

Modele de proiectare a aplicațiilor de întreprindere

Cursul 1 - 2

Prezentare disciplină

- Tematica disciplinei
- Modalitatea de desfășurare
- Modalitatea de evaluare

Tematica disciplinei

- Principii de design OO, modele de design, anti-modele
- Modele de proiectare (creaționale, structurale și comportamentale)
- Modele pentru aplicații concurente
- Modele de design în programarea aplicațiilor Web
- Modele de design în programarea aplicațiilor de întreprindere

Modalitatea de desfășurare

- Cursuri și laboratoare
- Laboratoare
 - Java (Eclipse, IntelliJIdea, Netbeans)
 - C# (Visual Studio)
 - Android - Java (Android Studio)
- Platforma
 - Google Classroom
 - <http://online.ase.ro>
- Fișa disciplinei
 - <http://fisadisciplina.ase.ro/>

Modalitatea de evaluare

- Seminar (50%)
 - Proiect în echipă
 - Prezentare la ultima întîlnire
 - Evaluare individuală
 - Joi, 14 ianuarie 2021, orele 17:30/19:30 – 21:20
- Curs (50%)
 - Probă practică, la calculator
 - Joi, 21 ianuarie 2021 sau vineri, 22 ianuarie 2020, ora 19:30

Sumar

- Aplicații de întreprindere
- Principii ale POO
- Modele de design
- Anti-modele de design
- Modele de proiectare (GoF)

Aplicații de Întreprindere

Aplicații de Întreprindere

- Destinate organizațiilor cu scopul de a asista componenta de business în rezolvarea diferitelor probleme specifice
- Caracteristici
 - Complexe
 - Scalabile
 - Distribuite
 - Bazate pe componente
- Deployment (Instalare)
 - Rețelele companiilor
 - Intranet
 - Internet

Provocări

- Cerințele aplicațiilor se modifică în timp
- Apar provocări și noi oportunități de afaceri
- Cerințele apărute în timpul dezvoltării pot conduce la modificarea sferei de cuprindere și cerințelor inițiale ale aplicației
- Aplicațiile sunt complexe și se dezvoltă în echipă

Cerințe

- Proiectarea aplicațiilor astfel încât acestea să poată modificate sau extinse cu ușurință în timp
- Proiectarea de componente individuale, independente, care pot fi dezvoltate și testate izolat
- Partiționarea aplicației în componente discrete și slab cuplate între ele, care pot fi integrate cu ușurință

Aplicații de întreprindere: cerințe de proiectare

Modelul de dezvoltare	• Echipa, procese, management, testare, livrabile.
Modelul de afaceri	• Obiective, resurse, timp, reguli
Modelul utilizatorilor	• UI, instruire, configurare
Modelul logic	• Structura aplicației, modelare obiectelor/datelor, definirea interfețelor
Modelul tehnologic	• Instrumente de dezvoltare, SGBD, platforme de instalare
Modelul fizic	• Arhitectura fizică a aplicației, distribuirea componentelor

Principii în dezvoltarea software

Restricții

- Programarea se face întotdeauna în contextul unor restricții
- Mantenabilitatea: întotdeauna o restricție importantă
- Eleganța codului: adaptarea ideală la restricții

Caracteristicile codului scris bine

- Lizibilitate
- Testabilitate
- Structurarea
- Dimensiune redusă, ușor de gestionat
- Funcționalitate simplă și specifică

Stiluri de scriere a codului

- <https://google.github.io/styleguide/javaguide.html>
 - <https://google.github.io/styleguide/cppguide.html>
 - <https://source.android.com/source/code-style.html>
-
- Suport
 - <https://pmd.github.io/>

Principii de dezvoltare software

- **DRY** – Don't repeat yourself
- **KISS** – Keep It Simple, Stupid
- **YAGNI** – You Aren't Gonna Need It

DRY – Don't repeat yourself

- "*Fiecare element de cunoaștere trebuie să aibă o reprezentare unică, lipsită de ambiguitate și autoritate în cadrul unui sistem*"
- Reducerea repetării informațiilor de orice natură
- Duplicarea în logică este eliminată prin abstractizare
- Duplicarea în proces este eliminată prin automatizare
- Împărțirea proiectului în componente care pot fi gestionate

KISS – Keep It Simple, Stupid

- Simplicitatea trebuie să fie un obiectiv în proiectarea entităților software
- Implementarea, cât mai simplă cu putință

YAGNI – You Aren't Gonna Need It

- Funcționalitățile trebuie adăugate doar atunci cînd este nevoie de ele
- Reducerea complexității prin reducerea numărului de module
- Simplicitate redusă pînă la neimplementarea de module
- 80% din timpul alocat unui proiect software este dedicat pentru 20% din funcționalități

Mecanisme de design orientat obiect

- Moștenire
- Compoziție
- Polimorfism
- Delegare
- Tipuri generice

Principii fundamentale de design (POO)

- Separă ceea ce variază
- Programează la interfață, nu la implementare
- Preferă compoziția moștenirii
- Scrie cod adaptabil la schimbare

Principii de design orientat obiect: SOLID

- **SRP** – Single Responsibility Principle
- **OCP** – Open Closed Principle
- **LSP** – Liskov Substitution Principle
- **ISP** – Interface Segregation Principle
- **DIP** – Dependency Inversion Principle

SRP – Single Responsibility Principle

- O clasă trebuie implementată pentru un singur scop (responsabilitate)
- Nu trebuie să existe mai mult de un motiv pentru a modifica o clasă
- Dacă există mai multe responsabilități, o clasă trebuie împărțită în mai multe clase
- Dacă ar fi mai multe responsabilități, modificarea uneia poate afecta cealaltă responsabilitate

Avantaje SRP

- Testabilitatea
- Cuplarea scăzută
- Reducerea complexității claselor

OCP – Open/Closed Principle

- Orice clasă trebuie să fie
 - deschisă pentru extindere și
 - închisă pentru modificare
- Funcționalitățile noi sunt adăugate cu schimbări minimale ale codului existent

LSP – Liskov Substitution Principle

- Orice clasă derivată trebuie să poată substitui clasa de bază
- Clasele derivate nu trebuie să modifice fundamental funcționalitățile din clasele de bază
 - Pot apărea efecte nedorite în modulele existente
- Dacă o clasă de bază este înlocuită cu o clasă derivată, funcționalitatea programului nu trebuie să fie afectată

ISP – Interface Segregation Principle

- Clasele nu trebuie să implementeze interfețe pe care nu le utilizează
- Interfețele trebuie proiectate astfel încât să nu includă metode care vor fi utilizate doar într-un anumit context, pe lângă metodele comune
- Este de preferat să fie definite mai multe interfețe

DIP – Dependency Inversion Principle

- Clasele de nivel înalt nu trebuie să depindă de clasele de nivel scăzut
 - Ambele trebuie să depindă de abstractizări
- Abstractizările nu trebuie să depindă de detalii
 - Detaliile vor depinde de abstractizări
- Proiectarea claselor
 - Nivel înalt
 - Nivel abstractizare
 - Nivel scăzut
- Soluția uzuală: *dependency injection*

Perspective asupra POO

- Design complex vs. YAGNI
- Worse is better vs. the right thing
 - simplitatea implementării vs. simplitatea interfeței
- Mantenabilitatea este cea mai importantă restricție asupra codului
- Codul menținabil este puternic decuplat

Pași fundamentali de design pentru POO (GoF)¹

- Identificarea abstracțiilor corecte
 - Care sunt obiectele prin care se poate reprezenta universul de discuție al problemei?
- Ce granularitate trebuie să aibă obiectele selectate?
- Ce interfețe trebuie să aibă obiectele?
- Ce implementare trebuie să aibă obiectele?

¹GoF: Gang of Four (Gamma, Helm, Johnson, Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, 1995)

Modele de proiectare

Contextul



Modele de proiectare

- Soluții reutilizabile pentru un anumit set de probleme
- Echipe separate de programatori vor ajunge la aceeași soluție sau soluții similare pe o perioadă suficient de mare de timp
- Soluția este elegantă în raport cu restricțiile definite
- Modelele nu sunt atât de complexe încât să devină framework-uri
- Modelele nu atât de simple încât să devină elemente de sintaxă a unui limbaj

Niveluri de abstractizare pentru modele de proiectare

• Arhitectură

- Furnizează un set de subsisteme predefinite
- Specifică responsabilitățile acestora
- Include reguli și linii directoare pentru organizarea relațiilor dintre ele
- Exemplu: organizarea aplicațiilor pe niveluri

• Proiectare

- Oferă o schemă de rafinare a subsistemelor sau componentelor unui sistem software sau relațiile dintre acestea
- Descrie o structură frecvent recurrentă a componentelor care comunică, care rezolvă o problemă de proiectare generală într-un anumit context
- Exemplu: MVC (Model View Controller)

• Implementare

- Model de nivel scăzut specific unei anumite platforme
- Descrie modul de implementare a anumitor aspecte ale componentelor sau a relațiilor dintre ele, utilizând caracteristicile unei platforme date
- Exemplu: implementări specifice Java, .NET etc.

Perspective ale modelelor de proiectare

- Baze de date
 - Nivelul de asigurare a persistenței
- Aplicație
 - Aspectele executabile ale soluției
- Deployment (Instalare)
 - Asocierea componentelor aplicației la infrastructură
- Infrastructură
 - Include toate componentele hardware și de rețea necesare rulării soluției

Gruparea modelelor de proiectare pentru aplicații de întreprindere



Model de design vs. Framework

- Mecanisme pentru reutilizarea designului
- Modele de design
 - Nu sunt concrete
 - Independente de limbajul de programare
 - Elementele arhitecturale de bază pentru framework-uri
- Framework
 - Compilabile
 - Uzual conțin clase abstracte și interfețe
 - Pun la dispoziție infrastructura care permite extinderea și dezvoltarea soluțiilor proprii

Limitări ale modelelor de proiectare

- Pot îmbunătăți sau nu înțelegerea unui proiect sau a unei implementări.
- Pot reduce inteligibilitatea prin adăugarea de indirectări sau creșterea numărului de componente (clase și interfețe)
- Pot conduce la creșterea complexității
- Tendința de a forța utilizarea modelelor de proiectare
- Alegerea necorespunzătoare a unui model de proiectare

Managementul complexității codului sursă

- Utilizarea modelelor de proiectare poate conduce la reducerea complexității ciclomatice a codului sursă

$$CC = V - N + 2 \times C$$

- V – numărul de vîrfuri
- N – numărul de noduri
- C – numărul de componente conexe

- Complexitatea este redusă prin furnizarea de abstractizări gata de utilizare

API (Application Programming Interface)

- O modalitate de comunicare pusă la dispoziție de diferite componente software:
 - Biblioteci software
 - Framework-uri
 - Sisteme de operare
 - Sisteme la distanță
 - Web
- Acces: Privat, partener și public
- Ascunderea informației

Modele de design vs mecanisme de limbaj

- Modelele de design extind capabilitățile anumitor limbaje de programare
- Modelele de design folosite frecvent într-un limbaj pot fi invizibile sau triviale într-un alt limbaj
- Exemplu: 16 din 23 modele de proiectare sunt invizibile sau mai ușor de utilizat în Lisp

Modele de design vs mecanisme de limbaj

- Anumite modele de proiectare au fost înlocuite cu facilități ale limbajelor de programare
 - Funcții generice
 - Expresii lambda
 - Funcții anonime
 - Module
 - Dicționare, tabele de dispersie
 - Delegați
- Modelele de design nu pot fi incluse în totalitate într-un limbaj de programare

Pași de aplicare a unui model de design

- Identificarea unei situații care reprezintă o problemă standard
- Identificarea modelelor aplicabile
- Identificarea interacțiunilor dintre modele
- Aplicarea modelelor la situația dată

Anti-modele

Anti-modele

- **Nevalidarea intrărilor**

- Nu se specifică și nu sînt puse în aplicare gestionarea eventualelor intrări invalide

- **Race Hazard**

- Neobservarea consecințelor ordinii diferite a evenimentelor

- **Dependențe circulare**

- Introducerea dependențelor reciproce directe sau indirecte între obiecte sau module software

- **Obiectul central**

- Un obiect care are prea multe informații sau prea multă responsabilitate.
 - Includerea multor funcții într-o singură clasă.
 - Adesea codul pentru un model și un view sunt combinate în aceeași clasă

- **Interfețe utilizator supraîncărcate**

- Efectuarea unei interfețe atât de puternică și complicată încât este greu de reutilizat sau implementat

Anti-modele

- **Numerele magice:**

- Includerea de valori constante direct în cod, fără nici o explicație a semnificației acestora

- **Șiruri magice:**

- Includerea șirurilor literale în cod, pentru comparații, tipuri de evenimente etc.

