Proiectarea de detaliu -1

Prof. univ. dr. ing. Florica Moldoveanu

Curs *Ingineria programelor* – UPB, Automatică şi Calculatoare
2020-2021

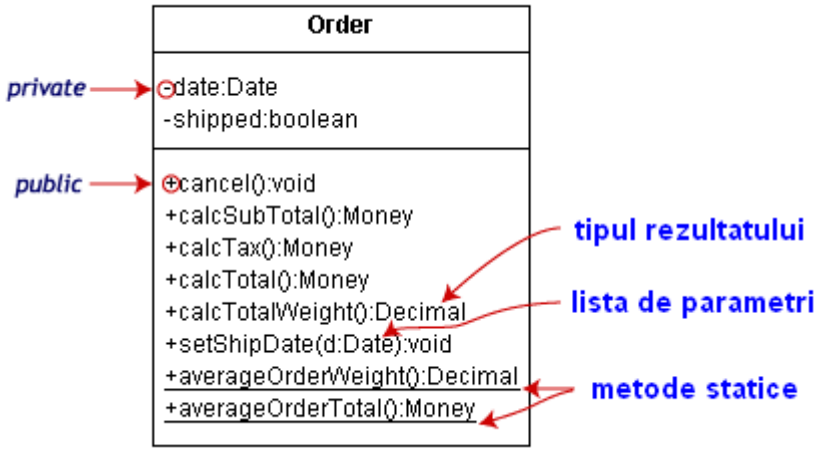
Proiectarea de detaliu

Activități:

- **Detalierea modelului arhitectural**: detalierea claselor, adaugarea de noi clase.
- **Optimizarea modelului arhitectural**: clasele existente în modelul arhitectural pot fi modificate pentru a se îmbunatăți timpul de raspuns, utilizarea memoriei, pentru a se reduce complexitatea codului, pentru a reduce cuplarea subsistemelor, pentru a obtine clase reutilizabile, s.a.
- **Reutilizarea**: folosirea soluțiilor existente.
- **Definirea interfeţelor subsistemelor** în limbajul de implementare.

Detalierea claselor şi specificarea interfețelor

Detalierea claselor



Pot fi identificate operații şi obiecte adiționale necesare pentru transferul datelor între subsisteme.

Interfeţele subsistemelor se specifică folosind interfeţele claselor care le implementeaza: operațiile, tipul parametrilor şi al valorii întoarse, tratarea excepțiilor.

Specificația interfeței furnizate de un subsistem, în limbajul de implementare, este adesea numita **Interfața de Programare a Subsistemului: subsystem API (Application Programming Interface)**

Recomandari pentru proiectarea de detaliu a claselor (1)

❖ Derivate din experienta mai multor proiecte industriale (**Lorenz,**1993)

- Dimensiunea medie a unei metode (Lines Of Code - LOC):

< 24 LOC pentru programe C++

- Numarul mediu de metode/clasa:

< 20

un numar mediu mai mare indica prea multa responsabilitate in putine clase

- Numarul mediu de variabile instanta (atribute) / clasa:

< 6 ; mai multe inseamna ca o clasa “face” prea mult.

- Numarul mediu de linii comentariu / metoda: >1

- Adancimea arborelui de mostenire (Depth of Inheritance Tree):

< 6, incepand de la radacina sau de la clasele bibliotecii de dezvoltare

Recomandari pentru proiectarea de detaliu a claselor (2)

- Numarul de relatii clasa - clasa in fiecare subsistem:

- trebuie sa fie relativ mare → coeziune in interiorul fiecarui subsistem
- daca o clasa dintr-un subsistem nu interactioneaza cu multe alte clase din subsistem→ ar trebui plasata in alt subsistem.

- Numarul de relatii subsistem - subsistem:

< numarul de relatii clasa - clasa din fiecare subsistem

- Utilizarea atributelor:

daca grupuri de metode dintr-o clasa utilizeaza seturi diferite de atribute

→ clasa ar putea fi divizata in mai multe clase

(sa existe coeziune prin date la nivelul fiecarei clasei)

- Numarul de reutilizari ale unei clase:

daca o clasa nu poate fi reutilizata in diferite aplicatii (mai ales o clasa abstracta)

→ ar putea fi necesar sa fie reproiectata

Reutilizarea

Reutilizare: identificarea soluțiilor existente

La mai multe niveluri:

- Reutilizarea unor componente existente: se pot folosi c**omponente “of-the-shelf”** (componente binare) pentru implementarea componentelor din arhitectura sistemului.
- Reutilizarea unor soluții de proiectare: **sabloane de proiectare**
- Reutilizarea codului:

  - Reutilizarea codului unei clase existente: **mostenire**

  - Reutilizarea unei operații dintr-o clasă existentă: **delegare**

Adesea, subsistemele definite trebuie ajustate pentru folosirea componentelor “of-the-shelf” iar şabloanele de proiectare trebuie să fie adaptate pentru a fi utilizate în realizarea subsistemelor. Adaptarea se poate efectua prin clase “wrapper” sau utilizând moştenirea şi delegarea.

