PRINCIPII DE DEZVOLTARE A INSTRUMENTELOR DE PROIECTARE A BAZELOR DE DATE

Vitalie Cotelea

Academia de Studii Economice din Moldova, Chişinău

vitalie.cotelea@gmail.com

Rezumat: În acest articol se tratează probleme legate de principiile de organizare a produselor program elaborate pentru automatizarea procesului de proiectare a bazelor de date. Sunt expuse tendinţele utilizate azi în dezvoltarea sistemelor informatice cu scopul de a separa componenta logică de cea grafică în crearea interfeţelor. Se caracterizează beneficiile paradigmei funcţionale de programare şi aspectele ce ţin de principiile de elaborare a codurilor-sursă în F#, precum şi se argumentează aplicarea paradigmei logice de programare pentru anumite tipuri de probleme ce apar în procesul proiectării bazelor de date. Se propune o soluţie de elaborare a sistemelor de automatizare a procesului de proiectare a bazelor de date. Soluţia se bazează pe integrarea mai multor paradigme de programare, pe conceptul MVC de dezvoltare a aplicaţiilor şi tehnologia WPF utilizată pentru elaborarea interfeţelor cu un design destul de atractiv.

Cuvinte cheie: proiectare baze de date, paradigme de programare, arhitectura MVC, tehnologia WPF.

Abstract: The paper deals with problems related to principles of organizing software products elaborated to automate the database design process. There are exposed current trends in development of information systems, which aim to separate the logical and graphical components in the interfaces construction. Benefits of functional programming paradigm and aspects of the design principles of F# source code are characterized. Also, it is argued the application of logic programming paradigm for certain types of problems that appear in the database design. And finally, a solution for developing automation systems for database design is proposed, which is based on the integration of several programming paradigms, on the MVC concept of application development and on WPF technology used to develop interfaces with a quite attractive design.

Keywords: database design, programming paradigms, MVC architecture, WPF technology.

1. Introducere

Instrumentele de proiectare a bazelor de date elaborate până acum s-au concentrat, în principal, asupra normalizării. Lipsesc, în totalitate, sistemele care ar face o analiză a bazelor de date deja proiectate. Acest lucru împiedică dezvoltarea sistemelor informatice existente, şi adaptarea acestora la cerinţele noi ale zilei de azi. În afară de aceasta, instrumentele au o destinaţie mai mult didactică, pentru instruire. Algoritmii utilizaţi în acestea sunt cei clasici (de complexitate înaltă), care sunt aplicabili doar asupra unor exemple de laborator şi nu pot servi pentru proiectarea bazelor de date reale.

Tendinţele, care sunt, la moment, în dezvoltarea sistemelor informatice, demonstrează separarea foarte evidentă a componentei logice şi a componentei grafice, precum şi a componentei de realizare a algoritmilor de componenta logică. Acestea pot fi dezvoltate separat, de echipe separate, în limbaje diferite. Limbajele utilizate, din motive de integrare, erau din aceeaşi paradigmă, de regulă, cea procedurală. Astăzi, când sistemele elaborate sunt complexe şi utilizatorul trebuie să fie îndepărtat de la procese, el trebuie să cunoască doar ce doreşte de la sistem, se schimbă şi principiile de elaborare ale acestora [5]. Acum, de când Microsoft oferă pe platforma .NET un limbaj funcţional, F#, platformă de care sunt legate şi Sistemele de Gestiune a Bazelor de Date SQL Server [2] produse de această companie, integrarea diferitelor paradigme de programare a devenit realizabilă.

Trebuie menţionat că există tendinţa de apelare la diverse limbaje pe o singură platformă şi a altor companii producătoare de SGBD-uri. După cum se ştie, firma *Sun Microsystems,* care a creat şi susţinea limbajul Java, a fost cumpărată de Oracle în 2009 [3], deci, în acest moment, Oracle este compania care dictează, într-un fel, viitorul limbajului (nu abuziv, deoarece limbajul este specificat prin JSR-uri, de exemplu, pe [11], într-un mod liber şi colaborativ, şi are o licenţă deschisă pentru cei care îl utilizează).

O paradigmă de programare permite specificarea unui model de rezolvare a unei probleme. Un model este o reprezentare simplificată sau o abstractizare a unui sistem.

