

Cuprins

Capitolul 1	Introducere.....	1
1.1	Prezentare generala.....	1
1.2	Motivatie.....	2
1.3	Obiective.....	3
1.4	Structura lucrarii.....	3
1.5	Exemplu rapid.....	4
Capitolul 2	Proiectarea arhitecturilor software.....	6
2.1	Componentele unei aplicatii distribuite.....	6
2.1.1	Niveluri si componente in aplicatii.....	6
2.1.2	Tipuri de componente.....	8
2.1.3	Recomandari generale pentru proiectarea aplicatiilor.....	12
Capitolul 3	Proiectarea arhitecturii de persistenta.....	14
3.1	Decizii de proiectare.....	14
3.1.1	Tipul arhitecturii.....	14
3.1.2	Runtime vs. generare de cod.....	16
3.1.3	Modelarea datelor relationale prin clase de persistenta.....	17
3.2	Proiectarea nivelului de acces la date.....	19
3.2.1	Recomandari pentru proiectarea componentelor.....	19
3.2.2	Ierarhia de clase.....	20
3.2.3	Operatiile implementate.....	21
3.2.4	Folosirea procedurilor stocate.....	22
3.2.5	Operatii avansate.....	23
3.3	Proiectarea claselor de persistenta.....	26
3.3.1	Reprezentarea entitatilor business.....	26
3.3.2	Ierarhia de clase.....	28
3.3.3	Operatiile implementate.....	29
3.3.4	Modelul avansat al claselor de persistenta.....	30
Capitolul 4	Implementare si utilizare.....	34
4.1	Tehnologii folosite.....	34
4.1.1	Platforma .NET.....	34
4.1.2	Microsoft SQL Server.....	35
4.1.3	Generatoare de cod.....	35
4.1.4	Proiectare vizuala cu Visio.....	36
4.2	Implementarea arhitecturii de persistenta.....	37
4.2.1	Schema exemplu.....	37
4.2.2	Clasele de persistenta.....	39
4.2.3	Persistarea obiectelor.....	47
4.2.4	Cautarea informatiilor.....	49
4.2.5	Reprezentarea relatiilor.....	57
Capitolul 5	Aplicatie demonstrativa.....	65
5.1	Structura proiectului.....	65
	Componentele aplicatiei.....	66
Capitolul 6	Concluzii.....	68
6.1	Viitoare imbunatatiri.....	69
Anexe	70

Capitolul 1 Introducere

1.1 Prezentare generala

Obiectul acestei lucrari este prezentarea unei arhitecturi software folosita pentru persistarea datelor din aplicatii .NET intr-un mediu de stocare (*persitence framework*). In cazul acesta este vorba de persistarea obiectelor instantiate la runtime in baze de date relationale, mai precis Microsoft SQL Server.

Pe scurt, arhitectura prezentata aici reprezinta o componenta esentiala a oricarui mediu de dezvoltare modern, oferind programatorilor o unealta eficienta, prin care sa reduca la minim timpul de dezvoltare si numarul de bug-uri din interiorul aplicatiilor. Cu ajutorul ei se elimina necesitatea de a repeta la nesfarsit aceleasi sectiuni de cod care se ocupa de salvarea de informatii si cautarea lor ulterioara. Fiind automatizate, toate aceste operatii sunt mult mai sigure, fara erori si permit o intretinere usoara a aplicatiei, putand reduce cu pana la 40% timpul total de dezvoltare.

Arhitectura este impartita pe 2 niveluri, dupa cum urmeaza:

1. Primul si cel mai abstract nivel este reprezentat de clase .NET prin care se realizeaza o **mapare entitate/relatie** a claselor folosite in aplicatie peste tabelele din baza de date (*O/R Mapper, Business Entities*).
2. Nivelul urmator este **nivelul de acces de date** (*Data Access Layer*) si reprezinta o modalitate de a abstractiza legatura cu mediul de stocare — prin schimbarea acestuia se poate migra catre un alt server de baze de date sau la o metoda complet diferita de a transporta si stoca informatiile: fisiere XML, servicii SOAP, *s.a.m.d.* — in mod transparent pentru nivelele superioare ale aplicatiei. Acesta integreaza si un sub-nivel implementat direct in motorul de

baze de date ce consta dintr-un set de **proceduri stocate** care implementeaza setul standard de **operatii CRUD** (*Create, Retrieve, Update, Delete*).

1.2 Motivatie

La ora redactarii acestei lucrari exista un numar mare de arhitecturi software care abordeaza aceeasi problema ca si lucrarea mea, multe dintre ele *open-source*. Motivul principal pentru care am ales sa dezvolt o astfel de arhitectura — costurile fiind destul de insemnate — este lipsa unui *framework* de persistenta pentru platforma .NET in urma cu doi ani, cand am inceput dezvoltarea unei aplicatii foarte complexe din domeniul medical. Componenta mica a echipei de dezvoltare m-a determinat sa iau aceasta hotarare, de a investi in construirea acestei arhitecturi, inainte de inceperea proiectului propriuzis.

Desi a pus la dispozitia dezvoltatorilor o platforma avansata (.NET) si un mediu ideal de dezvoltare (Visual Studio), Microsoft nu a reusit sa vina in intampinarea nevoilor dezvoltatorilor si cu o arhitectura de persistenta. Lansarea arhitecturii Microsoft, *ObjectSpaces*, a fost amanata pe o perioada nedeterminata, ultimele stiri plasand aceasta data in 2007, odata cu lansarea sistemului de operare *Microsoft Vista*. Cu toate acestea, site-ul MSDN (*Microsoft Developer Network*) este o sursa bogata in informatii referitoare la design-ul arhitecturilor software, prezentand pe larg notiunea de **arhitectura de persistenta** si motivele pentru care am vrea sa folosim asa ceva.

Documentele *Application Architecture for .NET: Designing Applications and Services* si *Designing Data Tier Components and Passing Data Through Tiers*, din sectiunea *patterns & practices* a site-ului MSDN, au reprezentat o majora sursa de inspiratie in proiectarea acestei arhitecturi.

Dintre celelalte solutii prezente la vremea respectiva, *Hibernate* este cea care m-a influentat cel mai mult, fiind o solutie de referinta in lumea Java. Intre timp a fost realizata si o portare a acestei arhitecturi pentru platforma .NET, *NHibernate*. Ambele, ca si solutia prezentata de mine, implementeaza cu succes multe din dezideratele unei arhitecturi de calitate, prezentate in capitolul urmator.

1.3 Obiective

Lucrarea isi propune:

- sa analizeze necesitatea unei astfel de arhitecturi software, precum si partea teoretica din spatele ei, punand accentul pe dificultatile intampinate la realizarea propriu-zisa a arhitecturii;
- sa prezinte tehnologiile folosite la implementare si motivele din spatele acestor alegeri;
- sa discute toate detaliile legate de implementarea si utilizarea arhitecturii, cu exemple numeroase de cod sursa;
- sa demonstreze usurinta dezvoltarii de aplicatii cu ajutorul acestei arhitecturi, prin intermediul unei aplicatii demonstrative, dezvoltate special in acest scop.

1.4 Structura lucrarii

Lucrarea este structurata in felul urmator:

Capitolul 2, *Proiectarea arhitecturilor software*, analizeaza de la un nivel superior necesitatea acestei arhitecturi si unde cade ea intr-o aplicatie bine proiectata. Vom vedea care este cea mai buna impartire pe niveluri (*tiers* sau *layers*) a unei aplicatii si care sunt componentele cele mai intalnite in aplicatii.

Capitolul 3, *Proiectarea arhitecturii de persistenta*, va analiza deciziile pe care trebuie sa le luam atunci cand proiectam o arhitectura de persistenta si va prezenta detaliile de implementare ale celor doua niveluri principale. De asemenea vom analiza diverse metode de reprezentare a informatiilor si transferul acestora intre diversele niveluri ale programului.

Capitolul 4, *Implementare si utilizare*, va prezenta tehnologiile pe care se bazeaza arhitectura si va parcurge intreaga ierarhie de clase prin detalii de implementare si exemple de cod. Vom vedea cum se creaza, cauta si sterg obiecte, cum se mapeaza relatii de mostenire, asociere sau compunere, cum se valideaza automat informatiile primite de la utilizator, precum si alte aspecte mai avansate.

In Capitolul 5, *Aplicatie demonstrativa*, voi incerca sa pun in valoare toate aspectele

arhitecturii, prin folosirea ei la dezvoltarea unei aplicații de gestionat o flotă de taximetre. Vom vedea cum putem avea un prototip funcțional al programului în mai puțin de o oră de la schitarea bazei de date.

În încheiere vom trage câteva concluzii despre starea actuală a proiectului și vom prezenta planuri de dezvoltare ulterioară, iar la sfârșitul lucrării veți putea găsi anexate bibliografia și codul complet al arhitecturii și al aplicației demonstrative.

1.5 Exemplu rapid

Înainte de a trece la capitolul următor, *Proiectarea arhitecturilor software*, vreau să vă ofer un mic exemplu de cod scris cu ajutorul arhitecturii prezentate, pentru a vă face rapid o idee despre puterea acesteia.

Operatii cu angajatii unei firme de taximetre

```
// adaugare angajat
Angajat a = new Angajat();
a.nume = "Popescu";
a.prenume = "Ion";
a.Save();

// adaugare masini
Masina m1 = new Masina();
Masina m2 = new Masina();
m1.marca = "Renault";
m2.marca = "Toyota";

// adaugare la lista de masini conduse (relatie 1..n)
m1.angajatId = m2.angajatId = a.angajatId;
// sau direct prin obiectul parinte obtinut automat prin parcurgerea
relatiei
m1.angajat = m2.angajat = a;
// salvare masini
m1.Save();
m2.Save();

// actualizare obiect
a.nume = "George";
a.Save();

// cautare masini dupa angajatId -- get by foreign key
MasinaList ml = MasinaDB.GetByFK("angajatId", 2);

// sau folosind lista creata automat in obiectul de tip Angajat
a = AngajatDB.Get(2);
ml = a.masinaList;
```

```
// stergere masini
foreach (Masina m in ml) {
    m.Delete();
}
```

Capitolul 2 Proiectarea arhitecturilor software

Aceasta lucrare se adreseaza in primul rand proiectantilor de aplicatii software — atat celor care participa activ la dezvoltarea lor, cat si celor care doar realizeaza design-ul *high-level* al arhitecturii aplicatiilor respective. Scopul final al lucrarii este de a le oferi acestora un model de urmat atunci cand abordeaza un proiect nou, indiferent daca este vorba despre o simpla aplicatie desktop, sau de o complexa aplicatie distribuita, alcatuita din mai multe componente si bazata in mare parte pe servicii.

Cu toate ca pana la sfarsitul lucrarii vom aborda si problema implementarii acestor deziderate, oferind o arhitectura de persistenta a datelor complet functionala si utilizabila, capitolul *Proiectarea arhitecturilor software* se concentreaza pe design-ul arhitecturii si al componentelor unei aplicatii.

2.1 Componentele unei aplicatii distribuite

Pentru a ajunge la un design de calitate, care sa faca fata oricaror cerinte, vom pleca de la premiza ca proiectam o aplicatie distribuita, aceasta avand de obicei o structura complexa, bazata pe mai multe componente, care pot rula pe masini diferite. O aplicatie desktop normala reprezinta o simplificare a acestui model, principala diferenta fiind modalitatea de transfer a informatiei intre diversele componente. Daca se pastreaza insa logica de separare in componente, orice aplicatie desktop poate fi mai tarziu extinsa pentru a obtine o aplicatie distribuita.

2.1.1 Niveluri si componente in aplicatii

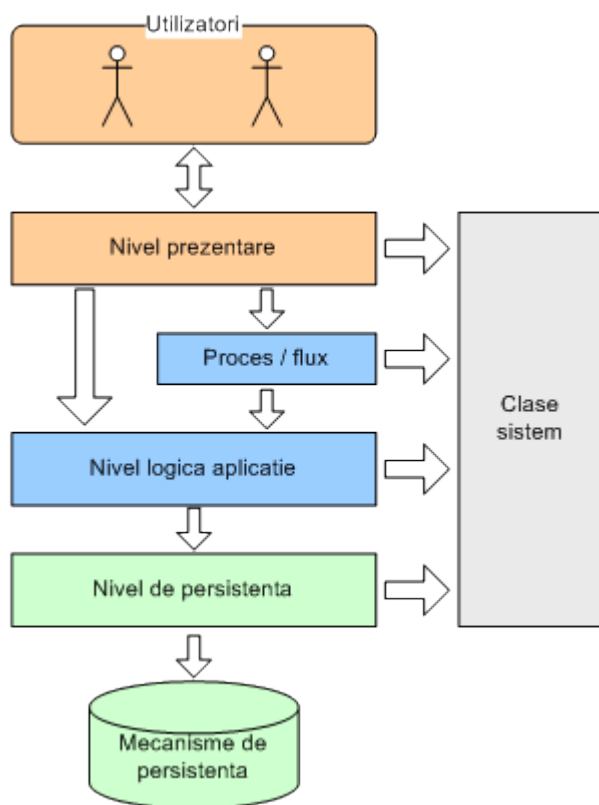
Intr-o arhitectura moderna este in general acceptata ca practica recomandata impartirea aplicatiei in mai multe componente, care ofera servicii de prezentare, de

2.1 Componentele unei aplicatii distribuite

logica (componenta *business*) si de transport/stocare de informatii. Componentele care realizeaza functii similare pot fi grupate in niveluri (*layers*), care de cele mai multe ori sunt organizate sub forma unei stive. Astfel, componentele de “deasupra” unui nivel folosesc serviciile expuse de acesta — pentru ca o anumita componenta sa-si faca treaba, ea va chema aceste servicii si alte servicii de sub ea.

Cuplarea intre niveluri trebuie realizata intr-o maniera cat mai slaba, prin interfete bine definite. Aceasta metoda permite schimbarea detaliilor de implementare fara ca nivelurile superioare/adiacente sa stie acest lucru, cu conditia pastrarii interfetei definite initial. Rezultatul final este un produs al carui cod sursa este usor de intretinut si imbunatatit.

Figura 1.1 prezinta o vedere simplificata a unei aplicatii, impartita pe 3 niveluri:



Pentru a atinge mai usor aceste deziderate, se poate opta pentru o arhitectura si mai stricta din punct de vedere a comunicarii intre niveluri: componentele aplicatiei de la un anumit nivel nu trebuie sa comunice decat cu alte componente adiacente sau cu cele de pe nivelul imediat urmator. Urmand acest principiu, putem formula

urmatoarele reguli (marcate si vizual prin sagetile din Figura 1.1):

1. **Clasele din interfata cu utilizatorul nu ar trebui sa acceseze mecanismele de persistenta.** Componentele cu care interactioneaza utilizatorul vor prelucra/servi informatii primite doar de la nivelul de logica al aplicatiei (componentele si procesele *business*). Totodata, prin incapsularea unor procese complicate in componentele *business*, aceasta logica va putea fi refolosita in program, in locuri complet diferite.
2. **Componentele *business* nu ar trebui sa acceseze mecanismele de persistenta.** In acest fel, ele vor fi protejate de eventualele schimbari survenite mecanismului de persistenta, ele fiind preluate integral de nivelul de acces la date.
3. **Arhitectura aplicatiei este ortogonala pe arhitectura hardware/retea.** O solutie distribuita se poate implementa in cadrul mai multor organizatii, peste mai multe masini — caz in care comunica prin intermediul retelelor de calculatoare, expunand servicii prin care diversele componente ale programului schimba informatii.

