Tehnici de programare

Danciu Gabriel Mihail

Universitatea Transilvania, Facultatea de Inginerie Electrică și Știința Calculatoarelor

Cuprins

1 Introducere

1.1 Scop

Scopul acestui curs este de a oferi o mai buna înțelegere a tehnologiilor si a confruntarilor ce pot sa apara in cariera unui inginer software.

Anumite aspecte din dezvoltarea unui soft sunt mai putin tratate in universitate, desi concepte precum versionarea, unit testing, debug sunt folosite zilnic.

Deoarece orice incercare de acoperi toate instrumentele sau tehnologiile existente este imposibila, in acest curs se vor gasi referinte catre cartile de specialitate.

1.2 Obiective Propuse

1. Deprinderea unor cunoștințe și capacități de a concepe o aplicație respectând standardele de programare.
2. Utilizarea Java sau .Net C# ca limbaje pentru a implementa cunoștințele avansate de programare.
3. Dezvoltarea de aplicații scalabile trecând prin toate etapele procesului de implementare.
4. Utilizarea facilă a conceptelor de mai sus folosind alte limbaje, pentru a realiza aplicații asemănătoare.

1.3 Subiecte

In cadrul acestui curs se vor discuta urmatoarele concepte:

- Dezvoltare software: Aspecte generale
- Evenimente, fire de execuție
- Binding
- Jurnalizare
- Unitati de testare
- Sintaxa CRUD. Comunicare cu o baza SQL
- O.R.M.
- Design Patterns
- SOLID
- Interoperabilitate între limbaje
- Modele de comunicare intre procese
- Dezvoltare Agile

1.4 Desfășurarea activităților didactice

1. Conceptele cheie vor fi prezentate în curs și demonstrate folosind exemple concrete
2. În cadru orelor de laborator studenții vor dezvolta o aplicație pornind de la zero. Conceptele prezentate la curs vor fi aplicate în cadrul laboratorului pentru a fixa mai bine cunoștințele.
3. Nota pe fiecare laborator va fi corelată cu procentul realizat.
4. Examenul va fi de tip open books
5. Platforma de comunicare online cu studenții va fi http://slack.com
6. Materialele vor fi pe http://etc.unitbv.ro/~danciu/
7. Proiectele realizate se vor afla pe https://github.com/

1.5 Prerechizite

Se presupune ca studentul este familiar cu C++, Java sau .NET. Intenția nu este de a repeta conceptele de programare orientata pe obiecte ci de a folosi aceste cunoștiințe pentru a rezolva problemele propuse.

Metodele și tehnicile menționate in acest curs nu sunt singurele existente. In urma parcurgerii acestui curs, intentia este ca studentul sa inteleaga anumite concepte și sa se adapteze usor la altele asemanatoare.

2 Dezvoltare software: Aspecte generale

Deși curricula universitara acopera cu succes aspecte generale despre programare, companiile vor solicita pe langa acestea si anumite cunoștințe cu privire la modul de dezvoltare a unui produs. O parte (introductiva) din aceste cunoștințe și aptitudini vor fi acoperite de acest curs.

Totodata se va descrie o anumita mentalitate cu privire la dezvoltarea software ce presupune asumarea unor responsabilitati si înțelegerea evolutiei unui produs și a modului in care fiecare participant contribuie la acesta.

Pe de alta parte cursul nu ofera retete pentru un mod sanatos de a dezvolta un produs software. Aceasta pentru ca exista deja carti de specialitate:

https://www.amazon.com/Beginning-Application-Lifecycle-Management-Rossberg/dp/1430258128

De aseamenea, dimensiunea, direcțiile sau modul de organizare a companiilor sunt diferite de la companie la companie.

Pentru a exemplifica pa;sii de dezvoltare a unui produs vom folosi un limbaj de nivel inalt la alegerea studentului Java sau C#. Laboratoarele vor fi corelate cu sectiunile acestui curs urmarindu-se elaborarea pasilor ce duc in final la o aplicatie dezvoltata de student. In cadrul laboratorului se va folosi implicit Java si seria de tehnologii care duc la indeplinirea scopului final.

2.1 Modele de interfeţe grafice

Orice aplicație trebuie sa permita utilizatorului interac'iunea vizuala cu aceasta. O interfața grafica este o reprezentare a modului in care utilizatorul interacționeaza cu un program sau aplicatie si a modului in care aceasta raspunde comenzilor. Pentru a descrie actiunile si actorii unei interfete grafice, exista limbaje de modelare precum UML: https://www.amazon.com/Unified-Modeling-Language-User-Guide/dp/0321267974.

Exista doua directii a de adezvolta o interfata grafica: WIMP(Windows Icons Menus Pointer) si post-WIMP. Intr-un sistem WIMP exista o fereastra in care ruleaza firul principal, identificarea vizuala a procesului facandu-se pe baza unei iconite. Optional poate exista un meniu cu ajutorul caruia se pot executa actiunile aplicației. Cu ajutorul unui pointer utilizatorul poate controla diferitele elemente din interiorul ferestrei.

Aplicatiile ce ruleaza pe dispozitivele inteligente cu ecran tactil folosesc o alta clasa de GUI(Graphical User Interface) si anume post-WIMP. Exemple de astfel de interactiuni sunt cele de tip ciupitura, extindere sau rotatie. Aceste interactiuni sunt posibile tocmai datorita existentei unui ecran tactil si a altori senzori pe care nu ii gasim la modelul clasic de computer.

2.2 Design-ul unei aplicaţii din punct de vedere vizual

Pașii pe care trebuie sa ii avem in vedere atunci cand creem o interfața sunt:

1. Analiza scopului interfetei
   - Cine sunt utilizatorii
   - Ce valoare le va aduce produsul
   - Cum se rezolva problema propusa in acest moment
   - Cum se integreaza produsul propus in solutia existenta
   - Care sunt sarcinile indeplinite
   - Care sunt pașii pe care ii vor face utilizatorii (flow)
   - Cum vor fi înștiințați utilizatorii de rezultatul pașilor
   - Cat de usor le este utilizatorilor sa faca acești pași
2. Design
   - Creearea pe hartie a unui model care sa ilustreze cum utilizatorii vor indeplini sarcinile propuse. Acestea pot fi definite ca secvente de ecrane.
   - Portarea schitelor in mediu digital. Aceasta se poate face folosind unelte specializate, ca de exemplu https://www.visual-paradigm.com
3. Evaluare
   - Inainte de a prezenta produsul utilizatorilor se vor testa scenarii de funcționare
   - Testarea produsului de persoane care nu au participat la elaborarea sa. Scopul acestui pas este de a descoperi eventualele lacune de funcționare
   - Efectuarea unui demo in fața utilizatorilor reali. La acest pas se urmaresc dificultatile in efectuarea pașilor
4. Repetarea
   - Dupa ce s-au colectat toate informațiile de la pasul 3 se vor nota probleme identificate si se revine la pasul 1
   - Se itereaza pașii 1-4 pana cand pasul 3 nu mai produce lacune sau dificultati de utilizare

2.3 Indicatii in elaborare unei interfete

Pentru elaborarea unei interfețe cat mai ”prietenoasa” ar trebui sa se tina cont de urmatoarele sfaturi:

1. Sa fie simplu de utilizat. O interfata trebuie sa fie intuitiva si bine organizata. Mai multe detalii despre cum se poate realiza acest lucru se pot gasi in carți de specialitate cum ar fi http://shop.oreilly.com/product/0636920000556.do
2. A se utiliza limbajul utilizatorului. Altfel spus, rareori se folosesc termeni foarte specifici
3. Sa se utilizeze cunoștințele apriori utilizarii aplicatiei. Daca exista modele de utilizare populare, nu are rost sa se reinventeze roata.
4. Sa se incerce minimizarea numarului de sarcini pe care utilizatorul trebuie sa le învețe
5. Resursele folosite la o acțiune trebuie eliberate atunci când acțiunea s-a încheiat: RAII.
6. Consistența. Utilizatorul trebuie sa fie in stare sa învețe un modul al produsului și apoi sa aplice ușor cunoștiințele in alt modul. Felul in care sunt plasate controalele, denumirea acestora, a scurtaturilor s.a.m.d, trebuie sa fie același in orice formular al aplicatiei

2.4 Modele de programare a interfetelor grafice

Dezvoltarea interfețelor grafice poate fi clasificata din cel puțin doua perspective: cel arhitectural sau cel al implmentarii. Daca prima varianta este descrisa de Martin Fowler in http://www.martinfowler.com/eaaDev/uiArchs.html, vom vorbi in continuare despre a doua perspectiva.

Pentru a implmenta o aplicatie folosind un limbaj de nivel inalt, o prima intrebare este legata de mediul in care va rula. Din acest punct Java are un avantaj asupra .NET deoarece limbajul este independent de platforma. Cu toate acestea optiunea pe care Microsoft a luat-o dezvoltand .NET Core https://www.microsoft.com/net/core#windows, indica faptul ca acest avantaj va scade in anii ce urmeaza.

In continuare vom detalia modurile de implementare unei interfețe grafice. Microsoft a promovat modelul Windows Presentation Foundation http://www.wpftutorial.net/ care inlocuieste modelul Windows Forms https://www.scribd.com/doc/100799637/Windows-Forms-Programming-in-C. Un tutorial video pentru a intelege mai bine WPF se gaseste aici:
https://www.youtube.com/watch?v=SLqoDyqU76E&list=PL1chK76wN3A6u3rFJ-HXHVA2QipayU6tE.

Atat WPF cat si JavaFX http://www.apress.com/9781484211434 folosesc un sistem de redare grafica. Interfata grafica foloseste un graf din scene a carui noduri sunt elementele UI. JavaFX va putea preda calculele pentru redarea grafica placii grafice. Sistemul grafic va folosi DirectX in Windows si OpenGL pe alte platforme.

Corespondentul modelului Windows Forms in Java se numeste Swing si cele doua sunt foarte asemanatoare. Pentru design-ul interfetei codul face parte din clasa ce trateaza si evenimentele sau legaturile dintre controalele din formular. Pe de alta parte JavaFX foloseste un limbaj markup FXML ce este similar XAML folosit de WPF. Aceste limbaje ce se bazează defapt pe XML sunt folosite pentru a crea controalele de pe formulare. De asemenea modul in care se tratează evenimentele, legaturile dintre componente se poate defini direct în fișierul XAML sau FXML. Aceasta este una din marile diferente dintre modelul XML si windows forms. Proprietatile precum culoarea, dimensiunea fontului, aspectul general al interfetei se poate controla usor folosind CSS in JavaFX sau ”style sheets” in XAML.

Cu toate acestea modelul GUI pe sistemele Android sau iOS difera de oricare din cele doua. Pentru a implementa usor o interfata care sa functioneze pe orice sistem cea mai buna varianta este modelul web: HTML5. Pentru a nu rescrie aplicatia desktop se poate usor adauga un control care sa incarce pagina web.

Pentru acest curs insa, s-a ales folosirea Java si Swing pentru exemplificarea conceptelor pentru ca este un model simplu de inteles. Totusi, pentru implementarea conceptelor, se poate folosi de catre orice student alte modele din cele de mai sus, dupa preferinte.

3 Evenimente

3.1 Programarea aplicațiilor folosind evenimente

Pentru a înțelege mai bine conceptul de proces ”event-driven” și diferența dintre o aplicație de acest tip și una ce rulează sincron executând în funcție de datele de intrare instrucțiuni de la A_i la A_j, putem sa ne imaginam doua modele:

- un proces automat in care din exterior nu se poate interveni. Exemplu: scanarea unei pagini (fara posibilitatea interuperii curentului sau interacțiunii cu mașina până la terminarea scanării)
- un proces care așteaptă tot timpul acțiunea utilizatorului cum ar fi un joc pe calculator.

Orice model de programare bazata pe evenimente va conține doua componente:

- Sursa - responsabila cu producerea evenimentului
- Ascultator - responsabila cu tratarea evenimentului

Programarea bazată pe evenimente reprezintă o paradigmă de programare în care cursul de execuție al programului este determinat de evenimente. Exemple de evenimente ar fi click-ul mouse-ului, apăsarea unei taste sau un mesaj primit de la sistemul de operare sau un alt program. O aplicație bazată pe evenimente este conceputa să detecteze evenimentele pe parcurs ce ele se produc, și apoi să se ocupe de ele folosind o procedura specifică de tratare a evenimentelor.

Programele bazate pe evenimente pot fi scrise in orice limbaj de operare, cu toate că unele dintre ele (ex. Visual Basic) sunt special concepute pentru a ușura acest tip de programare. Acestea oferă un mediu de dezvoltare integrat (IDE - integrated development environment) ce automatizează partial scrierea codului și oferă o selectie de obiecte și controale standard, fiecare dintre ele putând raspunde unui set de evenimente. Teoretic, toate limbajele orientate pe obiecte și limbajele vizuale, suportă programarea bazată pe evenimente.

Multe medii de dezvoltare vizuale oferă chiar template-uri pentru crearea metodelor de tratare a evenimentelor, astfel incât programatorul trebuie doar sa se ocupe de scrierea codului ce defineste acțiunea pe care programul trebuie să o execute atunci când evenimentul are loc. Fiecare metodă de tratare a evenimentelor este de obicei legată de un anume obiect sau control din interfață. Orice alte subrutine, metode și proceduri necesare sunt de obicei conținute în module de cod separate, și pot fi apelate din alte părți ale programului oricând este nevoie.

