

# Modele de proiectare a aplicațiilor de întreprindere

Cursurile 3 - 5

## Sumar

- Modele de proiectare creaționale
- Modelul de proiectare Singleton
- Modelul de proiectare **Multiton**
- Modelul de proiectare Simple Factory
- Modelul de proiectare Factory Method
- Modelul de proiectare **Abstract Factory**
- Builder
- Prototype
- **Object Pool**
- **Lazy Initialization**
- **Resource Acquisition Is Initialization (RAII)**

# Modele de proiectare creaționale

- Mecanisme de creare a obiectelor
- Obiectele sunt create în funcție de situație
- Controlează modul în care sunt create obiectele

# Modele de proiectare creaționale

- **Abstract Factory**
  - Crearea de familii de obiecte înrudite sau dependente, fără a preciza clasa concretă
- **Builder**
  - Crearea de obiecte complexe în mod incremental
- **Factory Method**
  - Definește o metodă pentru crearea de obiecte din aceeași familie
- **Prototype**
  - Clonarea instanțelor unui prototip existent
- **Singleton**
  - Crearea unei instanțe unice

# Modele de proiectare creaționale

- **Multiton (Registry Singleton)**
  - Crearea unei instanțe unice, pe baza unei chei
- **Simple Factory**
  - Crearea de obiecte din aceeași familie, pe baza unui tip
- **Lazy Initialization**
  - Crearea de obiecte este întârziată până la prima referire a acestora
- **Object Pool**
  - Obiectele sunt create anterior și sunt furnizate în momentul în care are loc o cerere

## Singleton

## Problema

- Unele clase au o singură instanță din punct de vedere conceptual
- Adăugarea de noi instanțe ar crește complexitatea și ar supraîncărca programele
- Exemple
  - Manager de configurare
  - Sistem de jurnalizare
  - Conexiune baza de date

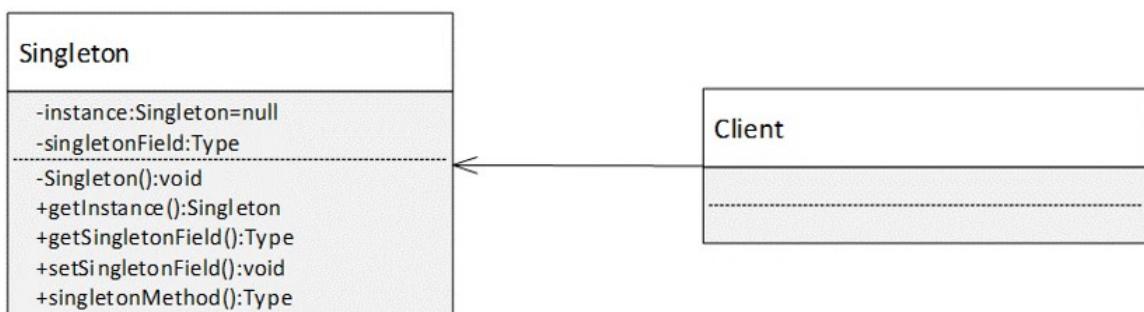
## Scop

- Existența unei singure instanțe a unei clase
- Existența unui punct unic de acces
- Inițializarea obiectelor se face la cerere sau la definire

# Implementare

- Crearea unui singur obiect cu asigurarea unei singure instanțe
- Reutilizarea acestuia
- Încapsularea codului care gestionează obiectele în cadrul unei clase
  - Constructorul este privat
  - Metode și câmpuri statice pentru accesul la instanța clasei
- Imposibilitatea de a instantia direct obiectul
- În medii cu fire de execuție multiple, sincronizarea creării obiectului

## Diagrama de clase



# Componente

- **Singleton**

- Definește mecanismul de creare și returnare a unei singure instanțe
- Include și membrii specifici clasei

- **Client**

- Utilizatorul obiectele de tip singleton

## Singleton vs. Clase/membri statici

- Singleton respectă principiile POO
- Un obiect singleton poate moșteni alte obiecte sau implementa interfețe
  - Se poate crea o structură de obiecte singleton
- Un clasă statică nu poate conține decât metode statice
- Obiectele de tip Singleton pot fi transmise ca parametri în funcții

## Implementări

- Inițializare timpurie (Eager initialization)
- Inițializare prin bloc static (Static block initialization)
- Inițializare întârziată (Lazy initialization)
- Thread Safe Singleton
- Clasă statică imbricată (Inner static helper class)
- Utilizare de constante enumerative (Enum)

## Dezavantaje

- Starea globală reduce modularitatea
- Creează dificultăți de testare

# Multiton

## Problema

- Coordonarea între utilizatorii unei stări complexe
- Stare unică fără a folosi variabile globale
- Flexibilitate crescută față de o tabelă asociativă statică
- Posibilitatea de extindere la un număr limitat de instanțe pentru fiecare cheie
- Ușurință de sincronizare

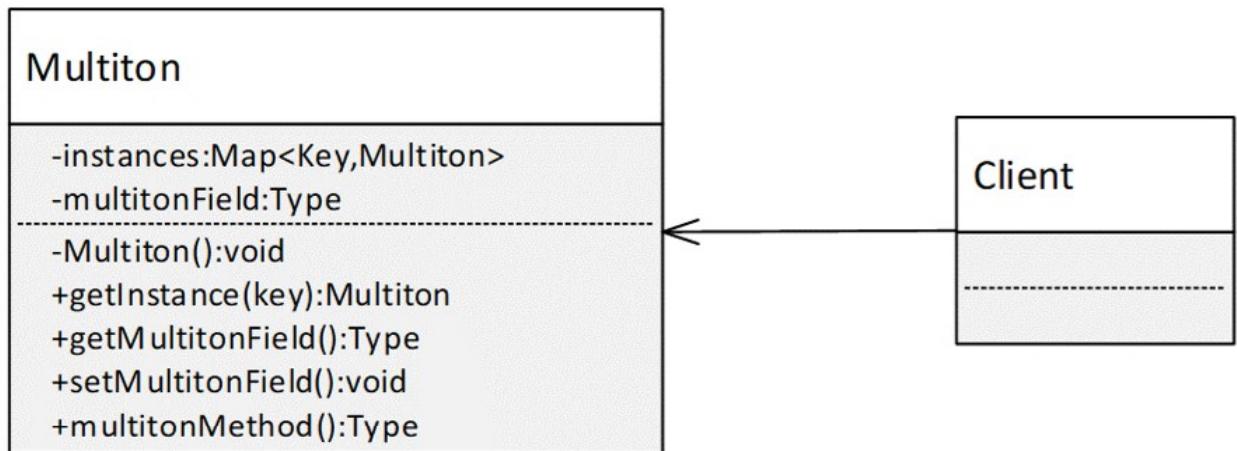
## Scop

- Existența unei singure instanțe de un anumit tip a unei clase
- Existența unui punct unic de acces
- Inițializarea obiectelor se face la cerere sau la definire

## Implementare

- Asigurarea unei singure instanțe per cheie
- Imposibilitatea de a instanția direct obiectele corespunzătoare unei chei
- În medii cu fire de execuție multiple, sincronizarea creării obiectului

## Diagrama de clase



## Componente

- **Multition**
  - Definește mecanismul de creare și returnare a unei singure instanțe dintr-o anumită categorie
  - În plus, include și membrii specifii clasei
- **Client**
  - Utilizatorul obiectele generate

## Dezavantaje

- Starea globală reduce modularitatea
- Creează dificultăți de testare

## Simple Factory

## Problema

- Crearea de obiecte pe baza unor informații care nu sunt cuprinse în descrierea clasei
- Obținerea de obiecte neomogene de la o singură sursă
- Gruparea creației de obiecte neomogene într-o locație centrală
- Delegarea tipului obiectului creat
- Conectarea de ierarhii de obiecte diferite fără a le cupla
- Exemple
  - Crearea de obiecte grafice
  - Crearea de documente
  - Crearea de produse catalog virtual

## Scop

- Definirea unei interfețe pentru crearea unui obiect
- Obiectele sunt create pe baza unui tip precizat

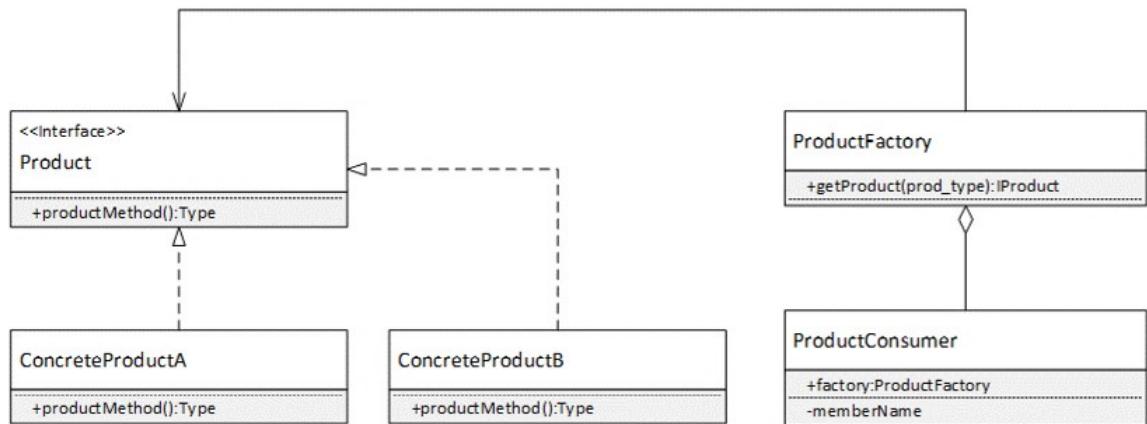
## Restricții de implementare

- Obiectele create trebuie să respecte o interfață comună
- Uzual, crearea obiectelor se realizează prin intermediul unei metode statice
  - Se evită necesitatea instanțierii unui obiect de tip *factory*

## Implementare

- Definește o interfață comună pentru crearea obiectelor
- Instanțierea obiectelor se realizează la nivelul clasei de tip Factory
- Clasa de tip Factory instantiază clasele pe baza unui identificator

# Diagrama de clase



# Componente

- **Product**

- Definește interfața obiectelor create prin intermediul clasei de tip ProductFactory

- **ConcreteProductA, ConcreteProductB**

- Implementează interfața Product

- **ProductFactory**

- Definește clasa de tip *factory* pentru crearea obiectelor de tip Product

- **ProductConsumer**

- Utilizează clasa ProductFactory pentru crearea de obiecte de tip Product

## Dezavantaje

- Modificarea codului existent pentru a folosi aceste model de proiectarea are implicații destul de ample
- Adăugarea unui nou obiect conduce la modificarea clasei de tip factory
- Obiectele sunt generate doar prin extindere
- Clasele nu pot fi extinse, constructorii fiind privați

## Factory Method

## Problema

- Crearea de obiecte pe baza unor informații care nu sunt cuprinse în descrierea clasei
- Obținerea de obiecte neomogene de la o singură sursă
- Gruparea creării de obiecte neomogene într-o locație centrală
- Delegarea tipului obiectului creat
- Conectarea de ierarhii de obiecte diferite fără a le cupla

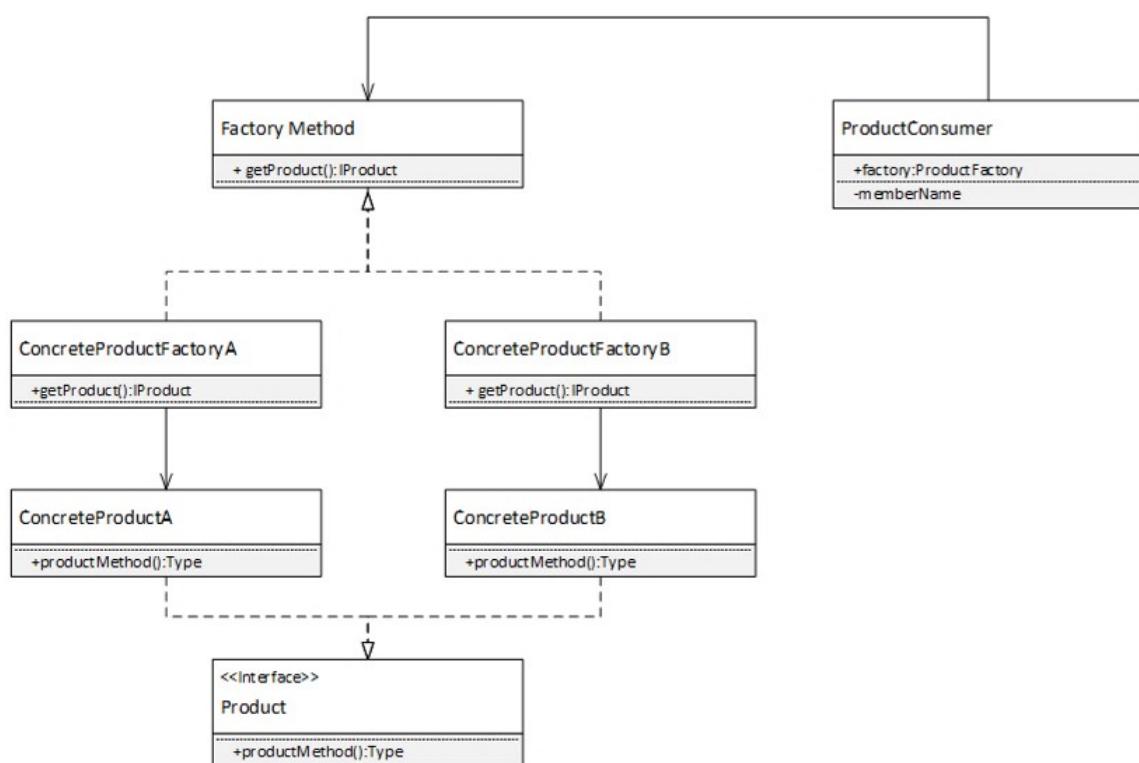
## Scop

- Definirea unei interfețe pentru crearea unui obiect, dar subclasele decid care clasă va fi instantiată
- Metoda delegă instanțierea obiectelor către subclase
- Definirea unui constructor "virtual"
- Operatorul new nu este folosit direct

# Implementare

- Obiectele create trebuie să respecte o interfață comună
- Crearea obiectelor se realizează prin intermediul unei metode statice
  - Se evită necesitatea instanțierii unui obiect de tip factory
- Se definește o interfață comună pentru crearea obiectelor
- Subclasele decid ce clasă să instanțieze
- Instanțierea obiectelor se realizează la nivelul subclaselor
- Clientul este responsabil cu instanțierea obiectelor