- **Programarea copy/paste**

- Copierea și modificarea codul existent fără a crea mai multe soluții generice
 - Se folosesc aceleași secvențe de cod în mai multe locuri, cu mici modificări
 - Nu se respectă principiul DRY

- **Reinventarea roții**

- Nu sunt folosite soluții existente și adecvate și, în schimb, să alege o soluție personalizată, care se comportă mult mai rău decât cea existentă
 - Se face totul prin noi însine și se scrie totul de la zero

Anti-modele

- **Optimizare prematură**
 - Opus YAGNI
 - Efecte: reducerea lizibilității codului, depanarea și întreținerea devin mai greu de realizat și adăugarea unor părți inutile la codul scris
- **Prea multe dependențe**
 - Se utilizează prea multe biblioteci terțe care se bazează pe versiuni specifice ale altor biblioteci
 - Apar incompatibilități
- **Cod Spaghetti**
 - Cod greu de depanat sau modificat din cauza lipsei unei arhitecturi adecvate
- **Programarea prin permutări**
 - Încercarea de a găsi o soluție pentru o problemă prin experimentarea succesivă cu mici modificări, testarea și evaluarea acestora una câte una și, în final, implementarea celei care a funcționat la început
 - Nu se știe dacă soluția va funcționa în toate scenariile sau nu

Anti-modele

- **Lava Flow**
 - Codul care are componente redundante sau de calitate slabă, care par a fi parte integrantă a programului, dar nu fără a înțelege cu desăvârșire ce anume face sau cum influențează întreaga aplicație
 - Uzual, codul este preluat (scris de altcineva) sau proiectul se derulează prea rapid
- **Hard coding**
 - Încadrarea ipotezelor despre mediul unui sistem în implementarea acestuia
- **Soft Coding**
 - Anumite componente care ar trebui să fie în codul sursă sunt plasate în surse externe
- **Cargo cult programming**
 - Scrierea codului fără înțelegerea acestuia
 - Teama de modificare: poate nu mai rulează codul corect

Modele de proiectare (GoF)

Modele de proiectare

- Tipul
- Sfera de cuprindere
 - Nivel de clasă
 - Nivel de obiect
- Definire
 - Denumirea
 - Problema
 - Context, exemple
 - Scopul
 - Soluția
 - Structura
 - Implementarea
 - Consecințe
 - Variante, utilizări frecvente

Modele de proiectare (GoF)

Creaționale	Structurale	Comportamentale
<ul style="list-style-type: none">• Factory Method• Abstract Factory• Builder• Prototype• Singleton	<ul style="list-style-type: none">• Adapter (clasă)/Adapter (obiect)• Bridge• Composite• Decorator• Façade• Flyweight• Proxy	<ul style="list-style-type: none">• Interpreter• Template• Chain of Responsibility• Command• Iterator• Mediator• Memento• Observer• State• Strategy• Visitor

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
- Enterprise-application-e-commerce software development , <https://www.heminfo.com/enterprise-application.html>

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 pînă 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

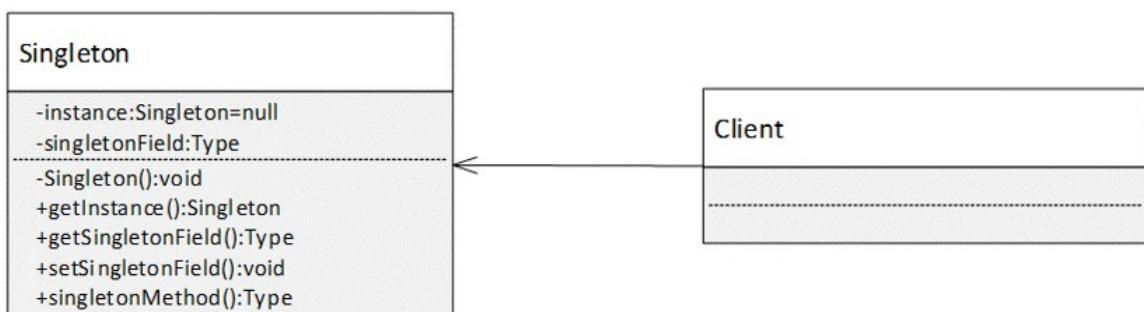
Scop

- 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

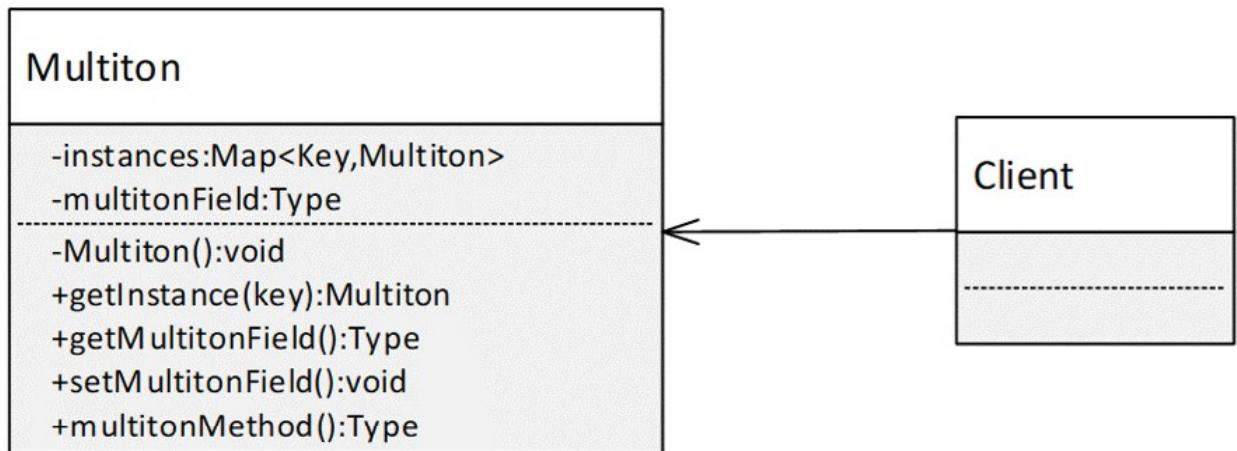
Scop

- 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

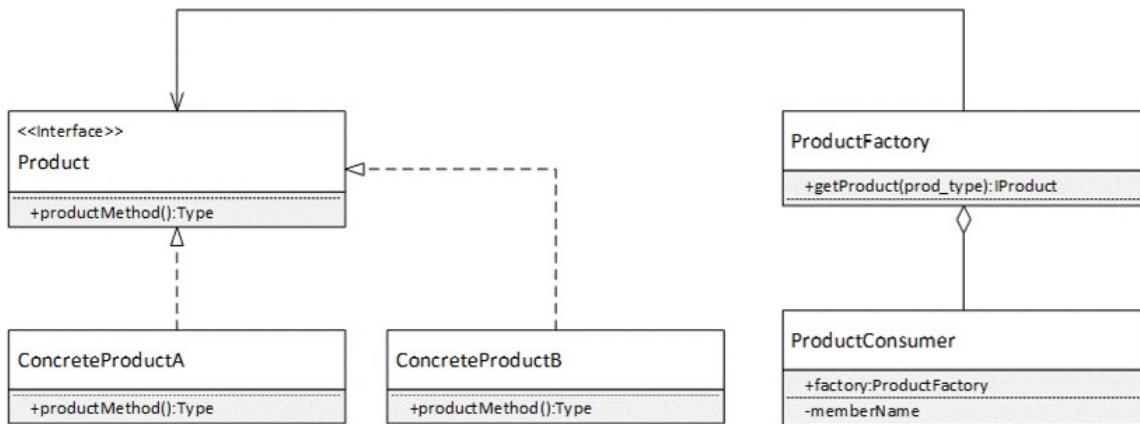
Restricții de implementare

- 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

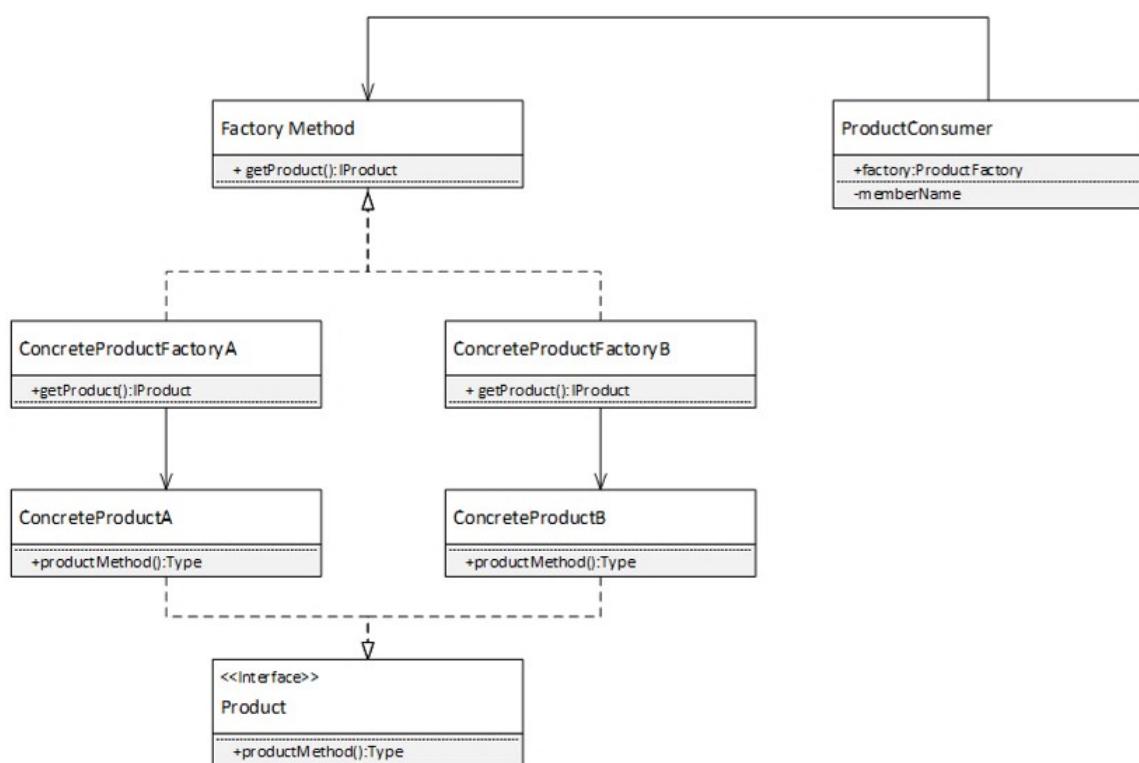
Scop

- 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

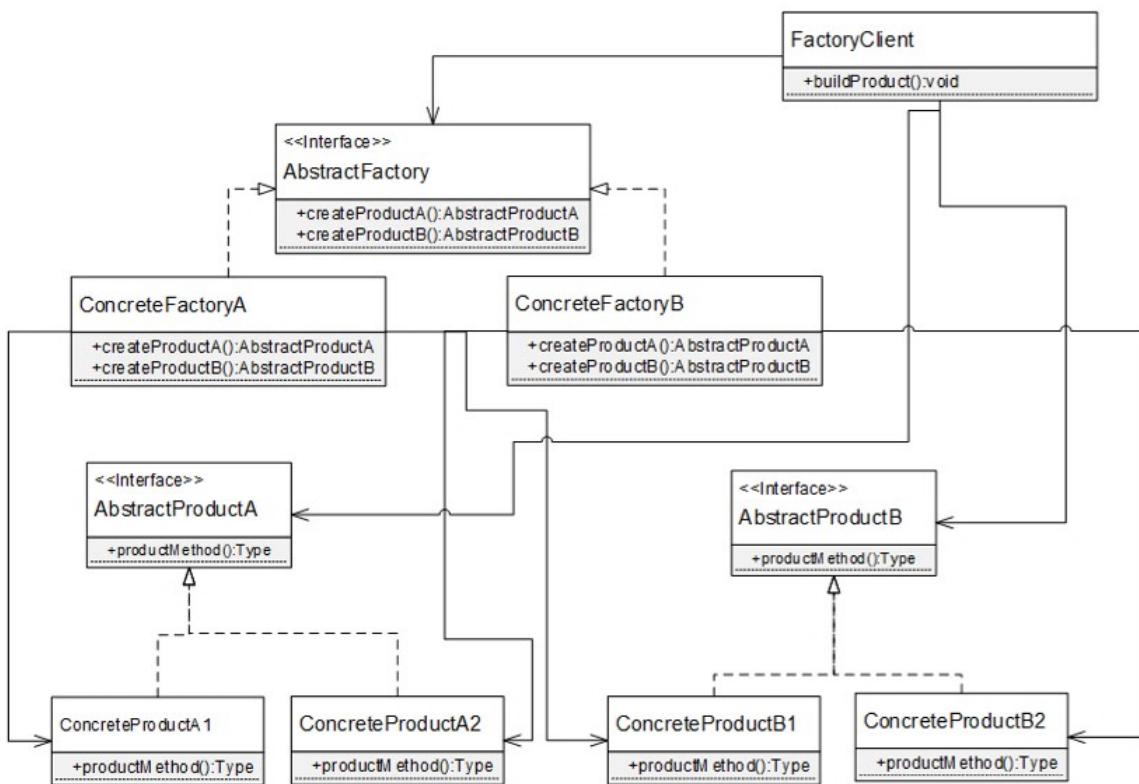
Scop

- 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

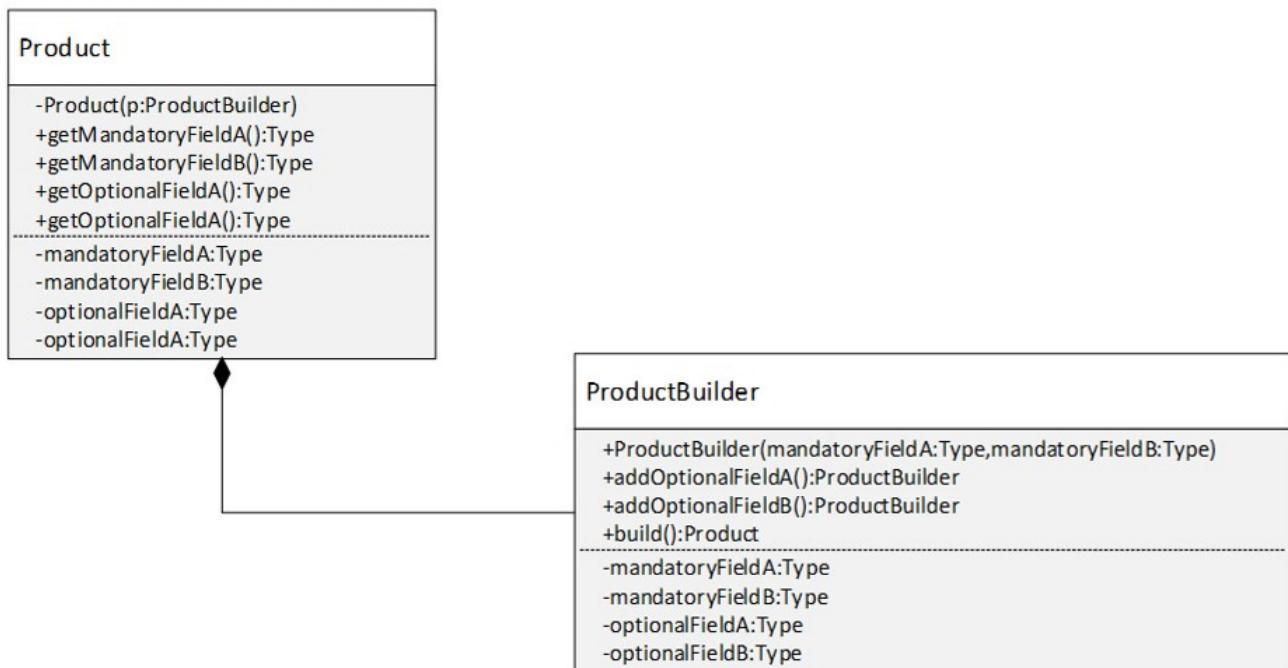
Scop

- 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)