Una din ideile de bază din spatele programarii orientate pe obiecte este aceea de a reprezenta entități programabile ca obiecte. O entitate în acest context ar putea reprezenta orice lucru de interes pentru aplicație. De exemplu, o aplicație folosită pentru urmărirea unor comenzi intr-o fabrică, ar putea conține obiecte precum ”client”, ”comandă” si ”obiect comandat”. Un obiect conține atat datele (atributele) ce apartin entității, acțiunile (metodele) ce pot fi folosite pentru a accesa si modifica atributele, cât și evenimentele ce pot determina apelarea metodelor asociate entității. Structura de baza a unui obiect si relatiile cu aplicatia de care apartine sunt ilustrate in diagrama urmatoare:

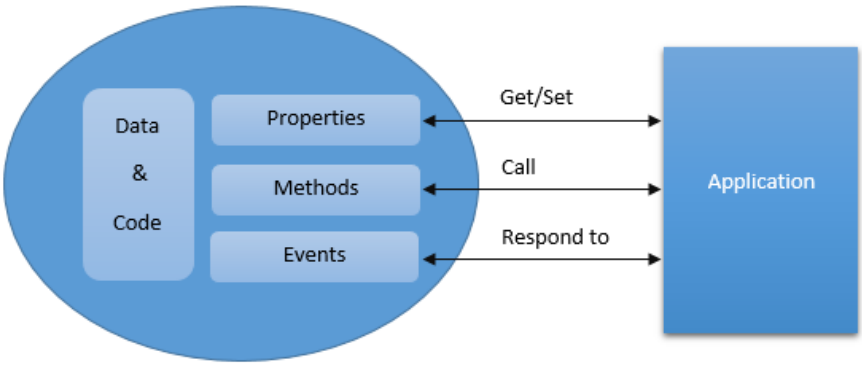

Figura 1. Interacţiunea dintre obiecte si aplicaţie

Obiectele unui limbaj de programare vizual (de obicei numite controale) pot fi impartite in clase (ex: butoane, textBox-uri, etc.), iar intr-o singura fereastra pot aparea mai multe instanțe ale fiecarui control. Fiecare clasa are

atribute (de obicei numite proprietați) ce sunt comune tuturor obiectelor ce-i aparțin (ex: dimensiuni, culoare, etc.), si fiecare clasa definește o lista de evenimente pe care un obiect de tipul respectiv le va genera. Metodele pentru tratarea unor evenimente specifice sunt de obicei oferite ca template-uri la care programatorul trebuie sa adauge codul ce trateaza actiunea.

Evenimentele sunt de obicei actiuni produse de utilizator pe parcursul executiei programului, dar pot fi de asemenea și mesaje generate de sistemul de operare sau de o alta aplicație, sau o intrerupere generata de componenta periferica sau de hardware.

Exemple de evenimente ar fi acțiunea utilizatorului de apasare a unui buton cu mouse-ul sau apasarea unei taste, terminarea acțiunii de citire a unui fișier sau apariția unei erori hardware sau software. Evenimentele sunt procesate de o componenta centrala de tratare a evenimentelor (de obicei numita expeditor) ce ruleaza continuu și asteapta apariția evenimentelor.

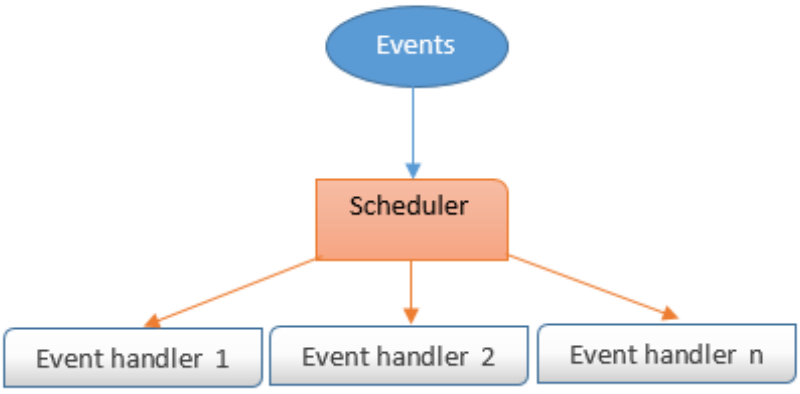

Figura 2. Producerea, transmiterea si tratarea evenimentelor

Metodele de tratare a evenimentelor pot fi vazute ca mici secvente de cod procedural ce se ocupa de tratarea unor cazuri specifice. De obicei, acestea produc un raspuns vizual pentru a informa sau a ghida utilizatorul, si de multe ori vor schimba starea sistemului. Starea sistemului contine atat datele folosite de sistem (de exemplu valoarea continuta de un camp in baza de date), cat si starea interfetei grafice (de exemplu, obiectul ce are focusul la un moment dat).

O metoda de procesare a evenimentelor poate la randul sau genera un alt eveniment, ceea ce va determina apelul unei alte metode de procesare. Trebuie avuta grija la implementarea metodelor de tratare a evenimentelor ce apeleaza alte metode asemanatoare, astfel incat sa se evite intrarea aplicatiei intr-o bucla infinita.

Deasemenea, o metoda se tratare a evenimentelor poate genera indepartarea unor evenimente din lista de asteptare (de exemplu, cand utilizatorul apasa butonul ce inchide aplicatia). Diagrama din imaginea 2 arata relatiile dintre evenimente, planificator si metodele de tratare a evenimentelor dintr-o aplicație.

3.2 Exemplu

Urmatoarea secventa de pseudocod arata cum functioneaza un ”scheduler” de evenimente simplu. Contine o bucla principala ce ruleaza continuu pana la aparitia unei conditii de ieșire din bucla. Cand un eveniment are loc, planificatorul trebuie sa determine tipul evenimentului si sa apeleze metoda de tratare a evenimentelor asociata acestuia, sau sa trateze el evenimentul in cazul in care nu exista o metoda de procesare adecvată.

```
do forever: // bucla principala a planificatorului de evenimente

get event from input stream

if event type == EndProgram:
quit // terminarea buclei la aparitia conditiei de terminare a programului

else if event type == event_01:
call event-handler for event_01 with event parameters
```

```
else if event type == event_02:
call event-handler for event_02 with event parameters

.
.
.

else if event type == event_nn:
call event-handler for event_nn with event parameters

else handle unrecognized event // ignora evenimentul sau arunca o exceptie

end loop
```

Într-un mediu de programare bazat pe evenimente, un număr (mare) de evenimente pot avea loc intr-un interval scurt de timp. Este posibil ca planificatorul, cât și metodele de tratat evenimente, să nu poată procesa toate evenimentele imediat ce acestea au apărut.

Soluția o reprezintă plasarea evenimentelor netratate intr-o coadă de evenimente până la momentul în care pot fi procesate. Evenimentele sunt asezate la finalul cozii pe masura ce ajung, si sunt procesate de planificator in momentul in care ajung la inceputul cozii.

Este posibilă însă și o prioritizare a evenimentelor, astfel incât unele evenimente să fie procesate mai repede fata de altele. Acest tip de evenimente poate fi mutat la inceputul cozii pentru a fi preluat de planificator, dar deasemenea poate fi folosita o coada separata pentru evenimentele prioritare.

Existența cozii garantează că toate evenimentele vor fi procesate la un moment dat, și într-o aparentă ordine. Lungimea cozii si timpul in care evenimentele sunt procesate vor depinde de factori cum ar fi viteza procesorului, dimensiunea memoriei RAM si de numarul altor aplicatii ce rulează la un moment dat (ce folosesc resurse ale sistemului, deasemenea). În majoritatea timpului totuși, coada va fi goală, iar planificatorul va fi intr-o stare de asteptare a unui nou eveniment.

4 Programare in paralel

Un mod elegant de a rezolva o problemă ce are ca intrare un set de date foarte mare, este de a împarți setul respectiv in părți mai mici si a rezolva subproblemele simultan. Un program scris sa ruleze în paralel va folosi procesoare multiple pentru a rezolva o sarcină. Fiecare procesor va executa pașii ce rezolvă o subsarcină, independent de celelalte procesoare. Schimbul de informații dintre procesoare poate avea loc in timp ce subsarcinile sunt executate, sau la finalul execuției lor.

