Academia Română
Secția Ştiința şi Tehnologia Informației
Institutul de Cercetări pentru Inteligență Artificială

Referat III.

Sistem suport pentru decizii bazat pe comunicații: rezultate experimentale.

Doctorand: ing. Mihai BÎZOI
Coordonator ştiințific: Acad. dr. ing. Florin-Gheorghe FILIP

Cuprins

I. Introducere ... 3
II. Construirea modelului experimental ... 4
A. Strategia orientată obiect ... 4
1. Principii ale programării orientate pe obiecte ... 4
2. Obiecte ... 6
3. Mesaje şi metode ... 6
4. Clase şi instanţe ... 7
5. Avantajele programării orientate pe obiecte ... 9
6. Reguli generale de programare orientat-obiect ... 10
7. Modelul MVC ... 11
8. Limbajul Perl şi interfaţa CGI ... 12
B. Metoda iterativă de dezvoltare ... 14
III. Demonstrarea funcţionalităţii modelului experimental ... 17
A. Caracteristicile aplicaţiei *Allego* ... 17
B. Organizarea bazei de date ... 18
C. Modelul aplicaţiei *Allego* ... 23
D. Generarea interfeţei web ... 25
E. Aspecte de securitate ... 28
IV. Concluzii ... 30
V. Referinţe bibliografice ... 32
VI. Anexe ... 35
A. Cod sursă. Clasa AppModel::Stare::Sesiune ... 35
B. Cod sursă. Clasa AppModel::Generare_idei::Brainstorming ... 36
C. Cod sursă. Clasa AppModel::Configurare::Generare ... 36
D. Cod sursă. Pagina *index* ... 38

I. Introducere

Lucrarea de faţă, *Sistem suport pentru decizii bazat pe comunicaţii: rezultate experimentale*, reprezintă o continuare a celor două referate prezentate anterior: *Sistem suport pentru decizii. Utilizare, tehnologie, construire* şi *Arhitectură pentru sistem suport pentru decizii bazat pe comunicaţii*. Referatul vizează prezentarea modului de implementare al unui sistem suport pentru decizii bazat pe comunicaţii din cele două perspective: teoretică şi practică.

Pentru realizarea aplicaţiei (*Allego*), s-a ales abordarea mixtă (care a fost prezentată în primul referat), strategia de dezvoltare fiind cea orientată pe obiecte, iar metoda de implementare aleasă: metoda iterativă.

Capitolul *Construirea modelului experimental* prezintă pe larg elemente fundamentale din teoria programării orientate pe obiecte: principiile de bază ale programării orientate pe obiecte, ce sunt obiectele, mesajele şi metodele, clasele şi instanţele, precum şi avantajele programării orientate pe obiecte. Această fundamentare teoretică fiind absolut necesară pentru implementarea modelului aplicaţiei, deoarece limbajul de programare ales funcţionează atât în mod procedural cât şi în mod orientat obiect.

În acelaşi capitol sunt prezentate reguli generale pentru realizarea aplicaţiilor orientate obiect şi o prezentare a limbajului de programare ales (Perl), însoţit de motivaţia alegerii acestuia. Deoarece modelul aplicaţiei a fost conceput în arhitectura prezentată în referatul al doilea, ca fiind separat de interfaţă, a fost acordată o atenţie deosebită modelului MVC – un concept teoretic care presupune separarea funcţională a interfeţei de aplicaţia model.

În capitolul *Demonstrarea funcţionalităţii modelului experimental*, sunt prezentate caracteristicile aplicaţiei *Allego* prin comparaţie cu caracteristicile ideale ale unui sistem suport pentru decizii.

De asemenea, se prezintă modul de organizare al bazei de date, exemplificând prin câteva exemple cum a fost realizată structura bazei de date. Modul de realizare al claselor aplicaţiei model şi cum se generează interfaţa se prezintă prin câteva exemple practice. De altfel, codul sursă pentru câteva clase ale aplicaţiei model şi prima pagină a interfeţei web sunt anexate acestui document.

II. Construirea modelului experimental

A. Strategia orientată obiect

În vederea construirii sistemelor informatice pot fi identificate mai multe strategii de dezvoltare. Printre acestea se numără: strategia descompunerii funcționale, strategia fluxului de date, strategia orientată pe structura datelor şi strategia orientată obiect.

Strategia orientată obiect foloseşte conceptul de obiect, considerat o entitate care poate fi distinsă de alte entități şi care are sens în cadrul aplicației. Obiectul asociază datele şi acțiunile de procesare în cadrul aceleiaşi entități, doar interfața obiect fiind vizibilă în exterior (Suduc & Bîzoi, 2008).

Abordarea structurală specifică strategiei orientate obiect presupune un caracter conceptual pronunțat care diminuează distanța semantică dintre modelul sistemului şi realitate. Interacțiunea redusă dintre obiecte şi coeziunea puternică obținută prin încapsulare şi polimorfism permite o localizare mai bună a modificărilor, conducând la un nivel de risc scăzut al efectelor neaşteptate.

(Filip, 2004) consideră că strategia de dezvoltare orientată obiect este cea mai potrivită pentru proiectarea şi dezvoltarea unui sistem suport pentru decizii.

1. Principii ale programării orientate pe obiecte

Există în principal patru principii ale programării orientate pe obiecte (Bîzoi & Suduc, 2008):

1. *Încapsularea sau ascunderea datelor*. Încapsularea este procesul ascunderii tuturor detaliilor unui obiect care nu contribuie la caracteristicile lui esențiale. În esență ce se află în interiorul clasei este ascuns, alte obiecte cunoscând doar interfața externă. Avantajul încapsulării este acela că utilizatorul datelor nu trebuie să cunoască cum, unde şi în ce formă sunt stocate datele. Aceasta înseamnă că utilizatorul datelor nu va fi afectat în cazul în care intervin modificări în modul în care sunt stocate datele. Limbajele de programare orientate pe obiecte furnizează facilități de încapsulare prin prezentarea utilizatorului unui obiect a unui set de interfețe externe. Aceste interfețe stabilesc ce va răspunde obiectul la o anumită cerere. Aceste interfețe nu numai că evită necesitatea apelantului de a înțelege cum lucrează detaliile interne de implementare, de fapt, previne posibilitatea ca utilizatorul să obțină acces la date. De aceea, un utilizator nu poate accesa în mod direct datele păstrate într-un obiect din moment ce acestea nu sunt vizibile pentru el.

2. *Moştenirea*. În multe cazuri obiectele pot avea proprietăţi similare (dar nu identice). O metodă de a clasifica aceste proprietăţi este de a crea o ierarhie a claselor. În această ierarhie clasa moşteneşte atât clasa părinte aflată superior în ierarhie cât şi clasa superioară clasei părinte. Acest mecanism de moştenire permite definirea o singură dată a caracteristicilor comune ale obiectelor care sunt folosite în locuri diferite. Moştenirea ne permite să afirmăm că o clasă este similară cu altă clasă, dar cu un anumit set de diferenţe. O altă modalitate de aplicare a acesteia poate fi considerată posibilitatea de a defini toate lucrurile care sunt comune despre o clasă de lucruri şi apoi definirea a ceea ce este special despre fiecare grupare în cadrul unei subclase.
3. *Abstractizarea*. Reprezintă caracteristica esenţială a unui obiect de a se distinge faţă de toate celelalte obiecte şi care furnizează graniţe conceptuale clare, relative din perspectiva utilizatorului. Aceasta este de fapt starea prin care un obiect diferă de toate celelalte. În anumite limbaje de programare, abstractizarea este considerată în legătură cu protecţia datelor. De exemplu, limbajele C++ şi Java au posibilitatea de a stabili dacă o subclasă poate suprascrie date sau proceduri. Alte limbaje, cum ar fi Smalltalk, nu furnizează posibilitatea de a stabili dacă o procedură poate fi suprascrisă, dar permite programatorului să stabilească dacă o procedură (sau metodă) este responsabilitatea unei subclase.
4. *Polimorfismul*. Este abilitatea de a transmite acelaşi mesaj unor instanţe diferite care par să îndeplinească aceleaşi funcţii. Modul în care mesajul este înţeles diferă însă de clasa fiecărei instanţe. În programarea orientată pe obiecte există două tipuri de polimorfism: *supraîncărcat* şi *superior*. Diferenţa dintre cele două nume se referă la modul în care mecanismul folosit determină ce cod este executat. Polimorfismul supraîncărcat apare când procedurile au acelaşi nume dar sunt aplicate unor tipuri de date diferite. Compilatorul poate prin urmare să determine care operator va fi utilizat la momentul compilării şi dacă poate folosi versiunea corectă a operatorului. Este considerat polimorfism superior când procedura este definită atât în clasă cât şi în subclasă. Aceasta înseamnă că atât clasa cât şi subclasa vor răspunde la cererea pentru această procedură (considerând că nu a fost stabilită ca privată pentru clasă). În anumite limbaje cum ar fi Smalltalk şi Java, alegerea versiunii de procedură ce trebuie executată nu se determină la momentul compilării, în schimb ea este aleasă la momentul execuţiei (*run time*). Astfel, compilatorul trebuie să poată determina ce tip de obiect va fi operat şi ce versiune de procedură va executa în cele din urmă.

2. Obiecte

Obiectul este conceptul fundamental care stă la baza programării orientate pe obiecte. Un obiect este o combinație a două componente:

- *date* – reprezintă starea unui obiect;
- *operații* – toate mecanismele de accesare şi manipularea a stării obiectului.

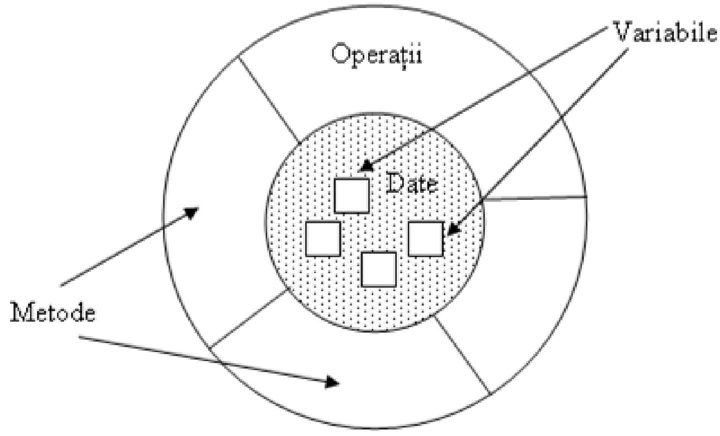

Figura II.1. Reprezentarea unui obiect.

Reprezentarea internă a unui obiect constă din variabile care se comportă ca nişte containere sau dacă nu, ca indicatori către alte obiecte. Variabilele interne nu pot fi accesate în mod direct din afara obiectului. În figura II.1. este prezentată structura internă a unui obiect.

Metodele sunt similare cu subrutinele, funcțiile sau procedurile din alte limbaje de programare. Acestea sunt fragmente de cod identificate printr-un nume, care pot fi invocate individual şi care returnează o valoare când se termină de rulat (Bîzoi & Suduc, 2008).