Uneori, limbajul de programare în care va fi implementată soluţia, oferă aceleaşi construcţii

şi mecanisme ca şi paradigma. În acest caz, implementarea, verificarea soluției se fac mult mai simplu. Se poate considera atunci că o paradigmă de programare este o clasă de limbaje.

De multe ori, un limbaj de programare poate aparține mai multor paradigme. Astfel, C++ include caracteristici ale paradigmelor imperativă şi procedurală, ca predecesorul C, dar şi ale celei de orientare pe obiecte.

Paradigma logică [8] oferă posibilitatea scrierii programelor pentru probleme greu formalizabile, ce țin de apelarea la elemente inteligente de realizare. Paradigma funcțională [4] oferă un limbaj din perspectiva definirii mediului, şi formularea întrebărilor către sistem va genera răspunsuri în baza definițiilor, adică sistemul însuşi decide cum să dea răspuns la întrebare. Această paradigmă este adecvată în expunerea părții logice a sistemului (business logic) care, deseori, are o forma strictă. Paradigma procedurală, care presupune scrierea cum sistemul trebuie să proceseze, se aplică cu succes în stratul de integrare şi prezentare către utilizator, datorită gradului de dezvoltare a instrumentelor cu elemente vizuale (*drag and drop*), limbaje cu suport extins şi mult mai simple în învățare.

2. Utilizarea paradigmei de programare funcțională

O paradigmă de programare constituie un stil de a programa. Programarea funcțională este o paradigmă de programare, care tratează calculul ca evaluare de funcții matematice şi evită starea şi datele muabile. Se pune accent pe aplicarea de funcții, spre deosebire de programarea imperativă, care foloseşte, în principal, schimbările de stare.

Modelul matematic al programării funcționale îl reprezintă calculul lambda. Limbajele funcționale moderne pot fi considerate extensii ale calculului lambda. Noțiunea de bază, în această paradigmă, este cea de *funcțională* sau *funcție de nivel înalt*, o funcție care poate accepta ca argument sau returna ca valoare o altă funcție.

Se cuvine notat că avantajul programării funcționale este predictabilitatea mare a programului. Teoretic, cel puțin, un algoritm poate fi demonstrat că merge prin demonstrații matematice.

Unul din limbajele moderne de programare funcțională este F# [4]. Acesta este un limbaj derivat din Caml şi este conceput special pentru platforma .NET. El beneficiază de o integrare bună (contrar altor limbaje care au fost aduse pentru .NET). Aceasta permite, de asemenea, accederea la una din cele mai mari biblioteci, posibilitatea de a se utiliza cu ASP.NET, cu Silverlight, etc.

Este posibilă scrierea programelor, care pot fi compilate, la fel, de F# şi de Caml. Acesta este cazul pentru însuşi compilatorul F#. În ciuda acestei proximități, acestea sunt două limbaje bine distincte. Unele funcționalități ale lui Caml nu sunt acceptate de F#, şi invers, F# posedă numeroase adăugiri în comparație cu Caml.

F# este un limbaj adevărat funcțional (acest mod de programare este incitant) şi posedă funcționalitățile care se aşteaptă în această paradigmă. În întregime, este orientat pe obiecte (chiar şi întregii sunt obiecte şi pot avea metode) şi suportă programarea imperativă. El permite crearea structurilor sale de date în mod foarte concis şi puternic. Posedă o tipizare statică puternică şi susține inferarea tipurilor. Acest fapt poate reduce considerabil depănarea şi permite elaborarea unui cod sigur.

F# este un limbaj foarte concis şi expresiv. Verbozitatea lui este comparabilă cu cea a limbajelor Python sau Ruby. El este simplu, poate utiliza tipuri dinamice, dacă este necesar, poate utiliza evaluarea *lazy* când trebuie, posedă un sistem de filtrare pe cazuri (recunoaşterea formelor – *pattern matching*), extensibil, permite manipularea fluxurilor (prin proprietăți caracteristice şi printr-un sistem de monade).

