Universitatea “Alexandru Ioan Cuza” Iaşi
Facultatea de Economie şi Administrarea Afacerilor
Master: Sisteme Informaţionale pentru Afaceri

Sistem de management pentru publicarea lucrărilor
Partea II: Aplicarea șabloanelor de proiectare

Coordonator ştiinţific,
Profesor: Florin Dumitriu

Realizat de:
Ciobanu Ioana-Teodora
Cirsmar Daniel-Ioan
Ghimp Sergiu

Problema 1

Lucrările din aplicaţie sunt de mai multe tipuri. Astfel trebuie să existe posibilitatea de a crea un nou tip de lucrare, mai exact domeniul din care face parte lucrarea. În funcţie de tipul lucrării şi de domeniul pe care îl are fiecare lucrare, trebuie să i se aloce un revizor. Un revizor este responsabil pentru examinarea mai multor lucrări. Sistemul oferă o singură interfaţă pentru a selecta revizorii. Prin urmare vom rezolva:

- Folosind sablonul creational de proiectare Factory - problema adăugării unui nou tip de lucrare
- Folosind sablonul Lanţul de responsabilitate - Problema atribuirii recenzorilor
- Folosind sablonului structural Facade - Problema accesării mai multor clase de recenzori printr-o singură interfaţă

1.1. Aplicarea modelului sablonului creational Factory

1.1.1. Definirea problemei

Lucrările trimise de autori sunt de diferite tipuri (ştiinţifice, fictive, exploratorii, studii de caz etc.). În acest caz, vom avea mai multe tipuri de clase pentru fiecare articol care va implementa metoda trimite().

Autorul poate alege tipul de articol pe care doreşte să îl trimită din formularul cererii şi trimite lucrarea.

1.1.2. Soluţie fără a aplica modelul sablonului creational Factory

Având în vedere alegerea autorului, aplicaţia va crea un obiect ştiinţific, fictiv, teoretic, exploratoriu etc. şi va invoca metoda trimite().

```
//autorii adauga tipul lucrării

if (tiplucrare == "stiintific")
{
// apeleaza tipulstiintific şi metodele lui
    Stiintific s = new stiintific()
    s.trimite();
}

if (tiplucrare == "fictiv")

// capeleaza tipulfictiv si metodele lui
    Fictiv f = new fictiv()
    f.trimite()
    }
```