In special ultimul punct este important, deoarece el scoate in evidenta flexibilitatea intrinseca pe care trebuie sa o aiba arhitectura unei aplicatii, daca dorim ca ea sa fie extensibila si sa scaleze bine odata cu cresterea complexitatii domeniului de activitate. In Tabelul 1.1 putem vedea solutiile posibile pentru implementarea catorva topologii hardware/retea diferite:

Tip componenta/arhitectura	Client izolat	<i>Thin client</i> -Server	<i>Fat client</i> -Server	Aplicatie distribuita
Interfata cu utilizatorul	Client	Client	Client	Client
Proces, controller	Client	Server	Client	Server de aplicatii
Domeniu, <i>Business</i>	Client	Server	Client	Server de aplicatii
Persistenta	Client	Server	Server	Server de baze de date

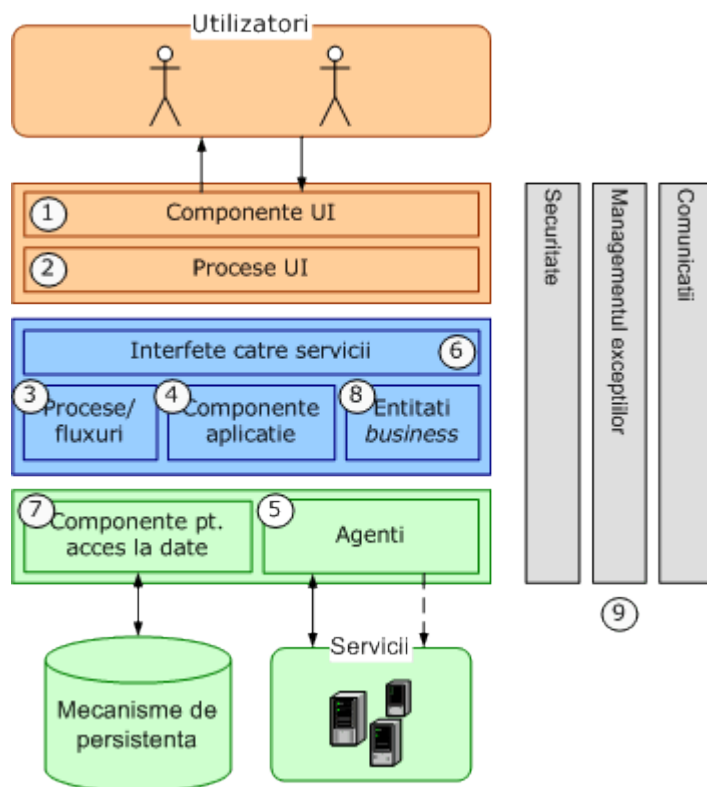
2.1.2 Tipuri de componente

In continuare vom detalia impartirea schematica pe niveluri din sectiunea anterioara.

2.1 Componentele unei aplicatii distribuite

E important de observat ca prin nivel al aplicatiei (*layer*) se intelege in general o grupare logica de module software, care impreuna compun aplicatia finala. Rolul *layerelor* este de a ajuta la intelegerea **rolului** si a **tipului** fiecarei componente, fiecarui modul — ajutand astfel la crearea unor componente reutilizabile de-a lungul aplicatiei. Fiecare nivel este compus dintr-un numar finit de tipuri de componente grupate in sub-niveluri, cu fiecare sub-nivel implementand o anumita functionalitate. Prin identificarea tuturor **tipurilor** de componente se poate realiza o diagrama generica a aplicatiei, care poate fi apoi utilizata ca plan al implementarii, ajutand la pastrarea imaginii de ansamblu.

Figura 1.2 este o versiune mai detaliata a Figurii 1.1, incluzand sub-nivelurile si componentele cel mai des intalnite in aplicatii:



Acestea sunt:

1. **Componente de interfata cu utilizatorul (UI).** Este vorba de diferitele ecrane ale aplicatiei, utilizate la a strange informatii sau la afisarea acestora. Acestea sunt singura parte vizibila a aplicatiei si deci la proiectarea lor trebuie avuta o

grija deosebita. Pentru o buna proiectare sunt necesare cunostiinte de Design, Arhitectura Informatiei (*Information Architecture*), Ergonomie (*Usability*), Accesibilitate. In aplicatiile desktop ele sunt construite prin ferestre si controale iar in aplicatiile web prin pagini cu tabele, formulare *s.a.m.d.*

2. **Procese de interactiune cu utilizatorul.** In multe cazuri, strangerea de date de la utilizatori sau completarea unei actiuni presupune existenta unui proces bine definit. Ganditi-va la vrasjitorii din aplicatii (*wizards*), utilizati pentru instalarea unor programe sau ducerea la capat a unei operatii mai complicate. Gruparea acestui set de actiuni intr-un proces bine orchestrat asigura strangerea completa de informatii, inainte de a executa propriuzis actiunea respectiva. Separarea acestui cod de interfata cu utilizatorul permite refolosirea lui in interfete diferite, fara a-l duplica (web si desktop) si simplifica intretinerea aplicatiei. Dezavantajul este ca sistemul devine mai complex, pentru ca trebuie sa luam in calcul pastrarea *state*-ului curent al aplicatiei si transferul de date intre pasii procesului.
3. **Procese/fluxuri in logica aplicatiei.** La fel ca la procesele de interactiune cu utilizatorul, si activitatile interne ale aplicatiei sunt guvernate uneori de un proces complicat, care trebuie sa fie dus la bun sfarsit prin executarea secventiala a unui set de operatii mai mic. De exemplu, cumpararea online a unui produs presupune alegerea acestuia din stoc, calcularea valorii totale a comenzii prin includerea de taxe, verificarea cartii de credit si realizarea aranjamentelor pentru livrare. Aceste operatii pot dura o perioada nedeterminata de timp, de aceea aplicatia trebuie sa salveze in permanenta starea procesului pentru a putea fi reluat/finalizat mai tarziu.
4. **Componente de logica a aplicatiei.** Acestea sunt componentele de baza ale aplicatiei, care executa calcule pentru afisarea de rezultate, sau proceduri complexe pentru stocarea informatiilor. Indiferent daca este vorba de salvarea unui utilizator sau determinarea drepturilor sale asupra unui set de date, sau chiar afisarea acelu set, exista "bucati" de cod care vor duce la capat o singura actiune clar specificata.
5. **Agenti pentru interactiune cu serviciu.** Atunci cand o aplicatie se bazeaza pe

servicii exterioare, indiferent ca acestea sunt accesibile in cadrul sistemului sau puse la dispozitie de o companie externa, ea trebuie sa implementeze componente care comunica cu acele servicii. Exemple pot fi o componenta care vorbeste cu serverul unei banci pentru a verifica o carte de credit, sau o alta componenta care interogheaza baza de date a unui curier pentru a putea calcula taxele de transport.

6. **Interfete catre servicii oferite.** Atunci cand aplicatia insasi pune la dispozitia unor agenti externi servicii asemanatoare celor de mai sus, ea trebuie sa expuna aceste servicii prin intermediul unei interfete bine documentate (API sau *Application Programming Interface*). Acestea vor defini toate operatiile puse la dispozitia agentilor si vor realiza transferul de date prin protocoale de retea, mseaje, micro-formate si altele.
7. **Componente de acces la date.** Acestea sunt necesare pentru comunicatia cu elementele de persistenta, indiferent de modalitatea de stocare sau transport. O aplicatie poate folosi mai multe componente de acces la date simultan, iar daca ele sunt construite peste o interfata bine definita, schimbarea serverului de baze de date sau al protocolului de comunicatie folosit la schimbarea de date intre doua servicii poate fi schimbat complet transparent pentru aplicatie.
8. **Entitati *Business*.** Toate componentele mentionate mai sus manipuleaza si transfera informatii dintr-o parte in cealalta, intre doua calculatoare diferite, sau intre doua niveluri ale aceleiasi aplicatii. Entitatile Business sunt blocurile de informatie folosite in acest scop, iar ele pot fi implementate in multe feluri: DataSets, DataReaders sau stream-uri XML. Cel mai adesea vor fi insa implementate sub forma unor clase bine specificate, modelate pe domeniul aplicatiei, pentru a reprezenta usor entitatile din lumea reala cu care opereaza aplicatia: *Produse, Comenzi, Angajati, Furnizori*, etc.
9. **Alte componente.** O aplicatie complexa va folosi probabil si multe alte componente, cum ar fi:
 - *jurnalizarea actiunilor si managementul exceptiilor;*
 - *sistem de securitate*, pentru a acorda utilizatorilor drepturi la diferitele componente ale aplicatiei si la informatii;

- *comunicatii* cu alte componente ale aplicatiei sau cu servicii externe.

Toate componentele de mai sus depind bineinteles de sistemul de operare si de platforma de dezvoltare aleasa de dezvoltatori (Ex: *Windows Server* + *.NET framework*).

2.1.3 Recomandari generale pentru proiectarea aplicatiilor

In sectiunea anterioara am trecut in vedere cele mai intalnite tipuri de componente care apar in aplicatii distribuite. Lista nu este una exhaustiva, insa ele isi gasesc locul in majoritatea aplicatiilor moderne.

Tinand cont de faptul ca nu toate aplicatiile sunt la fel, puteti folosi lista urmatoare ca pe un ghid util in proiectarea aplicatiilor:

- **Identificati componentele necesare aplicatiei.** Nu toate aplicatiile au nevoie de toate componentele avansate, cum ar fi accesul la servicii sau fluxuri logice complicate. Daca numarul ecranelor aplicatiei este mic probabil nici procesele de interactiune cu utilizatorul nu sunt necesare.
- **Proiectati toate componentele de acelasi tip dupa un model bine definit,** astfel incat lucrul cu ele sa fie cat mai consecvent cu putinta. In acest fel comportamentul lor va fi usor predictibil, iar intretinerea va fi usor de realizat chiar si de o echipa numeroasa de programatori. Este o buna practica definirea de interfete care sa fie mai apoi implementate de toate clasele de acelasi tip. Interfetele definesc un contract pe care toate clasele se angajeaza sa-l respecte. Acest model poate fi extins prin plasarea de clase abstracte intre interfata si clasele propriuzise, asigurand din start un minim de functionalitate care nu va trebui redefinit apoi in fiecare dintre clasele extinse.
- **Intelegeti cum comunica intre ele diversele componente ale aplicatiei** inainte de a alege limitele "fizice" dintre ele. E de preferat o cuplare slaba, implementata prin cateva apeluri importante decat una mai stransa, realizata prin multe apeluri nenecesare, mai ales la comunicatii la distanta intre servicii si agenti.
- **Pastrati formatul intern de reprezentare interna a datelor (entitatile**

business) consistent de-a lungul aplicatiei. Folosi un singur format, sau daca folosirea unui format aduce beneficii evidente intr-o anumita situatie, pastrati totusi numarul acestora cat mai mic cu putinta. Amestecarea unui numar mare de formate complica aplicatia, face dificila intretinerea ei si creaza un surplus inutil de cod, in general pentru a realiza conversia de la un format la altul.

- **Scrieti codul care implementeaza politici de securitate, comunicatii sau management al exceptiilor cat mai simplu cu putinta.** Realizati librarii cu aceste componente, care vor fi apelate din restul aplicatiei cat mai usor, printr-o singura linie de cod daca este posibil. In acest fel logica aplicatiei nu este poluata cu elemente generale iar codul este mai usor de urmarit si de intretinut.
- **Stabiliti de la inceput daca impartirea in niveluri se face intr-un mod mai strict sau mai relaxat.** Intr-o impartire stricta, componentele dintr-un nivel vor apela doar componente din nivelul imediat urmator, nu au voie sa apeleze functii cu 2 niveluri mai jos. Aceasta abordare izoleaza complet nivelurile de deasupra de cele de dedesupt, oferind o mai mare flexibilitate in luarea deciziilor locale, deoarece acestea nu ar trebui sa influenteze comportarea celorlalte niveluri. Daca optati pentru o maniera mai relaxata, in care componentele de la un nivel au acces la componente mult mai jos in stiva de niveluri, puteti realiza oarece economie de cod, pentru ca vor disparea componente care nu fac decat sa inainteze apelurile catre nivelurile inferioare. In orice caz, trebuie evitate cu desavarsire apelurile de jos in sus, care introduc dependinte nedorite in aplicatie — un nivel nu ar trebui sa stie cine il apeleaza, ci doar sa puna la dispozitie setul stabilit de functionalitati.

Cu aceste recomandari la indemana putem incepe sa construim arhitectura de persistenta. Capitolul urmator se va concentra pe arhitectura de persistenta, alcatuita din clasele de persistenta (entitatile *business*), clasele de acces la date si mecanismul de persistenta propriuzis.

Capitolul 3 Proiectarea arhitecturii de persistenta

In capitolul 2, *Proiectarea arhitecturilor software*, am prezentat arhitectura si componentele unei aplicatii distribuite, incheind cu un set de recomandari utile pentru proiectarea unei aplicatii oricat de complexe. In continuare vom analiza deciziile pe care trebuie sa le luam atunci cand proiectam o arhitectura de persistenta, dupa care vom trece la proiectarea propriuzisa a acestia.

3.1 Decizii de proiectare

3.1.1 Tipul arhitecturii

Principala decizie care trebuie luata atunci cand se proiecteaza o arhitectura de persistenta este daca se doreste o modelare a aplicatiei dupa baza de date, sau daca se construiesc baza de date in urma crearii claselor din aplicatie. Cu alte cuvinte, daca ne referim la asezarea nivelurilor din Figura 1.2, trebuie sa hotaram daca optam pentru un design **de jos in sus** (baza de date spre clasele de persistenta) sau unul **de sus in jos** (clase mapate in baza de date).

Alte arhitecturi mai avansate implementeaza chiar ambele solutii, in sa in prezenta lucrare vom alege o abordare de jos in sus, oarecum constransi de uneltele folosite la generarea codului. Iata ce presupun cele doua metode.

Proiectarea de jos in sus

Abordarea solutiei de jos in sus presupune mai intai proiectarea bazei de date. Folosind meta-informatiile stocate de motoarele SQL avansate din ziua de azi, se poate reconstitui intreaga schema a informatiei, folosita mai apoi de arhitectura de

persistenta pentru a genera atat codul SQL necesar interogarii bazei de date, cat si clasele de persistenta folosite pentru transportul datelor in interiorul aplicatiei.

Proiectarea de sus in jos

O abordare de sus in jos presupune scrierea claselor la nivelul limbajului de programare, dupa care li se adauga mecanismul de persistenta. Aceasta inseamna de cele mai multe ori modelarea informatiei dupa principii POO, dupa care, bazandu-ne pe constructorii ai limbajului sau pe mecanisme de *reflection*, vom genera automat codul pentru interogarea tabelelor si relatiilor necesare. Privind implementarea propriu-zisa a acestei solutii avem doua posibilitati:

1. Cea mai simpla metoda presupune crearea “de mana” atat a claselor de persistenta, cat si a bazei de date. Arhitectura se reduce la un set generic de clase — cunoscut sub numele de *O/R Mapper (Object/Relationship Mapper)* — care genereaza la runtime codul SQL necesar pentru interogarea bazei de date;
2. Cealalta metoda se regaseste in arhitecturile cele mai avansate (de ex. Hibernate) si presupune definirea intregii scheme a informatiei in format XML. Acest format va fi apoi folosit pentru generarea atat a claselor de persistenta cat si a codului SQL, ba chiar a tabelelor si relatiilor.

Folosirea unei scheme XML pentru reprezentarea schemei informatiei poate fi vazuta ca o solutie de mijloc, inasa introduce un nivel suplimentar de complexitate arhitecturii: ea va trebui sa implementeze mecanisme de conversie de la numitorul comun oferit de solutia XML atat catre clasele de persistenta, cat si catre o schema SQL concreta, inasa odata realizat acest lucru putem alege oricare dintre sensuri pentru a dezvolta aplicatia.

Puterea acestei solutii consta in posibilitatea de a porta usor schema pe un alt motor SQL, folosind meta-informatiile definite in XML. Continuand pe aceasta idee, se pot scrie scripturi ajutatoare care sa converteasca intre schema XML si diverse scheme SQL, automatizand foarte mult procesul de portare (mecanism implementat de arhitectura de persistenta *Propel*, scrisa pentru limbajul PHP).

3.1.2 Runtime vs. generare de cod

Indiferent de solutia aleasa mai sus, implementarea acesteia se poate face in doua feluri: prin operatii complicate la runtime sau prin generarea de cod sursa care urmeaza a fi compilat.

Procesare la runtime

Aceasta metoda, mentionata si in sectiunea anterioara, consta in realizarea unui *O/R mapper* care realizeaza majoritatea muncii la *runtime*. Acesta se bazeaza pe diverse metainformatii, cum ar fi attribute puse la dispozitie de limbajul de programare (C#) sau comentarii inserate in cod (PHP). Cu ajutorul acestora, sau folosind mecanisme de *reflection*, se pot construi clase de persistenta prin fabrici de obiecte si se poate genera tot codul SQL necesar interogarii bazei de date.

Avantajul acestei metode este o toleranta mai mare la schimbari in schema informatiei, pentru ca este suficienta alterarea bazei de date sau a schemei XML, iar la urmatoarea rulare a programului, acesta va detecta automat noile campuri sau relatii, fara a fi necesara o recompilare sau modificarea codului.

Dezavantajul major este puterea de calcul mai mare, necesara pentru a parse structuri XML complexe, de a crea la runtime noi tipuri si a instantia obiectele corespondente, precum si de a genera codul SQL necesar si trimiterea acestuia catre baza de date. Vom vedea care sunt dezavantajele aduse de codul SQL generat ad-hoc in sectiunea urmatoare.

Generarea de cod

Aceasta solutie presupune inspectarea schemei informatiei — fie ea schema definita in XML sau direct baza de date — si generarea de cod SQL si de clase de persistenta: clasele de acces la date si entitatile *business*.

