



UNIVERSITATEA  
DIN BUCUREȘTI

# Metode de dezvoltare software

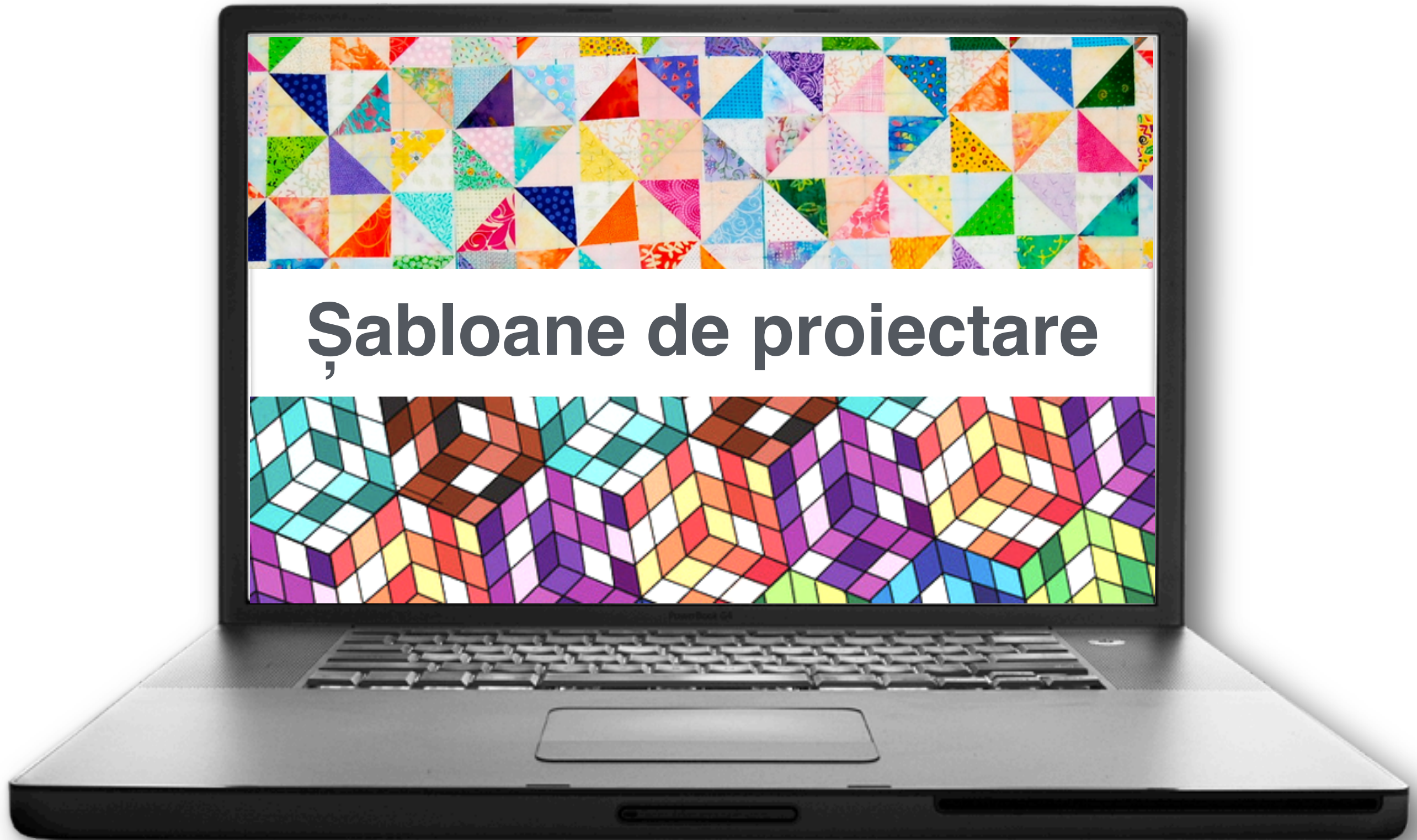
---

Șabloane de proiectare

22.03.2022

Alin Ștefănescu





Prezentare bazată pe materiale de Matthias Keil - Univ. of Freiburg și altele

# Probleme și soluții... în practică



# Șabloane de proiectare - design patterns

- șabloane de proiectare = **soluții** generale reutilizabile la **probleme** care apar frecvent în proiectare (orientată pe obiecte)
- engl. “**design patterns**”

# Șabloane de proiectare

- originare în **ingineria construcțiilor și arhitectură**
  - Christopher Alexander, arhitect
    - *A Pattern Language: Towns, Buildings, Construction* (1977)
  - [http://en.wikipedia.org/wiki/Pattern\\_language](http://en.wikipedia.org/wiki/Pattern_language)
- un **șablon** este **suficient de general** pentru a fi aplicat în mai multe situații, dar **suficient de concret** pentru a fi util în luarea deciziilor

# Șabloane software

În ingineria software:

- un **șablon** este o soluție la o problemă într-un context, unde:
  - **contextul**: situațiile recurente în care se aplică șablonul
  - **problema**: scopurile și constrângerile
  - **soluția**: regula de proiectare



# Avantaje și dezavantaje

Șabloanele sunt **folositoare** în următoarele feluri:

- ca mod de a învăța practici bune
- aplicarea consistentă a unor principii generale de proiectare
- ca vocabular de calitate de nivel înalt (pentru comunicare)
- ca autoritate la care se poate face apel
- în cazul în care o echipă sau companie adoptă propriile șabloane: un mod de a explicita cum se fac lucrurile acolo

Însă trebuie **folosite cu grijă** deoarece:

- sunt folositoare **doar dacă** există într-adevăr problema pe care ele o rezolvă
- pot crește complexitatea și scădea performanța

# Diverse categorii de șabloane

Există diverse tipuri de șabloane software:

## ■ șabloane arhitecturale

- la nivelul arhitecturii (d.ex. MVC, publish-subscribe, etc.)
- [http://en.wikipedia.org/wiki/Architectural\\_pattern](http://en.wikipedia.org/wiki/Architectural_pattern)

## ■ șabloane de proiectare

- la nivelul **claselor**/modulelor
- [http://en.wikipedia.org/wiki/Software\\_design\\_pattern](http://en.wikipedia.org/wiki/Software_design_pattern)



în restul cursului

## ■ idiomuri

- la nivelul limbajului
- [http://en.wikipedia.org/wiki/Programming\\_idiom](http://en.wikipedia.org/wiki/Programming_idiom)



# Câteva principii de bază

- programare folosind multe interfețe:
  - interfețe și clase abstracte pe lângă clasele concrete
  - framework-uri generice în loc de soluții directe
- se urmărește decuplarea:
  - obiecte cât mai independente
  - obiecte “ajutătoare”
- vezi și principiile SOLID:
  - [http://en.wikipedia.org/wiki/SOLID\\_\(object-oriented\\_design\)](http://en.wikipedia.org/wiki/SOLID_(object-oriented_design))

# După 25 de ani...

## Originile:

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (“Gang of Four”, GoF)
  - “Elements of Reusable Object-Oriented Software” (1994)
- au propus **23 de șabloane** împărțite în **trei clase**
  - șabloane **creaționale**: instanțierea
  - șabloane **structurale**: compunerea
  - șabloane **comportamentale**: comunicarea
- există bineînțeles **multe alte șabloane**, dar cele 23 de mai sus sunt “**clasice**”