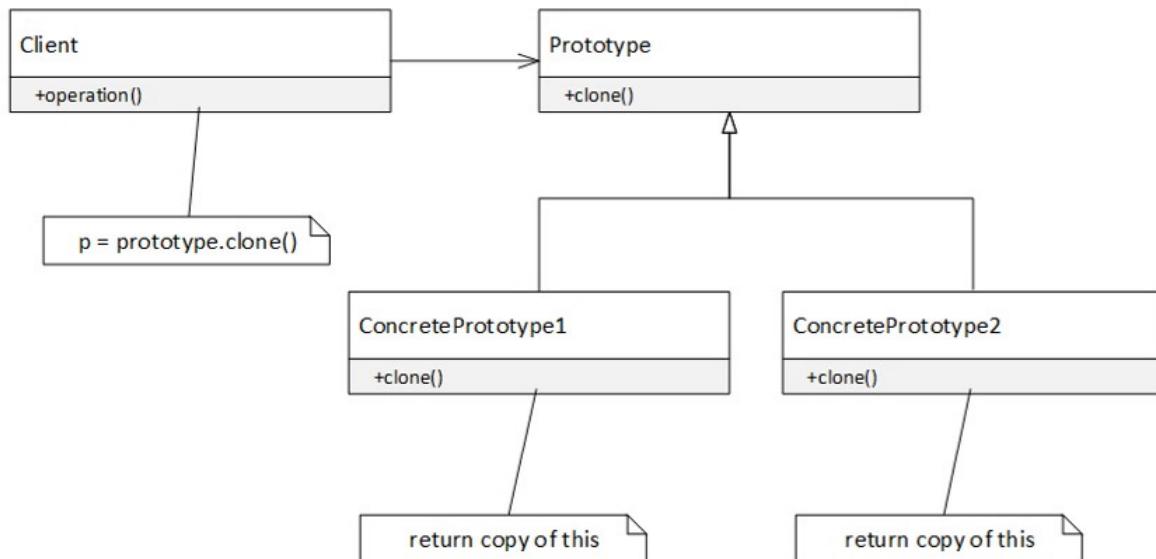
Scop

- 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

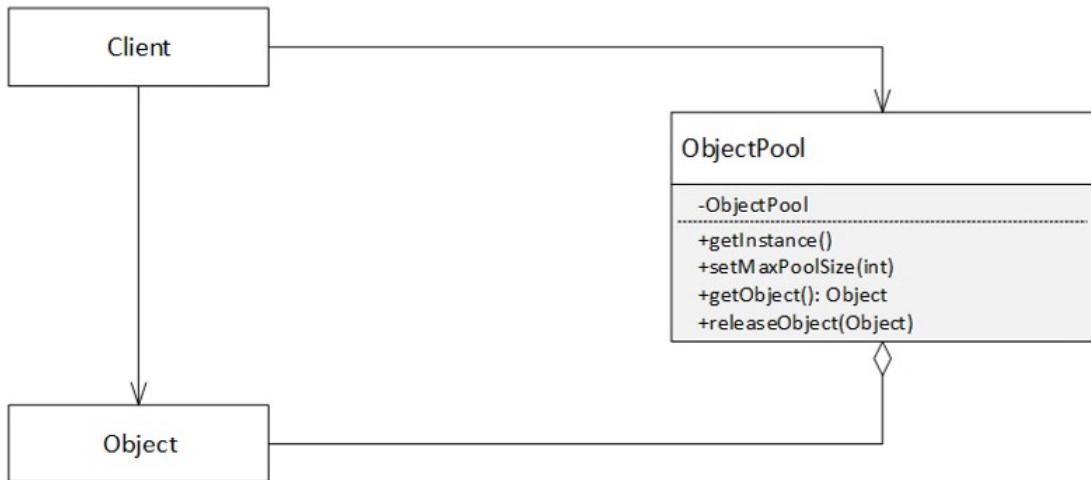
Scop

- 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

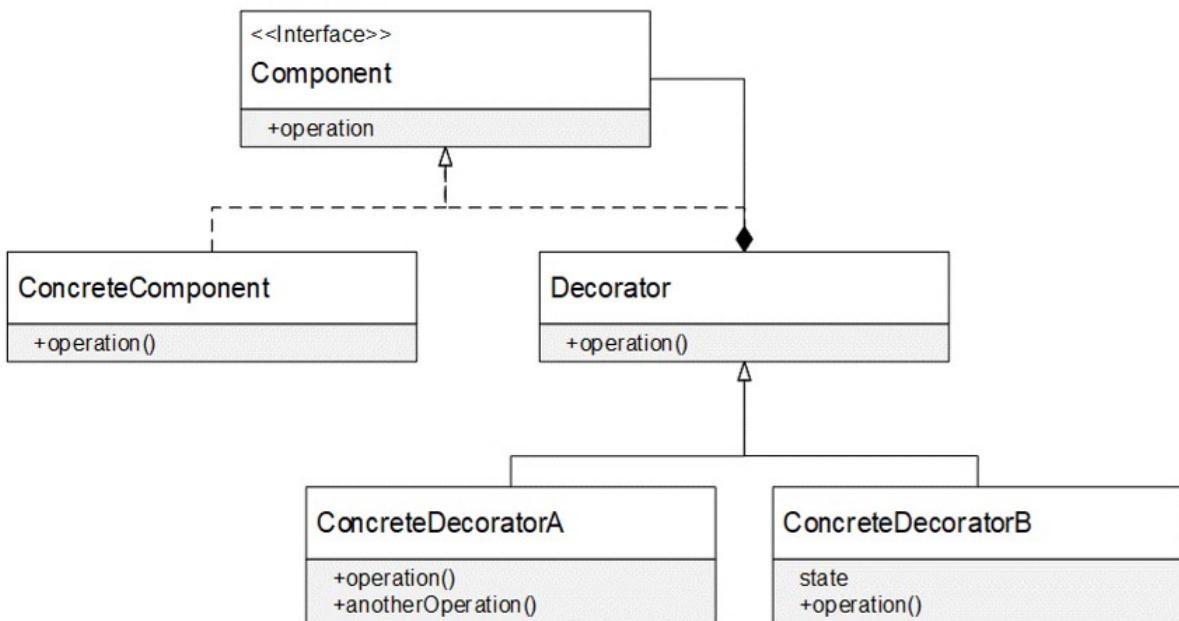
Scop

- 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

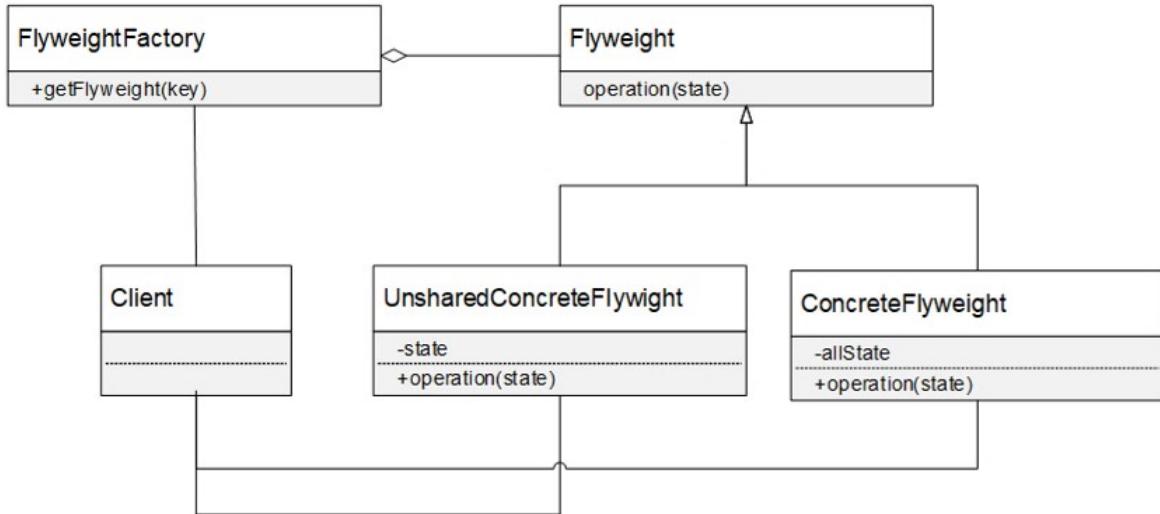
Scop

- 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

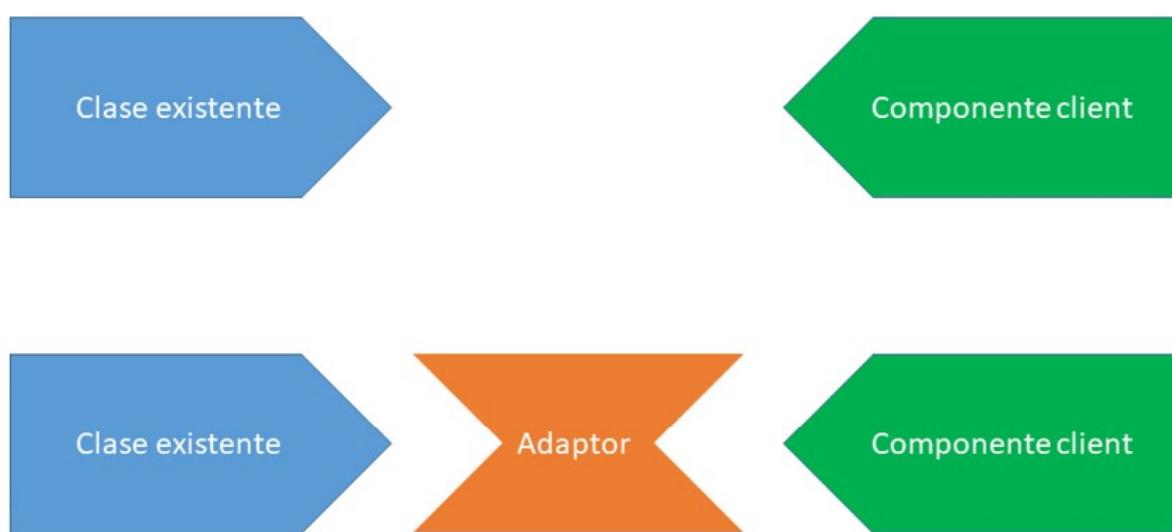
Problema

- 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)

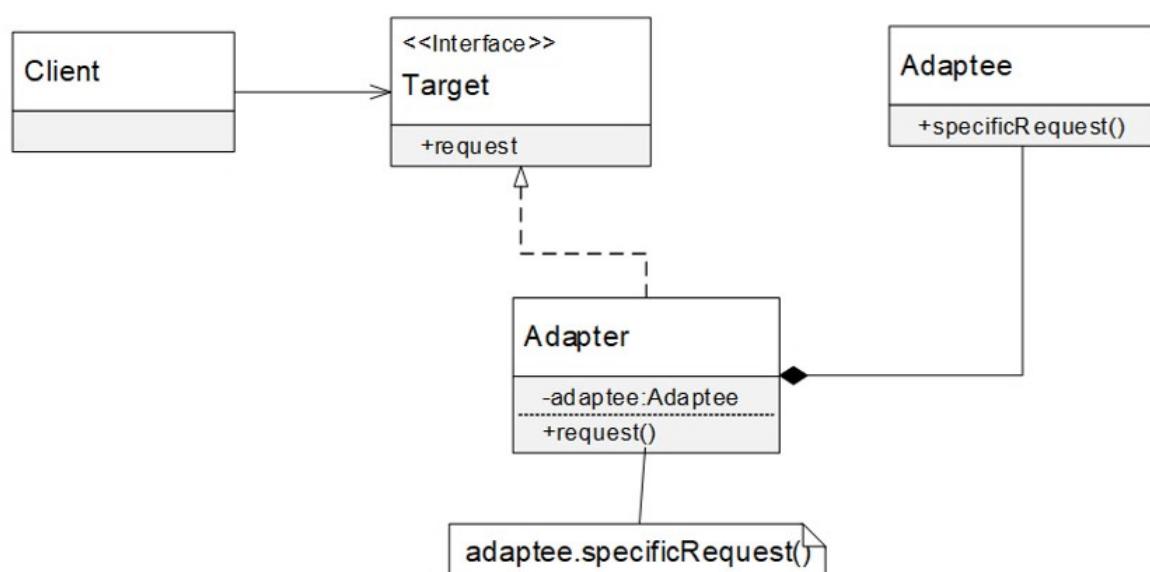
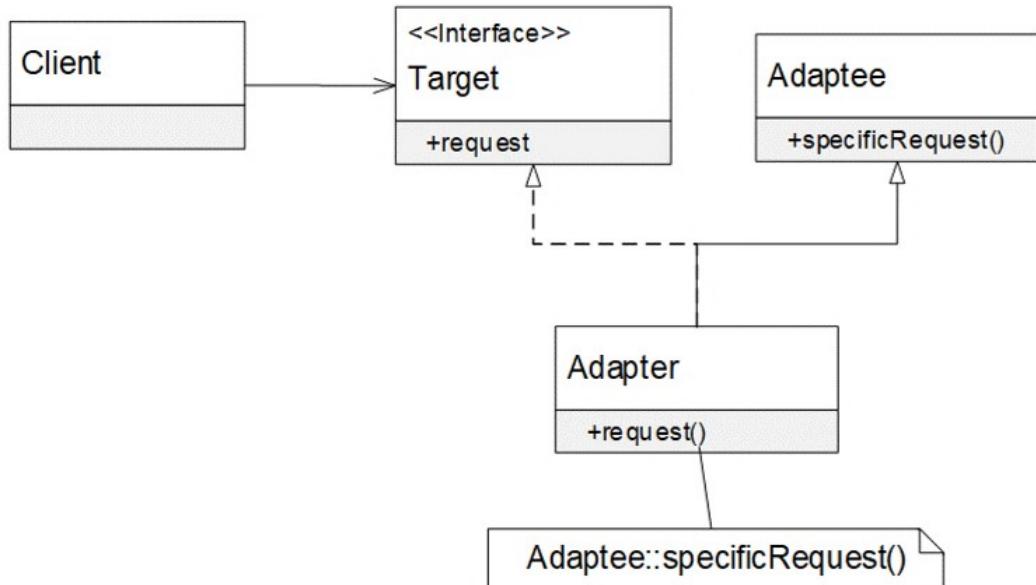


