Refactorizare utilizand sablonul de proiectare Visitor

Aritoni Ovidiu

October 20, 2010

Scopul acestui tutorial este de a prezenta sablonul de proiectare Visitor, respectiv procesul de refactorizarea proiectului privind evaluarea functiilor de o singura variabila reala. Sunt prezentate atat notiunile teoretice necesare, cat si doua studii de caz : o aplicatie simpla ilustrativa pentru sablonul de proiectare, cat si un exemplu de refactorizare. Se cere parcurgerea si implementarea celor doua exemple propuse ca studiu individual, in cadrul laboratorului cat si acasa.

Termen de predare : 10 noiembrie 2009.

1 Sablonul de proiectare Visitor

1.1 Definitie

Sablonul de proiectare Visitor raspunde cerintei formulate de Principiul Responsabilitatii Unice. Acest sablon de proiectare este utilizat in situatia in care este necesar sa efectuam una sau mai multe operatii pe elemntele unei structuri de obiect. Sablonul de proiectare permite definirea de noi operatii fara a necesita modificari asupra obiectului asupra caruia urmeaza a se efectua acele operatii. Conform acestui sablon de proiectare actiunile pe care le poate intreprinde un obiect nu vor fi incapsulate in obiectul respectiv, ci va exista un obiect special creat si dedicat actiunii pe care trebuie sa o intreprinda. Conform principiului responsabilitatii unice este necesar ca pentru fiecare responsabilitate a unui obiect A sa existe un obiect R care sa efectueze o singura operatie pe elementele structurii obiectului A. Sablonul de proiectare Visitor asigura separarea responsabilitatii de obiectul in cauza (A) , respectiv asigura unicitatea responsabilitatii, evitand in acest sens modificari in structura interna a obiectului A.

1.2 Motivatie

Sa reconsideram exemplul de mai sus al unui editor de text. Un editor de text lucreaza cu documente care sunt compuse din paragrafe, propozitii, cuvinte respectiv caractere. Toate acestea sunt unitati constituente ale unui document. O parafrazare utilizand o diagrma UML ar fi cea din figura urmatoare :

Asupra documentului respectiv sunt cerute operatii de afisare la terminal a documentului, de editare (copiere, taiere, lipire) , de tiparire, de cautare in document, etc.

Un caracter poate fi afisat la terminal, poate fi tiparit, editat respectiv poate fi cautat. In mod analog aceste responsabilitati le poate avea orice cuvant, propozitie, pargraf, respectiv intreg documentul. O noua varianta de clasa va putea arata ca in figura urmatoare :

Conform principiului responsabilitatii unice o clasa nu poate avea mai mult de o singura responsabilitate. In acest sens pentru operatia de afisare la terminal vom construi o clasa, pentru operatia de tiparire o alta clasa, pentru operatia de copiere o alta, si asa mai departe. O varianta de clasa imbunatita este urmatoarea :

Toate aceste clase - Tiparire, Afisare, Copiere, Taiere, etc - inspecteaza valoarea obiectului . Asadar sunt inspectate caractere, cuvinte, propozitii, paragrafe, documente. Operatia de inspectare a tuturor acestor unitati este ceruta de catre fiecare din clasele de mai sus : Tiparire, Afisare, Copiere, Taiere, etc. Deoarece metodele *void inspecteaza(Caracter c),* respectiv *void inspecteaza(Cuvant c), void inspecteaza(Propozitie p)* si asa mai departe apar in toate clasele - Tiparire, Afisare, Copiere, Taiere, etc - este util conform principiului dependentei inverse sa definim o clasa abstracta numita Inspector. Clasele Tiparire, Afisare, Copiere, Taiere si asa mai departe vor mosteni aceasta clasa.

Obiectele de tip Caracter, Cuvant, Propozitie, Paragraf, Document vor trebui sa poate fi inspectate utilizand oricare dintre eventualele obiecte de tip Inspector. In acest sens aceste obiecte vor trebui sa posede

1.3 Aplicabilitate

Aceste sablon de proiectare se utilizeaza cand asupra unor obiecte complexe, de tipuri diferite sau de acelasi tip este necesar a fi efectuate mai multe operatii neinrudite.