3. Mesaje şi metode

Limbajele de programare orientate pe obiecte folosesc trimiterea de mesaje ca fiind singura posibilitate de a efectua operații. Dacă obiectul destinatar înțelege mesajul care i-a fost transmis, atunci una din operațiile sale (sau metode) va fi efectuată. În aceste condiții se vor efectua anumite calcule şi întotdeauna va fi returnat un rezultat (de exemplu un obiect).

Deoarece starea unui obiect este privată şi nu poate fi accesată în mod direct din afara lui, singura posibilitate de a accesa starea acestuia este de a transmite un mesaj obiectului. În figura II.2. este reprezentat modul în care se transmit mesaje între obiecte. Setul de mesaje la

care răspunde un obiect se numeşte *interfață mesaj*. Trebuie remarcat faptul că un mesaj specifică doar ce operații sunt cerute şi nu cum trebuie acestea efectuate (Bîzoi & Suduc, 2008).

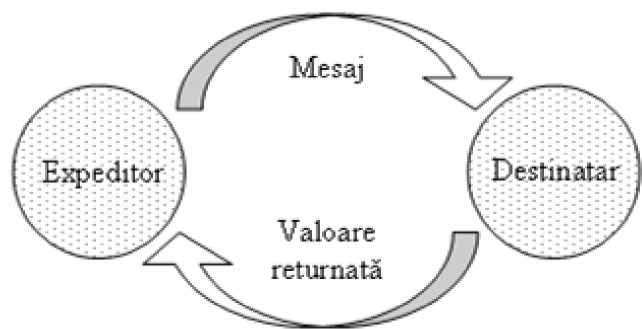

Figura II.2. Transmiterea mesajelor între obiecte.

Un mesaj este transmis unui obiect care este instanța unei anumite clase. Se face o căutare în dicționarul metodelor clasei respective pentru metoda corespunzătoare mesajului selector. Dacă metoda este găsită acolo, atunci aceasta este evaluată şi este întors un răspuns corespunzător. În cazul în care metoda corespunzătoare nu este găsită, atunci se va face o căutare în instanța clasei superclasă imediat următoare. Acest proces se repetă în ierarhia claselor până când metoda este localizată sau nu mai există nici o superclasă. În ultimul caz, sistemul anunță programatorul că a apărut o eroare la rulare.

Mecanismul de trecere a mesajului orientat pe obiecte poate fi comparat cu un apel de funcție în limbajele de programare convenționale cum ar fi C. Este similar cu acesta deoarece punctul de control este mutat la receptor (destinatar), iar obiectul care transmite mesajul (expeditorul) este suspendat până primeşte un răspuns. Diferă totuşi de acesta deoarece receptorul (destinatarul) mesajului nu este determinat când codul a fost creat (timpul compilării), dar este determinat când mesajul este de fapt, transmis (timpul de funcționare).

4. Clase şi instanțe

În teorie, un programator poate implementa un obiect în termenii variabilelor conținute şi a setului de mesaje la care răspunde sau îl înțelege. Oricum, este mult mai util să partajăm informații între obiecte similare. Această abordare permite nu numai ocuparea unui spațiu mai mic de memorie, dar şi posibilitatea de a reutiliza codul.

Informația este partajată prin gruparea acelor obiecte care reprezintă acelaşi tip de entitate într-o structură numită *clasă*. Într-un limbaj de programare orientat pe obiecte, orice obiect este membru al unei singure clase, acesta purtând numele de *instanța* acelei clase. De asemenea, orice obiect conține o referință a clasei căreia îi este instanță.

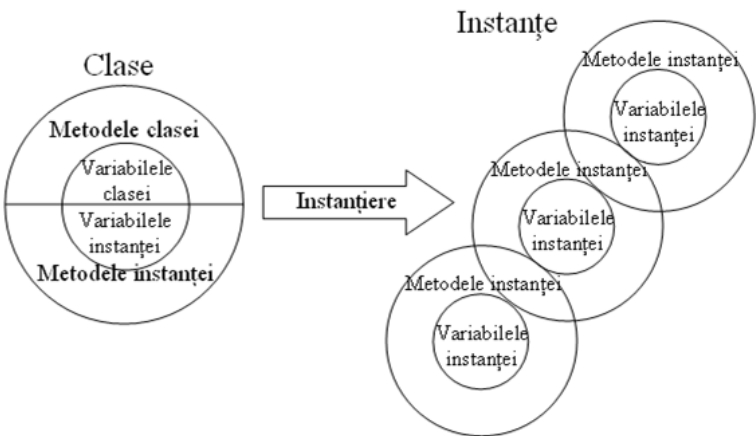

Figura II.3. Clasele – şabloane pentru crearea instanțelor

Clasa unui obiect se comportă ca un şablon pentru a determina numărul variabilelor interne pe care le va avea instanța şi păstrează o listă a metodelor care corespund mesajelor la care toate instanțele clasei vor răspunde. Modul de creare al instanțelor, prin reprezentarea metodelor şi a variabilelor este prezentat în figura II.3.

Această relație dintre clase şi instanțe semnifică faptul că obiectele clase combină două tipuri de cod şi date. Astfel, este codul şi datele pe care clasa obiect îl conține în sine (cunoscute sub numele *metodele clasei* şi *variabilele clasei*) dar şi un şablon pentru codul şi datele pe care instanța clasei îl va conține (cunoscute sub numele de *metodele instanței* şi *variabilele de instanță*).

Obiectele clasă înțeleg numai metodele clasei, iar obiectele instanță înțeleg numai metodele instanței. Programatorul trebuie să fie foarte atent deoarece foarte des pot apărea confuzii. Dacă se aşteaptă ca mesajele transmise obiectelor să fie înțelese, atunci trebuie transmise claselor - mesaje de clasă şi instanțelor - mesaje de instanță.

Orice instanță dintr-o clasă particulară are propriul set de variabile de instanță definit în acea clasă. Acestea nu sunt partajate între instanțe. Astfel, dacă o clasă defineşte o variabilă de instanță numită *cont*, orice instanță a acelei clase va avea o variabilă separată numită *cont*. Valoarea acestei variabile - *cont*, va fi probabil diferită între instanțe.

Situația variabilelor de clasă diferă față de ce a fost prezentat mai sus. Variabilele de clasă sunt definite în clasă şi sunt vizibile şi partajate între toate instanțele acelei clase (Bîzoi & Suduc, 2008).

5. Avantajele programării orientate pe obiecte

Aplicațiile realizate în limbaje de programare orientate pe obiect sunt mai uşor de proiectat datorită organizării pe clase, permit integrarea mai uşoară a modulelor dezvoltate de terțe persoane, sunt robuste, pot fi extinse cu uşurință, iar codul sursă este mai uşor de înțeles şi depanat.

Unul dintre beneficiile cele mai importante ale programării orientate obiect este reutilizabilitatea. Posibilitatea de a utiliza codul creat de alt programator (mai ales cel din biblioteca de clase) şi de a scrie propriile clase reutilizabile, e caracteristica care face din programarea orientată obiect mai productivă decât programarea convențională.

Există câteva metode de reutilizare a codului existent.

Prima modalitate de reutilizare a codului e moştenirea. Moştenirea permite definirea de noi clase prin simpla specificare a diferențelor față de clasele existente. Cum diferențele sunt în mod normal mai mici decât întreaga funcționalitate, se câştigă mult timp aplicând această metodă.

O altă modalitate, poate mai puțin evidentă, dar foarte utilizată în limbajele de programare orientate pe obiecte este simpla creare de instanțe a claselor existente şi folosirea lor fără moştenire. Oricând se utilizează numere, şiruri, colecții şi toate celelalte clase din bibliotecă, de fapt se aplică această metodă de reutilizare a codului.

A treia modalitate de reutilizare o reprezintă folosirea limbajului de programare ca un kit software de construire prin conectarea instanțelor din clasele existente unele la altele, fără a se crea vreo altă clasă.

Reutilizarea codului este o facilitate puternică a limbajelor de programare orientate pe obiecte. Totuşi, clasele care se doresc a fi reutilizate trebuie să îndeplinească anumite criterii:

- Trebuie să fie de calitate înaltă: corecte, de încredere şi robuste. O eroare sau o slăbiciune într-o clasă reutilizată, poate conduce la consecințe pe termen lung. Este important ca programatorii să aibă încredere deplină în clasele oferite pentru reutilizare;
- Trebuie să fie gata pentru folosire. Cerințele pentru documentarea claselor reutilizabile sunt mult mai mari decât în cazul unor aplicații particulare;

- Trebuie să fie uşor de adaptat. Frecvent, o altfel de clasă reutilizabilă nu îndeplineşte nevoile aplicației curente. În cazul în care programatorul original al clasei anticipează schimbările, el le poate implementa sub forma metodelor schelet (numele metodei cu comentarii, dar fără cod). Una din căile care asigură adaptabilitatea unei clase o reprezintă realizarea claselor generale.
- Dacă este posibil, ar trebui să fie portabile între platforme şi domenii. Pot fi folosite în programe diferite, pentru domenii de aplicații diferite şi în diferite dialecte.

6. Reguli generale de programare orientat-obiect

Realizarea aplicațiilor orientate-obiect este o problemă dificilă chiar şi pentru programatorii experimentați în cod procedural. Conform unor studii, programatorii în cod procedural au nevoie de minim 8-12 luni pentru a deprinde ideile gândirii programării orientate-obiect.

Programatorii care folosesc limbaje orientate-obiect, trebuie să aibă în vedere următoarele:

- Realizarea de metode scurte. De exemplu, media numărului de linii din metodele limbajului Smalltalk este 7. Este bine să ne stabilim ca regulă realizarea metodelor scurte, care să nu depăşească fereastra editorului în care lucrăm. În cazul în care nu este posibil, este recomandat să se găsească ideile conceptuale care să permită generarea mai multor metode;
- Realizarea metodelor simple. Se va încerca realizarea metodelor atât de simple, încât funcționarea lor să fie evidentă şi să nu aibă nevoie de documentație;
- Nu se vor folosi obiecte super-inteligente. Obiectele super-inteligente cu numeroase metode reprezintă o gândire procedurală. Gândirea orientată-obiect se bazează pe realizarea de obiecte numeroase de acelaşi tip care se află în cooperare;
- Nu se vor folosi obiecte manager. Este foarte simplu de realizat un obiect de tip manager care să transmită instrucțiuni celorlalte obiecte. Un bun sistem orientat-obiect se bazează pe lucrul cooperativ între obiecte, prin distribuirea rezonabilă a încărcării şi responsabilităților;
- Se vor crea obiecte cu responsabilități clare. Fiecare clasă ar trebui să aibă responsabilități clare care pot fi descrise clar într-o singură propoziție;

- Numărul variabilelor de instanță va fi limitat. Dacă sunt declarate prea multe variabile de instanță, implicit încărcarea pe obiecte nu mai este distribuită corect. Se recomandă definirea unui alt obiect care să joace un rol suport.

7. Modelul MVC

Modelul MVC (*Model-View-Controller*) a fost conceput la sfârşitul anilor 1970 şi are cel mai important rol în istoria dezvoltării interfețelor utilizator orientate obiect. Modelul are la bază descompunerea aplicațiilor prin separarea elementelor de programare a interfeței utilizator de cele ale aplicației reale.