F# este un limbaj inovat care posedă numeroase funcționalități unice. El este rezultatul cercetărilor din cadrul Microsoft. El este încă tânăr, lipsit, uneori, de maturitate, dar care evoluează foarte rapid. În afară de aceasta, F# se caracterizează prin faptul că: evoluează foarte rapid, are calități generice, oferă proprietăți caracteristice pentru liste; asigură colectarea

automată a obiectelor moarte, corectitudinea codului, recuperarea din erori şi introspecţia; posedă o sintaxă simplă şi concisă, biblioteci şi un mediu de dezvoltare; susţine tipuri polimorfe, tipizarea dinamică, supraîncărcarea, precum şi programarea asincronă.

Cea mai importantă caracteristică, care a motivat utilizarea limbajului F# în elaborarea instrumentelor de proiectare a bazei de date, este determinată de susţinerea acestui limbaj al programării asincrone.

Programarea sincronă este puţin diferită de alte forme de programare paralelă. Modelul de programare asincronă în F#, şi aplicaţiile acestuia în programarea reactivă, paralelă şi concurentă este descris în [12]. Una din caracteristicile programării asincrone constă în faptul că o funcţie continuă să ruleze în paralel cu programul din care este apelată. Finalizarea apelului este realizată folosind o metodă de sincronizare: blocare, aşteptare activă sau pasivă, fie printr-o funcţie de callback, fapt ce permite F # să opereze lin şi să compileze fără probleme şi eficient. O versiune adaptată a acestei abordări a fost recent anunţată pentru o versiune viitoare a C #.

3. Aplicarea paradigmei de programare logică

Limbajele de programare logică, precum Prolog, utilizează clauze Horn pentru reprezentarea dependenţelor între obiecte. Motorul de inferenţă se bazează pe respingere rezolutivă, iar strategia de control este *backward chaining*, cu căutare în adâncime (de*pth-first*). Strategia permite concentrarea pe demonstrarea sau respingerea unui scop fixat. Ordinea de căutare (în adâncime) indică faptul că *Prolog* nu este un demonstrator complet de teoreme, deoarece poate urma o cale infinită din arborele de demonstrare [8].

Una din principalele idei ale programării logice constă în faptul că un algoritm e constituit din două elemente disjuncte: logică şi control. Componenta logică corespunde definiţiei problemei ce trebuie să fie soluţionată, în timp ce componenta control stabileşte cum soluţia poate fi obţinută. Un programator trebuie să descrie numai componenta logică a unui algoritm, lăsând controlul executării să fie exercitat de sistemul de programare logică utilizat. Cu alte cuvinte, sarcina programatorului este specificarea problemei ce trebuie soluţionată. Astfel,limbajul logic poate fi conceput simultan ca limbaj de descriere, specificare formală a problemei şi ca un limbaj de programare a calculatoarelor.

Paradigma fundamentală a programării logice este cea de *programare declarativă,* în opoziţie cu *programarea procedurală, imperativă* tipică limbajelor obişnuite. Punctul focal al programării logice este identificarea noţiunii de *calcul* şi noţiunii de *deducţie*. Mai exact, sistemele de programare logică reduc executarea unui program la căutarea prin respingere a clauzelor programului împreună cu negaţia propoziţiei ce exprimă întrebarea. Aici, se urmează regula: o respingere e o deducţie de la contrariu.

Astfel, cunoştinţele (programul şi/sau datele) se pot exprima în Prolog prin intermediul clauzelor de două tipuri: fapte şi reguli. Un fapt denotă un adevăr necondiţionat, în timp ce o regulă defineşte condiţiile care trebuie satisfăcute pentru ca declaraţia să fie considerată adevărată. Faptele şi regulile pot fi utilizate împreună şi nu e nevoie nici de o componentă deductivă adiţională. În plus, regulile recursive şi nedeterminismul permit programatorului să obţină descrieri foarte clare, concise şi neredundante de informaţie ce se doreşte reprezentată. Deoarece nu există distincţie între argumentele de intrare şi cele de ieşire, diverse combinaţii de argumente pot fi folosite.

Caracteristicile mai marcante ale sistemelor de programare logică, în general – şi ale limbajului Prolog, în particular – sunt următoarele: descrierile sunt programe, capacitatea deductivă, nedeterminismul, reversibilitatea relaţiilor (sau calculul bidirecţional), interpretări de programe logice şi recursia.