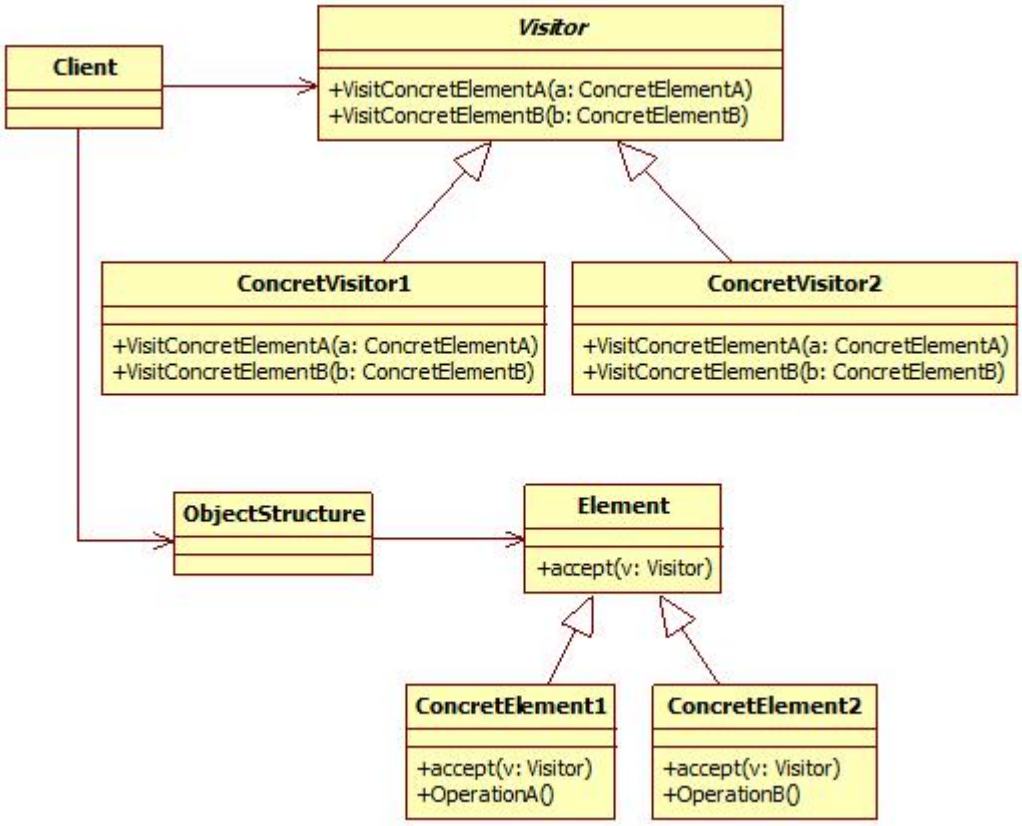
De asemenea, daca obiecte de acelasi tip sau tipuri diferite, au responsabilitati multiple este necesar sa se construiasca cate un visitor care sa realizeze sarcinile cerute de catre acele responsabilitati in mod unic.

In situatia in care o clasa implementeaza mai multe interfete, se recomanda sa se renunte la acest procedeu si sa se utilizeze sablonul visitor. De exemplu clasa M implementeaza interfetele

```
interface Q{
public T m1();
}
interface W{
public T m2();
}
interface R{
public T m3();
}
```

In aceasta situatie in clasa M vor trebui implementate cele trei metode m1, m2, m3. Pentru a evita aceasta se pot defini trei Visitori QQ,WW si RR care sa implementeze cele trei metode m1, m2, m3 in loc ca acestea sa fie implementate de catre clasa M. Operatiile m1, m2, m3 care depind de clasele lor concrete pot fi mutate cu usurinta in interiorul unui vizitor lasand o flexibilitate mai mare clasei concrete M.

1.4 Structura



Participantti:

1. Visitor

   - Declara o operatie Visit pentru fiecare clasa ConcreteElement din structura de obiecte. Numele si semnatura operatiei precizeaza clasa care trimite cererea Visit catre vizitator. Acest lucru permite vizitatorului sa determine clasa concreta a elementului vizitat. Apoi, vizitatorul poate accesa direct elementul, prin intermediul interfetei lui particulare.