4.1 Arhitecturi paralele

Pornind de la numarul de instructiuni si a datelor care pot fi procesate simultan, sistemele de calcul sunt clasificare in patru categorii :

1. O singura instructiune, o singura data (SISD)
2. O singura instructiune, date multiple (SIMD)
3. multiple instructiuni, o singura data (MISD)
4. multiple instructiuni, date multiple (MIMD)

Sistemul SISD este o masina uniprocesor ce executa un set de instrucțiuni executate secvențial ce opereaza pe un singur stream de date.

Figura 3. Arhitectura SISD

In cadrul unui ciclu, CPU executa urmatoarele operatii:

– Fetch: CPU primeste data si instrucțiunile din memorie (registru)
– Decode: CPU decodifica instructiunile de executat
– Execute: se executa operatiile si rezultatul este stocat in alt registru

Sistemul MISD este un model cu n procesoare, fiecare cu propriul CPU, care partajeaza o singura unitate de memorie. La fiecare ciclu data ce vine din memorie este procesata simultan de toate procesoarele. Paralelismul se obține prin faptul ca se efectueaza mai multe operațiuni pe aceeași data.

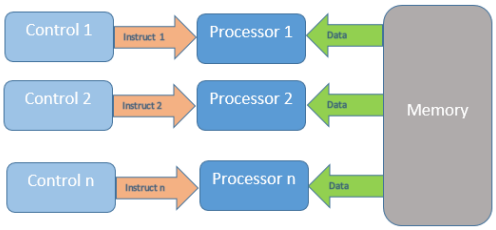

Figura 4. Arhitectura MISD

Un model SIMD consta din n procesoare identice, fiecare cu memoria locala proprie. Toate procesoarele vor controla același stream de date, dar fiecare procesor va lucra pe porțiunea proprie.

Implementarea acestui model se poate realiza fie cu programarea GPU https://developer.nvidia.com/cuda-zone, fie folosind o biblioteca specializata cum ar fi OpenCL https://www.khronos.org/opencl/.

Sistemul MIMD este reprezentat de o clasa de n procesoare și n stream-uri de date. Fiecare procesor are unitatea proprie de procesare și memorie locală proprie. În acest model fiecare procesor poate rezolva o subproblema care este total diferită de restul subproblemelor.

Această arhitectură se poate implementa cu ajutor firelor de execuție și/sau proceselor. Aceasta înseamnă că procesoarele opereaza asincron. Aceasta este arhitectura întâlnită pe majoritatea dispozitivelor, supercomputerelor sau a platformelor ”cloud”.

Mai multe detalii despre acest subiect se pot citi în

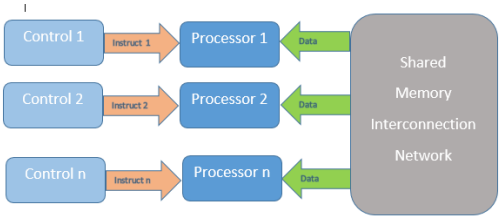

Figura 5. Arhitectura MIMD

https://www.amazon.com/Parallel-Programming-Cookbook-Giancarlo-Zaccone/dp/1785289586.

Un curs foarte bun pentru programatorii in C++ este http://pages.tacc.utexas.edu/~eijkhout/pcse/html/index.html

In cadrul laboratorului se va insista pe programarea multithreading.

5 Legaturi intre date

”Binding” adica legaturile intre date reprezinta o technica de a corela o sursa cu un (sau mai multi) consumator si de a sincroniza cei doi actori. In ceea ce priveste legaturile intre controale de GUI, ”binding” inseamna schimb de informatii de acelasi tip intre obiecte diferite. Uneori informatiile schimbate nu au acelasi tip si atunci apare nevoia de conversie

Avantajele aduse de aceasta tehnica sunt:

1. Reducerea codului
2. Dezvoltarea mai rapida a aplicatiilor
3. Controlul datelor interschimbate (prin validarea lor)

Exista doua tipuri de ”binding”: unidirectional sau in ambele sensuri.

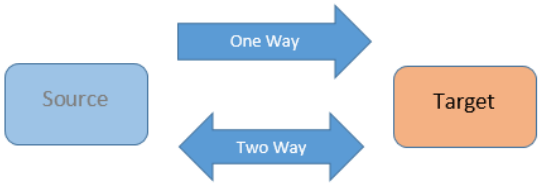

Figura 6. Tipuri de binding

Un alt mod de a imparti tipurile de ”binding” ar fi:

– Early binding. Limbajele native implementeaza de obicei acest tip de binding. Exemplu:

```
Dim oXL As Excel.Application
Set oXL = New Excel.Application
```

  – planul de alocare a memoriei se face la compilare
  – tipul variabilelor se verifica in mod static

  Un alt exemplu in Java:

```
class Dog{
        private void eat(){System.out.println("dog is eating...");}

        public static void main(String args[]){
                Dog d=new Dog();
                d.eat();
        }
}
```

– Late binding. Limbajele interpretate implementeaza acest tip de binding. Exemplu:

```
Dim oXL As Object
Set oXL = CreateObject("Excel.Application")
```

  – permit unei variabile sa aibe mai multe tipuri de data pe parcursul existenței ei
  – verificarea tipului este dinamica

  Un exemplu de binding dinamic in Java:

```
class Animal{
       void eat(){System.out.println("animal is eating...");}
}
class Dog extends Animal{
       void eat(){System.out.println("dog is eating...");}

       public static void main(String args[]){
               Animal a=new Dog();
               a.eat();
       }
}
```

Mai multe exemple de binding vor fi oferite in cadrul laboratorului.

6 Baze de date

6.1 Sistem de management al unei baze de date

Un sistem pentru managenent-ul unei baze de date (DBMS - database management system) păstreaza datele in așa fel încât să usureze accesarea, manipularea si producerea de informatii.

Caracteristicile unui DBMS

- (Real-world entity) Un DBMS foloseste entități din lumea reală pentru a-și modela arhitectura. În acest scop foloseste deopotrivă comportamentul si atributele obiectelor. Spre exemplu, o bază de date pentru o școală poate folosi elevii ca entități si vârsta lor drept atribut.
- Tabele bazate pe relații (Relation-based tables) - DBMS permite formarea de tabele ce sa conțină atât entități cât și relații între acestea. Un user poate să înțeleagă arhitectura unei baze de date doar uitându-se la numele tabelelor și la diagrama bazei de date.
- Separarea datelor de aplicație - Un sistem de management al unei baze de date este total diferit de datele pe care aceasta le conține. O bază de date este o entitate activă, in vreme ce datele sunt entități pasive, ce sunt prelucrate și organizate de către aceasta. Pentru a își ușura munca, DBMS păstrează deasemenea metadate, acestea reprezentând date despre datele stocate.
- Mai puțină redundanță - DBMS folosește regulile de normalizare, ce împart o relație atunci când unul dintre atributele sale are valori redundante.
- Consistență - Consistența este o stare a bazei de date în care fiecare relație a sa ramâneneschimbată. Exista metode de detectare a stărilor inconsistente ale unei baze de date.
- Limbaj bazat pe interogari - DBMS foloseste un limbaj bazat pe interogări, ceea ce il face eficient în accesarea și manipularea datelor. Un utilizator poate aplica oricâte opțiuni de filtrare diferite sunt necesare pentru a returna setul de date dorit.
- Proprietati ACID - DBMS foloseste conceptele: Atomicitate, Consistență, Izolare și Durabilitate (ACID - Atomicity, Consistency, Isolation and Durability). Aceste concepte sunt folosite pentru manipularea datelor dintr-o baza de date și asigură păstrarea stării corecte a acesteia in medii multitranzacționale și în caz de apariție a unor erori.
- Acces concurent pentru mai mulți utilizatori - DBMS suportă accesul în paralel al mai multor utilizatori la baza de date și le permite manipularea datelor simultan. Cu toate că există restricții asupra tranzacțiilor când mai multi utilizatori folosesc aceleași date, aceștia nu sunt conștienti de existența acestora.
- Vizualizări multiple - DBMS oferă multiple vizualizări ale unei baze de date pentru diferiți utilizatori, în funcție de cerințele fiecăruia.
- Securitate - Anumite caracteristici ale DBMS, precum vizualizările multiple, oferă măsuri de securitate într-o anumită masură, astfel încât unii utilizatori nu pot accesa date ale altor utilizatori sau categorii de utilizatori. Deasemenea, DBMS oferă metode de a impune constrângeri la introducerea datelor în baza de date și în returnarea acestora ulterior. Astfel, este posibil ca diferiți utilizatori să aibă vizualizari diferite cu date diferite.