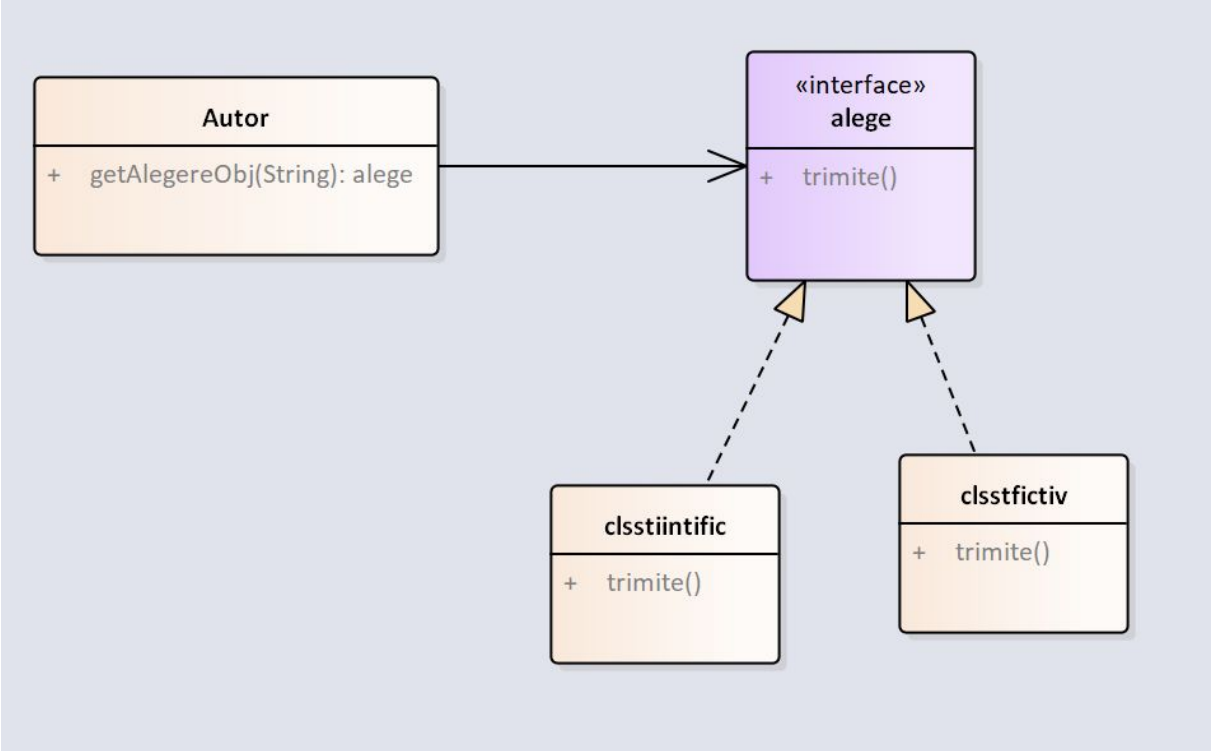
1.1.3. Dezavantajele soluţiei propuse

- Un nou tip de articol trimis implică modificarea codului de cerere;
- Codul nu are securitate deoarece se pot vedea cele două clase şi metodele implicate.

1.1.4. Soluţia când aplicăm modelul sablonului creational Factory

Utilizând sablonul Factory:

- vom crea o interfaţă pentru definirea metodei submit(), care va fi implementată de clasele metodologice şi teoretice.
- vom crea o clasă cu rol de Factory care va avea o metodă cu scopul „fabricării" obiectelor ştiinţifice şi fictive;
- clasa va fi utilizată între aplicaţia client şi cele două clase specifice ale domeniului.




```
public interface alege
{
        String trimite()
}
//Clasa Lucrari Stiintifice
public class clsstiintific:alege
{
        public string trimite()
        {
        return(" Aţi ales ştiinţific")
        }
}
//Clasa Lucrari Fictive
public class clsstfictiv:alege
{
        public string trimite()
```

```
        {
        return(" Aţi ales fictiv")
        }
}
//Clasa Factory
public class FactoryAlege
{
        static public Alege getAlegereObj(String Alegere)
        {
                IfAlegere objAlegere=null;
                if(Alegere.toLower()=="stiintific")
                {
                        objAlegere=newclasstiintific();
                }
                else if (a.Alegere.toLower()=="fictiv")
                {
                        objAlegere = newclsfictiv();
                }
                else
                {
                        objAlegere=newAlegereInvalida();
                }
                return objAlegere
        }
}

//Clasa Autor

        Alege objArticle;
        objTipLucrare = FactoryAlege.getAlegereobj(txtAlege.Text.Trim());
        MessageBox.Show(obLucrare.Trimite());
```

1.1.5. Comentarii cu privire la aplicarea sablonului creational Factory

- Logica de creare a obiectelor nu este expusă părţii client a aplicaţiei;
- Referirea la noul obiect creat se face printr-o interfaţă, clientul va vedea şi interacţiona cu un obiect generic;
- Evită implementarea logicii alegerii unei clase în partea client - va implica o dependenţă prea mare;
- Codul din partea client nu se modifică atunci când introducem un nou tip de obiect sau logica alegerii unei clase pentru a instanţia modificările;
- Modificările sunt localizate într-o singură clasă şi o metodă.

1.2. Aplicarea sablonului comportamental Lanţul de responsabilitate

1.2.1. Definirea problemei

Fiecărei lucrări trimise editurii îi pot fi atribuite mai mulţi revizori.

Problema atribuirii revizorilor la fiecare lucrare poate fi rezolvată folosind Lanţul de responsabilitate, deoarece există mai mulţi revizori care pot analiza o lucrare.

1.2.2. Soluţie fără aplicarea sablonului comportamental Lanţul de responsabilitate

De fiecare dată când se adaugă un nou tip de lucrare dintr-un domeniu de interes diferit (geografie, de exemplu), trebuie să verificăm tipul articolului şi să atribuim recenzorul manual.