2. ConcretVisitor

   - Implementeaza fiecare operatie declarata de clasa Visitor. Fiecare operatie implementeaza un fragment din algoritmul definit pentru clasa corespondenta obiectului din structura. Clasa Concrete Visitor asigura contextul algoritmului si stocheaza starea lui locala. Deseori, aceasta stare acumuleaza rezultate in timpul traversarii structurii.

3. Element

   - Defineste o operatie Accept care ia ca argument un vizitator.

4. ConcretElement

   - Implementeaza care ia ca argument un vizitator.

5. ObjectStructure

   - Isi poate numara elementele.
   - Poate furniza o interfata de nivel inalt care sa permita unui vizitator sa-si viziteze elementele.
   - Poate fi un obiect compus sau o colectie, ca de exemplu o lista sau un set.

1.5 Consecinte

Avantajele si dezavantajele acestui datorate aplicarii acestui sablon de proiectare sunt :

- Acest sablon de proiectare permite adaugarea de noi operatii cu usurinta. Adaugarea de noi operatii, ce reprezinta responsabilitati ale unor obiecte, se realizeaza prin definirea unui sablon care va incapsula modalitatile de implementare ale acelei operatii la nivelul a diferite obiecte simple. In situatia neaplicarii acestui sablon de proiectare, atunci cand se adauga o noua operatie era adeseori impusa modificarea mai multor clase.

- Un vizitator "aduna" operatii inrudite si le separa pe cele neinrudite. In acest fel comportamentele inrudite sunt reunite si localizate in cadrul unui singur obiect, si nu mai sunt imprastiate prin clasele care ajuta la definirea obiectului. Comportamentele care sunt neinrudite pot fi pastrate la nivelul obiectelor, desi conform principiului responsabilitatii unice nu se recomanda aceasta.

- Dezavantajul acestui sablon consta in faptul ca se complica lucrurile la definirea de noi tipuri ConcretElement. Acest sablon de proiectare face mult mai anevoios procesul de definire de noi subtipuri ale tipului Element. Pentru a adauga noi tipuri este necesar ca noul tip sa contina o metoda acceptVisitor, si de asemenea se cere o noua metoda abstracta in clasa Visitor care sa trateze problema inspectarii noului tip, respectiv implementarea acestei metode pentru toate clasele de tip ConcreteVisitor.

- Acest sablon permite cu usurinta inspectare unor ierarhii de clase. Iteratorii permit doar vizitarea unor structuri omgene de date, in timp ce vizitori au suport pentru tipuri de date multiple, respectiv pentru ierarhii de clase.

- Sablonul de proiectare Visitor permite acumularea starii. Aceasta acumulare ar fi posibila utilizand variabile globale sau facand apel la diferite metode cu argumente.

- Incapsularea este "sparta" de catre acest sablon de proiectare care va asigna operatiile unui obiect unor obiecte special create in acest sens. Aceste operatii fiind externe obiectului , nu mai are loc incapsularea atributelor si a starii obiectului conform principiului ascunderii informatiei.

1.6 Sabloane inrudite

Sabloanele de proiectare inrudite cu acesta sunt Composite si Interpretor.

Visitorul utilizeaza sablonul Composite in foarte multe cazuri deoarece operatia de inspectare vizeaza obiecte complexe construite de cele mai multe ori pe o structura recursiva. Codul HTML este constiuit din taguri, fiecare tag la randul sau este constituit din alte taguri si asa mai departe. Pentru modelarea unei pagini HTML se va folosi sablonul de proiectare Composite. Aceste taguri trebuiesc inspectate de catre un obiect specializat, de tipul Visitor, pentru afisarea continutului in interiroul unui browser web.

Sablonul de proiectare Interpretor are drept scop interpretarea continutului unor obiecte. In situatia unor obiecte compuse este necesara inspectarea si interpretarea tuturor subcomponentelor obiectului complex.

