Laborator 12: Design Patterns

1 Responsabil

Gabriel Guțu-Robu, gabriel.gutu@upb.ro
Publicat: 14 ianuarie 2023, 01:47

2 Introducere

Șabloanele de proiectare (en., *design patterns*) ușurează dezvoltarea programelor și ajută programatorii în alegerea alternativelor care să facă aplicațiile reutilizabile (Freeman et al., 2008; Gamma, 1995). Orice șablon de proiectare este alcătuit din 4 elemente:

1. *Numele* – ajută la descrierea problemei, soluțiilor și consecințelor, într-un cuvânt sau două;
2. *Problema* – definește unde se utilizează șablonul. Exprimă problema și contextul, precum și cum se transpun algoritmii ca obiecte. Uneori, problema include o listă de condiții care trebuie îndeplinite înainte de aplicarea șablonului;
3. *Soluția* – descrie elementele de proiectare, relațiile, responsabilitățile și modul de comunicare. Soluția nu descriere o implementare concretă. În schimb, șablonul furnizează o descriere abstractă a proiectării problemei și o aranjare generală a elementelor sale (clase și obiecte);
4. *Consecințele* – reprezintă rezultatele și compromisurile aplicării șablonului. Ele sunt critice pentru evaluarea alternativelor de proiectare și pentru înțelegerea costurilor și beneficiilor aplicării șablonului din punctul de vedere al complexității timpului și spațiului, reutilizării codului, impactul asupra flexibilității, extensibilității și portabilității sistemului.

Clasificarea șabloanele de proiectare se face după două criterii: 1) scop – ce face acel șablon: creaționale (crearea obiectelor), structurale (alcătuirea claselor) și comportamentale (modul în care clasele și obiectele interacționează și își distribuie responsabilitățile); 2) domeniu – specifică dacă șablonul se aplică claselor sau obiectelor. Șabloanele de clasă se ocupă de relația dintre clase și subclase, stabilite prin moștenire. Șabloanele de obiect se ocupă de relațiile dintre obiecte, relații care se pot modifica la rulare, fiind mult mai dinamice. De multe ori, șabloanele sunt folosite împreună. De exemplu, Composite este folosit cu Iterator sau Visitor. Unele șabloane au alternative. Prototype poate fi o alternativă la Abstract Factory.

Tabelul 1 prezintă clasificarea șabloanelor de proiectare, urmat de o detaliere a fiecăruia.

Tabelul 1. Clasificarea șabloanelor de proiectare.

		Scop		
		Creațional	*Structural*	*Comportamental*
Domeniu	*Clasă*	Factory Method	Adapter	Interpreter Template Method
	Obiect	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Abstract Factory – Furnizează o interfață pentru crearea de familii de obiecte înrudite sau dependente unele de altele, fără a specifica clasa lor concretă.

Adapter – Convertește interfața unei clase într-o altă interfață pe care o solicită clientul. Adapter permite claselor care au interfețe incompatibile să lucreze împreună.

Bridge – Decuplează abstractizarea de implementare, astfel încât cele două pot varia independent.

Builder – Separă construcția unui obiect complex de reprezentarea sa, astfel încât același proces de construcție poate conduce la diverse reprezentări.

Chain of Responsibility – Evită cuplarea expeditorului unei cereri de destinatar și dă șansa mai multor obiecte să se ocupe de acea cerere.

Command – Încapsulează o cerere ca obiect, parametrizează clienții cu diferite cereri și suportă operații ireversibile.

Composite – Compune obiectele în structuri ierarhice și lasă clientul să trateze obiectele simple și compuse în mod uniform.

Decorator – Atașează diverse responsabilități unui obiect în mod dinamic. Decoratorii furnizează o alternativă flexibilă pentru extinderea funcționalității în procesul de moștenire.

Facade – Furnizează o interfață uniformă unui set de interfețe dintr-un subsistem. Definește o interfață de nivel mai înalt care face subsistemul mai ușor de utilizat.

Factory – Definește o interfață pentru crearea unui obiect, dar lasă subclasele să decidă în privința instanțierii.

Flyweight – Intervine în partajarea unui număr mare de obiecte.

Interpreter – Fiind dat un limbaj, definește o reprezentare pentru gramatica sa, alături de un interpretor care utilizează această reprezentare pentru a traduce propozițiile în limbaj.

