}
```

Vizualizarea stocurilor se face din tree-ul aplicației (**Figura 4.20.**), pentru utilizator nu prezintă interes să poată vizualiza aceleași produse în mai multe stocuri. Astfel înainte de afișarea stocurilor, ele trebuie să treacă printr-o operație care pregătește datele. Funcția care se ocupă se numște `GetAllGroupedByProdus` și primește ca parametru numele unui produs, parametru care este opțional, dacă se dorește aflarea cantității unui anumit produs din stoc.

```
public static List<Stoc> GetAllGroupedByProdus(string produsName = "")  
{  
    var query = from s in Stoc.GetAll()  
        where s.NumeProdus.ToLower().Contains(produsName.ToLower())  
        group s by s.IdProdus into s_nou  
        select new Stoc  
        {  
            ID = s_nou.Max(p => p.ID),  
            Cantitate = s_nou.Sum(p => p.Cantitate),  
            IdPozitieFacturaIntrare =  
                s_nou.Select(p => p.IdPozitieFacturaIntrare).First(),  
            IdProdus = s_nou.Select(p => p.IdProdus).First()  
        };  
    return query.ToList();  
}
```

Capitolul 4 - Prezentarea aplicației

Funcția `GetAllGroupedByProduce` grupează toate stocurile pe produse, iar pentru fiecare grup rezultat se face însumarea cantităților stocurilor, astfel fiecare grup devine un stoc nou în care cantitatea este cantitatea totală pe produs.



Produce	Cantitate	Unitate de masura
Zahar	0.00	Kg
Apa	1253.96	litru
Faina	2914.00	Kg
mere	68.00	kg
Pere	4.00	kg

Figura 4.20. Vizualizarea stocurilor

Fiecare produs are un atribut numit stoc minim care precizează care este stocul minim recomandat pentru un produs. În **Figura 4.20** stocurile fiind grupate pe produse se poate printr-o operație de cuplare cu tabela de produse să se afle care este cantitatea minimă și să se compare cu stocul curent, utilitatea este exemplificată în **Figura 4.20** prin colorarea celulelor în funcție de cantitate:

```
if (stocGrupat.ProdusObject.StocMinim > stocGrupat.Cantitate)
{
    if (stocGrupat.Cantitate == 0)
    {
        e.CellStyle.BackColor = Color.Red;
        e.CellStyle.SelectionBackColor = Color.DarkRed;
    }
    else
    {
        e.CellStyle.BackColor = Color.Orange;
        e.CellStyle.SelectionBackColor = Color.DarkOrange;
    }
}
```

Capitolul 4 - Prezentarea aplicației

4.5. Efectuarea plăților

Facturile au scopul de a vinde sau cumpăra bunuri, dar în momentul în care se eliberează o factură ea nu se înregistrează odată cu plata sa, astfel achitarea facturilor devine o componentă individuală. Pentru a putea plăti o factură se deschide din tree-ul aplicației *Plăți* ca în **Figura 4.21**. Deschiderea tabului de efectuarea a plăților deschide un control nou în care se poate alege tipul de facturi ce se pot plăti (**Figura 4.21**).

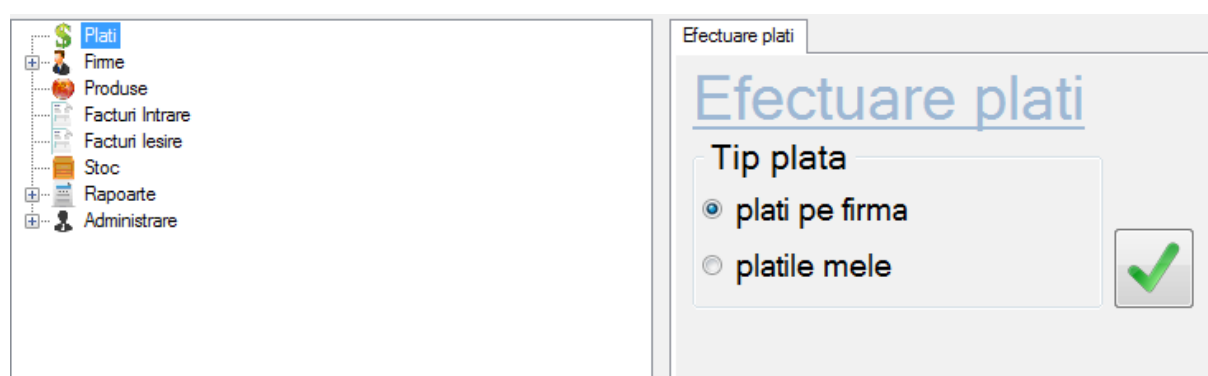



Figura 4.21. Efectuarea plăților

Alegerea tipului de plată se face folosind  butonul .

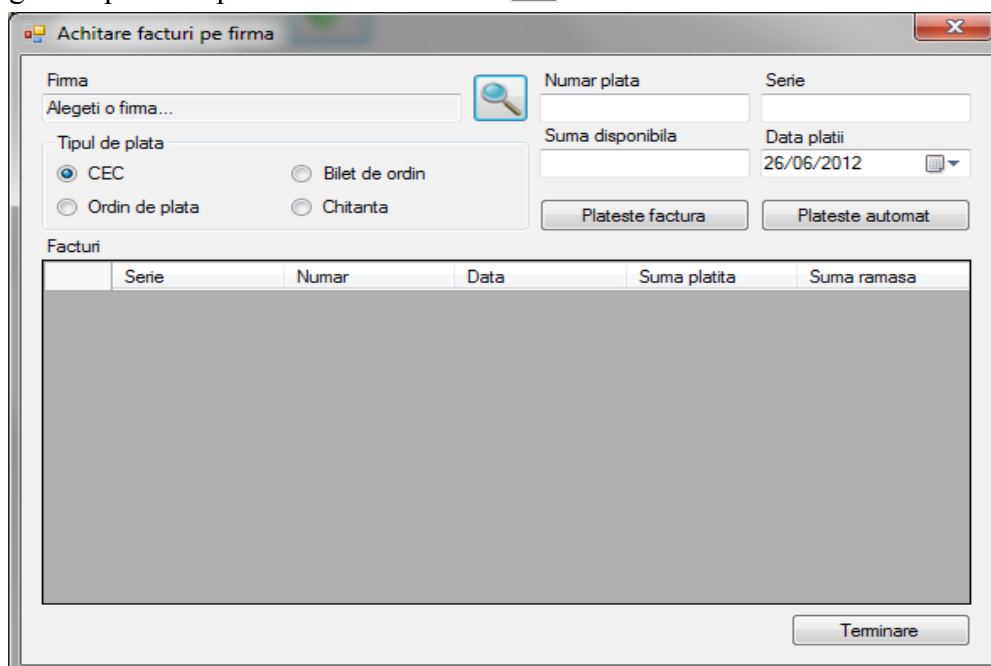


Figura 4.22. Achitarea facturilor pe firmă

Capitolul 4 - Prezentarea aplicației

Plăți pe firmă reprezintă plățirea facturilor de ieșire, această alegerea implică o filtrare în rândul facturilor de ieșire în funcție de firmă. Alegerea firmei se face folosind butonul de căutare, iar în funcție de firma aleasă se încarcă toate facturile împreună cu sumele plătite și restul de plată.

Fiecare factură de ieșire are trei proprietăți ajutătoare care calculează suma plătită, suma totală de plătit și suma rămasă de plătit, acestea sunt:

```
public decimal SumaPlatita
{
    get
    {
        return (SumaTotalaDePlatit - SumaRamasaDePlatit);
    }
}

public decimal SumaTotalaDePlatit
{
    get
    {
        var result = from p in PozitieFacturaIesire.GetAll()
                      where p.IdFacturaIesire == this.ID
                      select (p.PretUnitar * p.Cantitate);
        var pretFaraTva = result.Sum(p => p);
        return (pretFaraTva * this.CotaTva / 100) + pretFaraTva;
    }
}

public decimal SumaRamasaDePlatit
{
    get
    {
```

Capitolul 4 - Prezentarea aplicației

```
var sumaTotala = SumaTotalaDePlatit;  
var list = from p in PlataFactura.GetAll()  
           where p.IdFactura == this.ID && p.TipFactura == "Iesire"  
           select p.PlataObject.Suma;  
var sumaPlatita = list.Sum(p => p);  
return (sumaTotala - sumaPlatita);  
}  
}
```

Suma totală de plătit pe o factură se calculează cautând în pozițiile facturilor de ieșire toate rândurile care au legătură cu factura căutată. Găsind toate pozițiile facturii de ieșire se poate calcula suma de plătit înmulțind prețul unitar cu cantitatea vândută, rezultatul îl adunăm la suma de plătit pe factură. Suma plătită pe factură se calculează cu o operație de cuplare între plată pe factură și factură. Din această cuplare obținem toate plățile pentru o factură, iar însumarea acestor plăți reprezintă suma plătită.

Plata unei facturi se poate face automat sau manual. Plata manuală necesită introducerea unei sume pentru plățirea unei facturi. Iar pe măsură ce facturile se plătesc această sumă scade. Plata automată are următoare secvență de cod:

```
while (sumaDisponibila > 0)  
{  
    var factura = FacturaIesire.GetAll().OrderBy(p => p.Data).Where(p => p.IdFirma  
== SelectedFirma.ID && p.SumaRamasaDePlatit > 0).FirstOrDefault();  
    if (factura == null) return;  
    if (!PlatesteFactura(factura)) return;  
}
```

Cât timp mai sunt bani disponibili (adică suma introdusă pentru efectuarea plăților este mai mare decât zero) și cât timp mai sunt facturi de plătit se caută facturile în ordinea crescătoare a datelor în care au fost eliberate și se achită una câte una. Dacă suma este mai mare decât suma rămasă de plătit pe factură atunci factura se achită integral, iar dacă suma

Capitolul 4 - Prezentarea aplicației

este mai mică decât suma rămasă de plătit pe factură atunci se plătește doar o parte din factura de ieșire. Dacă o plată nu se efectuează din motive tehnice atunci nu se scade din sumă și se semnalizează cu un mesaj corespunzător.

Alegerea tipului de plată *Plățile mele* deschide un dialog care permite plățirea facturilor de intrare (**Figura 4.23**).

	Serie	Numar	Data	Suma platita	Suma ramasa
▶	F6542	0012	26/06/2012	0.00	2,952.00

Figura 4.23. Plătirea facturilor de intrare

Datele necesare pentru o plată sunt: număr plată, serie și data plății. Facturile se încarcă automat la deschiderea dialogului. Factura de intrare are aceleași trei proprietăți ajutătoare ca și factura de ieșire: suma plătită, suma totală de plătit și suma rămasă de plătit. De fapt funcționalitate plătirii facturilor de intrare este asemănătoare cu cea a plătirii facturilor de ieșire, doar că facturile de intrare nu mai necesită nicio filtrare. Toate facturile de intrare se înregistrează pe aceeași firmă, și anume firma care utilizează această aplicație.

Capitolul 4 - Prezentarea aplicației

Următoarea secvență de cod ilustrează diferența între plățirea unei facturi de ieșire și plățirea unei facturi de intrare:

```
if (facturaIntrareBound == null) return;
    while (sumaDisponibila > 0)
    {
        var factura = FacturaIntrare.GetAll().OrderBy(p => p.Data).Where(p =>
p.SumaRamasaDePlatit > 0).FirstOrDefault();
        if (factura == null) return;
        if (!PlatesteFactura(factura)) return;
    }
```

Principiul este același: cât timp mai am bani cu care să plătesc o factură și mai am facturi de plătit atunci se plătesc facturile în ordinea crescătoare a introducerii lor.

Achitarea facturilor de intrare și de ieșire se fac în mod similar folosind funcția AchitareFactura, care nu primește niciun parametru deoarece achită factura din care se apelează:

```
var pr = plataPeFactura.Save();
if (pr.Status == Enums.StatusEnum.Errors)
    throw new Exception(pr.Message, pr.ExceptionOccurred);
var plataFactura = new PlataFactura
{
    IdFactura = this.ID,
    IdPlata = plataPeFactura.ID,
    TipFactura = "Iesire"
};
pr = plataFactura.Save();
if (pr.Status == Enums.StatusEnum.Errors)
    throw new Exception(pr.Message, pr.ExceptionOccurred);
}
```

Dacă plata nu se poate salva atunci se aruncă o excepție care este tratată și se afișează un mesaj corespunzător.

Capitolul 4 - Prezentarea aplicației

4.6. Generarea rapoartelor

Aplicația permite generarea rapoartelor pe baza datelor introduse de-a lungul timpului. Aceste date sunt preluate și prelucrate și dispuse în diferite moduri. Aplicația dispune de șase rapoarte, acestea sunt: evoluția vânzării pe un anumit produs de-a lungul timpului (*Evoluție vânzări pe produs*), situație generală a firmelor care nu și-au plătit toate facturile (*Situație plăți pe firme*), evidența produselor care sunt sub stocul minim (*Produse sub stoc minim*), o situație a facturilor de intrare neplătite (*Facturi de intrare neachitate*), evidența facturilor de ieșire neplătite pe firme - firmele "rău-platnice" (*Facturi de ieșire neachitate pe firmă*) și un raport care reprezintă practic factura de intrare sau de ieșire eliberată.

Toate rapoartele, mai puțin cel care reprezintă o factură, se pot accesa din tree-ul aplicației (**Figura 4.24**). Raportul pentru factură se generează din listele cu facturi și automat la adăugare unei noi facturi, de intrare sau de ieșire.

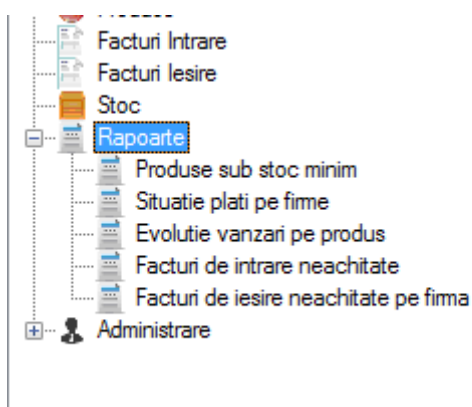


Figura 4.24. Rapoartele aplicației

Raportul *Produse sub stoc minim*, după cum sugerează numele afișează un raport cu toate a căror cantitate pe stoc este sub stocul minim (**Figura 4.25**).

Capitolul 4 - Prezentarea aplicației

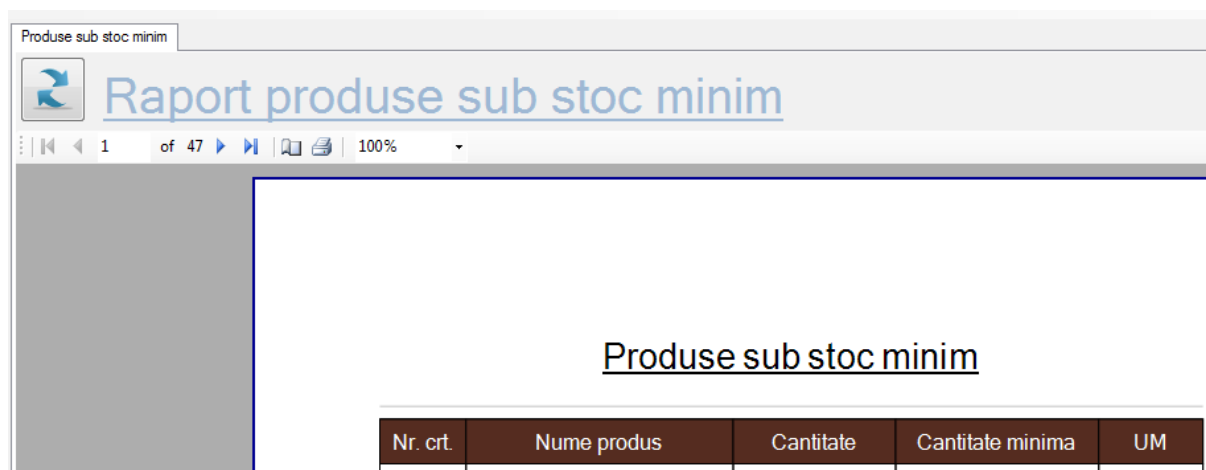



Figura 4.25. Raportul produse sub stoc minim

Generarea raportului se face apăsând butonul , funcționalitatea acestui buton se aplică pentru toate rapoartele, după care se pot vizualiza toate produsele în cantitățile lor din stoc. Utilitatea acestui raport se observă în momentul în care se dorește aflarea produselor care se află în cantități mici, pentru a mări stocul lor. Codul care se execută pentru generarea raportului este:

```
var stocList = Stoc.GetAllGroupedByProdus().Where(p => p.Cantitate <
                                                    p.ProdusObject.StocMinim);

var report = new StocReport();
report.objectDataSource.DataSource = stocList;
```

Sursa de date pentru acest raport este reprezentată de lista stocurilor grupate pe produse, dar cu condiția ca stocul minim să fie mai mare decât cantitatea din stoc.

Situație plăți pe firme (Figura 4.26) este un raport care prezintă toate firmele împreună cu totalul de plată. Acest total de plată se obține însumând toate facturile de ieșire pentru o anumită firmă. Secvența de cod care generează acest raport este:

```
foreach (var firma in Firma.GetAll())
{
    var facturi = FacturaIesire.GetAll().Where(p => p.IdFirma == firma.ID &&
```

Capitolul 4 - Prezentarea aplicației

```
p.SumaRamasaDePlatit > 0).ToList();  
    var sumaPlatita = facturi.Sum(p => p.SumaPlatita);  
    var sumaRamasa = facturi.Sum(p => p.SumaRamasaDePlatit);  
    listaDatorii.Add(new DatorieFirma  
    {  
        NumeFirma = firma.Nume,  
        SumaPlatita = sumaPlatita.ToString("0.00"),  
        SumaRamasa = sumaRamasa.ToString("0.00")  
    });  
}
```

Se parcurg toate firmele din baza de date și pentru fiecare în parte se află facturile de ieșire și se însumează suma rămasă de plătit și suma plătită.



Situatie plati pe firme

Raport situatie plati pe firme

1 of 1 100%

Situatie plati pe firme

Nr. crt.	Nume Firma	Suma Platita	Suma Ramasa
1	SC ROMTOP SA	1007.77	42813579.31

Figura 4.26 Raportul situație plăți pe firme

Raportul *Evoluție vânzări pe produs* este practic un grafic (**Figura 4.27**) care primește ca și sursă de date o serie temporală ce conține data vânzării produsului și cantitatea de produs care s-a vândut în data respectivă. Utilitatea acestui raport se evidențiază în detectarea produselor care "se vând bine" și nu doar atât ci și perioada în care vânzarea produsului a crescut sau a scăzut, fapt ce interesează pe orice vânzător.

Capitolul 4 - Prezentarea aplicației

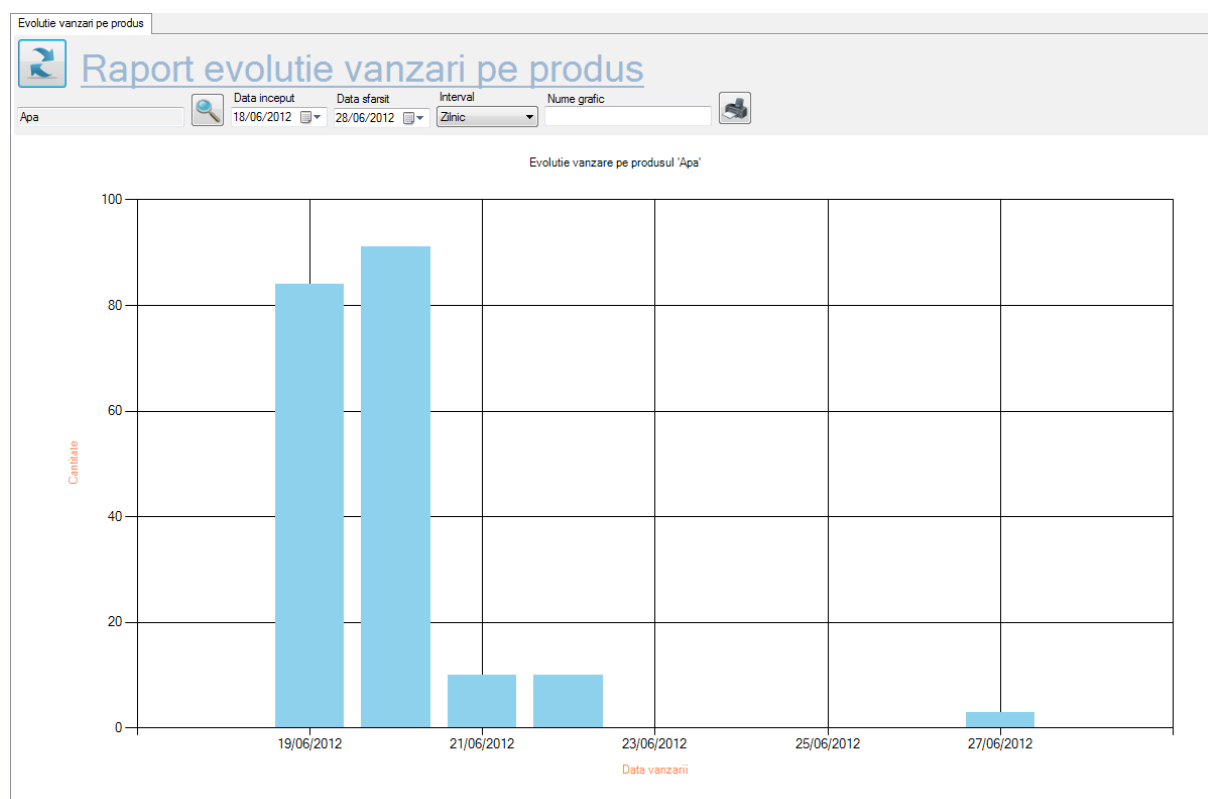



Figura 4.27. Raportul evoluție vânzări pe produs

Acest raport necesită un produs, care se poate căuta folosind butonul , are nevoie de un interval de timp care se poate alege ca în **Figura 4.28**, și permite gruparea rezultatelor pe zile sau pe luni.

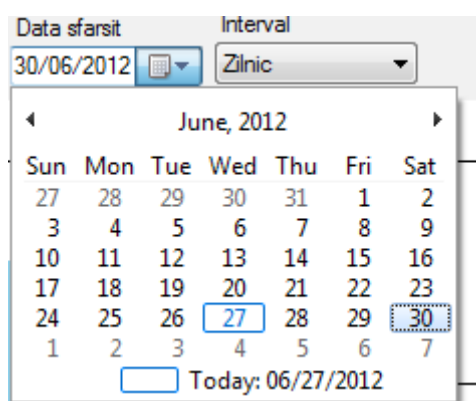


Figura 4.28. Alegerea zilei pentru data de sfârșit

Alegerea intervalului de timp determină perioada pe care se va afișa graficul. Secvența

Capitolul 4 - Prezentarea aplicației


de cod care generează acest grafic este:

```
var vanzariPeProdot = new List<VanzareProdot>();
while (beginDate.Date <= endDate.Date)
{
    var result = from poz in PozitieFacturaIesire.GetAll()
        where
            poz.FacturaIesireObject.Data.ToString(comparer) == beginDate.Date.ToString(comparer)
            && poz.StocObject.IdProdot == produs.ID
        group poz by poz.FacturaIesireObject.Data.ToString(comparer)
        into poz_nou
        select new VanzareProdot
        {
            Cantitate = poz_nou.Sum(p => p.Cantitate).ToString("0.00"),
            DataVanzarii = beginDate.Date.ToString(comparer)
        };
    VanzareProdot vanzare = null;
    vanzare = result.FirstOrDefault() == null ? new VanzareProdot { Cantitate = "0",
        DataVanzarii = beginDate.Date.ToString(comparer) } : result.FirstOrDefault();
    vanzariPeProdot.Add(vanzare);
    switch (interval)
    {
        case IntervalType.Zilnic:
            beginDate = beginDate.AddDays(1);
            break;
        case IntervalType.Lunar:
            beginDate = beginDate.AddMonths(1);
            break;
    }
}
```

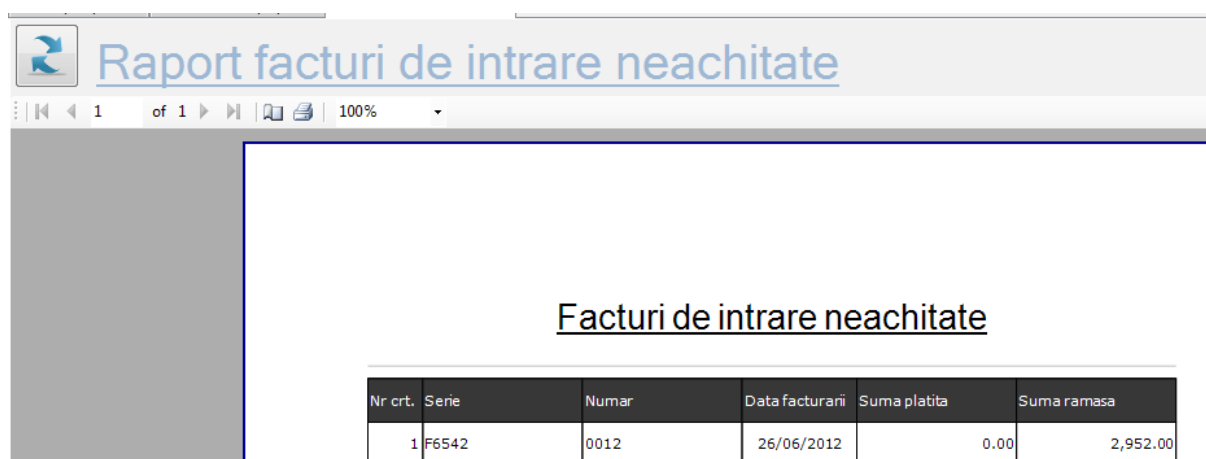
Generarea raportului funcționează astfel: cât timp timp data de început nu a depășit data de sfârșit a raportului se selectează toate facturile de ieșire din data respectivă și se

Capitolul 4 - Prezentarea aplicației

Însumează toate cantitățile cu produsul ales care s-au vândut în acea dată, după care se mărește data de început cu o zi și se reia de la început.

După generarea graficului putem introduce un nume pentru grafic după care acesta poate fi printat folosind butonul .

Raportul *Facturi de intrare neachitate* (**Figura 4.29**) este un raport ce enumeră toate facturile de intrare care nu au fost achitate integral.



Nr crt.	Serie	Numar	Data facturarii	Suma platita	Suma ramasa
1	F6542	0012	26/06/2012	0.00	2,952.00

Figura 4.29. *Raportul facturi de intrare neachitate*

Secvența de cod ce generează acest raport este:

```
var raport = new DatoriiReport();  
raport.FacturiIntrareDataSource.DataSource =  
    FacturaIntrare.GetAll().Where(p => p.SumaRamasaDePlatit > 0).ToList();
```

Generarea raportului se face prin selectarea tuturor facturilor de intrare care au suma rămasă de plătit mai mare decât zero. Acest raport este util în momentul în care firma ce utilizează această aplicație vrea să afle ce facturi mai are de achitat.

Raportul *Facturi neachitate pe firmă* (**Figura 4.30**) este un raport care listează toate facturile de ieșire care nu sunt plătite integral de către o firmă. Acest raport este similar cu raportul *Facturi de intrare neachitate* deoarece amândouă listează facturile care nu sunt plătite pe o firmă, diferența este că raportul *Facturi de intrare neachitate* se referă la o firmă care utilizează aplicația.

Capitolul 4 - Prezentarea aplicației

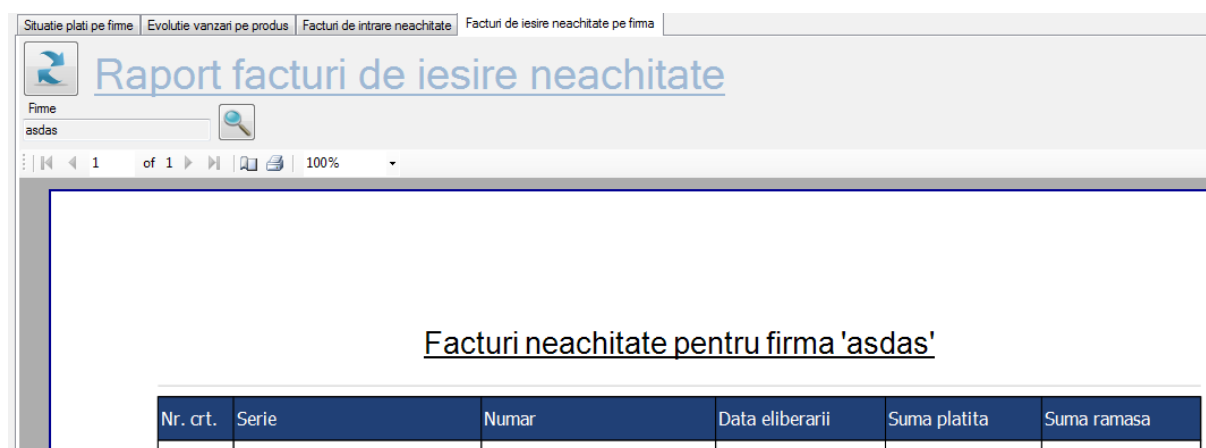


Figura 4.30. Raportul facturi de ieșire neachitate

Generarea acestui raport necesită alegerea unei firme, după cum îi sugerează și numele, iar secvența de cod care generează raportul este:

```
var facturi = FacturaIesire.GetAll().Where(p => p.SumaRamasaDePlatit > 0  
                                     && p.IdFirma == SelectedFirma.ID);
```

Se selectează toate facturile care au suma rămasă de plătit mai mare decât zero, dar cu condiția să fie pe firma selectată.

Raport care reprezintă factura nu se găsește în același loc cu celelalte rapoarte, el poate fi generat din lista de facturi ieșire sau facturi intrare folosind butonul din **Figura 4.31**. Raportul cu factura (**Figura 4.32**) se generează automat și la salvarea unei facturi de intrare sau de ieșire.

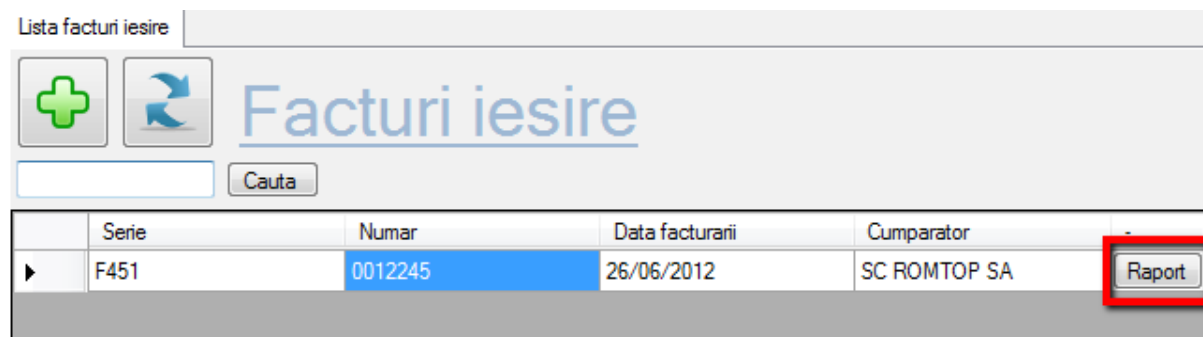


Figura 4.31. Butonul de generare raport factură

Capitolul 4 - Prezentarea aplicației

Furnizor SC ROMTOP SA C.U.I.: J90/555/1960 Reg. com.: RO 334055 Adresa: Bacau str. Aurel Vlaicu nr. 3 IBAN: RO0001RBNC1233390011 Banca: Banc Post	Cumparator SC ROMTOP SA C.U.I.: J90/555/1960 Reg. com.: RO 334055 Adresa: Bacau str. Aurel Vlaicu nr. 3 IBAN: RO0001RBNC1233390011 Banca: Banc Post																																																																								
<h3 style="margin: 0;">FACTURA FISCALA</h3> <p style="margin: 5px 0;">SERIE : F5564789</p> <p style="margin: 5px 0;">NUMAR : 002157</p> <p style="margin: 5px 0;">DATA : 28/06/2012</p> <p style="margin: 10px 0;">Cota TVA: 24 %</p> <table border="1" style="width: 100%; border-collapse: collapse; text-align: center;"> <thead> <tr> <th style="width: 10%;">Nr. crt.</th> <th style="width: 30%;">Denumire produs</th> <th style="width: 10%;">U.M.</th> <th style="width: 15%;">Cantitate</th> <th style="width: 15%;">Pret Unitar</th> <th style="width: 20%;">Valoare</th> </tr> </thead> <tbody> <tr><td>1</td><td>Zahar</td><td>Kg</td><td>10.00</td><td>123.00</td><td>123.00</td></tr> <tr><td>2</td><td>Zahar</td><td>Kg</td><td>12.00</td><td>123.00</td><td>123.00</td></tr> <tr><td>3</td><td>Zahar</td><td>Kg</td><td>12.00</td><td>123.00</td><td>123.00</td></tr> <tr><td>4</td><td>Zahar</td><td>Kg</td><td>12.00</td><td>123.00</td><td>123.00</td></tr> <tr><td>5</td><td>Zahar</td><td>Kg</td><td>4.00</td><td>123.00</td><td>123.00</td></tr> <tr><td>6</td><td>Apa</td><td>litru</td><td>25.36</td><td>35.00</td><td>35.00</td></tr> <tr><td>7</td><td>Apa</td><td>litru</td><td>24.64</td><td>35.00</td><td>35.00</td></tr> <tr><td>8</td><td>Faina</td><td>Kg</td><td>50</td><td>123.00</td><td>123.00</td></tr> <tr> <td colspan="3" style="text-align: left; padding-left: 5px;">Stampila</td> <td>3,372.00</td> <td>14,050.00</td> <td>17,422.00</td> </tr> <tr> <td colspan="3"></td> <td>Valoare TVA</td> <td>Total fara TVA</td> <td>Total general</td> </tr> <tr> <td colspan="4" style="text-align: left; padding-left: 5px;">Nume delegat: 155842</td> <td colspan="2" style="text-align: left; padding-left: 5px;">Semnatura</td> </tr> </tbody> </table>		Nr. crt.	Denumire produs	U.M.	Cantitate	Pret Unitar	Valoare	1	Zahar	Kg	10.00	123.00	123.00	2	Zahar	Kg	12.00	123.00	123.00	3	Zahar	Kg	12.00	123.00	123.00	4	Zahar	Kg	12.00	123.00	123.00	5	Zahar	Kg	4.00	123.00	123.00	6	Apa	litru	25.36	35.00	35.00	7	Apa	litru	24.64	35.00	35.00	8	Faina	Kg	50	123.00	123.00	Stampila			3,372.00	14,050.00	17,422.00				Valoare TVA	Total fara TVA	Total general	Nume delegat: 155842				Semnatura	
Nr. crt.	Denumire produs	U.M.	Cantitate	Pret Unitar	Valoare																																																																				
1	Zahar	Kg	10.00	123.00	123.00																																																																				
2	Zahar	Kg	12.00	123.00	123.00																																																																				
3	Zahar	Kg	12.00	123.00	123.00																																																																				
4	Zahar	Kg	12.00	123.00	123.00																																																																				
5	Zahar	Kg	4.00	123.00	123.00																																																																				
6	Apa	litru	25.36	35.00	35.00																																																																				
7	Apa	litru	24.64	35.00	35.00																																																																				
8	Faina	Kg	50	123.00	123.00																																																																				
Stampila			3,372.00	14,050.00	17,422.00																																																																				
			Valoare TVA	Total fara TVA	Total general																																																																				
Nume delegat: 155842				Semnatura																																																																					

Figura 4.32. Model de factură

Pentru generarea raportului pentru o factură, acest raport are nevoie, ca sursă de date, de factura și toate pozițiile de intrare sau ieșire care s-au generat pe parcursul adăugării produselor pe factură.

Capitolul 4 - Prezentarea aplicației

4.7. Utilizatori. Roluri. Drepturi

Aplicația dispune și de o parte de administrare. Intrare în aplicație se face pe baza unei autentificări (**Figura 4.31**).

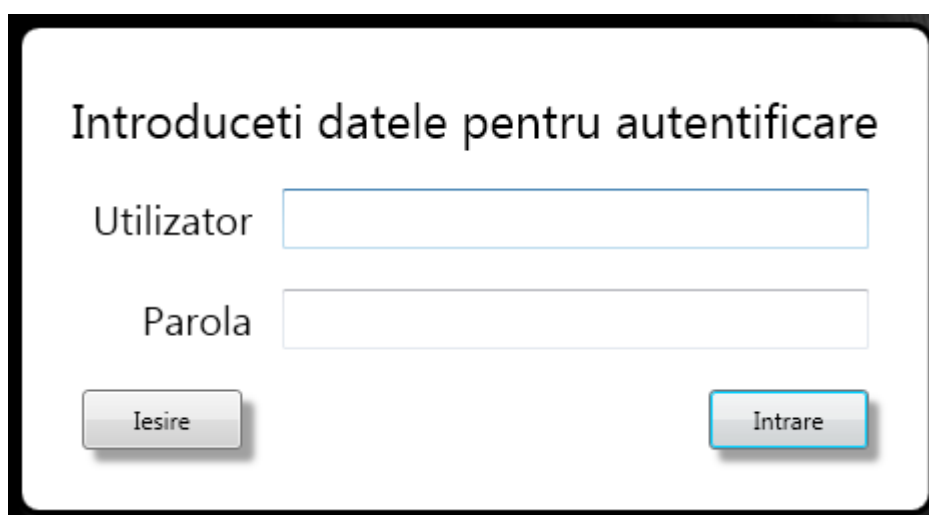
Forma de autentificare a aplicației este prezentată într-un cadru negru. În interior, titlul "Introduceti datele pentru autentificare" este scris în negru. Sub titlu, sunt două câmpuri de text: primul este etichetat "Utilizator" și al doilea "Parola". În partea de jos a formei, există două butoane: "Iesire" (în stânga) și "Intrare" (în dreapta). Butoanele au un aspect 3D cu umbre.

Figura 4.31. *Introducerea datelor pentru autentificare*

Persoana care dorește să folosească aplicația trebuie să dispună de un cont, acest lucru facilitează folosirea organizată a aplicației. Introducerea de date greșite se semnalizează cu un mesaj corespunzător. Pentru început nu va exista niciun utilizator care să acceseze aplicația, din acest motiv există un utilizator special *admin* (administrator) care nu poate fi șters sau modificat, parola sa inițială fiind 'admin', însă aceasta se poate schimba cu condiția confirmării parolei vechi.

Autentificarea corectă înseamnă că utilizatorul poate să folosească aplicația. În tree-ul aplicației la secțiune de administrare putem deschide setările aplicației, lista cu utilizatorii și rolurile aplicației (**Figura 4.32**).

Capitolul 4 - Prezentarea aplicației

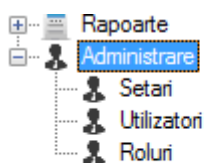


Figura 4.32. Opțiunile de administrare

Vizualizare rolurilor de face deschizând rolurile din tree-ul aplicației (**Figura 4.33**):

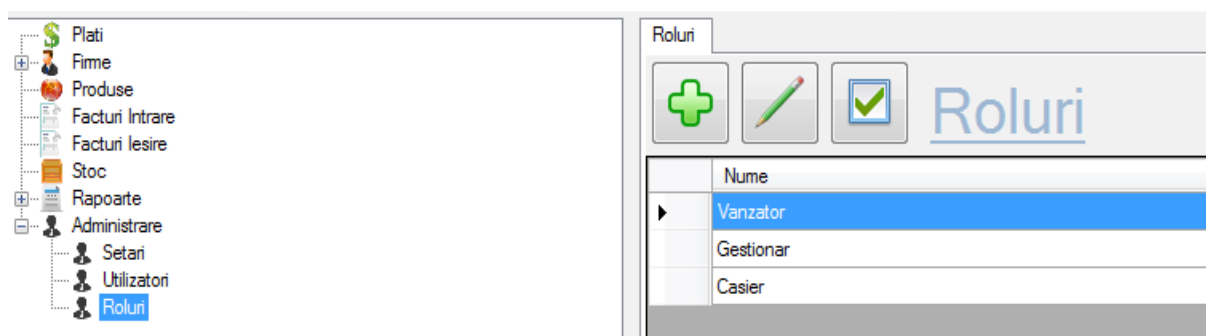


Figura 4.33. Deschiderea listei cu roluri

Înainte de adăugarea unui utilizator pentru aplicație trebuie să definim un rol. Acest rol constă dintr-un nume și o descriere (**Figura 4.34**). Salvarea rolului se face prin procedura stocată:

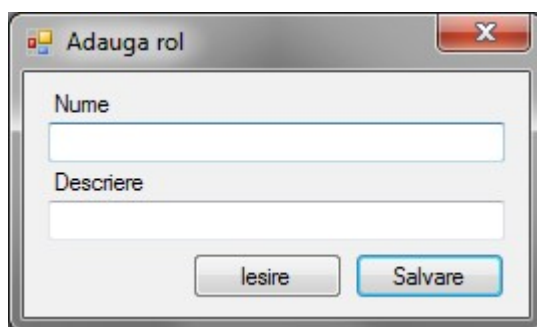
```
CREATE Procedure [dbo].[sp_Roluri_Insert]
    @Nume nvarchar(50),
    @Descriere nvarchar(50)
As
Begin
    Insert Into Roluri
        ([Nume],[Descriere])
    Values
        (@Nume,@Descriere)