# Șabloanele

## ■ Creționale:

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

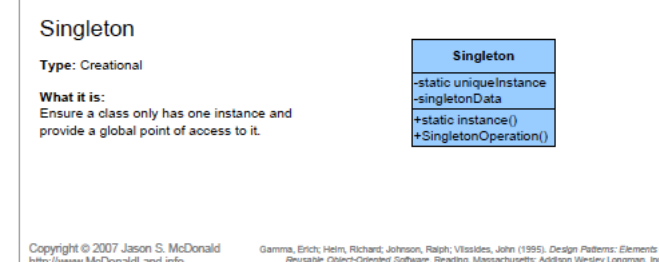
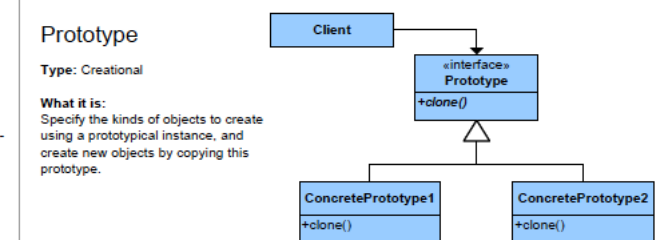
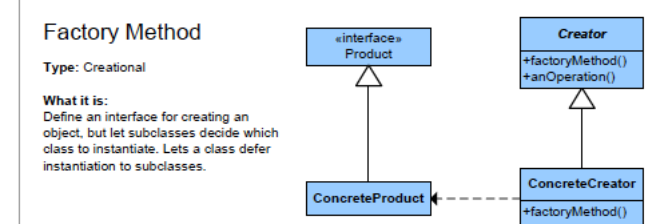
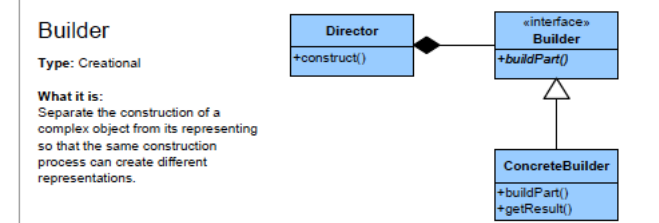
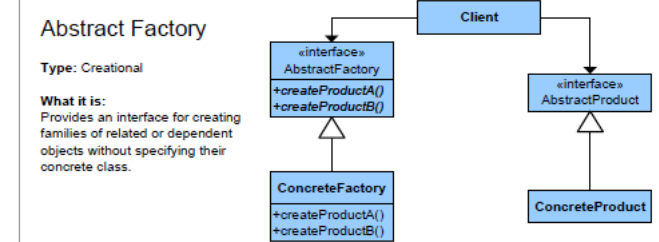
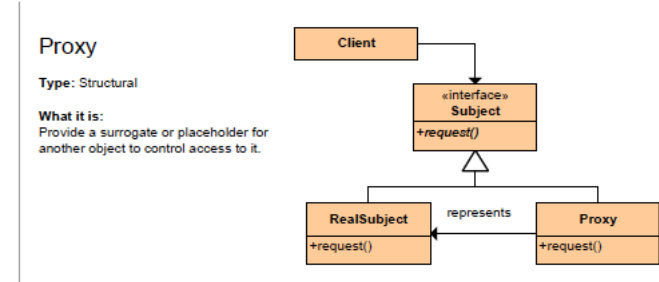
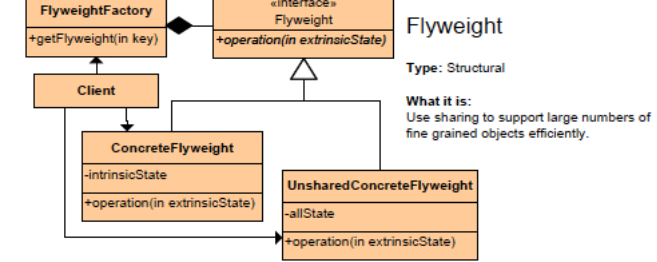
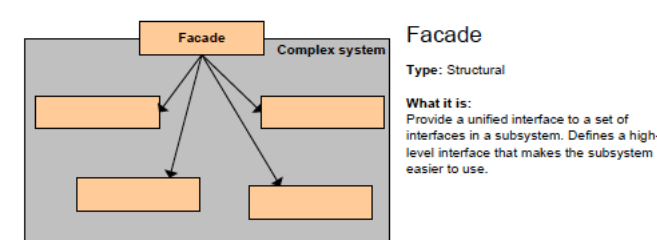
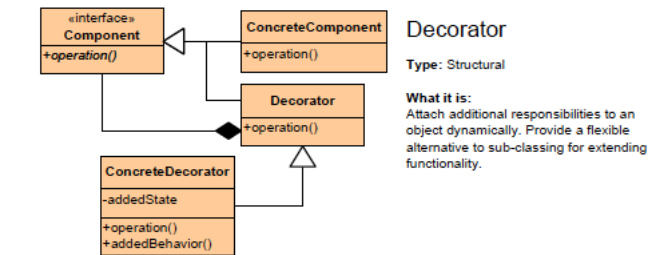
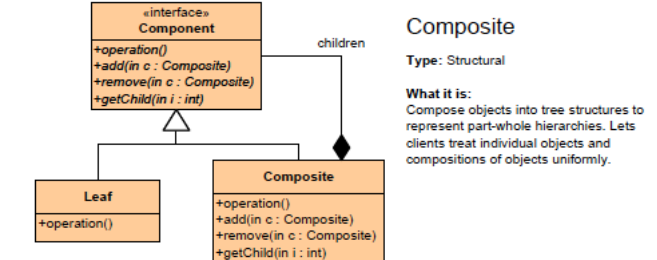
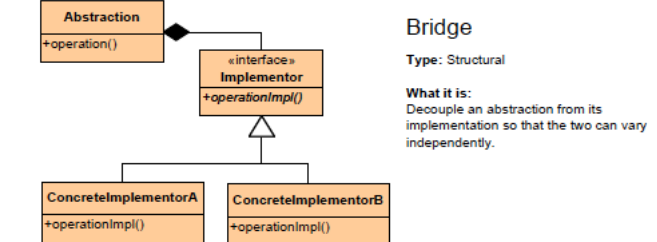
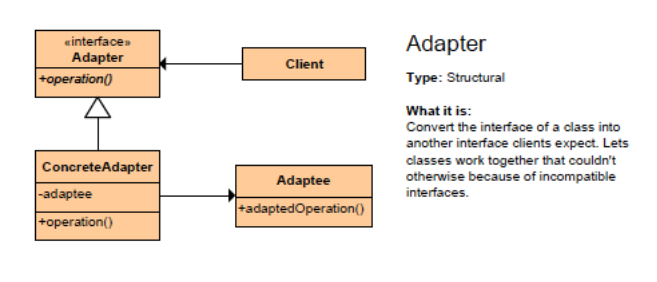
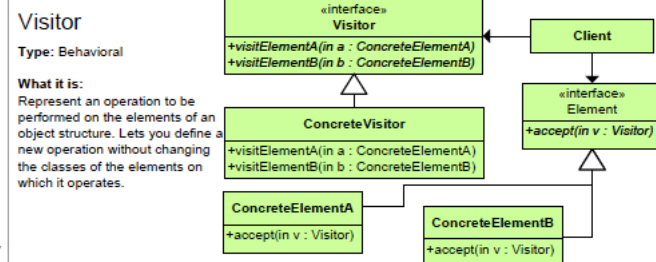
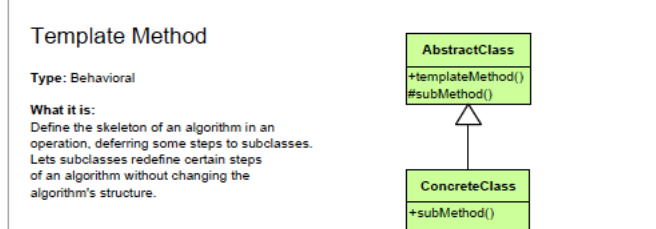
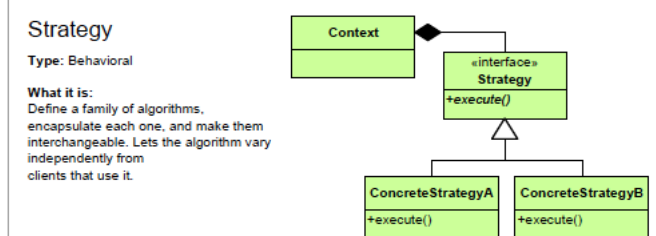
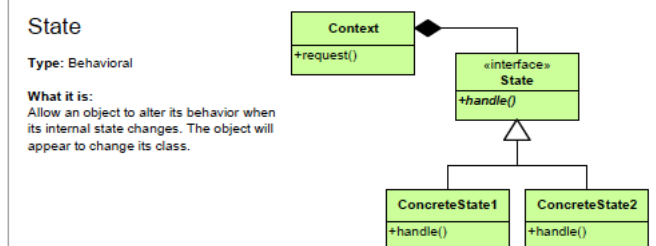
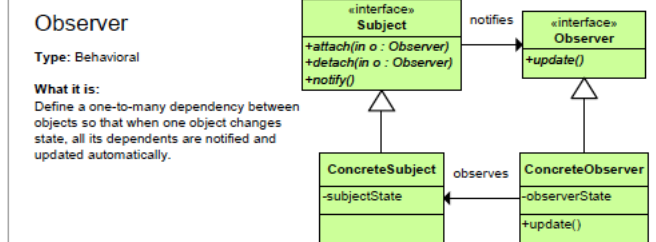
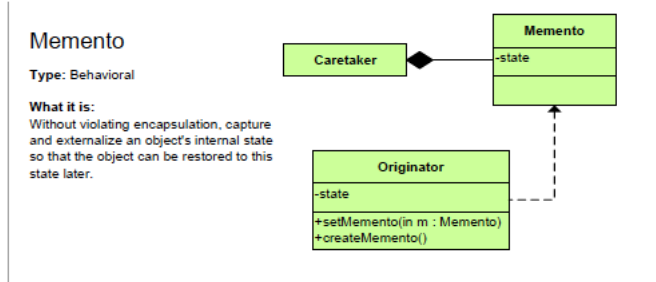
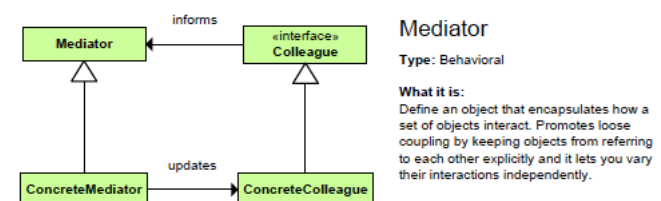
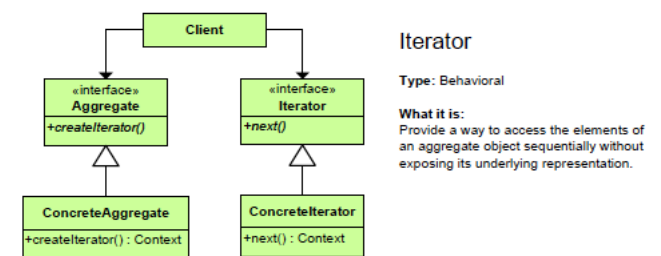
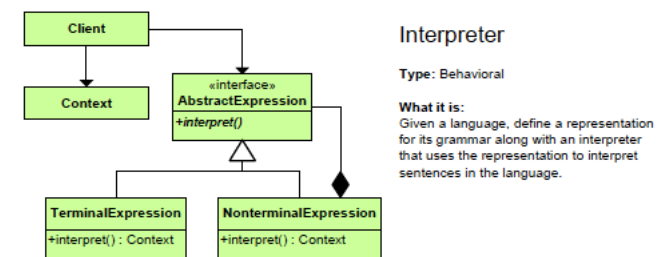
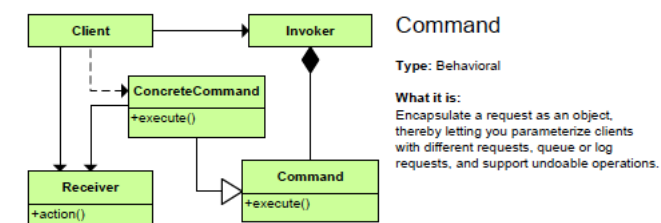
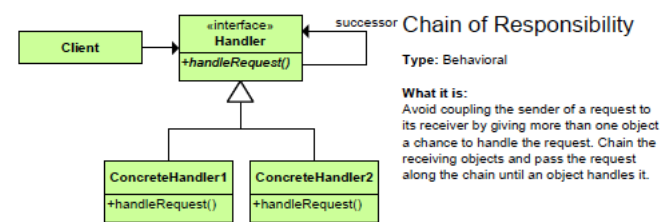
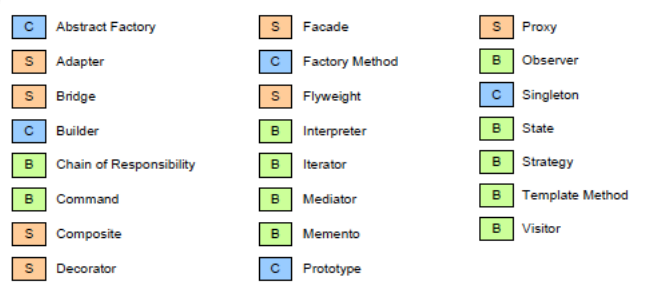
## ■ Structurale:

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

## ■ Comportamentale:

- Chain of responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template method
- Visitor

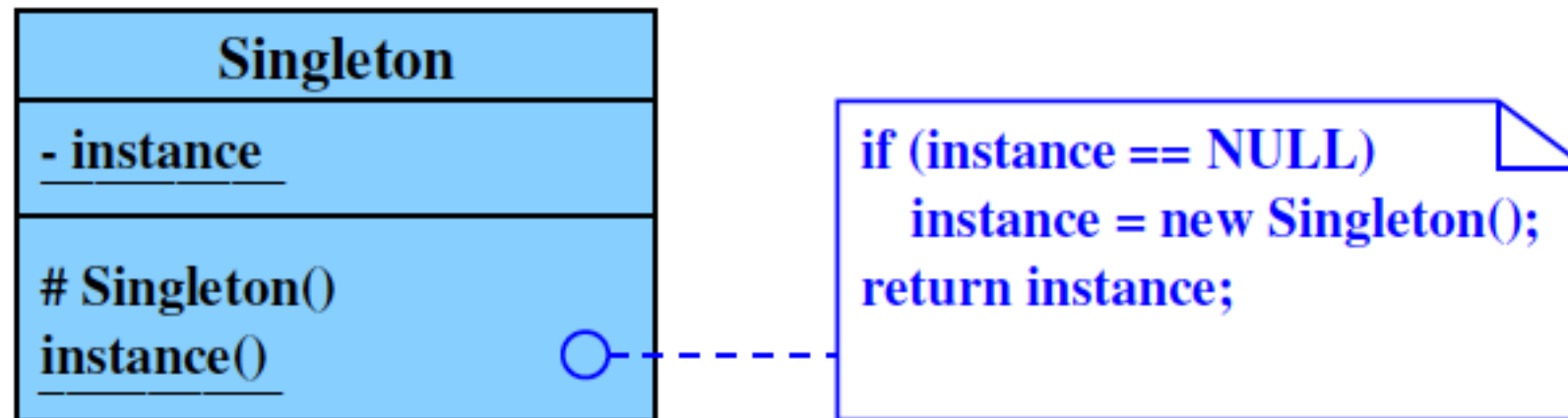




# Șablonul creațional: Singleton

- uneori este important de a avea doar **o singură instanță** pentru o clasă.
  - de exemplu, într-un sistem ar trebui să existe un singur manager de ferestre sau doar un sistem de fișiere.
- **Singleton** este un șablon simplu:
  - implică numai o singură clasă
  - ea este responsabilă pentru a se instanția
  - ea asigură că se creează **maxim o instanță**
  - în același timp, oferă un **punct global de acces** la acea instanță
  - în acest caz, aceeași instanță poate fi utilizată de oriunde, fiind imposibil de a invoca direct constructorul de fiecare dată.

# Singleton



```
class Singleton
{
    private static Singleton instance;

    protected Singleton()
    {...}

    public static synchronized Singleton instance()
    {
        if (instance == null)
            instance = new Singleton();
        return instance;
    }
    ...
}
```



# Singleton

## ■ aplicabilitate:

- când doar un obiect al unei clase e necesar
- când instanța este accesibilă global
- este folosit în alte șabloane (*factories* și *builders*)

## ■ consecințe:

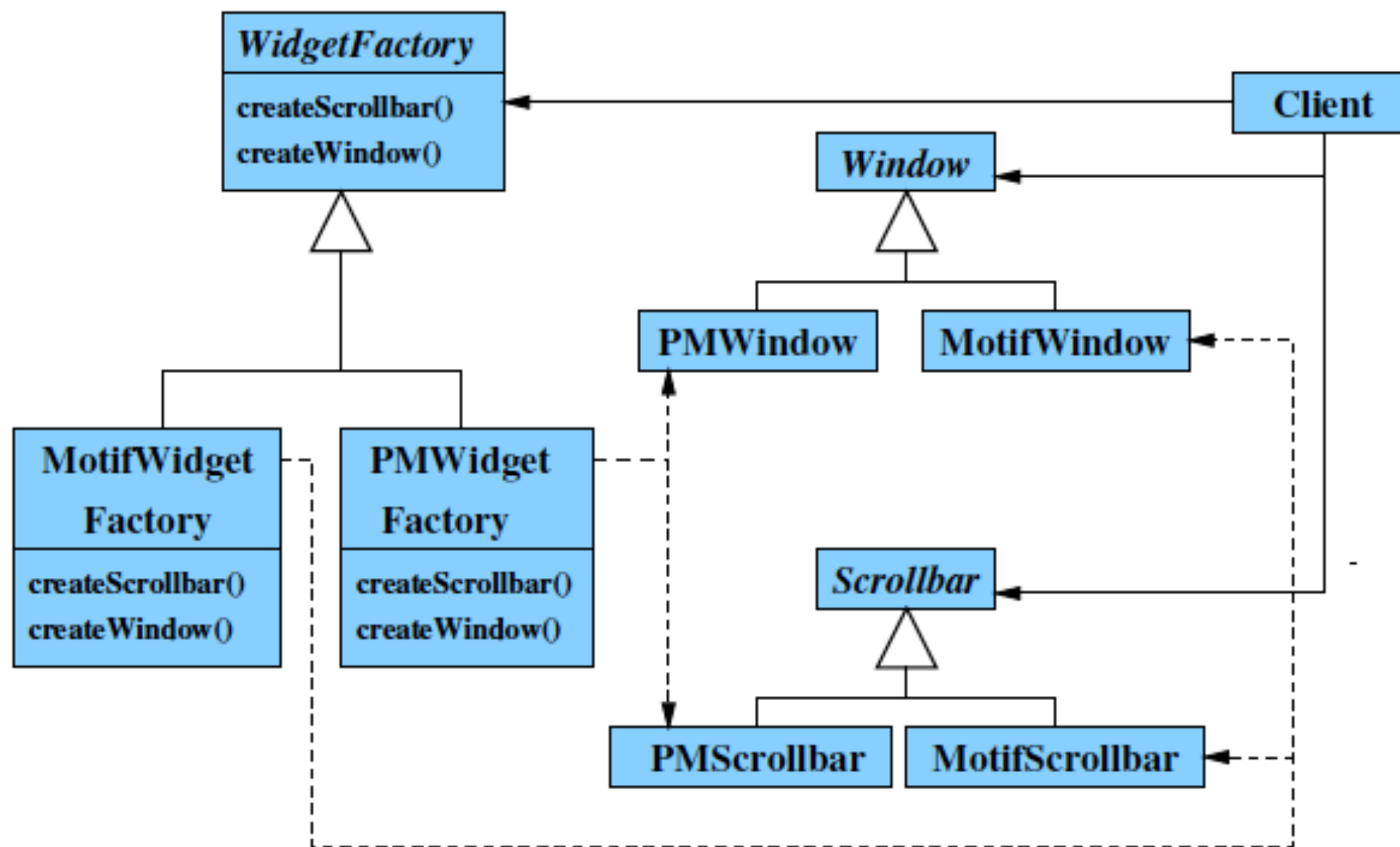
- accesul controlat la instanță
- spațiu de adresare structurat (comparativ cu o variabilă globală)