Unul dintre cele mai discutate aspecte ale limbajului Smalltalk (primul limbaj de programare orientat obiect pur) dar şi cel mai puțin înțeles, îl reprezintă arhitectura MVC (*Model-View-Controller*). MVC descrie un mod particular de creare a aplicațiilor care includ interfețe grafice. Când a fost creat Smalltalk-ul, interfețele grafice erau la începutul dezvoltării lor. Cei care au dezvoltat limbajul au creat ulterior MVC pentru a permite crearea acestor categorii de interfețe.

Idea de bază a arhitecturii MVC este ca interfața cu utilizatorul trebuie să fie separată de aplicația în sine (de funcționalitate). Această premisă permite dezvoltarea lor separată şi, cel mai important, permite conectarea cu uşurință a unei noi interfețe la o aplicație existentă. Permite, de asemenea, reutilizarea unor componente ale unei interfețe existente într-o altă aplicație. O aplicație poate folosită şi fără interfața sa, eventual de o altă aplicație. Toate aceste aspecte sunt legate de modularitatea, reutilizabilitatea şi încapsularea specifice programării orientate obiect.

În unele limbaje de programare, cum ar fi *Smalltalk*, separarea aplicației, în partea funcțională şi interfața cu utilizatorul, este realizată prin utilizarea de obiecte separate în implementarea celor două părți. Cele mai importante obiecte din partea funcțională a aplicației sunt denumite *models* (*modele*), iar cele din partea legată de interfața cu utilizatorul sunt denumite *views* (indicatoare) sau *controllers* (controlere). De aici şi denumirea MVC.

Modelul MVC furnizează o cale de a porta o aplicație de pe o platformă pe alta: obiectele modelului ar trebui să se transfere fără sau cu puține probleme – modelele bune sunt transparente față de codul specific platformei. Problema portabilității se reduce astfel numai la rescrierea interfeței utilizator.

Modelul MVC prezintă şi dezavantaje în anumite situații, cum ar fi:

- datorită folosirii interogărilor simple, elementele de grafică ar trebui realizate direct în interfața grafică;
- interfața grafică poate fi programată să realizeze funcții de validare a câmpurilor, accelerând astfel funcționarea aplicației;
- în modelul clasic, interfața informează modelul de orice schimbare, iar modelul transmite informații de actualizare interfeței utilizator. Această situație conduce la o încetinire în funcționalitate. De aceea, se în practică se foloseşte o actualizare parțială a interfeței;

8. Limbajul Perl şi interfața CGI

Cu toate că atenția presei este îndreptată către Java şi ActiveX, rolul real de “activator al Internet-ului” îi revine limbajului Perl; un limbaj care reprezintă tot, dar este invizibil lumii analiştilor de tehnologie profesionişti şi care țese larg în mintea oricui webmaster, administrator de sistem sau programator, aceleaşi zile de lucru care implică construirea de aplicații web personalizate sau integrează programe în scopuri în care proiectanții lor nu au fost destul de prevăzători (O'Reilly & Associates şi alții). Aşa cum Hassan Schroeder, primul webmaster al firmei Sun a remarcat: “Perl este banda adezivă a Internetului”.

Perl a fost dezvoltat inițial de Larry Wall ca un limbaj interpretat pentru UNIX, în încercarea de a crea un limbaj mixt care să combine uşurința în folosire a interpretorului (*shell*) UNIX cu un limbaj de programare asemănător cu C. Perl a devenit rapid limbajul ales de administratorii sistemelor UNIX.

Perl (Limbaj Practic pentru Extragere şi Rapoarte) este un limbaj interpretat. Interpretorul Perl este un program căruia i se furnizează o listă de comenzi care constituie programul Perl. Deoarece interpretorul citeşte şi execută comenzile Perl, programatorii numesc adeseori programele Perl script-uri (scenarii).

Perl are o structură foarte asemănătoare cu cea a limbajului de programare C, de fapt, la prima vedere arată ca un program C. Conține toți operatorii C şi majoritatea structurilor de control (cum ar fi *if* şi *for*), dar într-o formă uşor modificată. Ceea ce lipseşte în Perl sunt pointerii, structurile şi tipurile definite.

Facilități oferite de limbajul Perl:

- Perl a preluat o parte din facilitățile altor limbaje de programare cum ar fi: *C*, *awk*, *sed*, *sh*, şi *BASIC*, printre altele;
- Interfața de integrare a bazelor de date (*DBI*) suportă baze de date de la terțe părți, incluzând: *Oracle, Sybase, Postgres, MySQL* şi altele;

- Perl lucrează cu *HTML, XML* şi alte limbaje care utilizează marcaje;
- Perl suportă *Unicode* şi este compatibil *Y2K*;
- Perl suportă atât programarea procedurală cât şi cea orientată obiect;
- Perl este interfaţat cu librăriile externe C/C++;
- Este extensibil. Astfel, au fost dezvoltate peste 500 de module ale unor terţe părţi, acestea fiind disponibile prin intermediul CPAN (Comprehensive Perl Archive Network);
- Interpretorul Perl poate fi încapsulat în alte sisteme.

Un bun limbaj interpretat este un limbaj de dezvoltare software de nivel înalt, care permite realizarea rapidă şi uşoară a unor aplicaţii neînsemnate, dar conţine şi un flux de procese şi tehnologii de organizare a datelor necesare pentru dezvoltarea aplicaţiilor complexe. Trebuie să fie rapid când se execută. Trebuie să fie eficient când apelează resurse ale sistemului, cum ar fi operaţii cu fişiere, comunicare inter-proces sau controlul proceselor. Un foarte bun limbaj interpretat este acela care rulează pe orice sistem de operare popular, este optimizat pentru procesarea informaţiilor (de tip text), poate prelucra date (numere, date binare), poate fi încapsulat şi extins. Perl îndeplineşte toate aceste criterii.

Odată cu naşterea *World Wide Web*, utilizarea Perl a explodat. *Common Gateway Interface* (CGI) furnizează un mecanism simplu de a transfera date de la serverul web la un alt program şi să returneze rezultatul interacţiunii programului ca o pagină web. Astfel, Perl a devenit rapid limbajul dominant pentru programarea CGI.

CGI (*Common Gateway Interface*) este un standard care permite programelor diferite de pe site-ul Web să interacţioneze cu utilizatorii care vizitează site-ul. Deoarece este un standard, nu este dependent de clientul web, nici de serverul web şi poate fi mutat pe o altă maşina în condiţii de maximă funcţionalitate.

Marele avantaj al folosirii CGI îl reprezintă crearea contextului dinamic, fără ca programele să fie întotdeauna interactive. Se pot folosi scripturi neinteractive pentru a furniza informaţii dinamice, care nu necesită informaţii introduse de utilizator.

Importanţa Perl pentru web poate fi sintetizată în câteva idei astfel:

- Perl este cel mai popular limbaj de programare web datorită capacităţilor de manipulare ale textului şi a ciclului rapid de dezvoltare;
- Modulul Perl *CGI.pm*, parte din distribuţia standard, permite manipularea cu uşurinţă a formularelor HTML;
- Perl poate manipula date criptate web, incluzând tranzacţii de comerţ electronic;

- Perl poate fi încapsulat în servere web pentru a creşte procesarea cu mai mult de 2000%;
- *mod_perl* permite serverului web *Apache* să încapsuleze un interpretor Perl;
- Pachetul *DBI* permite integrarea cu uşurinţă a bazelor de date în web.

B. Metoda iterativă de dezvoltare

Metoda iterativă de dezvoltare este o metodă care implică un dialog permanent între dezvoltator şi utilizator, utilizatorul fiind implicat în dezvoltarea sistemului, iar dezvoltatorul în utilizarea sistemului. O porţiune din sistemul suport pentru decizii este construit rapid, apoi testat, îmbunătăţit şi dezvoltat în paşi sistematici.

Procesul de elaborare este structurat în mai multe cicluri, câte unul pentru fiecare dezvoltare de sub-sistem. Metoda descrisă de Sprague şi Carlson constă din următorii paşi:

- Proiectantul şi utilizatorul definesc împreună o sub-problemă care va reprezenta începutul dezvoltării sub-sistemului. Această sub-problemă trebuie să fie mai puţin importantă ca dimensiune, delimitată clar, dar suficient de importantă ca utilitate pentru decident;
- În acelaşi timp, problema este analizată şi un prototip este elaborat cu uşurinţă. Acest prototip trebuie să includă funcţionalităţile principale ale sistemului;
- Sub-sistemul este utilizat şi evaluat adăugând noi reprezentări, modele şi structuri de control după fiecare ciclu de dezvoltare.

De principiu, prima fază a procesului de dezvoltare iterativ este similar cu procesul clasic până în momentul în care primul prototip este creat. Pornind din acest moment, prototipul este dezvoltat constant până când sistemul final este creat (Marakas, 2003). În figura II.4. este prezentată metoda iterativă de dezvoltare a unui sistem suport pentru decizii.

Sistemul de dezvoltare progresist implică pe lângă avantajele prezentate mai sus şi o cooperare strânsă a categoriilor de actori implicaţi în construcţia sistemului suport pentru decizii. Acesta permite utilizatorilor mai activi implicarea în mare măsură în dezvoltarea SSD.

Această metodă are de asemenea, avantajul de a permite o evaluare constantă a sistemului şi nu numai o evaluare la sfârşitul realizării sistemului, cum este cazul metodei clasice. Acest tip de sistem orientat pe utilizator este flexibil şi permite crearea de noi versiuni pentru a corecta diferite probleme apărute sau pentru a adăuga noi opţiuni (Suduc & Bîzoi, 2008).