## Diagrama de clase



# Componente

- **Product**
  - Definește interfața obiectelor create prin intermediul metodei de tip *factory*
- **ConcreteProductA, ConcreteProductB**
  - Implementări ale interfeței Product
- **FactoryMethod**
  - Declară metoda de tip *factory* care creează obiecte de tip Product
    - Poate avea și o implementare implicită
- **ConcreteProductFactoryA, ConcreteProductFactoryB**
  - Redefinesc metoda de tip *factory* pentru crearea de obiecte concrete de tip Produs
- **ProductConsumer**
  - Utilizează clasele de tip FactoryMethod pentru crearea de obiecte de tip Product

# Abstract Factory

## Problema

- În vederea asigurării portabilității unei aplicații, trebuie încapsulate componentele specifice platformelor pe care acestea vor rula
- Exemple
  - Sistemul de fișiere
  - Sistemul de gestiune a ferestrelor
  - Conexiunile la baze de date

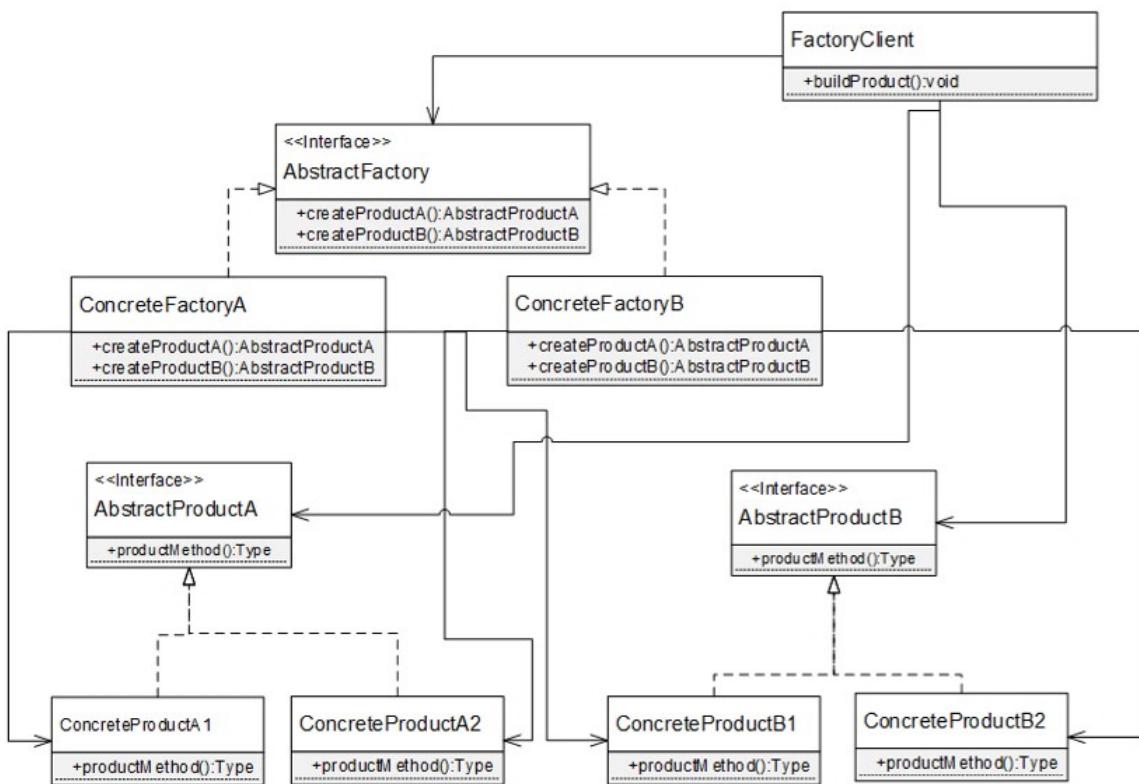
## Scop

- Pune la dispoziție o interfață pentru crearea de familii de obiecte sau obiecte înrudite, fără precizarea claselor concrete
- Se prezintă sub forma unei ierarhii de platforme și o suită de produse specifice

# Implementare

- Obiectele concrete produse se bazează pe o interfață comună
- Clasele concrete pe baza cărora sunt construite instanțele de obiecte se bazează pe o interfață comună

## Diagrama de clase



# Componente

- **AbstractFactory**
  - Declară o interfață pentru crearea de produse abstracte
- **ConcreteFactoryA, ConcreteFactoryB**
  - Implementează operațiile pentru crearea obiectelor concrete
- **AbstractProductA, AbstractProductB**
  - Declară o interfață de tipul produselor
- **ProductA1, ProductA2, ProductB1, ProductB2**
  - Defineste clase concrete pentru produse ce vor fi create prin clasele corespunzătoare
- **FactoryClient**
  - Utilizează interfețele declarate de AbstractFactory și AbstractProduct

# Avantaje

- Clientul este decuplat de clasa care generează obiectele
- Crearea obiectelor este controlată
- Posibilitatea de extindere a ierarhiei de clase

## Dezavantaje

- Complexitatea poate părea ridicată
- Adăugarea unui obiect produs care extinde interfața presupune modificarea tuturor implementărilor concrete

## Builder

## Problema

- Crearea unor obiecte care au nevoie de un număr mare de constructori cu formă variabilă
- Obiectele create au un număr de câmpuri obligatorii și un număr de câmpuri opționale care pot apărea în orice combinație
- Posibilitatea de a ignora o parte dintre proprietăți, care nu sunt relevante pentru o anumită instanță
- Utilizarea unei abordări tradiționale poate conduce la:
  - anti-modelul *constructor telescopic*
  - un obiect care poate ajunge în stări invalide
- Uzual, obiectele create sunt inițializate doar prin constructor și proprietățile acestora nu pot fi modificate ulterior
- Exemple
  - Crearea de meniuri în aplicații în funcție de rolul utilizatorilor

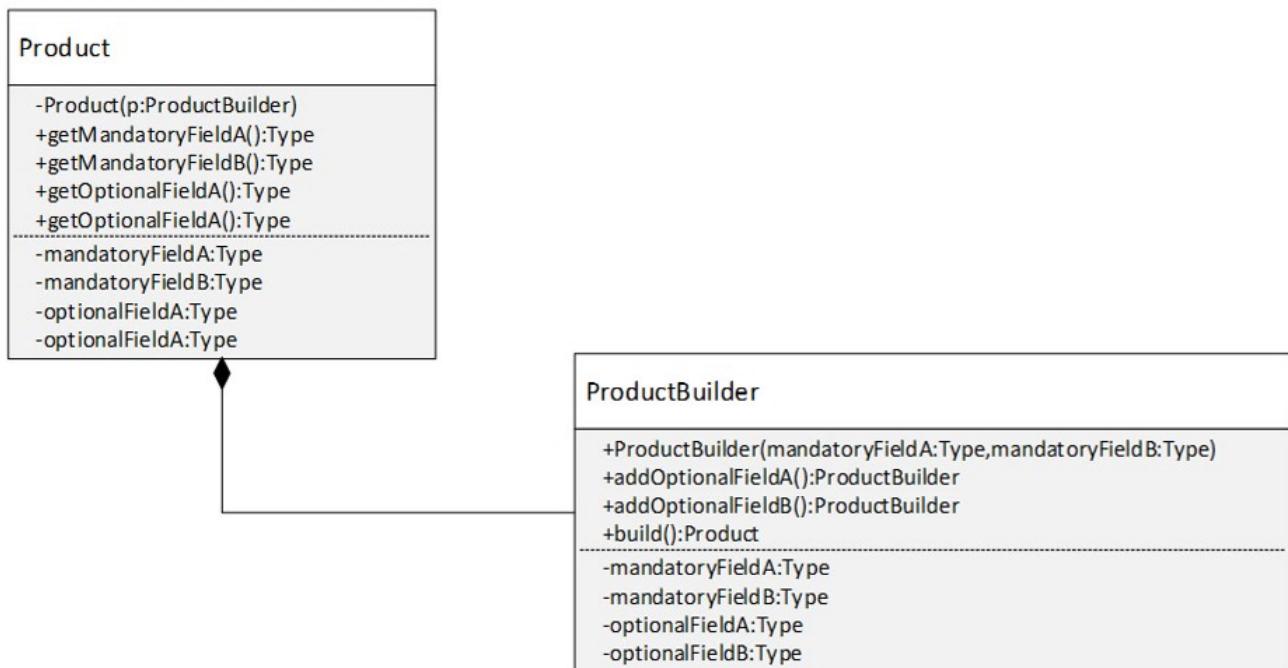
## Scop

- Separarea creării unui obiect de reprezentarea acestuia
- Același proces de creare poate conduce la reprezentări diferite
- Plecînd de la o reprezentare complexă, se creează obiecte specifice contextului

# Implementare

- Obiectul rezultat nu poate fi construit decât prin intermediul obiectului de tip Builder
- Obiectul de tip Builder poate adăuga proprietățile optionale în orice ordine
- Dacă obiectul produs este imutabil, se vor utiliza câmpuri finale pentru proprietăți
- Dacă proprietățile nu suportă decât un număr limitat de valori se vor folosi enumerări

## Diagrama de clase



# Componente

- **ProductBuilder**
  - Clasă pentru crearea obiectelor de tip Product
- **Product**
  - Obiecte create prin intermediul clasei ProductBuilder
- **Client**
  - Creează obiecte Product prin intermediul clasei Builder

# Avantaje

- Crearea obiectelor complexe se realizează independent de părțile componente
- Flexibilitate în crearea obiectelor

## Dezavantaje

- Uzual, un obiect construit pe baza unui builder nu este poate fi modificat ulterior
- Un obiect construit pe baza unui builder nu este ușor persistabil
  - Practic, nu este un POJO
- Anumite atribute pot fi omise

## Implementare particulară: Step Builder

- Comportament de tip asistent (wizard)
- Se stabilește ordinea de adăugare a proprietăților
- Clientul intervine la fiecare pas

# Prototype

## Problema

- Necesitatea de creare a unor obiecte, cu reutilizarea resurselor alocate și reducerea timpilor de reinicializare
- Costul ridicat al creării obiectelor (resurse)

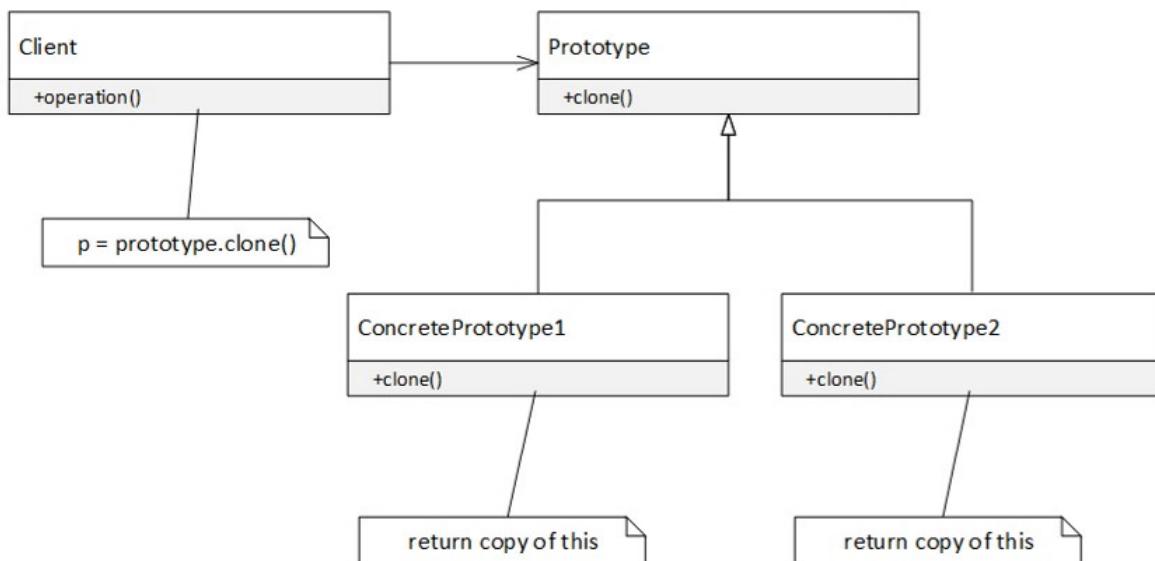
## Scop

- Definirea de obiecte ce vor fi create pe baza unei instanțe prototip
- Crearea de obiecte noi pe baza prototipului
- Se creează doar o instanță a clasei ce va fi utilizată în viitor

## Implementare

- Se definește o interfață care pune la dispoziție o metodă pentru crearea unei copii a instanței prototip
- Când este necesar un nou obiect, se va crea o copie a obiectului prototip

# Diagrama de clase



# Componente

- **Prototype**
  - Declară o interfață pentru propria clonare
- **ConcretePrototypeA, ConcretePrototypeB**
  - Implementează o operație pentru propria clonare
- **Client**
  - Creează obiecte noi prin cereri de clonare către prototip
  - Doar primul prototip va fi creat prin constructor

# Object Pool

## Problema

- Costul instantierii obiectelor este ridicat
- Frecvența de instantiere este mare
- La un moment dat, numărul de obiecte instanțiate utilizate este redus
- Exemple
  - Fire de execuție
  - Conexiuni la baze date

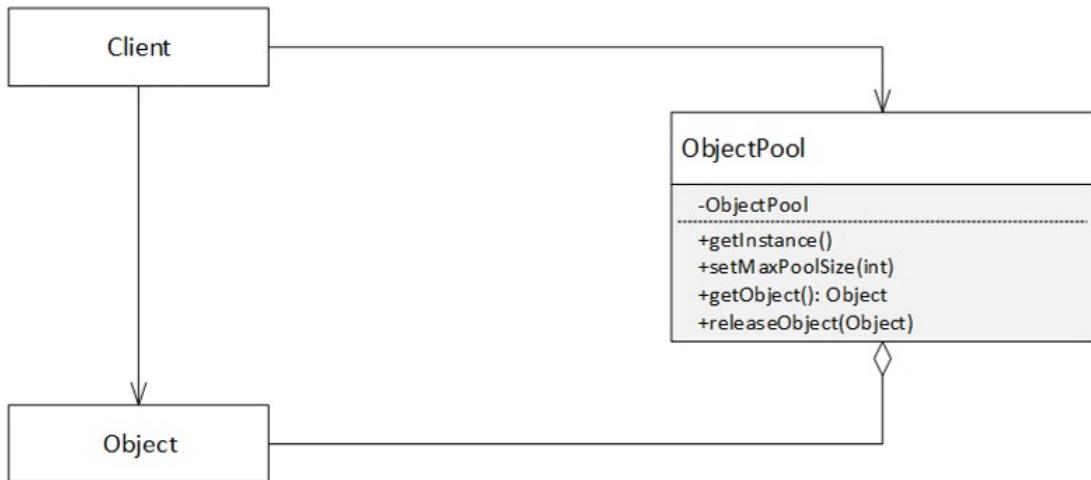
## Scop

- Gestiunea obiectelor reutilizabile
- Îmbunătățirea performanțelor la crearea obiectelor prin utilizarea unui grup de obiecte inițializate în prealabil