Iterator – Furnizează o modalitate de acces la elementele unui obiect agregat în mod secvențial, fără să îi expună implementarea.

Mediator – Definește un obiect care încapsulează modul în care un set de obiecte interacționează. Mediator previne obiectele de a se referenția unul pe altul explicit și lasă utilizatorului posibilitatea de a varia interacțiunea lor în mod independent.

Memento – Fără a încălca principiile încapsulării, captează și externalizează reprezentarea internă a unui obiect.

Observer – Definește o dependență unul-la-mai-multe între obiecte, astfel încât atunci când un obiect își schimbă starea, toate obiectele dependente sunt notificate și modificate automat.

Prototype – Specifică tipul de obiecte ce vor fi create utilizând un prototip și creează noi obiecte prin copierea acestui prototip.

Proxy – Furnizează un surogat pentru controlul accesului la un anumit obiect.

Singleton – Asigură crearea unei singure instanțe dintr-o clasă și furnizează un punct global de acces la ea.

State – Permite unui obiect să își schimbe comportamentul atunci când reprezentarea sa internă se modifică.

Strategy – Definește o familie de algoritmi, încapsulează pe fiecare și îi face interschimbabili.

Template – Definește un schelet al unui algoritm într-o operație și lasă subclasele să redefinească anumiți pași ai algoritmului fără să schimbe structura acestuia.

Visitor – Permite definirea unei noi operații fără să schimbe structura claselor asupra cărora operează.

Dintre șabloanele anterioare a fost selectată o serie de șabloane reprezentative, frecvent utilizate, pentru care sunt furnizate exemple detaliate.

Printre problemele pe care le rezolvă șabloanele de proiectare se numără:

- Identificarea modalității de abstractizare a claselor și obiectelor ce compun sistemul;
- Determinarea granularității obiectelor, care pot varia în dimensiune și număr;
- Specificarea interfeței obiectului, adică setul de cereri ce pot fi trimise acestuia și totalitatea semnăturilor operațiilor care lucrează cu un anumit obiect;
- Implementarea claselor si obiectelor – variabile membru, metode, dacă sunt abstracte sau nu, interne sau nu etc;
- Structurarea și ierarhizarea claselor – compunere vs. moștenire.

3 Design Patterns creaționale

Șabloanele de proiectare creaționale determină sistemul să fie independent de modul în care obiectele sunt create, compuse și reprezentate.

3.1 Singleton

Șablonul *Singleton* asigură faptul că o clasă are o singură instanță și furnizează un punct global de acces la ea. Este important pentru anumite clase să aibă o singură instanță, de exemplu poate exista un singur sistem de fișiere într-o aplicație. Cum ne asigurăm că o clasă are o singură instanță și că este ușor accesibilă? O variabilă globală face un obiect accesibil, dar nu previne instanțierea multiplă. O soluție mai bună este de a face clasa responsabilă de a ține evidența singurei sale instanțe. Clasa trebuie să se asigure că nicio altă instanță nu va fi creată și că furnizează o modalitate de a accesa instanța.

Șablonul Singleton se folosește atunci când:

- Trebuie să avem o singură instanță a clasei, care să fie accesibilă clienților dintr-un punct de acces bine cunoscut;
- Când singura instanță trebuie să fie accesibilă prin moștenire și clienții trebuie să fie capabili să utilizeze versiunea derivată a instanței fără să îi modifice codul.

În exemplul de mai jos, clasa `Singleton` conține variabila membru `unicaInstanta`, privată și statică, de tipul *Singleton*. Constructorul privat asigură că nu putem crea obiecte din afara clasei. Metoda statică `Instanta()` creează o instanță a obiectului, dacă acesta nu există, sau îl returnează pe cel existent. În metoda `main`, se creează un singur obiect din clasa `Singleton`, utilizându-se metoda `Instanta()` și se afișează adresa acestui obiect.

```
public class Singleton {
     private static Singleton instantaUnica;
     private Singleton() {}

     public static Singleton Instanta() {
          if (instantaUnica == null)
               instantaUnica = new Singleton();
```

```
            return instantaUnica;
      }

      public static void main(String[] args) {
            Singleton obj = Instanta();
            System.out.println(obj);
      }
}
```

Se va afișa:

```
Singleton@70dea4e
```

3.2 *Factory*

Șablonul de proiectare *Factory* definește o interfață pentru crearea unui obiect, dar lasă subclasele să decidă ce clasă să instanțieze. Framework-urile utilizează clase abstracte pentru a defini și menține relațiile dintre obiecte. Framework-urile sunt de multe ori responsabile pentru crearea acestor obiecte.

