Programare Orientata pe Obiecte

Laboratorul 12

Rotaru Alexandru Andrei

Facultatea de Automatică şi Calculatoare

Universitatea Politehnica din Bucureşti

Anul universitar 2015 - 2016

Seria CC

1 Design Patterns

1.1 Introducere

Ce este un Design Pattern ?

Pentru a putea intelege mai bine conceptul de design pattern putem face referire la conceptul de clasa. O clasa reprezinta o abstractizare peste tipuri de date, anume desi obiectele instantiabile pot avea o gama larga de valori in functie de de tipurile de date din care sunt compuse, acestea au in esenta aceleasi proprietati. De exemplu pentru o clasa Student cu un membru String nume si un membru Integer, numarul de moduri in care o putem instantia este egal cu numarul de stringuri posbile inmultite cu numarul de intregi posibili care practic sunt infinit. Desi aveam posbilitatea de a da un numar infinit de instantieri diferite pentru o clasa esenta ramane aceeasi si anume: un string si un intreg. Cu alte cuvinte abstractizarea acelui numar infinit de obiecte posibile este data de o clasa ce contine un String si un Integer. Folosindu-ne de conceptul de abstractizare reamintit anterior putem sa dam o definitie decenta a unui Design Pattern. Un design pattern este o abstractizare peste modul in care clasele sunt create, sunt structurate sau interactioneaza cu alte clase.

Pentru a intelege mai bine acest concept putem porni de la urmatoarele 2 exemple.

Exemplul1: Sa presupunem ca vrem sa modelam modul in care un brutar reuseste sa-si vanda produsele anumitor companii care pot sa le revanda sau sa le ofere angajatilor. In primul rand o sa avem un obiect pe care o sa-l numim brutar care produce obiecte de tip prajituri dar nu in mod continuu ci in reprize. In functie de anumiti factori externi (oboseala, resurse) numarul de prajituri poate sa varieze de la repriza la repriza si nici reprizele nu au durata constanta, uneori brutarul face 40 de produse intr-o rerpriza de 30 de minute, alte ori ii ia 50 de minute sa faca 25 de produse.

Din perspectiva unui consumator apar urmatoarele probleme: venirea cand brutarul tocmai si-a vandut toate prajiturile pe care le avea sau venirea cand brutarul nu are produse cat are consumatorul nevoie etc. La prima vedere, ar putea fi o solutie ca la anumite intervale de timp sa vina consumatorul si sa verifice daca brutarul are suficiente prajituri si in caz afirmativ, sa le cumpere. Totusi aceasta nu se garanteaza eficienta, un consumator se poate sincroniza prost si sa vina de fiecare data cand brutarul termina de vandut prajiturile produse pe rerpriza curenta. Este evident ca strategia consumatorilor de a verifica din cand in cand daca brutarul are produsele este una destul de slaba. Pentru a nu consuma timpul, energia si nervii consumatorilor, venim cu o solute mai desteapta. Fiecare client sa-si treaca numarul de telefon pe o lista iar brutarul ori de cate ori are prajituri sa le trimita mesaje la toti ca pot veni sa si le cumpere.

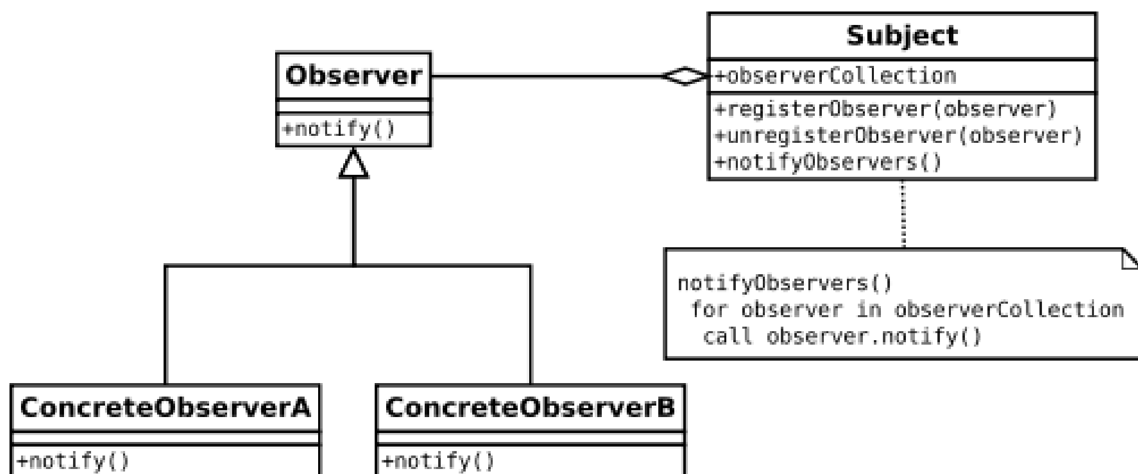
Exemplul 2: Pornim de la o componenta grafica swing si anume un buton. Cum putem sa tratam evenimentele care pot aparea in cadrul acestuia (de exemplu un click sau mouse