2 Refactorizarea prototipului software de evaluare a functiilor de o singura variabila reala

Functiile reale se pot reprezenta intern cu ajutorul unei structuri de date arborescente. Reprezentarea interna a unein functii face apel la mai multe tipuri de noduri :

- noduri terminale care reprezinta functia constanta sau functia identica.

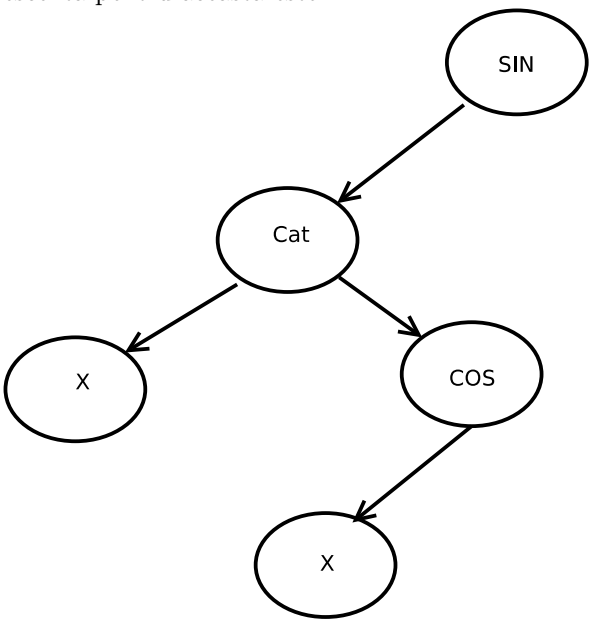
- noduri cu un singur fiu care corespund functiilor.

- noduri cu doi fii, care corespund operatorilor.

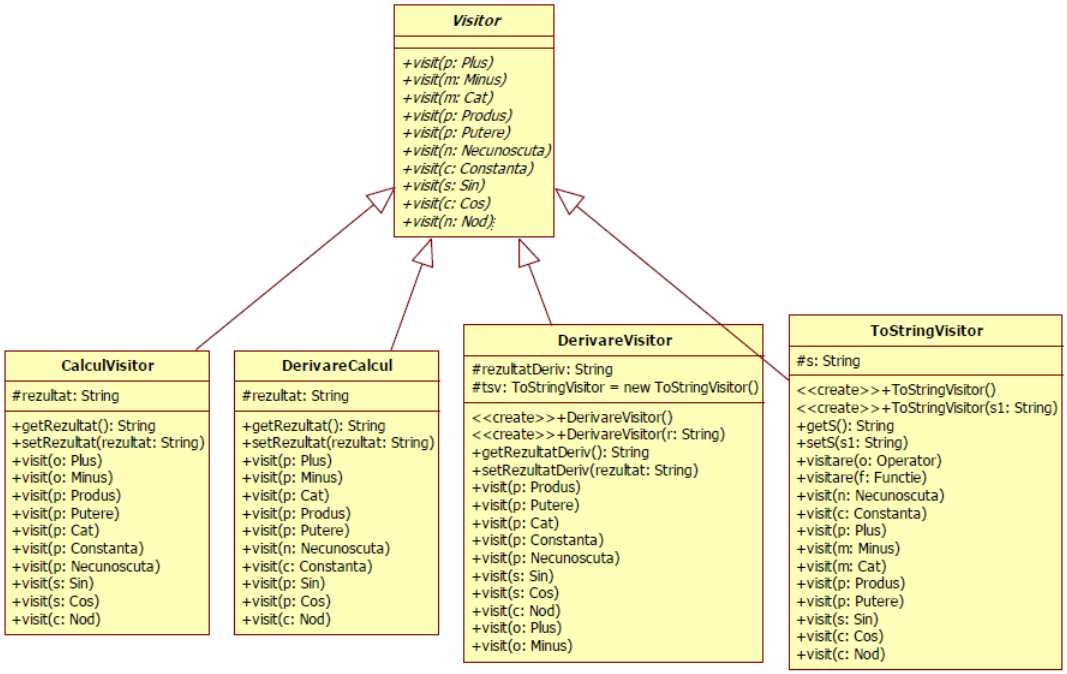
- nodul radacina. In cele mai multe cazuri nodul radacina este un operator, insa exista si cazuri in care acesta poate fi si o functie simpla sau compusa.

