Universitatea din București

Metode de dezvoltare software

Șabloane de proiectare

06.05.2020

Alin Ștefănescu

Șabloane de proiectare

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

Șabloane de proiectare

- originare în **ingineria construcţiilor şi arhitectură**
  - Christopher Alexander, arhitect
    - *A Pattern Language: Towns, Buildings, Construction* (1977)
  - http://en.wikipedia.org/wiki/Pattern_language
- un **şablon** este **suficient de general** pentru a fi aplicat în mai multe situaţii, dar **suficient de concret** pentru a fi util în luarea deciziilor

Șabloane software

În **ingineria software**:

- un **şablon** este *o soluţie la o problemă într-un context*, unde:
  - **contextul**: situaţiile recurente în care se aplică şablonul
  - **problema**: scopurile şi constrângerile
  - **soluţia**: 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 organizație 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
- **șabloane de proiectare**
  - la nivelul **claselor**/modulelor
  - http://en.wikipedia.org/wiki/Software_design_pattern
- **idiomuri**
  - la nivelul limbajului
  - http://en.wikipedia.org/wiki/Programming_idiom

în restul cursului

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)

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 **creaţionale**: instanţierea
  - șabloane **structurale**: compunerea
  - șabloane **comportamentale**: comunicarea

- există bineînțeles **multe alte șabloane**, dar cele 23 de mai sus sunt “**clasice**”

Șabloanele

- **Creaționale**:
  - 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

http://www.mcdonaldland.info/files/designpatterns/designpatternscard.pdf

C Abstract Factory
C Adapter
C Bridge
C Builder
H Chain of Responsibility
H Command
H Composite
H Decorator

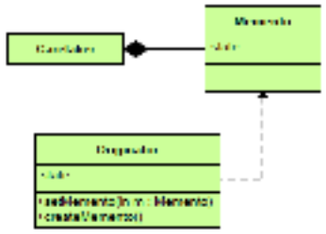
C Facade
C Factory Method
C Flyweight
C Interpreter
H Iterator
H Mediator
H Memento
H Prototype

C Proxy
H Observer
H Singleton
C State
H Strategy
C Template Method
H Visitor

Memento

Type: Behavioral

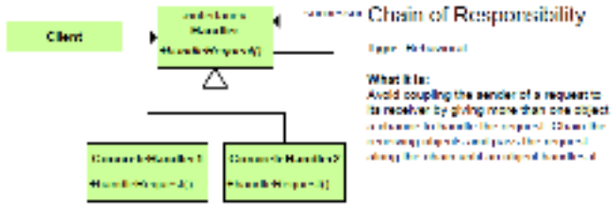
What it is:
Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.



Chain of Responsibility

Type: Behavioral

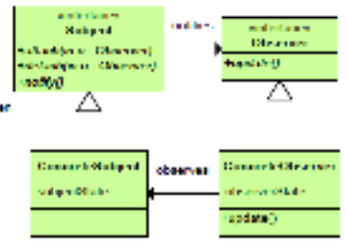
What it is:
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.



Observer

Type: Behavioral

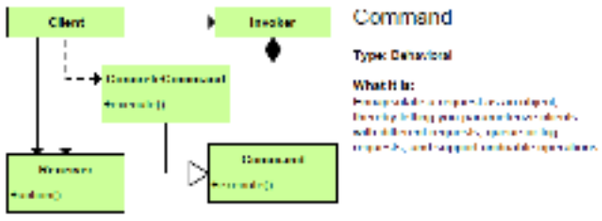
What it is:
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



Command

Type: Behavioral

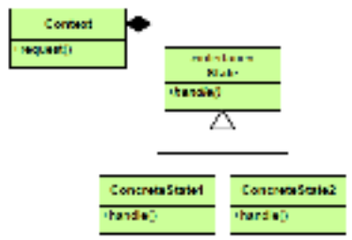
What it is:
Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.



State

Type: Behavioral

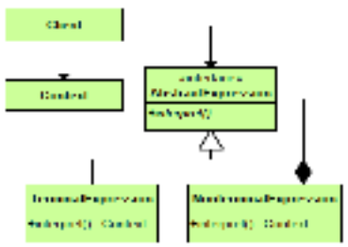
What it is:
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.



Interpreter

Type: Behavioral

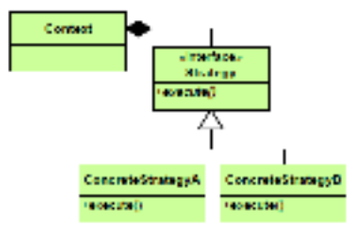
What it is:
Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.



Strategy

Type: Behavioral

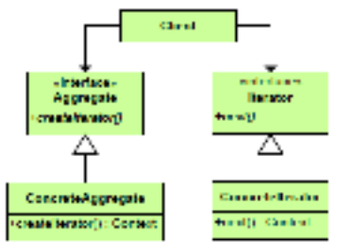
What it is:
Define a family of algorithms, encapsulate each one, and make them interchangeable. Lets the algorithm vary independently from clients that use it.



Iterator

Type: Behavioral

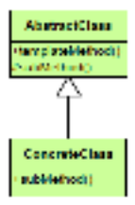
What it is:
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.



Template Method

Type: Behavioral

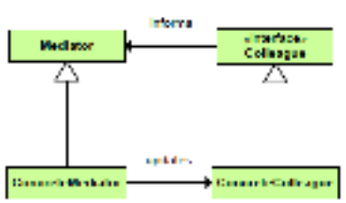
What it is:
Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.



Mediator

Type: Behavioral

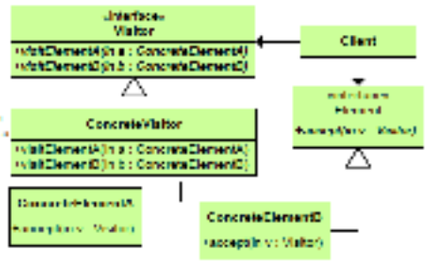
What it is:
Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly and it lets you vary their interactions independently.



Visitor

Type: Behavioral

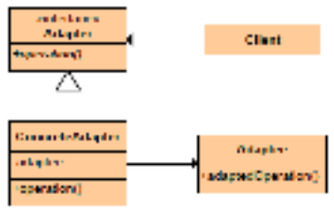
What it is:
Represent an operation to be performed on the elements of an object structure. Lets you define a new operation without changing the classes of the elements on which it operates.



Adapter

Type: Structural

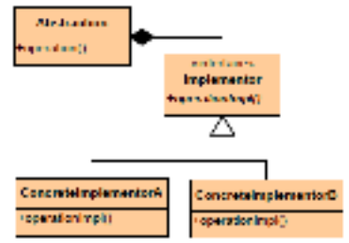
What it is:
Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.



Bridge

Type: Structural

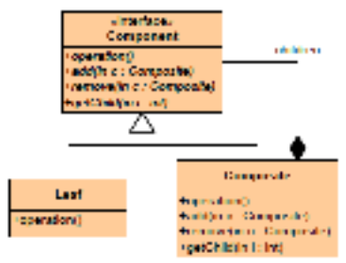
What it is:
Decouple an abstraction from its implementation so that the two can vary independently.



Composite

Type: Structural

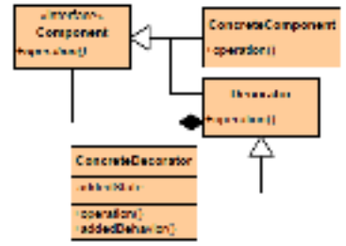
What it is:
Compose objects into tree structures to represent part-whole hierarchies. Lets clients treat individual objects and compositions of objects uniformly.