hover)? O solutie la prima vedere ar fi sa avem un obiect intr-un thread separat care sa verifice la fiecare T interval de timp daca s-a dat click sau s-a declansat vreun eveniment pe acel buton. In functie de T, daca acest interval este foarte scurt si butonul se apasa rar sau deloc avem multa procesare in van, daca acest interval este lung, butonul si implicit toata aplicatia o sa dea impresia ca functioneaza lent. O solutie mai buna ar fi insa sa punem intr-o lista undeva in cadrul componentei grafice toate aceste obiecte care asteapta un evenimente, si cand evenimentul apare, toate aceste obiecte sa fie "apelate".

Observatie

Practic asta ati facut pana acum cu acele obiecte ActionListener pe care le-ati extins in cadrul laboratoarelor.

Desi la prima vedere problemele enuntate mai sus n-au nicio legatura intre ele, in esenta sunt aceasi: In cadrul unui obiect la anumite intervale de timp sau anumite contexte pot aparea evenimente pe care alte obiecte asteapta sa le proceseze.



Mapand pe diagrama cele 2 exemple: In primul caz Subject este brutarul iar Observes sunt consumatorii, in cel de-al doilea caz Subject este componenta grafica iar Observers sunt Action-Listeners. In momentul in care se produce evenimentul asteptat de observatori acestia sunt anuntati folosind o metoda de notify.

Acum ca am vazut ce este un Design Pattern putem sa ne intrebam:De ce sa le folosim? Nu ar fi mai simplu ca fiecare programtor sa scrie cod dupa cum considera atata timp cat aplicatia functioneaza corect? Raspunsul desi poate parea contra intuitiv, este NU. In lumea reala aplicatiile nu sunt scrise in maniera in care se scriu temele de programare: o data scrise sa

treaca testele, sa se ia punctajul aferent si apoi sa fie lasata pe "vecie" pe hard-disk.

Aplicatiile ce au ca utilizatori nu doar programatorii care le scriu nu pot fi scrise intr-o maniera close-ended. Constant aceastea vor trebui inbunatatite, imbogatite cu noi functionalitati pentru a putea face fata pe piata sau pentru a satisface nevoile clientilor, care sunt dinamice. Nu ne putem permite sa scriem cod caruia daca vrem sa-i adaugam un feature dupa 6 luni va trebui sa ne fortam sa-l intelegem. Design Pattern-urile sunt o solutie eleganta la aceste probleme si asigura simplitate, mentenabilitate si extensibilitate.

Simplitatea se refera la efortul de a intelege si a comunica niste idei. Este usor de recunoscut un pattern ce apare intr-un cod scris de altcineva sau de o instanta a voastra din trecut. De asemenea sunt foarte usor de comunicat, din primul exemplu din introducere observam instant ca este mult mai simplu sa spui ca brutarul este subiectul si consumatorii sunt observatorii in cadrul patternului Observer decat sa explicam in detaliu fiecare clasa si obiect impreuna cu rolul acestora.

Mentenabilitate se refera la usurinta cu care un cod bine scris poate fi depanat. Toate pattern-urile vin cu o separare destul de intuitiva si logica a problemelor in obiecte astfel incat izolarea si rezolvarea bug-urilor devine o opeartie mult mai putin costisitaore decat in cazul unei aplicatii in care absolut totul este un sir foarte lung de apeluri imbricate de functii. De exemplu observati ca la modelarea brutariei, un client nu primeste niciodata produse, problema cea mai probabila este neabonarea clientului.