## Implementare

- Numărul de obiecte gestionate poate fi
  - Fix
  - Modificat în funcție de utilizare
- Uzual, clasa este gestionată prin intermediul unui singleton
- Obiectele create sunt urmărite, pentru a putea fi reutilizate

## Diagrama de clase



## Componente

- **Object**
  - Clasa asociată obiectelor reutilizabile
- **ObjectPool**
  - Clasa care gestionează obiectele reutilizabile
- **Client**
  - Clasa care utilizează obiectele reutilizabile

## Dezavantaje

- Număr limitat de resurse
  - În anumite situații
- Sincronizare
- Resurse expirate
  - Rezervate, dar neutilizate

## Lazy Initialization

## Descriere

- Crearea de obiecte este întîrziată pînă la prima referire a acestora
- Exemple
  - Încărcarea din baze de date
  - Preluarea datelor din rețea

Resource Acquisition Is  
Initialization (RAII)

## Descriere

- Asocierea resurselor cu ciclul de viață al obiectelor
- Inițializarea resurselor se realizează la crearea obiectelor
- Eliberarea resurselor are loc la distrugerea obiectelor
- Exemple
  - Deschiderea/închiderea fișierelor
  - Alocarea/eliberarea memoriei
  - Conectarea/Deconectarea serviciu

## Sumar, modele creaționale

- **Abstract Factory**
  - Crearea de familii sau mulțimi de obiecte
- **Builder**
  - Crearea de obiecte pas cu pas
- **Factory Method**
  - Delegarea claselor derivate în vederea instanțierii obiectelor

## Sumar

- Modele structurale
- Decorator
- Flyweight
- Adapter
- Façade
- Bridge
- Composite
- Proxy
- Aggregate
- Private class data
- Pipes and Filters

## Modele structurale

- Compoziție clase și obiecte
- Decuplare interfețe și clase
- Suport pentru identificarea și descrierea relațiilor dintre entități
- Abordează modul în care clasele și obiectele sunt compuse și structuri complexe
- La nivel de clasă sau obiect

# Modele structurale

- **Adapter**
  - Adaptează interfața unei clase la o altă interfață
- **Bridge**
  - Decouplează modelul abstract de implementare
- **Composite**
  - Agregarea a mai multor obiecte similare într-o ierarhie arborescentă
- **Decorator**
  - Extinde responsabilitățile unui obiect în mod dinamic

# Modele structurale

- **Façade**
  - Furnizează o interfață unică pentru interfețele unui subsistem
- **Flyweight**
  - Utilizarea partajării pentru utilizarea eficientă a unui număr mare de obiecte
- **Proxy**
  - Furnizează o componentă vidă pentru un obiect pentru a controla accesul la acesta

# Decorator

## Problema

- Necesitatea extinderii în mod dinamic a unei clase
- Clasa existentă nu trebuie să fie modificată
- Utilizarea unei abordări tradiționale, prin derivarea clasei, duce la ierarhii complexe ce sunt greu de gestionat
  - Derivarea adaugă comportament nou doar la compilare
- Exemple
  - Fluxurile de intrare/ieșire: decorate cu alte fluxuri compatibile
  - Comportamentul ferestrelor grafice în timpul execuției

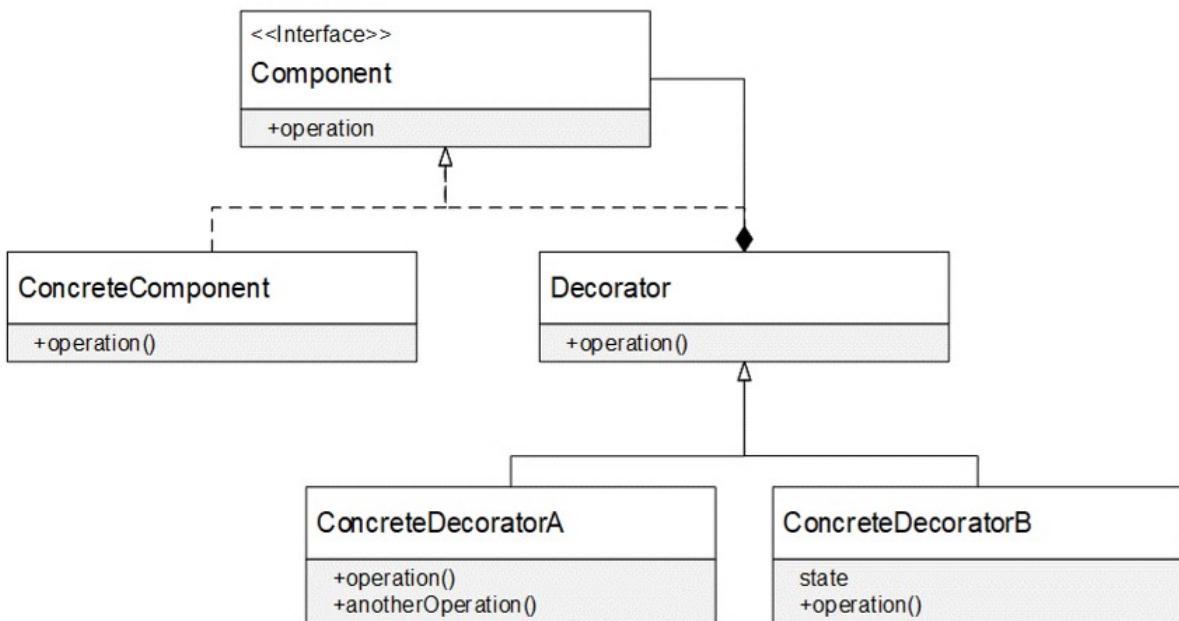
## Scop

- Extinderea (decorarea) la execuție a funcționalității unor obiecte, independent de alte instanțe ale aceleiași clase
- Alternativă flexibilă la extinderea claselor pentru noi funcționalități , fără afectarea interfeței

## Implementare

- O interfață comună implementată atât de componenta care urmează să fie extinsă, cât și de clasa decorator
- Decoratorul include o referință către componenta care urmează să fie extinsă
- Clasa decorator este extinsă cu noile funcționalități specifice

# Diagrama de clase



## Componente

- **Component**
  - Declără interfața obiectelor ce pot fi decorate cu noi funcții
- **ConcreteComponent**
  - Definește obiectele ce pot fi decorate
- **Decorator**
  - Gestionează o referință de tip Component către obiectul decorat
  - Metodele moștenite apelează implementările specifice din clasa obiectului referit
  - Poate defini o interfață comună claselor de tip decorator
- **ConcreteDecoratorA, ConcreteDecoratorB**
  - Clase concrete care adaugă funcții noi obiectului referit

## Avantaje

- Extinderea funcționalității a unui obiect particular se face dinamic, la execuție
- Decorarea este transparentă pentru utilizator
  - Clasa moștenește interfața specifică obiectului
- Decorarea se face pe mai multe niveluri, transparent pentru utilizator
- Nu impune limite privind un număr maxim de decorări
  - Obiectul poate să fie extins prin aplicarea mai multor decoratori

## Dezavantaje

- Un decorator este un container pentru obiectul inițial
  - Nu este identic cu obiectul încapsulat
- Utilizarea excesiva generează o mulțime de obiecte care arată la fel dar care se comportă diferit
  - Dificil de înțeles și verificat codul

# Flyweight

## Problema

- Existența unei mulțimi de obiecte
  - Structură internă complexă
  - Ocupă un volum mare de memorie
- Obiectele au atribute comune însă o parte din starea lor variază
  - Memoria ocupată poate fi minimizată prin partajarea stării fixe între ele
- Exemple
  - Caractere într-un editor de texte
  - Obiecte grafice decorative în aplicații

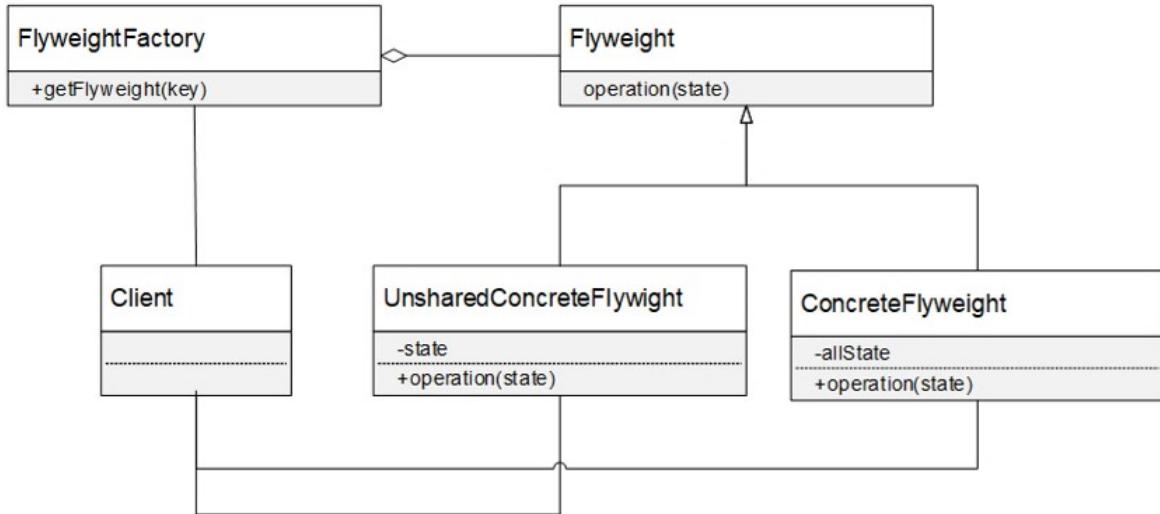
## Scop

- Utilizarea partajării pentru a gestiona un număr foarte mare de obiecte în mod eficient
- Obiectele au o stare internă și o stare externă
  - Starea externă variază
- Starea internă va fi partajată

## Implementare

- Starea obiectelor este gestionată prin structuri externe
  - Numărul de obiecte efectiv create este minimizat
- Utilizarea unui obiect
  - Aplicarea stării variabile unui obiect existent

# Diagrama de clase



# Componente

- **Flyweight**
  - Interfață care declară comportamentul prin intermediul căruia obiectele recepționează și reacționează la starea externă
- **ConcreteFlyweight**
  - Implementează interfața Flyweight
  - Stochează starea internă a obiectelor
  - Starea externă este gestionată prin intermediul metodelor din interfață
- **UnsharedConcreteFlyweight**
  - Clasa implementează interfața de tip Flyweight, dar nu permite partajarea stării
  - Pot exista clase derivate din aceasta care vor partaja starea
- **FlyweightFactory**
  - Construiește și gestionează obiecte de tip Flyweight
  - Menține o colecție de obiecte diferite, astfel încât acestea să fie create o singură dată
- **Client**
  - Utilizează obiectele de tip Flyweight
  - Gestionează referințele și starea externă a obiectelor

## Avantaje

- Reducerea memoriei ocupate de obiecte prin
  - partajarea acestora între clienți
  - partajarea stării acestora între obiecte de același tip
- Pentru a gestiona corect partajarea obiectelor de tip Flyweight între clienți și fire de execuție,
  - acestea trebuie să fie nemodificabile

## Dezavantaje

- Necesitatea unei analize a pentru determina starea internă și starea externă
- Efectele implementării modelului sunt vizibile pentru soluții în care numărul de obiecte este mare
- Performanțele sănt influențate de numărul categoriilor de obiecte

# Adapter

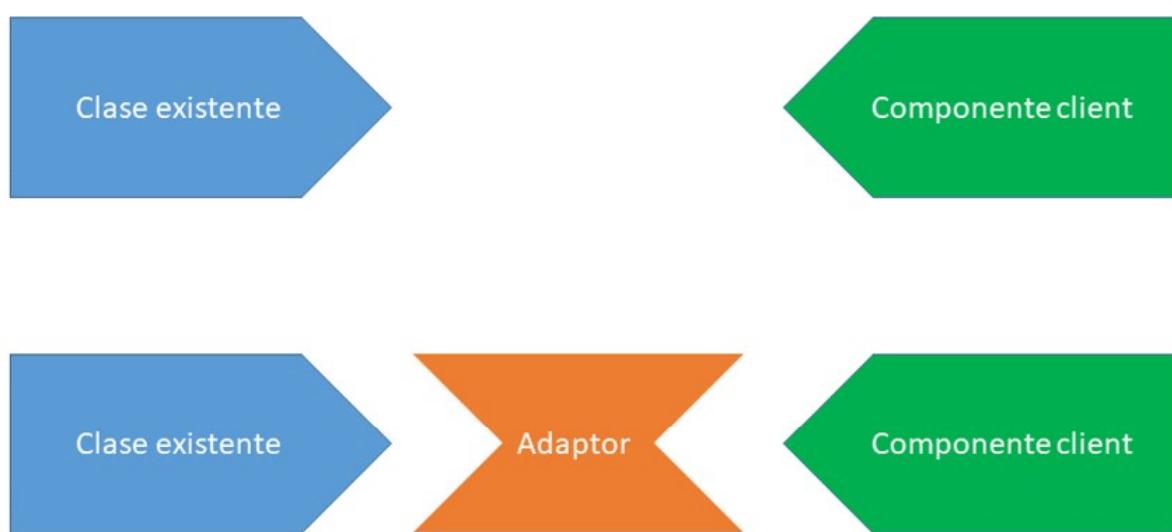
## Problema

- Utilizarea împreună a unor clase care nu au o interfață comună
- Transformarea datelor dintr-un format în altul

## Scop

- Clasele sunt adaptate la un nou context
- Apelurile către interfața clasei sunt mascate de interfața adaptorului