Cel mai mare dezavantaj al acestei metode este necesitatea de a regenera toate clasele si codul SQL afectate de modificari la nivelul schemei informationale. Chiar daca acest lucru este in mare parte automatizat, este nevoie de atentie suplimentara pentru a nu suprascrisa modificari facute manual (lucru ce se poate preveni printr-o ierarhizare

a claselor) si a stabili ce parti din aplicatie trebuie regenerate.

Odata intrata in obisnuinta dezvoltatorilor, generarea de cod are insa numeroase avantaje:

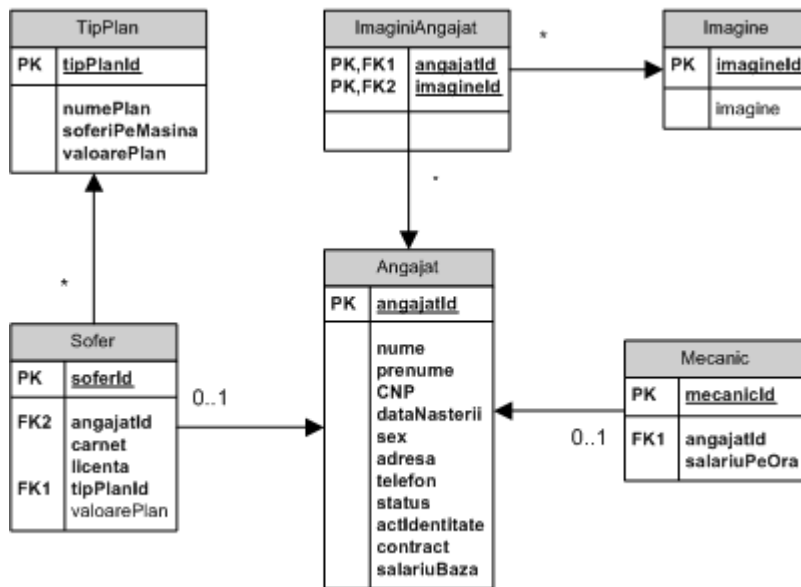
- viteza sporita la runtime deoarece codul generat va fi compilat;
- adaugare usoara de operatii avansate claselor, deoarece codul este generat si nu se mai consuma din timpul dezvoltatorilor;
- clasele asociate tabelor sunt tipuri deja definite in limbajele *strongly-typed*;
- posibilitatea folosirii si intretinerii usoare a procedurilor stocate pentru interogarea bazei de date.

Unelte pentru generarea codului pot fi sau nu parte integranta a arhitecturii de persistenta, insa exista generatoare de cod foarte puternice si flexibile, cu ajutorul carora se poate ataca orice proiect odata ce arhitectura este gandita. In acest fel, timpul de dezvoltare poate scadea simtitor. Un alt avantaj al unui generator de cod universal este un API prin intermediul caruia se poate genera codul plecand de la orice motor de baze de date suportat — astfel se realizeaza mai usor dezideratul portabilitatii si al abstractizarii bazei de date, concept cheie in orice arhitectura de persistenta.

3.1.3 Modelarea datelor relationale prin clase de persistenta

Dupa cum am mentionat la inceputul acestei lucrari, scopul unei arhitecturi de persistenta este de a usura scrierea unei aplicatii care lucreaza cu o baza de date, ascunzand programatorilor toate detaliile de interogare a serverului SQL si punandu-le la dispozitie entitatile necesare pentru a manipula si salva datele.

Inainte de a investiga in detaliu proiectarea nivelurilor de acces la date si de clase de persistenta, sa vedem ce fel de scheme informationale poate modela/utiliza o arhitectura de persistenta. Deoarece am ales sa implementam o arhitectura **de jos in sus**, vom pleca de la schema unei baze de date. In Figura 3.1 vedem o diagrama care contine cateva tabele si relatiile dintre ele.



Arhitectura va trebui sa modeleze atat **tabelele** din baza de date (sub forma de entitati business) cat si **relatiile** dintre ele, bazate pe *chei primare* si *chei straine*. De exemplu, in diagrama de mai sus avem:

- relatii **1:1** (*one-to-one*) → Angajat:Mecanic, Angajat:Sofer;
- relatii **1:N** (*one-to-many*) → TipPlan:Sofer;
- relatii **M:N** (*many-to-many*) → Angajat:Imagine.

Alte aspecte pe care trebuie sa le ia in considerare arhitectura de persistenta sunt:

- cum reprezinta o singura instanta a unei entitati?
- cum reprezinta o colectie?

si mai avansate:

- cum reprezinta o ierarhie, o compunere sau o asociere?
- cum se tranzactiile si cum se trateaza accesul concurent la baza de date?

Recomandarile urmatoare trebuie avute in vedere la proiectarea ambelor niveluri din arhitectura de persistenta:

- pentru a modela schema informationala folositi unelte vizuale si formate cum ar fi UML (*Unified Modelling Language*), care permit surprinderea tuturor aspectelor schemei: asocieri, compuneri, mosteniri, interactiuni, procese, etc.;

- pabelele folosite la reprezentarea relatiilor **N:M** nu trebuie reprezentate explicit ca entitati; in schimb for fi implementate colectii si metode pentru a reprezenta obiectele din relatie (ex: tabela `ImaginiAngajat`);
- fiecare clasa entitate care reprezinta o tabela trebuie sa aiba o clasa echivalenta de acces la date care va servi la salvarea si interogarea obiectelor de acest tip in baza de date;
- uneori este util sa grupam 2 sau mai multe entitati intr-o componenta, atunci cand in aplicatie se vor folosi deseori impreuna; acest lucru se poate face manual in situatii aparte sau automat, folosindu-ne de ierarhii de clase si mecanisme de compunere sau asociere.

3.2 Proiectarea nivelului de acces la date

Consistent cu prezentarea de pana acum, vom incepe proiectarea arhitecturii cu nivelul de acces la date. Componentele de acces la date sunt folosite pentru salvarea entitatilor in baza de date si gasirea lor ulterioara. Ele sunt implementate prin clase *stateless*, adica nu se mentin informatii in memorie intre apeluri, iar apelurile catre ele pot fi procesate independent, mai putin in cazul tranzactiilor.

3.2.1 Recomandari pentru proiectarea componentelor

La proiectarea componentelor de acces la date contribuie:

- alegerea unui mecanism de persistenta (in acest caz serverul de baze de date);
- designul componentelor propriuzise;
- modalitatea de reprezentare a datelor (entitatile business).

Nivelul de acces la date ascunde urmatoarele detalii de implementare de restul aplicatiei:

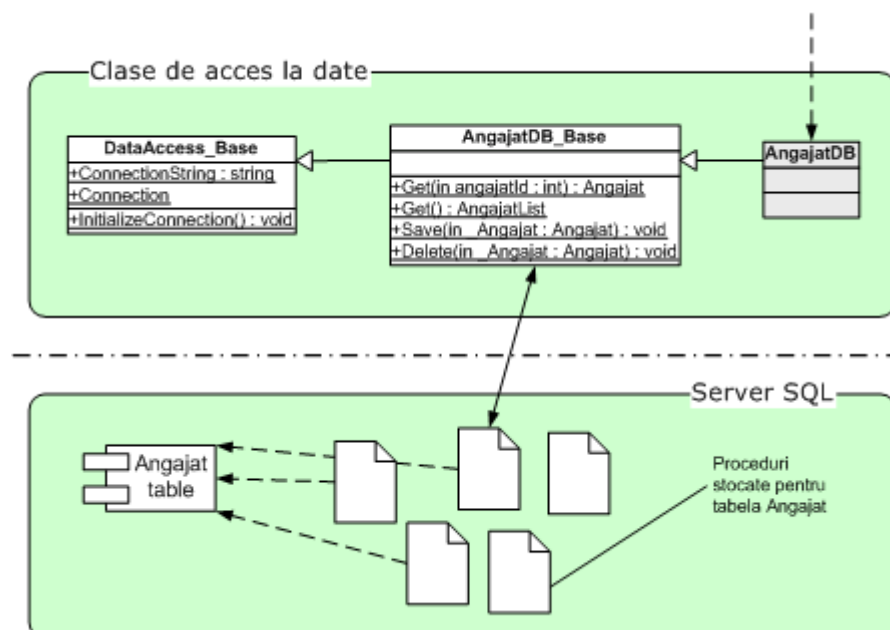
- abstractizarea bazei de date prin folosirea de librarii ajutatoare sau prin generarea mai multor tipuri de componente, cate unul pentru fiecare mecanism de persistenta utilizat;
- managementul conexiunilor la baza de date si implementarea politicilor de securitate si acces la date;

- paginarea rezultatelor din criterii de performanta;
- managementul erorilor;
- tratarea valorilor null;
- serializarea datelor pentru schimbul de informatii cu alte servicii, in format XML.
- managementul schemelor de locking si acces concurent;
- implementarea tranzactiilor pentru executarea in bloc a mai multor operatii;
- implementarea unui nivel intermediar de *caching* pentru a putea opera in mod offline.

Nu toate cerintele de mai sus vor fi implementate de toate arhitecturile de persistenta, mai ales a doua parte a listei fiind alcatuita de subiecte avansate, pe care le vom trece sumar in revista la sfarsitul acestui articol.

3.2.2 Ierarhia de clase

Dupa cum se vede din Figura 3.2, in aceasta arhitectura am implementat sistemul de acces la date printr-un model hibrid, partial implementat prin clase de acces la date, partial prin proceduri stocate definite la nivelul serverului SQL.



Clasele din nivelul de acces la date mostenesc toate o clasa de baza comuna, care centralizeaza operatiile legate de conectarea la baza de date si parametrii conexiunii. Urmatoarele doua niveluri sunt formate dintr-o clasa de baza in care sta tot codul generat si o clasa externa, cea folosita cu adevarat, in care poate fi adaugata logica aplicatiei.

Aceasta spargere este necesara pentru a permite programatorilor sa adauge “de mana” logica suplimentara componentelor de acces (cum ar fi cautari particularizate), fara a compromite mecanismul de generare de cod. In caz contrar, la fiecare regenerare a claselor, toata munca programatorilor ar fi fost suprascrisa.

3.2.3 Operatiile implementate

Operatiile de baza care sunt implementate de componentele de acces la date sunt:

- **Crearea** de entitati noi si salvarea lor — INSERT la nivel SQL;
- **Gasirea** entitatilor deja salvate — SELECT;
- **Actualizarea**, editarea entitatilor — UPDATE;
- **Stergerea** lor — DELETE.

Aceste 4 operatii sunt cunoscute in literatura de specialitate drept operatii **CRUD**, acronim de la denumirile in engleza *Create, Retrieve, Update, Delete*. Alte operatii uzuale implementate de clasele de acces la date sunt:

- gasirea de entitati sau colectii de entitati *legate* de obiectul primar al *query*-ului (obiecte care fac parte din asocieri sau compuneri) prin JOINS;
- gasirea de informatii stranse din mai multe tabele, in mod *read-only*, folosind VIEWS;
- cautari avansate, avand ca parametru de intrare o entitate partial completata;
- paginarea setului de rezultate in cazul unor colectii extinse, pentru a nu transfera dintr-o data mai multe informatii decat este necesar.

Metodele claselor primesc ca parametri valori scalare (de ex. id-urile obiectelor) sau entitati *business* (pentru salvare) si intorc entitatile completate cu date sau colectii de entitati. Interactiunea cu serverul SQL se face prin apelarea unui set de proceduri

stocate, definit pentru fiecare tabela, cate o procedura pentru fiecare operatie de mai sus — ele sunt apelate de metodele corespondente din nivelul de acces la date. Rolul acestor metode este de a ascunde nivelurilor superioare detaliile comunicarii cu serverul de baze de date, oferind un mecanism usor pentru a abstractiza baza de date.

Deoarece clasele sunt *stateless*, toate operatiile sunt implementate sub forma de **metode statice**, practic niste **fabrici** de entitati *business*.

3.2.4 Folosirea procedurilor stocate

Am ales sa extind modelul prin introducerea unui nivel suplimentar format din proceduri stocate in SQL, deoarece acestea prezinta urmatoarele avantaje fata de codul SQL inclus direct in clasele de acces la date:

- in general beneficiaza de performanta sporita, deoarece baza de date poate optimiza planul de executie si il va salva in cache pentru folosiri ulterioare (serverele moderne pot face totusi acelasi lucru si cu query-urile obisnuite);
- procedurile stocate pot fi protejate individual la nivelul bazei de date, rezultand in securitate sporita — unui client ii pot fi date drepturi pe proceduri fara a avea drepturi pe tabelele de sub ele;
- folosirea lor asigura o intretinere mai usoara, deoarece sunt mai usor de actualizat decat alte componente in cazul unei aplicatii distribuite (in plus, se actualizeaza doar serverul SQL, nu toti clientii in parte);
- ele pot adauga un nivel suplimentar de abstractizare fata de schema existenta a bazei de date — un client care le acceseaza nu va trebui sa cunoasca detaliile de implementare ale acestora;
- procedurile stocate reduc traficul pe retea, deoarece nu se trimite tot codul spre a fi executat, ci doar parametrii de intrare si iesire; ele pot fi executate si in mod batch, fara a trimite cereri multiple din partea clientului.

Folosirea lor are si cateva dezavantaje, inasa aceasta se refera la aspecte neexploatate de arhitectura de persistenta. Un exemplu poate fi folosirea extinsa a procedurilor stocate pentru a implementa logica aplicatiei, lucru ce poate duce la o incarcare exagerata a serverului SQL. E recomandat ca logica de aceasta natura sa fie

implementata la nivelul aplicatiei, mai ales la cele distribuite, unde procesarea informatiilor se va face de fiecare client in parte.

3.2.5 Operatii avansate

Am mentionat la inceputul acestei sectiuni ca in sarcina nivelului de acces la date cad mai multe operatii menite sa ascunda detaliile de implementare a legaturii cu serverul SQL, in afara operatiilor legate de manipularea efectiva a datelor. Dintre acestea, cateva merita detaliate, chiar daca in arhitectura prezentata ele nu au fost implementate, fiind considerate prea avansate pentru a avea o utilitate imediata.

Gestionarea conexiunilor la baza de date

Autentificarea si crearea de conexiuni la baza de date se realizeaza automat in clasa de baza pe care o mostenesc toate clasele de acces la date. Aceasta permite pastrarea string-ului de acces intr-un loc comun si deschiderea transparenta de conexiuni, fiecare clasa avand acces la constructorul static din clasa de baza.

Componenta Microsoft folosita pentru conectare gestioneaza automat pool-ul de conexiuni, inasa ele trebuie mentinute deschise cat mai putin cu putinta, pentru a folosi eficient accesul concurent la baza de date. Astfel, gestiunea conexiunilor obtinute din clasa de baza trebuie facuta de fiecare metoda care executa query-uri.

Regula de baza dupa care trebuie sa ne ghidam este *deschide tarziu/inchide devreme* — aceasta asigura incarcarea cat mai mica a serverului. De asemenea, trebuie sa interceptam posibilele erori aparute, testand exceptiile astfel incat sa ne asiguram ca va fi inchisa conexiunea chiar daca a aparut o eroare:

```
SqlConnection connection = Connection;
try {
    // executa comenzi SQL
    ...
}
catch(Exception ex) { throw ex; }
finally { connection.Dispose(); }
```

Seturi mari de rezultate

Atunci cand interogam baza de date pentru a obtine colectii de obiecte, uneori aceste

colectii pot fi foarte mari, de ordinul zecilor, chiar sutelor de mii de inregistrari. Deoarece este foarte probabil sa nu avem nevoie de toate odata, este important de a implementa mecanisme de lucru cu aceste seturi mari de rezultate.

Prima si cea mai usoara abordare este de a limita setul de rezultate direct din query, prin filtrarea dupa anumite criterii. Daca chiar si atunci obtinem prea multe rezultate, putem implementa sisteme de *incarcare la cerere* prin paginare (manual) sau prin mecanisme automate oferite de serverul SQL.

Tratarea erorilor

Este puternic recomandata realizarea unor componente refolosibile de tratare unificata a exceptiilor aparute la nivelul aplicatiei. Aceasta componenta trebuie chemata cat mai usor cu putinta (de obicei printr-o singura linie de cod) si permite chiar jurnalizarea erorilor — astfel incat sa nu se mai intrerupa fluxul aplicatiei cu clasicele dialoguri `MessageBox.Show()`, care oricum nu pot fi prinse bine intr-o aplicatie multi-threading.

Pentru a putea distinge si mai usor intre tipurile de erori, e recomandat sa se mosteneasca clasa `Exception` (oferita de .NET framework) de tipuri separate pentru erorile provenite din cauze tehnice (erori SQL, memorie, etc) si erorile “logice”, produse de aplicatie la validari, atunci cand nu sunt respectate anumite conditii s.a.m.d.

Tranzactii

De multe ori in aplicatii complexe avem nevoie de operatii atomice (vezi exemplul clasic al unui cont bancar interogat simultan din doua locatii), care desi sunt implementate usor la nivel SQL prin blocuri `BEGIN TRANSACTION ... COMMIT`, trebuie gestionate si de catre arhitectura de persistenta.