Extensibilitatea se refera la posbilitatea de a adauga o noua functionalitate fara a modifica cod scris deja. Acest lucru este una dintre cele mai importante avantaje ale limbajelor ce suporta Orientarea spre Obiecte. De ce este atat de important sa nu modificat cod? De fiecare data cand modifcati o clasa chiar si minimal adaugati LATENTA la procesul industrial de writecode->test->debug->release prin timpul de retestare a intregului cod pentru asigurarea functionalitatii si tolerantei la erori, prin timpul de compilare: de fiecare data clasa modificata va trebui recompilata, prin timp de depanare care creste o data ce devine incomod de citit codul cu probleme.

Pe langa toate astea pattern-urile ofera un avantaj major si anume stabilitatea. In decurs de 20 de ani de cand au fost documentate prima data, aplicatiile ce au aplicat principiile si au reusit sa reziste, demonstrand puterea acestor concepte. De asemenea un alt avantaj este reutilizarea codulu. Puteti sa folositi cod deja scris pentru a implementa ceva nou.

1.2 Categorii de Design Patterns

Design Patterns se impart in 3 categorii in functie de aspectul din cadrul unui obiect pe care il trateaza, astfel enumarm:

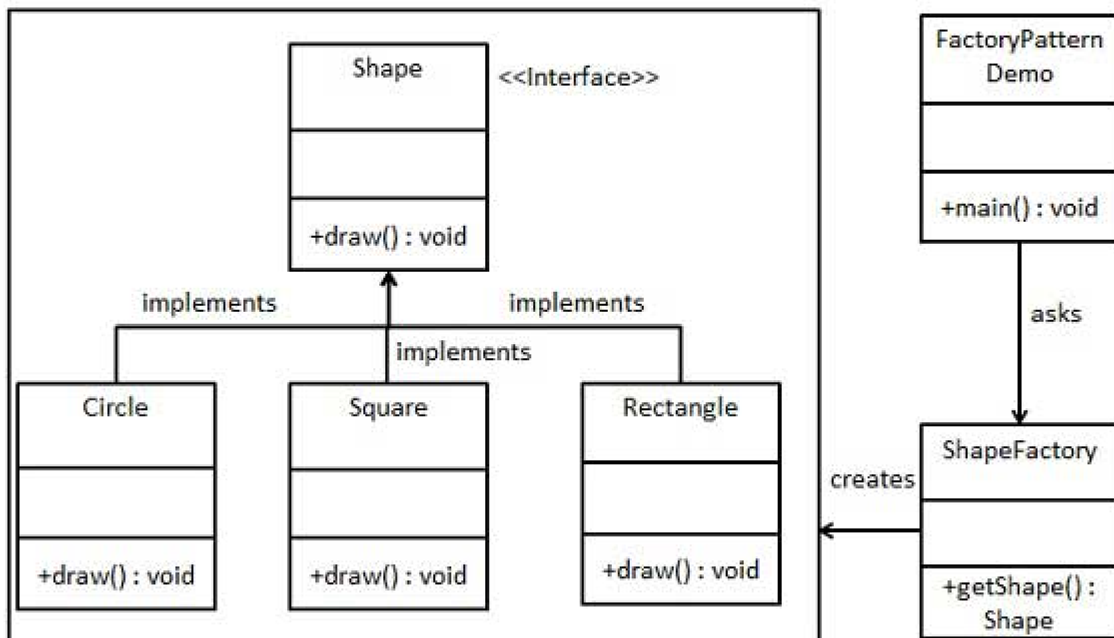
- Creational
- Structural
- Behavioral

1.3 Creational Design Patterns

Acestea se ocupa cu probleme legate de modul in care se creeaza obiectele.

1.3.1 Factory Method

De multe ori intr-o aplicatie vom avea nevoie sa instantiam obiecte la runtime in functie de anumite evenimente sau chiar in functie de inputul utilizatorului si pentru aceasta folosim Factory Method. Ori de cate ori trebuie sa cream obiecte in functie de alte variabile, o centralizare si compactare a logicii nu face decat sa clarifice codul. Sa presupunem ca avem urmatoarea ierarhie de clase:



Clasa factory arata in felul urmator:

Cod sursa Java

```java
public class ShapeFactory {
   //use getShape method to get object of type shape
    public Shape getShape(String shapeType){
        if(shapeType == null){
            return null;
        }
        if(shapeType.equalsIgnoreCase("CIRCLE")){
            return new Circle();
        } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
            return new Rectangle();
        } else if(shapeType.equalsIgnoreCase("SQUARE")){
            return new Square();
        }
        return null;
    }
}
```

Dupa cum se observa, in functie de string-ul dat ca parametru se creaza obiecte de tip Shape.