    Declare @ReferenceID int
    Select @ReferenceID = @@IDENTITY
```

Capitolul 4 - Prezentarea aplicației


Return @ReferenceID

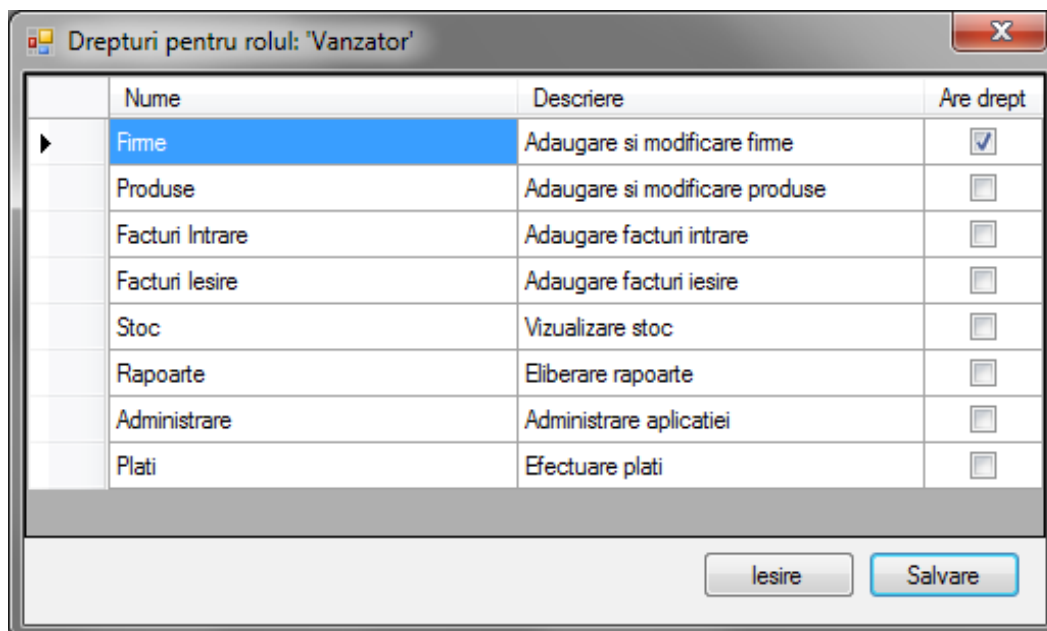
End



Dialog box titled "Adauga rol" (Add role). It contains two text input fields: "Nume" (Name) and "Descriere" (Description). Below the fields are two buttons: "Iesire" (Exit) and "Salvare" (Save).

Figura 4.34. Adăugarea unui rol

Rolul este important pentru că acestuia i se asociază drepturi, astfel limitându-se accesul utilizatorului la aplicație. Alegerea drepturilor pentru un rol se face folosind butonul  și se deschide un dialog cu toate drepturile și activarea lor (**Figura 4.35**).



Dialog box titled "Drepturi pentru rolul: Vanzator" (Rights for role: Vanzator). It displays a table of permissions with columns: Nume, Descriere, and Are drept. The 'Firme' row is selected, and its 'Are drept' checkbox is checked. Other rows include 'Produce', 'Facturi Intrare', 'Facturi Iesire', 'Stoc', 'Rapoarte', 'Administrare', and 'Plati'. At the bottom are buttons for 'Iesire' (Exit) and 'Salvare' (Save).

Nume	Descriere	Are drept
Firme	Adaugare si modificare firme	<input checked="" type="checkbox"/>
Produce	Adaugare si modificare produse	<input type="checkbox"/>
Facturi Intrare	Adaugare facturi intrare	<input type="checkbox"/>
Facturi Iesire	Adaugare facturi iesire	<input type="checkbox"/>
Stoc	Vizualizare stoc	<input type="checkbox"/>
Rapoarte	Eliberare rapoarte	<input type="checkbox"/>
Administrare	Administrare aplicatiei	<input type="checkbox"/>
Plati	Efectuare plati	<input type="checkbox"/>

Figura 4.35. Modificarea drepturilor unui rol

Capitolul 4 - Prezentarea aplicației

Încercarea de a modifica drepturile rolului unui utilizator care folosește aplicația produce un mesaj de avertisment, deoarece drepturile unui rol nu pot fi modificate decât dacă nu este folosit (**Figura 4.36**). Dacă se dorește modificarea drepturilor unui rol care folosește aplicația atunci acel utilizator trebuie să părăsească aplicația pentru momentul în care drepturile acestuia sunt modificate de către un alt utilizator.

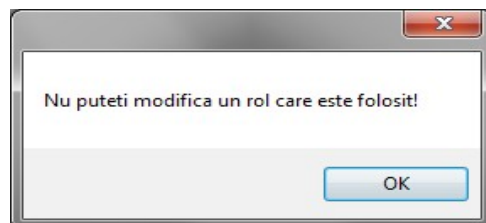


Figura 4.36. Avertisment la încercarea modificării drepturilor unui rol care este folosit

Dacă pentru un rol dezactivăm anumite drepturi atunci în momentul în care un utilizator încearcă să acceseze o funcționalitate la care nu are drept atunci acesta este semnalat în mod corespunzător (**Figura 4.37**).

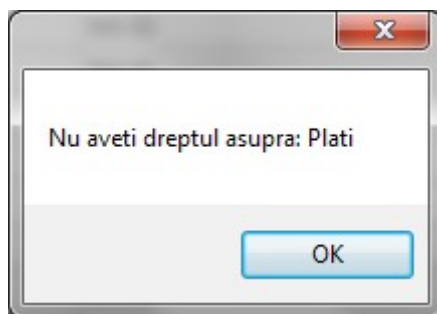


Figura 4.37. Avertisment în momentul în care un utilizator încearcă să acceseze o funcționalitate care nu este inclusă în rolul său

După ce au fost definite rolurile se poate crea un utilizator. Deschidem din tree-ul aplicației lista de utilizatori (**Figura 4.38**) pentru a crea un utilizator.

Capitolul 4 - Prezentarea aplicației

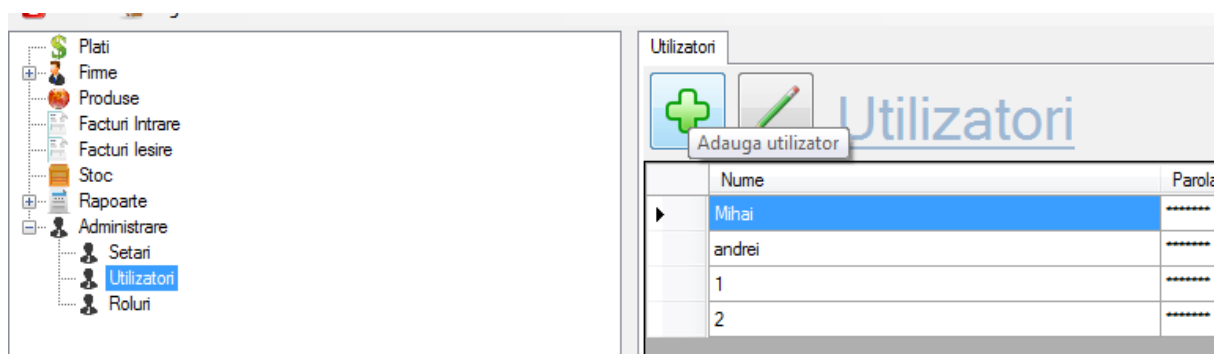


Figura 4.38. Deschiderea listei de utilizatori

Relația utilizator conține o cheie străină către un rol, de aceea crearea utilizatorului depinde de existența a cel puțin un rol. Datele necesare pentru un utilizator sunt: numele, parola și rolul (**Figura 4.39**).

Adauga utilizator

Nume

Parola curenta

Parola

Rol

Vanzator

☐ Activ

Iesire Salvare

Figura 4.39. Introducerea datelor pentru crearea unui utilizatorii

Parola curentă este necesară doar în cazul în care se dorește modificarea unui utilizator, din motive de siguranță, pentru că altfel orice utilizator ar putea schimba parola oricărui alt utilizator. Orice utilizator are o proprietate în plus care spune dacă acesta este activ sau nu. Verificarea dacă un utilizator este activ sau nu se face în momentul autentificării:

```
if (utilizator.Activ == false)
{
```

Capitolul 4 - Prezentarea aplicației

```
MessageBox.Show("Utilizatorul nu este activat!");  
return false;  
}
```

Încercarea de autentificare cu un utilizator blocat provoacă un mesaj de avertisment, ilustrat în **Figura 4.40**.

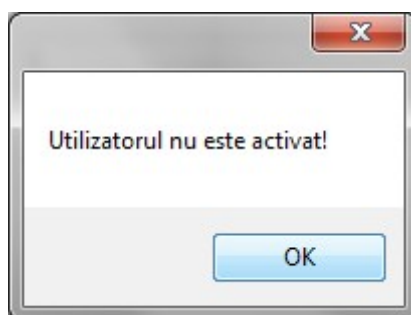


Figura 4.40. Mesaj afișat pentru un utilizator care nu este activ

Setările pentru aplicație se fac tot din administrare, se deschide tabul cu setări (**Figura 4.29**). În această fereastră se poate modifica parola administratorului și 'firma mea', care reprezintă firma ce utilizează această aplicație. *Firma mea* nu poate fi schimbată decât de administrator.

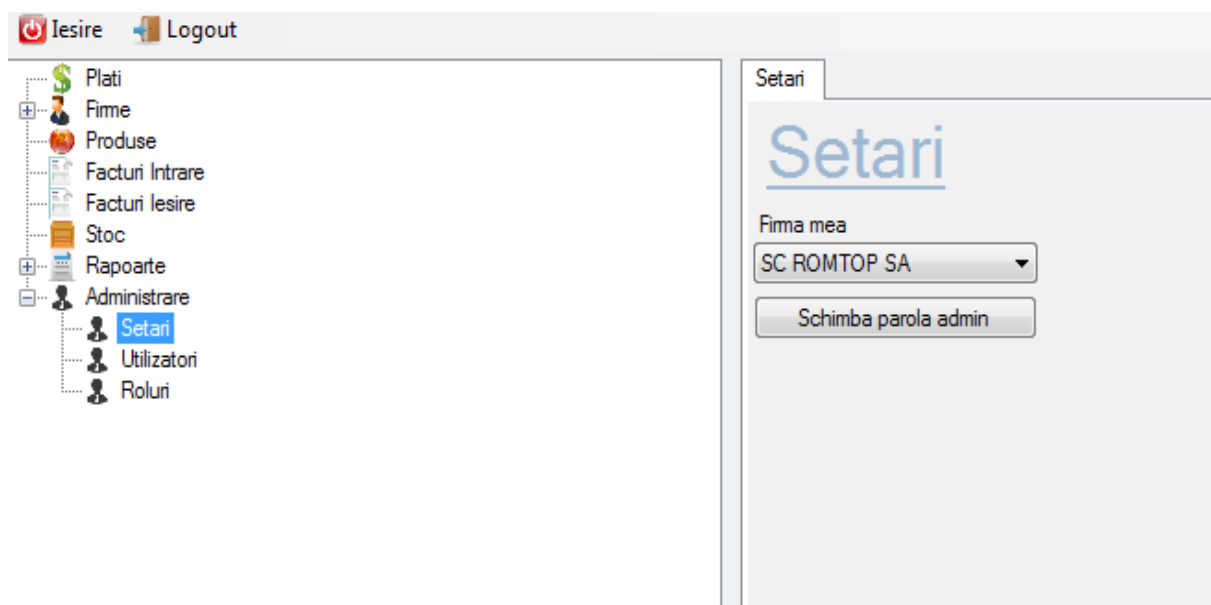


Figura 4.41. Setările aplicației

Capitolul 4 - Prezentarea aplicației

Schimbarea firmei se face automat la selectarea altei firme, secvența de cod ce schimbă firma se află într-un eveniment:

```
Setare.GetSetare().MyFirmaId = (int)firmeCmb.SelectedValue;  
var result = Setare.GetSetare().Save();  
if (result.Status != GestiuneBusiness.Enums.StatusEnum.Saved)  
{  
    MessageBox.Show(result.Message);  
}
```

Practic pentru schimbarea firmei nu se schimbă decât atributul care reține cheia primară a firmei, el fiind o cheie străină în relația setare. Printr-o operație de cuplare cu tabela firmelor se pot afla toate informațiile despre firmă.

Pentru schimbarea parolei pentru administrator trebuie mai întâi confirmată, din motive de siguranță, parola curentă a administratorului, pentru că altfel orice utilizator cu drept de administrare ar putea schimba parola celui mai important utilizator al aplicației (Figura 4.42).

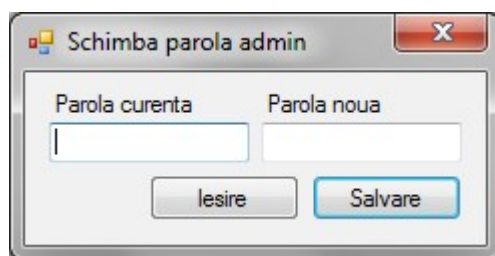


Figura 4.42. Schimbarea parolei pentru administrator

Bibliografie

- [1] Baze de date, Cristian Georgescu, Mihaela Georgescu
- [2] Platforma .NET, MSDN
- [3] Limbajul C#, Ph.D. Lucian Sasu
- [4] http://en.wikipedia.org/wiki/.NET_Framework
- [5] Baze de date oracle limbajul SQL, Prof.univ.dr. Ion LUNGU
- [6] http://ro.wikipedia.org/wiki/Microsoft_SQL_Server
- [7] Introducere în programarea .NET Framework, Adrian Niță