Operatii asupra nodurilor Asupra acestor noduri trebuie executate mai multe operatii pentru a raspunde cerintelor functionale ale aplicatiei. Operatiile cerute sunt : calculul valorii functiei intr-un punct, derivata simbolica a functiei, derivata valorica intr-un punct, respectiv ca plan de viitor integrarea simbolica si valorica. Toate aceste noduri necesita inspectarea nodurilor arborelui functional.

Sa consideram de exemplu functia $f(x) = sin(\frac{x}{cos(x)})$. Reprezentarea arborescenta pentru aceasta este :



Pentru a deriva aceasta functie, de exemplu, este nevoie de a deriva mai intai nodul sin. Nodul sin paseaza operatia de derivare nodului cat,care inca nu poate intoarce un raspuns, si apeleaza la randul sau un mesaj nodurilor X si Cos. Doar nodurile de tip frunza, adica functia identica, functia constanta si functia putere pot returna un rezultat direct la orice inspectare. Restul nodurilor necesita atat inspectarea nodurilor fii, cat si a nodului insusi. Dat fiind faptul ca operatiile de derivare simbolica, derivare valorica, calculul valorii functiei intr-un punct necesita inspectarea nodurilor arborelui, iar acest proces este recursiv se impune ca aceste operatii sa fie definite in clase separate. De altfel un obiect nu poate manifesta mai multe comportamente diferite in acelasi timp. Mai exact, de exemplu un obiect de tipul constanta nu trebuie sa aibe decat un singur comportament, iar in cazul arhitecturii propuse in varianta 1.0 obiectul constanta putea fi derivat simbolic/ valoric, putea fi evaluat si asa mai departe.



Asadar conform principiului responsabilitatii unice se cere ca pentru fiecare responsabilitate a obiectelor de tip nod sa se creeze o clasa care sa incapsuleze comprtamentul dedicat indeplinirii acelei responsabilitati. Astfel se vor crea urmatorele clase destinate a satisface responsabilitatile corespunzatoare : ToStringVisitor, DerivareVisitor, CalculVisitor, CalculDerivataVisitor.

1. Creati un nou proiect denumit Functions 1.1 .
2. Importati din proiectul Functions 1.0 toate pachetele si toate clasele.
3. Creati un pachet *visitor* si in interiorul acestuia o clasa abstracta Visitor. Din aceasta clasa abstracta vor fi derivate celelalte clase de tip visitor. Aceasta clasa va trebui sa fuznizeze metode de inspectare pentru toate tipurile de noduri. Metodele de inspectare vor fi abstracte si vor supraincarca acelasi nume *visit*. Metodele de inspectare nu returneaza nimic, rezultatul inspectiei va fi transmis cu ajutorul unui atribut al clasei.

```
1  package Visitor;
2  import Operatori.*;
3  import Functii.*;
4  import Arbori.Nod;
5
6  public abstract class Visitor {
7
8      public abstract void visit(Plus p);
9      public abstract void visit(Minus m);
```

```
10      public abstract void visit(Cat m);
11      public abstract void visit(Produs p);
12      public abstract void visit(Necunoscuta n);
13      public abstract void visit(Constanta c);
14      public abstract void visit(Sin s);
15      public abstract void visit(Cos c);
16
17      public abstract void visit(Nod n);
18
19  }
```