6.2 Utilizatori ai DBMS

Un DBMS obișnuit are utilizatori cu diferite drepturi și permisiuni ce folosesc baza de date in diferite scopuri. Aceștia pot fi împărțiți în următoarele categorii:

- Administratori - Administratorii întrețin DBMS-ul și sunt responsabili cu administrarea bazei de date. Ei trebuie să urmarească cum și de către cine este folosită aceasta. Ei crează profile de acces pentru utilizatori și aplică limitări pentru a pastra securitatea. Administratorii au grijă și de resursele necesare DBMS-ului, cum ar fi licențele de sistem, uneltele necesare și alte unelte software si hardware ale acestuia.
- Design-eri - Reprezintă un grup de persoane ce lucrează la structura și implementarea bazei de date. Ei sunt atenți la ce date ar trebui păstrate și în ce format. Ei identifică și modelează toate entitățile, relațiile, constrângerile și vizualizările.
- Utilizatorii finali - Aceștia sunt utilizatorii ce folosesc beneficiile DBMS-ului. Aceștia pot fi de la simpli utilizatori ce vizualizează datele până la utilizatori complecși, cum ar fi analiști de date.

6.3 Arhitecturi DBMS

Deoarece în ceea ce privește o aplicație scalabilă, se impune stabilirea unei arhitecturi predefinită, vom alege în cadrul acestui curs o ahitectura pe mai multe nivele, argumentele fiind prezentate aici.

Arhitectura unui DBMS poate fi centralizată, descentralizată sau ierarhică, și este fie pe un nivel fie pe nivele multiple (tier sau multi-tier). O arhitectură pe n nivele împarte intregul sistem în n module relaționate, dar independente, ce pot fi modificate, schimbate, sau înlocuite independent.

În arhitectura pe un nivel, DBMS-ul este singura entitate folosită de utilizator. Orice modificari ale utilizatorului vor fi automat aplicate DBMS-ului. Aceasta arhitectură nu ofera multe unelte pentru utilizatorul final, insă este preferată pentru utilizare de către design-erii de baze de date și programatori.

Dacă arhitectura DBMS-ului este pe două nivele, atunci este nevoie de o aplicație prin care acesta să fie accesat. În acest caz, nivelul ce conține aplicația este in întregime independent față de baza de date în ceea ce privește operarea, design-ul si programarea.

Arhitectura pe 3 nivele Acest tip de arhitectura își separă nivelele în funcție de complexitatea utilizatorilor și de modul în care aceștia folosesc datele din baza de date. Este cea mai folosită arhitectură pentru modelarea DBMS-urilor.

- Nivelul bazei de date (Database Tier) - Pe acest nivel se găsește baza de date împreună cu limbajele de procesare bazate pe interogari. Tot pe acest nivel regăsim și relațiile ce definesc datele și constrangerile aplicate lor.

- Nivelul aplicatiei (Application Tier) - Pe acest nivel intalnim aplicatia server si programele ce acceseaza baza de date. Pentru un utilizator, acest nivel reprezinta o vizualizare abstracta a bazei de date. Utilizatorii finali nu sunt constienti de prezenta unei baze de date in spatele aplicatiei. Deasemenea, nici baza de date nu stie de prezenta utilizatorului final dincolo de aplicatie. Astfel, nivelul aplicatiei are rolul de mediator intre baza de date si utilizatorul final.

- Nivelul utilizator (User Tier) - Acest nivel este rezervat utilizatorului final si consta in multiple vizualizari ale bazei de date oferite de catre aplicatie. Acestea sunt generate de aplicatii aflate pe nivelul aplicatie.

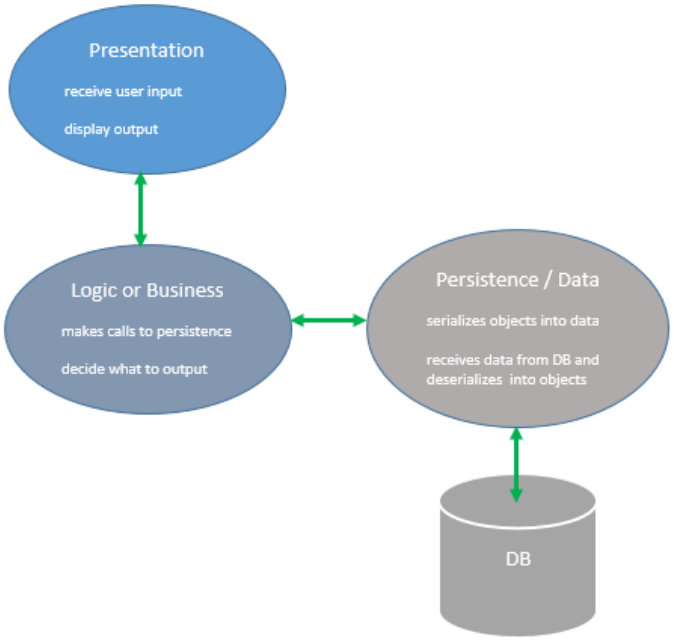

Figura 7. Arhitectură pe trei nivele

6.4 Modele de date

Modelele de date definesc modul in care este modelata structura logica a unei baze de date. Acestea sunt entitati fundamentale prin care se introduc abstractizari in DBMS. Modelele de date definesc modul in care datele sunt conectate intre ele si cum sunt procesate si retinute in sistem.

Modelul Entitate-Relatie Modelul Entitate-Relatie (ER) este bazat pe notiunea de entitati din lumea reala si relatiile dintre ele. In implementarea unui scenariu din lumea reala, modelul ER creaza seturi de entitati si de relatii, atribute generale si constrangeri.

Modelul ER este util pentru modelarea conceptuala a unei baze de date. Acesta se bazeaza pe: - Entitati si atributele lor - Relatii intre entitati

- O entitate intr-un model ER este reprezentata de o entitate din lumea reala ce are anumite proprietati numite atribute. Fiecare atribut este definit de un set de valori numit domeniu. De exemplu in baza de date a unei scoli, elevii sunt entitatile, iar atributele acestora pot fi: nume, varsta, clasa.