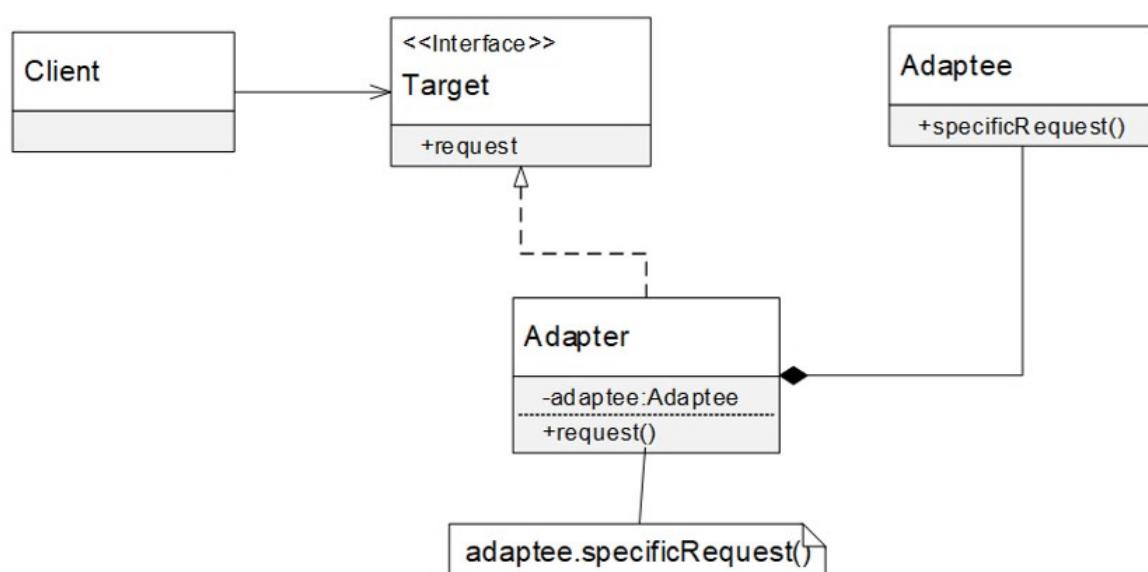
## Soluția



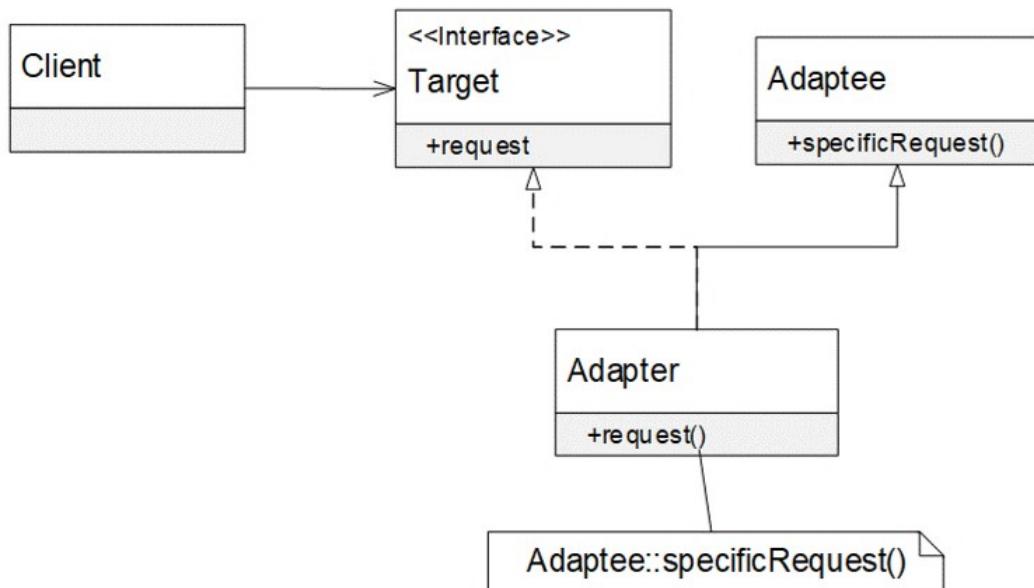
# Implementare

- Se declară o interfață ce permite utilizarea claselor existente într-un alt context
  - Clasele existente nu se modifică
- Adaptor la nivel de obiect
- Adaptor la nivel de clasă
  - Moștenire multiplă

## Diagrama de clase (1)



## Diagrama de clase (2)



## Componente

- **Adaptee**

- Clasa existentă ce trebuie adaptată la o nouă interfață

- **Target**

- Declara interfața specifică noului domeniu

- **Adaptor**

- Adaptează interfața clasei existente la cea a clasei din noul context

- **Client**

- Componenta care utilizează interfața specifică noului domeniu

## Avantaje

- Clasele existente (la client și la furnizor) nu sunt modificate pentru a putea fi folosite într-un alt context
- Se adaugă doar un nivel intermediar
- Pot fi definite cu ușurință adaptoare pentru orice context

## Dezavantaje

- Adaptorul de clase se bazează pe derivare multiplă
  - Nu este posibil în anumite limbi de programare (Java, C# etc.)

# Façade

## Problema

- Soluția existentă conține o mulțime de clase
- Execuția unei funcții presupune apeluri multiple de metode aflate în aceste clase
- Utilă în situația în care framework-ul crește în complexitate și nu este posibilă rescrierea lui pentru simplificare

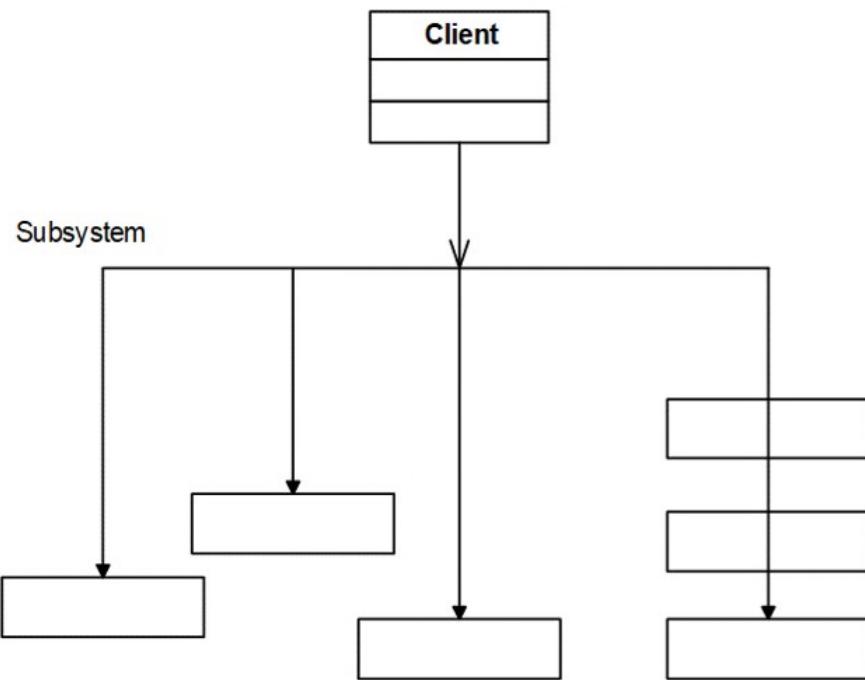
## Scop

- Definirea unei interfețe comune pentru accesul simplificat la componentele unui subsistem existent
- Interfața este creată special pentru atingerea acestui obiectiv

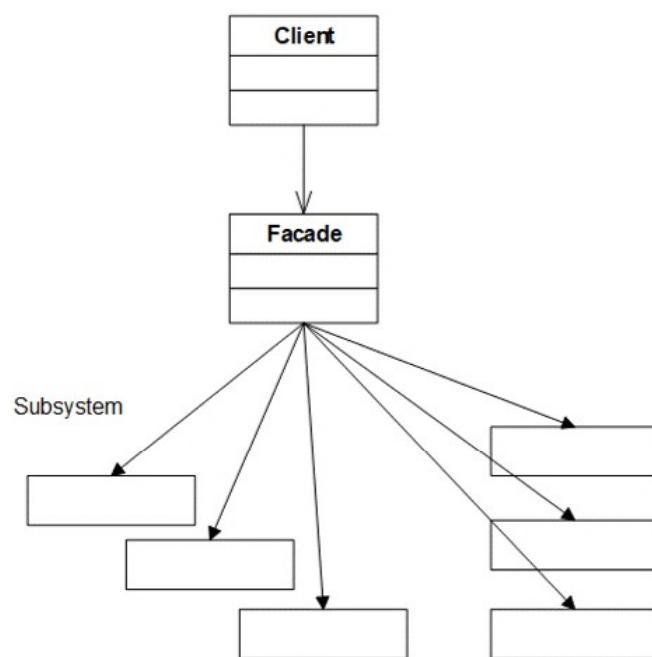
## Implementare

- Se construiește un nivel intermediar
  - Permite apelul facil al metodelor din mai multe interfețe
- Apelurile către multiplele interfețe sunt mascate de acest nivel intermediar
  - Interfață nouă
- Clasele existente nu se modifică

## Context



## Diagrama de clase



# Componente

- **Subsystem**

- Structura existentă de componente ce pun la dispoziție diferite interfețe

- **Façade**

- Declară o interfață simplificată pentru contextul existent

- **Client**

- Componenta care utilizează interfața specifică noului domeniu

# Avantaje

- Sistemul existent nu se modifică

- Se adaugă un nivel intermediar ce ascunde complexitatea

- Pot fi definite cu ușurință metode care să simplifice orice situație

- Implementează principiul *Least Knowledge* (Legea lui Demeter)

- Reducerea interacțiunilor între obiecte la nivel de *prietenii* (și nu cu prietenii prietenilor)

## Dezavantaje

- Crește numărul de clase container
- Crește complexitatea codului prin ascunderea unor metode
- Poate avea un impact negativ asupra performanțelor aplicației

## Bridge

## Problema

- Extinderea claselor abstracte prin implementarea comportamentului specific nu permite
  - modificările independente la interfață/implementare
  - compunerile independente
- Necesitatea separării interfețelor de implementare
- Exemple
  - Drivere
  - Framework-uri pentru interfață grafică

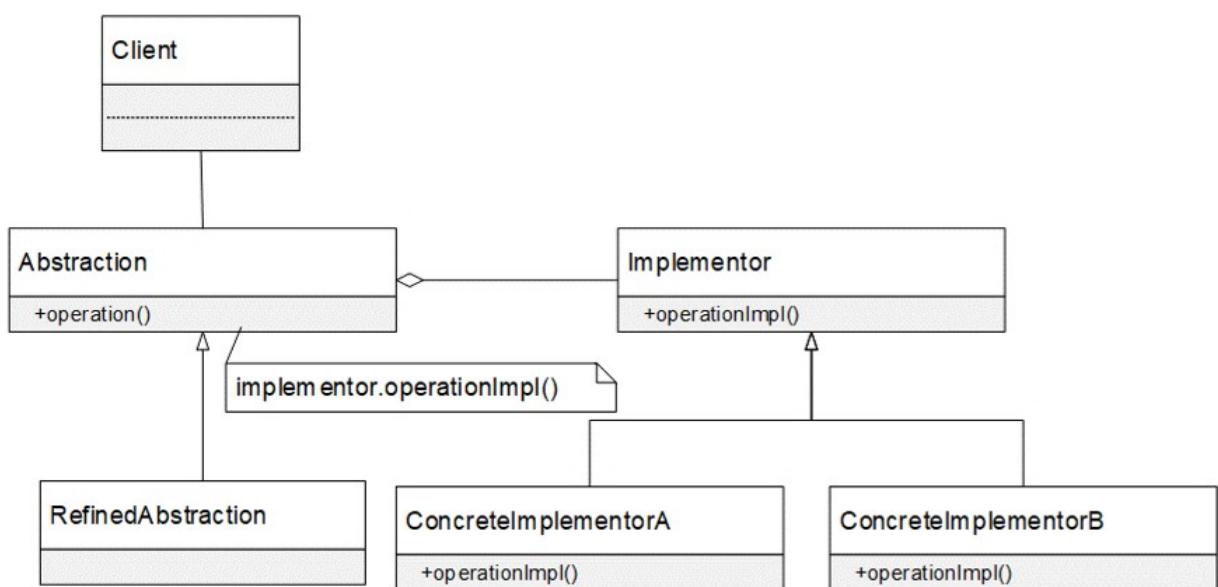
## Scop

- Decuplarea abstractizării (interfețelor) de implementare
- Modificarea în mod independent a interfețelor și a implementării
- Crearea de ierarhii pentru interfață și implementare

# Implementare

- Model de proiectare de nivel înalt
- Abstractizarea decuplează
  - Clientul
  - Interfață
  - Implementarea

## Diagrama de clase



# Componente

- **Abstraction**

- Declară o interfață
- Pune la dispoziție metode de nivel înalt
- Include o referință a unui obiect de tip *Implementor*

- **RefinedAbstraction**

- Extinde *Abstraction*

- **Implementor**

- Declară interfața pentru clasele de implementare
- Pune la dispoziție metode de nivel scăzut
- Metodele din *Abstraction* invocă metodele din *Implementor*

- **ConcreteImplementorA, ConcreteImplementorA**

- Implementează interfața *Implementor*
- Include comportamentul specific

## Avantaje

- Asigurarea independenței interfețelor și a implementărilor

## Dezavantaje

- Complexitate
- Clasele și interfețele existente trebuie modificate

## Composite

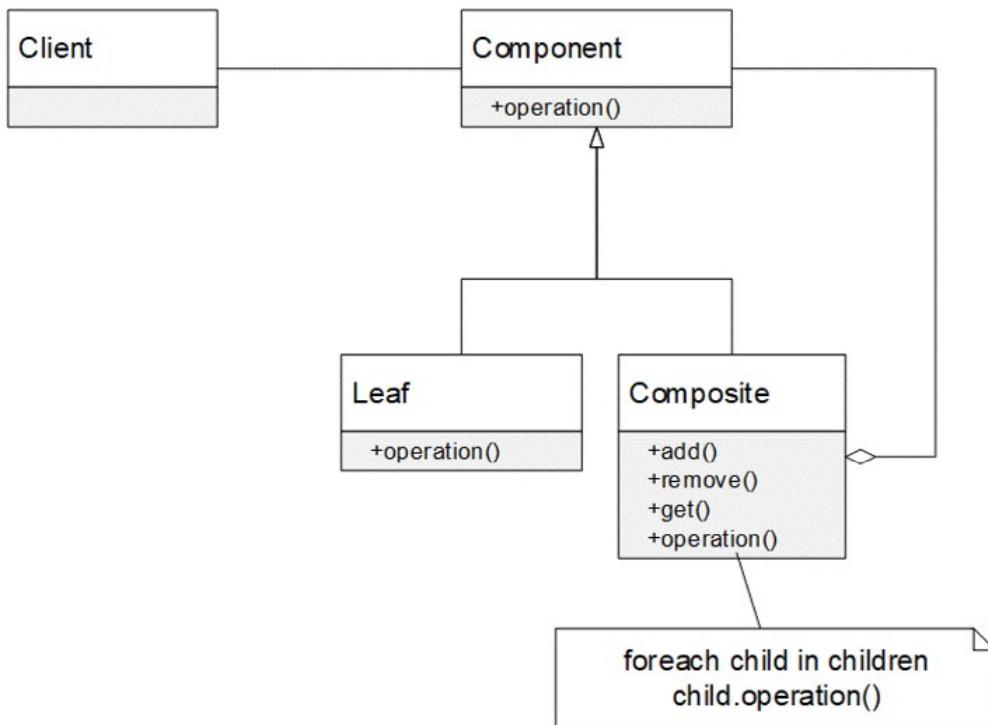
## Problema

- Soluția conține o mulțime de clase aflate în relație ierarhica și acestea trebuie tratate unitar
- Se construiesc structuri arborescente în care nodurile intermediare și cele frunza sunt tratate unitar
- Exemple
  - Editoare grafice: figurile grafice simple sau compuse
  - Sistemul de fișiere: fișiere și directoare