Sabloane de proiectare(1)
(Design patterns)

Ce este un şablon de proiectare?

- Un şablon de proiectare descrie o problemă care se întalneşte în mod repetat în proiectarea programelor şi soluția generală pentru problema respectivă, astfel încât să poată fi utilizată oricând dar nu în acelaşi mod de fiecare data.
- Soluția este exprimată folosind clase şi obiecte.
- Atat descrierea problemei cât şi a solutiei sunt abstracte astfel încat să poata fi folosite în multe situatii diferite.
- **Cartea de referinta**: “Design Patterns - Elements of Reusable Object-Oriented Software”, cunoscuta si sub numele “Gang of Four” [1]

Sabloane de proiectare(1)

Un sablon de proiectare are 4 elemente esentiale:

1. **Un nume** prin care poate fi referit →se creaza un vocabular de proiectare.
2. **Descrierea problemei**. Se explica problema pentru care ar trebui aplicat sablonul si contextul in care apare.
3. **Descrierea solutiei**: elementele care compun şablonul (clase, interfețe, obiecte), relațiile dintre ele, responsabilitatile si colaborarile.
4. **Consecintele aplicarii sablonului**: permit evaluarea alternativelor de proiectare, intelegerea costurilor si a beneficiilor aplicarii unui sablon.

Alte elemente: motivatia, aplicabilitatea**,** probleme de implementare, exemple de cod.

Șabloane de proiectare(2)

Clasificare - bazata pe scopul sabloanelor:

1. Creaționale (Creational patterns)

- Definesc **mecanisme de creare a obiectelor** adecvate unor anumite situații.

Exemple: Singleton, Abstract factory, Factory method, Builder, Prototype

2. Structurale (Structural patterns)

- Definesc **moduri de asamblare a claselor** pentru a obține structuri flexibile şi eficiente, adecvate unor anumite situații.

Exemple: Adapter, Bridge, Composite, Decorator, Façade

3. Comportamentale (Behavioral patterns)

- Definesc **şabloane de comunicare între obiecte**, care cresc flexibilitatea în realizarea comunicării.

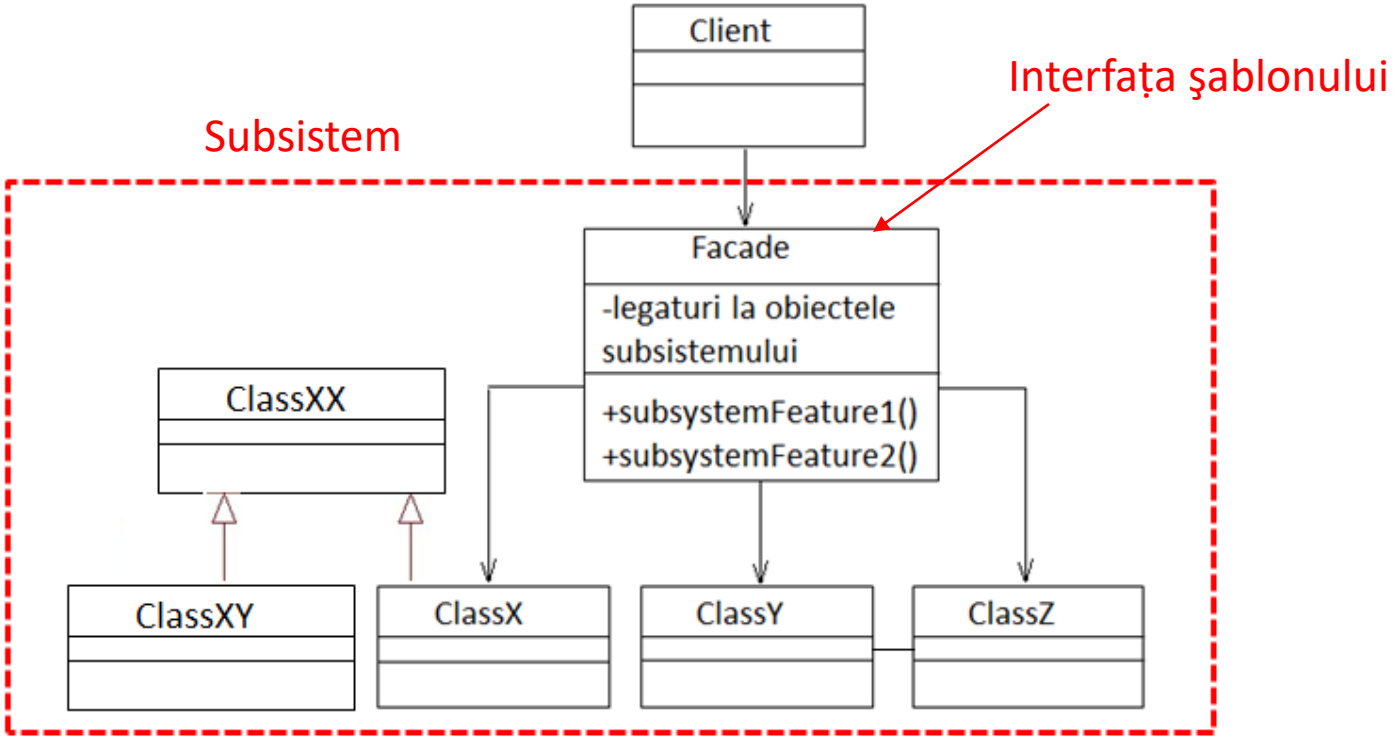
Exemple: Command, Mediator, Strategy, Observer, Visitor