Șablonul de proiectare *Factory* se aplică atunci când:

- O clasă nu poate anticipa clasele obiectelor pe care trebuie să le creeze;
- O clasă vrea ca subclasele să specifice obiectele pe care le creează;
- O clasă deleagă responsabilitate uneia dintre subclase ajutătoare.

În exemplul de mai jos, clasa `Factory` are metoda `creeazaObiectGeometric (String tip)`, ce creează un obiect geometric – `Cerc`, `Dreptunghi` sau `Pătrat`, în funcție de valoarea variabilei `tip`.

```
abstract class ObiectGeometric {
      public abstract void deseneaza();
}

class Cerc extends ObiectGeometric {
      int raza;
      public Cerc(int raza) {
            this.raza = raza;
      }
      public void deseneaza() {
            System.out.println("Cerc de raza " + raza);
      }
}

class Dreptunghi extends ObiectGeometric {
      int lungime, latime;
      public Dreptunghi(int lungime, int latime) {
            this.lungime = lungime;
            this.latime = latime;
```

```
     }
     public void deseneaza() {
          System.out.println("Dreptunghi de dimensiuni " +
lungime + " " + latime);
     }
}

class Patrat extends ObiectGeometric {
     int latura;
     public Patrat(int latura) {
          this.latura = latura;
     }
     public void deseneaza() {
          System.out.println("Patrat de latura " + latura);
     }
}

public class Factory {
   public ObiectGeometric creeazaObiectGeometric(String tip)
{
          switch (tip) {
          case "Cerc":
               return new Cerc(2);
          case "Dreptunghi":
               return new Dreptunghi(5, 3);
          case "Patrat":
               return new Patrat(6);
          default:
               return null;
          }
   }
   public static void main(String[] args) {
     Factory f = new Factory();
     ObiectGeometric c = f.creeazaObiectGeometric("Cerc");
     c.deseneaza();
     ObiectGeometric     d     =     f.creeazaObiectGeometric
("Dreptunghi");
     d.deseneaza();
     ObiectGeometric p = f.creeazaObiectGeometric("Patrat");
     p.deseneaza();
   }
}
```

Se va afșa:

```
Cerc de raza 2
Dreptunghi de dimensiuni 5 3
Patrat de latura 6
```

4 Design Patterns structurale – Decorator

Șablonul de proiectare *Decorator* atașează responsabilități adiționale unui obiect în mod dinamic. Decorator furnizează o alternativă flexibilă moștenirii pentru extinderea funcționalităților. Mai este cunoscut și sub denumirea de *Wrapper*.

Uneori, se dorește adăugarea de responsabilități obiectelor individuale, nu întregii clase. Spre exemplu, o interfață grafică poate adăuga proprietăți ca borduri, sau comportamente ca derularea (en., *scrolling*), oricărei interfețe cu utilizatorul. O posibilitate de adăugare a responsabilităților este prin moștenire. Moștenirea unei borduri dintr-o altă clasă pune o bordură fiecărei instanțe a subclasei. Acest lucru este inflexibil, deoarece opțiunea de adăugare a bordurii se face static. Un client nu poate controla cum și când să decoreze componenta cu o bordură. O abordare mult mai flexibilă este să încadrăm componenta într-un alt obiect care adaugă bordura. Obiectul care încadrează se numește *decorator*. Decoratorul trimite cereri componentei și realizează diverse acțiuni, cum ar fi desenarea unei borduri.

Șablonul de proiectare **Decorator** se folosește atunci când:

- Se dorește adăugarea de responsabilități unui obiect în mod dinamic și transparent, fără să afecteze celelalte obiecte;
- Există responsabilități ce trebuie retrase;
- Moștenirea este o soluție mai puțin practică. Câteodată un număr mare de clase derivate produce o explozie sau definiția unei clase poate fi ascunsă / indisponibilă pentru moștenire.