Un exemplu foarte bun de folosire al Factory ar fi incarcarea unei configuratii intr-o aplicatie la pornirea acesteia. In functie de preferintele utilizatorului salvate intr-un fisier de configurare, o clasa factory va instantia in sistem doar obiectele ce detin functionalitatile dorite de utilizator.

1.3.2 Singleton

In anumite situatii, intr-o aplicatie va trebui sa asigurati sau mai bine spus impuneti ca o anumita clasa sa se poate instantia doar o data in sistem. In viata reala exista un lung sir de exemple posibile pentru a evidentia Singleton. De exemplu presedintele unei tari , regele unei tari, seful unei firme, liderul unui grup, sunt toti singleton deoarece nu are sens sa poata exista mai multi in cadrul unui context. Pentru a asigura acest lucru pogramatic, se impune ca metoda Constructor sa fie privata. In conditiile aceastea singurul loc unde se poate apela constructorul este interiorul clasei. Deci pentru a putea avea totusi o instanta a clasei, declaram o variabila membru statica de acelasi tip cu clasa insasi pe care o instantiem la inceput. Aici mai apare o problema, existand doua variante de instantiere. O instantiere la inceputul clasei este thread-safe dar nu este utila in cazul in care obiectul singleton nu este folosit niciodata. O alta varianta ar fi instantiera la prima cerere din exterior pentru o instanta a clasei Singleton, aceasta varianta nefiind thread-safe dar asigurand economie de resurse. Nu o sa discutam varianta buna din ambele puncte de vedere

deoarece depaseste programa acestui curs, dar e important sa retineti ca pot aparea probleme legate de sincronizare intrun mediu multi-threading cu cea de-a doua varianta.

Cod sursa Java

```java
public class SingleObject {

    //create an object of SingleObject
    private static SingleObject instance = new SingleObject();

   //make the constructor private so that this class cannot be
   //instantiated
    private SingleObject(){}

    //Get the only object available
     public static SingleObject getInstance(){
        return instance;
    }

    public void showMessage(){
        System.out.println("Hello World!");
    }
}
```

O data cu o clasa Singleton apare un avantaj major si anume usurinta de accesare de oriunde in sistem. Nu mai aveti nevoie sa instantiati sau sa transmiteti prin parametru, pur si simplu apelati metodele de care aveti nevoie ca in exemplul urmator:

Undeva intr-o metoda indepartata

```java
SingleObject.getInstance().showMessage();
```

SINGURUL motiv pentru care este "permisa" crearea unui singleton este UNICITATEA acelui lucru in sistem. Daca prin natura obiectului acesta este unic, atunci programatic o sa-l descrieti prin singleton.

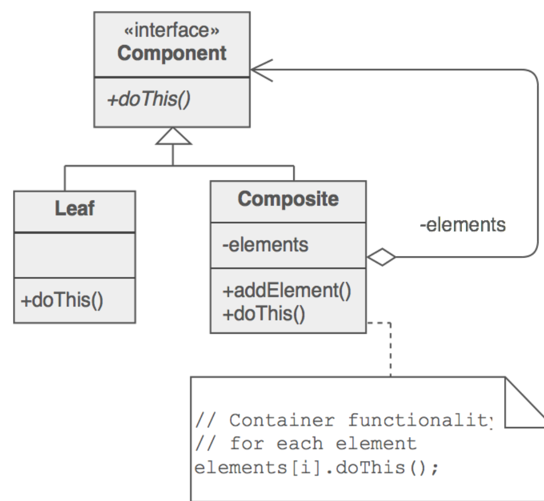
Atentie totusi, clasele Singleton actioneaza ca variabilele statice din C, daca puteti modifica starea singletonului pot aparea mari probleme in procesul de Debug.

1.4 Structural Design Patterns

Acestea trateaza problemele legate de modul in care un obiect este reprezentat.

1.4.1 Composite

In anumite cazuri o sa aveti nevoie in applicatiile voastre de anumite ierarhii arborescente. De exemplu aveti nevoia reprezentarii subordinii intr-o aramata, fiecare general putand avea in subordine soldati sau alti generali cu grad inferior. In esenta o sa implementati un arbore generic in care tipurile nodurilor variaza dinamic la runtime.