Șablonul Façade (1)

Nume: Façade; **tipul**: structural

Descrierea problemei: simplificarea comunicării clienților unui subsistem complex cu obiectele subsistemului.

Soluția: Clasa *Façade* furnizează o interfață simplă pentru un sistem complex, oferind numai serviciile necesare clienților subsistemului.



Șablonul Facade (2)

Consecințe:

- Clientul apelează operațiile clasei Façade în loc să apeleze operațiile claselor subsistemului.
- Clientul nu cunoaşte clasele subsistemului. Dependențele dintre Client şi subsistem sunt minimizate. Clasele subsistemului pot fi modificate fara a afecta clientul.
- Clasele subsistemului nu cunosc existenta clasei Façade. Ele coopereaza în cadrul subsistemului pentru a realiza functionalitatile oferite de subsistem prin clasa Façade.

- Un subsistem poate oferi mai multe clase Façade, pentru tipuri diferite de clienți.

Clasa *Client* acceseaza şablonul.

Clasele *Client* pot fi clase existente într-o bibliotecă de clase sau clase noi ale sistemului în dezvoltare.

Interfața unui şablon este partea şablonului vizibilă clasei client. Poate fi o clasă, o clasă abstractă sau o interfață.

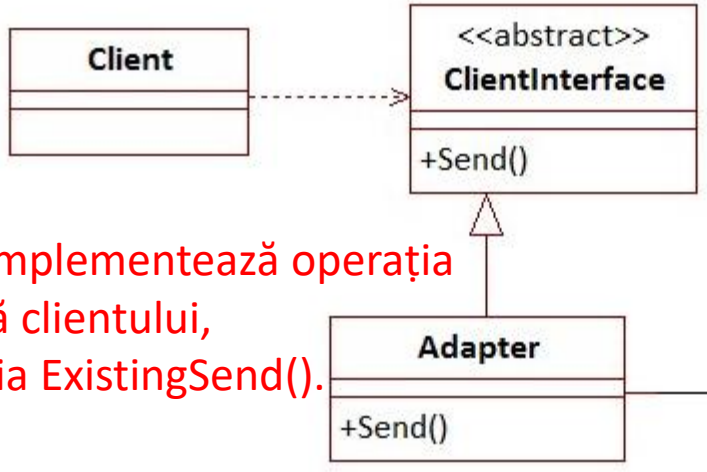
Interfața şablonului Façade este clasa Façade.

Șablonul Adapter (1)

Nume: Adapter; **tipul**: structural

Descrierea problemei: utilizarea în implementarea unui sistem (Client) a unei clase/componente existente, care furnizează o interfață diferită de interfața necesară Clientului.

Soluția: Clasa *Adapter* implementează interfața necesară Cllientului delegând cererile *clientului* către clasa existentă (*LegacyClass*), după efectuarea conversiilor necesare asupra structurilor de date sau a comportamentului, astfel încat *Adapter* oferă comportamentul aşteptat de client.