Pentru a putea folosi modelul tranzactional, apelurile realizate de catre clasele de acces la date trebuie sa fie orchestrate de o componenta externa, care initializeaza si inchide tranzactia. Entitatile business sau clasele de acces la date nu trebuie sa initializeze tranzactii de unele singure, ci doar sa faca parte din acestea, la cerere.

Locking si acces concurent

O componenta extrem de importanta (dar si foarte greu de implementat) a unei arhitecturi de persistenta pentru aplicatii distribuite este cea care permite accesul concurent la informatii de mai multi utilizatori, garantand consistenta datelor.

Daca in cazul operatiilor efectuate pe date deja salvate in baza de date, de multe ori sunt suficiente operatiile atomice oferite de modelul tranzactional, in cazul aplicatiilor bazate puternic pe interactiunea cu utilizatorul acest model nu mai este suficient. Intre momentul in care un utilizator cere un obiect din baza de date si pana cand il modifica si cere salvarea la loc poate trece o perioada nedeterminata, timp in care un alt utilizator a modificat deja obiectul, iar primul ii va suprascrie modificarile celui de-al doilea.

Pentru a acomoda astfel de scenarii de utilizare trebuie implementate mecanisme de locking la nivelul datelor. Acestea sunt de doua feluri:

- **pesimistic concurrency** — cand un utilizator citeste un rand cu intentia de a-l actualiza, pune un lock pe acel rand si nimeni nu-l poate modifica pana cand lock-ul nu este eliberat;
- **optimistic concurrency** — nu se pune lock la citire, in schimb se mentine un istoric al schimbarilor, iar daca acel rand a fost deja modificat intre timp este declansata o eroare.

Modelul pesimist este usor de implementat, inasa nu este viabil pentru procese in care perioada de timp intre citire si actualizare poate fi foarte lunga, sau chiar sa nu se mai intample niciodata in cazul caderii unei conexiuni (cum se intampla in cazul aplicatiilor web).

Pe de alta parte, modelul optimist face fata acestor provocari, inasa este mult mai greu de implementat, deoarece presupne modificari la nivelul schemei informationale sau clauze `WHERE` complicate. Exista doua metode principale pentru a implementa modelul optimist:

1. prin adaugarea unei coloane de tip `timestamp` fiecare tabele, care este actualizata cu ora exacta (printr-un trigger) de cate ori se actualizeaza

continutul coloanei — operatia esueaza daca data din obiectul de salvat difera de data pe care avut-o initial;

2. prin includerea in clauza `WHERE` a tuturor coloanelor obiectului, nu doar a cheii primare; daca acestea difera de cele cunoscute anterior, inseamna ca obiectul a fost modificat intre timp.

Caching

Aceasta componenta este atat de complexa incat merita o lucrare de acest gen doar pentru ea — voi mentiona insa foarte pe scurt despre ce este vorba.

Este posibil de a reface schema informationala a bazei de date intr-un mecanism de stocare virtual, la nivelul aplicatiei. Aici sunt salvate datele deja cerute din baza de date, astfel incat se poate face economie de trafic intre serverul SQL si masina pe care lucreaza aplicatia.

O alta aplicatie a unui nivel intermediar de caching este posibilitatea de lucra complet deconectat de serverul SQL (in mod *offline*), util de ex. pentru agenti care lucreaza in teritoriu si nu au o conexiune permanenta la intranetul firmei. Astfel programul retine toate modificarile facute in mod offline si initiaza un proces de sincronizare cu serverul central atunci cand detecteaza existenta unei conexiuni disponibile.

3.3 Proiectarea claselor de persistenta

Atunci cand spunem **clase de persistenta**, ne referim de fapt la **entitatile *business***, folosite la manipularea si transferul informatiilor intre nivelurile aplicatiei, dar si la persistarea (salvarea propriuzisa) acestor informatii in baza de date. In functie de cerintele aplicatiei, ele pot fi reprezentate in diverse feluri, cum ar fi XML, DataSets sau clase specifice, orientate pe obiecte.

3.3.1 Reprezentarea entitatilor business

Cand proiectam entitatile necesare pentru manipularea datelor, avem de ales dintre urmatoarele posibilitati (de preferat, o singura metoda, sau un numar cat mai mic, pentru a mentine aplicatia usor de intretinut):

- **XML** (*Extensible Markup Language*) sau **DOM** (*Document Object Model*) — un format flexibil, standardizat, larg raspandit —, util mai ales la schimbul de informatii intre aplicatii, prin intermediul serviciilor.
- **DataSet**. Un *DataSet* este o replica tinuta in memorie a tabelelor SQL sau XML, fiind un tip predefinit in *.NET framework*. Este usor de obtinut prin query-uri si se preteaza mai ales atunci cand manipulam colectii numeroase de obiecte, insa nu i se poate adauga usor logica suplimentara, cum ar fi validari extinse.
- **Typed DataSet**. Un tip mostenit din tipul *DataSet*, mentionat mai sus, insa prezinta avantajele obiectelor *strongly-typed* si poate fi extins mai usor.
- **Clase de persistenta/entitati business**. Acestea reprezinta clase *.NET* propriuzise, care au campuri ascunse pentru a memora valorile din coloanele SQL, expun proprietati pentru a manipula aceste valori si metode simple pentru a implementa logica aplicatiei. Pentru a obtine si salva obiectele instantiate se apeleaza direct nivelul de acces la date.
- **Clase de persistenta cu operatii CRUD incapsulate**. Ultima si cea mai complexa metoda este si cea pe care am implementat-o in aceasta arhitectura. Ea foloseste clasele de persistenta mentionate la punctul anterior, insa le adauga metode care implementeaza direct la nivelul obiectelor apeluri catre nivelul de acces la date, incapsuland efectiv acest comportament.

Avantajele oferite de solutia aleasa sunt numeroase si vor fi discutate pe larg in continuare. Prima consecinta a incapsularii apelurilor catre nivelul de acces la date este ca nu mai este necesar sa chemam direct acest nivel, iar **entitatile business** devin **clase de persistenta** in toata regula, capabile sa se gestioneze singure. Iata acest comportament in actiune:

Folosirea claselor de persistenta cu si fara operatii CRUD

```
// fara metode CRUD
Angajat a = AngajatDB.Get(23); // instantiem direct angajatul cu
id'ul 23
a.num = 'Ion'; // actualizam numele
AngajatDB.Save(a); // salvam modificarile

// metode CRUD incapsulate
```

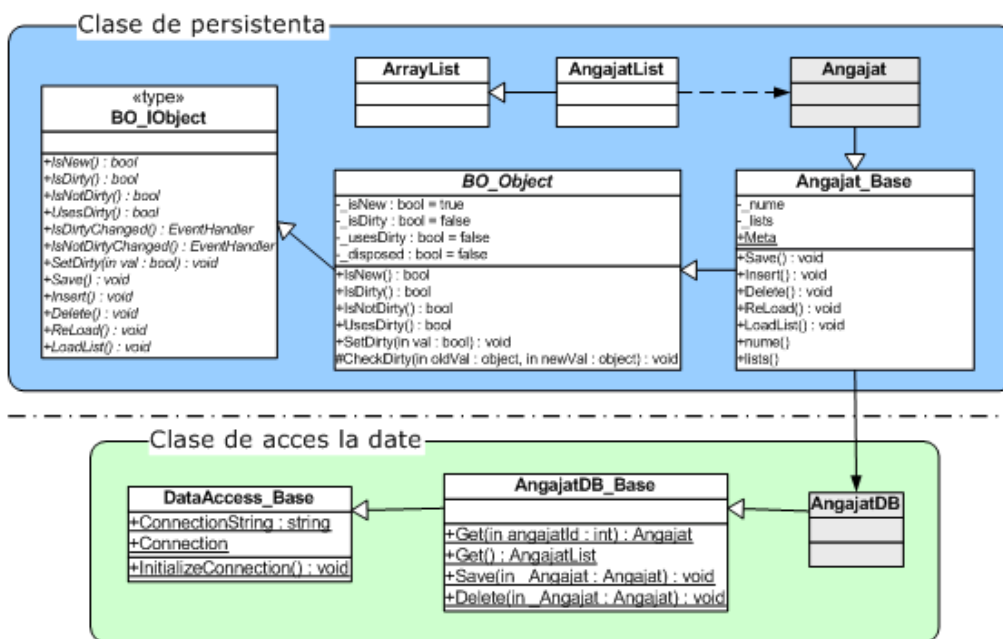
```

Angajat a = new Angajat(23);
a.ume = 'Ion';
a.Save();
    
```

3.3.2 Ierarhia de clase

Clasele de persistenta implementeaza o interfata pentru a garanta setul comun de operatii disponibile si sunt ierarhizate pe trei nivele de mostenire:

1. interfata care defineste contractul pe care toate clasele se angajeaza sa-l respecte (BO_IObject);
2. o clasa de baza abstracta care contine cod comun, util tuturor claselor (BO_Object);
3. o clasa de baza concreta in care sta tot codul generat, specific fiecarei table (ex. Angajat_Base);
4. o clasa externa care "imbraca" clasa de baza concreta, in care se poate adauga logica aplicatiei (ex. Angajat).



Ca si la nivelul de acces la date, acest mecanism evita pe de o parte repetarea de cod, definind codul comun tuturor claselor cat mai jos cu putinta in ierarhie, iar pe de alta parte ofera o arhitectura flexibila, care permite adaugare de logica particulara, fara a fi in pericol atunci cand regeneram clasele.

3.3.3 Operatiile implementate

Principalul motiv pentru care toate obiectele implementeaza o interfata nu este *garantarea* propriuzisa a unui set comun de functii — acest deziderat este usor asigurat avand in vedere ca folosim unelte de generare de cod. Acest mecanism devine cu adevarat util atunci cand in aplicatie putem trata toate obiectele ca fiind de tip `BO_IObject` — putem realiza module, componente care pot opera pe toate entitatile business fara sa stie cu adevarat ce obiect manipuleaza: liste de obiecte, ferestre de editare, etc.

In **clasa de baza abstracta** am definit cateva comportamente utile tuturor obiectelor:

- *IsDirty*, mecanism prin care stim daca un obiect a fost marcat ca “murdar”, cu alte cuvinte daca a fost modificat si trebuie salvat in baza de date;
- *IsNew*, un flag care indica daca un obiect exista anterior (daca a fost incarcat din baza de date) sau daca este nou, abia instantiat — ne ajuta de exemplu la a hotari daca trebuie sa facem `INSERT` sau `UPDATE`;
- *event*-uri care semnaleaza aplicatiei atunci cand valoarea unui camp al obiectului se schimba, util la legarea (*data+binding*) obiectelor de interfata aplicatiei.

Alte operatii comune tuturor obiectelor, declarate in interfata si implementate (prin generare de cod) in **clasa de baza concreta**, includ:

- metode pentru **incarcarea, salvarea, stergerea sau validarea automata** a obiectelor;
- **metainformatii** incarcate din baza de date, implementate sub forma unei *colectii statice* (unica pentru toate obiectele de acelasi tip) de proprietati ale fiecarei coloane (tip, lungime, valoare default, constrangeri) — folosita pentru o serie de validari automate a datelor primite;
- metode de incarcare a obiectelor sau colectiilor de obiecte **legate** de obiectul curent prin **relatii de asociere sau compunere**;

Clasa concreta finala, cea care va fi folosita de programatori in cadrul aplicatiei, incapsuleaza clasa concreta de baza si permite:

- definirea de logica suplimentara si validari complexe, particulara fiecarui tip de entitate;
- suprascrierea metainformatiilor extrase automat din baza de date, util de exemplu pentru a adauga *etichete* fiecarei coloane.

Reprezentarea colectiilor

Un element important in reprezentarea entitatilor business este reprezentarea colectiilor de entitati. Acestea sunt folosite pentru citirea si manipularea seturilor de informatii, primite atunci cand un query intoarce mai multor randuri simultan.

Avand in vedere ca pentru reprezentarea unei singure entitati am ales definirea de clase de persistenta, pentru reprezentarea colectiilor am ales sa exind tipul `ArrayList` oferit de platforma .NET, implemendand clase *strongly-typed*. Avantajele acestei abordari sunt:

- refolosirea unei cantitati de cod impresionante, implementate deja in .NET;
- eliminarea necesitatii de a converti in permanenta de la tipul `Object` la tipul concret folosit, daca am fi folosit direct colectii generice de obiecte `ArrayList`;
- posibilitatea de realiza legarea (*data-binding*) colectiilor de elemente de interfata care stiu sa afiseze direct toate campurile fiecarui obiect, lucru imposibil in cazul unui simplu `ArrayList`.

3.3.4 Modelul avansat al claselor de persistenta

In afara de operatiile aproape obligatorii pe care trebuie sa le implementeze clasele de persistenta, exista si operatii care desi nu sunt esentiale, aduc un avantaj arhitecturii de persistenta.

Puterea metainformatiilor: validari automate

Cele mai multe servere de baze de date expun foarte multe *meta-informatii* (adica informatii despre informatii) despre datele pe care le stocheaza. Schema informationala a fiecarei baze de date poate fi explorata, pana acolo incat aceasta se

poate reconstrui fara sa stim cum a fost creata (proces denumit uneori, de obicei in situatii mai complicate, *reverse-engineering*).

Folosind un generator de cod capabil putem extrage informatii despre toate aspectele unei baze de date: tabele si view-uri, tipul coloanelor, constrangeri stabilite, indecsi, chei primare sau chei straine, relatiile dintre tabele s.a.m.d.

Salvand toate aceste informatii intr-un format accesibil la runtime (fie el un fisier XML sau o colectie statica), se pot realiza multe lucruri, dintre care cel mai util este de departe validarea automata a datelor introduse. In modelul folosit in aceasta arhitectura, despre fiecare coloana a fiecarei tabele sunt extrase anumite date: tipul coloanei, lungime maxima (pentru string-uri), semnul sau precizia (pentru numere), intervale de valori acceptate (pentru numere sau date) si altele.

Chiar daca nu toate tipurile de date de intrare pot fi validate complet prin doar cateva informatii (*ex*: CNP-ul unei persoane), acestea sunt suficiente pentru a nu accepta date pe care baza de date nu le va putea stoca. Erori comune sunt string-uri mai lungi decat capacitatea coloanei, valori din afara intervalului sau trimiterea de caractere in locul cifrelor. De cele mai multe ori serverul SQL nu semnaleaza erori in aceste cazuri, inasa face conversii implicite care genereaza date salvate gresit sau incomplet.

Obiectele sunt astfel construite incat *setter*-ele incearca sa valideze datele primite, iar in cazul unui esec ridica o exceptie care trebuie tratata. Pentru a putea semnala exceptia cat mai devreme cu putinta, ea este ridicata chiar in momentul atribuirii, nu tocmai la salvare — astfel ea poate fi interceptata direct de interfata. Managementul exceptiilor se poate face folosind acelasi mecanism ca si in cazul nivelului de acces la date.

Evenimente si data binding

Am mentionat mai sus proprietatea *IsDirty* a fiecarui obiect, care specifica daca acesta a fost modificat. Aceasta este gestionata automat, tot prin intermediul *setter*-ului fiecarui camp, odata cu validarea. Daca se detecteaza schimbari in obiect, ele vor fi semnalate prin ridicarea unui eveniment (*event*) care poate fi interceptat de interfata.

Utilitatea acestui comportament vine atunci cand folosim una dintre cele mai utile functionalitati ale platformei .NET, *data-binding*-ul. Acesta consta in posibilitatea de a lega intre ele proprietatile anumitor obiecte, cum ar fi numele unui angajat de valoarea unui control de introducere text. In acest fel, printr-o singura declaratie ne asiguram ca cele doua campuri vor ramane sincronizate, fara a trebui sa facem explicit acest lucru.

Pana la aparitia *data-binding*-ului trebuiau implementate evenimente ale controlului de editare text, care transferau modificarile aparute in control catre obiect, insa toate operatiile trebuiau gestionate manual. In plus, mecanismul functiona intr-un singur sens, de la control catre obiect, pe cand *data-binding*-ul poate functiona in ambele sensuri (*two-way data-binding*) daca ambele obiecte au implementate evenimentele necesare:

- **de la control spre obiect:** controlul are evenimente care detecteaza modificarea valorii si scrie propria valoare in obiectul de care este legat;
- **de la obiect spre control:** prin evenimentul `IsDirtyChanged` se semnaleaza controlului ca s-a modificat programatic valoarea obiectului si ca trebuie sa reimprospateze valoarea afisata.