O serie de algoritmi ce ţin de proiectarea şi analiza bazelor de date, mai ales cei ce au necesitat reprezentarea soluţiilor în spaţiul de stări, prin satisfacerea unor mulţimi de constrângeri greu formalizabile [10], algoritmi genetici [6], reţele neuronale [9] şi alte tehnici

ale inteligenței artificiale, au fost elaborați în limbajul Turbo Prolog, apoi în varianta Windows, ce se numeşte Visual Prolog, acum versiunea 7.2. Dat fiind faptul că, în acest limbaj, lista este un termen structurat predefinit [7], foarte eficient se realizează algoritmii ce necesită prelucrări recursive ale structurilor de date.

Ca mediu complet de programare, Visual Prolog dispune de multe facilități printre care se remarcă portabilitatea, modularitatea, flexibilitatea, predefinirea funcțiilor, execuția rapidă a aplicațiilor, facilități de depanare a programelor, ample meniuri, grafică interactivă, editor cu posibilități de import-export programe, hipertexte, disponibilitate pe diverse platforme şi altele.

4. Integrarea paradigmelor de programare – o soluție de proiectare

În continuare, este prezentat modul de organizare a produselor program, de exemplu, în sistemul DB Designer, elaborat pentru asistarea studenților în procesul de proiectare a bazelor de date.

În proiectarea conceptuală a sistemului, au fost utilizate cele 13 tipuri de diagrame ale limbajului UML, divizate în trei categorii. Şase tipuri de diagrame reprezintă structura sistemului, iar şapte reprezintă tipuri generale de comportament, inclusiv patru care sunt grupate într-o categorie aparte, ele reprezentând diferite aspecte ale interacțiunilor [1].

Proiectarea sistemului DB Designer s-a bazat pe premisa modularității sistemului, extinderii şi menținerii. Astfel, s-a decis să fie aplicată următoarea îmbinare de limbaje: Prolog, F#, C# şi WPF (Windows Presentation Framework). Ultimele trei sunt produse Microsoft, integrate în mediul de dezvoltare Visual Studio 2010 şi menținute în cadrul platformei .NET. Îmbinarea acestor limbaje este strict limitată în ariile de întrebuințare:

- F# şi Prolog sunt utilizate pentru definirea regulilor funcționale ale sistemului;
- WPF este utilizat pentru partea de prezentare grafică a sistemului;
- C# e utilizat pentru integrarea părții logice cu partea grafică.

O astfel de integrare în conceptul modern de dezvoltare a aplicaților se numeşte modelul MVC (Model, View, Controller), unde în calitate de Model se consideră tot ce este scris în F# şi Prolog, partea Controller - tot ce este scris în C#, iar partea View – codul *xaml* cu utilizarea bibliotecilor WPF.

Windows Presentation Foundation este o tehnologie inovatoare ce poate fi folosită în dezvoltarea interfețelor vizuale ale aplicațiilor .NET 4.0. WPF beneficiază de un motor grafic extrem de puternic care are capabilități de randare atât a obiectelor 2D, cât şi 3D. Noul API integrează elemente din HTML, CSS, SVG precum şi din documente word sau aplicații concurente, de exemplu, Macromedia Flash. WPF mai aduce cu el şi un limbaj de programare declarativă, *eXtensible Application Markup Language* (XAML), care poate fi folosit pentru dezvoltarea rapidă a unei interfețe vizuale.

Cadrul WPF, prin intermediul **.xaml,* se foloseşte pentru definirea interfeței grafice a sistemului. Acesta permite separarea stratului (layer) logic de stratul grafic al sistemului, fapt ce oferă un grad de independență, adică fiecare strat poate fi modificat şi dezvoltat separat, şi integrate doar prin mapări între componenta grafică şi funcția logică.

Aranjarea pe ecran a diverselor componente ale unei aplicații, ulterior, poate fi complicată din cauza multitudinilor de posibilități de afişare, pe care le pot avea utilizatorii. WPF oferă un sistem de paginare extensibil pentru a aranja vizual elementele unei interfețe utilizator. Aceasta se poate redimensiona şi ajusta inteligent, în funcție de cum se defineşte layout-ul.