4. Pentru aplicarea sablonului de proiectare Visitor asupra prototipului in cauza sunt necesare mai multe modificari.

a) Clasa Nod din pachetul arbori nu va contine decat cele trei atribute - nodul fiu stang, nodul fiu drept, valoarea lui x- respectiv metodele getxxx si setxxx. In aceste conditii, in care metodele de derivare simbolica / valorica, calcul si toString, au fost eliminate se impune ca si clasa Nod sa nu mai fie o clasa abstracta. In interiorul clasei Nod trebuie adaugata o metoda denumita acceptVisitor care are ca parametru un obiect de tip Visitor si nu returneaza nimic. A accepta un visitor inseamna de fapt a apela o metoda a visitorului pentru vizitarea nodului curent. In acest scop a fost cosntruita metoda visit(Nod n). Clasa de tip abstract Visitor trebuie sa furnizeze metode de inspectare pentru obiectele concrete cat si pentru cele mai generale, care constituie generalizari ale obiectelor concrete.

```
.....
public void acceptVisitor(Visitor v){
v.visit(this);
}
```

b) In mod corespunzator se va modifica si clasa *ArboreFunctional*, avand un singur atribut numit *radacina* de tip *Nod*, un constructor explicit, metodele get si set pentru atributul radacina, si o metoda *void acceptVisitor(Visitor v)* care va apela metoda visit a parametrului v asupra nodului radacina.

c) Clasa Operator va suferi si ea modificari. Lipsind operatiile de derivare simbolica / valorica, calcul, toString aceasta clasa nu va contine decat metoda

```
public abstract String concatTermeni(String aux1,String aux2);
```

Deci se impune ca aceasta clasa sa fie o clasa abstracta.

d) Clasele Plus, Minus, Produs si Cat se vor limita la o implementare a metodei concatTermeni precum si la o varianta statica a acestei metode. Este oferit mai jos drept exemplu clasa Plus, restul claselor fiind similare, cu adaptarile necesare impuse.

```
1  package Operatori;
2  import Arbori.Nod;
3  public class Plus extends Operator{
4
```

```
5           public String concatTermeni(String aux1, String aux2){
6
7       String aux;
8       if (aux1.compareTo("0") == 0 && aux2.compareTo("0") == 0)
9           aux = "0";
10      else if (aux1.compareTo("0") == 0) aux = aux2;
11      else if (aux2.compareTo("0") == 0 ) aux = aux1;
12      else
13         aux = aux1 + "+" + aux2;
14      return aux;
15
16      }
17
18          public static String concatTermens(String aux1, String aux2){
19
20      String aux;
21      if (aux1.compareTo("0") == 0 && aux2.compareTo("0") == 0)
22          aux = "0";
23      else if (aux1.compareTo("0") == 0) aux = aux2;
24      else if (aux2.compareTo("0") == 0 ) aux = aux1;
25      else
26         aux = aux1 + "+" + aux2;
27      return aux;
28
29      }
30
31  }
```

Aceste metode de concatenare vor fi folosite adesea in cadrul proceselor de afisare a functiei (toString) sau de derivare simbolica.

e) Deoarece si functiile in procesul de derivare simbolica sau de afisare fac apel la concatenarea de siruri de caractere, pornind de la sirul obtinut de la fiul stang in varianta curenta de lucru se impune construirea unei metode denumite *concatTermeni* abstracte in interiorul clasei *Functie*. Deci clasa *Functie* va fi urmatoarea :

```
1  package Functii;
2  import Arbori.Nod;
3  public abstract class Functie extends Nod{
4          public abstract String concatTermeni(String aux);
5  }
```

f) Deoarece obiectele de tip Constanta, Necunoscuta si Putere constituie noduri frunza ale arborelui functional si pot sa returneze in mod direct si fara sa faca apel la alte noduri diferite rezultate cerute de catre client, aceste functii nu vor avea metoda concatTermeni, deci se poate concluziona ca nu este nici necesar sa mostenească clasa Functie. Aceste clase vor mosteni in mod direct clasa Nod. Mai jos este prezentat codul sursa pentru clasele Constanta si Necunoscuta.

```
1  package Functii;
2  import Arbori.Nod;
3  public class Constanta extends Nod{
4
5      protected double ct;
6
7          public Constanta(double ct) {
8         super();
9         this.ct = ct;
10     }
11
12         public double getCt() {
13        return ct;
14     }
15         public void setCt(double ct) {
16        this.ct = ct;
17     }
18
19 }
```

```
1  package Functii;
2  import Arbori.Nod;
3  public class Necunoscuta extends Nod{
4  }
```