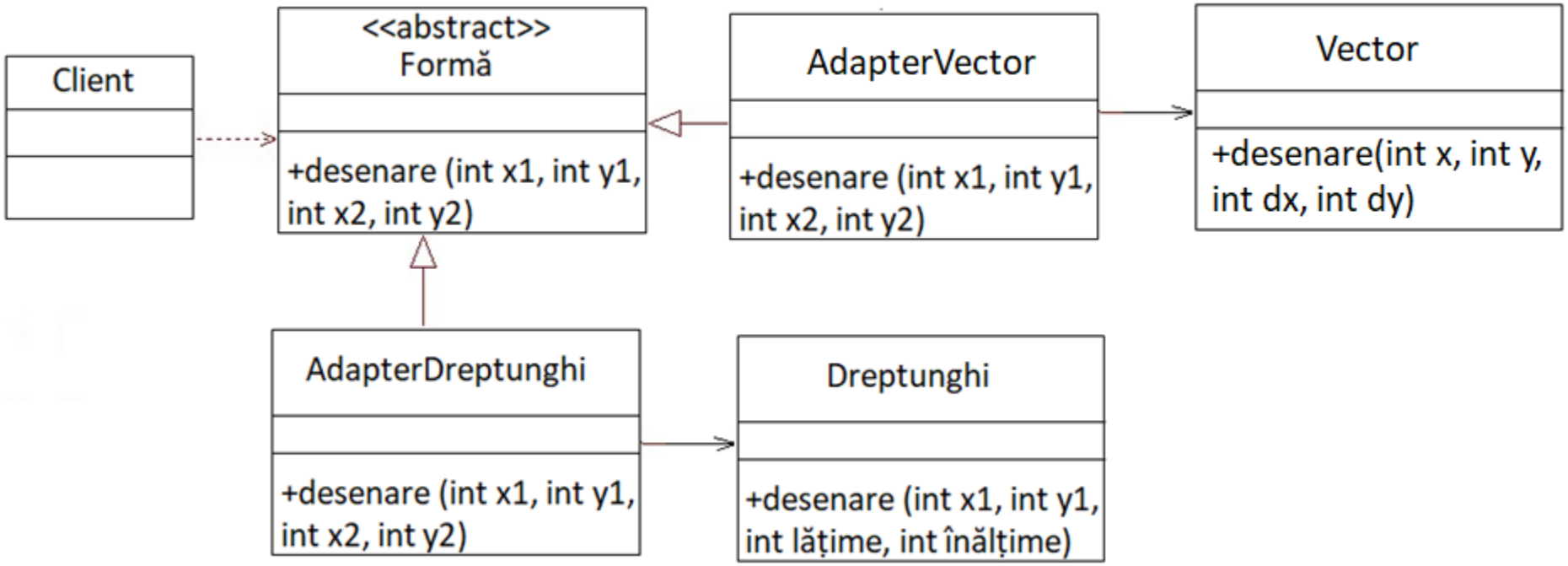
Interfața necesară clientului

Clasa Adapter implementează operația Send() necesară clientului, folosind operația ExistingSend().

Clasa existentă implementează operația ExistingSend()

Şablonul Adapter (2)

Exemplu de aplicare a şablonului



Șablonul Adapter (3)

Consecințe:

- Clasele Client si LegacyClass pot fi folosite împreuna fara modificarea niciuneia.
- Clasa Adapter se poate folosi împreună cu LegacyClass şi toate subclasele sale.
- **Pentru fiecare specializare (subclasa) a clasei ClientInterface trebuie creata o clasa Adapter.**

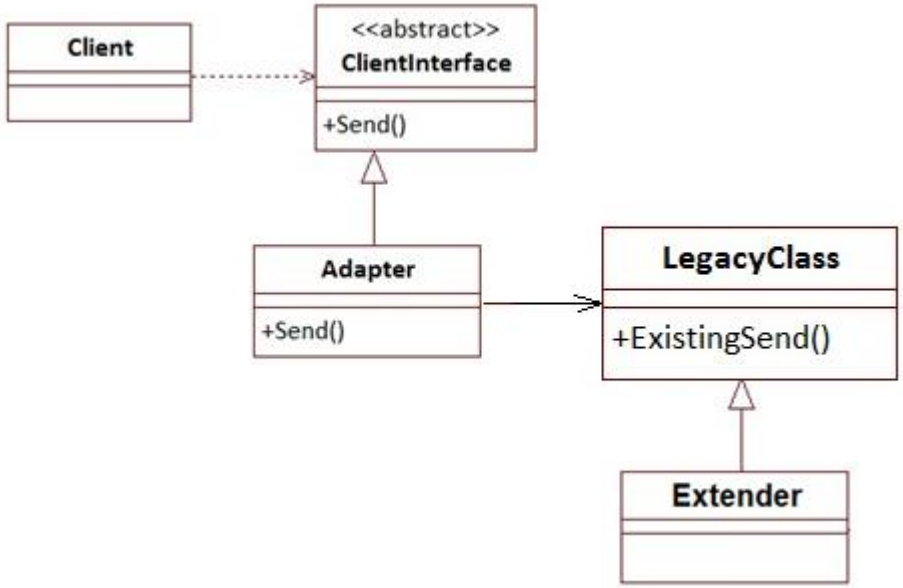
- **Interfata şablonului** este clasa abstractă *ClientInterface*.

  Clasele care implementează comportamentul sablonului Adapter sunt *LegacyClass* si *Adapter.*

- Intr-un sablon de proiectare pot exista si **clase *Extender***, care specializeaza o clasa de implementare pentru a furniza o implementare diferita sau a extinde comportarea sablonului.

Șablonul Adapter (4)

- In sablonul **Adapter**, subclasele clasei *LegacyClass* sunt clase *Extender.*



❖ Sablonul **Adapter**

- permite reutilizarea deoarece nici interfața necesară clientului (*ClientInterface*), nici clasa existentă (*LegacyClass*) nu trebuie sa fie modificate.