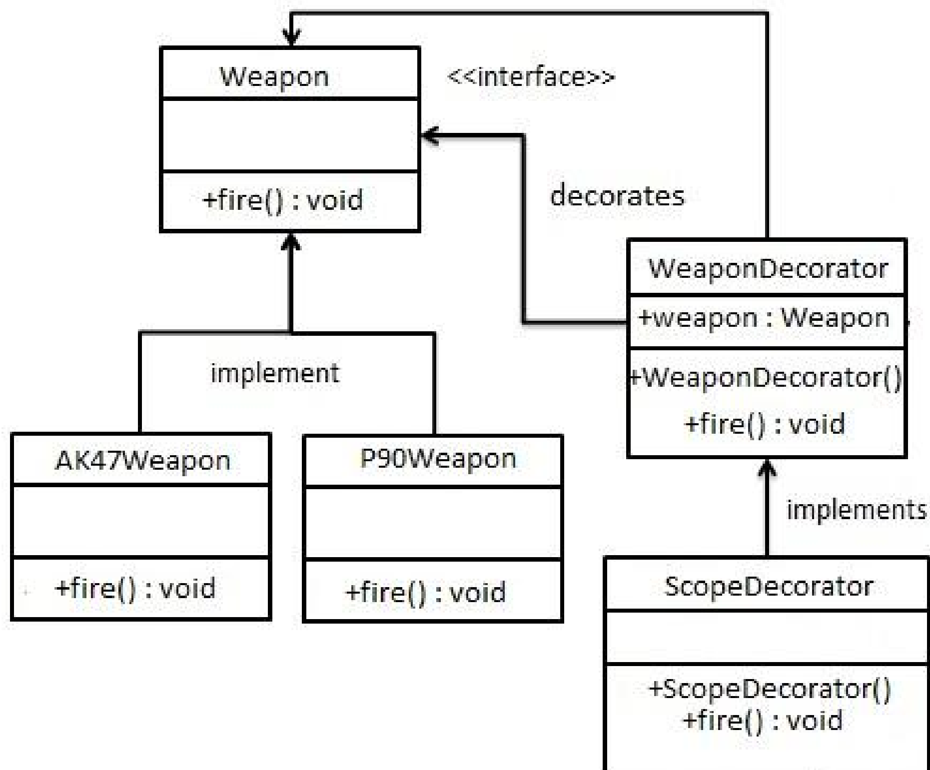
Mapand pe diagrama exemplul de mai sus: generalii sunt nodurile Composite deoarece pot avea in subordine alti generali sau soldati, in schimb soldatii vor fi de tip Leaf doarece ei nu pot avea pe nimeni in subordine.

In diagrama prezentata sunt folosite doar 2 tipuri de noduri, asta nu inseamna ca nu exista posibilitatea adaugarii si altor tipuri de noduri, din contra, aceste clase pot fi derivate intr-o ierarhie complexa si in continuare respecta structura de Composite.

1.4.2 Decorator

De multe ori in cadrul unei aplicatii apare urmatorul scenariu. Aveti de implementat o anumita functionalitate care trebuie sa poate fi extinsa fara sa stiti in momentul proiectarii in ce mod va fi extinsa. De exemplu lucrati la un joc de tipul shooter. Aveti o lista de arme pe care un user le poate achizitiona si utiliza pentru a-si impusca oponentii. La un moment dat observati ca incasarile din joc incep sa scada o data ce alternativele de pe piata ofera mai multe feature-uri pentru armele lor. Pentru a nu da faliment va trebui sa adaugati niste feature-uri noi armelor

voastre, dar trebui sa tineti cont ca orice modificare de cod creste fragilitatea si sansa de aparitie a unor bug-uri care initial nu existau. Rezolvarea este foarte simpla: O sa creati o clasa abstracta wrapper peste o instanta de arma. Daca de exemplu vreti ca toate armele voastra sa poata fi mortale dar silentioase, creati clasa WeaponSilencer ce implementeaza interfata de baza a armelor si are un obiect de tip arma intern. In metoda fire, in decorator, o sa setati eventual sunetul jocului mai incet.