Decorator

Type: Structural

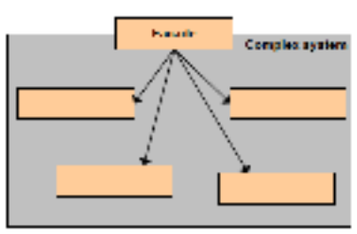
What it is:
Attach additional responsibilities to an object dynamically. Provides a flexible alternative to sub-classing for extending functionality.



Facade

Type: Structural

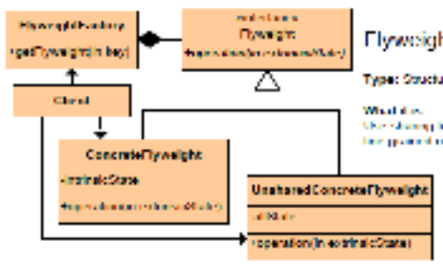
What it is:
Provide a unified interface to a set of interfaces in a subsystem. Defines a high-level interface that makes the subsystem easier to use.



Flyweight

Type: Structural

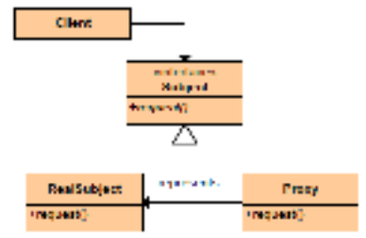
What it is:
Use sharing to support large numbers of fine-grained objects efficiently.



Proxy

Type: Structural

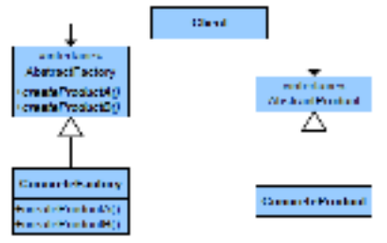
What it is:
Provide a surrogate or placeholder for another object to control access to it.



Abstract Factory

Type: Creational

What it is:
Provides an interface for creating families of related or dependent objects without specifying their concrete classes.



Builder

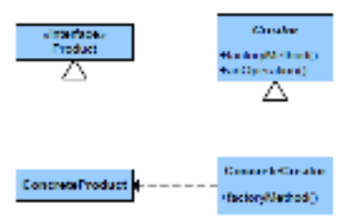
Type: Creational

What it is:
Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Factory Method

Type: Creational

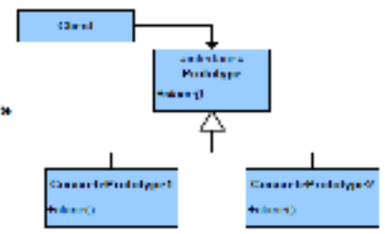
What it is:
Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.



Prototype

Type: Creational

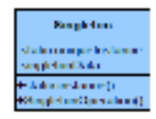
What it is:
Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.



Singleton

Type: Creational

What it is:
Ensure a class only has one instance and provide a global point of access to it.

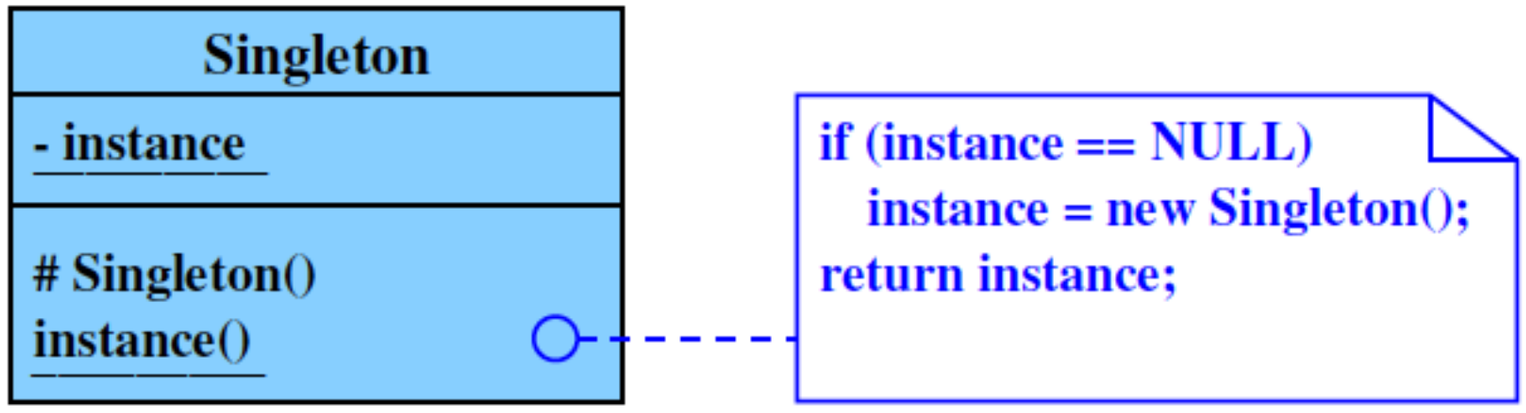


Șablonul creațional: Singleton

- 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



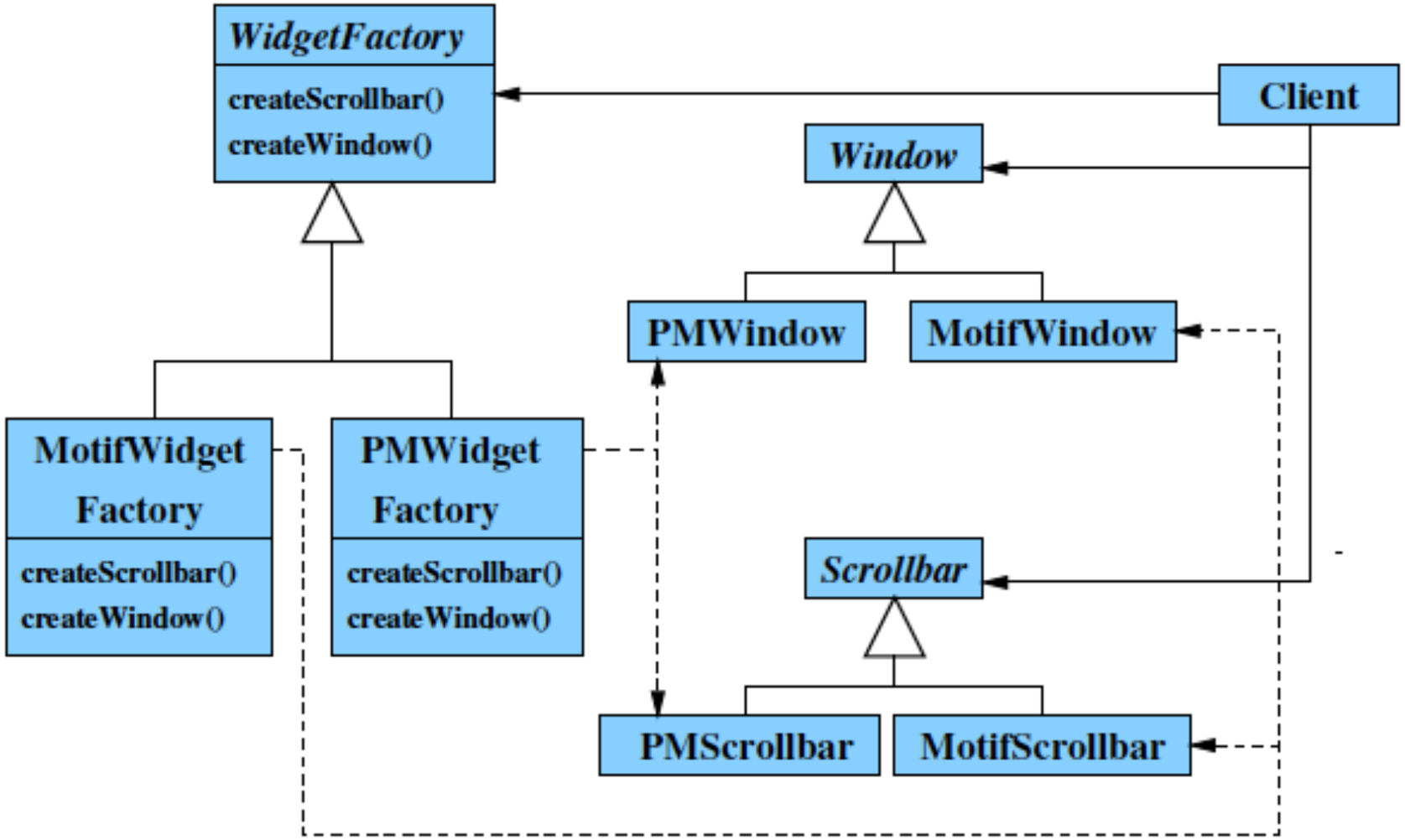

```
class Singleton
{
  private static Singleton instance;

  protected Singleton()
  {...}

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

}
```