În exemplul de mai jos, clasa `ObiectGeometricDecorator` este derivată din clasa abstractă `ObiectGeometric`. Ea conține ca variabilă membru un obiect de tipul `ObiectGeometric`, numit `obDecorat`. Clasa `DecoratorCuloare` extinde clasa `ObiectGeometricDecorator` și realizează, în cadrul metodelor sale, apeluri la metodele din clasa de bază. Ea are în plus metoda `colorare(ObiectGeometric obDecorat)`, ce afișează mesajul "`Obiect colorat`" și care se apelează în metoda `deseneaza()`.

```
public class ObiectGeometricDecorator extends ObiectGeometric
{
     protected ObiectGeometric obDecorat;
     public     ObiectGeometricDecorator     (ObiectGeometric
obDecorat) {
          this.obDecorat = obDecorat;
     }
     public void deseneaza() {
          obDecorat.deseneaza();
     }
     public static void main(String[] args) {
          ObiectGeometric cerc = new Cerc(3);
```

```
            ObiectGeometric cercColorat = new DecoratorCuloare
(cerc);
            cerc.deseneaza();
            cercColorat.deseneaza();
            ObiectGeometric dreptunghi = new Dreptunghi(5,2);
            ObiectGeometric     dreptunghiColorat     =     new
DecoratorCuloare(dreptunghi);
            dreptunghi.deseneaza();
            dreptunghiColorat.deseneaza();
      }
}

class DecoratorCuloare extends ObiectGeometricDecorator {
      public DecoratorCuloare(ObiectGeometric obDecorat) {
            super(obDecorat);
      }
      private void colorare(ObiectGeometric obDecorat) {
            System.out.println("Obiect colorat");
      }
      public void deseneaza() {
            obDecorat.deseneaza();
            colorare(obDecorat);
      }
}
```

Se va afișa:

```
Cerc de raza 3
Cerc de raza 3
Obiect colorat
Dreptunghi de dimensiuni 5 2
Dreptunghi de dimensiuni 5 2
Obiect colorat
```

5 Design Patterns comportamentale

Șabloanele de proiectare comportamentale (en., *behavioral*) sunt legate de algoritmi și de asignarea responsabilităților între obiecte, precum și de modul în care acestea comunică.

5.1 Observer

Șablonul de proiectare *Observer* definește o relație unul – la mai mulți între obiecte în așa fel încât, atunci când un obiect își schimbă starea, celelalte sunt notificate și actualizate imediat. Elementele cheie în acest pattern sunt subiectul și observatorul. Un subiect poate avea oricâți observatori asignați. Toți observatorii sunt notificați atunci când subiectul își schimbă starea. Astfel, fiecare observator va interoga subiectul pentru a-și sincroniza starea

cu a acestuia din urmă. Subiectul notifică chiar și fără să știe cine sunt observatorii săi. Șablonul de proiectare *Observer* se folosește în următoarele situații:

- Când există o relație de dependență între obiecte;
- Când modificarea unui obiect determină modificarea altora, chiar și în număr variabil;
- Când un obiect trebuie să notifice alte obiecte, chiar și fără să știe care sunt acestea.

Subiectul – își cunoaște observatorii. Oricâte obiecte de tipul Observator pot observa un subiect. Obiectul Subiect furnizează o interfață pentru a atașa și detașa observatori.

Observator – definește și actualizează interfața pentru obiecte care trebuie notificate de modificarea subiectului.

SubiectConcret – trimite notificări observatorilor când starea lui se schimbă.

ObservatorConcret – menține o referință la un obiect de tipul SubiectConcret; stochează starea care trebuie să fie consistentă cu a subiectului; implementează interfața Observator pentru a-și menține starea consistentă cu a subiectului.