1.2.3. Dezavantajele soluţiei propuse

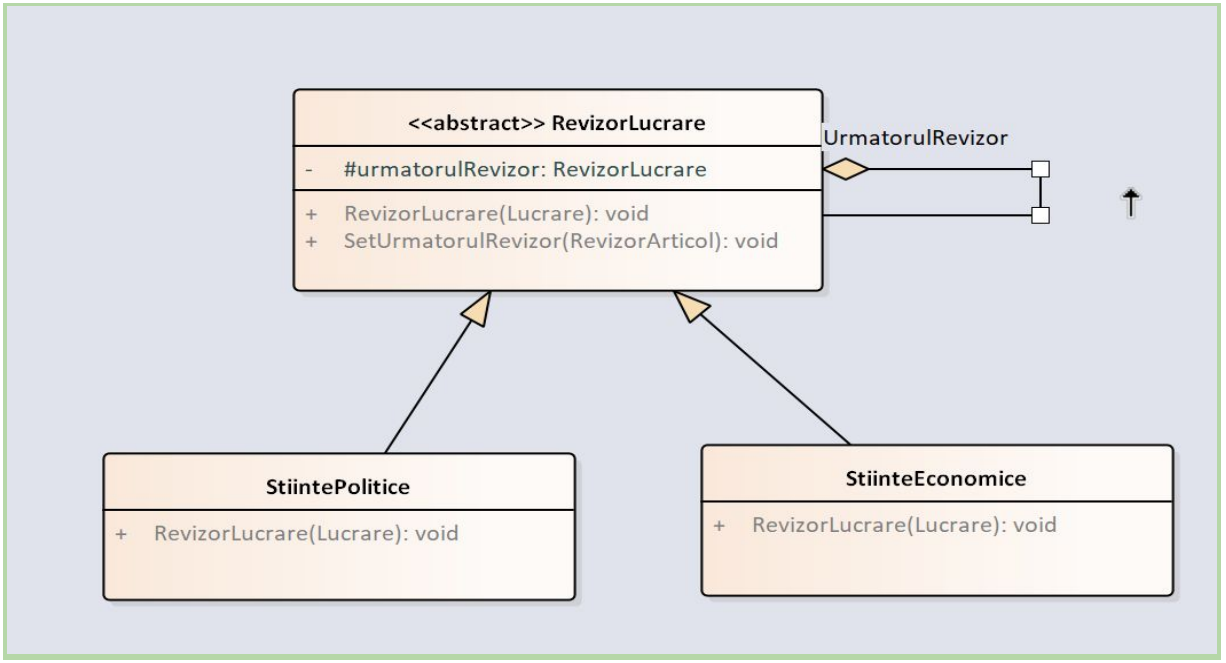
Apariţia unui alt tip de lucrare va determina selecţia manuală a unui revizor care urmează să fie atribuit.

1.2.4. Soluţia când aplicăm sablonului comportamental Chain of responsibility

Scopul aplicării sablonului comportamental Lanţul de responsabilitate este de a evita cuplarea directă între expeditorul unei cereri şi receptorul cererii.

Clientul trimite cererea către un lanţ de obiecte fără să ştie ce obiect îi va rezolva cererea.

Obiectele din lanţ decid singure cine va rezolva cererea.




```
class StiintePolitice: RevizorLucrare
  {
        publicoverrirde void RevizorLucrare(Lucrare l)
        if (l.DomeniuLucrare = "Economic")
```

```
 Console.WriteLine("Revizorul articolului"+l.DomeniuLucrare+"revizuit de
StiinteEconomice”);
                else
                        urmatorulRevizor.RevizorLucrare(l);
        }
        }

        class StiintePolitice: RevizorLucrare
        {
                publicoverrirde void RevizorLucrare(Lucrare l)
                {
 Console.WriteLine("Revizorul lucrarii"+l.DomeniuLucrare+"revizuit de Stiinte Politice");")
                        else
                                urmatorulRevizor.RevizorArticol(l);
                }
        }
```

1.2.5. Comentarii cu privire la aplicarea sablonului comportamental Lanţul de responsabilitate

Clasa abstractă **RevizorLucrare** este un părinte pentru toate obiectele care pot rezolva cererile.