g) Pentru clasele care mostenesc clasa Functie - Sin, Cos, Tg, Ctg, Ln, etc- se cere doar implementarea metodei concatTermeni care poate fi preluata din clasele corespunzatoare ale proiectului din varianta 1.0 . Cu scop ilustrativ este prezentat codul sursa al clasei Sin :

```
1  package Functii;
2  public class Sin extends Functie{
3
4          public String concatTermeni(String aux1) {
5
6      String aux;
7      int nrP;
8      nrP = aux1.charAt(0) == '(' ? 0 : 2;
9      aux = "sin";
10     if (nrP != 0) aux += "(";
11     aux += aux1;
12     if (nrP != 0) aux += ")";
13     return aux;
14
15     }
16
17    public static String concatTermens(String aux1) {
```

```
18
19      String aux;
20      int nrP;
21      nrP = aux1.charAt(0) == '(' ? 0 : 2;
22      aux = "sin";
23      if (nrP != 0) aux += "(";
24      aux += aux1;
25      if (nrP != 0) aux += ")";
26      return aux;
27
28      }
29
30  }
```

5. Cel mai simplu visitor care poate fi construit este cel care furnizeaza expresia sub forma de sir de caractere a functiei. In acest scop se va construi o clasa *ToStringVisitor* care mosteneste clasa Visitor. Pentru a colecta sirul de caractere se va folosi un atribut s de tipul String. In mod uzual clasa ToStringVisitor va avea metode get si set pentru atributul s, un constructor implicit care va seta sirul s pe valoarea sirului vid "", respectiv un constructor explicit. Pentru a vizita operatorul plus este nevoie sa ne asiguram ca cele doua noduri fiu fStang, fDrept exista. Daca aceasta cerinta este satisfacuta atunci se va vizita nodul stang, se va depune rezultatul vizitei intr-o variabila s1, apoi se va vizita nodul drept, iar rezultatul vizitei acestui nod se va depune in variabila s2. Ulterior se va apela metoda concatTermeni pentru parametrii s1 si s2. Pentru un obiect de tipul Plus vizitarea se va face cu ajutorul metodei expuse mai jos :

```
1  public void visitare(Plus p){
2   if(p.getFStang()!=null && p.getFDrept()!=null)
3    {
4    visit(p.getFStang());
5    String s1=this.s;
6    visit(p.getFDrept());
7    String s2=this.s;
8    s=p.concatTermeni(s1, s2);
9    }
10 }
```

In mod absolut identic se construiesc metode pentru nodurile Minus, Produs, Cat.

Metoda care inspecteaza nodul Necunoscuta trebuie sa atribuie atributului s al clasei sirul de caractere "x". Metoda care inspecteaza nodul constanta va atribui lui s valoarea constantei sub forma de sir de caractere.

```
1  public void visit(Necunoscuta n){
2    this.s="x";
3  }
4  public void visit(Constanta c){
```

```
5  this.s=Double.toString(c.getCt());
6  }
```

Pentru a vizita orice sir functie este necesar a testa daca exista nodul stang al acesteia, iar in caz afirmativ se impune vizitarea nodului stang, depunerea rezultatului vizitei intr-o variabila temporara s1, ulterior apelandu-se metoda conacatTermeni(s1) a obiectului functie respectiv, rezultatul acestui apel fiind depus in atributul clasei s.

Deoarece comportamentul pentru operatori este identic se va construi o metoda visitare(Operator o) care va fi apelata de catre metodele destinate a vizita operatori. Analog, pentru functii se va construi o singura metoda visitare(Functie f) ce va fi apelata de catre visit(Sin s), visit (Cos c), si asa mai departe.

La primul apel al metodei visit se apeleaza metoda visit(Nod n). Deoarece nici un obiect din arbore nu este o instanta a clasei Nod aceasta metoda se va ocupa cu plasarea responsabilitatii nodului competent. Asadar aceasta metoda va testa daca obiectul transmis drept parametru este de tipul Plus si va apela apoi metoda visit(Plus p), de tipul Sin si va apela metoda visit(Sin s), si asa mai departe.