## Scop

- Componerea obiectelor în structuri arborescente pentru reprezentarea ierarhiilor parte-întreg
- Clientii tratează uniform atât obiectele individuale, cât și pe cele compuse

## Diagrama de clase



## Componente

- **Component**
  - Declară interfața obiectelor aflate în compoziție
- **Leaf**
  - Asociată nodurilor frunză din compoziție
- **Composite**
  - Componenta compusă, include noduri fiu
  - Implementează metode prin care sunt gestionate nodurile fiu
- **Client**
  - Manipulează obiectele din ierarhie

## Avantaje

- Nu este necesară rescrierea claselor existente
- Permite gestiunea facilă a unor ierarhii de clase ce conțin atât primitive cât și obiecte compuse
- Simplificarea codului: obiectele din ierarhie sunt tratate unitar
- Adăugarea de noi componente se realizează facil

## Proxy

# Problema

- Interconectarea de API-uri diferite
  - aceeași mașină
  - în rețea
- Definirea unei interfețe între diferite framework-uri
- Exemple:
  - Servicii invocate la distanță
  - Gestiunea referințelor la obiecte (smart pointers)

# Scop

- "**Controls and manage access to the object they are protecting"**
- Furnizarea unui substituent pentru un obiect, pentru a controla accesul la acesta
- Utilizarea unui nivel de intermediar gestiunea unui acces distribuit sau controlat
- Adăugarea unui nivel intermediar pentru a proteja componenta reală de complexitatea nejustificată

# Tipuri de proxy

- **Virtual Proxy**

- Gestionează crearea și inițializarea unor obiecte
  - Procese costisitoare
- Crearea în momentul accesului sau partajarea unei instanțe între mai mulți clienți

- **Remote Proxy**

- Asigură o instantă virtuală locală pentru un obiect aflat la distanță (Java RMI, servicii Web etc.)

- **Protection Proxy**

- Controlează accesul la anumite obiecte
- Controlează accesul la metodele unui obiect

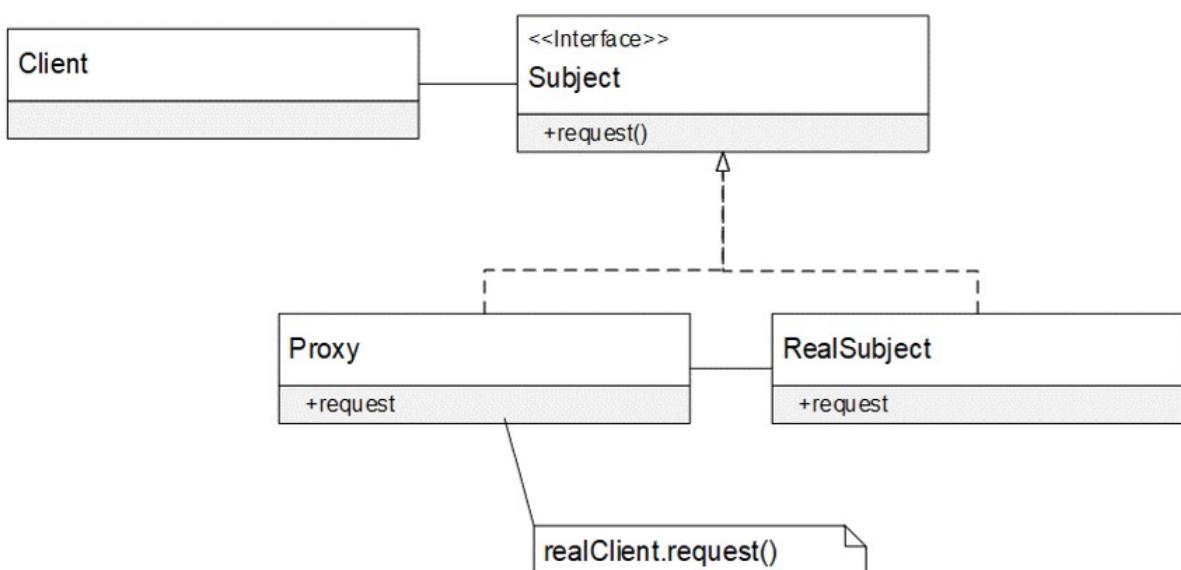
- **Smart References**

- Gestionează automat referințele către un obiect și eliberează resursele atunci când acesta nu mai este utilizat

- **Cache Proxy**

- Gestionează eficient apelurile costisitoare ale unui obiect concret
- Îmbunătățirea performanțelor

## Diagrama de clase



# Componente

- **Subject**

- Declără interfața obiectului real la care se face conectarea
- Interfața este implementată și de proxy

- **Proxy**

- Implementează interfața obiectului real
- Gestionează referința către obiectul real
- Controlează accesul la obiectul real

- **RealSubject**

- Obiectul real către gestionat prin intermediul proxy-ului

- **Client**

- Utilizează serviciile puse la dispoziție de către RealSubject, prin intermediul Proxy-ului

# Aggregate

Problema

Private class data

## Problema

- Încapsularea tuturor atributelor unui obiect
- Protejarea stării clasei prin minimizarea vizibilității atributelor acesteia

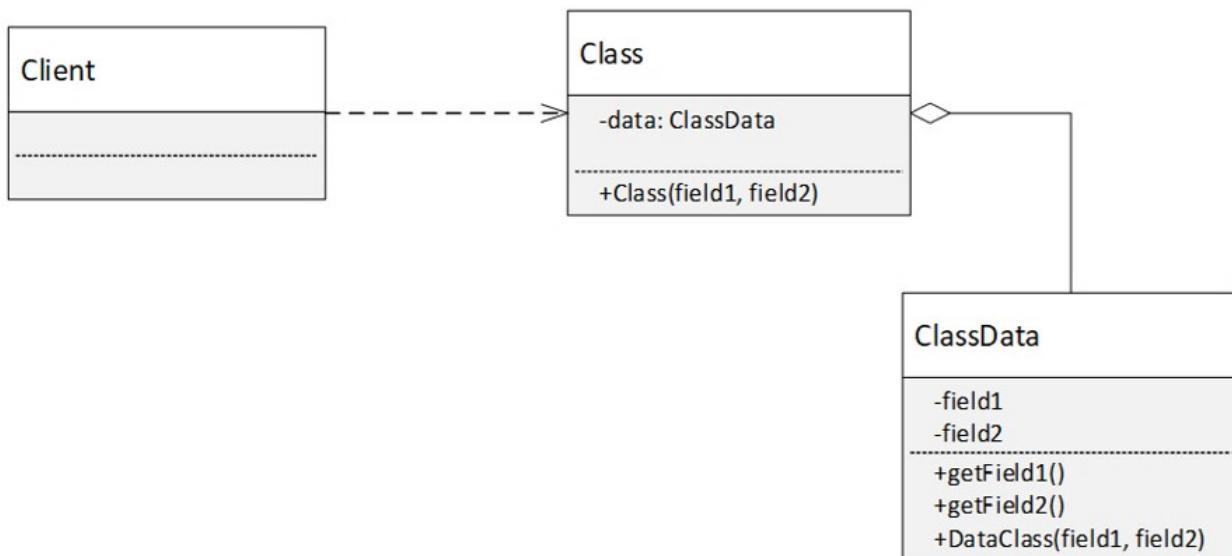
## Scop

- Controlul accesului la attributele clasei (protecție la modificare)
- Separarea datelor de metodele care le utilizează
- Încapsulează inițializarea datelor clasei

# Implementare

- Datele sunt încapsulate într-o clasă
- Clasa asociată datelor pune la dispoziție metode de acces (get)
- Clasa principală are un membru de tipul clasei datelor
- Membrul este inițializat în constructorul clasei principale

## Diagrama de clase



# Componente

- **Class**
  - Clasa principală
- **ClassData**
  - Clasa asociată datelor din clasa Class
  - Datele sunt private
  - Accesibile prin getter-i
- **Client**

# Pipes and filters

# Problema

- Model arhitectural
- Descompunerea unui proces complex
  - Activități (task-uri) – *Filters*
  - Canale de comunicare – *Pipes*
- Componentele implementează o interfață comună

## Sumar

- |  |   |
|--|---|
| <ul style="list-style-type: none"><li>• Modele comportamentale</li><li>• Strategy</li><li>• Observer</li><li>• Chain of Responsibility</li><li>• Command</li><li>• Template Method</li><li>• State</li></ul> | <ul style="list-style-type: none"><li>• Interpreter</li><li>• Iterator</li><li>• Mediator</li><li>• Visitor</li><li>• <i>Blackboard</i></li><li>• <i>Null Object</i></li><li>• <i>Specification</i></li></ul> |
|--|---|

# Modele comportamentale

- Distribuție responsabilități
- Interacțiune între clase și obiecte

# Modele comportamentale

- **Chain of Responsibility**
  - Gestionează tratarea unui eveniment de către un obiect
- **Command**
  - Încapsulează o cerere sub forma unui obiect, permitând parametrizarea acestuia
- **Memento**
  - Permite salvarea și restaurarea stării unui obiect
- **Observer**
  - Definește o relație de tipul unul la mai mulți între obiecte, astfel încât la modificarea stării unui obiect, celelalte obiecte să fie notificate

# Modele comportamentale

- **State**

- Permite modificarea comportamentelor obiectelor la schimbarea stării acestora

- **Strategy**

- Definește și încapsulează o familie de algoritmi

- **Template Method**

- Încapsulează un algoritm ai cărui pași sunt implementați în clase derivate

# Modele comportamentale

- **Interpreter**

- Definește o reprezentare a unui limbaj cu o gramatică simplă și un mecanism de interpretare a expresiilor

- **Iterator**

- Permite gestionarea accesului la elementele din cadrul unei colecții

- **Mediator**

- Definește un obiect care încapsulează modul în care interacționează mai multe obiecte asociate

- **Visitor**

- Definește operații care pot fi aplicate pe elementele unei structuri neomogene

# Modele comportamentale

- ***Null Object***

- Încapsulează absența unui obiect prin furnizarea unei alternative care să poată fi substituită

- ***Blackboard***

- Mai multe subsisteme specializate combină cunoștințele pentru a construi o soluție eventual parțială sau aproximativă

- ***Specification***

- Crearea unei specificații capabile să precizeze dacă un obiect candidat îndeplinește anumite criterii

# Strategy

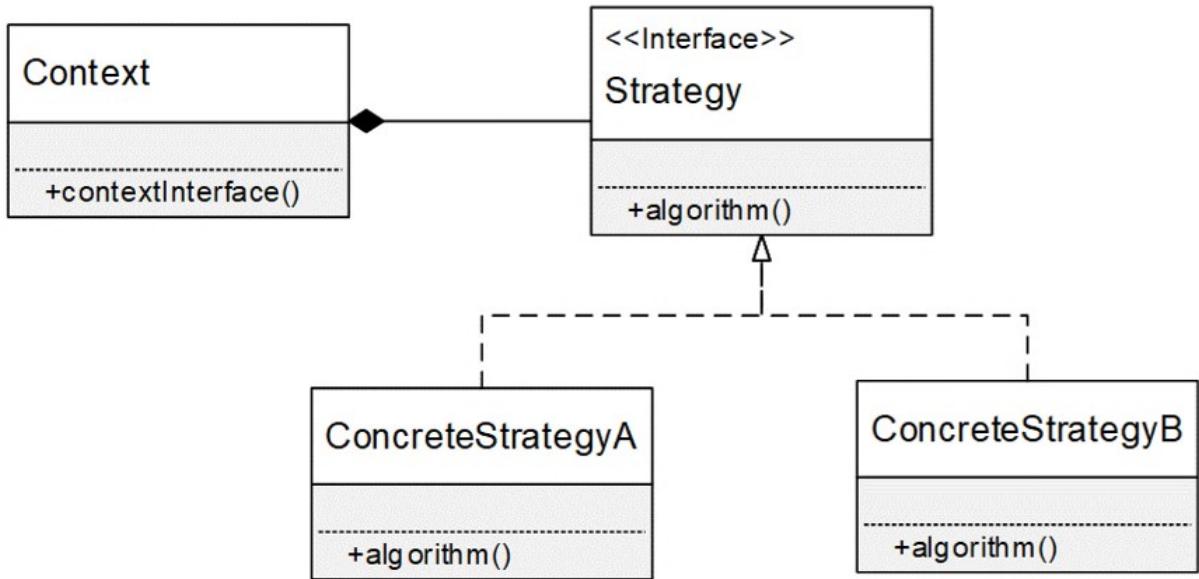
## Problema

- Selectarea unui algoritm/funcție, care să fie utilizată dinamic, pentru procesarea unor date
- Algoritmul/funcția se alege pe baza unor condiții descrise la execuție în funcție de context (date de intrare)
- Clasa existentă nu trebuie să fie modificată
- Utilizarea unei abordări tradiționale (includerea în clasă a tuturor metodelor posibile) conduce la ierarhii complexe, greu de gestionat.
- Prin derivare se adaugă comportament nou doar la compilare
- Exemplu
  - Algoritmi de sortare
  - Criterii de comparare
  - Salvarea fișierelor în diferite formate
  - Compresia fișierelor

## Scop

- Definirea unei familii de algoritmi, încapsularea acestora, cu posibilitatea de utilizarea interschimbabilă
- Posibilitatea de variere independentă a algoritmilor față de clientul care le utilizează
- Abstractizarea este declarată în cadrul interfețelor, iar detaliile implementării în clasele derivate

## Diagrama de clase



## Componente

- **Strategy**
  - Declără interfața pe care o implementează toți algoritmii
- **ConcreteStrategyA, ConcreteStrategyB**
  - Implementează algoritmul folosind interfața *Strategy*
- **Context**
  - Gestionează o referință de tip *Strategy*
  - Gestionează contextual în care au loc prelucrările

## Avantaje

- Selectarea algoritmului se realizează în mod dinamic, la execuție
- Permite definirea de noi algoritmi, independent de context
- Nu există limitări în ceea ce privește numărul de algoritmi care pot fi definiți