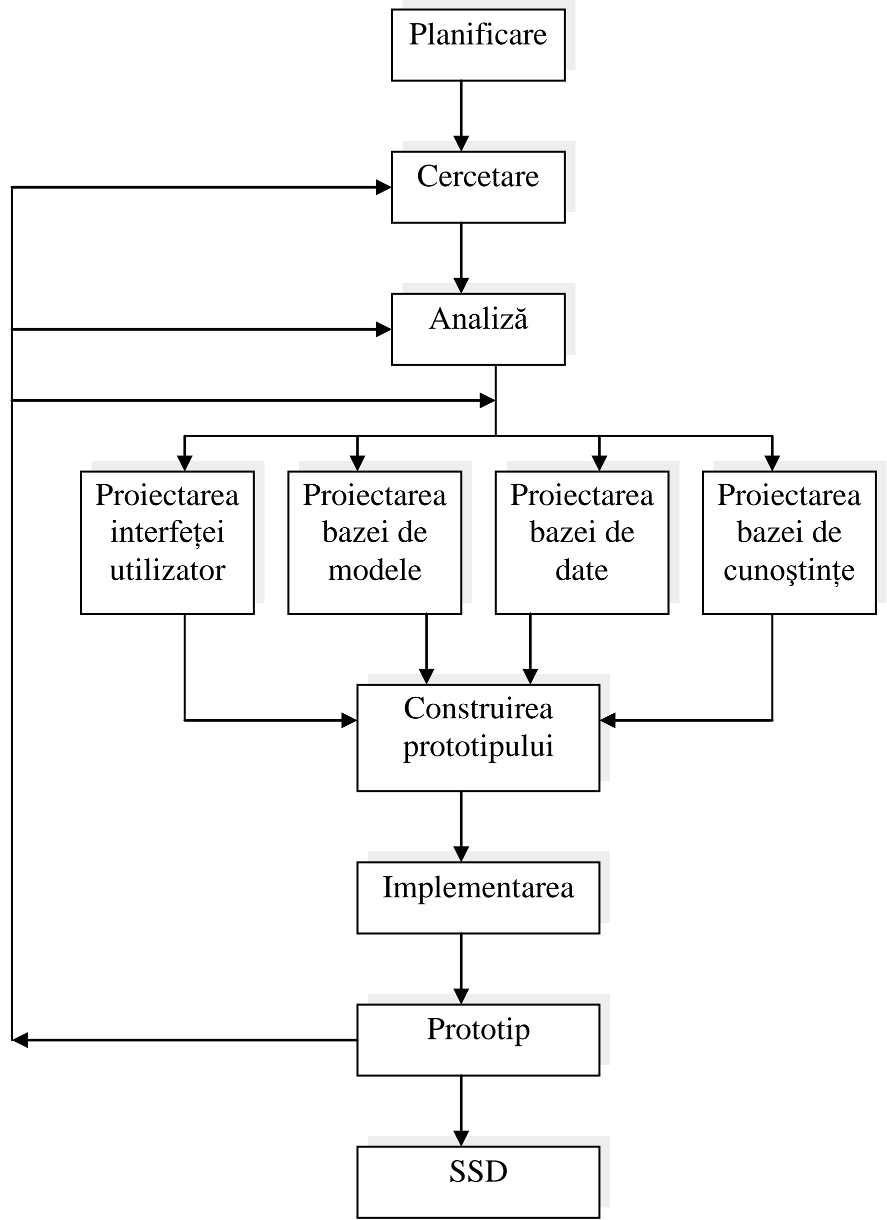

Figura II.4. Dezvoltarea iterativă a SSD

(Turban şi alții, 2005) consideră că metoda iterativă furnizează următoarele avantaje:

- Implică atât utilizatorul cât şi dezvoltatorul;
- Utilizatorul învață despre sistem pe parcursul construirii acestuia, deci poate să înțeleagă mai bine cum să îl folosească şi să-l utilizeze la potențialul maxim;
- Prototipizarea elimină necesitatea cunoaşterii unor informații;
- Timpul dintre două iterații este scurt;
- De regulă, implică costuri reduse.

Printre dezavantaje se pot menționa:

- Pot apărea cerințe pentru schimbări;

- Există riscul apariției neînțelegerilor referitoare la beneficii şi costuri;
- Condiții reduse de testare;
- Se pot ignora elementele de dependență, politicile de securitatea şi măsurile de siguranță;
- Grad de nesiguranță mare;
- Problema de rezolvat poate fi pierdută din vedere;
- Calitatea poate să nu fie atât de mare;
- Pot creşte costurile în urma multiplelor producții.

III. Demonstrarea funcționalității modelului experimental

A. Caracteristicile aplicației *Allego*

În vederea construirii unui sistem suport pentru decizii, trebuie urmate o serie de principii (Suduc & Bîzoi, 2008):

- Să existe o abordare globală a problemei care trebuie rezolvată cu ajutorul aplicației ce va fi create;
- Să existe o metodologie unitară pentru proiectarea şi dezvoltarea sistemului;
- Să se aplice cele mai noi soluții şi tehnici în proiectarea şi dezvoltarea sistemelor informatice;
- Sistemul informatic se va realiza în mod independent față de structura organizațională a companiei unde va fi implementat;
- Viitorii beneficiari direcți ai sistemului vor fi implicați în realizarea activităților de analiză, proiectare şi implementare a sistemului informatic;
- Dezvoltarea activităților de proiectare se va face în concordanță cu legea şi resursele utilizator disponibile;
- Să se prevadă şi eventual controla potențialele modificări în cadrul aplicației;
- Să se documenteze eventualele compromisuri moştenite în timpul construirii software.

Principalele caracteristici ale aplicației *Allego* sunt:

- Aplicația este creată pe baza abordării mixte, iar strategia aleasă este cea orientată obiect;
- Metoda de dezvoltare folosită este metoda iterativă;
- Arhitectura aplicației a fost modelată folosind limbajul UML (diagrame de tip clasă). Pentru dezvoltarea aplicației se foloseşte limbajul Perl în stilul orientat-obiect, tehnologiile software folosite fiind ultimele versiuni stabile la acest moment de timp;
- Aplicația a fost proiectată astfel încât să nu fie specifică unei anumite structuri organizatorice, ci să ofere flexibilitate prin opțiunile de personalizare;
- Datorită faptului că se utilizează metoda iterativă de implementare, potențiali utilizatori sunt implicați în dezvoltarea şi testarea aplicației;

- Aplicația a fost proiectată astfel încât în faza de implementare să se folosească numai tehnologii gratuite, iar eventualele implicații juridice să fie cât mai reduse. Utilizatorii au nevoie doar de existența unui calculator şi a unui *browser* web, deoarece interfața aplicației va fi web;
- Abordarea, strategia de programare şi metoda de implementarea au fost alese pentru a putea modifica şi extinde cu uşurință aplicația;
- Pentru a funcționa, aplicația necesită existența unor aplicații software, cum ar fi: sistem de operare, server de baze de date, server web, server de poştă electronică, limbajul Perl cu anumite module etc. Lista cerințelor funcționale minime va fi prezentată într-un document separat.

Facilități oferite de aplicația *Allego*:

- Aplicația este multiutilizator şi interfațată cu web-ul, neexistând nici o restricție conceptuală în ce priveşte accesul utilizatorilor la aceasta;
- Procesele decizionale ale unor utilizatori diferiți pot fi executate în paralel pe sistem;
- Generarea contextului interfeței se face în mod dinamic;
- Este interactivă, deoarece foloseşte formulare web şi elemente avansate de interfață;
- Datorită modului de implementare, aplicația poate fi tradusă cu uşurință în alte limbi de circulație internațională, prin folosirea metodei hibride de creare a siturilor web multi-lingvistice (Bîzoi şi alții, 2009);
- Poate fi portată cu uşurință pe alte sisteme de operare populare;
- Oferă un grad bun de securitate, care poate fi crescut prin folosirea unor elemente adiționale ale sistemului de operare;

B. Organizarea bazei de date

Pentru implementarea bazei de date a fost ales serverul *MySQL*. *MySQL* este cel mai popular *open source* software pentru baze de date, cu peste 100 de milioane copii ale soft-ului descărcate şi distribuite în istoria sa. Datorită vitezei superioare, fiabilității şi uşurinței în utilizare, *MySQL* a devenit alegerea preferată pentru web (inclusiv versiunea 2.0), pentru companiile de telecomunicații şi pentru consultanții IT deoarece elimină problemele majore asociate cu timpul de răspuns, activitatea de mentenanță şi administrare a aplicațiilor on-line moderne.

Multe din cele mai mari organizații și din cele cu creșterea foarte rapidă folosesc *MySQL* pentru a câștiga timp și bani prin susținerea siturilor web voluminoase, a sistemelor de afaceri critice și a aplicațiilor la pachet – incluzând lideri industriali cum ar fi: *Yahoo!, Alcatel-Lucent, Google, Nokia, YouTube, Wikipedia* etc.

Sistemul de baze de date *MySQL* este deținut și dezvoltat de firma *Sun Microsystems*, una din cele mai mari firme care participă la dezvoltarea soft-ului *open source*. Filosofia firmei pentru acest sistem de baze de date cuprinde următoarele idei:

- *MySQL* trebuie să fie cel mai bun și cel mai utilizat sistem de baze de date din lume pentru aplicații on-line;
- Disponibil și accesibil ca preț pentru toți;
- Ușor de utilizat;
- Îmbunătățit continuu pentru a rămâne rapid, sigur și fiabil;
- Distractiv de utilizat și îmbunătățit;
- Fără erori de programare.

Sistemul *MySQL* poate fi însoțit de instrumente grafice de administrare. În figura III.1. este prezentată interfața instrumentului *MySQL Administrator*, baza de date *allego*.

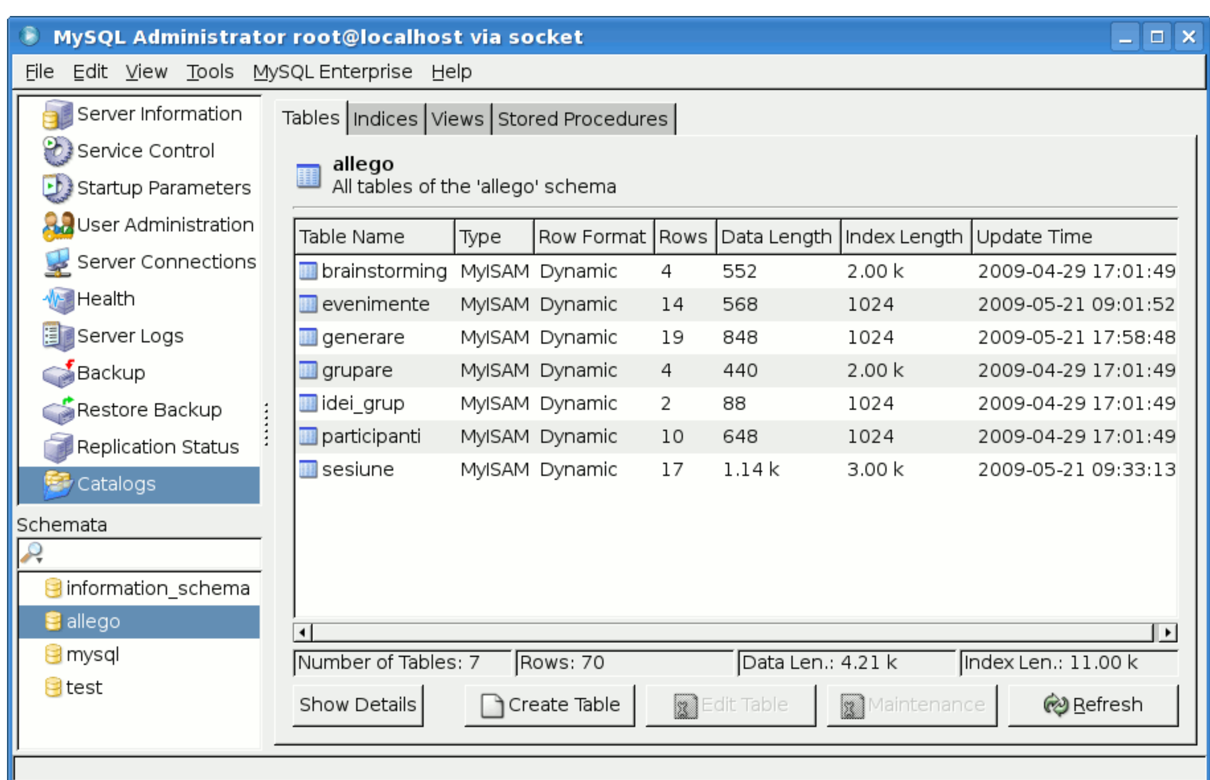

Figura III.1. Fereastra administrator – baza de date *allego*

În această fereastră sunt listate tabele bazei de date *allego*, disponibile la o fază intermediară de dezvoltare a sistemului. Baza de date cuprinde tabelele necesare modelului aplicației A*llego* pentru stocarea permanentă a datelor.