## ■ v. și:

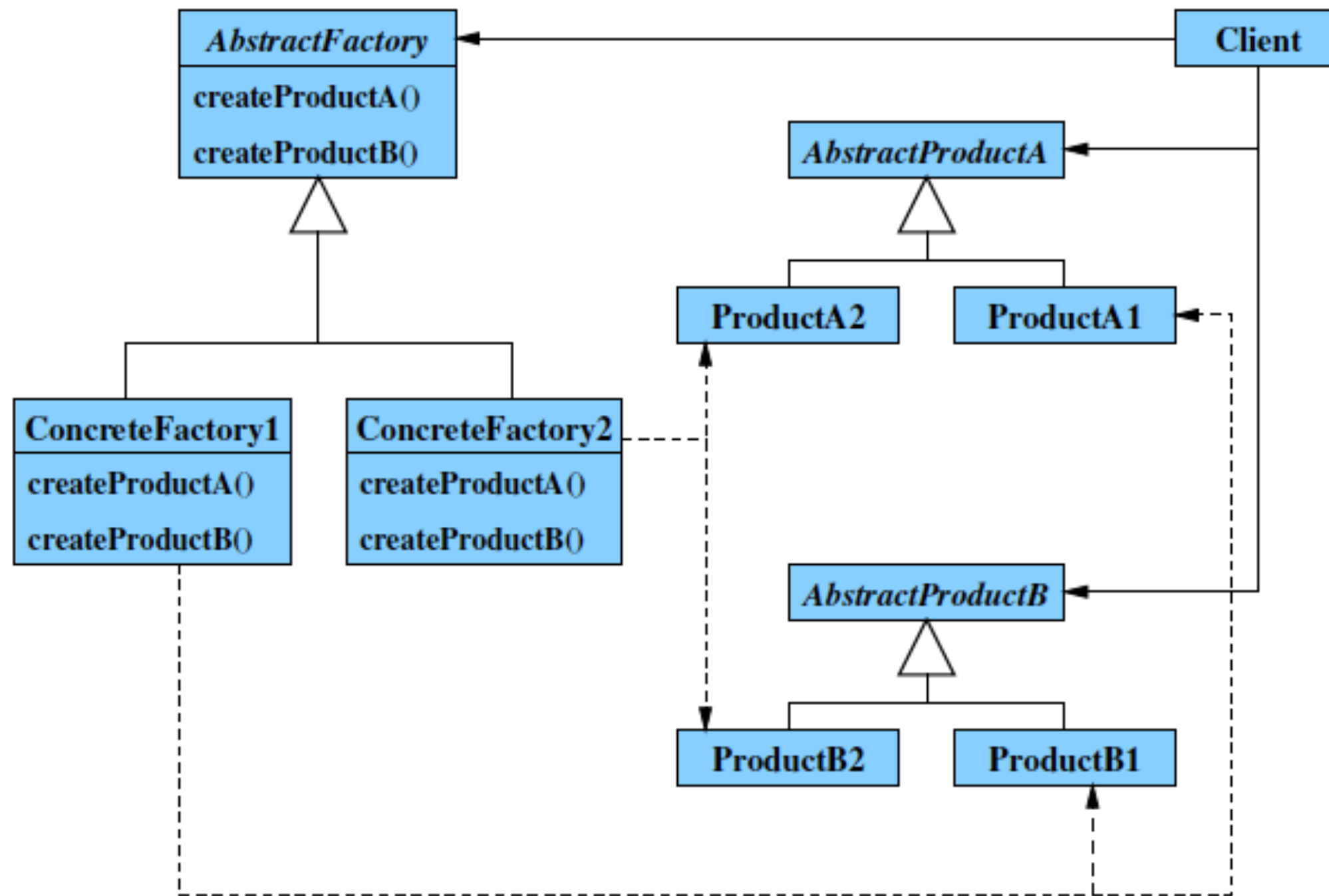
- [http://en.wikipedia.org/wiki/Singleton\\_pattern](http://en.wikipedia.org/wiki/Singleton_pattern)
- <https://refactoring.guru/design-patterns/singleton>
- <http://www.oodesign.com/singleton-pattern.html>

# Șablonul creațional: Abstract Factory

- oferă o interfață pentru crearea de familii de obiecte înrudite sau dependente fără a specifica clasele lor concrete
- exemplu clasic: un set de instrumente GUI (Widgets) care oferă look-and-feel multiplu, să zicem pentru două pachete diferite: Motif și Presentation Manager (PM).



# Abstract Factory - diagrama UML





# Abstract Factory

```
abstract class AbstractProductA{
    public abstract void
        operationAx();
    public abstract void
        operationAy();
}

class ProductA1 extends
    AbstractProductA{
    ProductA1(String arg){
        System.out.println("Hello"
            +arg);
    } // Implement the code here
    public void operationAx() { };
    public void operationAy() { };
}

class ProductA2 extends
    AbstractProductA{
    ProductA2(String arg){
        System.out.println("Hello"
            +arg);
    } // Implement the code here
    public void operationAx() { };
    public void operationAy() { };
}

abstract class AbstractProductB{
    public abstract void
        operationBx();
    public abstract void
        operationBy();
}

class ProductB1 extends
    AbstractProductB{
    ProductB1(String arg){
        System.out.println("Hello"
            +arg);
    } // Implement the code here
}

class ProductB2 extends
    AbstractProductB{
    ProductB2(String arg){
        System.out.println("Hello"
            +arg);
    } // Implement the code here
}

abstract class AbstractFactory{
    abstract AbstractProductA
        createProductA();
    abstract AbstractProductB
        createProductB();
}

class ConcreteFactory1 extends
    AbstractFactory{
    AbstractProductA
        createProductA(){
            return new
                ProductA1("ProductA1");
        }
    AbstractProductB
        createProductB(){
            return new
                ProductB1("ProductB1");
        }
}

class ConcreteFactory2 extends
    AbstractFactory{
    AbstractProductA
        createProductA(){
            return new
                ProductA2("ProductA2");
        }
    AbstractProductB
        createProductB(){
            return new
                ProductB2("ProductB2");
        }
}

//Factory creator – an indirect way
//of instantiating the factories
class FactoryMaker{
    private static AbstractFactory
        pf=null;

    static AbstractFactory
        getFactory(String choice) {
        if(choice.equals("1")) {
            pf=new ConcreteFactory1();
        } else if(choice.equals("2")){
            pf=new ConcreteFactory2();
        }
        return pf;
    }
}

// Client
public class Client{
    public static void main(String
        args[]){

        AbstractFactory
            pf=FactoryMaker.getFactory("1");

        AbstractProductA
            product=pf.createProductA();

        //more function calls on product
    }
}
```

**OUTPUT: Hello ProductA1**

# Abstract Factory

## ■ observații

- independent de modul în care produsele sunt create, compuse și reprezentate
- produsele înrudite trebuie să fie utilizate împreună
- pune la dispoziție doar interfața, nu și implementarea

## ■ consecințe:

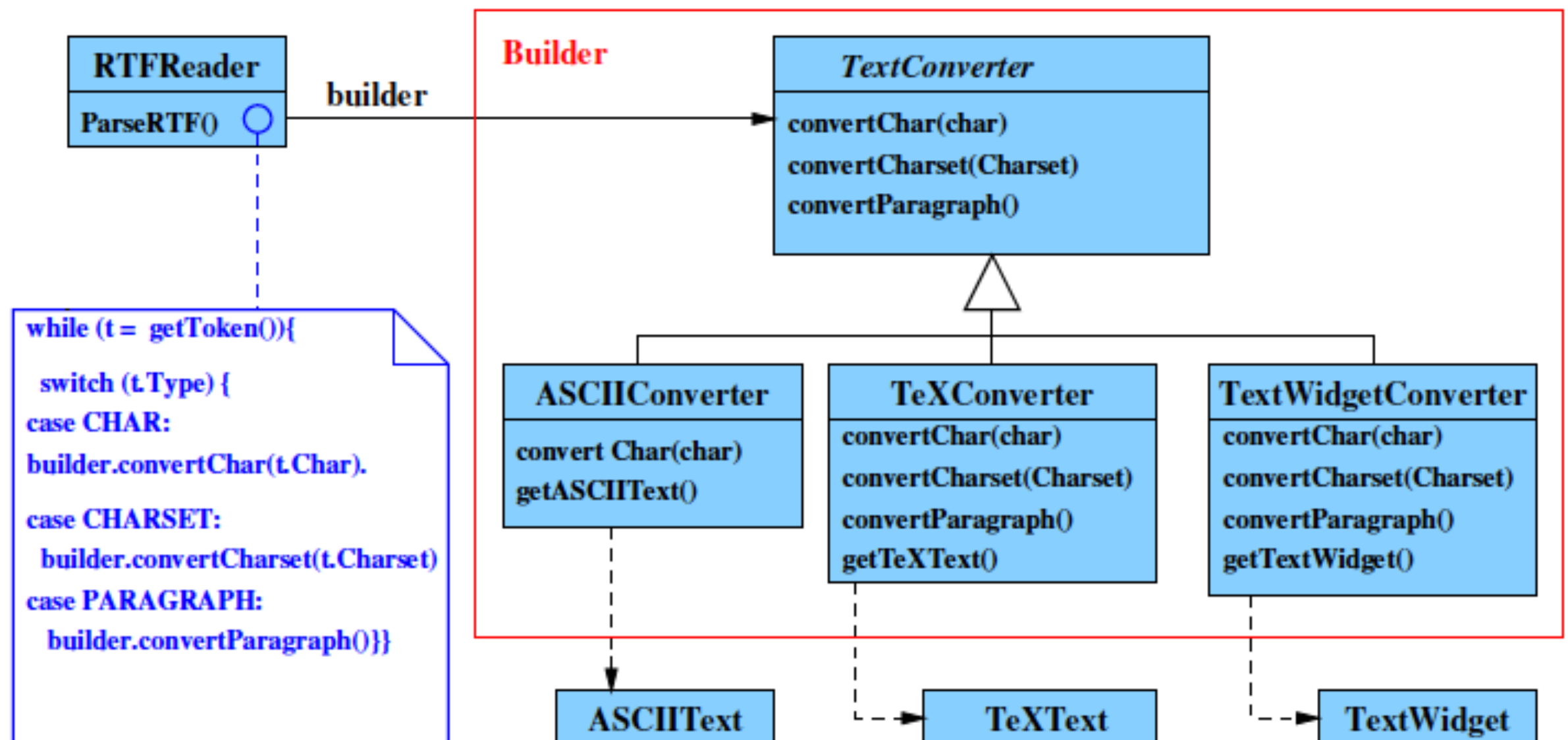
- numele de clase de produse nu apar în cod
- familiile de produse ușor interschimbabile
- cere consistență între produse

## ■ v. și:

- [http://en.wikipedia.org/wiki/Abstract\\_factory\\_pattern](http://en.wikipedia.org/wiki/Abstract_factory_pattern)
- <https://refactoring.guru/design-patterns/abstract-factory>
- <http://www.oodesign.com/abstract-factory-pattern.html>