## Observer

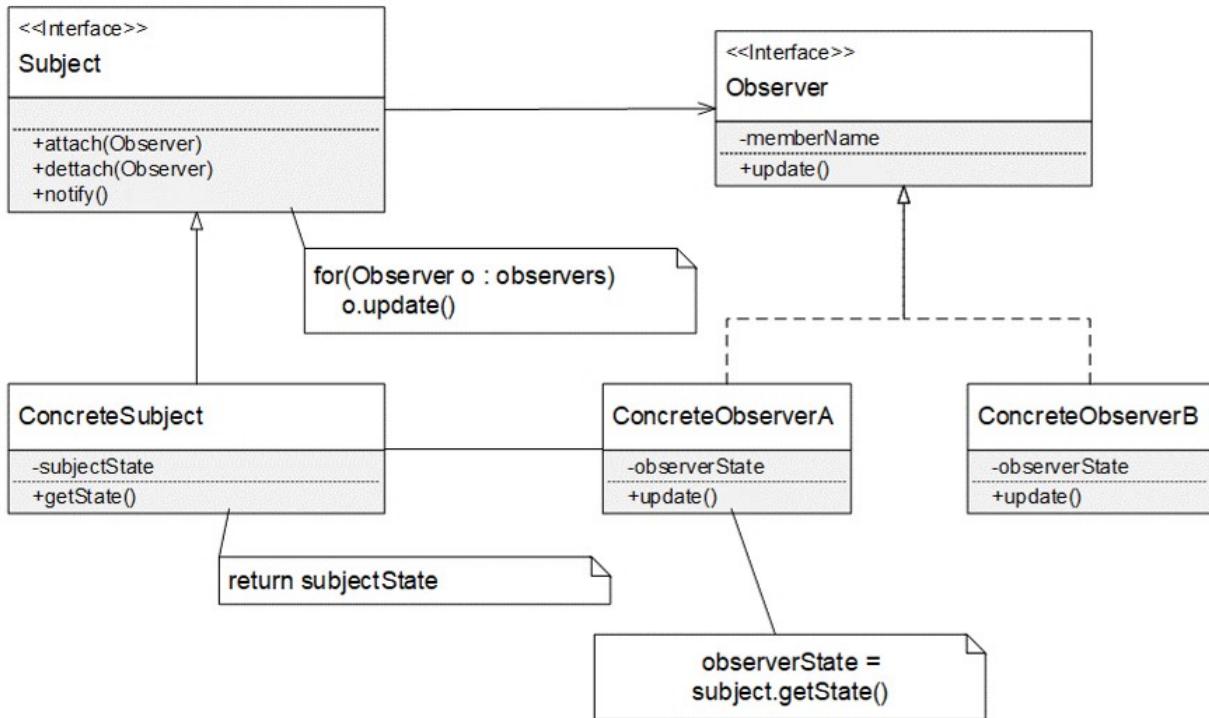
## Problema

- Există componente care trebuie să fie notificate la producerea unui eveniment
- Componentele se abonează/înregistrează la acel eveniment
  - Modificare de stare/acțiune
- La producerea unui eveniment pot fi notificate mai multe componente
- Exemple
  - Gestirea evenimentelor la nivelul interfeței cu utilizatorul
  - Notificările transmise de aplicații

## Scop

- Definirea unei dependențe de tipul unul la mulți între obiecte astfel încât, la schimbarea stării unui obiect, toate obiectele dependente să fie notificate
- Componentele care își modifică starea sunt încapsulate într-o abstractizare (*Subject*)
- Componentele variabile, cele care sunt notificate, sunt încapsulate într-o ierarhie de clase de tip *Observer*

# Diagrama de clase



# Componente

## • Subject

- Declară interfața obiectelor observabile
- Include metode pentru adăugarea și eliminarea obiectelor de tip *Observer*

## • ConcreteSubject

- Gestionează starea unui obiect și notifică observatorii în momentul schimbării acesteia

## • Observer

- Declară interfața de actualizare a observatorilor, atunci cînd sunt notificați

## • ConcreteObserverA, ConcreteObserverB

- Implementează metodele care sănătățe execuțate în urma notificării
- Gestionează starea obiectelor observator

# Modalități de notificare a observatorilor

- **Push**

- Obiectul notifică observatorul și transmite și datele asociate acestuia

- **Pull**

- Obiectul notifică observatorul
  - Observatorul cere datele cînd are nevoie de acestea

## Avantaje

- Externalizarea/delegarea funcțiilor către componente de tip observator
  - Dau soluții la anumite evenimente, independent de proprietarul evenimentului
- Conceptul este integrat în modelul arhitectural Model View Controller (MVC)
- Implementează conceptul POO de cuplare scăzută (loose coupling)
  - Obiectele sunt interconectate prin notificări și nu prin instanțieri de clase și apeluri de metode

# Chain of Responsibility

## Problema

- Existența unor cereri de prelucrare înlăncuite și a unor componente de prelucrare a cererilor
- Necesitatea eficientizării acestui proces fără încapsularea tuturor relațiilor, asocierilor și precedențelor dintre componentele de prelucrare
- Exemple:
  - Tratarea evenimentelor în cadrul ierarhiilor de componente grafice
  - Actualizarea interfeței grafice
  - Tratarea excepțiilor

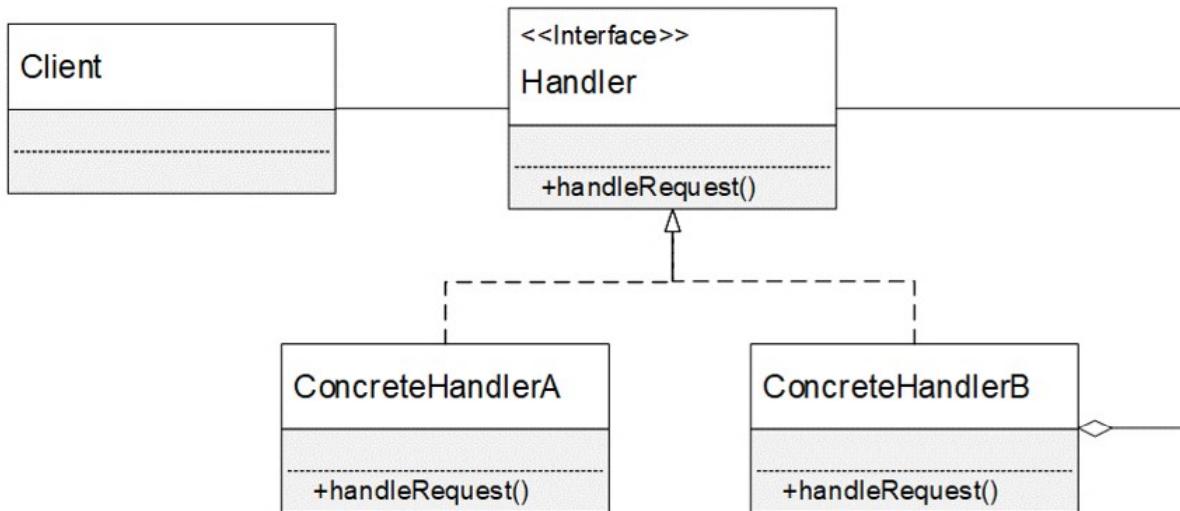
## Scop

- Evitarea cuplării inițiatorului unei cereri de receptorul acestuia
- Utilizarea unui mecanism de transmitere a responsabilității către alt obiect care poate asigura prelucrarea cererii
- Obiectele fie prelucrează cererea, fie o transmit mai departe
- Parcurserea obiectelor se realizează recursiv, printr-o listă înlănțuită

## Implementare

- Clientul transmite o cerere către un obiect responsabil cu prelucrarea
- Numărul și tipul obiectelor care pot prelucra cererea nu este cunoscut de la început și este configurat în mod dinamic
- Se definește o ierarhie de clase în care clasele derivate pot prelucra cererile clientilor
- În cazul în care un obiect nu poate prelucra o cerere, acesta va delega operația către obiectul următor
- Este necesară tratarea situației în care cererea nu poate fi prelucrată

## Diagrama de clase



## Componente

- **Handler**

- Declără interfața pentru gestiunea cererilor care urmează să fie procesate

- **ConcreteHandlerA, ConcreteHandlerB**

- Preia și procesează cererile adresate
  - Dacă nu poate procesa cererea, aceasta va fi transmisă următorului obiect din listă

- **Client**

- Inițiază cererea către primul obiect din lista de obiecte

# Command

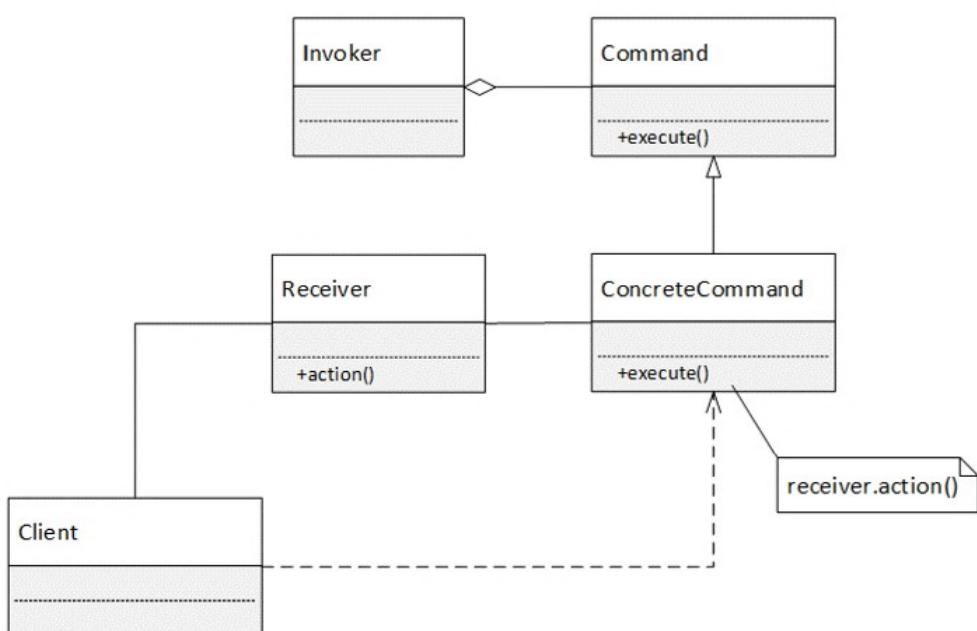
## Problema

- Necesitatea transmiterii unor cereri, fără a avea informațiile cu privire la
  - Operația invocată
  - Receptorul cererii
- Exemple
  - Meniuri de sistem (acțiune și revenire la starea anterioară)
  - Cozi de mesaje

# Scop

- Încapsulează cererea într-un obiect
- Suport pentru
  - Parametrizarea cererilor clienților
  - Înlățuirea cererilor
  - Jurnalizarea cererilor
  - Revenirea asupra acțiunilor efectuate

## Diagrama de clase



# Componente

- **Command**

- Declara o interfață pentru execuția unei operații

- **Receiver**

- Obiectul receptor
- Include logica sau datele necesare unei anumite comenzi
- Cunoaște modul de execuție a unei operații asociate unei cereri

- **ConcreteCommand**

- Definește o legătură între obiectul receptor și o acțiune
- Implementează execuția comenzi prin apelul operației corespunzătoare din receptor

- **Invoker**

- Gestionează obiectele de tip *Command*
- Invocă o comandă pentru execuția unei acțiuni

- **Client**

- Creează o comandă concretă și îi asociază un receptor
- Comenzile se transmit obiectelor de tip *Invoker*

## Avantaje

- Decouplează clasele care invocă operațiile de cele care le execută
- Suport pentru revenirea la starea anterioară
- Permite combinarea comenzi simple în comenzi complexe

# Template Method

## Problema

- Implementarea unui algoritm presupune o secvență predefinită și fixă de pași
- Anumiți pași nu se modifică
- Posibilitatea unor implementări diferite ale unor pași ai algoritmului
- Exemple:
  - Framework-uri care includ componente care pot fi personalizate de către dezvoltatori

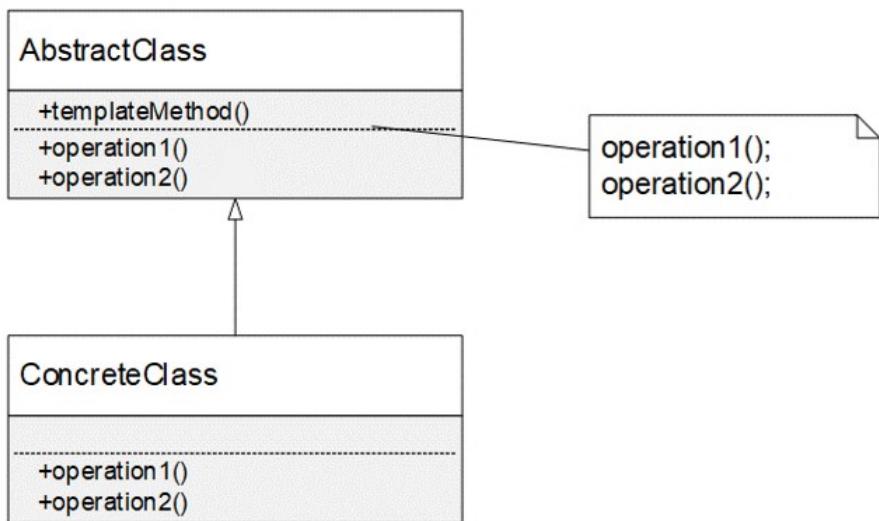
## Scop

- Definește structura unui algoritm
- Permite claselor derivate să implementeze și să adapteze pașii acestuia

## Implementare

- Metoda template
  - Metoda care definește structura algoritmului
  - Apelează metodele concrete de definesc pașii algoritmului
- Structura algoritmului nu este modificabilă
- Pot fi extinse/modificate metodele care implementează fiecare pas
- Implementează modelul de proiectare *inversarea controlului* (Inversion of Control)
  - sau principiul Hollywood: "Don't call us, we'll call you"

# Diagrama de clase



# Componente

- **AbstractClass**
  - Definește metodele abstracte asociate pașilor unui algoritm
  - Metodele vor fi implementate în clasele derivate
  - Implementează o metodă template care definește structura unui algoritm
    - Apelează metodele asociate pașilor algoritmului
- **ConcreteClass**
  - Implementează metodele asociate pașilor algoritmului
  - Permite rafinarea anumitor pași ai algoritmului