Data-binding intre numele unui angajat si un control text

```
Angajat a = new Angajat();  
control.DataBindings.Add("Value", a, "nume");
```

Tratarea valorilor NULL

In lucrul cu bazele de date este adeseori nevoie de a stoca valoarea `NULL`, al carei sens este "informatie inexistentă", dar insasi lipsa informatiei reprezinta o informatie, astfel incat aceasta valoare trebuie stocata ca atare. Este gresita practica unora de a stoca stringul vid, 0, -1 sau orice valoare considerata ca nu va aparea niciodata in loc de valoarea `NULL`, creata special in acest scop.

Din pacate platforma .NET (pana la versiunea 1.1) nu are implementata la nivelul limbajului sintaxa necesara reprezentarii valorilor `NULL`, mai ales in raport cu mecanismul de *data-binding* prezentat mai sus. Niciunul dintre tipurile primitive nu

poate stoca valoarea `NULL`, ea insemnand ca obiectul respectiv nu este instantiat. Principala problema vine din faptul ca nu se poate pastra tipul obiectului atunci cand plimbam datele intre nivelurile aplicatiei.

Pentru a trece peste aceste inconveniente o componenta open-source, *NullableTypes*. Aceasta ofera o alternativa pentru fiecare din tipurile .NET standard, imbogatindu-le cu posibilitatea stocarii valorii `NULL` si cu mecanisme de conversie intre tipuri.

Capitolul 4 Implementare si utilizare

In capitolele anterioare am discutat despre partea teoretica a unei arhitecturi software si am studiat detaliile proiectarii arhitecturii de persistenta. In acest capitol vom analiza tehnologiile si uneltele folosite si vom prezenta detaliat API-ul arhitecturii de persistenta, incluzand si exemple de cod pentru toate operatiile implementate.

4.1 Tehnologii folosite

Pe scurt, arhitectura de persistenta este realizata pentru a fi folosita de aplicatii scrise pentru platforma .NET in tandem cu Microsoft SQL Server. In afara de acestea, mai sunt folosite cateva aplicatii si tehnologii care merita mentionate.

4.1.1 Platforma .NET

Asa cum am precizat in capitolul introductiv, motivatia principala pentru scrierea unei arhitecturi de persistenta a fost o aplicatie complexa pe care am fost ales sa o dezvolt in anul 2004. Alegerea platformei .NET a venit oarecum de la sine, pentru ca in cadrul firmei se foloseau de mult tehnologii Microsoft. Chiar daca platforma .NET era atunci abia la inceput, ea a fost preferata tehnologiilor mai vechi, pentru ca ea reprezenta clar viitorul aplicatiilor scrise pentru sistemul de operare Windows.

Limbajul ales a fost **C#** (*C sharp*), fiind probabil cel mai potrivit dintre cele disponibile pentru platforma .NET: *C++*, *J#* si *Visual Basic*.

Componente Microsoft pentru .NET framework

In afara de librariile standard oferite de platforma .NET am mai folosit cateva componente (denumite *Application Blocks* de catre cei de la Microsoft), pentru a usura munca programatorilor si pentru a refolosi cod existent si de calitate, testat in conditii foarte variate. Acestea sunt:

- *Microsoft Data Access Application Block* — folosit ca interfata pentru accesul la SQL Server, acesta ofera metode pentru a realiza o conexiune cu baza de date si pentru a executa query-uri;
- *Microsoft Exception Management Application Block* — folosit pentru gestionarea exceptiilor aparute in aplicatie, ofera un mod unitar de a intercepta si trata exceptiile, fiind posibila si jurnalizarea acestora;
- *Microsoft Updater Application Block* — precursorul tehnologiei *One-Click Deployment*, aceasta componenta ofera functii necesare actualizarii aplicatiei prin internet, odata ce a fost instalata la client.

4.1.2 Microsoft SQL Server

Ca server de baze de date am ales *Microsoft SQL Server*. Motivele au fost din nou dictate oarecum de experienta avuta de cei care au lucrat la acest proiect cu SQL Server si de interoperabilitatea mai buna cu platforma .NET, fiind complet suportat de aceasta inca de la primele versiuni.

Un al motiv pentru care am ales *SQL Server* este disponibilitatea unei versiuni gratuite care poate fi distribuita fara alte obligatii impreuna cu aplicatia, **MSDE** (*Microsoft Desktop Engine*). Aceasta are cateva limitari la utilizare (2 GB/baza de date, executare pe un singur procesor, foloseste maxim 1 GB de RAM), insa pentru aplicatii de dimensiuni obisnuite nu reprezinta o problema.

Avantajul cel mai mare al *MSDE* este inasa comportarea si un set de functii identic cu *SQL Server*, ceea ce ofera o cale de upgrade absolut transparenta, in cazul in care nevoile aplicatiei cresc.

4.1.3 Generatoare de cod

In afara de tehnologiile alese, prezentate mai sus, exista un set de unelte fara de care aceasta arhitectura nu ar fi putut fi implementata. Acestea sunt generatoarele de cod *MyGeneration* si *CodeSmith*, ambele programe gratuite si *open-source*, folosite de dezvoltatori din intreaga lume pentru a automatiza procesul generarii de cod plecand de la schema unei baze de date.

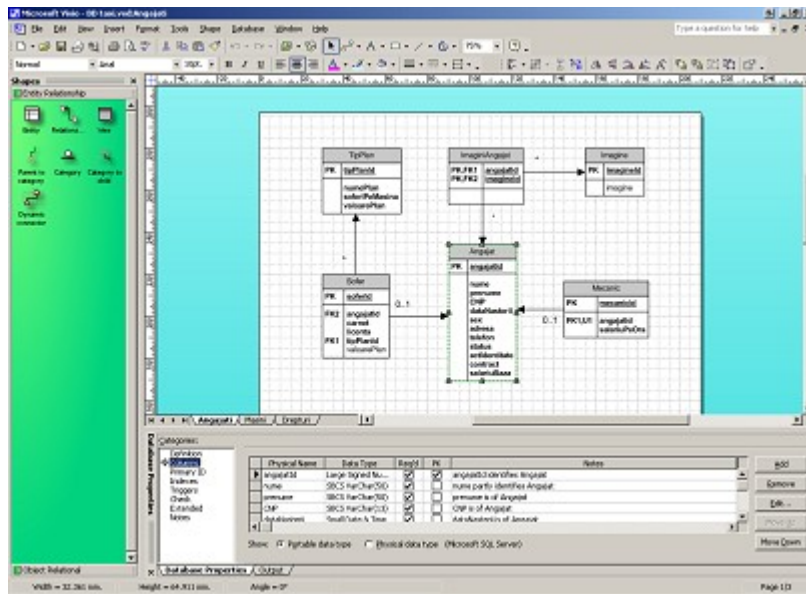
Principalele lor avantaje sunt existenta unor API-uri care abstractizeaza toate bazele de date suportate, permitand interogarea tuturor meta-informatiilor expuse de acestea: tabele, chei, indecsi, relatii, constrangeri *s.a.m.d.*

Procesul generarii de cod este cat se poate de simplu, bazandu-se pe niste scripturi (template-uri) care parcurg baza de date si genereaza fisierele necesare claselor (cod C#) si procedurilor stocate (cod SQL). Acest proces poate fi automatizat astfel incat sa fie generate toate clasele printr-un singur click, singura grija avuta fiind sa nu se suprascrie fisiere modificate de mana ulterior unei generari anterioare.

4.1.4 Proiectare vizuala cu Visio

In primele faze ale dezvoltarii aplicatiei este necesara construirea schemei informationale, mai precis a proiectarii bazei de date. Cea mai potrivita unealta s-a potrivit a fi Microsoft Visio, o aplicatie folosita pentru realizarea de diagrame in cele mai diverse domenii, de la Arhitectura Informatiei si Procese Business pana la Arhitectura si Electrotehnica.

Pentru designul aplicatiilor software sunt disponibile diagrame prin care se pot modela toate aspectele, pentru ca este implementat intreg setul de specificatii UML. Componenta cea mai utila este insa cea care permite proiectarea unei baze de date folosind elemente ale schemelor Entitate-Relatie. In figura 4.1 vedem ecranul principal al aplicatiei Visio:



Dintre functiile mai avansate oferite de Visio cea mai importanta este posibilitatea de a implementa baza de date implementata sub forma de diagrama ER, prin conectarea la serverul SQL si crearea tuturor tabelelor si relatiilor dintre acestea. Aceasta posibilitate anuleaza practic diferentele dintre un prototip si baza de date reala, scutrand considerabil timpul de proiectare. In plus, diagramele realizate sunt foarte bune pentru documentarea ulterioara a proiectului.

4.2 Implementarea arhitecturii de persistenta

In aceasta sectiune vom prezenta intreg API-ul pus la dispozitie de clasele de persistenta si vom vedea cum sunt implementate toate operatiile prezentate in capitolul anterior, de la entitatile business pana la procedurile stocate SQL.

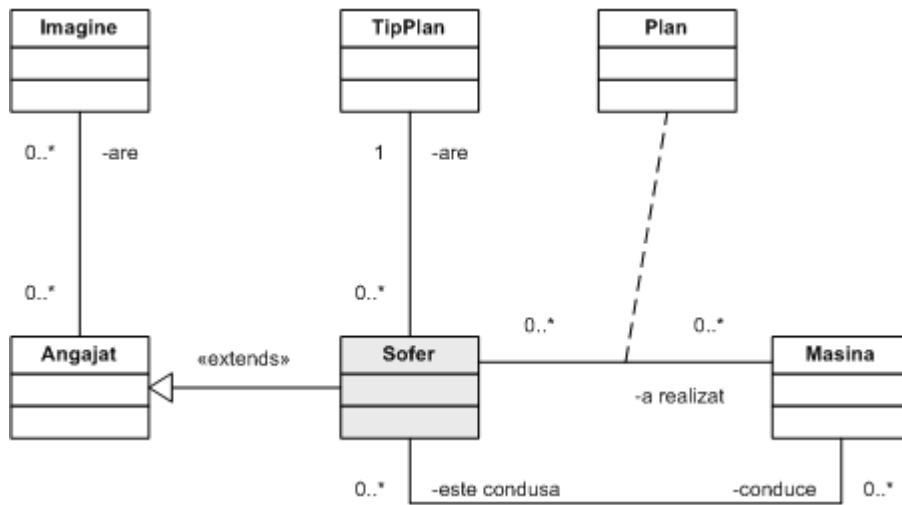
4.2.1 Schema exemplu

Pentru a pune in valoare arhitectura voi folosi o parte din baza de date alcatuita din tabelele legate de conceptul de Sofer. Toate operatiile prezentate vor face referiri la urmatoarea schema.

Diagrama UML

In Figura 4.1 puteti vedea o reprezentare simplificata (in *UML*) a entitatilor implicate:

4.2 Implementarea arhitecturii de persistenta



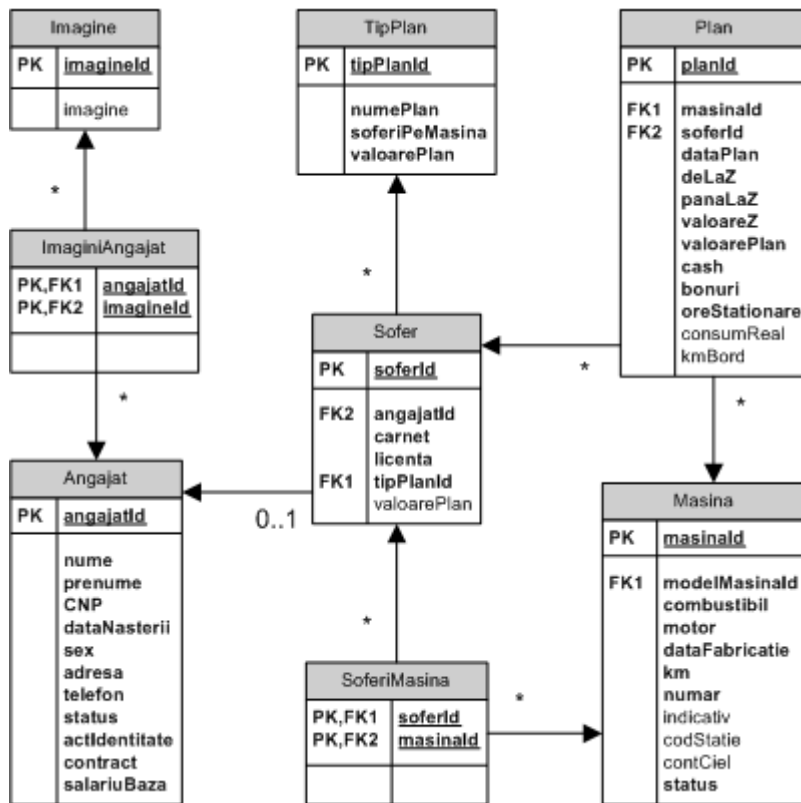
De remarcat diversitatea de tipuri de entitati si relatii prezente in diagrama:

- clase *simple* (Imagine, Angajat, Masina, TipPlan);
- clase *extinse* (Sofer extinde Angajat);
- relatii **1:1** (Sofer → Angajat);
- relatii **1:N** (Sofer → TipPlan);
- relatii **N:M** (Imagine → Angajat, Sofer → Masina);
- relatii **N:M cu attribute** (Plan zilnic prin relatia Sofer → Masina).

Diagrama relationala (SQL)

Daca traducem diagrama UML de mai sus intr-o schema relationala, gata de transpus in baza de date, obtinem urmatoarea schema:

4.2 Implementarea arhitecturii de persistenta



Principalele transformari suferite sunt urmatoarele:

- implementarea relatiilor *many-to-many* prin tabele de legatura (ImaginiAngajat, SoferiMasina);
- transformarea relatiei cu attribute dintre Soferi si Masini in entitatea de sine statatoare Plan, care printre attributele relatiei are si doua chei straine.

4.2.2 Clasele de persistenta

Vom incepe cu entitatile business, pentru ca acestea sunt blocurile de baza ale arhitecturii, entitatile folosite pentru a reprezenta si transporta informatia intre diversele niveluri ale aplicatiei.

De-a lungul lucrarii am alternat folosirea notiunilor de entitate business cu cea de clasa de persistenta pentru a ne referi la nivelul superior al arhitecturii. In continuare vom prefera denumirea romaneasca, clase de persistenta, chiar daca folosita intr-un context mai general ea include si clasele din nivelul de acces la date.

Fiecare clasa de persistenta este de fapt o mapare a unei tabele SQL din modelul relational al bazei de date in modelul orientat pe obiecte al arhitecturii de persistenta.

O clasa de persistenta este alcatuita din:

- campuri pentru **stocarea datelor** (cate un camp pentru fiecare coloana a tablei);
- campuri pentru **reprezentarea relatiilor** (*one-to-many* si *many-to-many*);
- metode pentru **cautarea si persistarea datelor**, care incapsuleaza apeluri catre nivelul de acces la date;
- *flag-uri* pentru a urmari starea obiectului (nou, modificat);
- **evenimente** pentru a semnala modificari ale continutului obiectului;
- **metainformatii** si **logica de validare** a datelor;
- metode pentru **persistarea relatiilor** de tip *many-to-many*.

Dupa cum am precizat in capitolul anterior, acestor campuri si metode se adauga logica din clasa de baza, comuna tuturor si clasa concreta care incapsuleaza clasa abstracta generata in intregime. Toata logica particulara, scrisa “de mana” va fi scrisa in clasa concreta, pentru a preveni suprascrierea codului la o eventuala regenerare. Sa prezentam pe rand componentele de mai sus (codul complet al clasei este prezentat in *Anexa B*).

Stocarea informatiilor

Fiecare clasa are un set de campuri pentru stocarea datelor din SQL, cate unul pentru fiecare coloana a tablei corespondente. Acestea sunt declarate `protected` si sunt accesate prin proprietati declarate `public`. Proprietatile sunt un constructor al limbajului *C#* asemanator metodelor *getter* si *setter* din Java, insa cu avantajul ca din exteriorul obiectelor se comporta ca niste campuri. Deoarece din exterior sunt vazute ca niste campuri pot fi folosite la databinding, dar in realitate fiind metode, acestea ne permit sa realizam validari si sa schimbam starea obiectului prin simple atribuirii.

Pentru exemplificare, sa luam tabela `Angajat`:

```
// campuri private din tabela
protected int _angajatId;
```

```
protected string _nume = String.Empty;
protected string _prenume = String.Empty;
protected string _CNP = String.Empty;
protected DateTime _dataNasterii = DateTime.Now;
protected bool _sex;
protected string _adresa = String.Empty;
protected string _telefon = String.Empty;
protected byte _status;
protected string _actIdentitate = String.Empty;
protected string _contract = String.Empty;
protected decimal _salariuBaza;

// campuri private din relatii (1:N, N:1, N:M)

private ImagineList _imagineList;
private MecanicList _mecanicList;
private SoferList _soferList;
private UtilizatorProgramList _utilizatorprogramList;
```

Campurile provenite din tabela au tipurile potrivite dupa o corespondenta intre tipurile SQL si tipurile .NET (tabela completa in *Anexa A*), iar valorile sunt initializate automat la valoarea default a tipului respectiv. Campurile provenite din relatii vor fi discutate pe larg in sectiunea urmatoare.