Diagrama de clase (2)



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

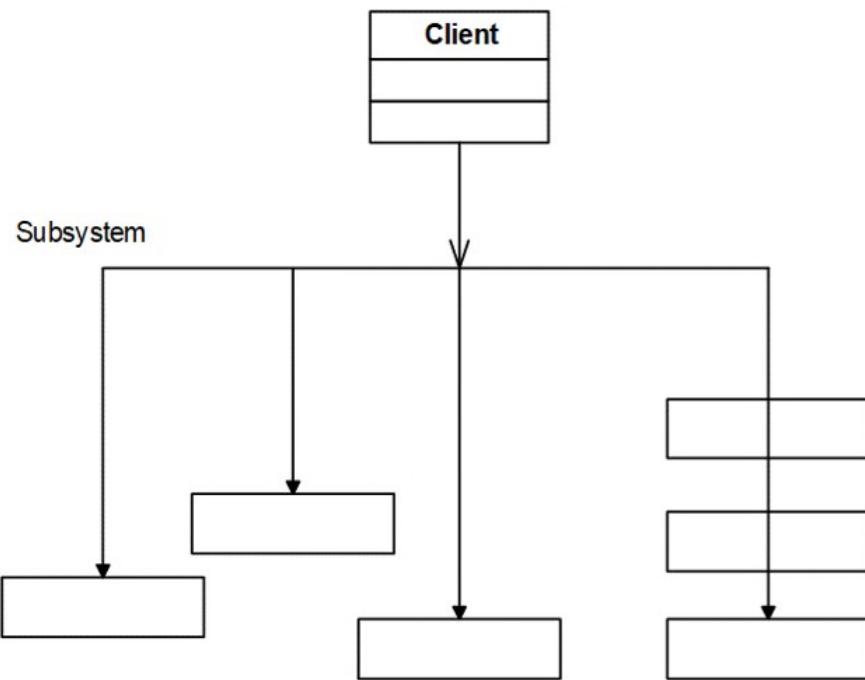
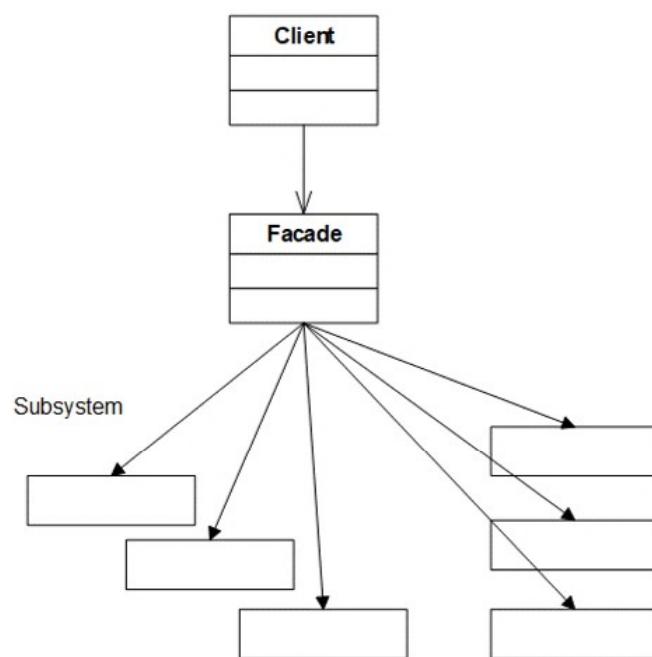