# State

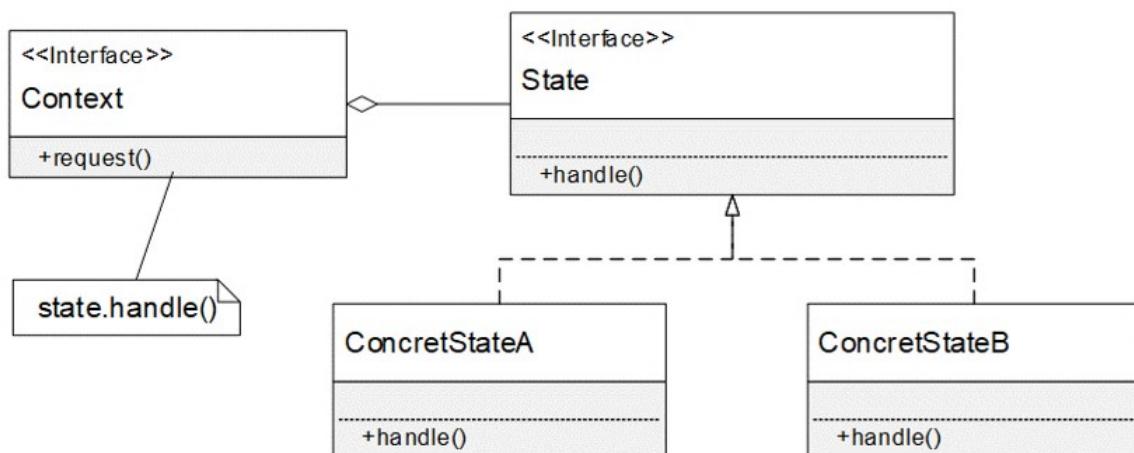
## Problema

- Comportamentul unui obiect depinde de starea acestuia
- Comportamentul se va modifica dinamic, în funcție de stare
- Uzual, implementarea se bazează pe structuri condiționale multiple prin intermediul cărora este controlat fluxul aplicației în funcție de stare
- Exemple:
  - Componente de redare a conținutului multimedia

## Scop

- Permite unui obiect să își modifice comportamentul la schimbarea stării interne
- Starea este gestionată prin intermediul unei ierarhii de clase

## Diagrama de clase



# Componente

- **Context**
  - Definește interfața utilizată de clienți
  - Gestionează starea curentă
- **State**
  - Definește o interfață pentru încapsularea comportamentului asociat unei anumite stări a contextului
- **ConcreteStateA, ConcreteStateB**
  - Implementează comportamentul asociat unei anumite stări a contextului

# Memento

## Problema

- Necesitatea aducerii unui obiect la o stare anterioară
- Posibilitatea salvării stării unui obiect și restaurarea acesteia
- Exemple:
  - Serializarea și deserializarea obiectelor

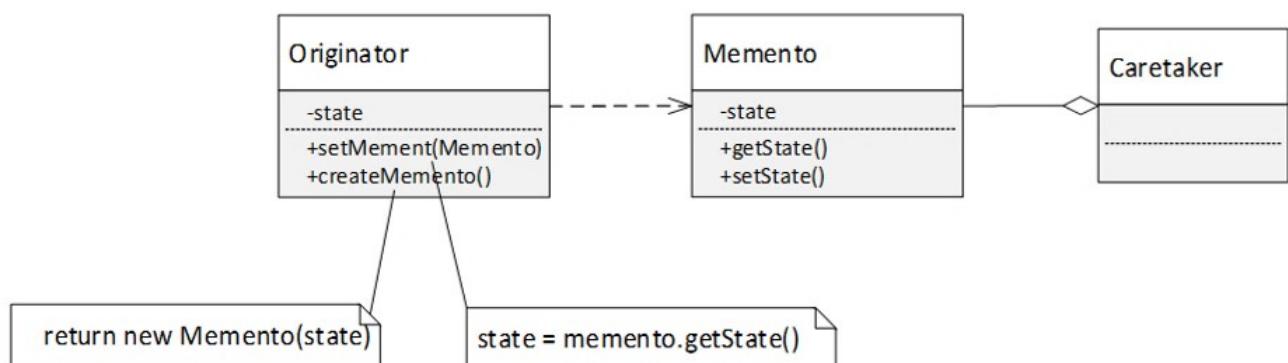
## Scop

- Captează și externalizează starea internă a unui obiect, în vederea restaurării ulterioare
- Nu se încalcă principiul încapsulării
- Furnizează suportul pentru revenirea obiectului la o stare anterioară

# Implementare

- Clientul face o cerere de obținere a unui obiect de tip *Memento*, de la obiectul sursă, în momentul în care dorește salvarea stării obiectului sursă
- Obiectul sursă creează și salvează un obiect de tip *Memento* cu starea sa internă
- Dacă dorește restaurarea unei stări, clientul va transmite obiectului sursă un obiect de tip *Memento*, care conține starea internă salvată la un moment dat

## Diagrama de clase



# Componente

- **Memento**

- Gestionează starea internă a unui obiect de tip *Originator*
- Este gestionat de *Caretaker*

- **Originator**

- Creează un obiect de tip *Memento* pentru salvarea stării interne
- Utilizează *Memento* pentru restaurarea stării sale interne

- **Caretaker**

- Gestionează obiectul de tip *Memento*
- Nu acționează asupra stării acestuia

# Interpreter

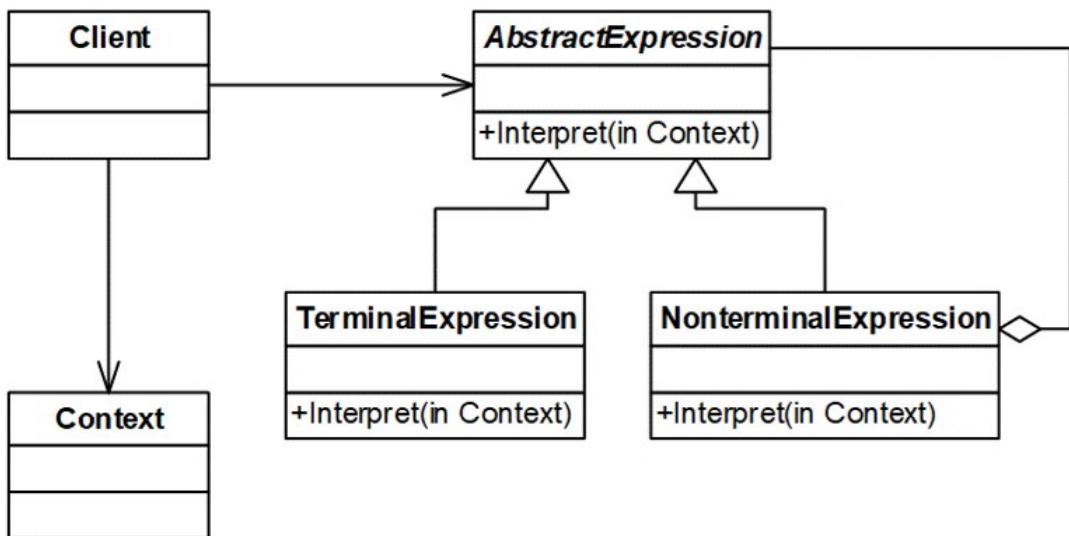
## Problema

- Existența unor probleme repetitive într-un domeniu bine definit și foarte cunoscut
- Domeniul este caracterizat printr-un limbaj
  - Problemele pot fi rezolvate prin intermediul unui interpreter
- Exemple
  - Limbaje simple de scripting pentru configurarea aplicațiilor
  - Traduceri
  - Interpretări muzicale
  - Motoare de reguli

## Scop

- Definirea unei reprezentări pentru gramatica unui limbaj
- Definirea unui mecanism care folosește reprezentarea pentru a interpreta expresiile limbajului
- Asociere
  - Domeniu → limbaj
  - Limbaj → gramatică
  - Gramatică → ierarhie orientată-obiect

## Diagrama de clase



## Componente

- **AbstractExpression**
  - Declară o interfață pentru executarea unei operații
- **TerminalExpression**
  - Implementează o operație asociată simbolurilor terminale din gramatică
- **NonterminalExpression**
  - Implementează o operație asociată simbolurilor non-terminale din gramatică
  - Instanțele sunt definite pentru orice regulă de compunere
- **Context**
  - Include informațiile globale utilizate de interpretor
- **Client**
  - Preia o expresie exprimată în limbajul în care este definită gramatica
  - Generează arborele de sintaxă asociat expresiei
  - Invocă operațiile de interpretare

# Iterator

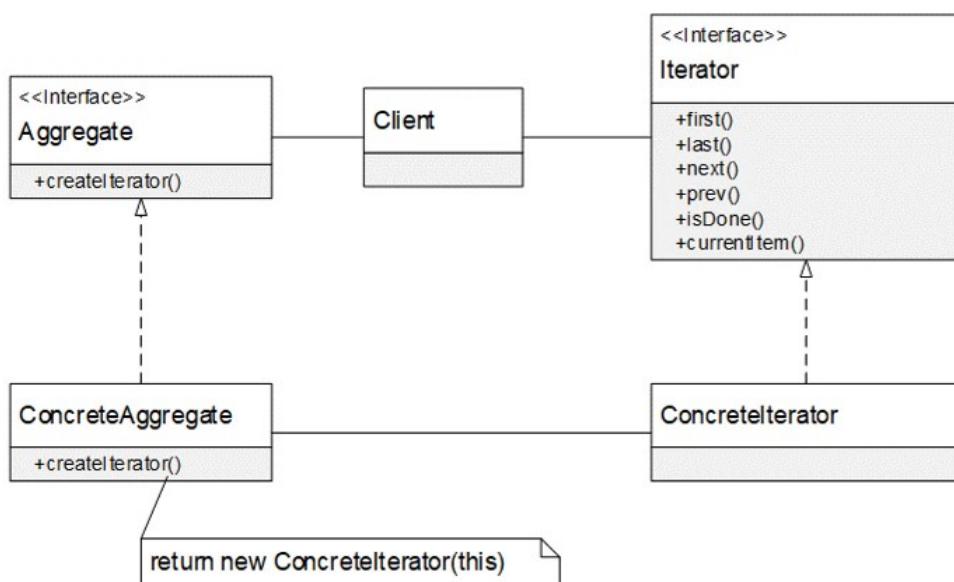
## Problema

- Definirea unui mecanism general pentru parcurgerea diferitelor structuri de date
- Definirea de algoritmi care pot opera asupra structurilor de date în mod transparent
- Exemple
  - Implementarea diferenților algoritmi (sortare, copiere etc.) asupra diferitelor tipuri de structuri de date

# Scop

- Pune la dispoziție un mecanism pentru accesul secvențial la elementele unei colecții
  - Fără expunerea reprezentării interne a acesteia

## Diagrama de clase



# Componente

- **Iterator**

- Definește o interfață pentru accesarea și traversarea elementelor unei colecții

- **ConcretIterator**

- Implementează interfața Iterator
- Gestionează poziția curentă în cadrul parcurgerii

- **Aggregate**

- Definește o interfață asociată colecției care poate fi parcursă prin intermediul unui Iterator
- Definește o interfață pentru crearea obiectelor de tip Iterator

- **ConcreteAggregate**

- Implementează interfața Aggregate

- **Client**

- Utilizează obiectele de tip Iterator pentru accesarea elementelor colecțiilor

## Avantaje

- Asigurarea independenței colecției față de mecanismele de traversare
- Posibilitatea traversării elementelor colecțiilor prin intermediul mai multor iteratori, simultan

## Dezavantaje

- Clasa de tip iterator nu este informată cu privire la modificările apărute în cadrul colecției de elemente

## Mediator

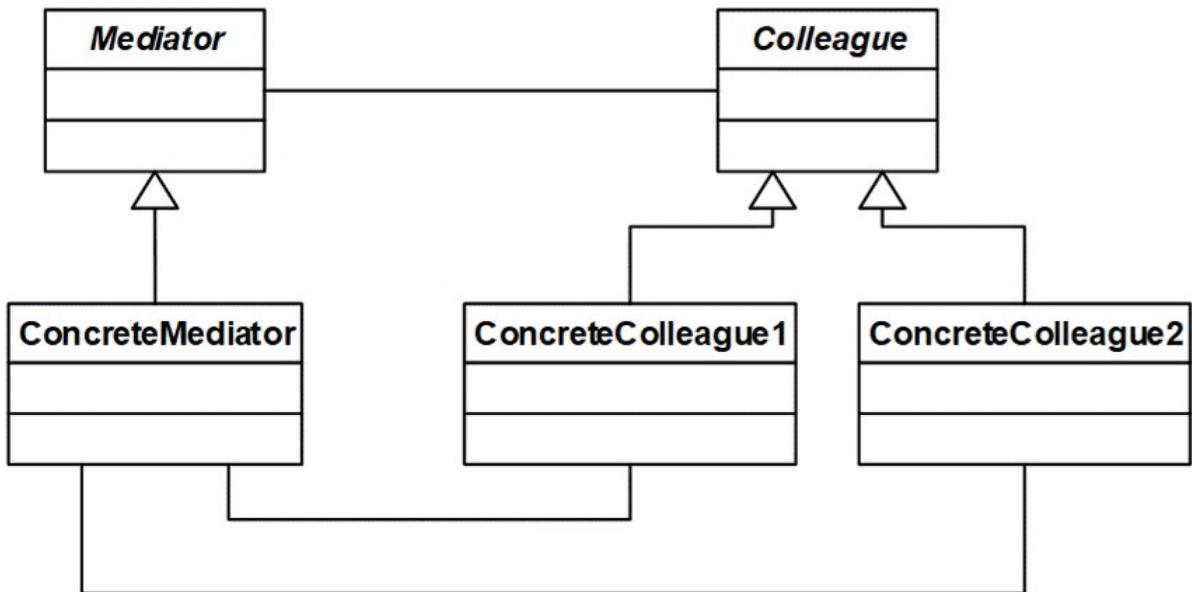
## Problema

- Existența unui număr ridicat de clase care comunică între acestea
- Pentru evitarea legăturilor strânse dintre componente este necesară implementarea unui mecanism care să faciliteze comunicarea dintre aceste componente
- Exemple
  - Comunicarea dintre controalele din cadrul ferestrelor grafice
  - Aplicații de tip chat

## Scop

- Definește un obiect care încapsulează modul în care interacționează o mulțime de obiecte
- Definește un intermediar pentru decuplarea relațiilor multiple

## Diagrama de clase



## Componente

### • **Mediator**

- Definește o interfață pentru comunicarea cu obiecte de tip *Colleague*

### • **ConcreteMediator**

- Implementează modalitatea de cooperarea dintre obiectele de tip *Colleague*
- Gestionează obiectele de tip *Colleague*

### • **Colleague**

- Are acces la o clasă de tip *Mediator*
- Interacționează cu colegii prin intermediul mediatorului