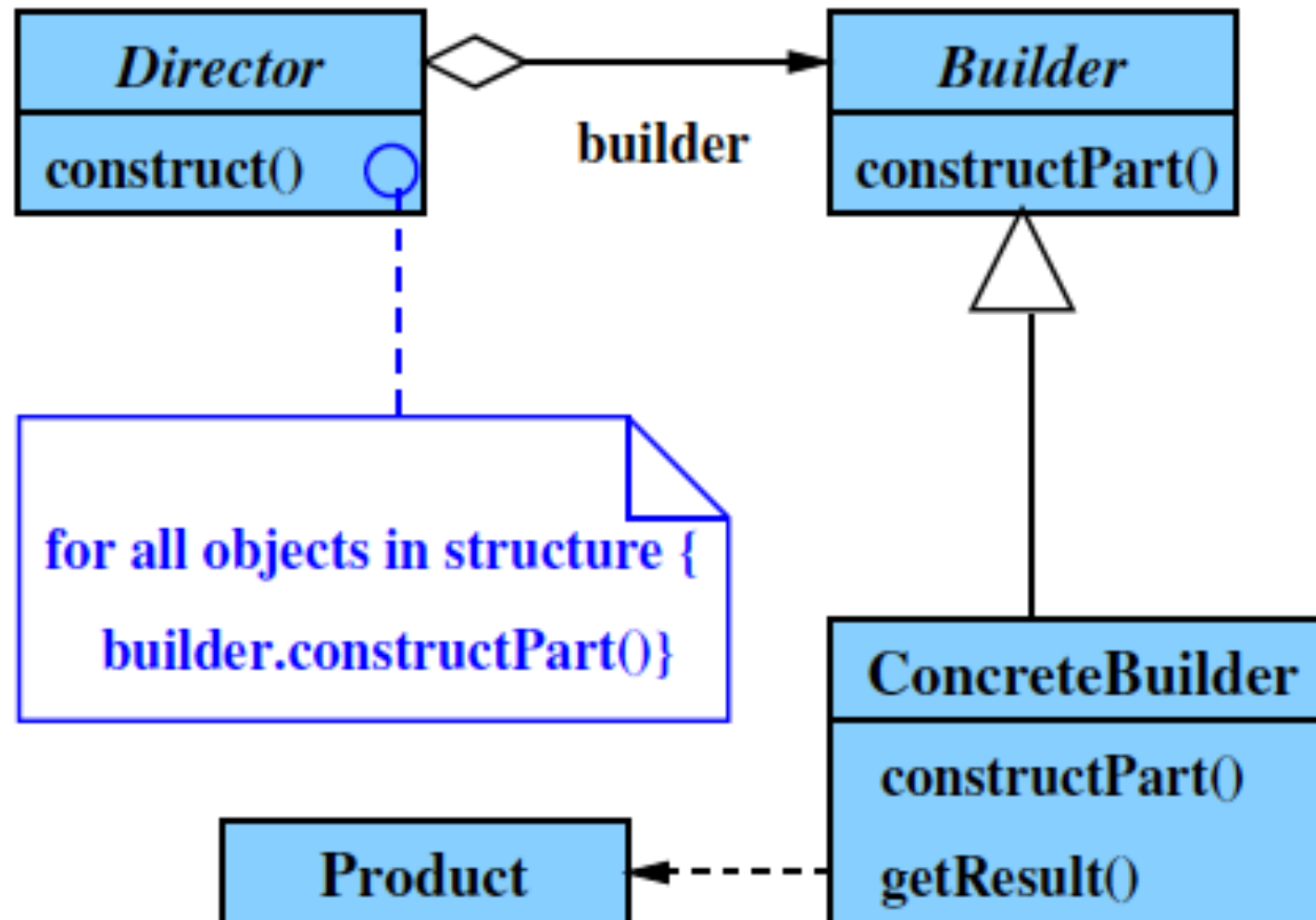
# Șablonul creațional: Builder

- separă construcția unui obiect complex de reprezentarea sa, astfel încât același proces de construcție poate crea reprezentări diferite.
- exemplu clasic: citește RTF (Rich Text Format) și traduce în diferite formate interschimbabile





# Builder - diagrama UML



# Builder - exemplu (Pizza)

```
/* "Product" */
class Pizza {
    private String dough = "";
    private String sauce = "";
    private String topping = "";

    public void setDough(String
                           dough)
    { this.dough = dough; }
    public void setSauce(String
                           sauce)
    { this.sauce = sauce; }
    public void setTopping(String
                           topping)
    { this.topping = topping; }
}
```

```
/* "Abstract Builder" */
abstract class PizzaBuilder {
    protected Pizza pizza;

    public Pizza getPizza()
    { return pizza; }

    public void
    createNewPizzaProduct()
    { pizza = new Pizza(); }

    public abstract void
    buildDough();
    public abstract void
    buildSauce();
    public abstract void
    buildTopping();
}
```

```
/* "ConcreteBuilder 1" */
class HawaiianPizzaBuilder
    extends PizzaBuilder {
    public void buildDough()
    { pizza.setDough("cross"); }
    public void buildSauce()
    { pizza.setSauce("mild"); }
    public void buildTopping()
    { pizza.setTopping("ham
                        +pineapple"); }
}
```

```
/* "ConcreteBuilder 2" */
class SpicyPizzaBuilder extends
    PizzaBuilder {
    public void buildDough()
    { pizza.setDough("pan baked"); }
    public void buildSauce()
    { pizza.setSauce("hot"); }
    public void buildTopping()
    { pizza.setTopping("pepperoni
                        +salami"); }
}
```

```
/* "Director" */
class Waiter {
    private PizzaBuilder
        pizzaBuilder;

    public void
    setPizzaBuilder(PizzaBuilder pb)
    { pizzaBuilder = pb; }

    public Pizza getPizza()
    { return
```

```
        pizzaBuilder.getPizza(); }

    public void constructPizza() {

        pizzaBuilder.createNewPizzaPr
            oduct();
        pizzaBuilder.buildDough();
        pizzaBuilder.buildSauce();
        pizzaBuilder.buildTopping();
    }
}
```

```
/* A customer/client ordering a
   pizza. */
class BuilderExample {

    public static void
    main(String[] args) {

        Waiter waiter = new
            Waiter();

        PizzaBuilder
            hawaiian_pizzabuilder = new
            HawaiianPizzaBuilder();
        PizzaBuilder
            spicy_pizzabuilder = new
            SpicyPizzaBuilder();

        waiter.setPizzaBuilder( hawai
            ian_pizzabuilder );
        waiter.constructPizza();
        Pizza pizza =
            waiter.getPizza();
    }
}
```

# Builder

## ■ observații

- folosit când algoritmul de creare a unui obiect complex este independent de părțile care compun obiectul și de modul lor de asamblare
- și când procesul de construcție trebuie să permită reprezentări diferite pentru obiectul construit

## ■ comparație cu *Abstract Factory*:

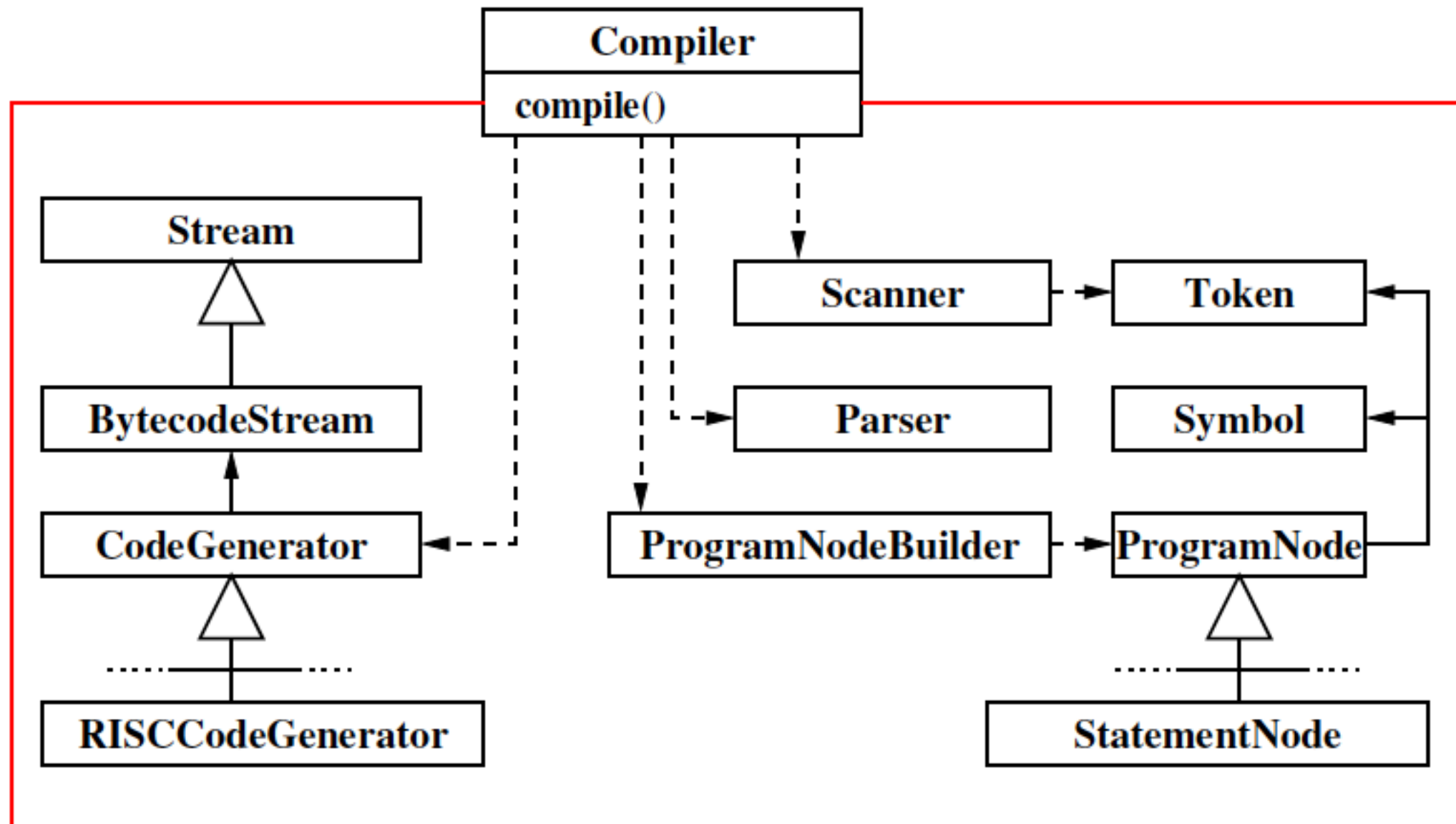
- *Builder* creează un produs complex pas cu pas. *Abstract Factory* creează familii de produse, de fiecare dată produsul fiind complet