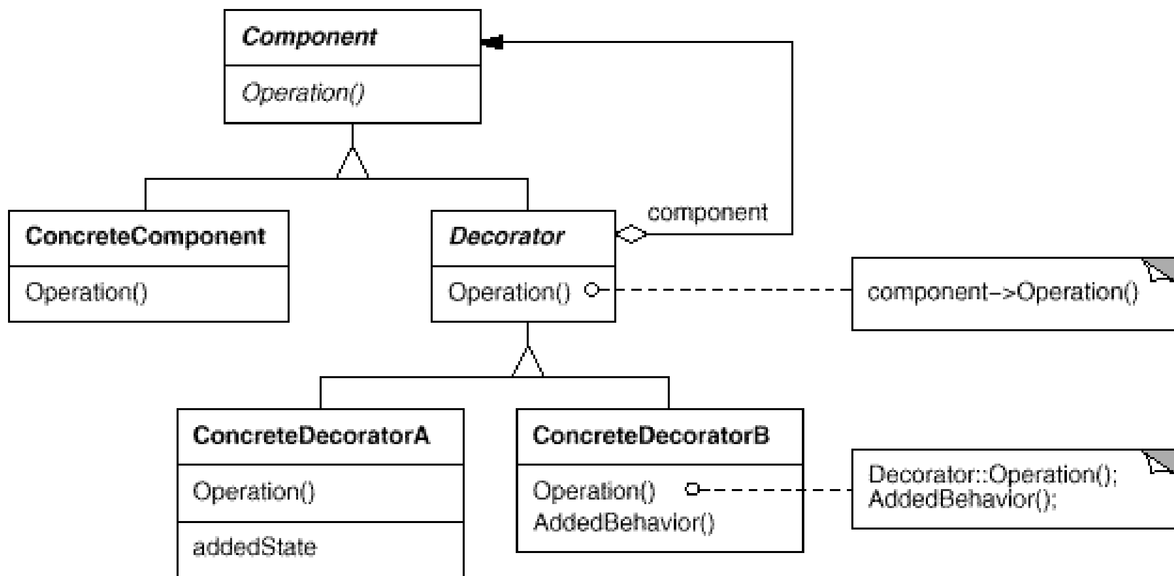


Un alt exemplu la indemana sunt fluxurile din java.

Meow

```
// some long mad code, c is declared
CipherOutputStream cos = new CipherOutputStream(new FileOutputStream("file"), c);
PrintWriter pw = new PrintWriter(new OutputStreamWriter(cos));
pw.println("Stand and unfold yourself");
pw.close();
```

La baza avem un flux de iesire care va scrie in fisierul "file" niste octeti. CipherOutputStream este o clasa decorator si adauga peste fluxul de octeti proprietatea ca acesta este criptat. Deoarece aceasta clasa nu ofera suport de lucru decat pentru octeti avem nevoie de un alt decorator care sa ne permita sa adaugam text in fluxul de iesire. De aceea folosim clasa PrintWriter. Observati ca atat PrintWriter, OutputStreamWriter cat si CipherOutputStream sunt doar niste decoratoare ele modificand intr-un anumit mod fluxul, totusi acest flux trebuie sa apara de undeva si de accea toata aceasta decorare trebuie sa aiba la baza un flux instantiat(FileOutputStream).



Mapand pe diagrama exemplul nostru, FileOutputStream joaca rolul de componenta concreta, PrintWriter, OutputStreamWriter si CipherOutputStream joaca rolul de decoratoare concrete.

Atentie: Daca in tot lantul de decorari nu exista un decorator care sa aiba la baza o componenta concreta constructia nu va functiona.