### • **ConcreteColleague1, ConcreteColleague2**

- Implementări concrete pentru *Colleague*

## Avantaje

- Asigură un nivel ridicat de înțelegere a logicii sistemului, prin integrarea acesteia într-o singură clasă
- Decuplarea claselor de tip *Colleague*
- Simplificarea comunicării prin eliminarea relațiilor de tip mulți la mulți și transformarea acestora în relații de tip unu la mulți și mulți la unu

## Dezavantaje

- Tendința componentelor de tip mediator de a deveni foarte complexe

# Visitor

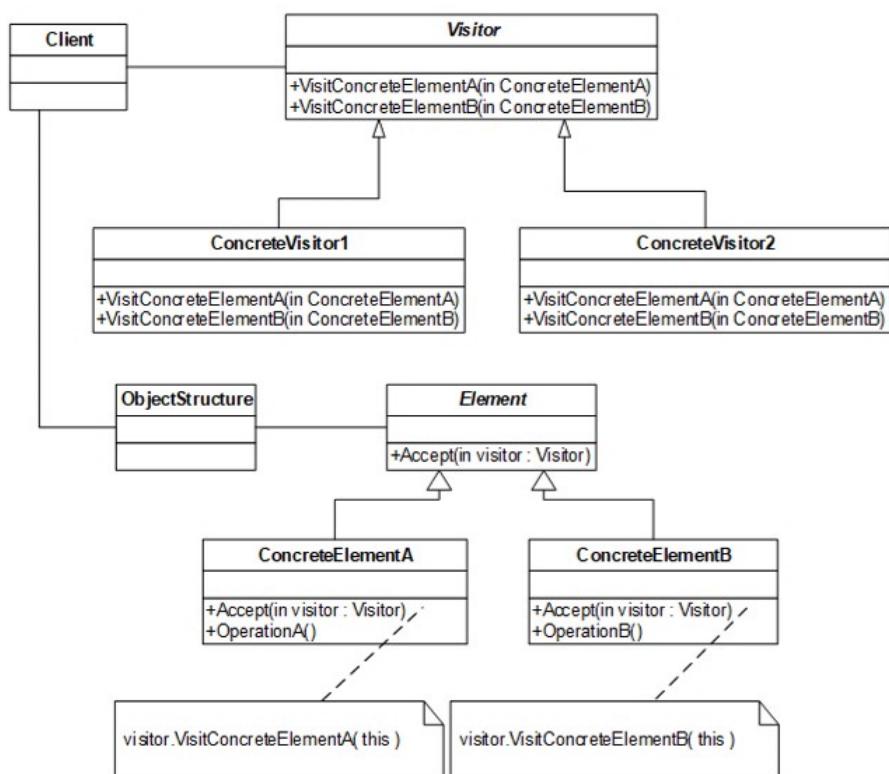
## Problema

- Necesitatea executării unor operații asupra elementelor unei structuri eterogene aggregate
- Evitarea implementării operațiilor în cadrul claselor de tip element
- Necesitatea verificării și conversiei elementelor în funcție de tipul acestora, pentru aplicare operației corecte
- Exemple
  - Definirea de operații noi pentru colecții de clase, fără modificarea ierarhiei

# Scop

- Definește o operație care va fi aplicată pe elementele unei structuri
- Operațiile sunt definite fără modificarea clasele elementelor asupra cărora operează

## Diagrama de clase



# Componente

- **Visitor**
  - Declară o operație de vizitare pentru fiecare element concret din structură
- **ConcreteVisitor1, ConcreteVisitor2**
  - Implementează operațiile definite de *Visitor*
- **ObjectStructure**
  - Permite enumerarea elementelor
  - Definește o interfață care accesul vizitatorilor la elemente
- **Element**
  - Definește o operație de acceptare, asociată unui obiect de tip *Visitor*
- **ConcreteElementA, ConcreteElementB**
  - Implementează operația de acceptare
- **Client**

# Avantaje

- Posibilitatea extinderii operațiilor efectuate asupra elementelor unei colecții
- Complexitatea claselor asociate elementelor colecțiilor este redusă

## Dezavantaje

- Complexitatea
- Pentru ierarhiile de clase existente, acestea trebuie să includă suport pentru operații care acceptă un obiect cu rolul de *Visitor*

## Null Object

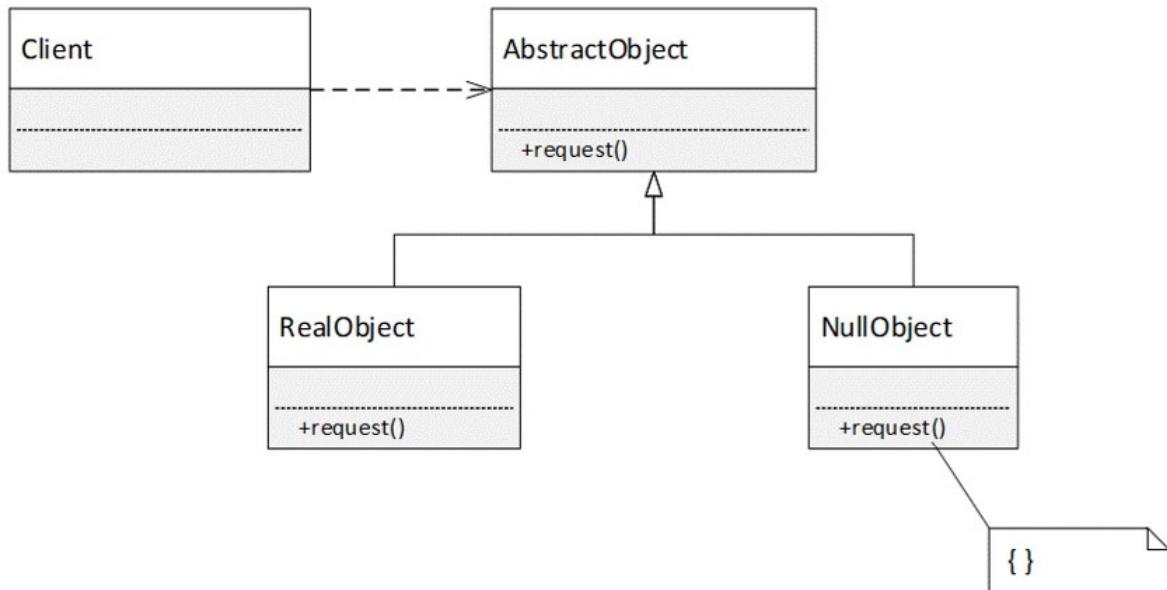
## Problema

- Existența unei ierarhii de clase între care există dependențe
- Situații în care aceste dependențe nu sunt necesare
- Utilizarea unei referințe nule ar conduce la complicarea codului sau la încălcarea principiilor de proiectare orientată obiect

## Scop

- Încapsulează absența unui obiect prin furnizarea unei alternative care să poată fi substituită
- Oferă un comportament adecvat implicit, fără nici o acțiune

## Diagrama de clase



## Componente

- **AbstractObject**
  - Declără interfața pentru client
  - Definește acțiunile care trebuie implementate
- **RealObject**
  - Implementează comportamentul normal, așteptat de client
- **NullObject**
  - Definește obiectele nule, care pot substitui obiectele reale
  - Acțiunea este implementată fără nici o operație
- **Client**

# Specification

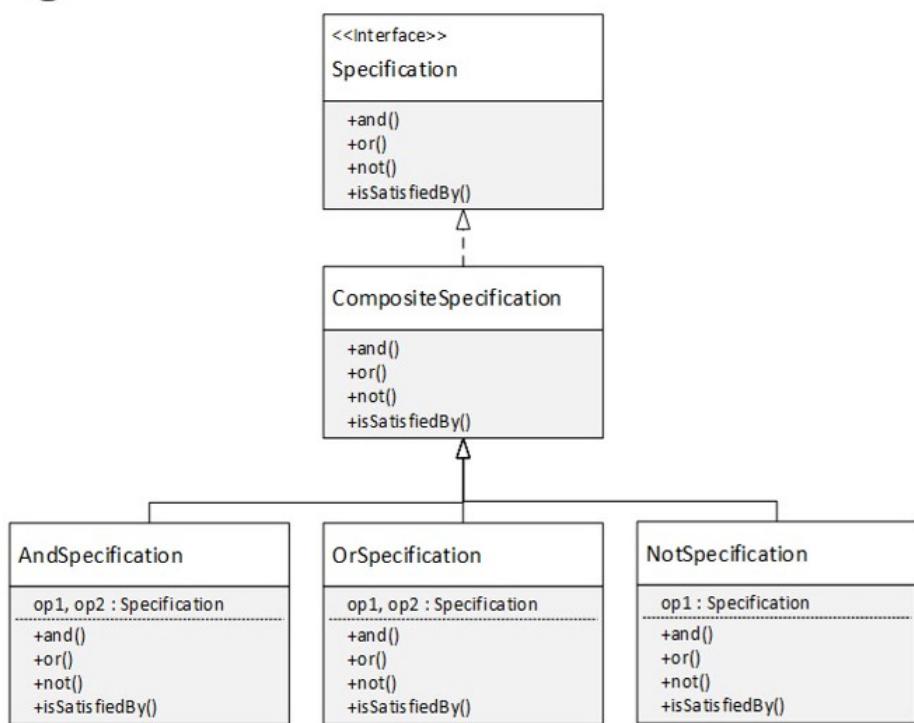
## Problema

- Necesitatea selectării unui subset de obiecte pe baza unor criterii
- Trebuie specificat doar ce poate face un obiect, fără a explica detaliile de implementare
- Exemple
  - Căutarea obiectelor într-o bază de date
  - Validarea obiectelor care îndeplinesc anumite criterii
  - Crearea de instanțe care îndeplinesc criteriile date

# Scop

- Crearea unei specificații capabile să precizeze dacă un obiect candidat îndeplinește anumite criterii

## Diagrama de clase



# Componente

- **Specification**

- Declară metodele pe care le au obiectele de tip specificații
- Metoda principală este *isSpecifiedBy()*
- Metode de compunere *and()*, *or()* și *not()*

- **CompositeSpecification**

- Definește modalitatea de compunere a specificațiilor
- Implementează metodele de compunere *and()*, *or()* și *not()*

- **AndSpecification**

- Definește modalitatea de compunere a specificațiilor prin operatorul *and*

- **OrSpecification**

- Definește modalitatea de compunere a specificațiilor prin operatorul *or*

- **NotSpecification**

- Definește modalitatea de negare a unei specificații prin operatorul *not*

# Avantaje

- Decouplează proiectarea cerințelor și validarea
- Permite crearea de definiții declarative de sistem

# Blackboard

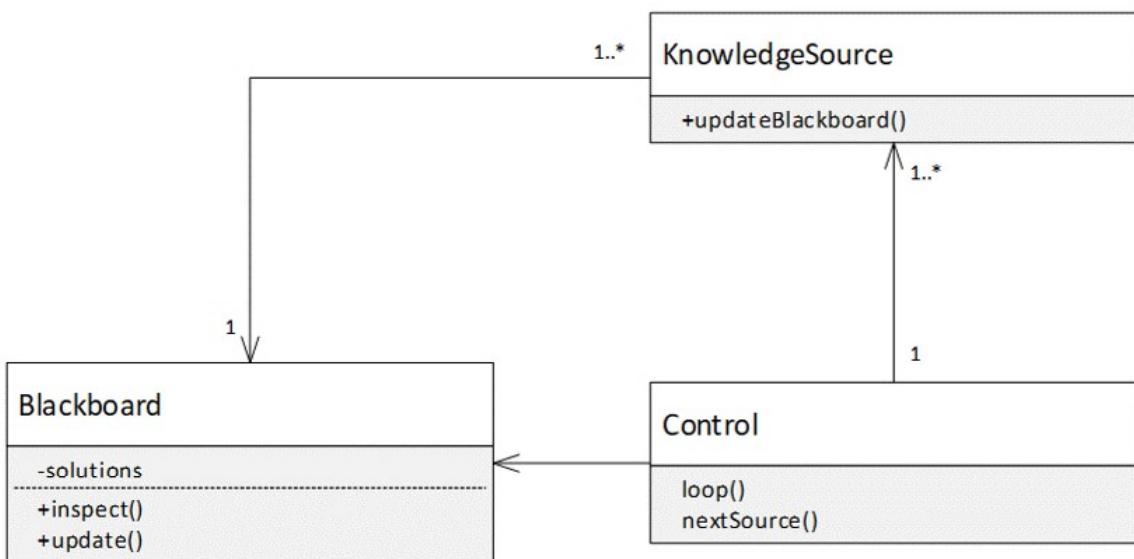
## Problema

- Existența unui domeniu în care nici o abordare la o soluție nu este cunoscută sau fezabilă
- Se identifică o serie de arii de expertiză
- Soluțiile la problemele parțiale necesită reprezentări și paradigme diferite
- Fiecare secvență de transformare poate genera soluții alternative
- Exemple
  - Recunoașterea vocală – transformările necesită expertiză acustică, fonetică și statistică
  - Identificarea autovehiculelor

## Scop

- Mai multe subsisteme specializate combină cunoștințele pentru a construi o soluție eventual parțială sau aproximativă

## Diagrama de clase



# Componente

- **Blackboard**

- Include obiectele din spațiul soluțiilor

- **KnowledgeSource**

- Module specializate cu reprezentări specifice

- **Control**

- Responsabil cu selectarea, configurarea și execuția modulelor

## Referințe

- .NET Design Patterns, <http://www.dofactory.com/net/design-patterns>
- Data & Object Factory, *Gang of Four Software Design Patterns*, Companion document to Design Pattern Framework™ 4.5, 2017
- Data & Object Factory, *Patterns in Action* 4.5, A pattern reference application, Companion document to Design Pattern Framework™ 4.5, 2017
- Design Patterns | Object Oriented Design, <http://www.oodesign.com/>
- Design patterns implemented in Java, <http://java-design-patterns.com/patterns/>
- M. Fowler, D. Rice, M. Foemmel, E. Hieatt, R. Mee, R. Stafford, *Patterns of Enterprise Application Architecture*, Addison Wesley, 2002
- E. Freeman și alii, *Head First Design Patterns*, O'Reilly, 2004
- J.D. Meier et al, Application Patterns, <http://apparch.codeplex.com/wikipage?title=Application%20Patterns>, 2009
- D. Schmidt, M. Stal, H. Rohnert and F. Buschmann, *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*, Volume 2, John Wiley & Sons, 2000
- A. Shivets, *Design Patterns Made Simple*, <http://sourcemaking.com>
- Stack Overflow, <https://stackoverflow.com/>
- D. Trowbridge et. al, *Enterprise Solution Patterns Using Microsoft .NET*, Version 2.0, Microsoft, 2003