- încurajează extensibilitatea: sablonul poate fi extins cu oricate clase *Extender* fara a fi necesara modificarea clasei *Adapter.*

Șablonul Bridge (1)

Nume: Bridge; **tipul**: structural

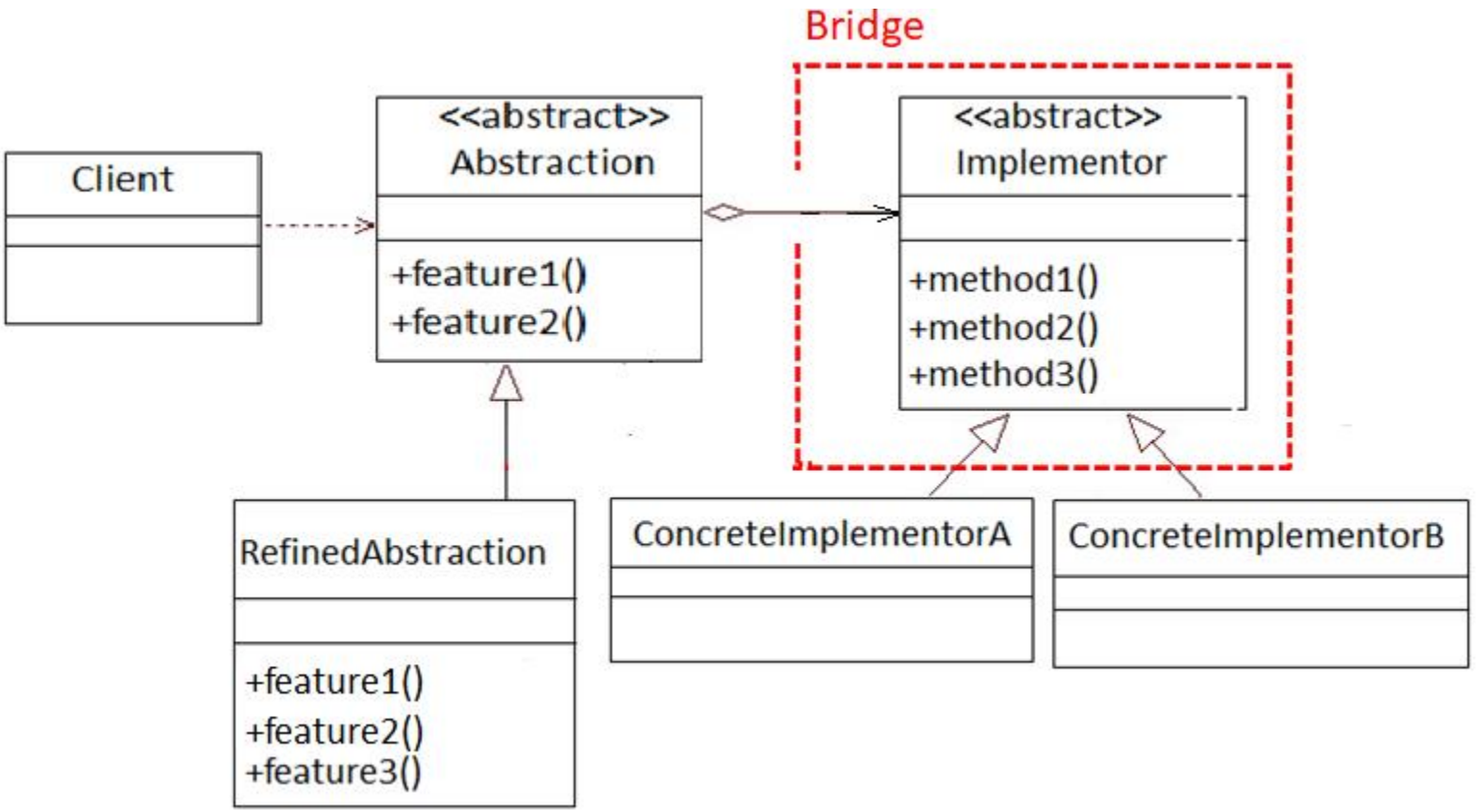
Descrierea problemei:

- Decuplarea unei abstracții de o implementare a sa, astfel încât ambele să poată fi modificate independent.
- Separarea unui set de clase corelate în două ierarhii de clase – *Abstraction* şi *Implementation* – care pot fi dezvoltate independent una de cealaltă.

Soluția: decuplarea se efectuează printr-o interfață care acționează ca un “pod” între abstracție şi implementările sale concrete

Șablonul Bridge (2)

- Clasa *Abstraction* defineşte interfața vizibilă clientului.
- *Implementor* este interfața care defineşte metodele necesare implementării clasei Abstraction.
- *RefinedAbstraction* implementează şi rafinează interfața *Abstraction.*
- Codul clientului leagă un obiect *RefinedAbstraction* la un obiect *ConcreteImplementor.*



Șablonul Bridge (3)

Consecințe

— Clientul este protejat de implementările concrete ale interfețelor *Abstraction* şi *Implementor*

— Ierarhiile *Abstraction* şi *Implementor* pot fi dezvoltate şi modificate independent.

Exemplu de aplicare: Clientul este o aplicație multi-platformă cu mai multe interfețe utilizator.