Variabila **UrmătorulRevizor** este o referinţă la următorul obiect din lanţ.

Metoda **RevizorLucrare** va fi implementată de toate clasele de copii.

Clasele **StiintePolitice, StiinteEconomice** reprezintă clasele care vor revizui articolele. Metoda **SetUrmatorulRevizor** verifică dacă articolul poate fi revizuit de o anumită clasă (revizori cu studii în economie) şi dacă nu, îl trimite la o altă clasă.

1.3. Aplicarea sablonului structural Façade

1.3.1. Definirea problemei

Problema: furnizarea unei singure interfeţe pentru a accesa mai multe clase de revizori.

1.3.2. Soluţie fără sablonul structural Façade

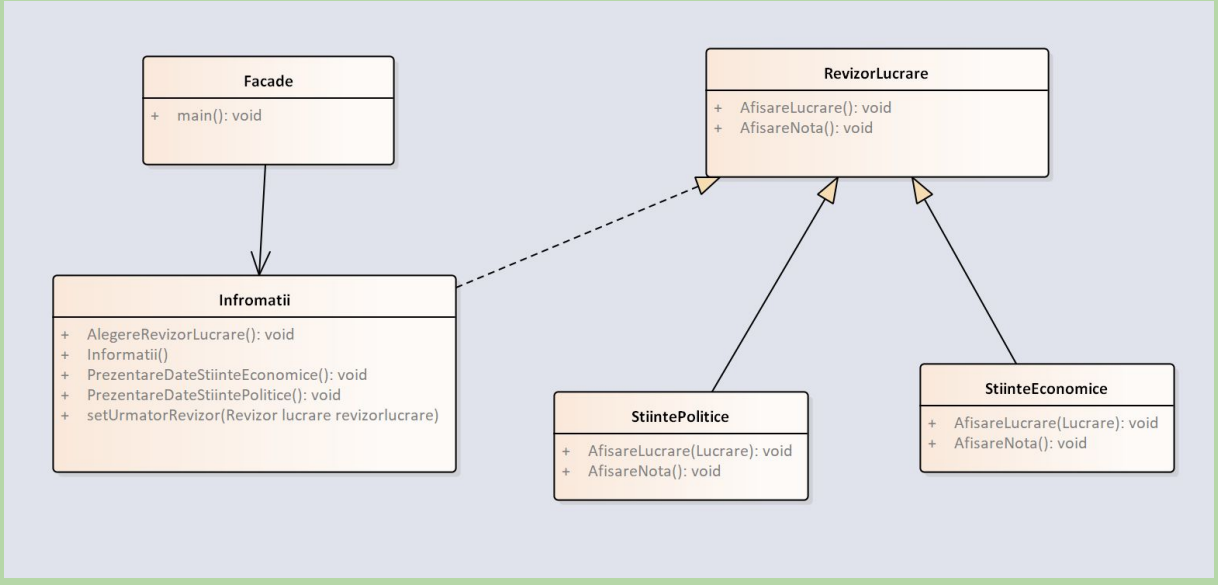
Crearea mai multor interfeţe pentru fiecare tip de revizor.

1.3.3. Dezavantajele folosirii soluţiei propuse

De fiecare dată când adăugaţi un revizor nou, trebuie să se creeze o interfaţă specifică.

Atunci când modelul Facade nu este aplicat, logica metodelor specifice ar trebui implementată direct în lucrări.

1.3.4. Soluţie când aplicăm sablonului structural Façade




```
interface RevizorLucrare {
    void AfisareLucrare();
    void AfisareNota();
}
class StiinteEconomice implements RevizorLucrare {
    @Override
    public void AfisareLucrare() {
        System.out.println("StiinteEconomice");
    }
    @Override
    public void AfisareNota() {
        System.out.println("8");
    }
}
class StiintePolitice implements RevizorLucrare {
    @Override
    public void AfisareLucrare() {
        System.out.println("StiintePolitice");
    }
    @Override
    public void AfisareNota() {
        System.out.println("9");
    }
}
class Informatii {
    private RevizorLucrare stiintePolitice;
    private RevizorLucrare stiinteEconomice;
    public Informatii() {
        stiinteEconomice = new StiinteEconomice();
        stiintePolitice = new StiintePolitice();
```

```
    }
    public void PrezentareDateStiintePolitice()
    {
        stiintePolitice.AfisareLucrare();
        stiintePolitice.AfisareNota();
    }
    public void PrezentareDateStiinteEconomice()
    {
        stiinteEconomice.AfisareLucrare();
        stiinteEconomice.AfisareNota();
    }
}
public class Main {
    public static void main(String[] args) {
        Informatii info = new Informatii();
        info.PrezentareDateStiinteEconomice();
    }
```