Singleton

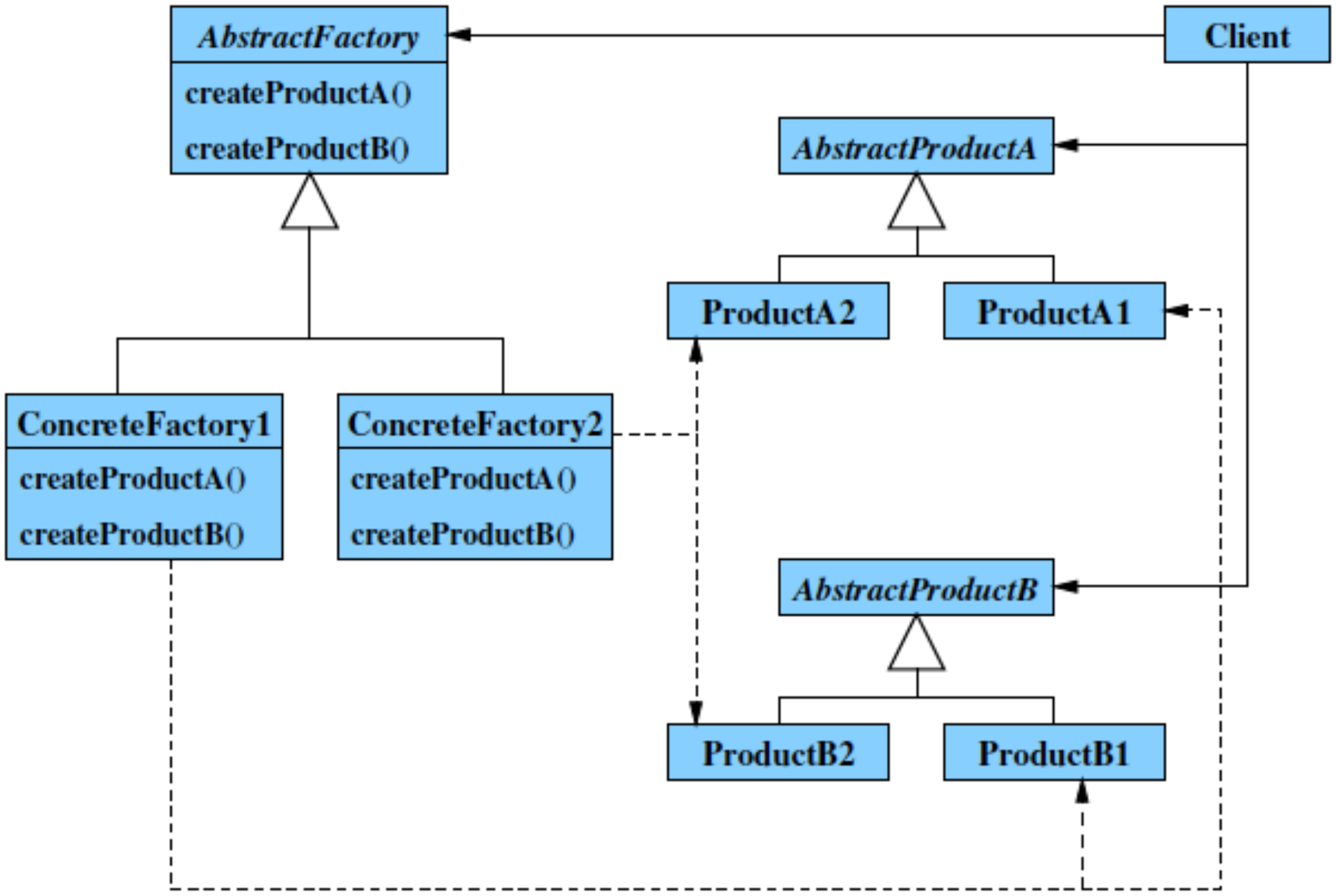
- aplicabilitate:
  - când doar un obiect al unei clase e necesar
  - când instanța este accesibilă global
  - este folosit în alte șabloane (*factories* și *builders*)
- consecințe:
  - accesul controlat la instanță
  - spațiu de adresare structurat (comparativ cu o variabilă globală)
- v. și:
  - http://en.wikipedia.org/wiki/Singleton_pattern
  - 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
 }
}
```

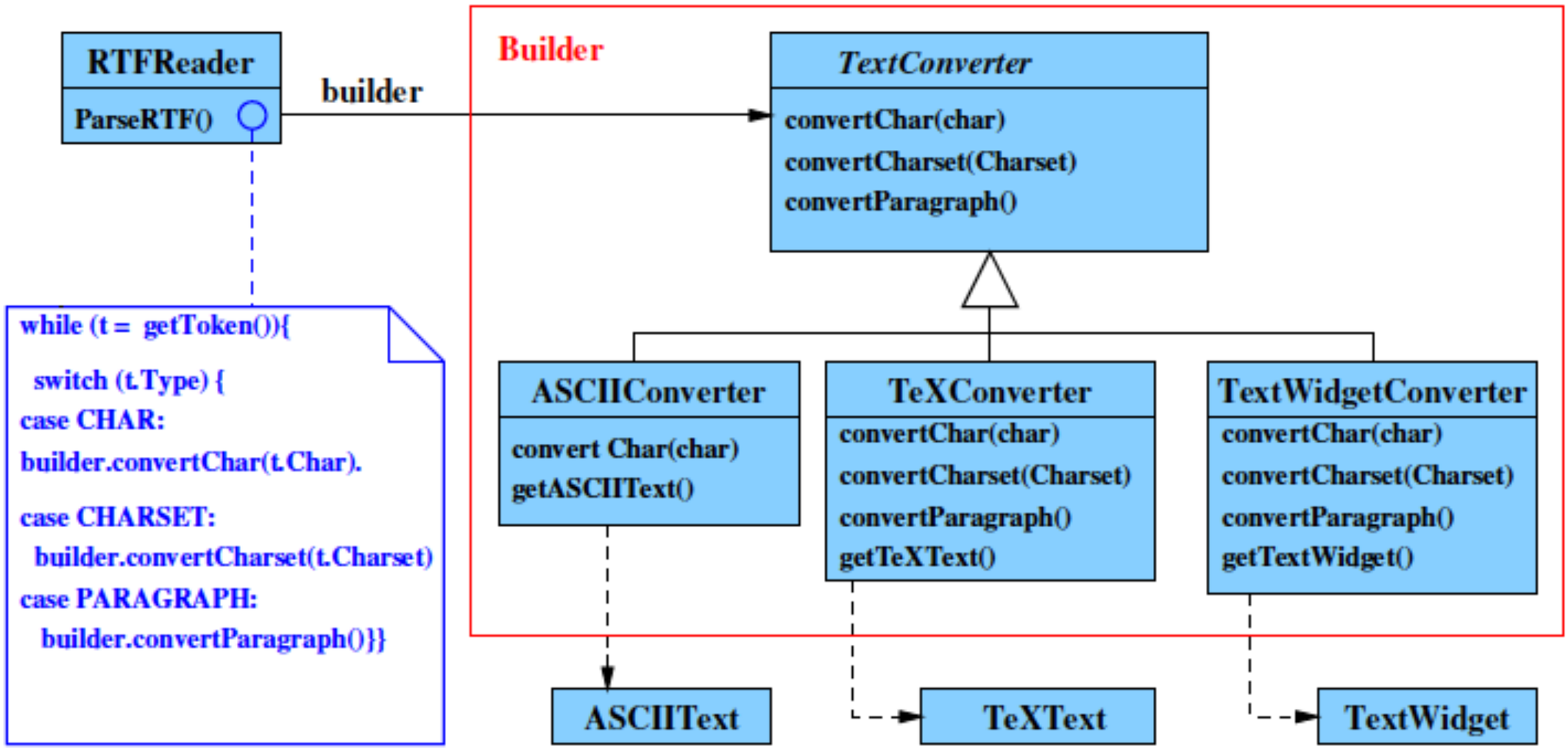
OUTPUT: Hello ProductA1

Abstract Factory

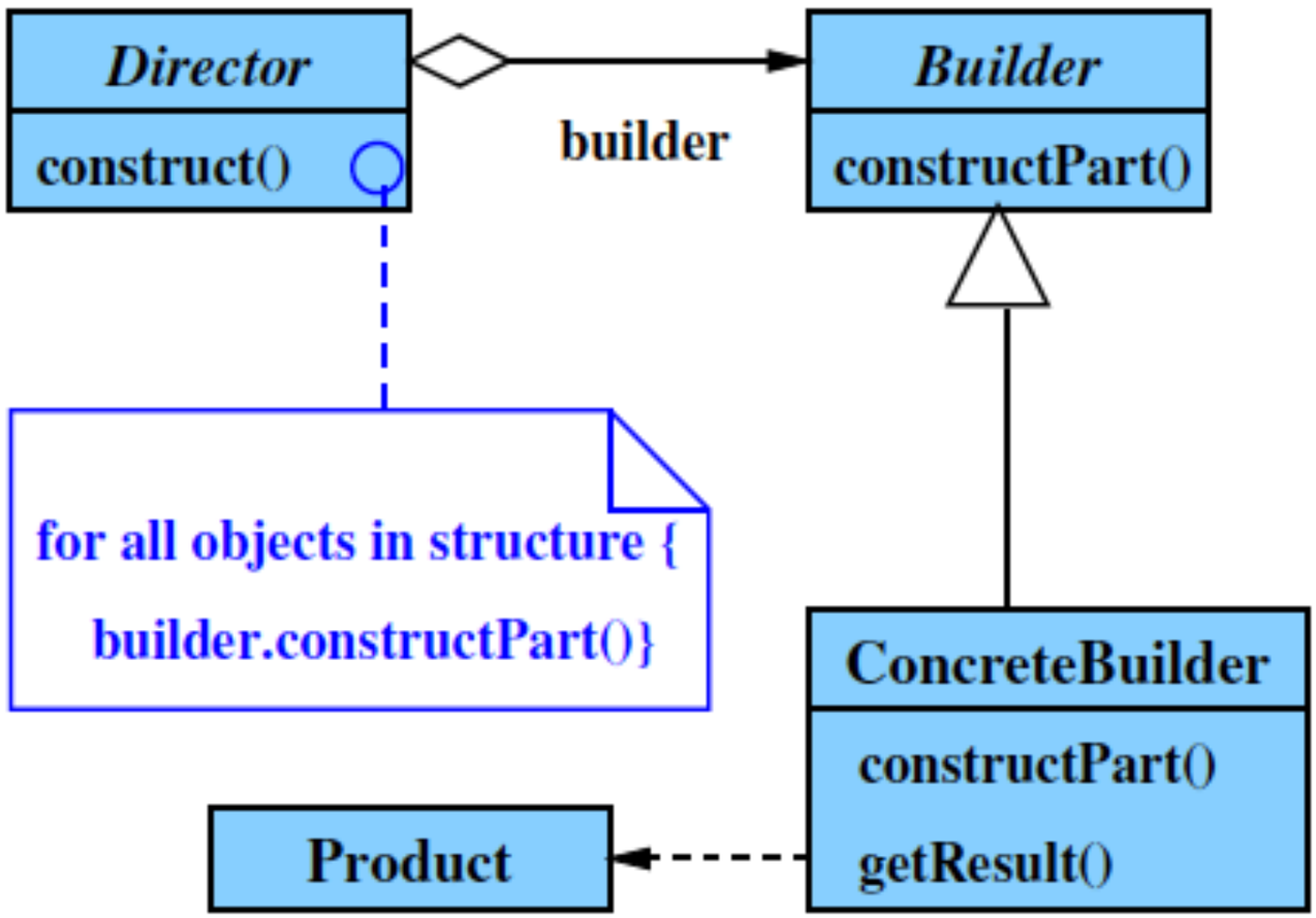