6. Un alt vizitor este cel destinat derivarii simbolice, denumite DerivareVisitor. Acesta va avea un atribut rezultat de tipul String in care se va depune valoarea obtinuta in procesul de derivare simbolica. In mod corespunzator exista metodele get si set pentru acest atribut, un constructor implicit, si unul explicit. Deoarece nodurile Plus si Minus prin derivare devin tot Plus respectiv Minus, metodele de vizitare sunt identice cu metodele de vizitare ale vizitorului ToStringVisitor. Vizitarea nodului Produs pune unele probleme datorate formulei de derivare. Pentru vizitarea nodului Produs se cere vizitarea nodurilor fStang, fDrept de catre un vizitor de tipul ToStringVizitor si de catre un vizitor de tipul DerivareVisitor. Vizitarea de catre vizitorul ToStringVizitor este ceruta de faptul ca in formula de derivare $(f \cdot g)' = f' \cdot g + f \cdot g'$, se cere afisarea sirului de caractere pentru f', g', f si g, adica vizitarea cu ajutorul lui ToStringVizitor a lui f si g. Pentru a efectua aceste vizite este necesar sa se construiasca un obiect de tipul ToStringVizitor care sa fie acceptat de catre f si g, adica de catre fStang respectiv fDrept. Pentru a nu construi foarte multe obiecte de tipul ToStringVizitor si a incarca inutil memoria masinii virtuale Java, se va construi un singur obiect ca atribut al clasei DerivareVisitor.

Mai jos este prezentat codul sursa pentru metoda visit(Produs p) a clasei DerivareVisitor:

```
public void visit(Produs p){
if(p.getFStang()!=null && p.getFDrept()!=null)
{
visit(p.getFStang());
String f1D=this.rezultat;
visit(p.getFDrept());
String f2D=this.rezultat;
p.getFStang().acceptVisitor(tsv);
```

```
String f1=tsv.getS();
p.getFDrept().acceptVisitor(tsv);
String f2=tsv.getS();
this.rezultat=Plus.concatTermens(Produs.concatTermens(f1D,f2),
                                 Produs.concatTermens(f1,f2D));
}
}
```

In mod similar se va construi si metoda de vizitare a nodului Cat.

Vizitarea nodului Constanta va atribui atributului *rezultat* al obiectului de tip DerivareVisitor valoarea sirului de caractere "0". Vizitarea nodului Necunoscuta va atrinui atributului *rezultat* valoarea sirului de caractere "1".

Vizitarea unui nod de tip Functie presupune testarea existentei nodului fiu, iar in caz afirmativ se va face vizita nodului fStang, rezultatul acestei vizite de va depune in variabila de tip String fsd, dupa care se va vizita cu ajutorul vizitorului ToStringVizitor nodul fStang, rezultatul acestei vizitari fiind depus in variabila de tip String fs. Atributul rezultat se va seta cu ajutorul metodei de concatenare statice a clasei Produs care va avea ca parametrii :

- rezultatul metodei statice concatTermens(fs) a obiectului functie curent.
- fsd

Corpul metodei visit(Sin s) este ilustrat mai jos. Pornind de la acest exemplu se vor construi metodele de derivare simbolica ale vizitorului DerivareVisitor pentru celelalte obiecte de tipul Functie.

```
public void visit(Sin s){
 if(s.getFStang()!=null){
    visit(s.getFStang());
    String fsd=this.rezultat;
    s.getFStang().acceptVisitor(tsv);
    String fs=tsv.getS();
    this.rezultat=Produs.concatTermens(Cos.concatTermens(fs),fsd);
  }
}
```

In mod asemanator se vor construi vizitorii pentru calculul valorii functiei pentru un argument real, respectiv calculul derivatei functiei pentru un argument real.

3 Tema

Termen de predare :

- Implementarea in Java a noului design arhitectural asupra prototipului de evaluare a functiilor de o singura variabila reala, propus la sectiunea 3. Implementara va avea in vedere toate obiectele de tip Functie - ln, log,

exp, sqrt, pow, etc - si de asemenea va fi construit si un visitor care sa permita derivarea valorica.

- Cititi si conspectati “Variations on Visitor Design Patterns” . Pentru descarcare vezi aici