Problema 2

Problema care va fi rezolvata cu trei cele 3 tipuri de sabloane de proiectare în sistemul de management pentru publicarea lucrărilor privind reducerea acordată de editură la plata lucrărilor în funcţie de mai multe criterii diferite.

1. Editura are multiple strategii de reducere în funcţie de diferite criterii - vom rezolva problema aplicând şablonul de proiectare *Decorator*.
2. Un nou tip de reducere poate fi adăugat în orice moment în sistem - vom rezolva problema aplicând şablonul de proiectare *Strategy*.

2.1 Aplicarea şablonului structural de proiectare *Decorator.*

2.1.1 Definirea problemei

În aplicaţia de gestiune pentru publicarea lucrărilor, trebuie să rezolvăm problema gestionării cazului de politici multiple de preţuri conflictuale pentru autori atunci când plătesc articolele. Editura are următoarele politici cu privire la preţuri:

1. 20% reducere pentru autorii care sunt doctoranzi;
2. 30% reducere dacă un autor plăteşte mai mult de o lucrare;
3. 15% reducere dacă un autor are publicate mai mult de 3 lucrări în ultima lună.

Din cauza ca pot exista mai multe strategii de reducere care să existe în acelaşi timp, o plată poate avea mai multe strategii de preţ, ceea ce înseamnă că o plată poate avea mai multe reduceri. O altă remarcă poate fi ca o strategie de stabilire a preţurilor poate sa fie legată de tipul autorului (de exemplu: doctorand, student). La fel se poate spune şi de stategia de stabilire a preţurilor care poate fi legată numărul de lucrări trimise şi publicate pentru acelaşi acelaşi autor.

2.1.2 Soluţii fără aplicarea unui şablon structural de proiectare *Decorator.*

Pentru a rezolva prima problemă a reducerilor multiple fără a aplica modelul, putem implementa în clasa Factură fiecare tip de reducere.

2.1.3 Dezavantajele soluţiilor propuse.

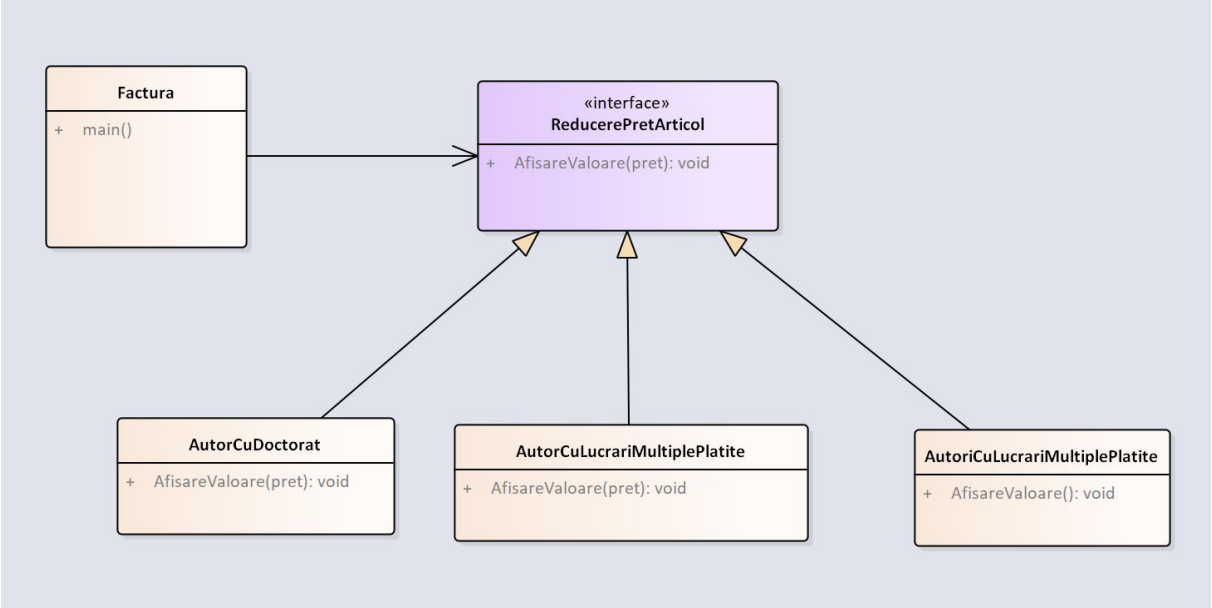
De fiecare dată când apare un nou tip de reducere, trebuie sa îl introducem manual (scriem cod).