Grafica în WPF este vectorială, în contrast cu grafica raster. Grafica vectorială este, în mod inerent, scalabilă şi, de obicei, necesită mai puțină memorie decât o imagine raster comparabilă.

Astfel, WPF separă interfața de utilizator de comportamentul acestuia. Aspectul de design este specificat în XAML, iar comportamentul este definit într-un limbaj de programare ,precum F#, C # sau Visual Basic. Cele două părți sunt legate împreună de evenimente *databinding* şi comenzi. Separarea aspectului şi comportamentului implică următoarele beneficii:

- Aspectul şi comportamentul sunt slab cuplate.
- Designerii şi dezvoltatorii pot lucra pe modele separate.
- Instrumentele de proiectare grafică pot lucra pe documente simple XML, în loc de efectuarea unei analize de cod.

XAML este un limbaj declarativ permite separarea foarte uşoară a UI-lui de logica aplicației, acest lucru fiind foarte util la dezvoltarea aplicațiilor, practic, oferă programatorului şi designerului posibilitatea de a lucra simultan pe acelaşi control unul la partea de UI şi altul la partea de logică. Fiind un limbaj care are la bază XML, fiecare document XAML trebuie să respecte regulile de sintaxă XML.

Handlerele evenimentelor şi, de asemenea, în mod normal, atributele acestora pot fi definite în tagurile respective. Acelaşi lucru se putea face şi în C#.

Cu toate acestea, XAML-ul nu este un limbaj procedural, şi astfel, el nu "face" nimic referitor la logica aplicației, fiind doar un limbaj markup, ca HTML-ul. Partea de programare o poate face codul din fişierul de code-behind. Codul de business logică poate fi în orice limbaj, iar librăriile la care acesta apelează pot fi în cu totul alt limbaj.

Integrarea proiectelor F# şi WPF are loc grație faptului că atât proiectul F#, cât şi proiectul WPF sunt părți ale unei şi aceleiaşi soluții-program. Proiectul F# se livrează ca o bibliotecă *.dll, care este referit în proiectul WPF. Iar cu ajutorul C# se cheamă funcțiile F#, rezultatul cărora să poată fi transmis către WPF pentru afişare utilizatorului.

În proiectarea interfețelor cu utilizatorul, în funcție de aspectul principal luat în considerare în dezvoltarea acestora, există două direcții de proiectare foarte utilizate: proiectarea centrată pe sarcini şi proiectarea centrată pe utilizator.

În sistemul DB Designer, este utilizată a doua abordare, care se bazează pe înțelegerea domeniului de muncă al proiectantului şi a modului în care aceştia interacționează cu calculatorul. În acest sens, interfața menajează, ajută proiectantul, proiectarea bazându-se pe sarcinile reale ale acestuia.

Pentru proiectarea interfețelor a fost utilizată paradigma tehnologiei bazate pe modele, care constă în premisa că dezvoltarea interfeței poate fi asistată în totalitate de un model generic, declarativ a tuturor caracteristicilor interfeței cu utilizatorul, precum componenta de prezentare, de dialog şi toate caracteristicile domeniului asociat cu sarcinile utilizatorului.

Un aspect al utilizării modelului MVC constă în faptul că atât timp cât logica este creată separat (care este un program în sine), acesta poate fi reutilizat pentru orice prezentare grafică în orice moment de timp.

Pe măsură ce dimensiunea unui proiect creşte, nu de puține ori, apare senzația că acesta începe să alunece şi este din ce în ce mai greu de menținut, senzația că fiecare modificare introduce regresii în alte părți, sau pur şi simplu anumite părți nu mai funcționează deloc. Acest lucru este cauzat, de cele mai multe ori, de un cod de proiectare defectuos. În cadrul instrumentelor software mari, unul din principalele deziderate, pe care le vizează arhitectura, este decuplarea cât mai bună a interfeței grafice de logică. Există motive pentru ca funcționalitatea logică să fie păstrată separat de interfață:

Astfel, Code-Behind este codul intermediar între prezentarea grafică şi business logica sistemului. Acest cod, de exemplu, realizează maparea dintre elementele grafice şi obiectele cu informații care trebuie să fie prezentate pe interfață.