În următorul exemplu, clasa `Subiect` are o variabilă membru numită `stare` și un observator atașat, din clasa `Observator`. Atunci când subiectul își schimbă starea, observatorul este notificat. El se actualizează prin afișarea ultimei stări a subiectului.

```
public class Subiect {
     private int stare;
     public Observator observator;
     public int Stare() {
          return stare;
     }
     public void seteazaStare(int stare) {
          this.stare = stare;
          notificaObservator();
     }
     public void ataseazaObservator(Observator observator) {
          this.observator = observator;
     }
     public void notificaObservator() {
          observator.actualizeaza();
     }
     public static void main(String[] args) {
          Subiect subiect = new Subiect();
          Observator observator = new Observator(subiect);
          subiect.seteazaStare(1);
          subiect.seteazaStare(2);
     }
}
```

```
class Observator {
     Subiect subiect;
     public Observator(Subiect subiect) {
          this.subiect = subiect;
          this.subiect.ataseazaObservator(this);
     }
     public void actualizeaza() {
          System.out.println("Stare = " + subiect.Stare());
     }
}
```

Se va afișa:

```
Stare = 1
Stare = 2
```

5.2 Command

Șablonul de proiectare *Command* încapsulează o cerere ca obiect, permițând parametrizarea clienților cu diverse cereri și suportând operații reversibile.

Câteodată, este necesar să trimitem cereri către obiecte fără să știm nimic despre operația solicitată sau despre obiectul care primește cererea. Se utilizează șablonul de proiectare *Command* atunci când:

- Dorim ca obiectele să realizeze anumite acțiuni;
- Trimitem, punem în coadă și executăm acțiuni la diverse momente de timp;
- Folosim operații reversibile. Efectele metodei `executa()` sunt anulate folosind operația `anuleaza()`. Comenzile executate sunt stocate într-un istoric. Operațiile de *anulează* și *reexecută* se realizează prin traversarea listei înainte și înapoi, și prin executarea metodelor `executa()` și `anuleaza()`;
- Se ține evidența modificărilor și a operațiilor executate asupra sistemului. Acestea pot fi reaplicate în cazul unei căderi de sistem, prin preluarea tuturor comenzilor stocate și executarea lor cu ajutorul metodei `executa()`;
- Structurarea sistemului în jurul unor operații de nivel înalt, construite pe baza operațiilor primitive. Tranzacțiile încapsulează un set de modificări aplicate datelor. Șablonul de proiectare *Command* oferă o modalitate de a modela tranzacțiile. Comenzile au o interfață comună, invocând tranzacțiile în același fel.

Comanda – declară o interfață pentru executarea unei operații.

ComandaConcreta – definește o legătură între obiectul Receptor și acțiune. Apelează metoda `executa()` prin invocarea operației corespunzătoare asupra obiectului Receptor.

Client – creează un obiect de tipul ComandaConcreta și setează receptorul.

Solicitant – trimite comenzi.

Receptor – realizează operații.

- *Clientul* creează un obiect de tipul *ComandaConcreta* și specifică receptorul.
- Obiectul *Solicitant* stochează obiectul *ComandaConcreta*.
- *Solicitantul* trimite o cerere prin apelul lui `executa()`. Când comenzile sunt reversibile, *ComandaConcreta* stochează o stare pentru operația de anulare.
- Obiectul *ComandaConcreta* invocă operații asupra receptorului pentru realizarea cererilor.

În exemplul de mai jos, clasa abstractă `Comanda` conține metoda abstractă `executa()`, ce realizează operații asupra unui obiect din clasa `Text`, ce are o variabilă membru numită `continut`, de tipul `String`. În clasa concretă `ComandaLitereMici`, derivată din clasa abstractă `Comanda`, în cadrul metodei `executa()`, se transformă în litere mici conținutul textului. În clasa concretă `ComandaLitereMari`, de asemenea derivată din clasa abstractă `Comanda`, în cadrul metodei `executa()`, se transformă în majuscule conținutul textului.

```
public abstract class Comanda {
     public abstract void executa(Text text);
     public static void main(String[] args) {
        Text text = new Text("Aceasta este o carte de POO");
        ComandaLitereMici c1 = new ComandaLitereMici();
        ComandaLitereMari c2 = new ComandaLitereMari();
        c1.executa(text);
        System.out.println(text.Continut());
        c2.executa(text);
        System.out.println(text.Continut());
     }
}
class ComandaLitereMici extends Comanda {
     public void executa(Text text) {
          text.seteazaLitereMici();
     }
}
class ComandaLitereMari extends Comanda {
     public void executa(Text text) {
          text.seteazaLitereMari();
     }
}
class Text {
     private String continut;
     public Text(String continut) {
          this.continut = continut;
     }
     public String Continut() {
          return continut;
     }
```

```
    public void seteazaLitereMici() {
        this.continut = this.continut.toLowerCase();
    }
    public void seteazaLitereMari() {
        this.continut = this.continut.toUpperCase();
    }
}
```

6 Aplicații

6.1 Fabio Pizza

Implementați clasa abstractă `Pizza` cu câmpurile `dimensiune` și `pret`. Extindeți această clasă prin 3 tipuri de pizza – `DiavolaPizza`, `PepperoniPizza` și `HawaiiPizza`, fiecare având implementată metoda `toString()` care va afișa tipul de pizza, dimensiunea și prețul.