2.1.4 Soluţia de aplicare a sablonului structural *Decorator*.

Numele sablonului de proiectare aplicat: *Decorator*

Problema: Cum se tratează un grup sau o structură de compoziţie a obiectelor în acelaşi mod (polimorf) ca un obiect compozit (atomic).

Soluţie: Am definit clasele pentru obiectele compozite şi atomice, astfel încât acestea să implementeze aceeaşi interfaţă.




```
interface ReducerePretArticol{
   abstract void AfisareValoare(int pret);
}
class AutorCuLucrariMultiplePublicate implements ReducerePretArticol{
   @Override
   public void AfisareValoare(int pret) {
      System.out.println(pret-0.15*pret);
   }
}
class AutorCuLucrariMultiplePlatite implements ReducerePretArticol{
   @Override
   public void AfisareValoare(int pret) {
      System.out.println(pret-0.3*pret);
   }
```

```
}
class AutorCuDoctorat implements ReducerePretArticol{
   @Override
   public void AfisareValoare(int pret) {
      System.out.println(pret-0.2*pret);
   }
}
public class Main {
   public static void main(String[] args) {
      ReducerePretArticol Reducere = new AutorCuDoctorat();
      Reducere.AfisareValoare(150);
   }
```

2.1.5 Comentarii privind aplicarea şablonului structural de proiectare *Decorator*.

Clasele compuse sunt **AutorCuDoctorat**, **AutorCuLucrariMultiplePublicate** şi **AutorCuLucrariMultiplePlatite** care aplica discountul pe produsul respectiv.

2.2 Aplicarea şablonului comportamental de proiectare *Strategy*

2.2.1 Definirea problemei

Problema constă în rezolvarea selecţiei unei anumite reduceri în funcţie de situaţia din momentul rulării, în care nu dorim să implementăm algoritmii şi logica acestora către client.