- *Abstraction* poate fi interfața grafică cu utilizatorul (GUI).
- *Implementor* poate fi API-ul sistemului de operare folosit de nivelul GUI – defineşte funcțiile apelate de implementarea GUI ca răspuns la interacțiunea cu utilizatorul.

Avantajele aplicarii şablonului: aplicatia Client poate fi extinsa:

- cu noi GUI - ierarhia GUI poate fi extinsă şi modificată independent de clasele de implementare (se adaugă/modifică clasele *RefinedAbstraction*)

- pentru a putea rula pe mai multe sisteme de operare- se adauga subclase *ConcreteImplementor*

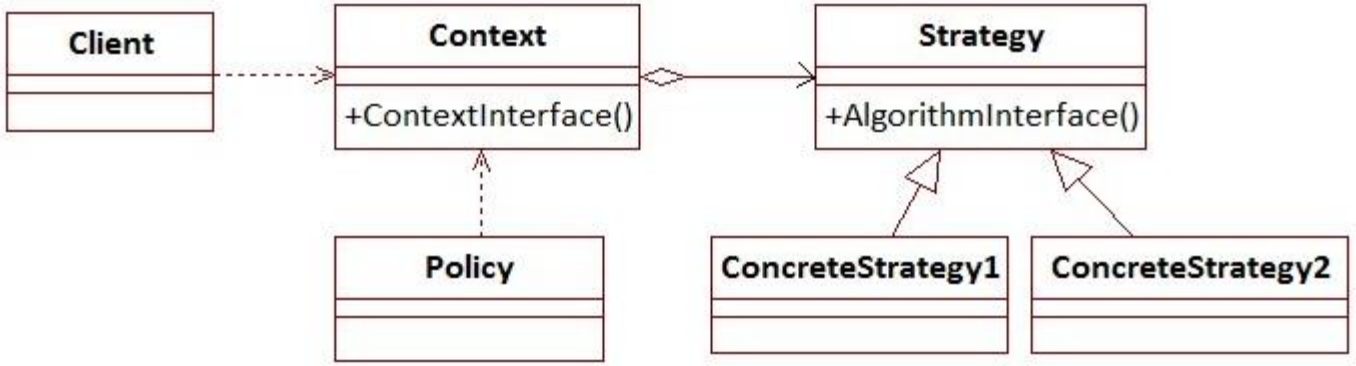
Șablonul Strategy (1)

Nume: Strategy; **tipul**: comportamental

Descrierea problemei: **Decuplarea unei clase care decide o politica de clasele care implementeaza diferite strategii**, astfel incat implementarile strategiilor sa poata fi inlocuite in mod transparent clientului.

Solutia: Un client acceseaza serviciile furnizate de o clasa Context, care sunt realizate de unul sau mai multe mecanisme, asa cum se decide intr-un obiect Policy.

Clasa Strategy descrie interfata comuna tuturor mecanismelor care pot fi utilizate de Context. Un obiect Policy creaza un obiect ConcreteStrategy si configureaza obiectul Context pentru a-l utiliza.



Șablonul Strategy (2)

Consecinte:

- Obiectele ConcreteStrategy pot fi inlocuite in mod transparent de Context.
- Obiectul Policy decide care strategie este mai buna in functie de cerinte (de ex., compromis intre viteza si spatiu de memorie, etc.)
- Pot fi adaugati noi algoritmi fara a modifica interfata sablonului (Context)