- O relatie reprezinta o asociere logica intre entitati. Relatiile sunt asociate entitatilor in diferite moduri: - unu la unu - unu la mai multe - mai multe la unu - mai multe la mai multe

Modelul Relational Este cel mai stiintific model pentru definirea DBMS-rilor, dar si cel mai popular. Acest model este bazat pe logica predicatului de ordin intai, si defineste un tabel ca o relatie n-ara.

Principalele avantaje ale acestui model sunt:

– Datele sunt pastrate in tabele numite relatii
– Relatiile pot fi normalizate
– In relatiile normalizate, valorile salvate sunt valori atomice.
– Fiecare coloana dintr-o relatie contine o valoare unica.
– Fiecare coloana dintr-o relatie contne valori din acelasi domeniu.

6.5 Conexiunea la un DBMS

Conexiunea la o bază de date înseamnă modul în care un server de baze de date comunică cu aplicația client. Clientul trimite o serie de comenzi serverului DB și primește raspunsuri sub diferite forme după cum se va vedea în cadrul laboratorului.

Conexiunile sunt implementate prin itermediul unor drivere sau providere iar activarea acestora se realizează printr-un string de conexiune de tipul:

```
Server=sql_box;Database=Common;User ID=uid;Pwd=password;
```

În funcție de tehnologia folosită există mai multe tipuri de provideri și ii vom enumera pe cei mai des întâlniți:

– OLE DB (ADO)
– ODBC
– JDBC
– PDO

Modul în care o conexiune trebuie menținută și mai ales cât timp poate fi aceasta deschisă, este descris în această carte.

Vom vedea în cadrul laboratorului că odată ce conexiune este deschisă vom putea executa instrucțiuni SQL și că vom putea citi, modifica date prin intermediul acestor comenzi.

Odata ce conexiunea este realizata interactiunea cu baza de date se va face prin intermediul unor comenzi SQL iar rezultatele acestora se pot stoca in cadrul unor obiecte de tip result. Mai departe aceste informatii vor fi parsate si vor popula obiecte din cadrul nivelului de persistenta.

Marea problema cu acest model de lucru este ca, odata ce structura bazei de date a fost modificata (tipul unui obiect dintr-un tabel a fost alterat de exemplu), codul care realiza parsarea trebuie modificat in concordanta cu update-ul din baza de date.

Vom vedea in cele ce urmeaza ca munca aceasta de sincronizare poate fi facuta cu ajutorul altor framework-uri.

7 ORM

ORM este un acronim ce provine din engleză de la Object/Relational Mapping. Pe scurt, un framework ORM folosește la persistența obiectelor din model într-o bază de date relațională. Pentru aceasta se folosește meta date pentru a comunica între nivelul de date și baza de date. Astfel în nivelul de persistență nu este necesar să cunoaștem structura bazi de date, ORM fiind framework ce acționează ca un intermediar între cele două.

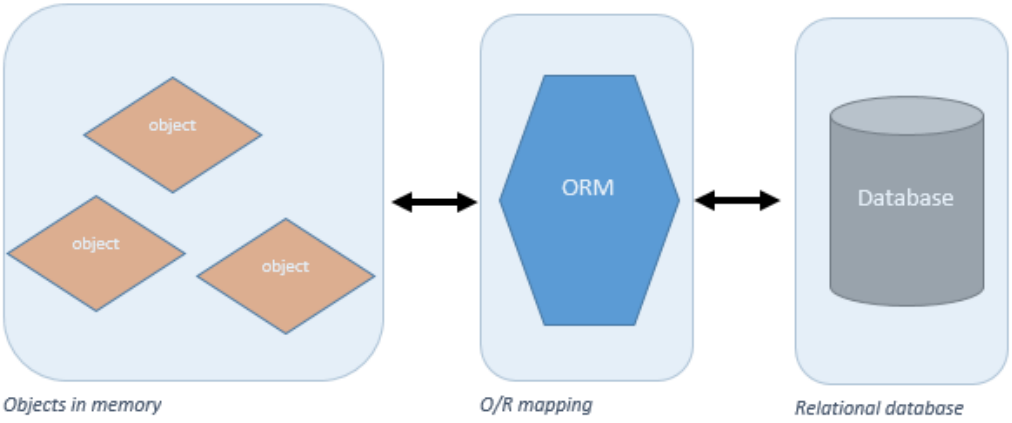

Figura 8. Arhitectura unei aplicaţii ce foloseste ORM

Nucleul unui ORM este modul de relaționare între cele două entități și anume informațiile din baza de date și datele din aplicație. Aceasta înseamnă ca un ORM gestionează cum o clasă și proprietățile ei sunt legate de una sau mai multe tabele din baza de date. Această informație este folosită de nucleul ORM pentru a construi dinamic cod SQL ce primește date și le transformă în obiecte sau ce salvează obiectele în baza de date. În general acest lucru se realizează prin intermediul un fișier XML. Unele tool-uri ORM folosesc atribute din interiorul claselor sau proprietăți ale claselor pentru a menține această mapare.

Pe lângă faptul că acest tool ușurează munca unui dezvoltator software de a trata interacțiunea cu o bază de date, un ORM tratează și coliziunile dintre obiecte și relațiile din baza de date, transformă query-urile în obiecte, corelează update-urile din baza de date cu starea obiectului și așa mai departe.

Există pe piața mai multe astfel de tool-uri, unele gratis altele comercializate. Se poate spune că Hibernate este cel mai puternic și mai stabil din acest moment. Entity Framework este de asemenea un tool de ORM foarte puternic ce rivalizează cu soluția de mai sus.

Cu toate că rezolvă multe probleme , acest tool prezintă o problemă și anume paradigma de mapare eronată. Altfel spus modelele obiectuale și modelele relaționale nu funcționează împreună. Un DBMS reprezintă data într-un format tabelar iar un limbaj OOP reprezită data ca un graf interconectat de obiecte. De aici rezultă cele 5 probleme de mai jos:

- Granularitate: uneori un model poate avea mai multe clase decât numărul de tabele corespondente în baza de date
- Moștenirea: este imposibil de implementat aceste concept într-un SGBD
- Identitatea. Într-un SGBD egalitatea dintre 2 obiecte se verifică diferit față de modul în care se face acest lucru în Java. În Java avem 2 metode de a verifica egalitatea a 2 obiecte: a==b sau a.equals(b)
- Accesul la date: in Java accesul la un obiect dintr-o colecție se face diferit față de modul de căutare într-o bază de date.

ORM-urile pot aduce valoare dar pot cauza probleme cand sunt folosite gresit (cel mai frecvent din pricina lazy-loading). Cele mai multe probleme apar cand programatorii folosesc properietăți adica folosesc un membru al clasei in loc sa apeleze metode. Deoarece aceste framework-uri au crescut in ultimii ani ca si complexitate, ele permit dezvoltatorului sa aiba control complet direct asupra elementelor GUI direct din nivelul de date. Implicit orice modificare a GUI-ului poate avea consecinte nefaste asupra datelor din baza de date. ORM este un tool foarte folositor insa va trebui utilizat cu atentie.

8 Principiile SOLID

Principiile de design software reprezintă un set de recomandări care ne ajută în a evita conceperea greșită a codului. Există trei elemente care trebuie evitate atunci când se concepe arhitectura unui produs:

1. rigiditate - daca un sistem este foarte inchis este greu de schimbat
2. imobilitate - reutilizarea unui modul este imposibilă atunci când exită o interdependență între acesta și restul modulelor
3. fragilitate - o schimbare poate aduce prăbușirea întregului sistem

De aceea atunci când se concepe/implementează un produs acesta trebuie să respecte cele 5 principii S.O.L.I.D.