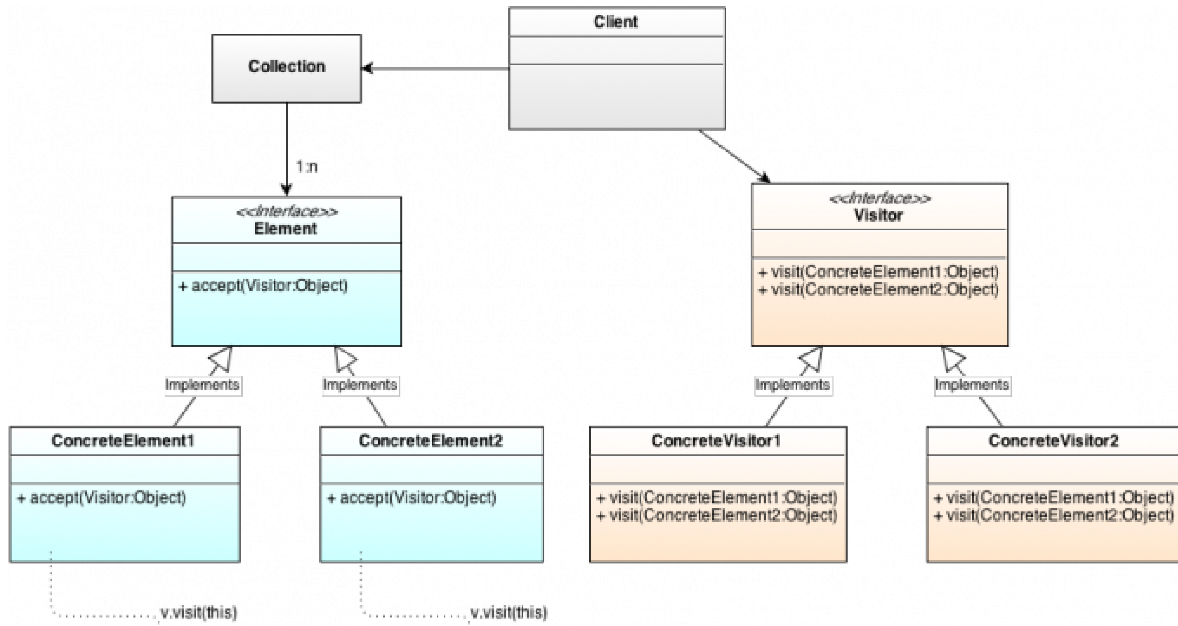
1.5 Behavioral Design Patterns

Acestea trateaza problemele legate de modul in care mai multe obiecte interactioneaza.

1.5.1 Visitor

Uneori intr-o aplicatie, va trebui sa separati operatiile pe un anumit tip de date de datele efecitve pentru a putea permite operatii care in momentul proiectarii nu sunt cunoscute. De exemplu sa presupunem ca avem la dispozitie mai multe tipuri de eroi intr-un joc MMORPG. La inceput puteti sa definiti niste variante de atac intre tipurile de eroi dar nu puteti sa garantati ca in

viitor nu va veni Game-Designer-ul cu o noua idee inedita si sa trebuiasca sa se adauge o noua posibilitate de atac intre eroi. Daca variantele de atac au fost definitie ca metode interne in clasele ce definesc eroii, atunci orice nou tip de atac presupune modificarea codului scris deja. In schimb daca tipurile si variantele de atac sunt implementate separat aveti o flexibilitate sporita in exindere.



Util de folosit cand

- avem mai multe obiecte si operatii pentru acestea
- dorim schimbarea/adăugarea operatiilor fără a modifica clasele
- avem nevoie de interactiune pe baza tipului dinamic al obiectelor

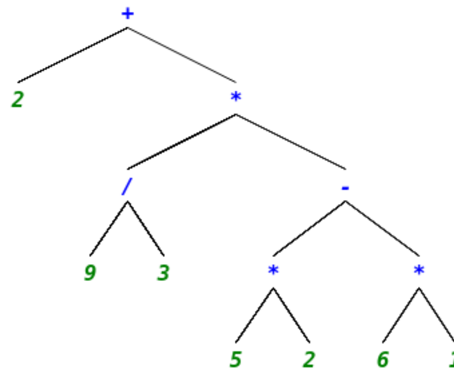
1.5.2 Observer

L-am discutat in introducere.

2 Probleme de laborator

Scopul vostru este sa dezvoltati o aplicatia care ajuta elevii din clasa a 3-a sa rezolve expresii aritmetice. Aplicatia voastra va primi ca input un string ce contine o expresie si va arata elevului pas cu pas cum se rezolva expresia. Aceasta aplicatie se va numi "Arithmo". Formatul expresiilor de input va fi: (operand operator operand) unde operatorul poate fi *,/,-,+ iar operanzii pot fi valori numerice sau alte expresii. Un posibl input poate fi (2 + ((9 / 3) * ((5 * 2) - (6 * 1)))) Evolutia expresiei va fi: (2 + (3 * (10 - 6))) -> (2 + (3 * 4)) -> (2 + 12) -> 14

Pentru a putea face acest lucru o sa transformam expresia intr-un Abstract Syntax Tree (AST) ce va arata asa.



Task0 **0p**

Inainte de a incepe laboratorul trebuie sa intelegeti ce design pattern-uri au fost deja implementate in scheletul de cod. Incercati sa va dati singuri seama de ele si daca nu reusiti intrebati-va asistentul. Nu incepeti laboratorul pana nu va este foarte clar in ce legatura se afla obiectele si clasele, ce rol au fiecare si ce design pattern-uri apar.