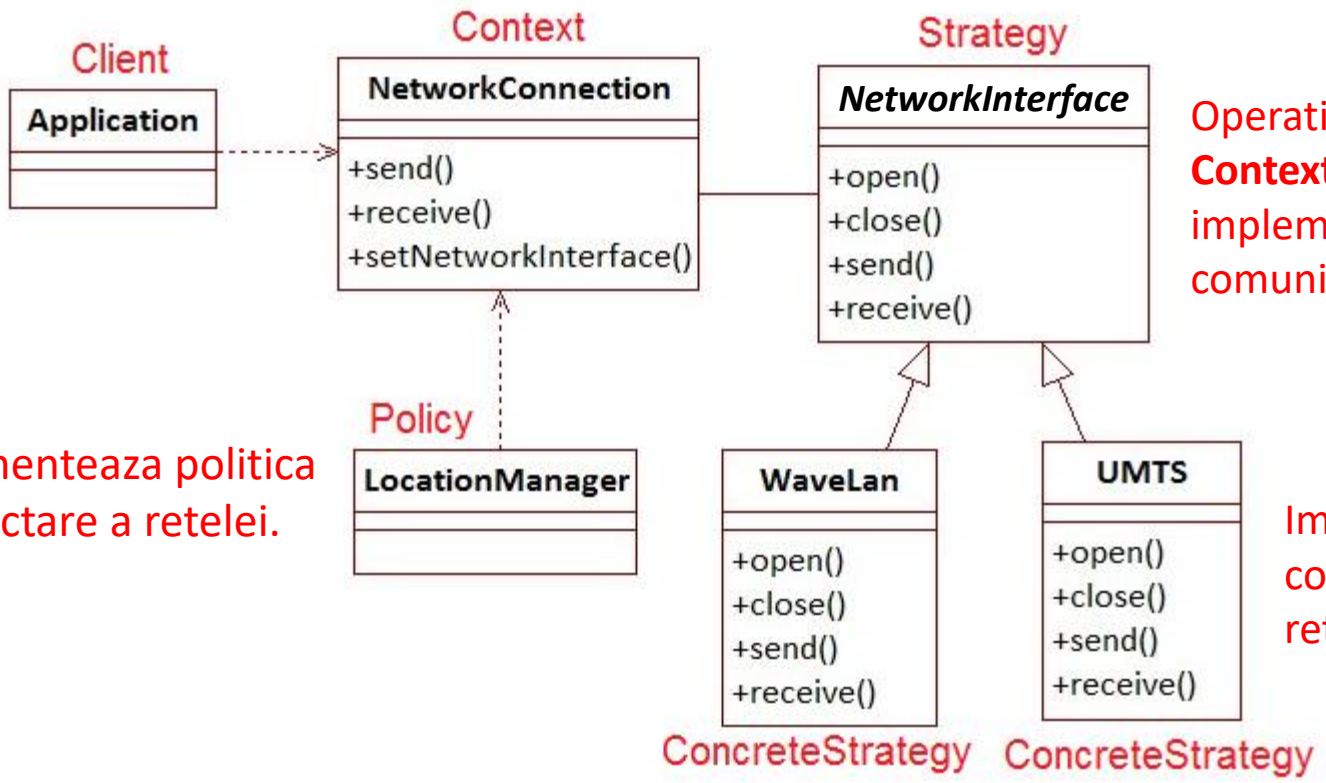
Sabloane corelate: Adapter si Bridge.

Exemplu: o aplicatie mobila trebuie sa schimbe in mod dinamic reteaua de comunicatie in functie de diferite criterii (locatia utilizatorului, costul comunicatiei, etc.).

Aplicatia trebuie sa nu depinda de reteaua de comunicatie folosita la un moment dat: are nevoie de o interfata comuna pentru diferitele retele.

Pentru a separa politica de selectare a retelei de interfata cu reteaua, se incapsuleaza implementarile protocoalelor de acces la retea folosind sablonul Strategy.

Șablonul Strategy (3)



- **LocationManager** configureaza dinamic **NetworkConnection** (**setNetworkInterface()**) cu un obiect **ConcreteStrategy** pe baza locatiei curente.
- Aplicatia utilizeaza **NetworkConnection** in mod independent de reteaua de comunicatie.

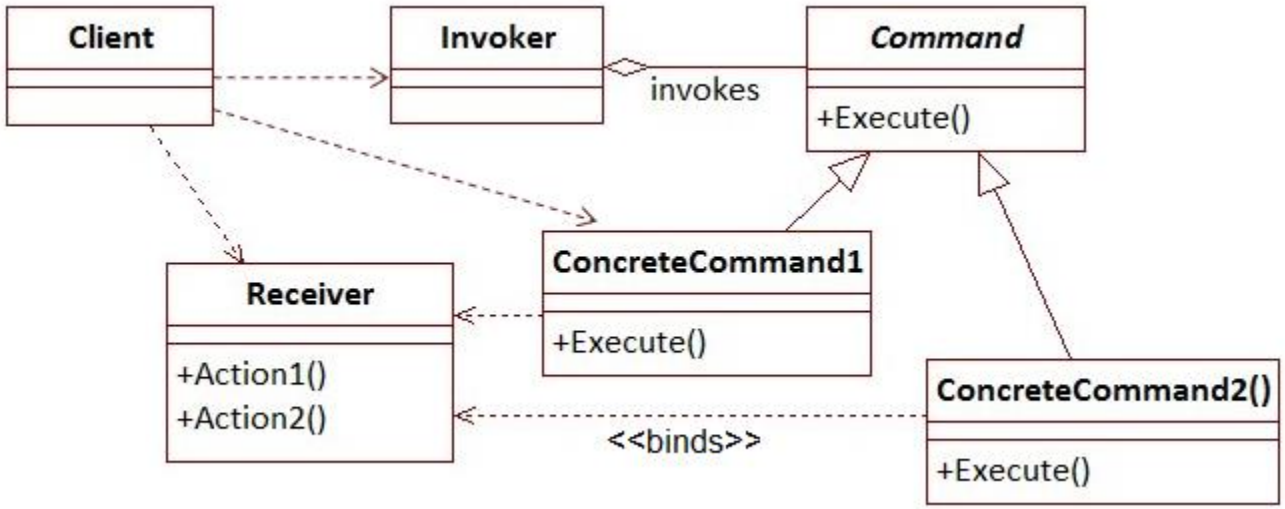
Șablonul Command (1)

Nume: Command; **tipul**: comportamental

Descrierea problemei: Separarea comenzilor de procesarea lor, astfel încat ele sa poata fi executate, anulate sau memorate într-o coada si procesate ulterior, independent de comanda.