8.1 Principiul responsabilității unice

Orice modul sau clasă trebuie să încapsuleze o singură funcționalitate.

O clasă sau funcție nu ar trebui să rezolve sau sa trateze mai mult decât un singur scop deoarece aceasta ar introduce cuplarea între cele două funcționalități și dorim să evităm acest lucru.

8.2 Principiul Deschis/Închis

Acest principiu exprimă faptul că o clasă sau o funcție trebuie să fie deshisă pentru extensie dar închisă pentru modificare. De exemplu, dacă avem o bibliotecă ce conține un set de clase ce trebuie extinse ar fi mai corect dacă am avea inițial un set de clase abstracte. Astfel în loc să modificăm clasele concrete din cadrul bibliotecii, putem extinde clasele abstracte. Cazuri particulare ale acestui principiu sunt menționate în șabloanele Template și Strategy.

8.3 Principiul Liskov

Principiul Liskov sau principiul substituției este următorul: ”Dacă S este subclasă a lui T atunci obiectele de tip T pot fi înlocuite cu S fără a introduce funcționalități nedorite sau erori”. Altfel spus noile clase extinse din parinte trebuie să fie capabile înlocuiască clasa de bază în toate funcțiile sale fără a fi nevoie să aducem modificări nicăieri.

8.4 Principiul Segregării Interfețelor

Acest princpiu expune ideea că niciun client nu are voie să fie forțat să depindă de metodele pe care nu le folosește

Aceasta se realizează prin ”spargerea” interfețelor în module mai mici pe care clasele au opțiunea de a le implementa sau nu. Acest mod va ajua și la ideea de decuplarea funcționalităților atunci când se doreste refactorizarea produsului.

8.5 Principiul Inversiunii Dependințelor

Acest princpiu se referă la o anumită formă de a decupla module software. Modulele de nivel înalt nu trebuie să depindă de modulele de nivel jos. Ambele module trebuie să depindă de abstractizări. Abstractizările nu trebuie să depindă de detalii ci detaliile trebuie să depindă de abstract.

8.6 Concluzie

Atunci când dependințele și organizarea sunt tratate greșit, codul poate deveni rigid și greu de folosit. Mai departe, va fi greu de modificat, de extins funcționalități sau de adaugarea de noi caracteristici. De asemenea, dacă apare o eroare aceasta se va multiplica mai ușor în cazul unui cod scris fără a urma principiile de mai sus.

9 Desgin Patterns

Design patterns reprezintă un set de practici folosit de dezvoltatorii software. Sunt soluții la probleme generale pe care le întâlnim în timpul dezvoltării de produse soft. Aceste soluții au fost obținute prin o serie de încercări și eșuări testate de-a lungul timpului de numeroși dezvoltatori.

Prima culegere de astfel de practici a fost publicată în această carte.

Există mai multe tipuri de șabloane structurate după funcționalitatea acestora.

- Șabloane de creere. Acestea permit crearea de obiecte și totodată ascunderea logicii de inițializare. Rezultă mai multă independență în deciziile de tipul: ce obiecte vor fi creeate și cu ce scop.
- Șabloane structurale. După cum spune și numele, acestea ajută la stabilirea modului în care se ierarhizează structura de moștenire și structura de compunere.
- Șabloane de comportament. Acesta definesc modul în care se comunică între obiecte.
- Șabloane de concurență. Acestea definesc modul în care se execută acțiunile în paralel

9.1 Șabloane de creere

- Factory. Definește o interfață gândită pentru a ceea un singur obiect însă clasele ce o implementează decid ce fel de obiect să instanțieze.
- Abstract Factory. Definește o interfață pentru a crea o ierarhie de obiecte fără a specifica clasele lor concrete. Practic introduce un nivel de abstractizare peste șablonul anterior.
- Singleton. Definește un mod de a impune existența unei singure instanțe a unei clase. De obicei se folosește atunci când oricare ar fi scenariul, aplicația trebuie să aibă o singură instanță a acelei clase și ea să fie activă.
- Builder. Acest șablon permite separarea unui obiect complex de reprezentarea lui astfel ca același proces să creeze reprezentări diferite ale obiectului.

9.2 Șabloane structurale

- Adapter. Scopul acestuia este de a converti o interfață la alta.
- Bridge. Decuplează clasa abstractă și clasa copil concretă păstrând o conexsiune intre ele. În acest fel ele pot fi alterate fără a afecta funcționalitatea existentă.
- Facade. Ascunde utilizatorului complexitatea implementării unei clase prin crearea unei interfețe simple.
- Proxy. Un proxy este o interfață ce este apelată pentru a accesa un alt obiect care realizează taskul dorit.

9.3 Șabloane de comportament

- Observer. Acesta se pretează atunci când ne interesează starea unui obiect și vrem să știm când se modifică aceasta.
- Strategy. Acesta este util când se definesc mai multe scenarii și aplicația decide care dintre acestea trebuie executat.
- Template Method. Acesta definește o serie de pași ce trebuie executați și totodată oferă implementare implicită ce poate fi comună pentru toate clasele ce moștenesc clasa de bază.
- Interpreter . Permite evaluarea unei expresii lingvistice sau logice.

9.4 Șabloane concurențiale

- - Active Object. Un obiect activ deține firul propriu de execuție și rulează instrucțiunile în cadrul acestui fir. Ca orice alt obiect acesta are date private și metode ce pot fi invocate de alte părți ale programului. Diferența este că atunci când o metodă este invocată din exterior, se invocă un mesaj către obiectul activ și predă apoi execuția funcției apelante. În felul acesta execuția are loc asincron și fără posibilitatea de blocare. Deoarece datele din cadrul obiectului pot fi accesate doar din cadrul firului de execuție deținut de obiect, nu este nevoie de un mutex pentru a efectua sincronizări. Aceasta permite execuția cu adevărat în paralel a codului.
- Monitor. Este mecanismul de sincronizare a mai multor fire de execuție ce necesită aceeași resursă. Aceasta se realizază prin intermediul mutex și a variabilelor condiționale.

10 Interoperabilitatea între limbaje

Interoperabilitatea între limbaje este capacitatea unor module/funcții din program de a interacționa cu alte module/funcții scrise în alt limbaj. În acest fel se poate optimiza reutilizarea codului scris dar și îmbunătățirea execuției în anumite cazuri după cum vom vedea.

Deoarece programatorii folosesc diferite tehnologii fiecare depinzând de alte biblioteci, interoperabilitatea poate fi dificil de implementat. Cu toate acestea majoritatea limbajelor permit integrarea unor funcționalități cu altele scrise în alte medii/limbaje de programare.

Modul în care se realizează integrarea depinde de mai mulți factori cum ar fi nivelul limbajului (de exemplu nivel înalt sau nativ), mediul unde rulează (integrare cross-platform, integrare distribuită în rețea). De asemenea interoperabilitatea se poate realiza atât în cadrul aceluiași proces cât și între două sau mai multe procese (prin intermediul resurselor comune, mesajelor, serviciilor web, socketi etc)

Vom trata așadar separat interoperabilitatea în cadrul aceluiași proces și interoperabilitatea între procese, în secțiuni separate.