Folosind șablonul de proiectare `Factory`, implementați clasa `PizzaFactory` care va fi folosită pentru a instanția obiecte de tip `Pizza` pe baza unui `enum TipPizza` și a valorilor `dimensiune` și `pret`.

Limitați crearea mai multor obiecte de tipul `PizzaFactory` folosind șablonul `Singleton`! Pentru testare, construiți un tablou care să conțină toate tipurile de pizza cu diferite dimensiuni și prețuri, și apoi afișați-l. Folosiți metoda din clasa `PizzaFactory` pentru crearea obiectelor în locul constructorului.

6.2 Decorator

Utilizați șablonul de proiectare *Decorator* pentru parcurgerea unui text în care spațiul este separator. Metoda `parcurge()` va întoarce următorul token din textul inițial.

- Definiți interfața `ISecventa` care va declara metoda `parcurge();`
- Implementați clasa `SecventaCuvant` care implementează interfața `Secventa`, cu variabilele membru:

```
String[] cuvinte;
int index = -1;
```

Clasa `SecventaCuvant` va primi în constructor un șir de caractere ce va trebui segmentat după spațiu.
Metoda `parcurge()` va întoarce câte un cuvânt pe rând, din șirul `cuvinte`, folosindu-se de `index`.
Implementați clasa abstractă `SecventaDecorator` care implementează interfața `Secventa`, având un constructor ce primește un obiect de tip `SecventaCuvant` și metoda `parcurge()` care apelează `parcurge()` pe acel obiect.

Implementați următoarele clase ce extind `SecventaDecorator`, modificând implementarea metodei `parcurge()`:

- `SecventaCaracter` – întoarce câte o literă; metoda `parcurge()` returnează câte un caracter pe rând;
- `SecventaMajuscule` – modifică toate literele mici din cuvintele întoarse în majuscule.

Testați implementarea prin parcurgerea unor texte.

6.3 Image Command

Să se creeze o clasă numită `Imagine`, cu variabilele membru `lungime`, `latime` și `nivelLuminozitate`, toate de tipul `int`. În clasa abstractă `Comanda`, se vor declara metodele abstracte `executa()` și `anuleaza()`. Din clasa abstractă `Comanda` se vor deriva clasele concrete `ComandaRedimensionare`, `ComandaDecupare` și `ComandaLuminozitate`, care implementează metodele `executa()` și `anuleaza()`. Pentru `ComandaRedimensionare`, în cadrul metodei `executa()`, se vor mări dimensiunile imaginii (lungimea și lățimea) cu `50%`, iar în cadrul metodei `anuleaza()`, se va reveni la dimensiunile anterioare. Pentru clasa `ComandaDecupare`, în cadrul metodei `executa()` se vor decupa un număr de `20` de pixeli din lățime și `30` de pixeli din lungime, iar în cadrul metodei `anuleaza()` se va reveni la dimensiunile anterioare. Pentru clasa `ComandaLuminozitate`, în cadrul metodei `executa()` se va mări luminozitatea imaginii, iar în cadrul metodei `anuleaza()` se va reveni la luminozitatea anterioară.

Să se definească clasa `Editor`, ce reține un tablou de comenzi. Acestea vor fi executate pe rând. De asemenea, se va defini metoda `reexecuta()`, ce reexecută ultima comandă din lista de comenzi și metoda `anuleaza()`, ce anulează ultima comandă executată.

Tabloul de comenzi poate fi reprezentat cu ajutorul variabilelor:

```
int index = 0;
public Comanda[] listaComenzi = new Comanda[10];
public void adaugaComanda(Comanda comanda) {
     listaComenzi[index] = comanda;
     index++;
}
public void executaComenzi(Imagine imagine) {
     for (int i = 0; i < index; i++) {
          listaComenzi[i].executa(imagine);
     }
}
```