Solutia: Clasa abstracta *Command* declara interfata care trebuie implementata de toate comenzile concrete (ConcreteCommands).

- Clientul creaza comenzi concrete si le "leaga" la obiecte *Receiver* specifice.
- Comenzile concrete sunt implementate folosind operatiile obiectului Receiver.
- *Invoker* executa sau anuleaza o comanda.

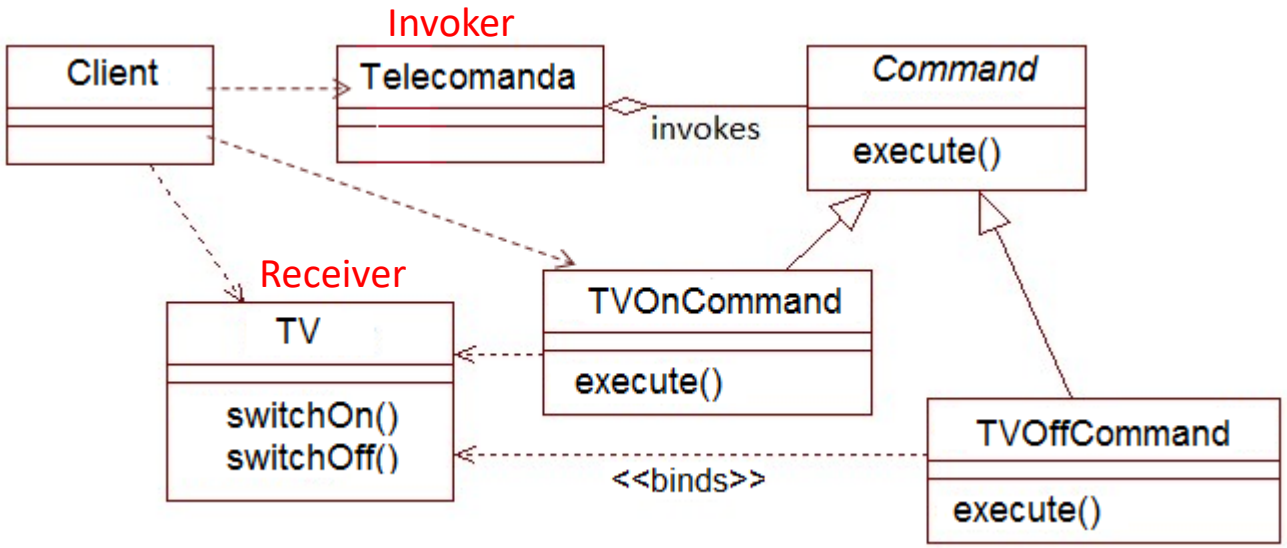


Șablonul Command (2)

Consecinte: Obiectul comenzii (Receiver) si algoritmul comenzii (ConcreteCommand) sunt decuplate.

- Invoker nu cunoaste comenzi specifice.
- ConcreteCommands sunt obiecte care pot fi create si memorate.
- Pot fi adaugate noi obiecte ConcreteCommands (noi functionalitati) fara modificarea codului existent.

Șablonul Command (3) – exemplu




```
//Command
public interface Command
{
   public void execute();
}
```

ConcreteCommand

```
//Comanda concreta „TVOn”
public class TVOnCommand implements Command
{  //referinta la un obiect Receiver
   TV tv;
   //  se leaga comanda concreta la un obiect Receiver
   public TVOnCommand (TV tv )
   {   this.tv = tv; }
   public void execute()
   {   tv.switchOn();  }
}
```

```
//Comanda concreta „TVOff”
public class TVOffCommand implements Command
{  //referinta la un obiect Receiver
   TV tv;
   //  se leaga comanda concreta la un obiect Receiver
   public TVOffCommand(TV tv )
   {   this.tv = tv; }
   public void execute()
   {   tv.switchOff();  }
}
```

Șablonul Command (4) – exemplu

```
//TV este clasa Receiver
 public class TV
{   private boolean on;
   public void switchOn()
   { on = true; }
   public void switchOff()
   { on = false; }
}
```

```
// Invoker: telecomanda
public class Telecomanda
{   private Command comanda;
   public void setCommand (Command comanda)
   { this.comanda = comanda; }
   public void pressButton()
   { comanda.execute(); }
}
```

```
// Client
public class Client
{   public static void main(String[] args)
   {
      Telecomanda control = new Telecomanda();
      TV myTV = new TV();
      Command TVOn = new TVOnCommand(myTV);
      Command TVOff = newTVOffCommand(myTV);
      //switch on
         control.setCommand(TVOn);
      //executa comanda
         control.pressButton();

      //switch off
         control.setCommand(TVOff);
         control.pressButton()
   }
}
```

Lecturi suplimentare

1. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns - Elements of Reusable Object-Oriented Software, cunoscuta si sub numele “Gang of Four” (GoF), Addison Wesley,1994.
2. https://sourcemaking.com/design_patterns
3. https://refactoring.guru/design-patterns