Aplicatia va functiona corect doar pentru inputuri de forma (operand operator operand) unde operanzii pot fi alte expresii. Pentru a fi siguri ca ruleaza corect dati input expresia din introducere: (2 + ((9 / 3) * ((5 * 2) - (6 * 1))))

Task1 **3p**

Deschideti arhiva auxiliara si analizati codul de acolo. In clasa ExpressionFactory trebuie sa completati codul astfel incat s-o faceti Singleton si de asemenea trebuie sa completati metoda getExpression(ExpressionPareser ep). Clasa ExpressionParser este o clasa ce separa o expresie binara data printr-un string in subexpresie stanga, un caracter ce retine operatorul si subexpresia dreapta. Atentie: metoda getExpression() poate returna obiecte de tip subclasa al clasei Expression. De asemenea inainte de a crea efectiv obiectul va trebui sa creati subexpresiile din care expresia curenta este compusa folosind tot metoda getExpression. Practic veti construi

recursiv AST-ul.

Folositi Singleton si Factory Method

Task2 **3p**

Analizati codul din clasa PrintVisitor. Aceasta clasa nu face decat sa parcurga un AST si sa-l afiseze ca o expresie aritmetica. Facand analogia cu aceasta clasa si folosind interfata Visitor data, completati clasa ComputeVisitor astfel in cat codul din main sa afiseze rezultatul final al expresiei date prin AST.

Cod sursa Java

```java
Visitor calculator = new ComputeVisitor();
System.out.println(calculator.visit(tree));
```

Aplicati Visitor Pattern

Task3 **3 + 1p**

Adaugati in clasa StepByStepSolving metodele ce lipsesc si creati o clasa iterator ce va itera peste solutiile partiale ale expresiei exp din clasa. Va trebui sa va folositi de obiectele sbscv si compute in metodele si clasa interna interator.
Testati programul pentru inputul: (v(2 + ((9 / 3) * (v((5 * 2) - (-(6 * 1))))))) si asigurati-va ca rezultatul este 3 (nu ne interseaza precizia operatiilor deocamdata).

Operatorul v reprezinta radacina patrata.

Task4 **2p**

Adaugat codul necesar si modificati clasele si interfetele date astfel incat aplicatia voastra "Arithmo" sa suporte ridicari la puteri si modulo. Pentru a realiza acest lucru va trebui sa adaugati in clasa factory inca doua case-uri in metoda getExpression, va trebui sa creati inca doua clasa ce extind BinaryOperation, si va trebui sa mai adaugati in visitori metodele supraincarcate aferente noilor tipuri de date. Pentru a reprezenta printr-un caracter operatia modulo folosim % iar pentru ridicare la putere ^ unde formatul operatiei va fi baza^ exponent.
(v(2 + ((9 / 3) * (v((5 ^ 2) - (-(6 % 1))))))) ar trebui sa dea rezultat 4.

3 Interviu

Intrebari de interviu

Anumite intrebari necesita cunostinte despre design pattern-uri nediscutate la laborator.

1.Analizand codul din arhiva de laborator, cu ce pattern puteti asemana ierarhia de tipuri de operatii si clasa Value?

2.Folosind pattern-ul Visitor ce trade-off apare? Ce este mai usor de adaugat: un nou visitor sau un nou tip de date?

3. Ce anume ar fi trebuit sa adaugati si sa modificati pentru ca aplicatia voastra sa suporte si operatii unare de genul: log2(x).

4. Ce design pattern-uri ati folosi pentru a crea o aplicatie de editare de imagini ?

5. Ce design pattern-uri si clase ati folosi pentru crea un editor de text?

6. Ce desing pattern ati folosi pentru a crea un Delegat asemanator cu cei oferiti de limbajul C#?

7. Cum ati implementa n singleton-uri in care al k-lea singleton deriveaza al k-1 singleton? Cum le-ati implementa intr-un mod economic (daca nu sunt folosite in program sa nu fie instantiate)?

8. Cum ati face design-ul unui joc de sah? Ce clase ati folosi? Ce design pattern-uri ati folosi?

9. Explicati conceptul de extindere al unei aplicatii.

10. Ce desing pattern ati folosi daca ar trebui sa faceti o aplicatie in care primiti blocuri de dimensiune fixa de date impreuna cu o informatie preliminara despre ele pe care sa le sortati in functie de informatia primita? De exemplu primul bloc poate sa vina cu informatia ca este deja sortat, urmatorul vine cu informatia ca este pseudo sortat, urmatorul bloc vine cu informatia ca este sortat in ordine inversa s.a.

Referinte

- Design Patterns, Gang of four
- Head First Desing Patterns