În continuare vor fi prezentate sub formă de exemplu trei structuri de tabele: *sesiune, generare şi brainstorming*. Tabela *sesiune* (figura III.2.) este folosită de clasa *Sesiune* pentru a stoca identificatorul de sesiune (în câmpul *sesiune*), adresa de email a moderatorului (în câmpul *moderator*) şi scopul general al procesului decizional (în câmpul *scop*).

Cele două câmpuri *id* şi *moment* vor avea valoarea stabilită în mod automat de către sistemul de baze de date la momentul de timp al adăugării unei noi înregistrări. Câmpul *id* are rolul de a identifica unic orice încercare a începe un proces decizional. S-a avut în vederea astfel situația în care acelaşi moderator doreşte programarea mai multor şedințe decizionale.

Câmpul *moment* va conține timpul exact la care va fi adăugată înregistrarea în baza de date. Deoarece procesul decizional se desfăşoară în etape, este necesară cunoaşterea momentului de timp exact pentru anumite operații. De asemenea, acesta va fi folosit şi pentru realizarea raportului final al şedinței decizionale.

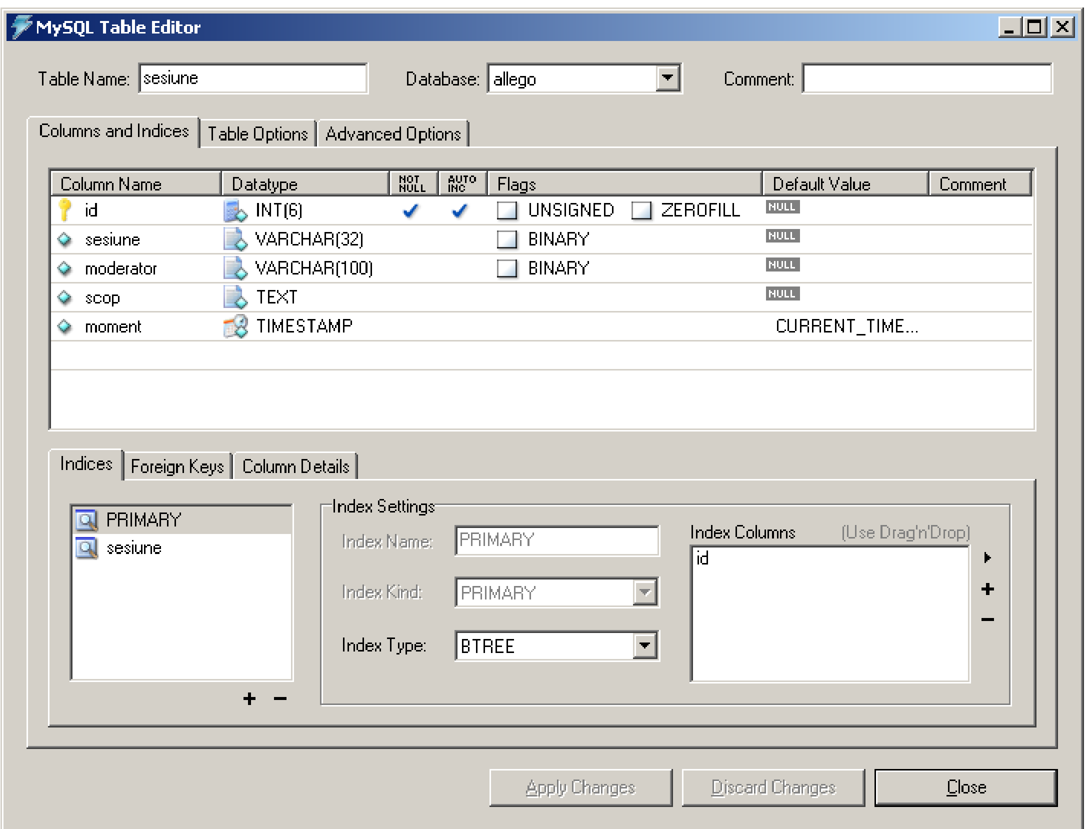

Figura III.2. Structura tabelei *sesiune*

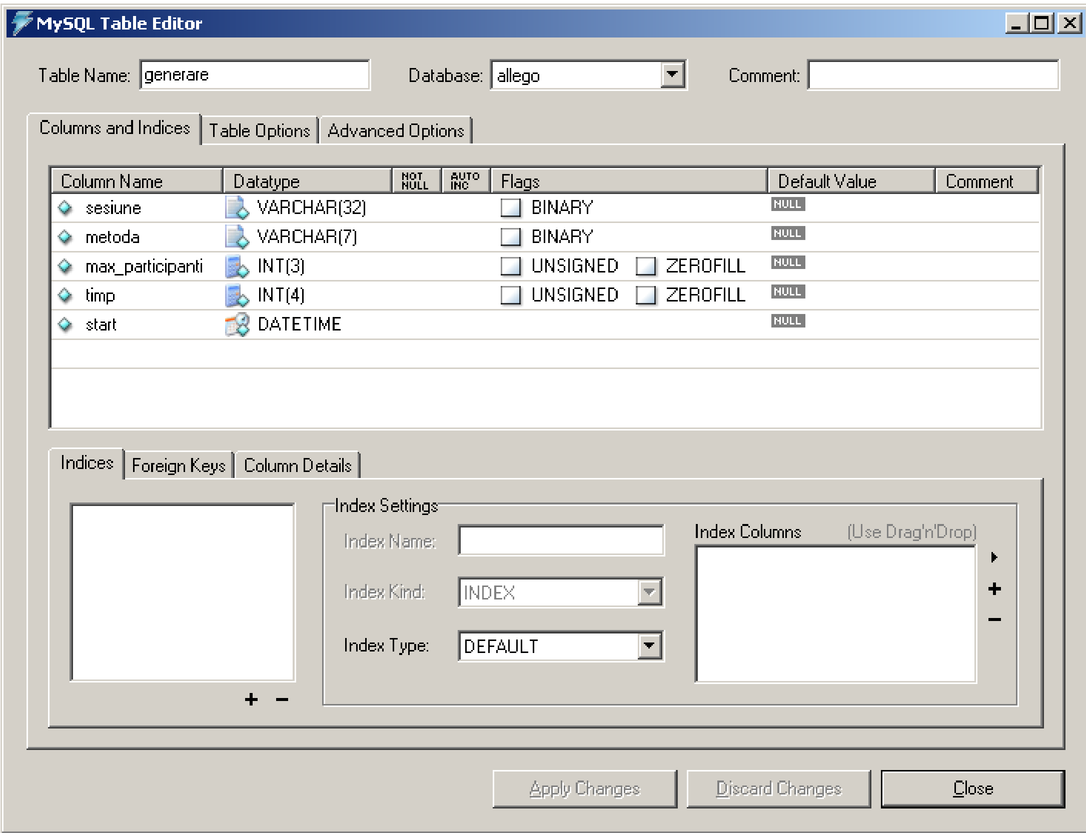

Figura III.3. Structura tabelei *generare*

În figura III.3. este prezentată structura tabelei *generare*. Aceasta este folosită de clasa *Generare*, care face parte din pachetul *Configurare*. Clasa *Generare* este utilizată la configurarea parametrilor inițiali ai ședinței decizionale. Câmpurile tabelei generare vor stoca următoarele informații:

- *sesiune*: identificatorul de sesiune;
- *metoda*: un cod de 7 caractere de forma _:_:_:_ (prezentat în tabelul III.1.);
- *max_participanti*: numărul maxim de participanți la ședința decizională;
- *timp*: timpul exprimat în minute pentru etapa de generare a ideilor;
- *start*: momentul de timp la care va începe ședința decizională.

Etapa procesului decizional	Metoda folosită	Cod
Generarea ideii	Brainstorming	1
	Comentarea subiectelor	2

	Conturarea de grup	3
Organizarea ideilor	Gruparea ideilor	1
	Analiza aparițiilor	2
Prioritizarea ideilor	Votarea ideilor	1
	Chestionar on-line	2
	Dicționarul grupului	3
Elaborarea politicilor	Formularea politicilor	1
	Analiza politicilor	2

Tabelul III.1. Codificarea valorii câmpului *metoda*

Tabela *brainstorming* este utilizată de clasa *Brainstorming* din pachetul *Generare_idei*. Structura acesteia este prezentată în figura III.4.

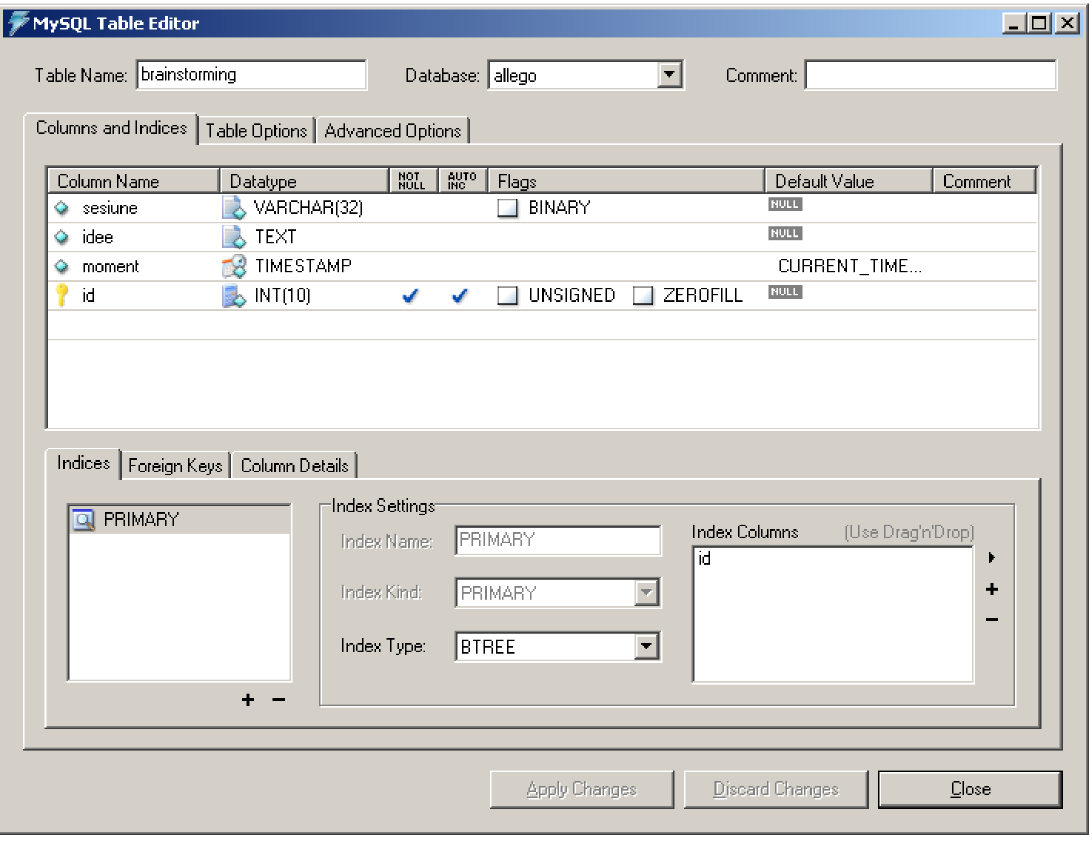

Figura III.4. Structura tabelei *brainstorming*

Tabela *brainstoming* conține toate ideile participanților care apar în etapa de generare a ideii şi în care se va folosi metoda brainstorming. Ideile pentru acelaşi proces decizional sunt identificate prin identificatorul de sesiune; totodată, ideile din aceeaşi sesiune sunt identificate prin valoarea câmpului *id*, un număr întreg generat de serverul de baze de date la momentul adăugării înregistrării unei idei noi.