Diagrama de clase



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

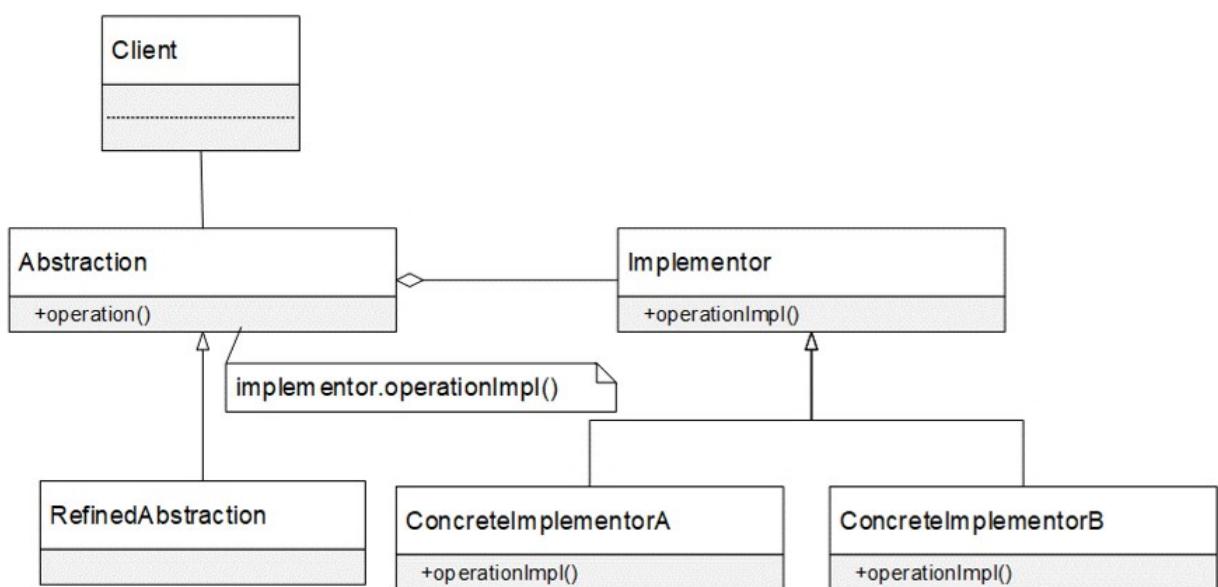
Scop

- 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

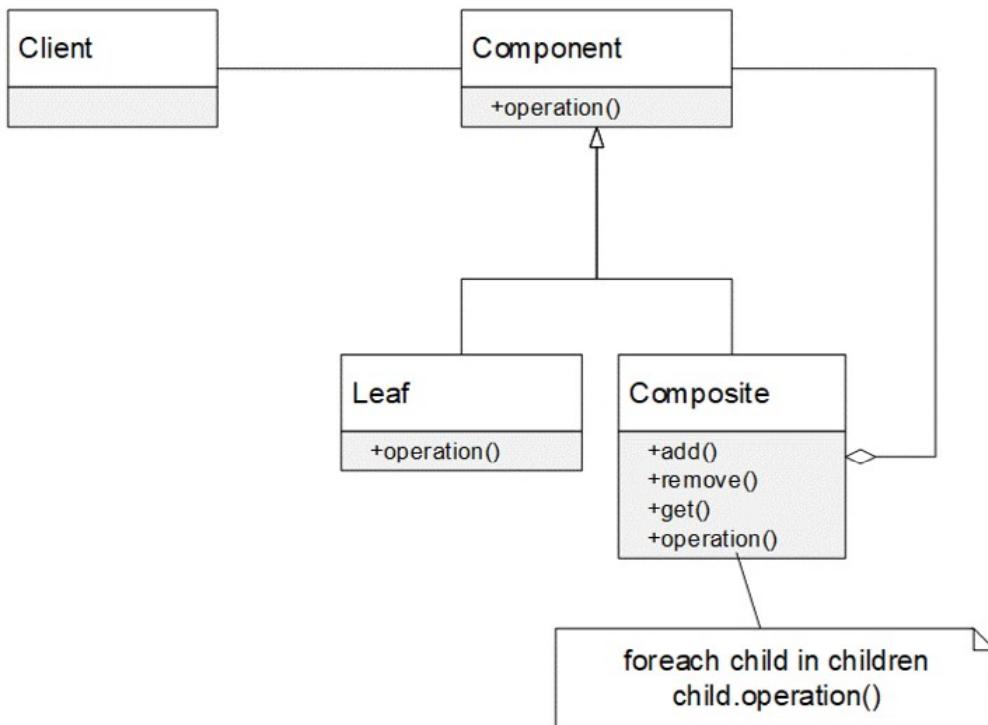
Problema

- 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

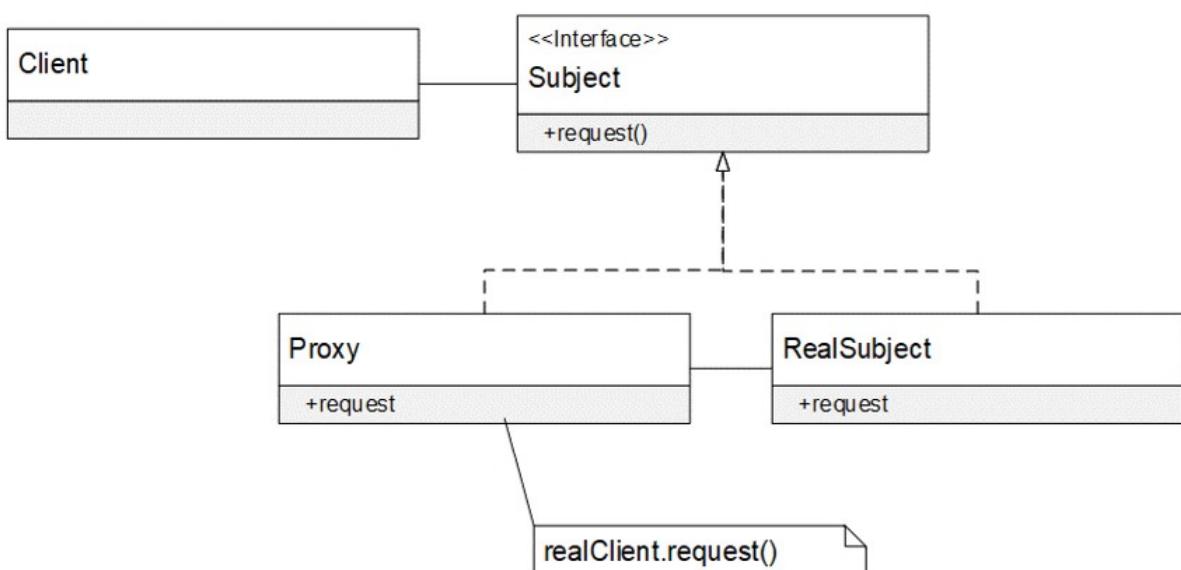
- **Smart References**

- 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

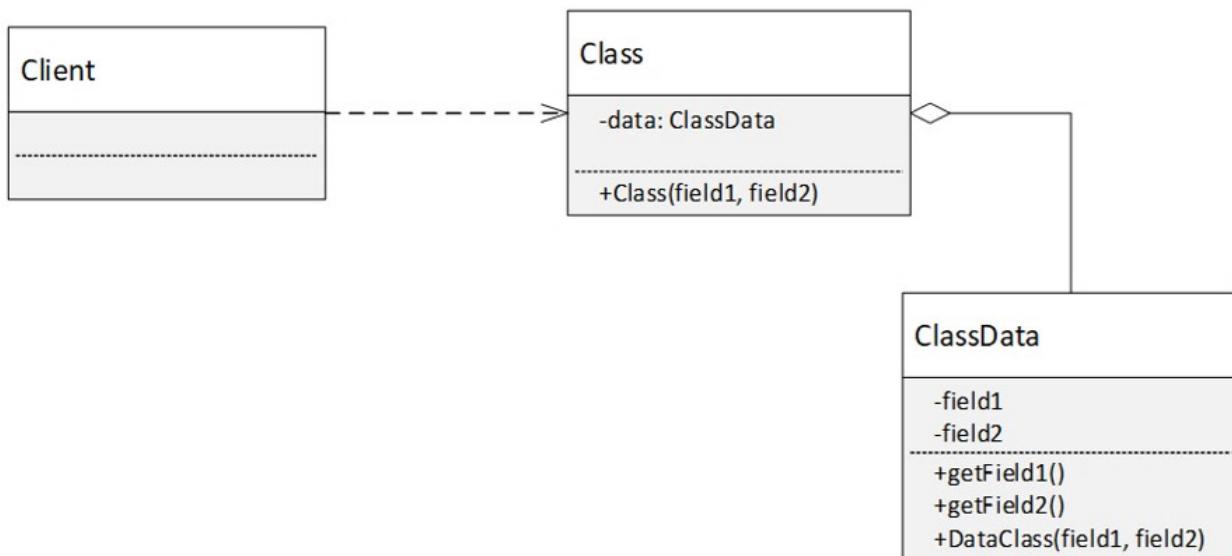
Scop

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

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

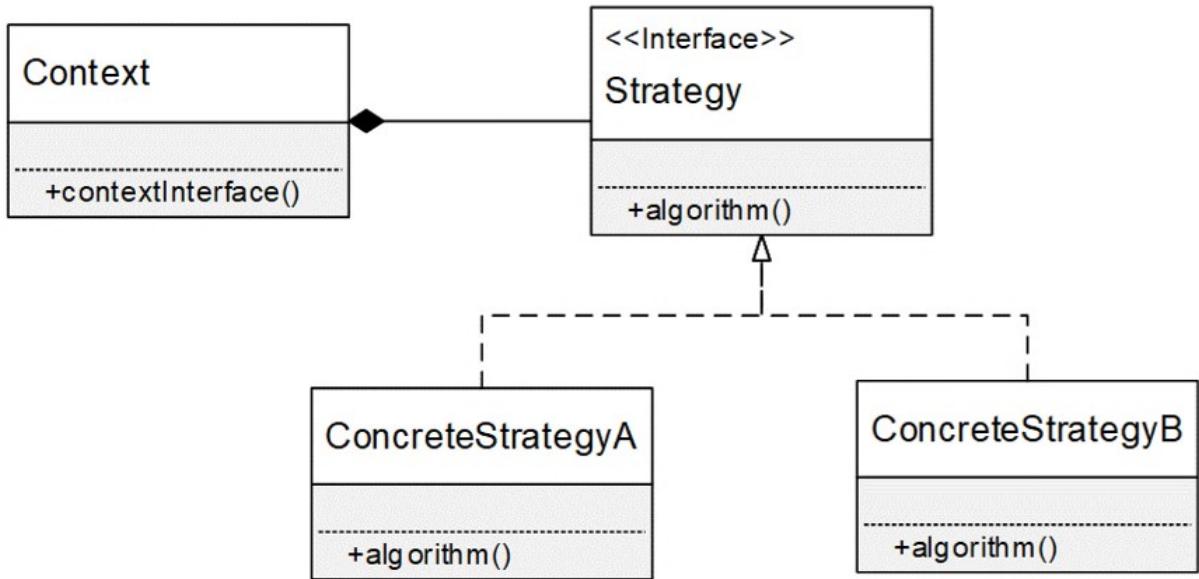
Problema

- 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

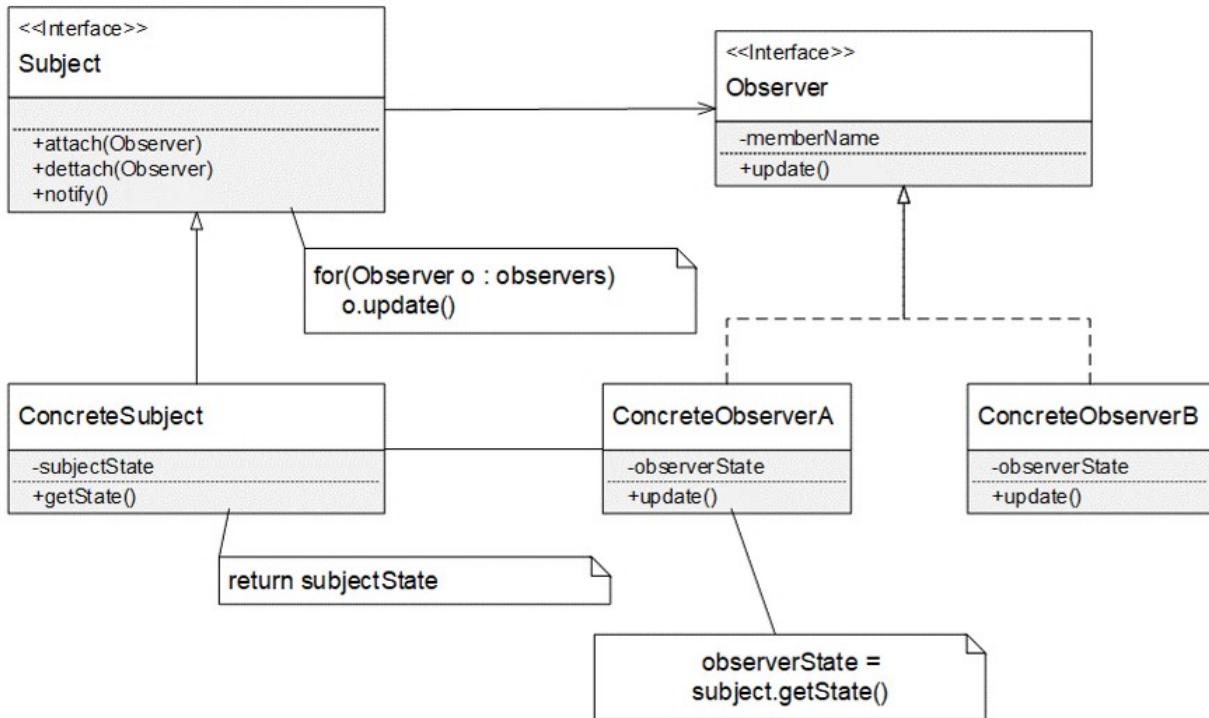
Problema

- 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
 - Gestionația 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 sînt executate 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