10.1 Interoperabilitatea în cadrul aceluiași proces

Interoperabilitatea între limbaje în cadrul aceluiași proces se realizează prin intermediul unor framework-uri care acționează ca un intermediar ce permite apelul de funcții între un limbaj de nivel înalt (Java, C#) și un limbaj nativ (C, C++).

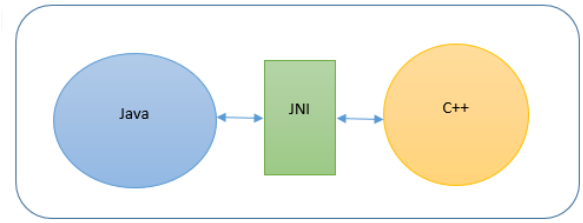

Figura 9. Java Native Interface

De exemplu între Java și C++ aceasta se poate realiza folosind JNI. Un manual ce conține aproximativ toate scenariile de folosire a acestui framework poate fi studiat aici. Evident atunci când apare necesitatea alocării memoriei pentru obiecte create în C++, sau a transferului acestor obiecte către și din Java apar următoarele întrebări: ”cum se referențiază aceste obiecte?”, ”cum se accesează?”, ”cum se dealocă?”, etc. Pentru a primi răspunsul la aceste întrebări trebuie studiat capitolul acesta.

Acest model de interoperabilitate este valabil pentru orice sistem de operare atâta timp cât codul nativ este compilat sub forma acceptată de acel sistem de operare (de exemplu .dll sau .lib în Windows, .so sau .a în Linux).

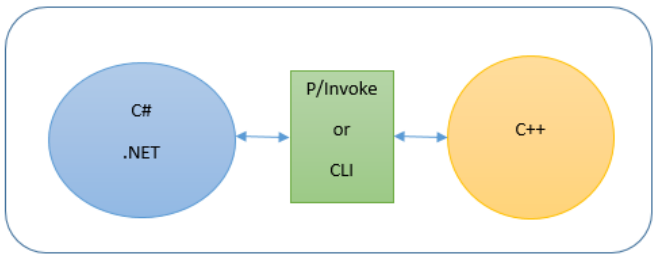

Figura 10. Intercomunicare între .NET și cod nativ

În sistemele de operare Windows există un alt model: COM, ce este mai degrabă un standard de a scrie interfețele obiectelor astfel ca acestea să fie folosite indiferent de mediul sub care au fost create.

O componentă COM oferă o implementare a unei sau mai multor interfețe definite independent de limbaj folosind IDL. Practic odată ce definim o interfață COM, fiecare componentă înregistrată va trebui să o implementeze.

Pentru a folosi o componentă înregistrată programatorul va trebui să:

– se asigure că acea componentă se află într-o locație accesibilă.
– cunoască GUID-ul componentei. Cu acest GUID clientul poate invoca instanțierea componentei folosind o funcție de tipul CoCreateInstance.

Tot referitor la sistemele Windows pentru a realiza intercomunicarea între limbaje de nivel înalt în cadrul aceluiași proces avem două modele de interoperabilitate: PInvoke și utilizarea unui limbaj intermediar C++/CLI.

PInvoke este un mecanism ce permite aplelul funcțiilor unmanaged implementate într-un DLL.

Pentru a putea translata structurile între limbaje, exemplu între C# și C++ se folosește un mecanism denumit marshalling.

Un alt model de a invoca module unmanaged în cadrul aplicațiilor .NET este folosind C++/CLI

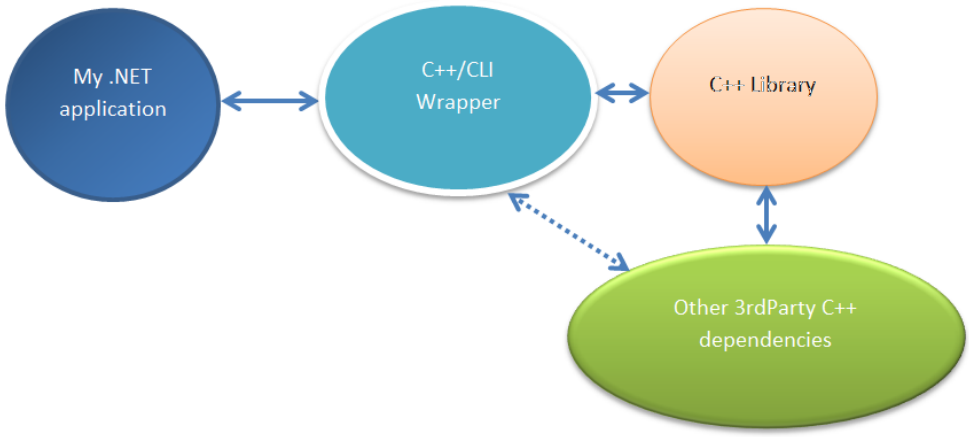

Figura 11. Modelul C++/CLI

C++/CLI este un limbaj de sine stătător cu un set nou de cuvinte, ce compilează C++. Setul nou de cuvinte este implementat pentru că această platformă suportă atât cod managed (lucrând cu structurile de bază din .NET). Totodată un proiect C++/CLI poate compila direct cod apelat din biblioteci scrise în C++. Aceste capabilități transformă proiectele scrise în C++/CLI în adevărate punți de legătură între platforma .NET și bibliotecile scrise în cod nativ, tocmai pentru că nu mai există nevoia explicită de marshalling.

10.2 Interoperabilitatea între procese

Aceasta se referă la mecanismele unui sistem de operare ce permite proceselor să partiționeze date comune.

În mod normal aplicațiile care folosesc intercomunicare între procese sunt clasificate ca aplicații client/server dar multe aplicații au rol atât de server cât și client. Avantajul oferit de acest model de lucru este că nu trebuie intervenit foarte mult asupra structurii proiectelor sau a dependințelor acestora.

În funcție de modul în care se organizează datele există mai multe modele de a realiza interoperabilitatea între procese:

– folosind fișiere. Unul sau mai multe procese vor scrie într-un fișier accesibil pentru alt/alte procese care vor citi informațiile din acesta. Sincronizarea între procese se realizează astfel: fiecare proces folosește un obiect diferit pentru a accesa resursa dar obiectele vor indica aceeași resursă comună. Se utilizează blocarea accesului la resursa comună doar la scriere.
– utilizând semnale. Se pot transmite semnale între procese active de obicei pentru a transmite comenzi între procese.
– prin socketi. Un socket este un obiect ce conține legătura de comunicare între două procese ce rulează fie pe aceeași mașină, dar de regulă în rețele diferite. Un socket este legat la un port astfel ca un strat TCP să poată identifica tipul de aplicație/destinație care va primi datele. Odată realizată legătura între doi socketi transmisia/citirea datelor se va face ca în cazul oricărui stream de date.
– cozi de mesaje. Acestea oferă un protocol de comunicare asincron, adică permite ca emițătorul sau cititorul să nu trebuiască să interacționeze simultan cu coada de mesaje.
– utilizând semafoare. Semaforul este o variabilă ce realizează sincronizarea mai multor procese pentru partajarea unei resurse comune.
– memorie partajată. Mai multe procese pot aloca o zonă comună de memorie, un buffer, pentru a partaja resurse comune.