Toate proprietatile sunt implementate astfel:

```
// campul angajatId
public virtual int angajatId {
    get { return this._angajatId; }
    set {
        Meta["angajatId"].Validate(value);
        if (this.UsesDirty) { this.CheckDirty(this._angajatId, value); }
        this._angajatId = value;
    }
}
```

Se remarca din cod ca in timp ce metoda `get` nu face decat sa intoarca valoarea campului `protected`, metoda `set` are mai multe roluri:

- validarea valorii primite — in caz de esec este generata o exceptie;
- schimbarea starii obiectului in `Dirty` daca se primeste o valoare diferita de cea veche;
- tot aici este generat un evident pentru a anunta schimbarea;
- in sfarsit, setarea noii valori in obiect.

Constructori

Clasele de persistenta au 3 constructori in clasa de baza concreta:

- unul care doar instantiaza un obiect nou, gol;
- unul care duplica un obiect deja existent;
- unul care instantiaza un obiect deja existent, incapsuland un apel catre cautarea dupa cheie primara din nivelul de acces la date.

```
public class Angajat_Base : BO_Object {
    ...
    // Base class constructor that creates a new Angajat object
    public Angajat_Base() {
    }

    // Base class constructor that duplicates a Angajat object
    public Angajat_Base(Angajat_Base _Angajat) {
        this.Set(_Angajat);
    }

    // Base class constructor that instantiatens an existing
    // Angajat object from the DB
    public Angajat_Base(int angajatId) {
        this.Set(Taxi.DataAccess.AngajatDB.Get(angajatId));
    }
    ...
}
```

Obiectele pot fi astfel instantiate in modul urmator:

```
// obiect nou
Angajat a = new Angajat();

// din baza de date
Angajat a = new Angajat(1);

// duplicare obiect existent
Angajat b = new Angajat(a);
```

Pentru fiecare dintre acesti constructori exista in clasa concreta cate un constructor care-l apeleaza mai intai pe cel din clasa abstracta, permitand insa adaugarea de logica suplimentara:

```
public class Angajat : Angajat_Base {
    ...
    public Angajat(int angajatId):base(angajatId) {
        // extra constructor logic here
        ...
    }
    ...
}
```

```
}
```

Metode publice

Cele cateva metode publice ale obiectului nu fac decat sa incapsuleze apeluri catre nivelul de acces la date, testand totodata daca obiectul chiar a fost modificat inainte de a-l salva:

```
// salvarea unui Angajat
public override void Save() {
    if (!this.UsesDirty || this.IsDirty) {
        Taxi.DataAccess.AngajatDB.Save(this);
    }
}
```

Aceste metode usureaza manipularea obiectelor, fiind mult mai usor sa scriem

```
a.Save();
```

decat

```
AngajatDB.Save(a);
```

Sunt generate metode pentru salvarea (INSERT sau UPDATE), stergerea sau resetarea (reincarcarea ultimelor valori salvate in baza de date) a unui obiect. Restul de metode pentru gasirea obiectelor sunt implementate ca fabrici de obiecte prin metode statice din nivelul de acces la date.

Starea obiectului, evenimente

Atunci cand modificam un camp al unui obiect, metoda get intercepteaza comanda si testeaza daca noua valoare este diferita de cea veche, in caz afirmativ modificand proprietatea `IsDirty` declansand un eveniment de schimbare, prin care eventuale controale legate de acest camp prin *data-binding* pot cere o reimprospatare a valorii afisate.

Acest cod este comun tuturor obiectelor si se afla in clasa abstracta de baza `BO_Object`:

```
// evenimentul care semnaleaza schimbarea
public event EventHandler IsDirtyChanged;

// marcheaza obiectul ca alterat
```

```
public void SetDirty(bool val) {
    this._isDirty = val;
    if (IsDirtyChanged != null) {
        IsDirtyChanged(this, new EventArgs());
    }
}

// verifica daca s-a trimis o valoare noua
public void CheckDirty(object oldVal, object newVal) {
    bool changed = false;
    if (oldVal == null) {
        if (newVal != null) {
            changed = true;
        }
    } else {
        changed = !oldVal.Equals(newVal);
    }

    if (changed) {
        this.SetDirty(true);
    }
}
```

Metainformatii si validare

Una dintre cele mai avansate tehnici folosite de clasele de persistenta este validarea automata a datelor stocate, asa cum a fost prezentata si in capitolul anterior. Ea se bazeaza pe meta-informatiile stocate (si expuse) de baza de date, informatii despre toate tabelele si relatiile folosite. Aceste meta-informatii sunt extrase de generatorul de cod si salvate in fiecare clasa, cate un set de informatii pentru fiecare tabela/coloana.

Metoda de implementare aleasa este de a folosi cate o colectie de elemente `BO_FieldMeta` declarata static pentru fiecare clasa in parte. Aceasta colectie este populata intr-un constructor static al clasei cu cate o intrare pentru fiecare camp al clasei/coloana a tabeli, intrucat fiecare camp este validat separat in momentul atribuirii. Constructorul static e garantat sa fie apelat inainte de instantierea primului obiect din acel tip, astfel incat procesul este executat o singura data, iar meta-informatiile vor fi disponibile tuturor instantelor.

Aceasta metoda are avantajul de a fi foarte rapida in momentul executiei, datorita colectiei statice, dar mai ales a faptului ca toate informatiile sunt interpretate la compilare si disponibile sub forma binara, spre deosebire de alte metode care implica

parsarea la *runtime* de scheme definite in XML.

Iata constructia acestui obiect pentru cateva dintre campurile tabelii Angajat:

```
// angajatId - camp numeric
Meta["angajatId"] = new BO_FieldMeta();
Meta["angajatId"].fieldType = typeof(int);
Meta["angajatId"].isNullable = false;
Meta["angajatId"].columnName = "angajatId";
Meta["angajatId"].columnAlias = "angajatId";
Meta["angajatId"].maxLength = 0;
Meta["angajatId"].precision = 10;
Meta["angajatId"].scale = 0;
Meta["angajatId"].isNumeric = true;
Meta["angajatId"].minValue = (double)int.MinValue;
Meta["angajatId"].maxValue = (double)int.MaxValue;

// nume - string de caractere
Meta["nume"] = new BO_FieldMeta();
Meta["nume"].fieldType = typeof(string);
Meta["nume"].isNullable = false;
Meta["nume"].columnName = "nume";
Meta["nume"].columnAlias = "nume";
Meta["nume"].maxLength = 50;
Meta["nume"].precision = 0;
Meta["nume"].scale = 0;

// dataNasterii - data calendaristica
Meta["dataNasterii"] = new BO_FieldMeta();
Meta["dataNasterii"].fieldType = typeof(DateTime);
Meta["dataNasterii"].isNullable = false;
Meta["dataNasterii"].columnName = "dataNasterii";
Meta["dataNasterii"].columnAlias = "dataNasterii";
Meta["dataNasterii"].maxLength = 0;
Meta["dataNasterii"].precision = 0;
Meta["dataNasterii"].scale = 0;

// sex - valoare booleana
Meta["sex"] = new BO_FieldMeta();
Meta["sex"].fieldType = typeof(bool);
Meta["sex"].isNullable = false;
Meta["sex"].columnName = "sex";
Meta["sex"].columnAlias = "sex";
Meta["sex"].maxLength = 0;
Meta["sex"].precision = 0;
Meta["sex"].scale = 0;
```

Dupa cum se vede, colectia este de fapt un array asociativ indexat dupa numele fiecarei coloane, pentru o reprezentare cat mai sugestiva. Nu vom prezenta aici intreaga structura a clasei `BO_FieldMeta` (este disponibila in anexa) insa iata ce functii indeplineste:

- retine toate proprietatile coloanelor SQL si ale campurilor claselor: nume, tip SQL, tip C#, daca poate fi null, alias coloana (folosit pentru etichete), daca este tip numeric, dimensiune, precizie, interval de valori acceptat;
- are cateva proprietati prin care poate fi imbogatita interfata aplicatiei, cum ar fi o masca de caractere acceptate (generata automat) folosita de controalele de introdus text;
- efectueaza validarea valorilor atribuite campurilor, in trei pasi:
 1. **complet automat**: tipul, dimensiunea sau capacitatea de stocare a coloanei, interval de valori pentru tipuri numerice sau date calendaristice (pentru a nu permite programului sa accepte date care nu vor putea fi stocate mai tarziu in baza de date);
 2. particularizare printr-o **expresie regulata** pentru siruri de caractere (util pentru adrese de email, numere de telefon, etc);
 3. printr-o **metoda custom** (specifica unei singure coloane), scrisa de mana si chemata la sfarsitul validarii automate (folosita de exemplu la validarea CNP-ului, care se face printr-un algoritm special).

Toate erorile de validare sunt semnalate folosind exceptii, o componenta importanta a limbajelor moderne.

Mecanismul de validare este asadar suficient de puternic incat sa faca majoritatea validarilor automat, insa in acelasi timp suficient de flexibil incat sa poata acomoda orice validare necesara. Sa presupunem ca vrem sa acceptam doar nume proprii care se termina in sufixul *escu*. Vom scrie de mana o clasa de validare (din motive didactice ignoram solutia banala, prin expresii regulate), pe care o punem in clasa concreta *Angajat*:

```
public class Angajat : Angajat_Base {
    ...
    // testarea numelui
    public class NumeRomanesc : BO_IFieldValidator {
        public void Validate(object val) {
            string nume = val as string;
            if (!nume.EndsWith("escu")) {
                throw new Exception("Numele nu este romanesc ;");
            }
        }
    }
}
```

```
}  
...  
}
```

Aceasta a trebuit sa implementeze interfata `BO_IFieldValidator`, care garanteaza existenta metodei `Validate`:

```
public interface BO_IFieldValidator {  
    void Validate(object val);  
}
```

Pentru a putea folosi aceasta validare, trebuie sa instiintam obiectul prin setarea campului `extendedValidator`, in constructorul static al clasei concrete:

```
static Angajat() {  
    ...  
    Meta["nume"].extendedValidator = new NumeRomanesc();  
    ...  
}
```

Iar testul se face printr-o simpla atribuire:

```
try {  
    a.numa = "Tanase"; // va declansa exceptia  
}  
catch (Exception e) {}
```

4.2.3 Persistarea obiectelor

Operatiile urmatoare se ocupa de salvarea obiectelor, de actualizarea informatiilor, si de stergerea acestora din baza de date.

Salvarea

Operatia de salvare include ambele operatii SQL: `INSERT` si `DELETE`. Daca obiectul este nou (nu are cheie primara) se va executa inserarea, daca exista se va executa update-ul. Sa luam exemplul clasei `Angajat`.

```
// adaugarea unui angajat  
Angajat a = new Angajat();  
a.numa = "Popescu";  
a.prenume = "Ion";  
...  
a.Save();  
  
// modificare nume
```

```
a.nume = "Georgescu";  
a.Save();
```

Apelul `a.Save()` se va traduce prin:

```
Taxi.DataAccess.AngajatDB.Save(this);
```

care la randul sau contine urmatoarea bucata de cod:

```
if (_Angajat.angajatId != 0) {  
    // Update Angajat  
    SqlHelper.ExecuteNonQuery(connection,  
CommandType.StoredProcedure, "Angajat_Update", arParams);  
} else {  
    // Insert Angajat  
    arParams[0].Direction = ParameterDirection.Output;  
    SqlHelper.ExecuteNonQuery(connection,  
CommandType.StoredProcedure, "Angajat_Insert", arParams);  
    // return new key  
    _Angajat.angajatId = (int)arParams[0].Value;  
}
```

Se observa ca se va chema una dintre procedurile `Angajat_Insert` si `Angajat_Update`, iar noua cheie este imediat disponibila in program, fiind automat salvata in obiect prin:

```
_Angajat.angajatId = (int)arParams[0].Value;
```

Stergerea

Cea mai simpla metoda de a sterge un obiect este de a apela direct metoda `Delete()`, pentru ca aceasta incapsuleaza apelul simplu din nivelul de acces la date.

```
// stergerea direct din obiect  
a.Delete();
```

Ca si mai sus, acest apel se traduce prin:

```
Taxi.DataAccess.AngajatDB.Delete(this);
```

Pentru stergerea obiectelor din acest nivel avem la dispozitie doua modalitati: prin trimiterea ca paramentru a obiectului propriuzis, sau prin trimiterea cheii primare. A doua metoda este mai rapida pentru situatiile in care avem o lista de id-uri si nu vrem sa mai instantiem fiecare obiect.

```
// Deletes 1 row by primary key
```

```
public static void Delete(int angajatId) {
    //returns a new opened connection
    SqlConnection connection = Connection;
    try {
        SqlHelper.ExecuteNonQuery(connection, "Angajat_Delete",
angajatId);
    }
    catch(Exception ex) { throw ex; }
    finally { connection.Dispose(); }
}

// Deletes 1 row received as object
public static void Delete(Angajat_Base _Angajat) {
    // use Delete by primary key
    Delete(_Angajat.angajatId);
}
```

Am prezentat metodele intregi pentru ca sunt suficient de scurte cat sa permita acest lucru. Ar mai fi de mentionat ca generatoarele de cod detecteaza automat tabelele cu mai multe coloane in cheia primara si le includ pe toate in lista de parametri a functiilor. Merita prezentata si procedura stocata in intregimea ei, pentru a intelege traseul complet:

```
CREATE PROCEDURE [dbo].[Angajat_Delete]
(
    @angajatId int
)
AS
SET NOCOUNT ON

DELETE
FROM [Angajat]
WHERE
    [angajatId] = @angajatId

RETURN @@Error
GO
```

4.2.4 Cautarea informatiilor

Odata salvate in baza de date, trebuie sa avem posibilitatea de a gasi obiectele, operatie necesara atat la afisarea de informatii cat si la procesarea de calcule avansate. Toate metodele de mai jos sunt implementate sub forma de metode statice in clasa de acces la date a fiecarei entitati. Acestea se comporta ca niste fabrici de obiecte, interogand baza de date si intorcand aplicatiei entitati sau colectii de entitati.

Arhitectura pune la dispozitia programatorilor mai multe metode de a interoga baza de

date, oferind un oarece echilibru intre operatii generale generate automat si posibilitatea de a realiza cautari particularizate. Dificultatea in orice proces automatizat de genul unei arhitecturi generata in intregime este ca structura de cod rezultata este de obicei destul de rigida. Asta inseamna ca nu sunt prea multe cai de a realiza lucrurile in afara de cele generate automat. Aceasta arhitectura beneficiaza de trei posibilitati de a realiza cautari *custom*:

- printr-o cautare dupa *conditie*: se trimite un string cu codul clauzei `WHERE` scris de programator;
- scrierea unei intregi proceduri stocate pentru cautare *custom*;
- folosirea de `VIEWS` pentru a agrega date din mai multe surse.

Toate cautarile de mai jos pot fi realizate si pe `VIEWS` (interpretate de aplicatie ca tabele *read-only*).

Cautare dupa cheie primara

Cautarea dupa cheia primara este singura cautare implementata diferit de celelalte metode — asta pentru ca este singura care intoarce un **obiect**, restul intorcand **colectii** de obiecte.

De asemenea, cautarea dupa cheia primara este singura care are un corespondent in nivelul de entitati business, sub forma unui constructor care primeste ca parametru valoarea cheii si instantiaza direct un obiect incarcat din baza de date. In caz ca nu exista, intoarce un obiect nou:

```
// instantiere obiect salvat anterior (angajatId = 2)
Angajat a = new Angajat(2);

// echivalent cu
Angajat a = AngajatDB.Get(2);
```

In toate metodele de cautare obiecte, logica este aceeaasi, prezentata mai jos:

```
reader = SqlHelper.ExecuteReader(connection, "Angajat_SelectByPK",
angajatId);
Angajat _Angajat = new Angajat();
_Angajat.angajatId = angajatId;

if(reader.Read()) {
    if (!reader.IsDBNull(1)) _Angajat.num = reader.GetString(1);
```

4.2 Implementarea arhitecturii de persistenta

```
if (!reader.IsDBNull(2)) _Angajat.prenume = reader.GetString(2);
...
// mark object as not new
_Angajat.IsNew = false;

return _Angajat;
} else {
return new Angajat();
}
```

Singura diferenta intre cautarea dupa cheie primara si alte tipuri de cautare este ca in loc de `if`-ul folosit pentru a testa existenta acelei chei, se foloseste un `while` pentru a parcurge toate randurile intoarse si a construi o colectie.