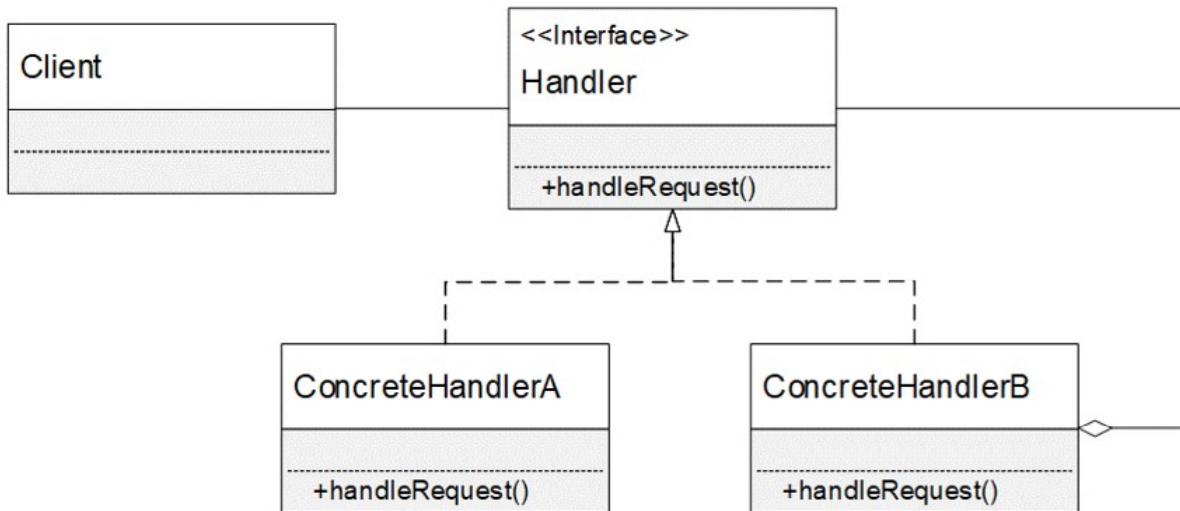
Scop

- 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

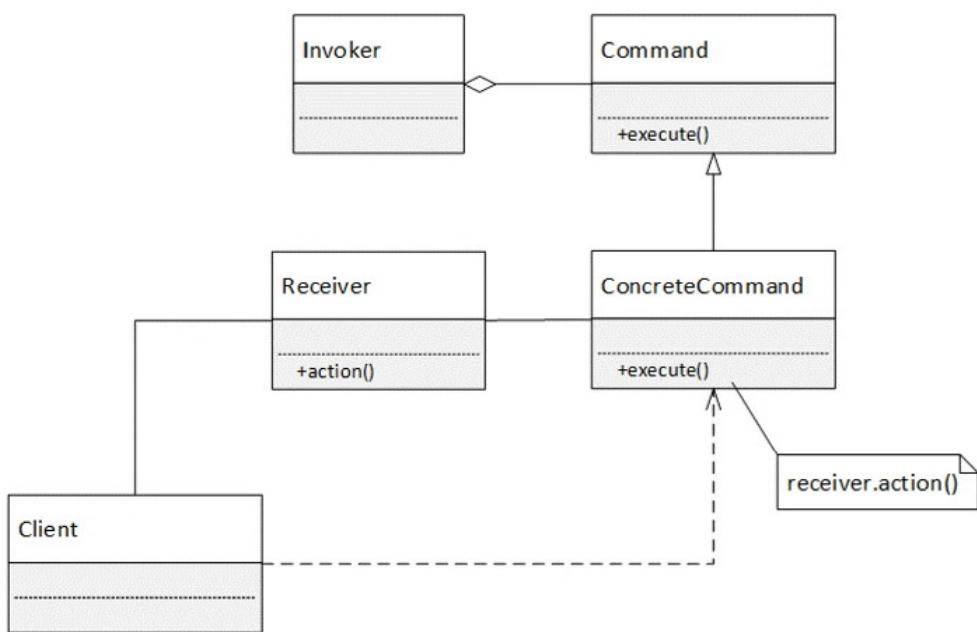
Problema

- 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

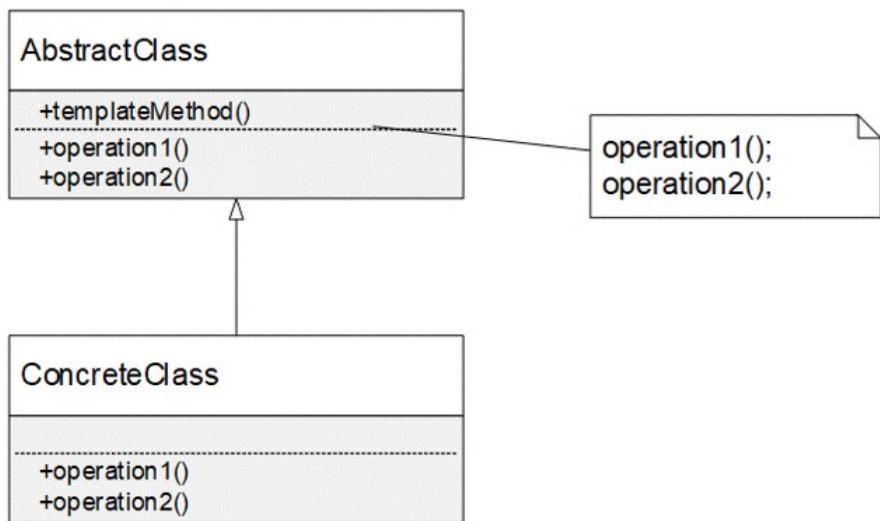
Scop

- 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

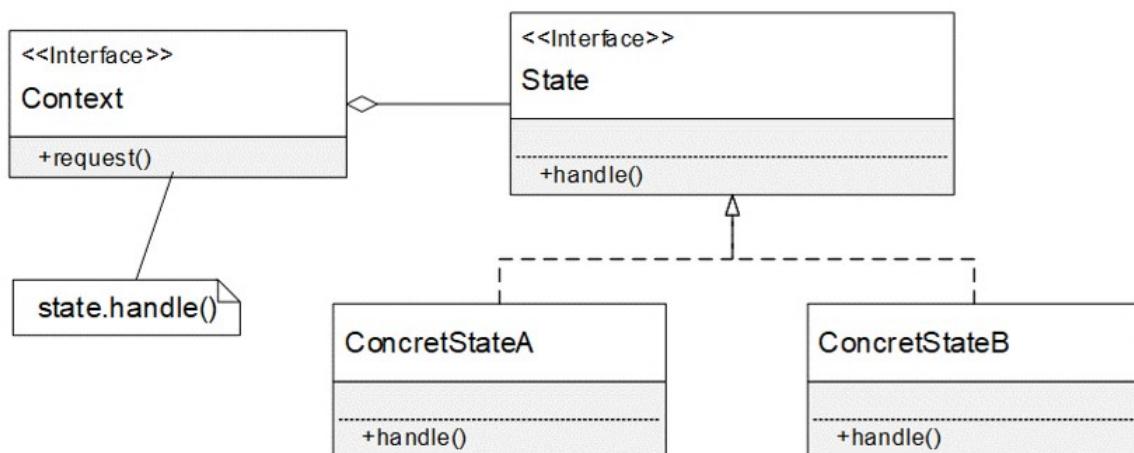
Problema

- 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

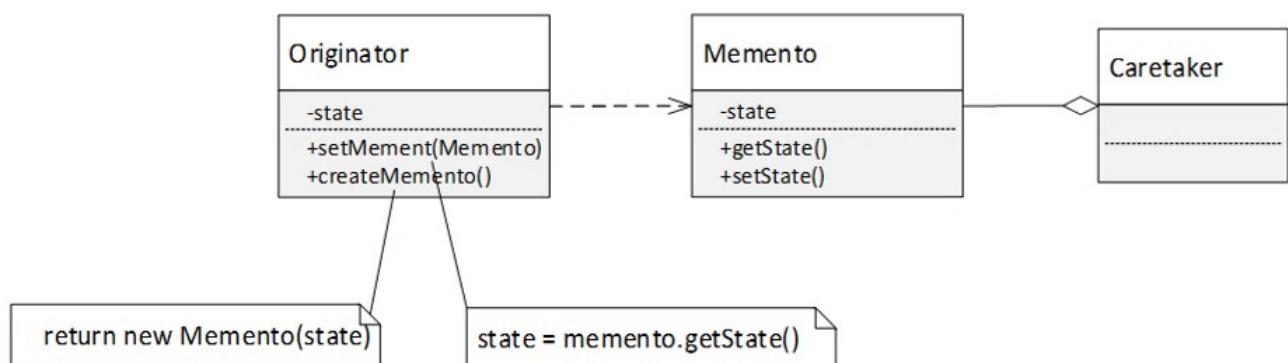
Scop

- 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

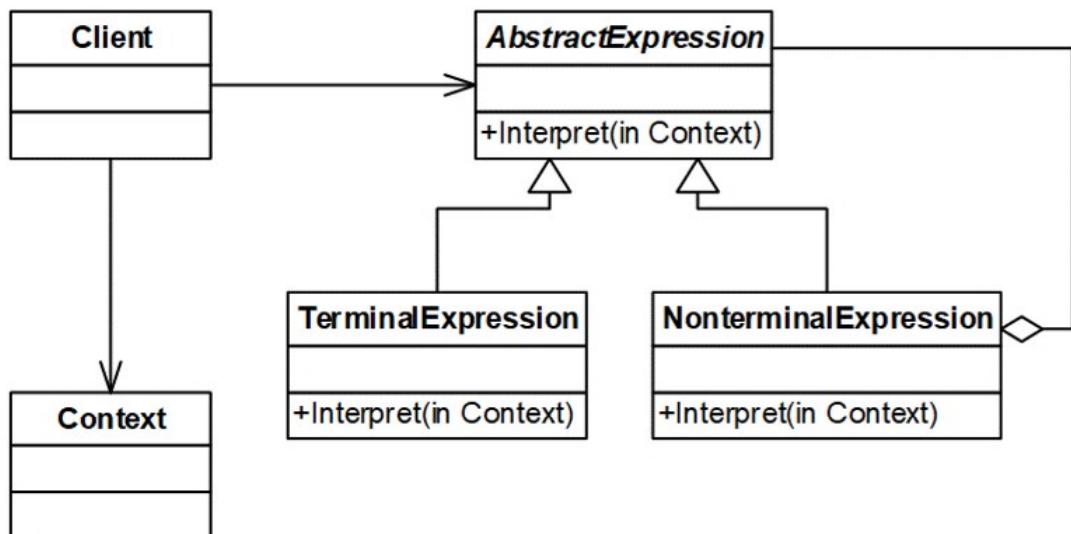
Problema

- 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

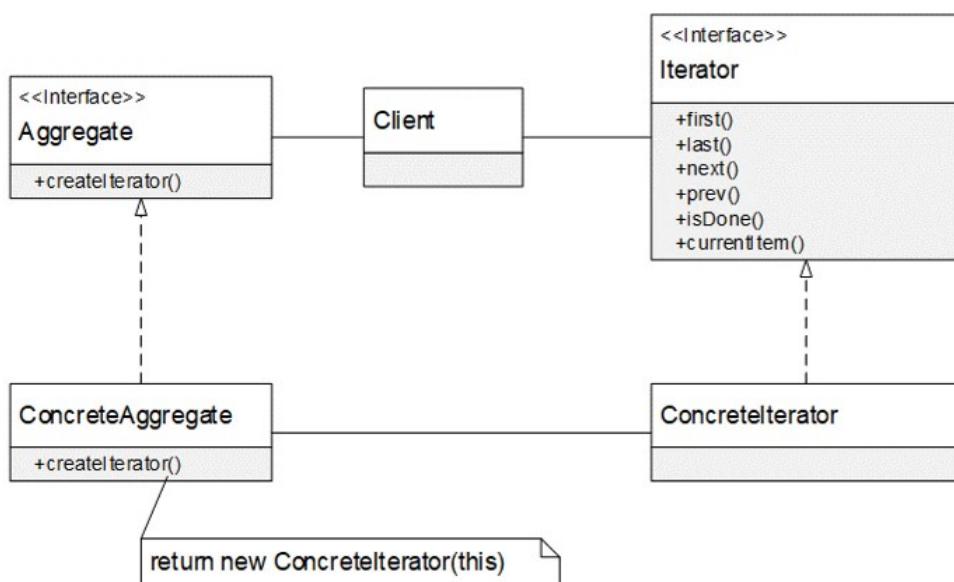
Problema

- 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

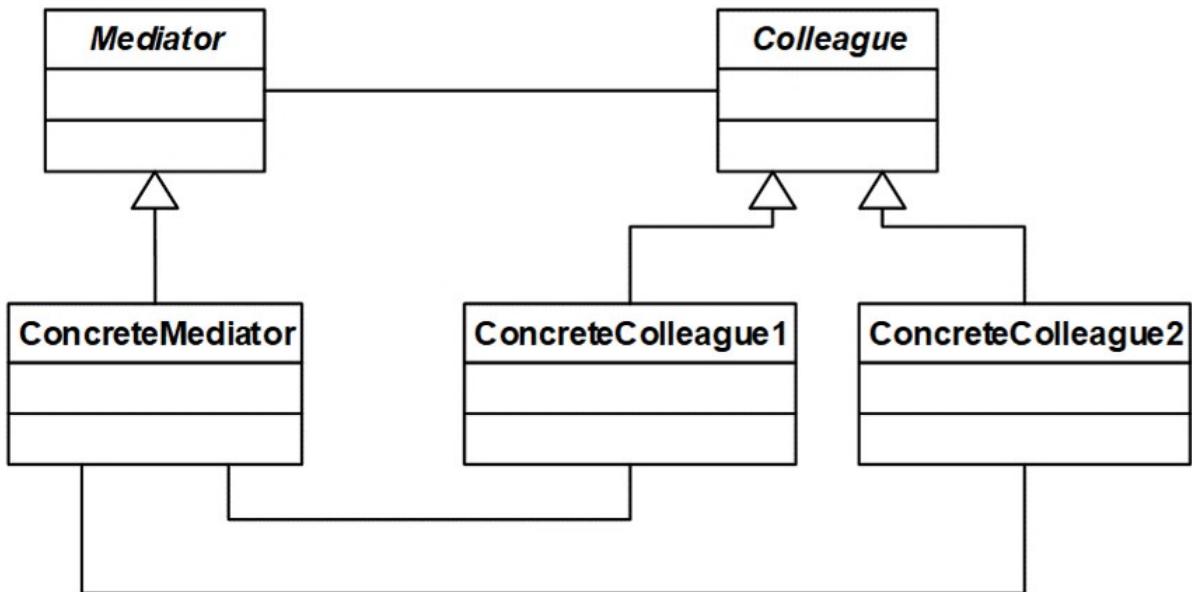
Problema

- 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

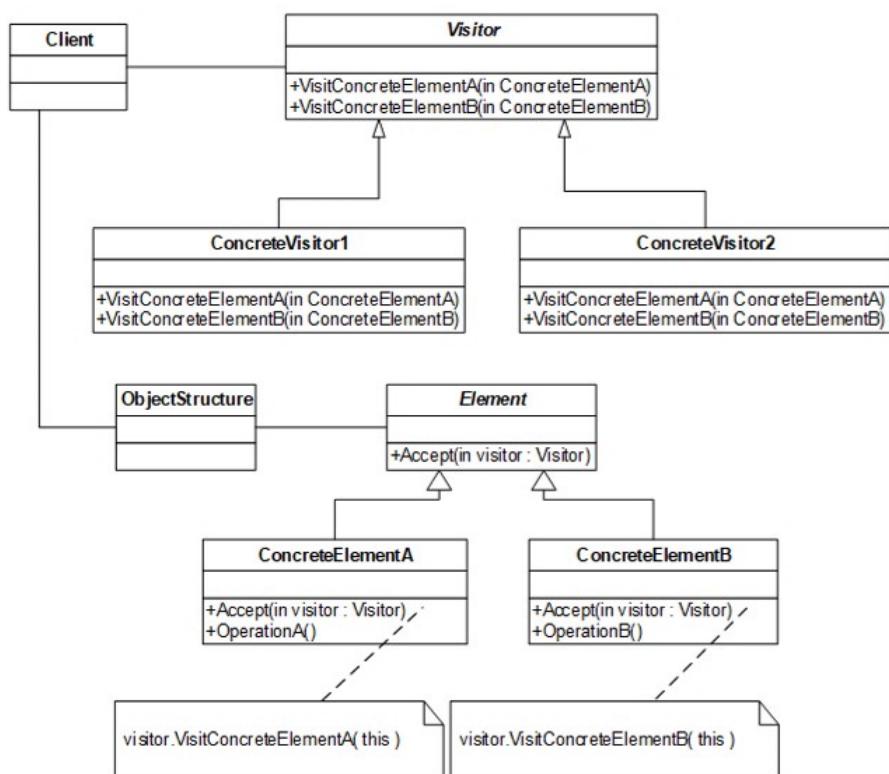
Problema

- 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

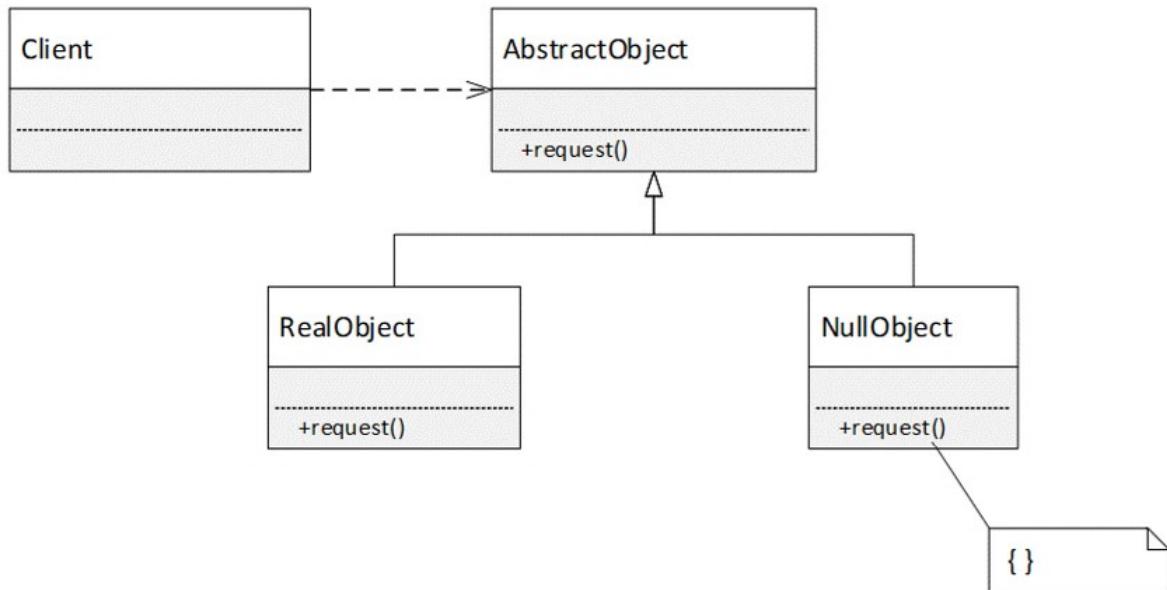
Problema

- 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

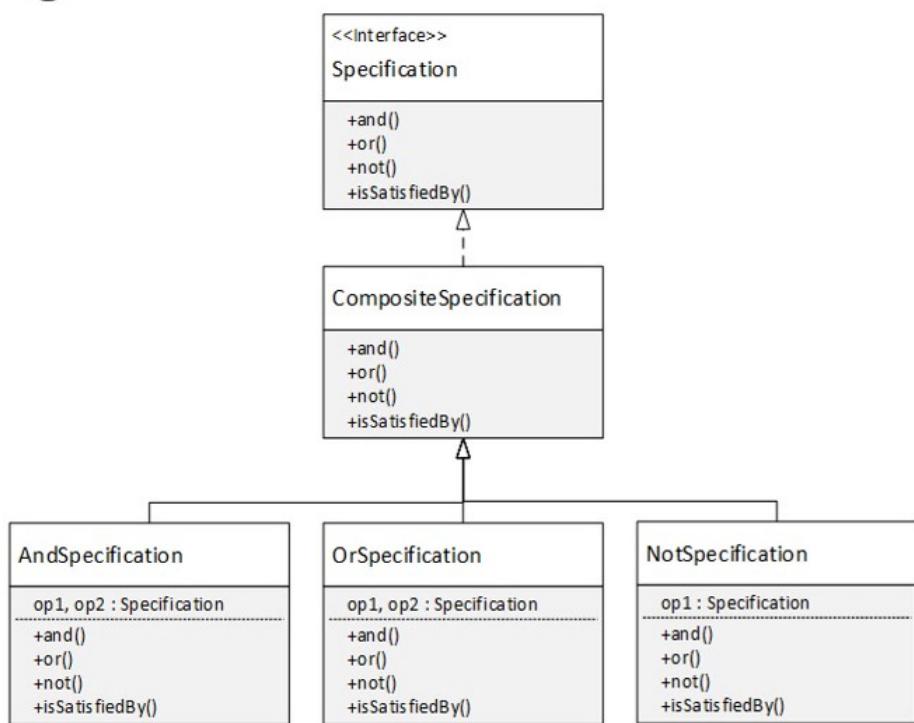
Problema

- 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

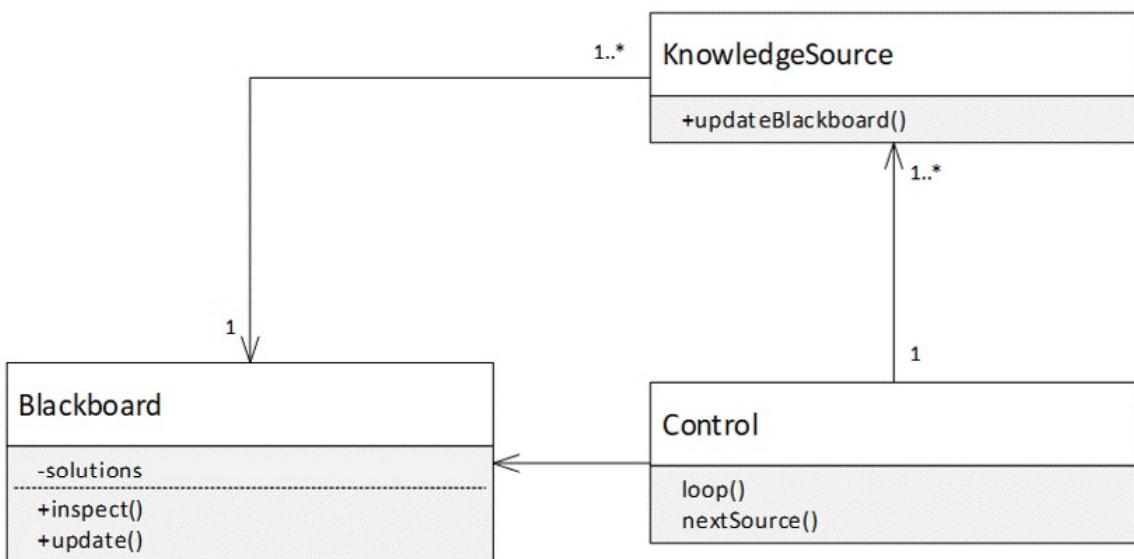
Problema

- 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

Modele de proiectare a aplicațiilor de întreprindere

Cursul 6 - 7

Sumar

- Aplicații concurente
- Modele de proiectare pentru aplicații concurente

Aplicații concurente

- Fire multiple de execuție
- Execuție asincronă
- Partajarea resurselor între mai multe fire de execuție sau procese
- Sincronizare
- Planificare
- Gestiunea evenimentelor