## ■ v. și:

- [http://en.wikipedia.org/wiki/Builder\\_pattern](http://en.wikipedia.org/wiki/Builder_pattern)
- <https://refactoring.guru/design-patterns/builder>
- <http://www.oodesign.com/builder-pattern.html>

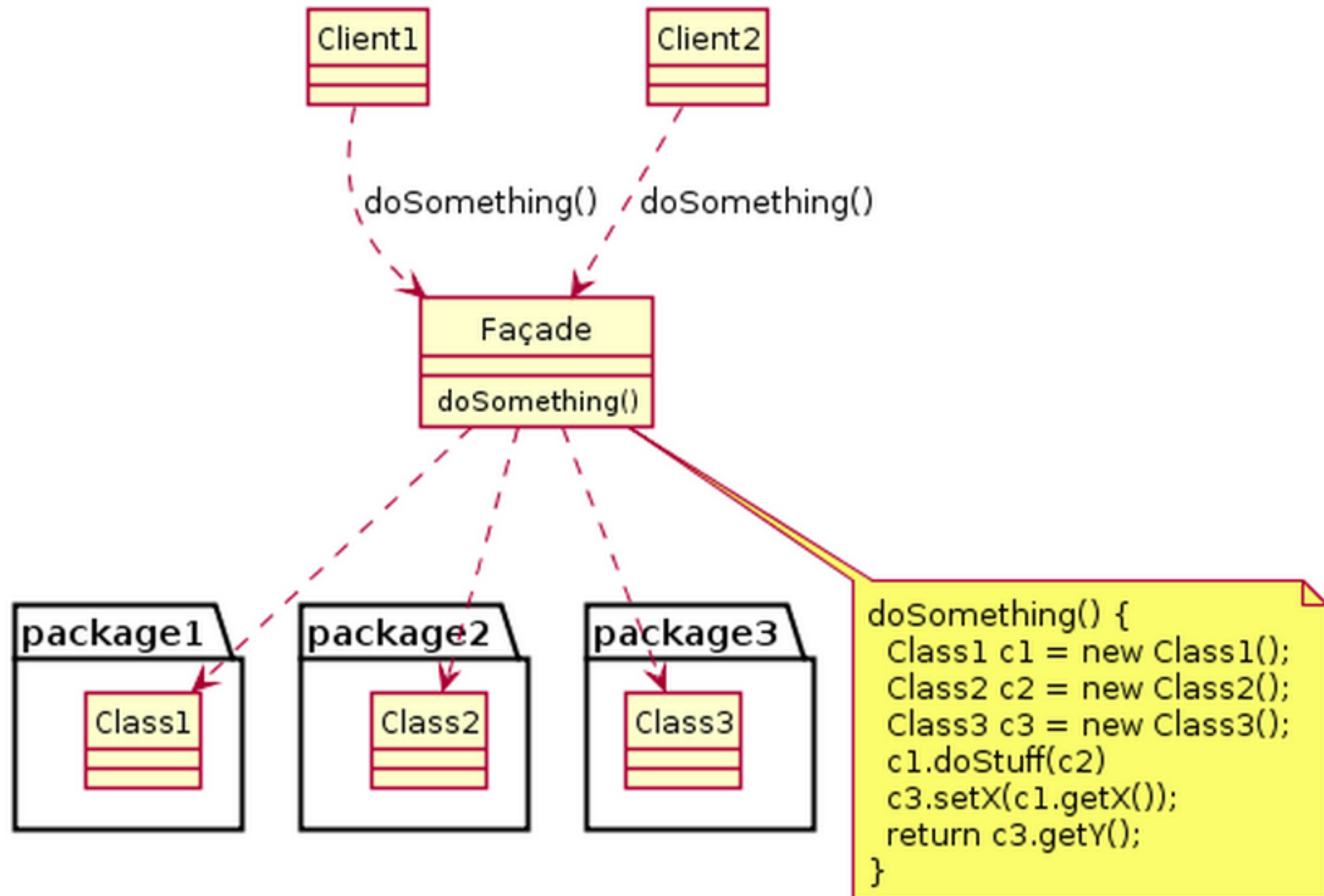
# Șablonul structural: Facade

- oferă o interfață unificată pentru un set de interfețe într-un subsistem
- exemplu: un subsistem de tip compilator care conține scanner, parser, generator de cod etc. Șablonul *Facade* combină interfețele și oferă o nouă operație de tip *compile()*





# Facade - diagramă UML



# Facade - exemplu

**/\* Complex parts \*/**

```
class CPU {
    public void freeze() { ... }
    public void jump(long position)
                    { ... }
    public void execute() { ... }
}

class Memory {
    public void load(long position,
                    byte[] data) { ... }
}

class HardDrive {
    public byte[] read(long lba,
                    int size) { ... }
}
```

**/\* Facade \*/**

```
class ComputerFacade {
    private CPU processor;
    private Memory ram;
    private HardDrive hd;
```

```
    public ComputerFacade() {
        this.processor = new CPU();
        this.ram = new Memory();
        this.hd = new HardDrive();
    }

    public void start() {
        processor.freeze();
        ram.load(BOOT_ADDRESS,
                hd.read(BOOT_SECTOR,
                        SECTOR_SIZE));
        processor.jump(BOOT_ADDRESS);
        processor.execute();
    }
}
```

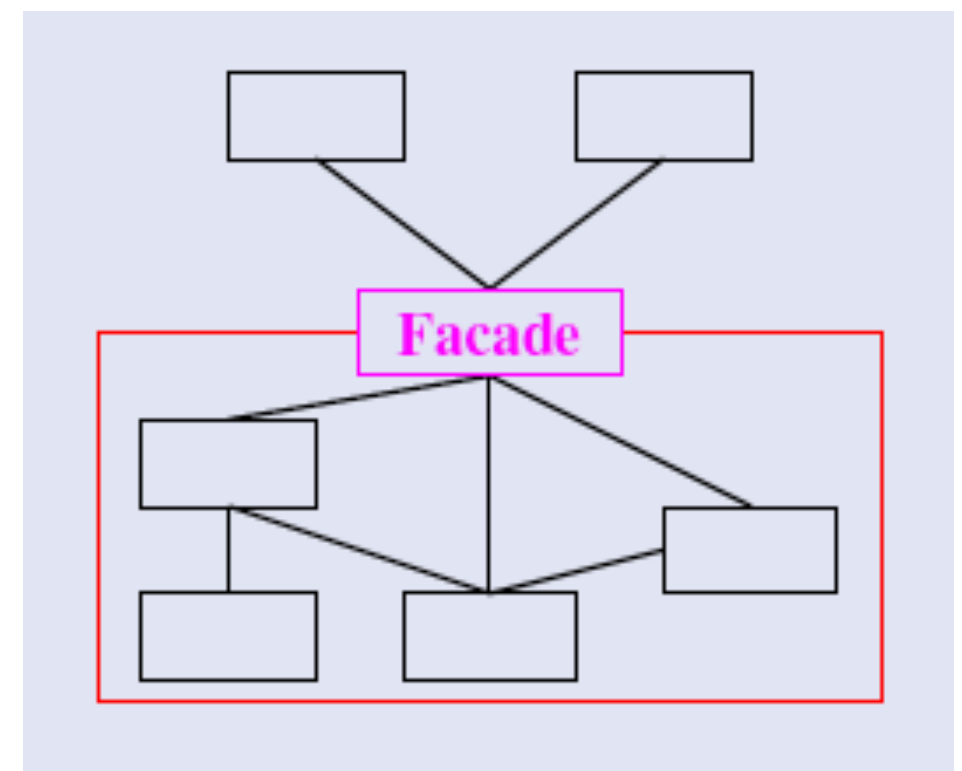
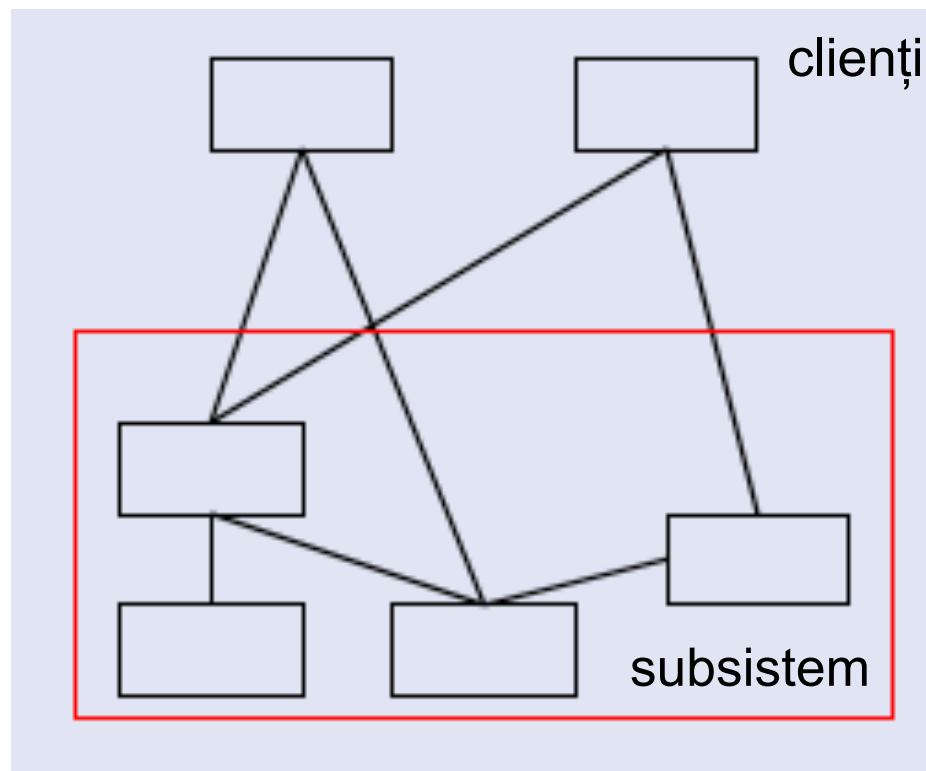
**/\* Client \*/**

```
class You {
    public static void main(String[]
        args) {
        ComputerFacade computer = new
            ComputerFacade();
        computer.start();
    }
}
```