- observații
  - independent de modul în care produsele sunt create, compuse și reprezentate
  - produsele înrudite trebuie să fie utilizate împreună
  - pune la dispoziție doar interfața, nu și implementarea
- consecinte:
  - numele de clase de produse nu apar în cod
  - familiile de produse ușor interschimbabile
  - cere consistență între produse
- v. și:
  - http://en.wikipedia.org/wiki/Abstract_factory_pattern
  - 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

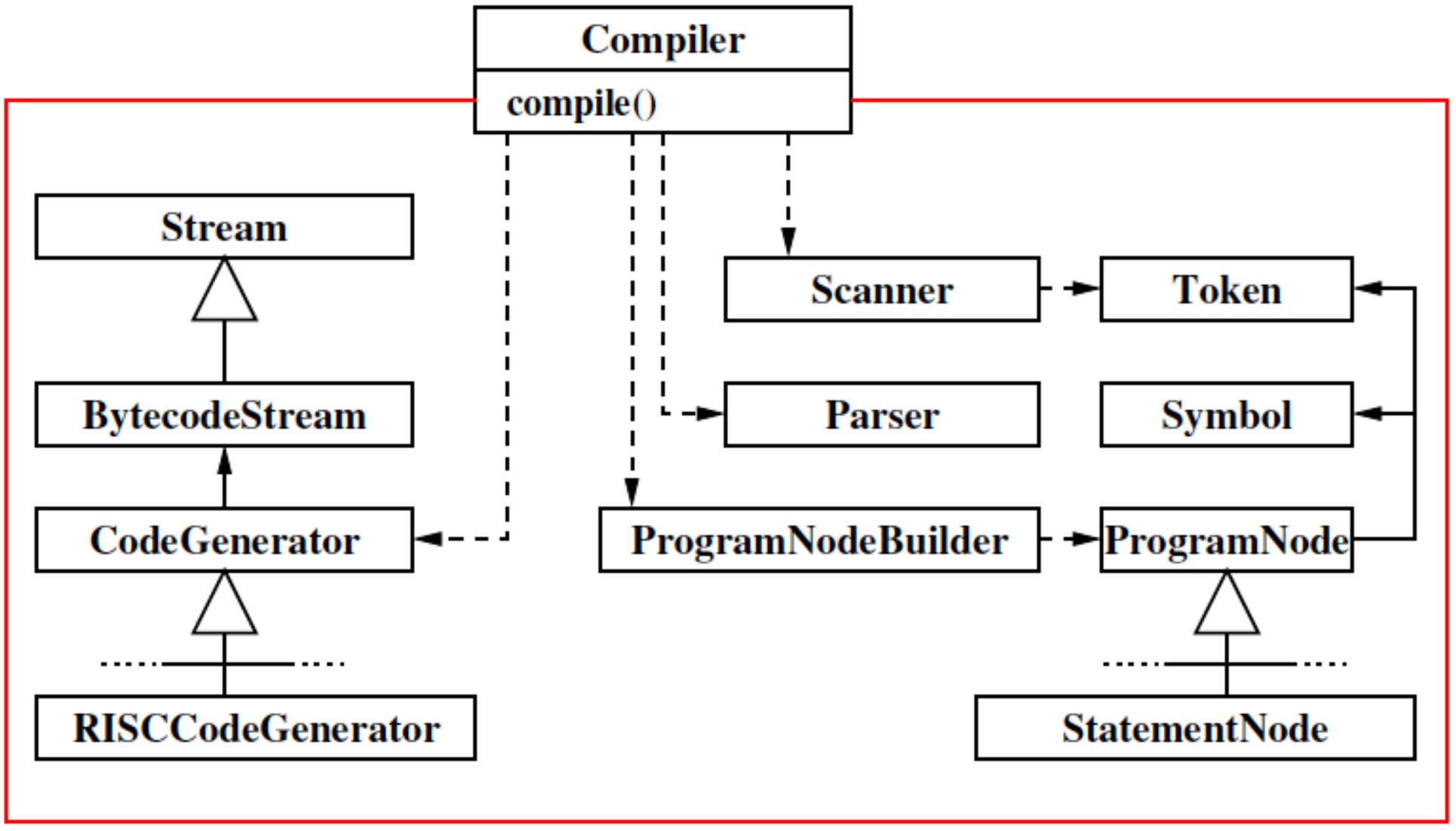
- observații