Probleme uzuale

- Resurse partajate
 - Blocaje (deadlock)
- Ordinea operațiilor
 - Dependențe între operații

Concepte specifice

- Proces
- Fir de execuție
- Resurse partajate
- Secțiune critică
- Resurse critice
- Sincronizare execuției
- Excludere mutuală
 - Semafoare
 - Mutex (Semafor binar)
- Variabile condiție
 - blocarea firului curent
 - notificare deblocare
- Blocare/Impas (Deadlock)
- Planificarea execuției

Java

- **ExecutorService**
 - Multime de fire de execuție
 - Metode execute(), shutdown()
- **Executors.newFixedThreadPool(n) -> ExecutorService**
- **wait, notify, notifyAll**
 - Firul curent va aștepta (wait) pînă când este obiectul este notificat (notify, notifyAll)
- **volatile**
 - garanteaza vizibilitatea unui obiect între fire de executie
- **Lock/ReentrantLock**
 - lock(), unlock()
- **Condition**
 - boolean awaitUntil(Date deadline)throws InterruptedException
 - Firul curent va astepta pînă când acesta este semnalat sau întrerupt, sau se ajunge la termenul specificat
 - awaitUninterruptibly()
 - Firul curent va aștepta pînă când acesta este semnalat

C#

- Thread
- Monitor
- Lock

Modele de proiectare pentru aplicații concurente

- Double-Checked Locking
- Single Threaded Execution
- Lock Object
- Scheduler
- Future
- Active Object
- Half-Sync/Half-Async
- Monitor Object
- Producer/Consumer
- Two-Phase Terminator
- Double Buffering
- Balking
- Guarded Suspension
- Reactor
- Read Write Lock
- Thread Pool
- Thread-Specific Storage
- Asynchronous Processing
- Leader/Followers

Double Checked Locking

Problema

- Există un acces concurent în crearea de obiecte
 - În cazul în care se dorește crearea unei singure instanță a aceleiași clase (de exemplu, modelul Singleton), poate nu este suficient să se realizeze o singură verificare pentru existența instanței, atunci cînd există mai multe fire de execuție
- Există un acces concurent la o metodă
 - Comportamentul acesteia se modifică în funcție de anumite constrângeri, care se modifică în cadrul acestei metode

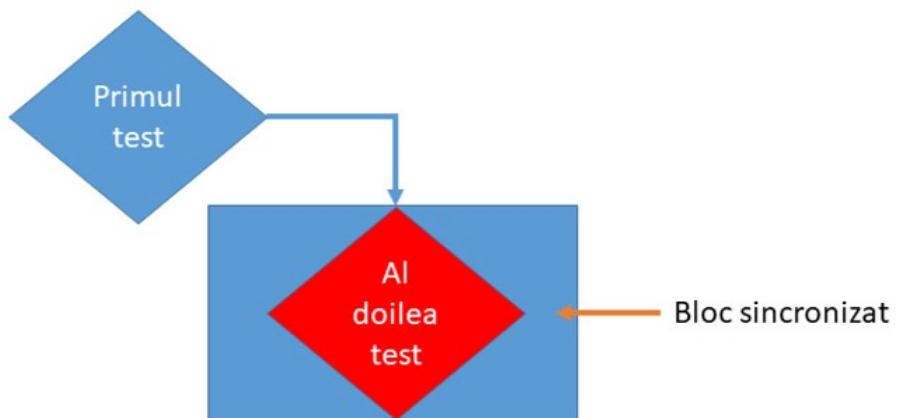
Scop

- Reducerea efortului de obținere a unei blocări, verificînd mai întâi criteriul de blocare, fără a obține, de fapt, blocarea
- Dacă verificarea criteriului de blocare indică faptul că este necesară blocarea, logica reală de blocare va continua

Implementare

- Utilizare la inițializarea întîrziată (singleton)
- Sincronizarea se realizează la nivel de bloc și nu de metodă

Diagrama



Single Threaded Execution

Problema

- Accesul la date/resurse partajate conduce la rezultate eronate în cazul apelurilor concurențiale
- Prevenirea apelurilor concurențiale la resurse sau la date
- Exemplu
 - Accesul la datele membre ale unui obiect

Scop

- Asigurarea că operațiile care nu se pot realiza corect într-un context concurențial nu se execută astfel
- Sincronizarea unor secvențe de cod care nu pot fi executate în mod concurrent

Diagrama de clase

ResursaPartajata	<ul style="list-style-type: none">• metodaSigura<ul style="list-style-type: none">• poate fi apelată în siguranță din mai multe fire de execuție• metodaNesigura<ul style="list-style-type: none">• nu poate fi apelată în siguranță din mai multe fire de execuție• se permite un singur apel la un moment dat
metodaSigura1 metodaSigura2 metodaSigura3 metodaNesigura1 {guarded} metodaNesigura2 {guarded}	

Implementare

- Declararea metodelor ca fiind sincronizate
- Utilizarea altor mecanisme specifice de sincronizare

Dezavantaje

- Performanțele codului se pot degrada
- Pot apărea situații de blocaj (deadlock)

Lock Object

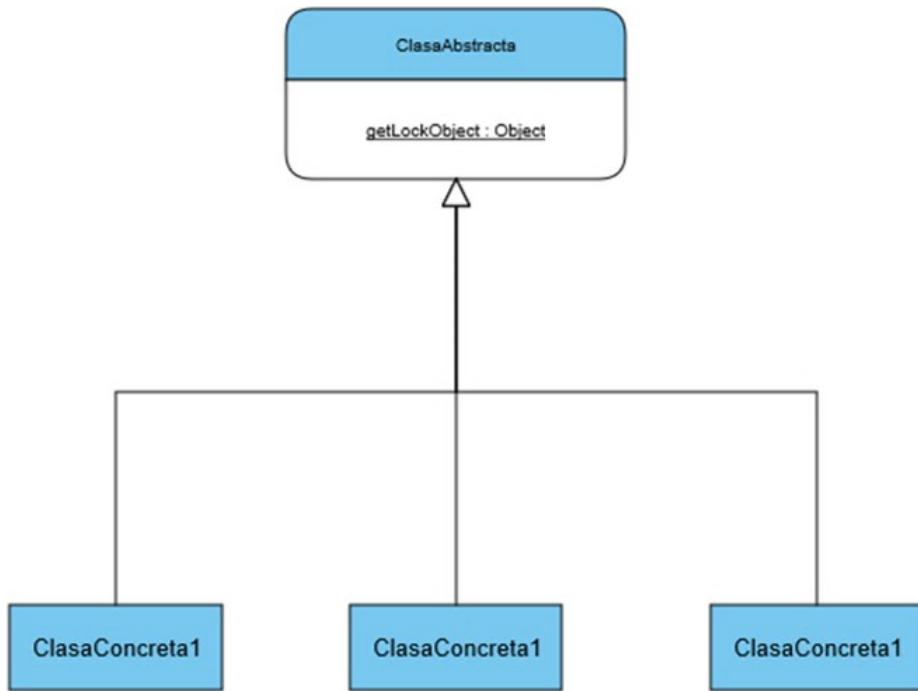
Problema

- Accesul la date/resurse partajate conduce la rezultate eronate în cazul apelurilor concurențiale
- Prevenirea apelurilor concurențiale
- Blocarea unui set arbitrar de obiecte crește complexitatea și scade eficiența

Scop

- Accesul exclusiv al unui fir de execuție la mai multe obiecte
- Utilizarea unui singur obiect care permite blocarea accesului
- Rafinare a modelului Single Threaded Execution

Diagrama de clase



Future

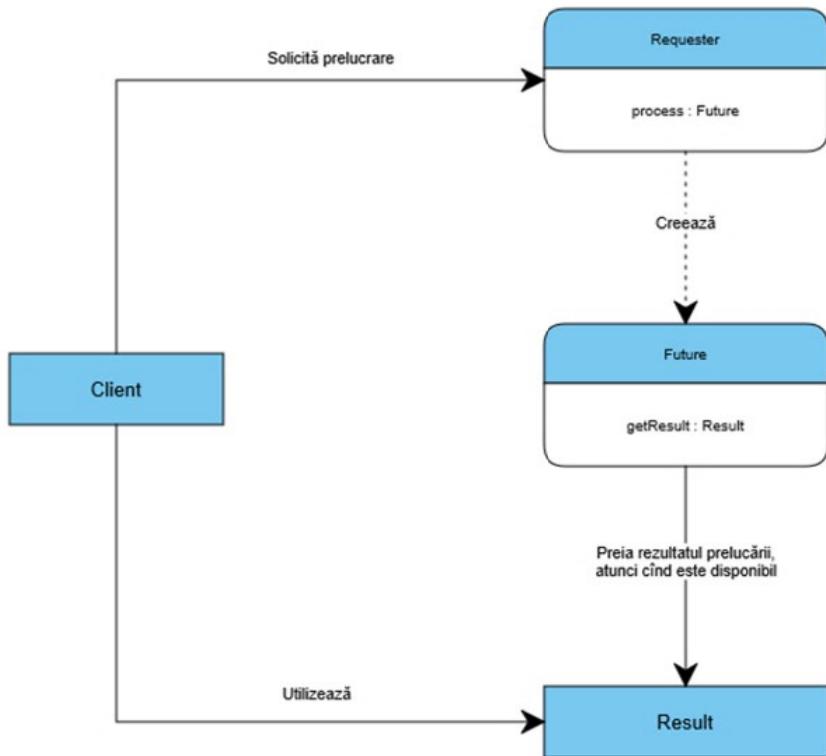
Problema

- Efectuarea unor prelucrări în alt fir de execuție față de cel care utilizează rezultatele
- Necesitatea preluării rezultatelor după ce obținerea acestora, independent de modul de lucru (sincron sau asincron)
- Utilizarea unui model de tip **Observer** ar putea fi implementat, dar cu acces dintr-un singur fir de execuție
 - Creștea complexitățea codului

Scop

- Încapsularea rezultatului unei prelucrări și punerea la dispoziție a unui mecanism care să ascundă modul de prelucrare (sincron/asincron)
- Rezultatul este accesibil prin intermediul unei metode dedicate
 - Dacă rezultatul este disponibil, va fi returnat imediat
 - Dacă rezultatul nu este disponibil, se aștepta pînă va fi deveni disponibil

Diagrama



Implementare

• **Result**

- Încapsulează rezultatul unei prelucrări

• **Client**

- Inițiază procesul de efectuare a prelucrărilor
- Preia rezultatul

• **Requester**

- Inițiază prelucrările
- Returnează un obiect de tip **Future**

• **Future**

- Încapsulează rezultatul unui obiect de tip **Result**

Scheduler

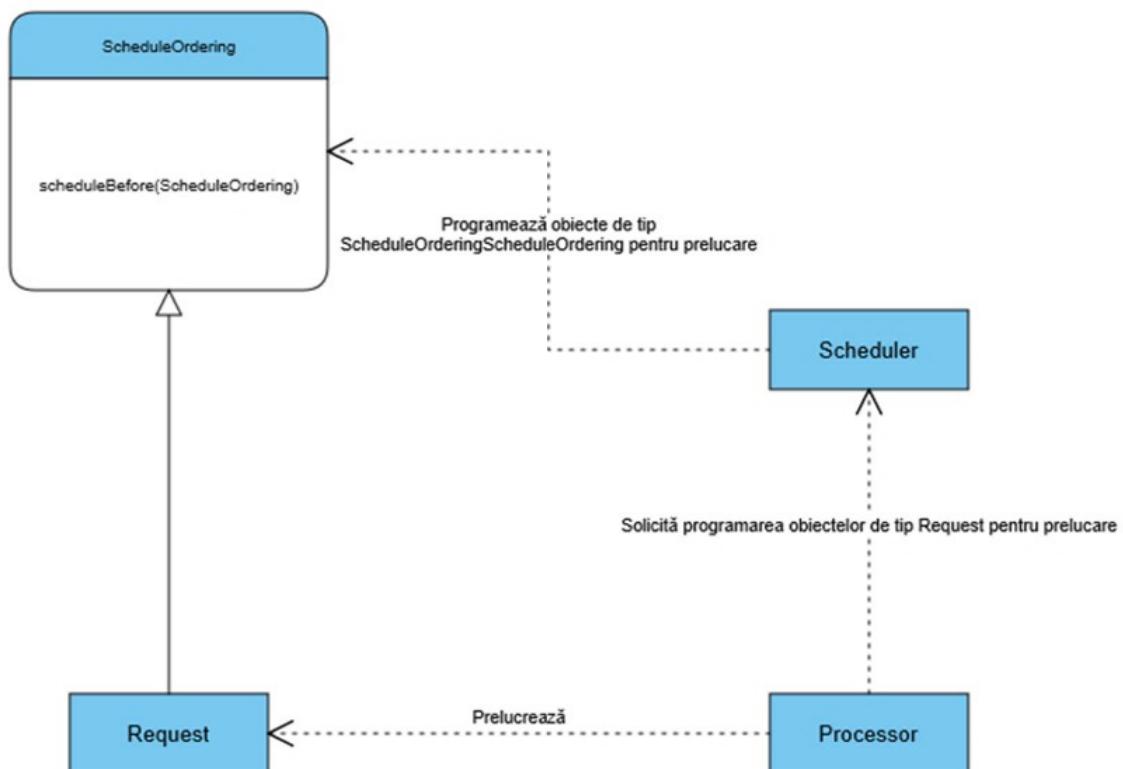
Problema

- La un moment dat, mai multe fire de execuție trebuie să acceseze o resursă, dar doar un singur fir de execuție poate accesa acea resursă
- Există restricții cu privire la ordinea în care firele de execuție pot accesa o resursă