C. Modelul aplicației *Allego*

Perl este un limbaj de programare care foloseşte module. Un modul este un pachet (*package*) Perl. Obiectele în Perl se bazează pe referințe la date din cadrul unui pachet. Un obiect în Perl este o simplă referință care cunoaşte cărei clasă aparține.

La programarea orientată pe obiecte cu alte limbaje de programare, se declară o clasă şi apoi se creează obiecte ale acelei clase. Toate obiectele unei clase particulare se comportă într-un anumit fel, pe baza metodelor din acea clasă. Se pot crea noi clase sau se pot defini altele noi prin moştenirea proprietăților dintr-o clasă existentă.

Programatorii care sunt deja familiari cu principiile programării orientate pe obiecte vor înțelege rapid terminologia. Perl este şi destul de mult a fost întotdeauna un limbaj de programare orientat obiect.

Există trei afirmații foarte importante pentru a înțelege cum clasele, obiectele şi metodele funcționează în Perl:

- O clasă este un pachet Perl. Acest pachet furnizează metodele pentru obiecte;
- O metodă este o subrutină Perl. Singurul lucru care trebuie precizat este acela că la aceste metode, primul argument este numele clasei;
- Un obiect în Perl este pur şi simplu o referință la anumite date din cadrul clasei.

Una din facilitățile oferite de programarea orientată obiect o reprezintă moştenirea. Moştenirea oferită de Perl presupune doar moştenirea metodelor, pentru a moşteni date, trebuie să creați propriile mecanisme.

Deoarece fiecare clasă este un pachet, are propriul spațiu de nume cu propriile tablouri asociative ale numelor de simboluri. Membrii unei clase pot fi adresați începând cu Perl 5 folosind caracterul două puncte dublat (*$class::$member*).

La momentul reprezentării modelului folosind limbajul UML, clasele aplicației model au fost grupate în pachete. Deoarece termenul de pachet are altă semnificație pentru limbajul Perl, se va preciza că implementarea pachetelor (cu rol de container) din aplicația de modelare s-a făcut sub forma de directoare la nivelul sistemului de fişiere al sistemului de operare.

Astfel, a fost creat directorul *AppModel* care va conține întreaga structură de subdirectoare şi fişiere care vor reprezenta modelul aplicației. Pentru a stoca într-un mod organizat clasele (pachetele) Perl au fost create următoarele 9 subdirectoare: *Generare_idei, Organizare_idei, Prioritizare, Elaborare, Export, Resurse, Comunicare, Stare, Configurare*.

În vederea implementării claselor modelului aplicație a fost realizat un fişier tipar simplu, care va fi utilizat pentru explicarea funcționalității claselor în Perl.

```
#!/usr/bin/perl


package Template;      #numele clasei


sub new {
  my $class = shift;
  my $self  = {};


  bless ($self, $class);  #crearea referintei
  return $self;
  }


1;
```

Tabelul III.2. Codul tipar pentru crearea unei clase

După cum se observă în tabelul III.2., o clasă în Perl este realizată prin declararea acesteia folosind cuvântul *package*. Pachetul trebuie să fie un fişier separat şi conține mai multe subrutine (metode). Subrutina *new()* este folosită pentru inițializarea clasei. Aceasta conține funcția *bless(),* care va crea o referință către obiectul specificat. Referința va fi întoarsă folosind funcția *return()*.

În tabelul III.3. este prezentat sub formă de exemplu codul sursă al subrutinei *sesiune()*.

```
sub sesiune {
  my ($self, $email, $id) = @_;
  my $dbh = DBI->connect("DBI:mysql:$schema:$host",$username,$password);
  my $sth = $dbh->prepare("select sesiune from sesiune where moderator=\"$email\" and
id=\"$id\"");
    $sth->execute;
```

```
my ($sesiune) = $sth->fetchrow_array;
  $sth->finish;
  $dbh->disconnect;
  return $sesiune;
}
```

Tabelul III.3. Codul sursă al subrutinei *sesiune()*

Subrutina *sesiune()* face parte din clasa *Sesiune* şi rolul acesteia este de a returna sesiunea şedinței decizionale specifice. Deoarece informațiile referitoare la sesiuni se găsesc în baza de date, se realizează o conexiune cu serverul de baze de date, după care este interogată tabela sesiune. Informațiile returnate sunt stocate temporar într-o variabilă locală, după care sunt returnate aplicației apelante.

În anexele A, B şi C este prezentat integral codul sursă al claselor *AppModel::Stare::Sesiune*, *AppModel::Generare_idei::Brainstorming*, *AppModel::Configurare::Generare*.

D. Generarea interfeței web

CGI - *Simple Common Gateway Interface Class* este o bibliotecă care foloseşte obiectele perl5 să creeze formulare web şi să prelucreze conținutul lor. Pachetul *CGI.pm* defineşte obiectele *CGI*, entități care conțin valorile interogării curente sau alte variabile de stare. Folosind metodele *CGI* pot fi examinate cuvinte cheie şi parametrii transmişi scriptului şi se pot crea formulare ale căror valori inițiale pot fi preluate din interogarea curentă (de altfel şi conservarea stării informațiilor).

Modulul furnizează funcții prescurtate care produc cod *HTML*, reducând astfel timpul de editare şi apariția erorilor. De altfel furnizează şi anumite funcționalități pentru facilități *CGI* mai avansate, incluzând încărcare de fişiere, cookies, *CSS (Cascading Style Sheets)*, operații cu serverul, cadre etc.

Sunt două stiluri de programare cu *CGI.pm*: un stil orientat obiect şi altul orientat pe funcții. În stilul orientat obiect, se pot crea unul sau mai multe obiecte CGI şi apoi utiliza metode pentru a crea diferite elemente ale paginii. Fiecare obiect CGI începe cu lista numelor parametrilor care sunt transmişi scriptului de către server.

Se pot modifica obiectele, se pot salva într-un fişier sau într-o bază de date şi se pot recrea mai târziu. Acest lucru este posibil pentru că fiecare obiect corespunde "stării" unui script CGI şi pentru că lista parametrilor fiecărui obiect este independentă de a altora.

În tabelul III.4. sunt prezentate o parte din liniile de cod ale fişierului *lib.cgi.* Acesta conține subrutina *head()* care va fi apelată în fiecare script generator de interfață. Această subrutină are rolul de a genera primele linii ale codului *HTML*. Titlul paginii va fi transmis ca parametru acestei subrutine.

Pentru a putea fi folosită, biblioteca *CGI* trebuie declarată. În exemplul de mai jos a fost folosită biblioteca *CGI::Apache*, deoarece scriptul se execută sub serverul web *Apache*. Crearea obiectului CGI se face transmițând mesajul *new* clasei *CGI*.

```
#!usr/bin/perl -w


use CGI::Apache;

use File::Basename;

use XML::Simple;


use AppModel::Comunicare::Comunicare;

use AppModel::Stare::Sesiune;


my $q = new CGI;                    # crearea obiectului CGI

  $q->autoEscape(undef);


sub head {

my ($title) = @_;

print $q->header(),

     $q->start_html(-title =>$title,

                -author=>'Mihai Bizoi & Ana-Maria Suduc',

                -head  =>[$q->Link({'rel'=>'shortcut
icon','href'=>'/favicon.ico','type'=>'image/x-icon'}),

                      $q->Link({'rel'=>'icon','href'=>'/favicon.ico','type'=>'image/x-icon'})],

                -style =>{'src'=>'/styles/main.css'},

                -script=>{'src'=>'/javascript/index.js'},
```

```
                -meta  =>{'keywords'=>'DSS, SSD, decision, decizie',
                          'copyright'=>'Mihai Bizoi & Ana-Maria Suduc, 2004-2009',
                          'robots'=>'index, follow',
                          'Languages'=>'en,ro'});
}
```

Tabelul III.4. Codul subrutinei *head()*

În urma execuției subrutinei *head()*, se generează un cod *HTML* care este prezentat în tabelul III.5. Obiectul *CGI* a preluat valorile parametrilor definiți şi le-a utilizat pentru generarea codului *HTML*.

```
<!DOCTYPE html
 PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en-US" xml:lang="en-US">
<head>
<title>Allego</title>
<link rev="made" href="mailto:Mihai%20Bizoi%20%26%20Ana-Maria%20Suduc" />
<meta name="keywords" content="DSS, SSD, decision, decizie" />
<meta name="copyright" content="Mihai Bizoi & Ana-Maria Suduc, 2004-2009" />
<meta name="Languages" content="en,ro" />
<meta name="robots" content="index, follow" />
<link rel="shortcut icon" href="/favicon.ico" type="image/x-icon" />
<link rel="icon" href="/favicon.ico" type="image/x-icon" />
<link rel="stylesheet" type="text/css" href="/styles/main.css" />
<script src="/javascript/index.js" type="text/javascript"></script>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
</head>
```

Tabelul III.5. Codul *HTML* generat de *head()*

Codul Perl folosit pentru generarea paginii web inițiale este prezentat în tabelul III.6. După cum se observă, toate mesajele pentru generarea codul *HTML* sunt transmise obiectului *CGI* care este referențiat prin *$q*.

```
# Generarea paginii initiale
  print $q->div({-id=>"initial"},
      $q->p($msg->{'about'}),
      $q->p($msg->{'huse'}),
      $q->start_form(-name=>"initial",-method=>"POST"),
```

```
$msg->{'modses'},$q->textfield(-name=>"modses",-size=>"30",-value=>""),
$q->hidden(-name=>"lang",-default=>$lang),
$q->submit(-name=>"initial",-value=>$msg->{'next'}),
$q->end_form,
$q->br,
$q->p({-id=>"disc"},$msg->{'disc'})
);
```

Tabelul III.6. Codul ecranului inițial

În exemplul de mai sus, obiectul CGI generează un mic formular HTML. Formularul va transmite scriptului valoarea a trei parametri. Exemplul nu conține texte utilizator, deoarece acestea vor fi preluate dintr-un *hash* care a fost creat prin citirea unui fişier *XML*. Fişierul *XML* conține toate mesajele utilizator.