# Facade

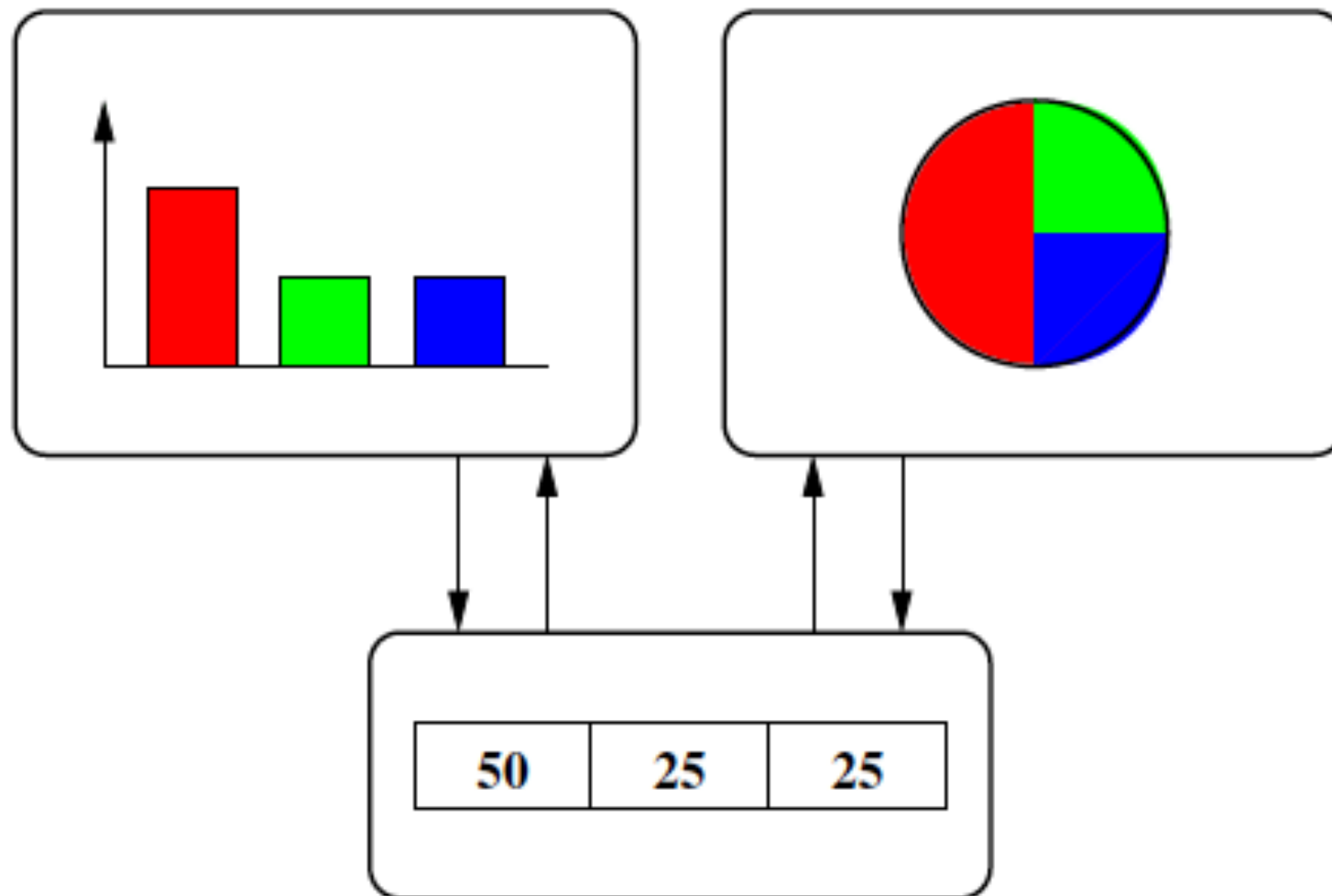
## ■ observații

- o interfață simplă la un subsistem complex
- când sunt multe dependențe între clienți și subsistem, este redusă cuplarea

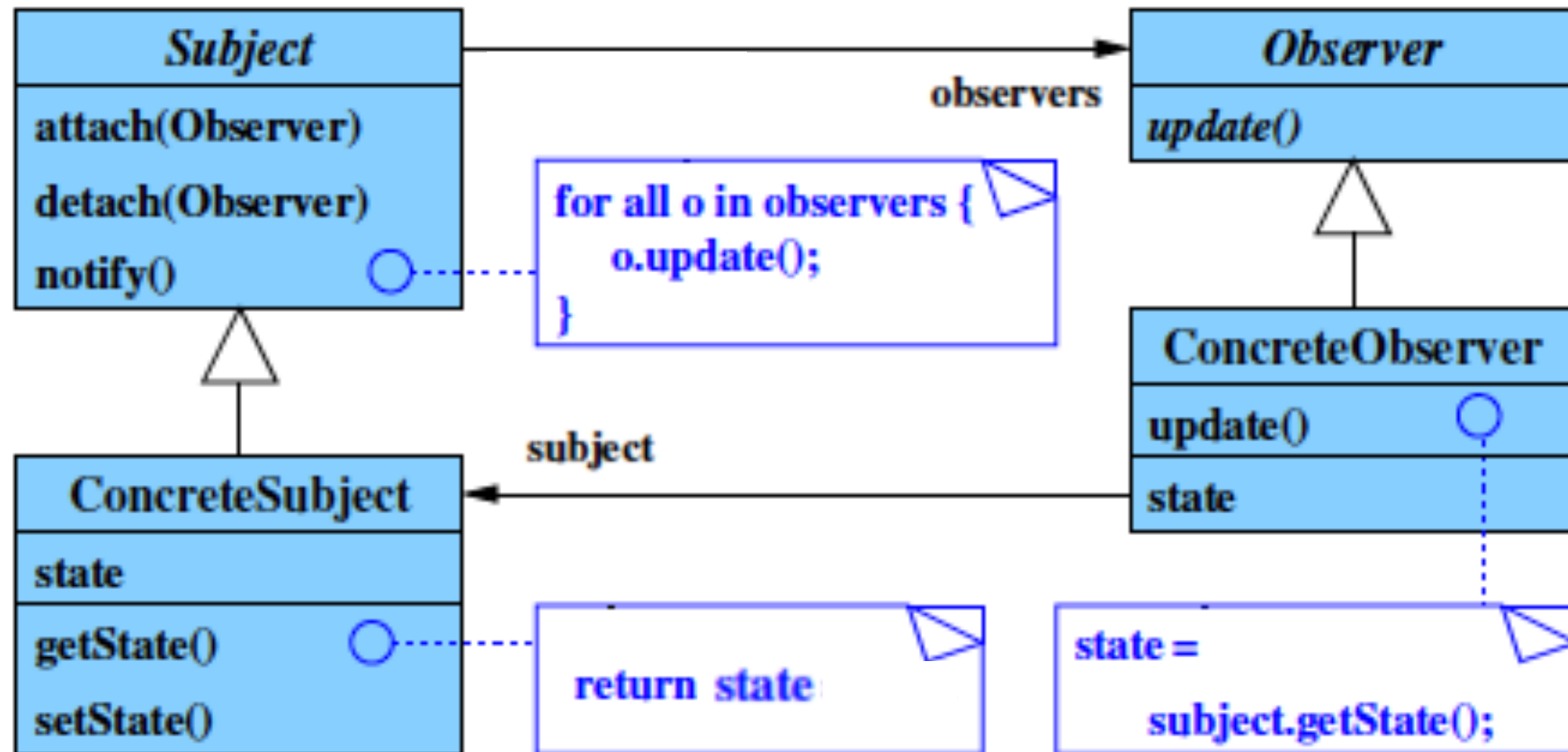


# Șablonul comportamental: Observer

- presupunem o dependență de **1:n** între obiecte
- când se schimbă starea unui obiect, toate obiectele dependente sunt înștiințate
- de exemplu: poate menține consistența între perspectiva internă și cea externă



# Observer - diagrama UML

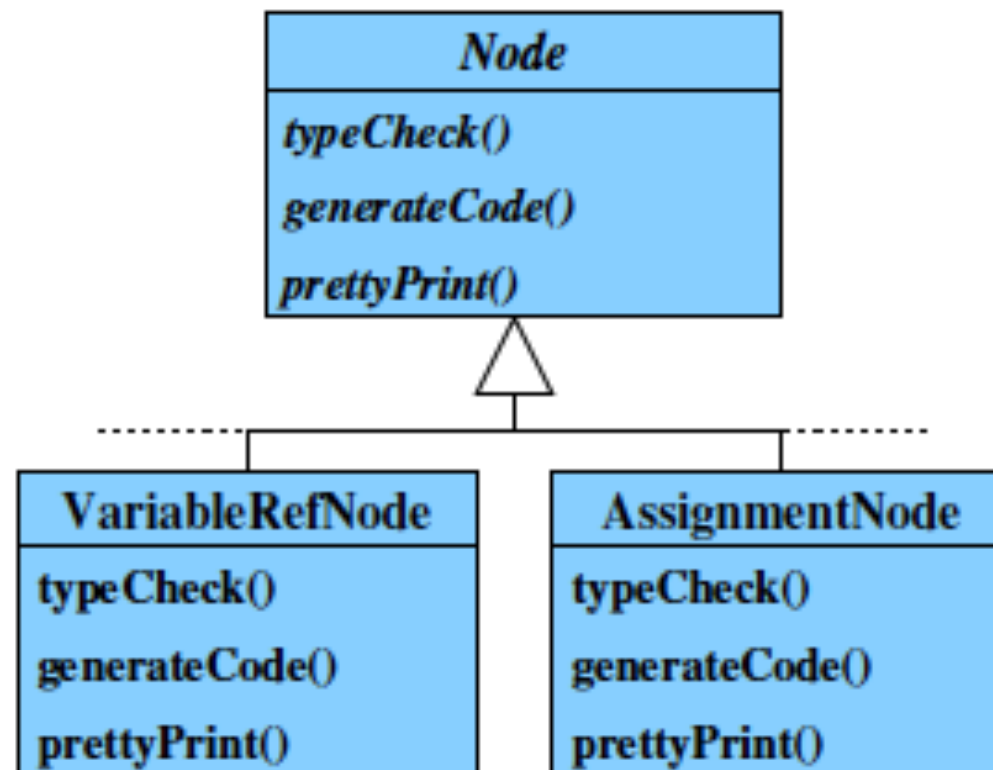


- propagarea schimbării stării / comunicare de tip broadcast
- notificare anonimă
- *Subject* și *Observer* sunt independenți (cuplare abstractă)
- observatorii sunt configurabili în mod dinamic