La nivel SQL aceasta se traduce prin:

```
SELECT *
FROM [Angajat]
WHERE
    [angajatId] = @angajatId
```

Cautare folosind proceduri stocate custom

Ca implementare interna, toate metodele care intorc o colectie de elemente sunt realizate pe baza ultimei metode, *Cautare folosind proceduri stocate custom* — sunt trimisi parametri procedurilor stocate generate automat, la fel cum utilizatorul ar trimite parametri unei proceduri scrisa de el.

Aceasta abordare evita repetarea codului si il face mai compact si mai usor de inteles si intretinut. Din acest motiv vom incepe cu aceasta metoda, chiar daca nu este prima ca importanta, celelalte se bazeaza pe ea din motive de economie.

Pentru a chema o procedura stocata custom avem la dispozitie o metoda cu urmatoarea semnatura:

```
public static AngajatList Get(ArrayList param, ArrayList vals, string
procedureName);
```

Aceasta construiește din cele doua array-uri `param` si `vals` colectia de parametri si valori pentru procedura stocata si apoi va invoca

```
reader = SqlHelper.ExecuteReader(connection,
CommandType.StoredProcedure, procedureName, arParams);
```

pentru a realiza selectia. Vom vedea mai jos ca pentru proceduri care nu primesc parametri, ea poate fi invocata cu null, null.

Sa presupunem ca trebuie sa gasim toti soferii care conduc un numar minim de masini. Vom scrie de mana o procedura stocata:

```
CREATE PROCEDURE [dbo].[Sofer_Select_BusyDrivers]
(
    @minCars int
)
AS
SELECT
    [Sofer].*
FROM
    [Sofer]
INNER JOIN
    [SoferiMasina] ON ([Sofer].soferId = [SoferiMasina].soferId)
INNER JOIN
    [Masina] ON ([Masina].masinaId = [SoferiMasina].masinaId)
GROUP BY
    [Sofer].soferId
HAVING
    COUNT([Masina].masinaId) >= @minCars

RETURN @@Error
GO
```

Iar apelarea ei din aplicatie se face in felul urmatoar:

```
// constructia parametrilor
ArrayList param = new ArrayList();
ArrayList vals = new ArrayList();
param.Add("@minCars");
vals.Add(3); // min. trei masini

// apelarea procedurii stocate
SoferList sl = Get(param,vals, "Sofer_Select_BusyDrivers");

// operatii pe soferi
foreach (Sofer s in sl) {
    ...
}
```

Toate randurile

Cea mai simpla functie de cautare este probabil cea care intoarce toate randurile unei tabele si construiește o colectie de obiecte cu acestea:

```
// toti angajatii
AngajatList al = AngajatDB.Get();
```

```
// operatie cu toti angajatii
foreach (Angajat a in al) {
    // cresterea salariului cu 10%
    a.salariu *= 1.1;
}
```

Codul metodei este:

```
// Gets all rows
public static AngajatList Get()
{
    return Get(null, null, "Angajat_SelectAll");
}
```

Iar la nivel SQL, procedura stocata `Angajat_SelectAll` invoca un simplu:

```
SELECT * FROM [Angajat]
```

Cautare dupa chei straine

In cazul tabelor care reprezinta capatul *copil* sau *strain* al unei relatii, este utila selectia tuturor elementelor care au o anumita valoare pentru o cheie straina (indiferent daca relatia este *one-to-many* sau *many-to-many*).

De exemplu, pentru tabela `Sofer`, este util sa gasim toti soferii care lucreaza sub un anumit `Plan` sau toti soferii care utilizeaza o anumita `Masina`. Pentru a putea realiza acest tip de query-uri, generatoarele de cod detecteaza toate cheile straine si construiesc la nivel SQL cate o procedura stocata pentru fiecare cheie straina detectata.

In cazul tablei `Sofer` s-au generat: `Sofer_SelectByFK_angajatId`, `Sofer_SelectByFK_masinaId` si `Sofer_SelectByFK_tipPlanId` pentru cheile straine `angajatId`, `masinaId`, respectiv `tipPlanId`. Sa vedem cum e realizata una dintre ele, pornind de la o relatie *many-to-one*:

```
-- Sofer_SelectByFK_angajatId
SELECT *
FROM    [Sofer]
WHERE   angajatId= @angajatId
```

O relatie *many-to-many* are nevoie sa parcurga tabela de legatura prin doua JOIN-uri consecutive, in felul urmator:

4.2 Implementarea arhitecturii de persistenta

```
-- Sofer_SelectByFK_masinaId
SELECT
    [Sofer].*
FROM
    [Masina]
INNER JOIN
    [SoferiMasina] ON (
        [Masina].masinaId = @masinaId AND
        [Masina].masinaId = [SoferiMasina].masinaId
    )
INNER JOIN
    [Sofer] ON (
        [Sofer].soferId = [SoferiMasina].soferId
    )
```

Chiar daca difera complexitatea implementarii, din aplicatie ele sunt chemate identic:

```
// soferii cu planul 2
SoferList sl = SoferDB.GetByFK("tipPlanId", 2);

// soferii care lureaza pe masina 3
SoferList sl = SoferDB.GetByFK("masinaId", 3);
```

Implementarea metodei este foarte asemanatoare cu celelalte:

```
// Gets rows by foreign key for Angajat table
public static SoferList GetByFK(string _SoferFKName, int
_SoferFKValue)
{
    ArrayList param = new ArrayList();
    ArrayList vals = new ArrayList();
    param.Add("@ " + _SoferFKName);
    vals.Add(NullableTypes.HelperFunctions.DBNullConvert.From(_SoferFKV
alue));
    return Get(param,vals, "Sofer_SelectByFK_" + _SoferFKName);
}
```

Cautarile dupa chei straine sunt folosite mai ales de arhitectura de persistenta, pentru a implementa relatiile, asa cum vom vedea in sectiunea urmatoare, *Implementarea relatiilor*.

Cautare folosind clauza WHERE

O cautare ceva mai usoara se face atunci cand conditia este simpla, iar selectul include doar tabela vizata. In acest caz se trimite direct string-ul clauzei WHERE, in felul urmatoar:

```
// toti angajatii care locuiesc pe strada Timisoara
AngajatList a = AngajatDB.Get("adresa like '%Timisoara%'");
```

Iata implementarea acesteia:

```
// Get rows by WHERE condition
public static AngajatList Get(string condition)
{
    ArrayList param = new ArrayList();
    ArrayList vals = new ArrayList();
    param.Add("@condition");
    vals.Add(condition);
    return Get(param,vals, "Angajat_Select_byCondition");
}
```

Codul SQL necesar este un pic mai interesant decat ce am folosit pana acum, deoarece procedura stocata va construi **dinamic** cod SQL (prin concatenarea conditiei la select), pe care-l va executa apoi cu ajutorul `sp_executesql`. Acest lucru este necesar deoarece parametrul contine de fapt cod SQL de executat.

```
CREATE PROCEDURE [dbo].[Angajat_Select_byCondition]
(
    @condition varchar(1000)
)
AS
DECLARE @strSQL nvarchar(4000)

SET @strSQL = '
SELECT *
FROM [dbo].[Angajat]
WHERE ' + @condition

EXEC sp_executesql @strSQL

RETURN @@Error
GO
```

Cautare folosind filtrarea implicita

Dintre toate metodele de cautare specificate pana acum, aceasta metoda este cea mai avansata. Ea poate fi folosita pentru a cauta intr-o tabela sau intr-un VIEW, avand posibilitatea de a adauga filtre pe toate coloanele. Pentru coloanele de tip string se foloseste o potrivire de tip LIKE `'%string_cautat%'`, iar pentru coloanele numerice si de tip data se poate cauta intr-un interval, adaugandu-se doua filtre, cate unul pentru fiecare capat.

Pentru implementarea acestui mecanism se foloseste o procedura stocata care construiesc **dinamic** o alta procedura (ca la cautarea dupa conditie), pe care apoi o

apeleaza cu setul initial de parametri. Datorita modului in care se construiesc un query SQL de cautare pe mai multe coloane, procedura exclude punerea de conditii din codul *dinamic* pe acele coloane pe care nu se cauta.

In caz contrar, daca s-ar apela direct o inlantuire de WHERE cond OR ISNULL(. . .) pentru toate coloanele, performantele ar fi foarte mici. Pentru cautari comune, care pun conditiile pe aceleasi coloane (Ex.: nume + prenume, data nasterii + sex, etc.), planul de executie al procedurii construite dinamic este salvat in cache si refolosit ulterior.

Iata procedura standard de filtrare pentru tabela `Sofer` (aleasa pentru ca are mai putine coloane):

```
CREATE PROCEDURE [dbo].[TipPlan_SelectAll_Filter]
(
    @tipPlanId int = NULL,
    @numePlan varchar(20) = NULL,
    @soferiPeMasina1 tinyint = NULL,
    @soferiPeMasina2 tinyint = NULL,
    @valoarePlan decimal(10,2) = NULL,
    @escape char
)
AS
DECLARE @strSQL nvarchar(4000),
        @paramlist nvarchar(4000)
SET @strSQL = '
    SELECT *
    FROM [dbo].[TipPlan]
    WHERE 1=1'
IF @tipPlanId IS NOT NULL
    SET @strSQL = @strSQL + ' AND TipPlan.tipPlanId = @xtipPlanId'
IF @numePlan IS NOT NULL
    SET @strSQL = @strSQL + ' AND TipPlan.numePlan like + ''%' +
    @xnumePlan + ''%' ESCAPE @xescape'
IF @soferiPeMasina1 IS NOT NULL AND @soferiPeMasina2 IS NOT NULL
    SET @strSQL = @strSQL + ' AND TipPlan.soferiPeMasina between
    @xsoferiPeMasina1 and @xsoferiPeMasina2'
IF @soferiPeMasina1 IS NOT NULL AND @soferiPeMasina2 IS NULL
    SET @strSQL = @strSQL + ' AND TipPlan.soferiPeMasina =
    @xsoferiPeMasina1'
IF @valoarePlan IS NOT NULL
    SET @strSQL = @strSQL + ' AND TipPlan.valoarePlan = @xvaloarePlan'
SET @paramlist = ' @xtipPlanId int,
    @xnumePlan varchar(20),
    @xsoferiPeMasina1 tinyint,
    @xsoferiPeMasina2 tinyint,
    @xvaloarePlan decimal(10,2),
    @xescape char '
```

4.2 Implementarea arhitecturii de persistenta

```
EXEC sp_executesql @strSQL, @paramlist, @tipPlanId, @numePlan,
@soferiPeMasina1, @soferiPeMasina2, @valoarePlan, @escape
    RETURN @@Error

GO
```

Ca sa cautam toate planurile cu 2–3 soferi si string-ul “seara” in denumire procedam astfel:

```
// construim lista de parametri
ArrayList param = new ArrayList();
ArrayList vals = new ArrayList();
param.Add("@numePlan");
vals.Add("seara");
param.Add("@soferiPeMasina1");
vals.Add(2);
param.Add("@soferiPeMasina2");
vals.Add(3);

// daca nu trimitem numele procedurii
// este apelata 'TipPlan_SelectAll_Filter'
TipPlanList tpl = TipPlanDB.Get(param,vals);
```

Codul SQL executat si cu planul *cache*-uit in aceasta combinatie de parametri va fi:

```
SELECT *
FROM [dbo].[TipPlan]
WHERE 1=1
AND TipPlan.numePlan LIKE '%seara%'
AND TipPlan.soferiPeMasina BETWEEN 2 AND 3
```

Implementarea este la fel de simpla:

```
// Gets filtered rows from table by calling SelectAll_Filter stored
proc
public static TipPlanList Get(ArrayList param, ArrayList vals)
{
    return Get(param, vals, "TipPlan_SelectAll_Filter");
}
```

Aceasta metoda de cautare poate parea anevoioasa la prima vedere, insa adevarata ei putere este ca poate fi folosita foarte usor din interfete de cautare avansata, unde se poate cauta usor pe o tabela sau pe un VIEW, pentru ca parametri si valorile se pot extrage automat tinand cont de casutele completate din formular.

4.2.5 Reprezentarea relatiilor

In orice arhitectura de persistenta un element important este reprezentarea relatiilor

dintre obiecte. Acestea sunt detectate cu ajutorul cheilor straine prezente in fiecare tabela — acestea sunt parcurse iar generatoarele de cod completeaza fiecare obiect prin atasarea de colectii de obiecte legate.

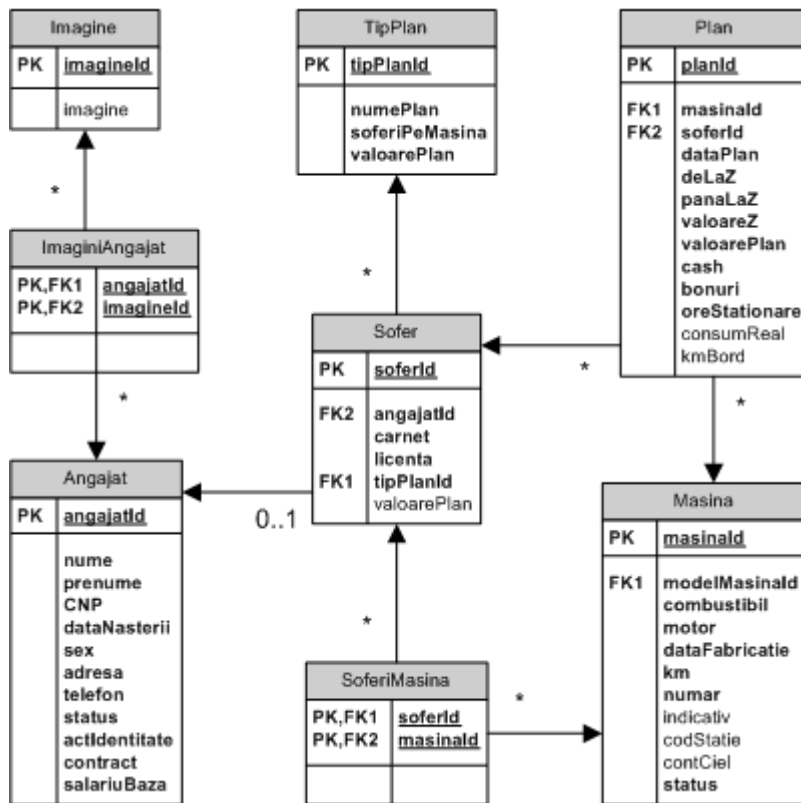
Se pot detecta automat relatiile **1:1** (*one-to-one*), **1:N** (*one-to-many*) si **N:M** (*many-to-many*). Pentru fiecare relatie, se adauga in tabela *parinte* un obiect sau o colectie de obiecte care extrage automat elementele din tabela *straina*, legate prin cheia straina cu *id*-ul obiectului parinte.

Relatiile pot fi parcurse in ambele sensuri, astfel incat in tabelele *copil* se adauga un obiect *parinte*, care se cere automat pe baza *id*-ului cheii straine.

Gestionarea relatiilor la nivelul arhitecturii s-a facut intr-un mod simplificat, deoarece este orientata pe *gasirea* obiectelor legate, nu si pe *persistarea* completa a tuturor tipurilor de relatii. Aceasta cerinta este mai greu de implementat pentru ca implica manipularea cu mare grija a listelor de obiecte legate la runtime, iar pentru a salva obiectele adaugate sau sterse dintr-o colectie este nevoie de gestionarea accesurilor concurente si a locking-ului, ceea ce dupa cum am mentionat in capitolul anterior nu am implementat in arhitectura de fata.

Sa revenim la exemplul prezentat mai anterior. In Figura 4.3 reluam diagrama entitate relatie, punand de data aceasta accent pe tipurile de relatii prezente:

4.2 Implementarea arhitecturii de persistenta



Pentru a putea fi reprezentate la nivelul relational, cateva dintre relatiile **N:M** din diagrama *UML* au fost sparte in cate doua relatii **1:N**, unite printr-o tabela de legatura:

- tabela *ImaginiAngajat* pentru relatia *Imagine* ↔ *Angajat*;
- tabela *SoferiMasina* pentru relatia *Sofer* ↔ *Masina*;
- tabela *Plan* pentru a reprezenta relatia cu attribute *Sofer* ↔ *Masina*.

Ultima este un caz special, deoarece la nivelul arhitecturii i-am adaugat o cheie primara proprie si este interpretata ca o entitate oarecare.

In afara de acestea mai avem o relatie **1:1** dintre *Sofer* si *Angajat* si relatiile **1:N** *TipPlan* ← *Sofer*, *Sofer* ← *Plan* si *Masina* ← *Plan*. Pentru moment relatiile 1:1 sunt interpretate ca si 1:N, deoarece ele sunt de obicei utile in mecanisme avansate de asociere sau compunere, discutate mai tarziu in acest capitol.