În anexa D este prezentat codul sursă al paginii *index* la un moment intermediar de implementare al aplicației.

E. Aspecte de securitate

Rețelele de comunicații sunt în general structuri deschise la care se pot conecta un număr mare şi variat de componente. Complexitatea arhitecturală şi distribuția topologică a rețelelor conduce la o lărgire necontrolată a cercului utilizatorilor cu acces nemijlocit la resursele rețelei (fişiere, baze de date, dispozitive periferice etc.).

Putem vorbi despre o vulnerabilitate a rețelelor care se manifestă pe două planuri:

- posibilitatea modificării sau distrugerii informațiilor (atac la integritatea fizică);
- posibilitatea folosirii neautorizate a informațiilor;

Importanța aspectelor de securitate în rețelele de calculatoare a crescut odată cu extinderea prelucrărilor electronice de date şi a transmiterii acestora prin intermediul rețelelor. În cazul operării asupra unor informații confidențiale, este important ca avantajele de partajare şi comunicare aduse de rețelele de calculatoare să fie susținute de facilități de securitate substanțiale (Bîzoi & Filip, 2008).

În programarea CGI, ca şi în celelalte procese de programare în rețea, siguranța este un criteriu esențial. Adesea trebuie să protejați fişiere şi alte resurse de sistem față de

utilizatorii neatenți sau răuvoitori. Acest lucru poate fi deosebit de important pentru serverele Web care sunt conectate la Internet, unde utilizatorii răuvoitori sunt prezenți cu certitudine.

O cale de a proteja un sistem față de acest tip de atacuri este filtrarea tuturor datelor printr-un program sigur. Astfel, în sistem nu intră decât datele pe care programul "poartă" le consideră sigure.

În mod tradițional, multe servere Internet lucrau sub UNIX şi erau scrise în C (lucru valabil şi astăzi). Cum limbajul de programare C este extrem de eficient, o utilizare eronată a pointerilor C de către programator poate duce la devierea necontrolată a execuției programului şi la slăbirea siguranței sistemului.

Un avantaj al scrierii de programe sigure în Perl este faptul că variabilele şir cresc automat la orice dimensiune solicitată, pentru stocarea caracterelor atribuite variabilei de către script. Cu Perl este exclusă posibilitatea ca scrierea de către program într-o variabilă să ducă la denaturarea valorii altei variabile.

IV. Concluzii

În vederea construirii sistemelor informatice pot fi identificate mai multe strategii de dezvoltare. Strategia orientată obiect foloseşte conceptul de obiect, considerat o entitate care poate fi distinsă de alte entități şi care are sens în cadrul aplicației. Obiectul asociază datele şi acțiunile de procesare în cadrul aceleiaşi entități, doar interfața obiect fiind vizibilă în exterior.

Există în principal patru principii ale programării pe orientate obiecte: încapsularea, moştenirea, abstractizarea şi polimorfismul.

Obiectul este conceptul fundamental care stă la baza programării orientate pe obiecte. Un obiect este o combinație a două componente: date şi operații.

Metodele sunt fragmente de cod identificate printr-un nume, care pot fi invocate individual şi care returnează o valoare când se termină de executat.

Limbajele de programare orientate pe obiecte folosesc trimiterea de mesaje ca fiind singura posibilitate de a efectua operații. Dacă obiectul destinatar înțelege mesajul care i-a fost transmis, atunci una din metodele sale va fi executată.

Informația este partajată prin gruparea acelor obiecte care reprezintă acelaşi tip de entitate într-o structură numită clasă. Într-un limbaj de programare orientat pe obiecte, orice obiect este membru al unei singure clase, acesta purtând numele de instanța acelei clase.

Clasa unui obiect se comportă ca un şablon pentru a determina numărul variabilelor interne pe care le va avea instanța şi păstrează o listă a metodelor care corespund mesajelor la care toate instanțele clasei vor răspunde.

Unul dintre beneficiile cele mai importante ale programării orientate obiect este reutilizabilitatea. Posibilitatea de a utiliza codul creat de alt programator (mai ales cel din biblioteca de clase) şi de a scrie propriile clase reutilizabile, e caracteristica care face din programarea orientată obiect mai productivă decât programarea convențională.

MVC oferă o arhitectură care permite crearea de aplicații cu interfețe grafice. De asemenea, oferă o gamă largă de clase (models, views şi controllers) care lucrează cu această arhitectură şi care pot fi refolosite în alte programe. Arhitectura se bazează pe câteva principii simple, în special pe noțiunea de încapsulare – separarea funcționalității aplicației şi datelor (model) de funcționalitatea modului de prezentare (view) şi de interacțiune (controllers).

O descompunere MVC clară forțează dezvoltatorul să separe noțiunile, aceasta fiind o mare realizarea pentru ingineria programării. Astfel, acesta se concentrează asupra modelului, nefiind interesat de ceea ce va apărea pe ecran. De asemenea, el poate inventa noi metode

pentru interfaţa utilizator, fără a fi nevoie să reproiecteze modelul. Pe scurt, modelul şi interfaţa pot fi realizate separat. Obiectele vor fi mai mici, mai puţin complicate şi cu posibilităţi mai mari de a reutiliza codul creat.

Perl (Limbaj Practic pentru Extragere şi Rapoarte) este un limbaj interpretat. Interpretorul Perl este un program căruia i se furnizează o listă de comenzi care constituie programul Perl. Deoarece interpretorul citeşte şi execută comenzile Perl, programatorii numesc adeseori programele Perl script-uri (scenarii).

Odată cu naşterea *World Wide Web*, utilizarea Perl a explodat. *Common Gateway Interface* (CGI) furnizează un mecanism simplu de a transfera date de la serverul web la un alt program şi să returneze rezultatul interacţiunii programului ca o pagină web. Astfel, Perl a devenit rapid limbajul dominant pentru programarea CGI.

CGI (*Common Gateway Interface*) este un standard care permite programelor diferite de pe site-ul Web să interacţioneze cu utilizatorii care vizitează site-ul. Deoarece este un standard, nu este dependent de clientul web, nici de serverul web şi poate fi mutat pe o altă maşina în condiţii de maximă funcţionalitate.

Metoda iterativă de dezvoltare este o metodă care implică un dialog permanent între dezvoltator şi utilizator, utilizatorul fiind implicat în dezvoltarea sistemului, iar dezvoltatorul în utilizarea sistemului. O porţiune din sistemul suport pentru decizii este construit rapid, apoi testat, îmbunătăţit şi dezvoltat în paşi sistematici.

V. Referințe bibliografice

***, *CGI - Simple Common Gateway Interface Class*, http://perldoc.perl.org/CGI.html, accesat la 15 mai 2009;

***, *Decision support system*, http://en.wikipedia.org/wiki/Decision_support_systems, accesat la 12 decembrie 2008;

***, *Decision Support Systems Glossary*, http://dssresources.com/glossary/dssglossary1999.html, accesat la 23 noiembrie 2008;

***, *MySQL :: About MySQL*, http://www.mysql.com/about/, accesat la 9 mai 2009;

***, *Object Oriented Programming in PERL*, http://www.tutorialspoint.com/perl/perl_oo_perl.htm, accesat la 1 mai 2009;

***, *perltoot - Tom's object-oriented tutorial for perl*, http://perldoc.perl.org/perltoot.html, accesat la 1 mai 2009;

Bîzoi M., Gorghiu G., Suduc A.M., Alexandru A., *Computer Supported Cooperative Work – An Example for Implementing a Web-based Platform Application*, STUDIES IN INFORMATICS AND CONTROL, Volume 15, Number 1, ISSN 1220-1766, pp. 71-78, March 2006;

Bîzoi M., Suduc A.M., Gorghiu G., *Technical Overview of The VccSSe Web System*, 8th Conference: Virtual University, Warsaw University of Technology, June 18 – 20, ISBN 978-83-927469-0-4, Poland, 2008;

Bîzoi, M., Filip, F.G., *Riscurile utilizării sistemelor de comunicații*, Academia Română, Seria Probleme Economice, Vol. 327-328, ISBN 978-973-159-046-2, 2008;

Bîzoi, M., Suduc, A.M., *Bazele programării orientate pe obiecte. Aplicații în limbajul Smalltalk*, Editura Bibliotheca, Târgovişte, ISBN 978-973-712-406-7, 2008;

Bîzoi, M., Suduc, A.M., Gorghiu, G., *Hybrid Method to Design Multi-Language Web Sites*, Proceedings of the 1st International Conference on Computer Supported Education, CSEDU 2009, Lisbon, Vol. 1, p. 427-430, ISBN 978-989-8111-82-1, Portugal, 2009;

FILIP, F., G., *Decizie Asistata de Calculator: Decizii, decidenti - metode si instrumente de baza*, Ed. Tehnica, Bucuresti, 2002;

FILIP, F., G., *Informatica industriala: Noi paradigme si aplicatii*, Ed. Tehnica, Bucuresti, 1999;

FILIP, F., G., *Sisteme suport pentru decizii*, Ed. Tehnica, Bucuresti, 2004;

Filip, F.G., *Decision support and control for large-scale complex systems*. Annual Reviews in Control, 2008;

Gronlund, A., *DSS in a Local Government Context – How to Support Decisions Nobody Wants to Make?*, Electronic Government: 4th International Conference, EGOV 2005, Copenhagen, Denmark, August 22-26, 2005, Proceedings p. 69;

Hättenschwiler, P., *Decision Support Systems*, University of Fribourg, Department of Informatics, DS Group, http://diuf.unifr.ch/ds/courses/dss2002/;

Jaramillo, P., Smith, R., A., Andreu, J., *Multi-Decision-Makers Equalizer: A Multiobjective Decision Support System for Multiple Decision-Makers*, Annals of Operations Research, Volume 138, Number 1, September 2005, pp. 97 - 111;

Koshiba, H., Kato, N., Kunifuji, S., *Awareness in Group Decision: Communication Channel and GDSS*, Knowledge-Based Intelligent Information and Engineering Systems: 9th International Conference, KES 2005, Melbourne, Australia, September 14-16, 2005, Proceedings, Part IV, p. 444;

Marakas, G.M., *Decision Support Systems in 21st Century,* 2nd Edition, Chap.14. Designing and Building Decion Support Systems, 2003, Prentice Hall, 2003;

O'Reilly & Associates, Inc. and Smith, B., House, R., *The Importance of Perl - O'Reilly Media*, http://www.oreillynet.com/pub/a/oreilly/perl/news/importance_0498.html, accesat la 2 mai 2009;

Sprague, R.,H., *A framework for the development of Decision Support Systems* http://web.njit.edu/~bieber/CIS677F98/readings/sprague80.pdf

Sprague, R.H., *A Framework for the Development of Decision Support Systems*, 1980. See also: http://web.njit.edu/~bieber/CIS677F98/readings/sprague80.pdf;

Suduc A. M., Bîzoi M., *An Overview of the DSS Development*, Scientific Bulletin of Electrical Engineering Faculty - 2008, Year 8, No. 1, ISSN 1843-6188, 2008;