  - folosit când algoritmul de creare a unui obiect complex este independent de părţile care compun obiectul şi de modul lor de asamblare
  - și când procesul de construcţie trebuie să permită reprezentări diferite pentru obiectul construit

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

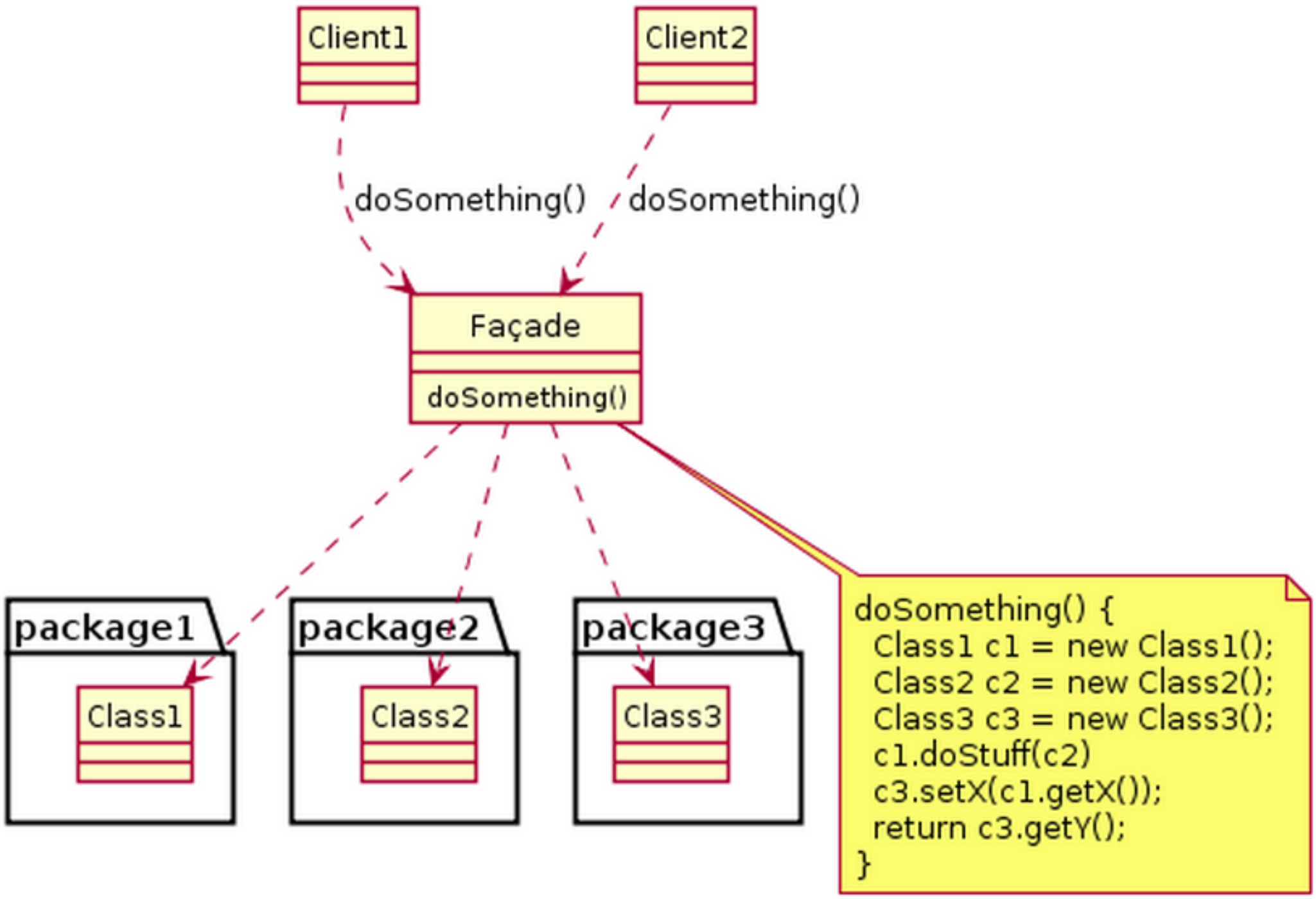
- v. și:
  - http://en.wikipedia.org/wiki/Builder_pattern
  - https://refactoring.guru/design-patterns/builder
  - http://www.oodesign.com/builder-pattern.html

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

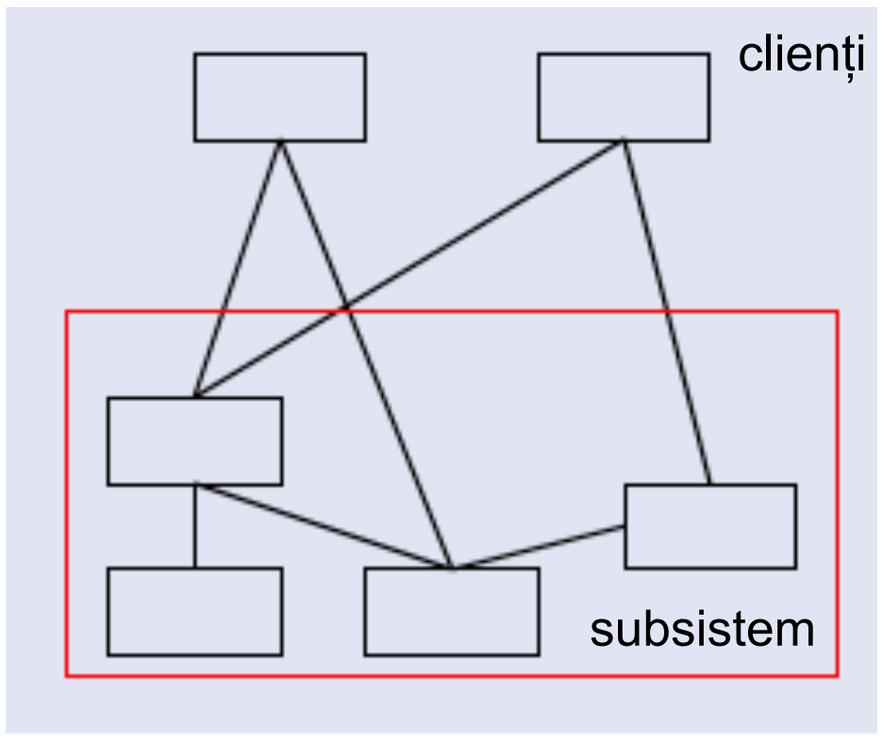


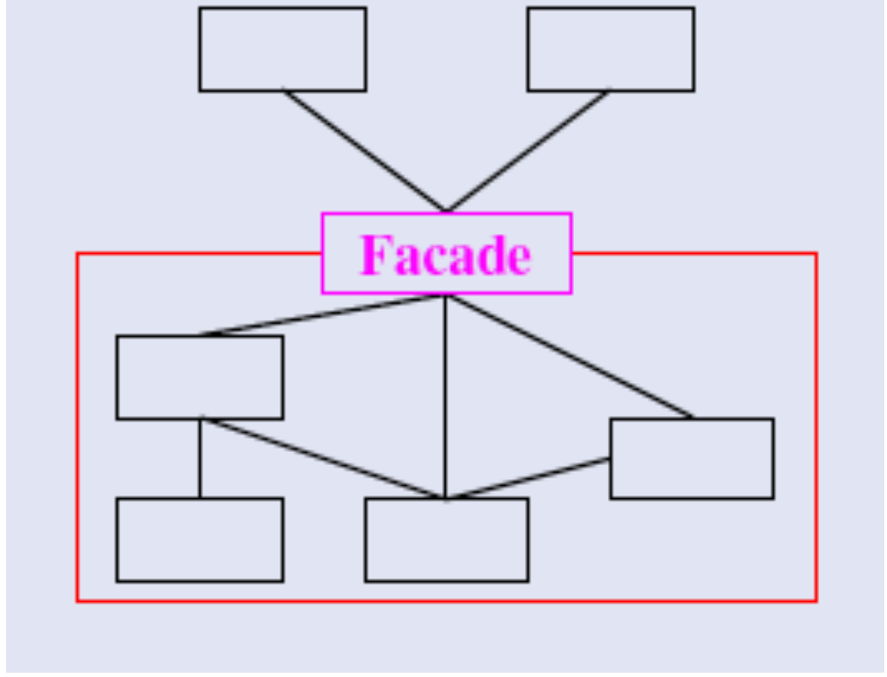
Facade - exemplu