Relatii one-to-many

Relatiile one-to-many sunt cel mai usor de detectat si implementat, deoarece API-ul

generatorului de cod extrage automat toate cheile straine dintr-o tabela. Sa luam exemplul tablei `Sofer` — aceasta participa in 4 relatii, dintre care 3 de tip *one-to-many*:

- este **parinte** in relatie cu tabela `Plan` (prin cheia primara `soferId`);
- este **copil** in relatiile cu tabellele `Angajat` si `TipPlan` (cheile straine `angajatId` si `tipPlanId`).

In cadrul obiectului de tip `Sofer`, aceasta se va traduce prin adaugarea a unei colectii de tip `PlanList` (fata de care este parinte) si a cate unui obiect de tip `Angajat` si `TipPlan`:

```
...
// copii din relatii (1:N)
private PlanList _planList;
// parinti din relatii
private Angajat _angajat;
private TipPlan _tipplan;
...
```

Accesul la aceste campuri se face prin cateva proprietati publice, care ne permit sa implementam un mecanism de incarcare la cerere (*load-on-request*) cunoscut sub numele de *lazy loading*.

Pentru implementare se foloseste sablonul (*pattern*) Singleton, care verifica daca a fost deja instantiata colectia, in caz contrar executand cererea catre baza de date. De remarcat operatia *set* a obiectelor parinte: id-ul cheii straine din obiect este actualizat automat, pentru a se persista corect.

```
// lista copil
public PlanList planList {
    get {
        if (_planList == null) {
            _planList = Taxi.DataAccess.PlanDB.GetByFK("soferId",
this._soferId );
        }
        return _planList;
    }
    set { _planList = value; }
}
// obiecte parinte
public Angajat angajat {
    get {
        if (_angajat == null) {
```

4.2 Implementarea arhitecturii de persistenta

```
        _angajat = Taxi.DataAccess.AngajatDB.Get(this._angajatId );
    }
    return _angajat;
}
set { _angajat = value; this._angajatId = _angajat.angajatId; }
}
public TipPlan tipplan {
    get {
        if (_tipplan == null) {
            _tipplan = Taxi.DataAccess.TipPlanDB.Get(this._tipPlanId );
        }
        return _tipplan;
    }
    set { _tipplan = value; this._tipPlanId = _tipplan.tipPlanId; }
}
```

Acest intreg mecanism inseamna de fapt traducerea reprezentarii relationale a schemei informationale intr-o reprezentare orientata pe obiecte, cu avantajul ca se face automat si, mai ales, **doar la cerere**.

Sa presupunem ca dorim sa cerem toate planurile facute de un anumit sofer si numele sau, iar apoi sa-i schimbam planul curent. Folosindu-ne de modelul obiectual, este suficient sa scriem (Print e o metoda fictiva):

```
Sofer s = new Sofer(23);

// toate planurile
foreach (Plan p in s.planList) {
    // extragem numarul masinii conduse
    // (inca o relatie traversata!)
    Print(p.masina.numar);
}

// numele soferului
Print(s.angajat.numa);

// actualizare tip plan
s.tipPlan = TipPlanDB.Get(34);
s.Save();
```

Fara gestionarea automata a relatiilor, acest cod ar fi trebuit scris asa, fiind dupa cum se observa mult mai greu de urmarit:

```
Sofer s = SoferDB.Get(23);

// toate planurile
PlanList pl = PlanDB.GetByFK("soferId", s.soferId);

foreach (Plan p in pl) {
    // extragem numarul masinii conduse
```

```
Masina m = MasinaDB.Get(p.masinaId);
Print(m.numar);
}

// numele soferului
Angajat a = AngajatDB.Get(s.angajatId);
Print(a.numa);

// actualizare plan
TipPlan p = TipPlanDB.Get(34);
s.tipPlanId = p.tipPlanId;
s.Save();
```

Dupa cum se vede, implementarea foloseste intens metodele de cautare dupa chei straine, mentionate in sectiunea anterioara. Din acest motiv nu mai prezentam codul SQL folosit.

Relatii many-to-many

Dupa cum am mai spus mai sus, dificultatea principala in reprezentarea relatiilor *many-to-many* in modelul orientat pe obiecte este ca in modelul relational apare o tabela de legatura in plus. Greul este insa transferat generatoarelor de cod care sunt capabile de a detecta o tabela de legatura (are exact doua coloane care sunt in acelasi timp si chei primare si chei straine) si de a folosi procedurile stocate corespunzatoare din SQL. Tabelele de legatura sunt “sarite” prin doua JOIN-uri consecutive, iar procedura se comporta exact ca una pentru relatii *one-to-many*.

In capitolele anterioare am facut recomandarea de a nu folosi explicit obiecte asociate tabelor de legatura — nici macar nu sunt generate clase pentru aceste tabele. In schimb, pentru **cautarea** elementelor legate sunt generate colectii ca si la relatiile *one-to-many*, iar pentru **persistarea** lor metode de tip Add si Remove in/din colectiile respective.

Sa luam exemplul relatiei *Imagine ↔ Angajat*. Pentru persistarea ei s-a construit tabela *ImaginiAngajat*, care nu este reprezentata explicit in arhitectura de persistenta. Pentru ea sunt generate doar 4 proceduri stocate la nivel SQL:

- *ImaginiAngajat_Insert*
- *ImaginiAngajat_Delete*
- *Imagine_SelectByFK_angajatId*

- Angajat_SelectByFK_imageId.

Acestea sunt toate operatiile necesare pentru gasirea si persistarea obiectelor legate de aceasta relatie. La nivelul arhitecturii de persistenta, persistarea este implementata prin adaugarea de operatii Add si Remove fiecarui tip de obiect. Tradus in cuvinte, aceste operatii suna in felul urmator:

- adauga Imagine la Angajat
- sterge Imagine de la Angajat
- adauga Angajat la Imagine
- sterge Angajat de la Imagine.

Implementarea din clasele de persistenta arata asa (pentru ambele obiecte e identica):

```
// la obiectele de tip Angajat
...
public void AddImagine(Image _Image) {
    Taxi.DataAccess.AngajatDB.AddImagineToAngajat(this, _Image);
}

public void RemoveImagine(Image _Image) {
    Taxi.DataAccess.AngajatDB.RemoveImagineFromAngajat(this, _Image);
}
...
```

In clasele de acces la date aceasta se traduce prin apelarea procedurilor stocate de inserare si stergere in tabela ImaginiAngajat:

```
public static void AddImagineToAngajat(Angajat_Base _Angajat,
Image_Base _Image) {
    SqlConnection connection = Connection;
    try
    {
        SqlHelper.ExecuteNonQuery(connection, "ImaginiAngajat_Insert",
_Angajat.angajatId, _Image.imageId);
    }
    catch(Exception ex) { throw ex; }
    finally { connection.Dispose(); }
}

public static void RemoveImagineFromAngajat(Angajat_Base _Angajat,
Image_Base _Image) {
    SqlConnection connection = Connection;
    try
    {
        SqlHelper.ExecuteNonQuery(connection, "ImaginiAngajat_Delete",
_Angajat.angajatId, _Image.imageId);
    }
}
```

```
    }  
    catch(Exception ex) { throw ex; }  
    finally { connection.Dispose(); }  
}
```

Cautarea obiectelor se implementeaza exact la fel ca la relatiile de tip *many-to-many* (prin adaugarea colectiilor care se incarca prin *lazy-loading*), singura diferenta fiind procedura stocata mai complexa, cu cele doua JOIN-uri succesive:

```
...  
private ImagineList _imagineList;  
...  
public ImagineList imagineList {  
    get {  
        if (_imagineList == null) {  
            _imagineList = Taxi.DataAccess.ImagineDB.GetByFK("angajatId",  
this._angajatId );  
        }  
        return _imagineList;  
    }  
    set { _imagineList = value; }  
}  
...  
}
```

Pentru exemplificare, sa presupunem ca vrem sa mutam toate imaginile unui angajat la un alt angajat:

```
// mutam toate imaginile de la 1 la 2  
Angajat a = new Angajat(1);  
Angajat b = new Angajat(2);  
  
foreach (Imagine i in a.imagineList) {  
    b.AddImagine(i);  
    a.RemoveImagine(i);  
}  
  
// verificare mutare  
Print(b.imagineList.Count);
```

Capitolul 5 Aplicatie demonstrativa

In capitolul anterior, *Implementare si utilizare*, am vazut cum se implementeaza complet o arhitectura de persistenta. Avand o arhitectura functionala, vom realiza o simpla aplicatie demonstrativa, care sa puna in valoare toate operatiile prezentate anterior. Baza de date aleasa este astfel proiectata incat sa acopere toate cazurile posibile pe care arhitectura le poate detecta si interpreta.

Aceasta va fi construita peste schema relationala utilizata si pentru exemplele din capitolul anterior, schema care modeleaza activitatea unei companii de taximetre. In realizarea aplicatiei nu vom insista pe implementarea completa a gestiunii companiei, ci mai degraba vom extrage exemple didactice, alcatuite din operatii care ar putea aparea in cadrul unei astfel de companii.

5.1 Structura proiectului

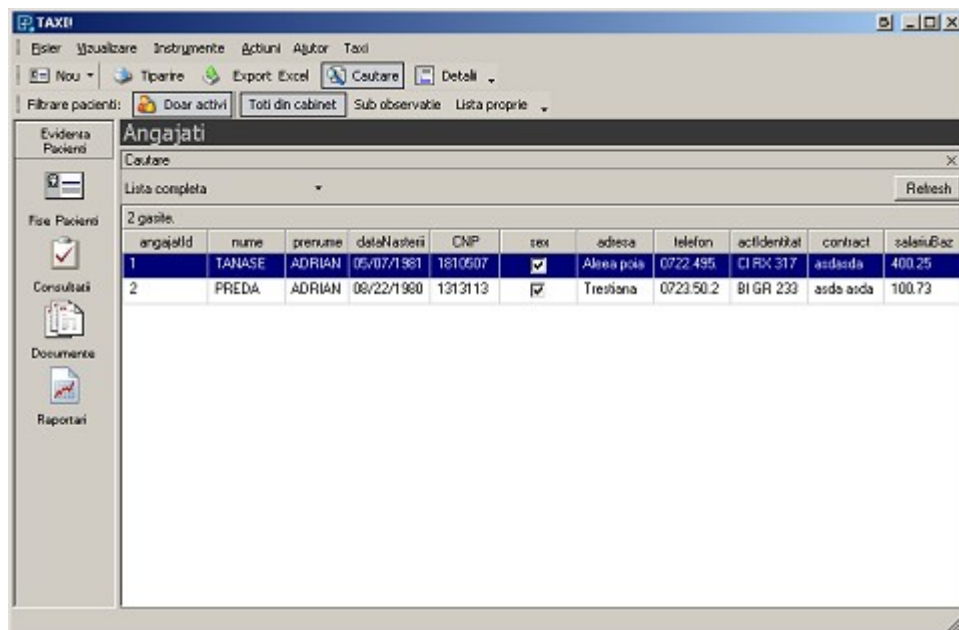
Aplicatia Taxi este structurata cu ajutorul unei solutii Visual Studio, alcatuita din trei proiecte distincte:

- proiectul `Taxi.BusinessObjects` care contine toate clasele generate ale arhitecturii si se compileaza intr-un dll (este deci o librerie); initial clasele de persistenta si clasele de acces de date au fost impartite in doua librarii diferite, pentru o intretinere mai usoara, insa din cauza modului in care un nivel incapsuleaza apeluri catre celalalt au aparut niste referinte circulare — acestea nu sunt suportate de .NET decat daca se intampla in cadrul aceleiasi *assembly* (librarie).
- proiectul `Taxi.BusinessLogic` (este tot o librerie) — contine procese logice care conduc strangerea de date si procesarea lor; aceste procese comanda interfata, pentru a separa logica de prezentarea informatiilor.

- proiectul `Taxi`, aplicatia propriu-zisa — interfata aplicatiei, alcatuita in mare parte din componente refolosibile.

5.2 Componentele aplicatiei

Interfata aplicatiei (prezentata in Figura 5.1) este o interfata clasica, avand in centrul atentiei lista de angajati, un meniu din care se aleg diversele comenzi, o bara de unelte pentru operatii uzuale si filtrari rapide si o bara laterala, pentru a selecta componentele principale ale aplicatiei.



Dintre cele mai interesante componente refolosibile, dezvoltate pentru o interfata mai bogata putem mentiona:

- un form de baza, care contine o lista de elemente `BO_IObject` si filtre default in partea superioara — aici se vad avantajele implementarii unei interfete comune de catre toate clasele de persistenta: lista de baza este extinsa doar prin specificarea coloanelor care trebuie sa apara si filtrele pe care se caute — mai departe nu conteaza, pentru ca toate clasele implementeaza operatii de tip `Get` prin care se filtreaza rezultatele.
- o fereastră de dialog (clasa de baza) pentru editarea obiectelor — aceasta are deja meniuri pentru navigarea prin lista, crearea unui obiect nou, salvare si

stergere — toate fara a sti tipul obiectului manipulat (aici sunt folosite metodele Save, Delete, Reload declarate in interfata).

Ca peste tot in arhitectura, am pus accentul pe refolosirea codului (pentru a intretine o singura varianta de cod) si pe utilizarea din plin a elementelor avansate de proiectare orientata pe obiecte, cum ar fi polimorfismul.

Capitolul 6 Concluzii

Lucrarea de fata a urmarit proiectarea si implementarea completa a unei arhitecturi de persistenta pentru platforma .NET. Pe parcurs am studiat ce implica proiectarea unei aplicatii software distribuite, care sunt componentele acesteia si care sunt cele mai bune practici pentru implementarea lor.

In capitolul 2, *Proiectarea arhitecturilor software*, am prezentat arhitectura si componentele unei aplicatii distribuite, incheind cu un set de recomandari utile pentru proiectarea unei aplicatii oricat de complexe. Capitolul 3, *Proiectarea arhitecturii de persistenta*, a urmarit indeaproape toate deciziile pe care un arhitect software trebuie sa le ia atunci cand isi propune sa proiecteze o astfel de arhitectura, dupa care am vazut ce implica proiectarea unui nivel de acces la date, precum si cum se reprezinta informatiile si cum se trimit acestea intre nivelurile aplicatiei.

Cel mai consistent capitol, *Implementare si utilizare*, a prezentat detaliat API-ul arhitecturii de persistenta, incluzand numeroase exemple de cod pentru toate operatiile implementate si cum sunt implementate toate aceste operatii. Am putut astfel vedea cata complexitate si putere se ascunde sub cele cateva apeluri de cautare si persistare a obiectelor, precum si cat de usor face aceasta arhitectura scrierea de cod, inlaturand procesul anevoios si repetitiv utilizat in lipsa ei.

Folosirea unei arhitecturi de persistenta elibereaza programatorul de scrierea intregului cod care se ocupa cu gasirea si persistarea datelor, permitandu-i acestuia sa se concentreze mult mai mult pe logica aplicatiei. Consideram esentiala folosirea ei in dezvoltarea aplicatiilor din ce in ce mai complexe din ziua de azi.

Rezultatul lucrarii consta intr-o arhitectura functionala, care implementeaza toate operatiile prezentate si deja si-a dovedit eficienta in mai multe aplicatii la care a fost folosita.

6.1 Viitoare imbunatatiri

Cu toate acestea, munca nu se va incheia aici, deoarece arhitectura de persistenta poate fi imbunatatita in mai multe moduri. Pe de o parte, raman neimplementate mecanismele avansate prezentate capitolul 3:

- lucrul cu seturi mari de rezultate;
- implementarea unui model tranzactional direct in arhitectura;
- mecanisme de locking si concurrency;
- nivel intermediar de caching.

In afara de acestea, mai pot fi aduse imbunatatiri unor operatii deja implementate, pentru cresterea performantei sau pentru modelarea unor scheme informationale mai complexe. Imbunatatiri posibile operatiilor de cautare a elementelor:

- cautarea folosind obiecte partial umplute (prin construirea automata a parametrilor pentru procedura `_SelectAll_Filter`);
- adugarea de JOIN-uri pentru a selecta obiectele legate prin chei straine dintr-un singur query, odata cu colectia principala;
- introducerea conceptului de *criteriu de cautare*: acesta ar permite adaugarea de conditii (clauze WHERE) si altor operatii, cum ar fi UPDATE sau DELETE, nu doar SELECT-urilor.

O alta imbunatatire a arhitecturii ar fi posibilitatea modelarii unor concepte avansate din programarea orientata pe obiecte, in afara de simplele asocieri pe baza cheilor straine:

- mostenirea simpla sau multipla (**IS A**);
- compunerea (**COMPOSED OF**) — la distrugerea obiectului parinte dispar si obiectele copil (ex: componentele unei masini);
- agregarea sau asocierea (**HAS**) – la distrugerea obiectului parinte, copii continua sa existe (ex: la stergerea unui sofer raman in evidenta contabila planurie zilnice realizate de acesta).

Anexe