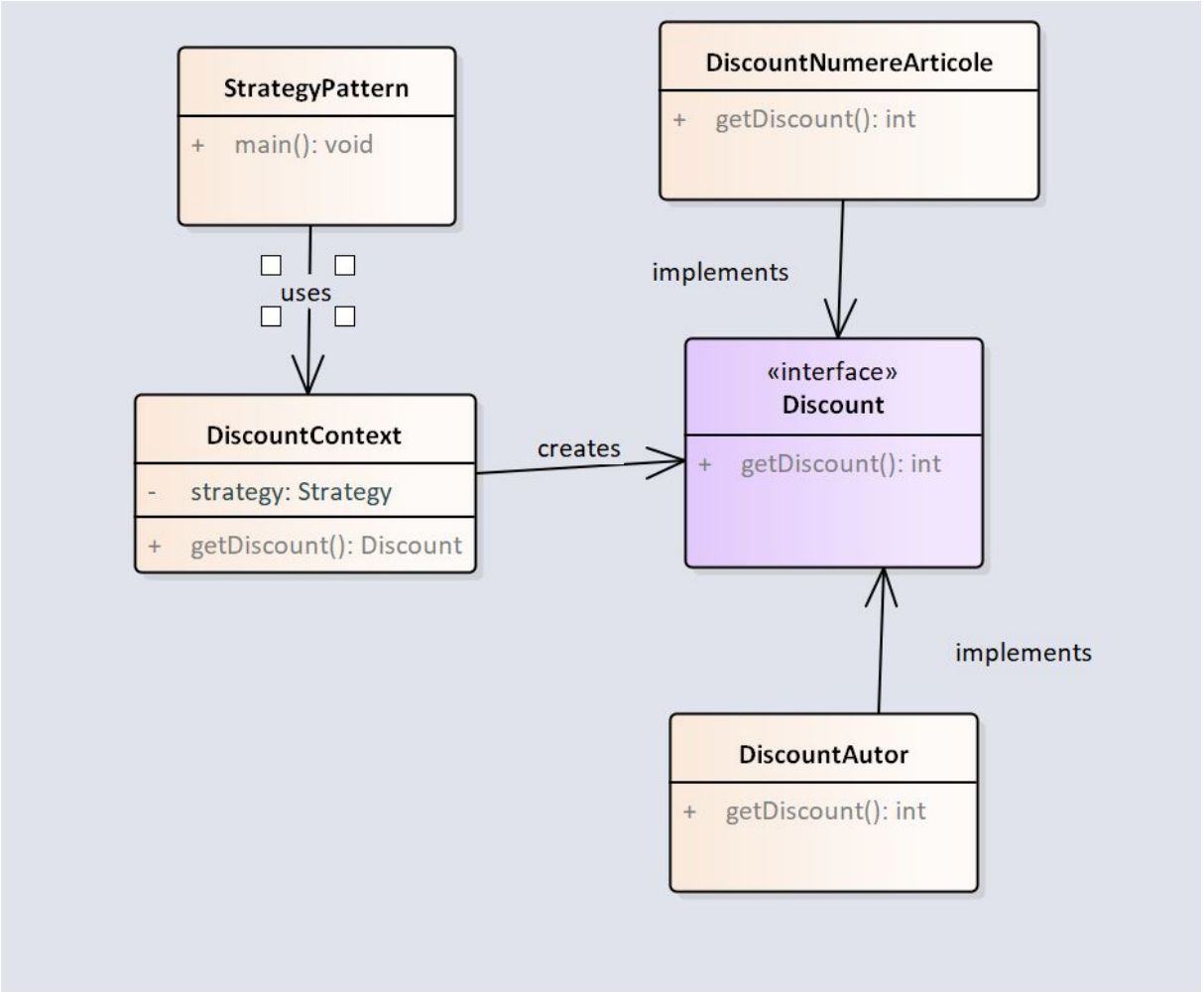
2.2.2 Soluţii fără aplicarea unui şablon comportamental de proiectare *Strategy*

Fiecare reducere pe care o deţinem trebuie sa o tratam separat.

2.2.3 Dezavantajele soluţiilor propuse.

De fiecare dată când apare un nou tip de reducere, trebuie să scriem cod pentru tipul de reducere apărut.

2.2.4 Soluţia de aplicare a sablonului comportamental *Strategy*.




```
public interface Discount {
	public int getDiscount(int valoare);

}
public class DiscountAutor implements Discount {
	@override
	public int getDiscount(int valoare) {
		return int(valoarePlata-(valoarePlata*0.2));
	}
}

public class DiscountNumereLucrari implements Discount {
	@override
	public int getDiscount(int valoare) {
		return int(valoarePlata-(valoarePlata*0.3));
	}
}
public class DiscountContext {
```

```
        private Discount discount;

        public void setDiscount( Discount discount) {
        this.strategy = strategy;
        }
        public Discount getDiscount() {
                return strategy;
        }
public int getDiscount(int valoare) {
return strategy.getDiscount(int valoare);
}
}
public class StrategyPattern {
        public static void main(String[] args) {
                // TODO Auto-generated method stub
                DiscountContext context = new DiscountContext();
                int valoarePlata;

                context.setDiscount(new DiscountAutor);
                valoarePlata = context.getDiscount(valoarePlata);

                context.setDiscount(new DiscountNumereLucrari);
                valoarePlata = context.getDiscount(valoarePlata);

                System.out.println("valoare de incasat = " + valoarePlata)
        }
}
```

2.3.5 Comentarii privind aplicarea şablonului comportamental de proiectare *Strategy*.

Permite implementarea mai multor algoritmi pentru a aplica reduceri.