```
/* 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

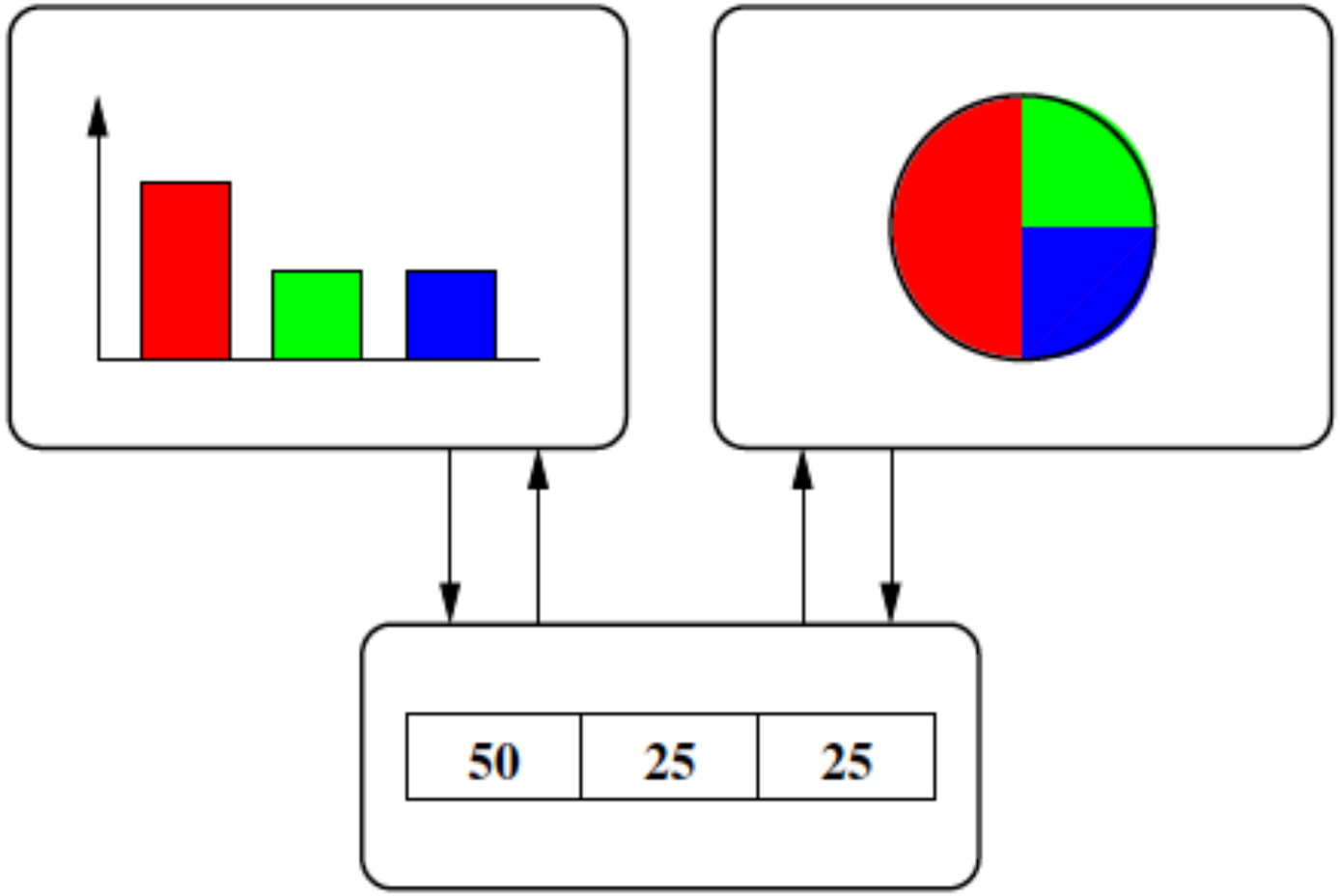
- observații
  - o interfață simplă la un subsistem complex
  - când sunt multe dependențe între clienți și subsistem, este redusă cuplarea