5. Concluzii

Cu toate că mai este mult până la automatizarea completă a procesului de proiectare a bazei de date, rezultatele de până acum s-a soldat cu succese în ce priveşte reducerea semnificativă a eforturilor umane în proiectarea unor baze de date viabile şi au punctat calea spre automatizarea ulterioară. Procesul de elaborare a acestui instrument a redus la algoritmi multe din acele

activităţi folosite şi considerate „artă a proiectării bazei de date”.

Există mai multe modalităţi în care instrumentele de proiectare ar putea fi dezvoltate în continuare. În primul rând, în special, se cere crearea unor interfeţe bazate pe web, care ar face instrumentele mai uşor utilizabile şi mai larg accesibile. Aceasta ar permite, în plus, îmbunătăţirea semnificativă la capacitatea de asigurare a disponibilităţii de vizualizare a explicaţiilor, care ar putea fi citite în ferestre pop-up şi ar asista proiectantul la toate etapele procesului de proiectare.

În al doilea rând, pentru a facilita interacţiunea cu proiectantul, instrumentele trebuie să ofere interfeţe semantice bogate şi uşor gestionabile. Instrumentul se cere proiectat ca un sistem expert în sensul că ar oferi o bază de cunoştinţe evolutive; ar accepta specificaţii incomplete; ar justifică şi explica rezultatele oferite; ar permite revenirea la orice etapă de proiectare, în scopul de a schimba specificaţiile sau pentru a cere explicaţii.

Desigur, toate aceste obiective sunt departe de a fi de ajuns pentru a pune la punct un sistem automatizat de proiectare. Cu toate acestea, arhitectura propusă permite atingerea acestor scopuri.

REFERINŢE

1. **AMBLER, S.W.:** The Object Primer 3-rd Edition: Agile Modeling Driven Development with UML2. Cambridge University Press, 2004, 545 p.
2. **COTELEA, VITALIE; COTELEA MARIAN:** Microsoft SQL Server 2008: Lucrări practice. Baze de date. Editura ASEM, Chişinău, 2009, 205 p.
3. **COTELEA, VITALIE; COTELEA MARIAN:** Oracle 11g: SQL, PL/SQL. Editura ASEM, Chişinău, 2011, 396 p.
4. **COTELEA, VITALIE, PRIPA STELA:** F# - limbaj funcţional în .NET. Tutorial, Editura ASEM, Chişinău, 2009, 200 p.
5. **COTELEA, VITALIE; PRIPA STELA:** Integrarea paradigmelor de programare funcţională şi procedurală în elaborarea unui sistem de analiză şi proiectare a bazelor de date. Lucrările Conf. ştiinţifice internaţionale “Competitivitatea şi inovarea în economia cunoaşterii”, 24-25 sept. 2010, Vol. I, Editura ASEM, Chişinău, 2010, pp. 306-310.
6. **COTELEA, VITALIE:** Fragmentarea verticală a bazei de date, utilizând algoritmii genetici. În: Drept, economie şi informatică, Nr.1(11), 2007, Chşinău, pp. 88-90.
7. **COTELEA, VITALIE:** Lists manipulation in turbo Prolog. În: Computer Science Journal of Moldova, Chişinău, Vol. 3, Nr. 1, 1995, pp. 10-23.
8. **COTELEA, VITALIE:** Programarea în logică. Editura Nestor, Chişinău, 2000, 394 p.
9. **COTELEA, VITALIE:** Reţele neuronale artificiale cu arhitectură evolutivă. În: Analele Academiei de Studii Economice din Moldova, Editura A.S.E.M., 2001, Chişinău, pp. 491-497.
10. **COTELEA, VITALIE:** Tehnici exhaustive de rezolvare a problemelor de satisfacere a constrângerilor. În: Drept, economie şi informatică, Nr. 4, 2000, Chişinău-Galaţi, pp. 149-164.
11. Java Community Process. http://www.jcp.org/en/home/index (vizitat 24.09.2011).
12. **SYME, DON; PETRICEK TOMAS; LOMOV DMITRY:** The F# Asynchronous Programming Model. Proc. 13th int. conf. on Principles and Applications of Declarative Languages, ACM SIGPLAN, Springer-Verlag Berlin, Heidelberg, 2011, pp. 175-189.