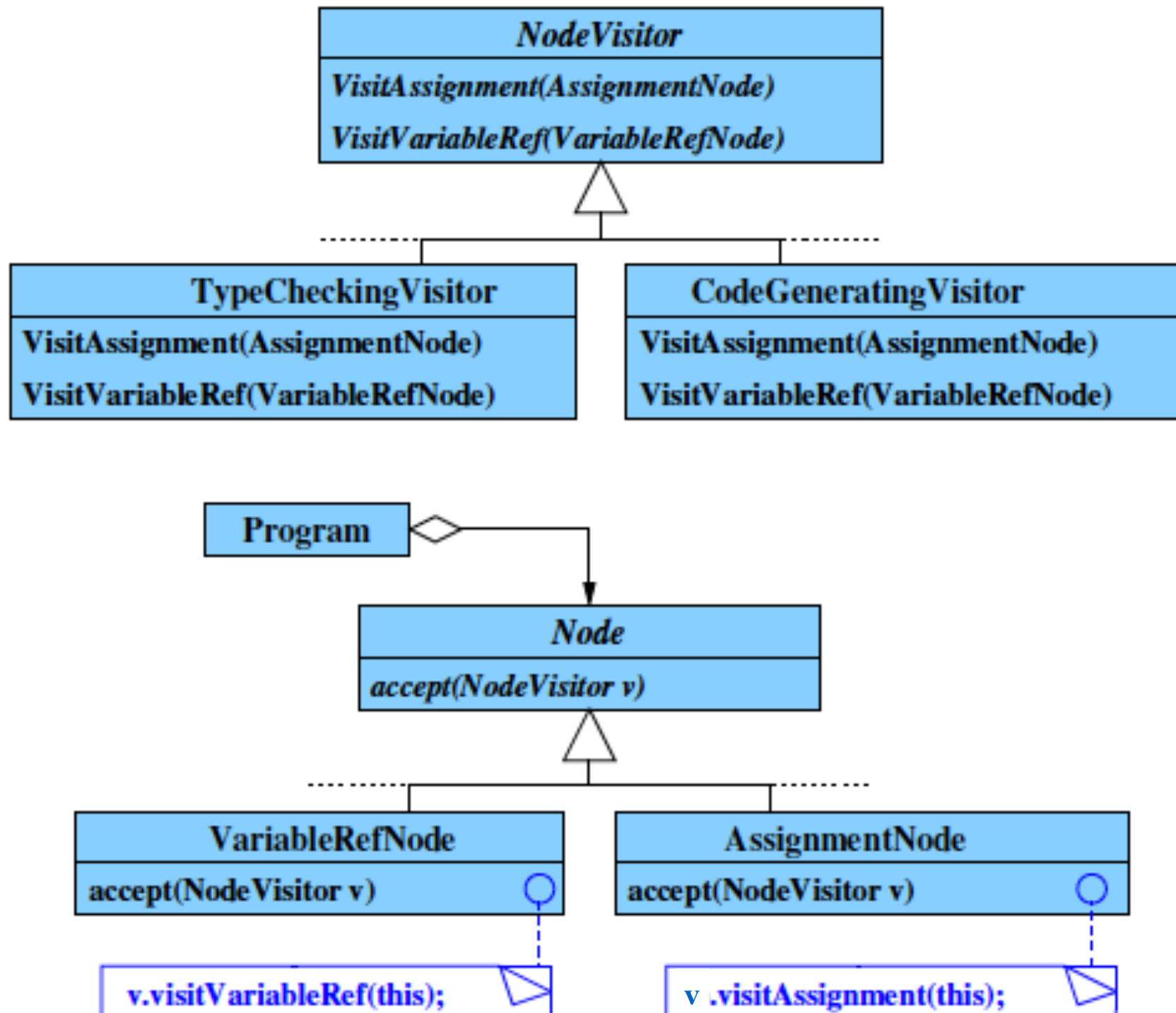


# Șablonul comportamental: Visitor

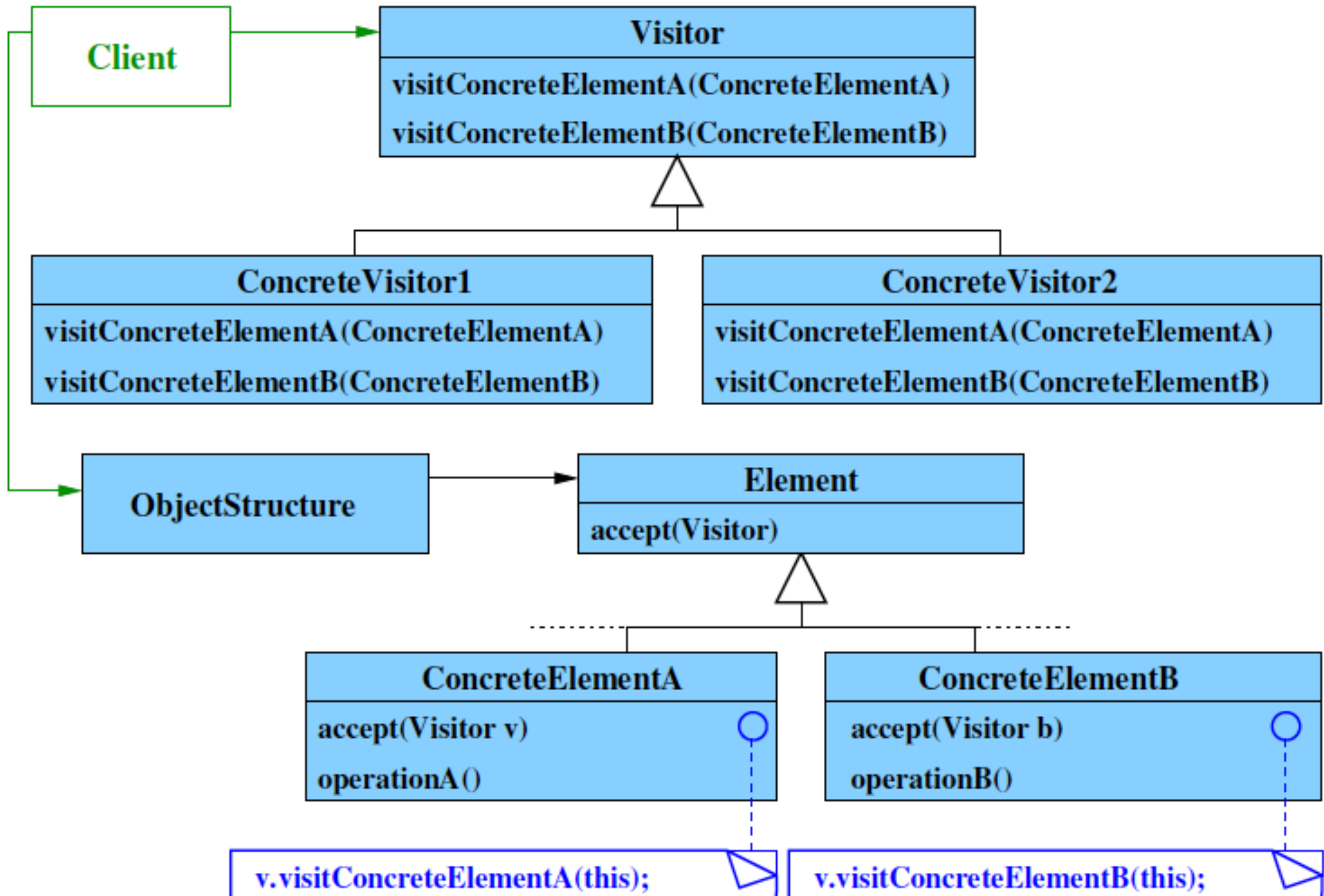
- reprezintă operații pe o structură de obiect
- adaugă noi operații, fără a modifica însă clasele
- de exemplu: procesarea arborelui sintactic într-un compilator (type checking, generare de cod, pretty print)
  - o primă versiune (neoptimă): operațiile sunt puse în clasele de tip nod



# Soluția cu Visitor



# Visitor - diagrama UML



# Bibliografie și legături

- “Dive into Design Patterns” (2019)
  - explicații și surse: <https://refactoring.guru/design-patterns>
- “Head First Design Patterns”
  - codul sursă al exemplelor din carte: <https://github.com/bethrobson/Head-First-Design-Patterns/archive/master.zip>
- <http://www.oodesign.com>
- <https://github.com/kamranahmedse/design-patterns-for-humans>
- Wikipedia, cartea GoF citată la început etc.



# Anti-şabloane





# Exemple de anti-șabloane (anti-patterns)

- abstraction inversion
- input kludge
- interface bloat
- magic pushbutton
- race hazard
- stovepipe system
- BaseBean
- God object
- circle-ellipse problem
- yo-yo problem
- poltergeists
- etc.

[http://en.wikipedia.org/wiki/Anti-pattern#Software\\_engineering](http://en.wikipedia.org/wiki/Anti-pattern#Software_engineering)