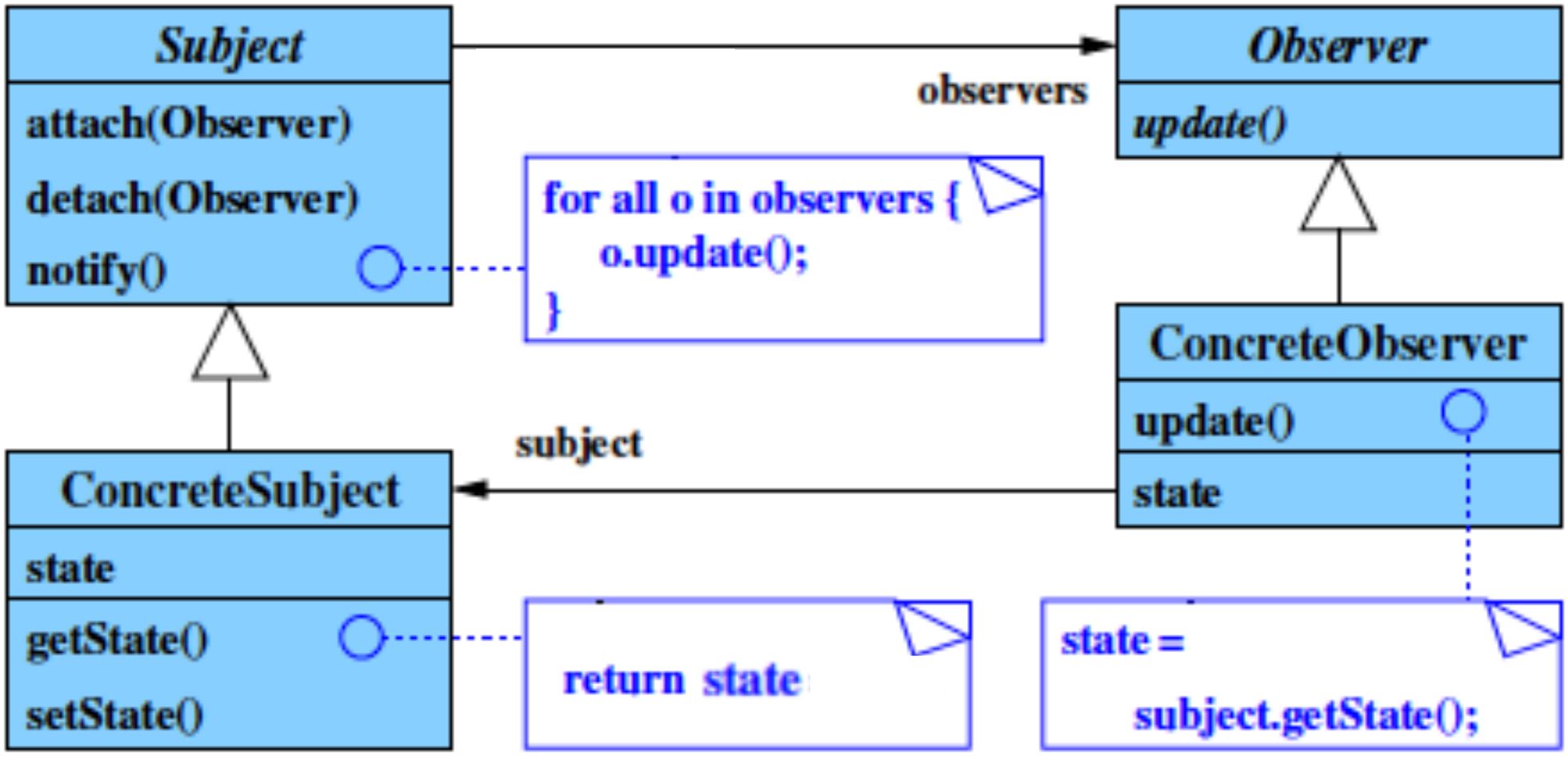


Șablonul comportamental: Observer

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



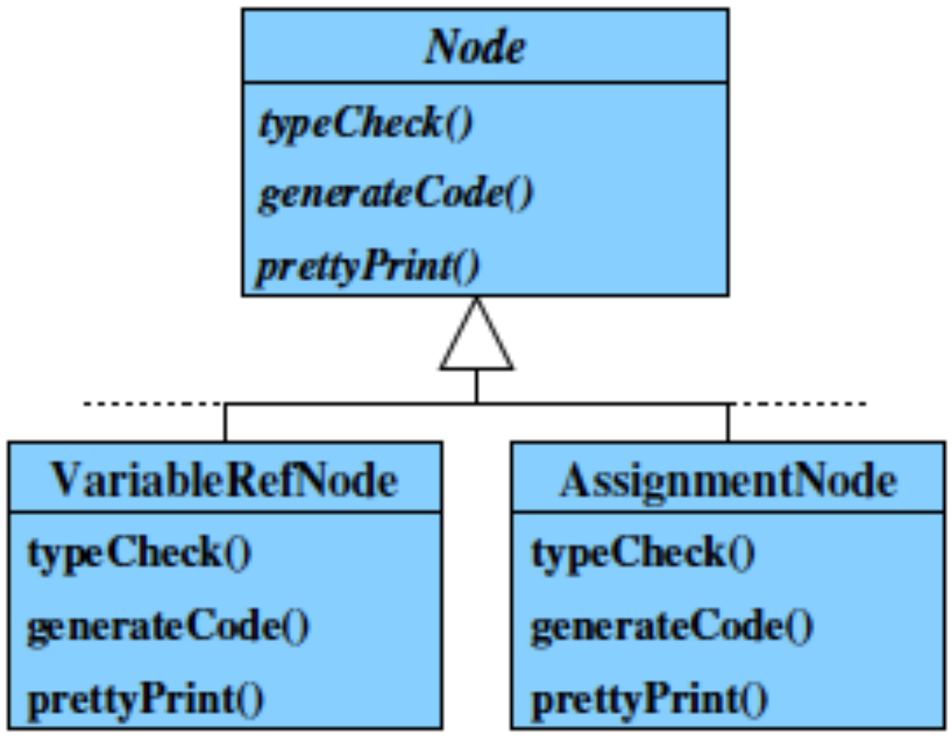
Observer - diagrama UML



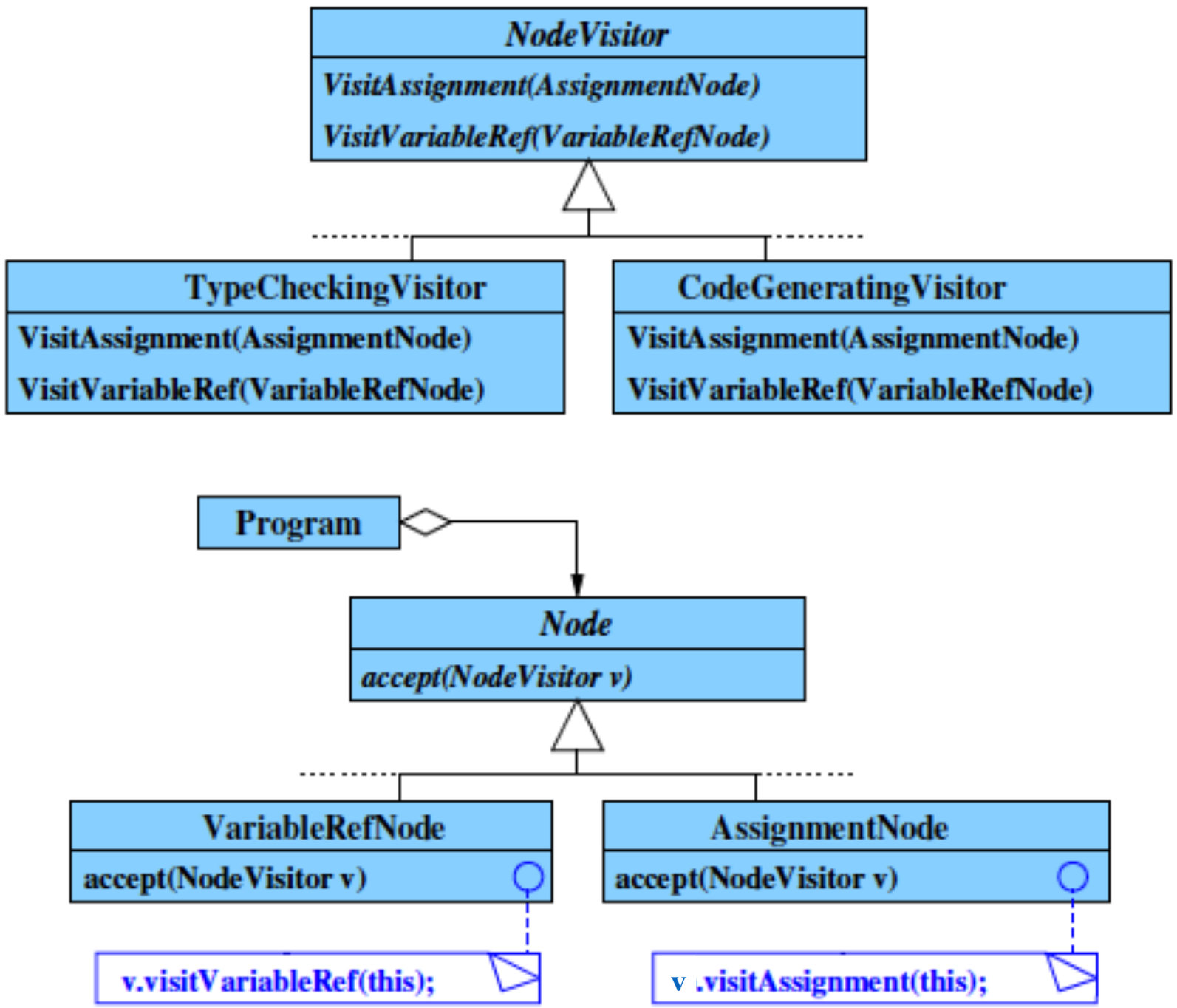
- propagarea schimbării stării / comunicare de tip broadcast
- notificare anonimă
- *Subject* și *Observer* sunt independenți (cuplare abstractă)
- observatorii sunt configurabili în mod dinamic

Șablonul comportamental: Visitor

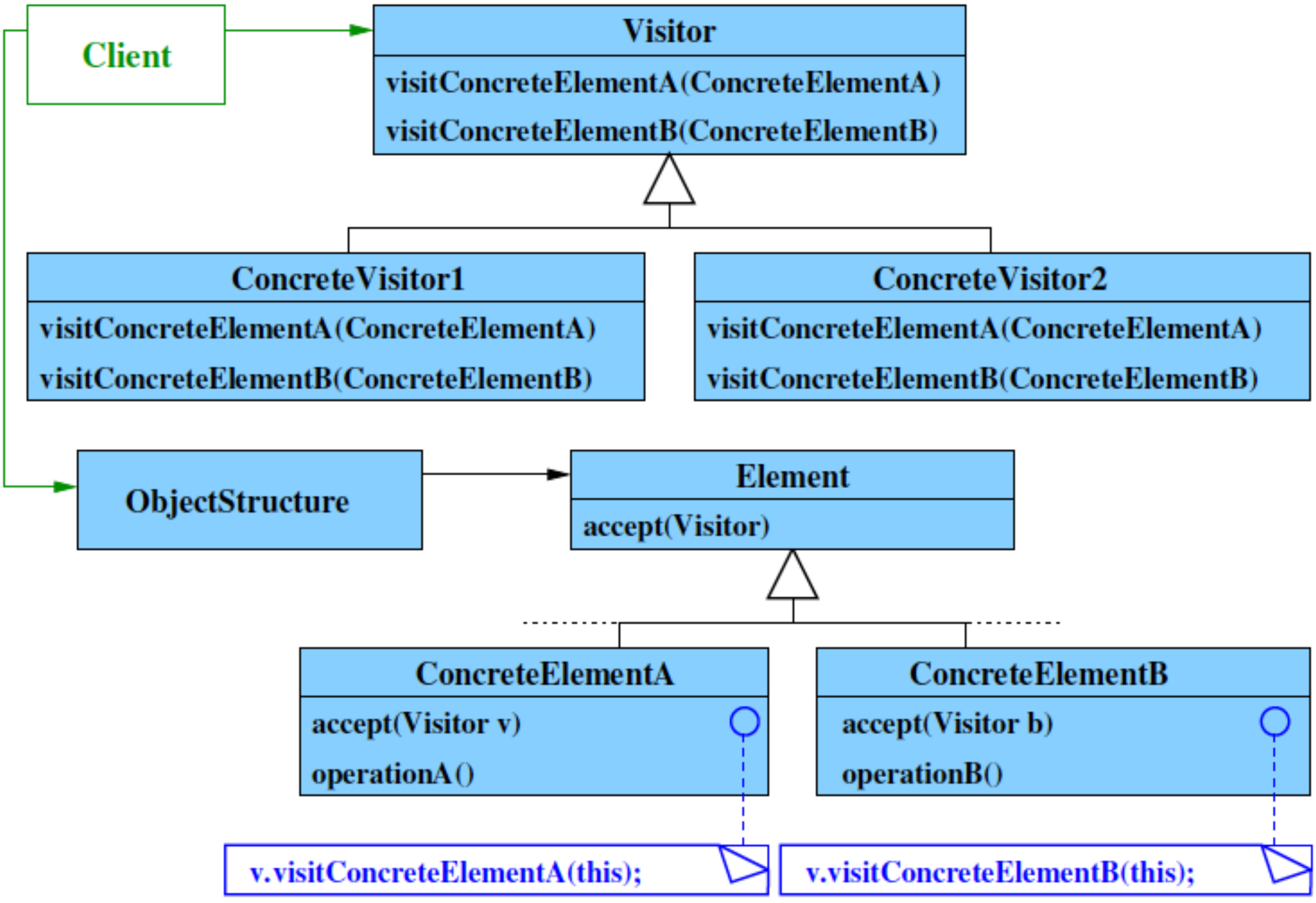
- reprezintă operații pe o structură de obiect
- adaugă noi operații, fără a modifica însă clasele
- de exemplu: procesarea arborelui sintactic într-un compilator (type checking, generare de cod, pretty print)
  - o primă versiune (neoptimă): operațiile sunt puse în clasele de tip nod



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-șabloane

Exemple de anti-șabloane (anti-patterns)

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

http://en.wikipedia.org/wiki/Anti-pattern#Software_engineering