Turban, Aronson, Liang, *Decision Support Systems and Intelligent Systems*, 7th edition, Prentice Hall, 2005;

Turban, E. & Aronson, J.E., 1998, *Decision Support Systems and Intelligent Systems*, Prentice-Hall International Inc., London;

VI. Anexe

A. Cod sursă. Clasa AppModel::Stare::Sesiune

```
#!/usr/bin/perl

package Sesiune;

use Digest::MD5;
use DBI;
require "./config.pl";

sub new {
  my $class = shift;
  my $self  = {};

  bless ($self, $class);
  return $self;
  }

sub sesiune {
  my ($self, $email, $id) = @_;
  my $dbh = DBI->connect("DBI:mysql:$schema:$host",$username,$password);
  my $sth = $dbh->prepare("select sesiune from sesiune where moderator=\"$email\" and
id=\"$id\"");
    $sth->execute;
  my ($sesiune) = $sth->fetchrow_array;
    $sth->finish;
    $dbh->disconnect;
    return $sesiune;
  }

sub stabileste_sesiune {
  my ($self, $email) = @_;
  my $ctx = Digest::MD5->new;
    $ctx->add($email);
    $ctx->add(localtime);
  my $digest = $ctx->hexdigest;

  my $dbh = DBI->connect("DBI:mysql:$schema:$host",$username,$password);
  my $sth = $dbh->do("insert into sesiune(sesiune, moderator)
values(\"$digest\",\"$email\")");
  my $oth = $dbh->do("insert into evenimente(sesiune) values(\"$digest\")");
  my $pth = $dbh->do("insert into generare(sesiune) values(\"$digest\")");
  my $ith = $dbh->prepare("select id from sesiune where moderator=\"$email\"");
    $ith->execute();
  my ($id) = $ith->fetchrow_array;
    $ith->finish;
    $dbh->disconnect;
    return $id;
```

```
  }

1;
```

B. Cod sursă. Clasa AppModel::Generare_idei::Brainstorming

```
#!/usr/bin/perl

package Brainstorming;

use DBI;
require "./config.pl";

sub new {
  my $class = shift;
  my $self  = { };

  bless ($self, $class);
  return $self;
}

sub adauga_idee {
  my ($self, $sesiune, $idee) = @_;
  my $dbh = DBI->connect("DBI:mysql:$schema:$host",$username,$password);
  my $sth = $dbh->do("insert into brainstorming(sesiune, idee)
values(\"$sesiune\",\"$idee\")");
    $dbh->disconnect;
  }

sub intoarce_idei {
  my ($self, $sesiune) = @_;
  my $dbh = DBI->connect("DBI:mysql:$schema:$host",$username,$password);
  my $sth = $dbh->prepare("select idee from brainstorming where sesiune=\"$sesiune\"");
    $sth->execute;
    while (my @fields = $sth->fetchrow_array) { push(@lista_idei, $fields[0]); }
    $sth->finish;
    $dbh->disconnect;
    return @lista_idei;
  }

1;
```

C. Cod sursă. Clasa AppModel::Configurare::Generare

```
#!/usr/bin/perl

package Generare;
```

```
use DBI;
require "./config.pl";

sub new {
  my $class = shift;
  my $self  = {};

  bless ($self, $class);
  return $self;
  }

sub stabileste_metoda {
  my ($self, $sesiune, $metoda) = @_;
  my $dbh = DBI->connect("DBI:mysql:$schema:$host",$username,$password);
  if ( defined $metoda ) {
   my $sth = $dbh->do("update generare set metoda=\"$metoda\" where
sesiune=\"$sesiune\"");
   } else {
    my $sth = $dbh->prepare("select metoda from generare where sesiune=\"$sesiune\"");
      $sth->execute;
    my ($metoda) = $sth->fetchrow_array;
      $sth->finish;
   }
    $dbh->disconnect;
    return $metoda;
  }

sub stabileste_nr_participanti {
  my ($self, $sesiune, $nr_participanti) = @_;
  my $dbh = DBI->connect("DBI:mysql:$schema:$host",$username,$password);
  if ( defined $nr_participanti ) {
   my $sth = $dbh->do("update generare set max_participanti=\"$nr_participanti\" where
sesiune=\"$sesiune\"");
   } else {
    my $sth = $dbh->prepare("select max_participanti from generare where
sesiune=\"$sesiune\"");
      $sth->execute;
    my ($nr_participanti) = $sth->fetchrow_array;
      $sth->finish;
   }
    $dbh->disconnect;
    return $nr_participanti;

  }

sub stabileste_perioada_timp {
  my ($self, $sesiune, $timp, $start) = @_;
  my $dbh = DBI->connect("DBI:mysql:$schema:$host",$username,$password);
  if ( defined $timp and defined $start ) {
   my $sth = $dbh->do("update generare set timp=\"$timp\",start=\"$start\" where
```

```
sesiune=\"$sesiune\"");
   } else {
    my $sth = $dbh->prepare("select timp,start from generare where sesiune=\"$sesiune\"");
       $sth->execute;
    my ($timp,$start) = $sth->fetchrow_array;
       $sth->finish;
   }
    $dbh->disconnect;
    return $timp,$start;

  }

1;
```

D. Cod sursă. Pagina *index*

```
#!/usr/bin/perl -w

use CGI::Apache;
use File::Basename;
use XML::Simple;
require "./lib.cgi";
require "./AppModel/config.pl";

my $q = new CGI;
   $q->autoEscape(undef);
   $lang = $q->param('lang');

if (!defined $lang) { $lang = 'ro'; }
if (! -e "../langs/".$lang."/".basename($0).".xml") { $lang = 'ro'; }
my $msg = XMLin("../langs/".$lang."/".basename($0).".xml",'suppressempty'=>'') if (-e
"../langs/".$lang."/".basename($0).".xml");

my $modses = $q->param('modses');
my $ses = $q->param('s');
my $sid = $q->param('i');
my $in  = $q->param('in');

&head($msg->{'title'});
&top;

if ( defined $modses ) {
  if ($modses =~ m!\s*(\S+@(\S+))\s*!) {
    # Incercare de a deveni moderator

&email_moderator($modses,'localhost','allego@ssai.valahia.ro','allego.ssai.valahia.ro',$lang)
and print $msg->{'confirm'};
  } elsif ( $modses =~ /[0-9a-f]{32}/ ) {
    # Incercare de a deveni participant
```

```
    &participant($modses);
  } else {
    # Sesiune denaturata
    print $msg->{'invalidmodses'};
  }
} elsif ( defined $ses and defined $sid ) {
  if ( defined $in ) {
  # Incarcarea configuratiei in baza de date
  my $scop = $q->param('scop');
  my $gen  = $q->param('gen');
  my $org  = $q->param('org');
  my $prio = $q->param('prio');
  my $pol  = $q->param('pol');
  my $nrp  = $q->param('nrp');
  my $timp = $q->param('timp');
  my $an   = $q->param('an');
  my $luna = $q->param('luna');
  my $zi   = $q->param('zi');
  my $ora  = $q->param('ora');
  my $min  = $q->param('min');
  my $metoda = $gen.":".$org.":".$prio.":".$pol;
  my $start = $an."-".$luna."-".$zi." ".$ora.":".$min.":00";
  my $dbh = DBI->connect("DBI:mysql:$schema:$host",$username,$password);
  $dbh->do("insert into generare values(\"$ses\",\"$metoda\",\"$nrp\",\"$timp\",\"$start\")")
and print "insert ok";
  $dbh->do("update sesiune set scop=\"$scop\" where sesiune=\"$ses\"") and print "update
ok";
  $dbh->disconnect;
  } else {
  # Generarea formularului de configurare
  print $q->div({-id=>"initial"},
       $q->fieldset($q->legend($msg->{'metoda'}),
       $q->start_form(-name=>"cfg",-method=>"POST"),$q->start_table({-border=>"0"}),
       $q->Tr($q->td([$msg->{'scop'},$q->textarea(-name=>"scop",-rows=>"4",-
columns=>"50",-default=>"")])),
       $q->Tr($q->td([$msg->{'gen'},$q->popup_menu(-name=>"gen",-values=>['1','2','3'],-
labels=>{'1'=>"$msg->{gen1}",'2'=>"$msg->{gen2}",'3'=>"$msg->{gen3}"})])),
       $q->Tr($q->td([$msg->{'org'},$q->popup_menu(-name=>"org",-values=>['1','2'],-
labels=>{'1'=>"$msg->{org1}",'2'=>"$msg->{org2}"})])),
       $q->Tr($q->td([$msg->{'prio'},$q->popup_menu(-name=>"prio",-values=>['1','2','3'],-
labels=>{'1'=>"$msg->{prio1}",'2'=>"$msg->{prio2}",'3'=>"$msg->{prio3}"})])),
       $q->Tr($q->td([$msg->{'pol'},$q->popup_menu(-name=>"pol",-values=>['1','2'],-
labels=>{'1'=>"$msg->{pol1}",'2'=>"$msg->{pol2}"})])),
       $q->Tr($q->td([$msg->{'nrp'},$q->textfield(-name=>"nrp",-size=>"10",-value=>"")])),
       $q->Tr($q->td([$msg->{'timp'},$q->textfield(-name=>"timp",-size=>"10",-
value=>"")])),
       $q->Tr($q->td($msg->{'start'}),$q->td(
         $q->popup_menu(-name=>"an",-values=>['2009','2010','2011','2012']),
         $q->popup_menu(-name=>"luna",-values=>['1','2','3','4','5','6','7','8','9','10','11','12']),
         $q->popup_menu(-name=>"zi",-
```

```
values=>['1','2','3','4','5','6','7','8','9','10','11','12','13','14','15','16','17','18','19','20','21','22','23','2
4','25','26','27','28','29','30','31']),
        $q->popup_menu(-name=>"ora",-
values=>['0','1','2','3','4','5','6','7','8','9','10','11','12','13','14','15','16','17','18','19','20','21','22','23'
]),
        $q->popup_menu(-name=>"min",-values=>['00','15','30','45'])
        )),
      $q->Tr($q->td({-colspan=>"2"},
      $q->hidden(-name=>"lang",-default=>$lang),
      $q->hidden(-name=>"s",-default=>$ses),
      $q->hidden(-name=>"i",-default=>$sid),
      $q->hidden(-name=>"in",-default=>"1"),
      $q->submit(-name=>"cfg",-value=>$msg->{'next'}))),
      $q->end_table,$q->end_form
      ));
  }
} else {
# Generarea paginii initiale
  print $q->div({-id=>"initial"},
     $q->p($msg->{'about'}),
     $q->p($msg->{'huse'}),
     $q->start_form(-name=>"initial",-method=>"POST"),
     $msg->{'modses'},$q->textfield(-name=>"modses",-size=>"30",-value=>""),
     $q->hidden(-name=>"lang",-default=>$lang),
     $q->submit(-name=>"initial",-value=>$msg->{'next'}),
     $q->end_form,
     $q->br,
     $q->p({-id=>"disc"},$msg->{'disc'})
     );
}

&bottom;

1;
```