Scop

- Furnizarea unui mecanism generic pentru programarea firelor de execuție
- Controlează ordinea de în care se execută firele

Diagrama de clase



Implementare

- **SchedulerOrdering**

- Interfață care permite stabilirea modalității de selecție a firului care va fi executat

- **Request**

- Încapsulează o cerere ce va fi prelucrată de un obiect de tip **Processor**
 - Implementează interfața **SchedulerOrdering**

- **Processor**

- Efectuează prelucrările reprezentate de obiectele de tip **Request**
 - Este prelucrat un singur obiect la un moment dat
 - Deleagă responsabilitatea programării prelucrării unui obiect de tip **Scheduler**

- **Scheduler**

- Clasă dedicată planificării procesării obiectelor de tip **Request** de către obiectele de tip **Processor**

Active Object

Problema

- Metodele invocate pe un obiect în același timp nu ar trebui să blocheze întregul proces pentru a preveni degradarea calității serviciilor oferite de alte metode
- Accesul sincron la obiectele partajate trebuie să fie simplu
- Aplicațiile trebuie concepute pentru a utiliza în mod transparent paralelismul disponibil pe o platformă hardware/software

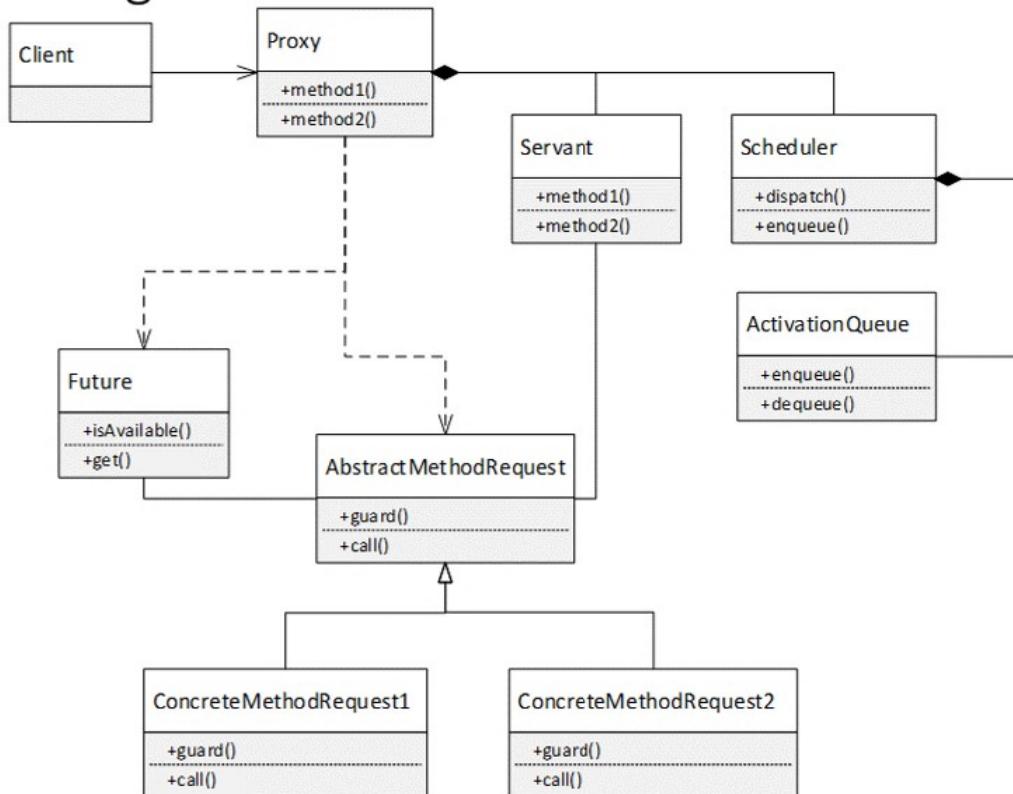
Scop

- Decuplarea execuției unei metode de invocarea acesteia
- Îmbunătățirea concurenței
- Simplificarea accesului la obiecte
- Se pot utiliza pentru implementare modelele Future și Asynchronous Processing

Implementare

- Clientul apelează o metodă prin intermediul unui obiect de tip proxy
- Dacă se așteaptă un rezultat, obiectul de tip proxy creează un obiect, asociat acestuia
- Obiectul de tip proxy creează o cerere către metoda care urmează să fie apelată
- Pentru a evita condițiile de concurență, cererile primite de la clienți sunt plasate în coadă și manipulate de un planificator
- Planificatorul alege un obiect în așteptare și îl lansează în execuție
- Este responsabilitatea obiectului să știe ce să facă atunci când se invocă
- Dacă metoda returnează un obiect, acesta este transmis către client
- Uzual, se utilizează transmiterea de mesaje

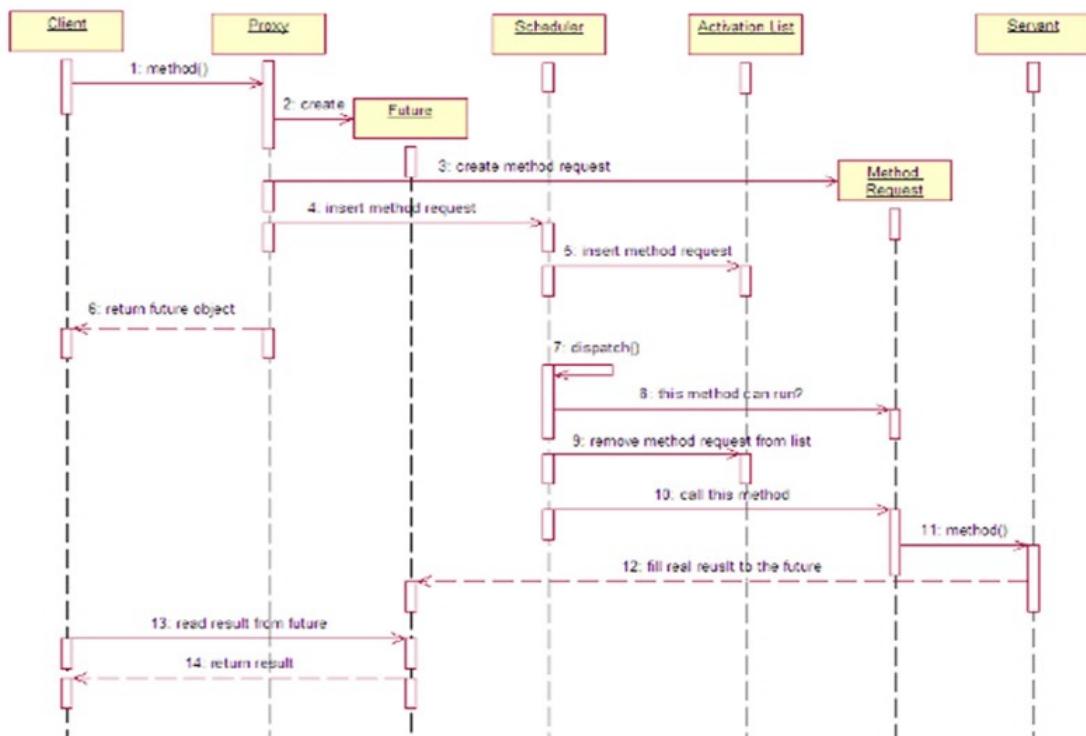
Diagrama de clase



Componente

- **Proxy**
 - Interfață care permite accesul clientilor la un obiect
- **AbstractMethodRequest, ConcreteMethodRequest**
 - Declararea și implementarea metodelor apelate prin intermediul obiectului de tip Proxy
- **Servant**
 - Implementează cererile de metode
- **Scheduler**
 - Recepționează cererile de apel la metode și le repartizează către Servant în momentul în care sunt disponibile
- **ActivationQueue**
 - Gestionează lista de așteptare a cererilor de metode
- **Future/Callback**
 - Permite accesul clientilor la rezultatele metodelor apelate prin intermediul obiectului de tip Proxy

Diagrama de stare



Avantaje

- Reducerea complexității codului:
 - Odată ce modelul este implementat, codul poate fi tratat ca un singur fir
- Nu este nevoie de sincronizare suplimentară
 - Solicitările simultane sunt serializate și manipulate de un singur fir intern

Dezavantaje

- Impactul asupra performanțelor
 - Planificarea apelurilor și gestiunea cererilor pot fi costisitoare din punct de vedere al memoriei
 - Contextul non-trivial se poate modifica
- Impactul asupra codului
 - Necesară crearea unui framework redus
 - Există mai multe componente (Proxy, Scheduler, Servant etc.)

Monitor Object

Problema

- Multe aplicații conțin obiecte ale căror metode sunt invocate simultan de clienți mulți
- Aceste metode modifică adesea starea obiectelor lor
- Pentru ca astfel aplicații concomitente să se execute corect este necesară sincronizarea și planificarea accesului la obiecte

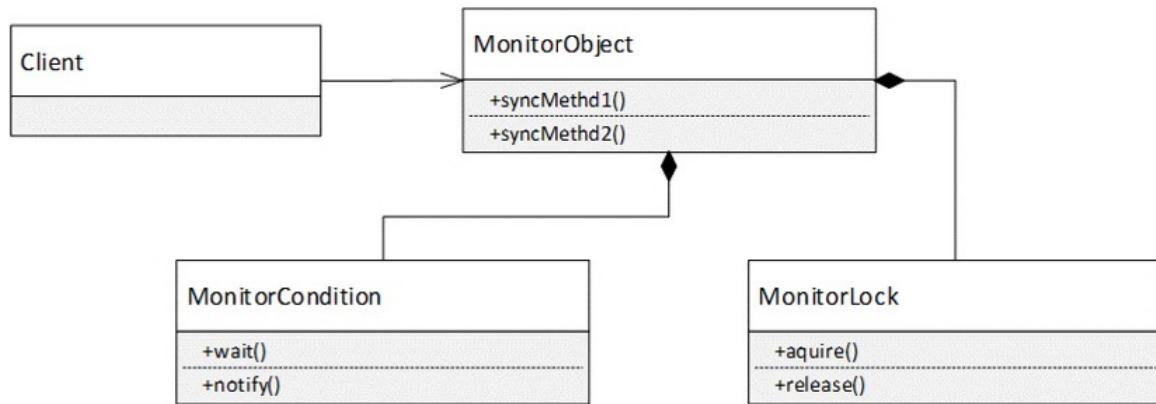
Scop

- Sincronizarea executării concurențială a metodelor pentru a se asigura că în cadrul unui obiect se execută o singură metodă
- Permite metodelor unui obiect de a programa în mod cooperativ secvențele de execuție ale acestora

Implementare

- Se definește un obiect responsabil cu sincronizarea
- Se asociază un obiect de blocare
- Se implementează un mecanism de suspendare și reluare a execuției
- Spre deosebire de modelul de proiectare *Active Object*, în cazul *Monitor Object*, nu există un fir separat de control
- Fiecare solicitare primită este executată în firul de control al clientului propriu-zis iar, până la revenirea metodei, accesul este blocat
- La un singur interval de timp, o singură metodă sincronizată poate fi executată într-un singur monitor

Diagrama de clase



Componente

- **MonitorObject**

- Definește obiectul care este accesat concurent

- **SynchronizedMethod** (`syncMethd1()`, `syncMethd2()` etc.)

- Implementează metodele care disponibile în cadrul obiectului accesat concurent

- **MonitorLock**

- Asigură că o singură metodă va fi executată la un moment dat

- **MonitorCondition**

- Stabilește circumstanțele în care se execută metodele sincronizate

Avantaje

- La un moment dat, doar o singură metodă sincronizată rulează în cadrul unui obiect
- Separarea blocării (realizată la nivel scăzut) de implementarea sincronizării
- Capacitatea de a suspenda și de a relua executarea în cadrul unei metode
- Mecanism de blocare bine cuplat pentru o performanță sporită

Dezavantaje

- Apelarea lui *wait* într-o stare instabilă
- Nu este eliberată blocarea atunci când apare o excepție
- Nu se realizează sincronizarea metodei când este necesar

Producer Consumer

Problema

- Decuplarea sistemul prin separarea efortului în două procese: producere și consum
- Abordează problema diferitelor durate necesare pentru a produce un rezultat sau a consuma o resursă

Scop

- Model clasic de concurență
- Reduce legătura dintre producător și consumator prin separarea identificării de executarea operațiilor
- Permite producerea secvențială și prelucrarea obiectelor, în mod coordonat, într-o aplicație concurrentă

Implementare

- Producătorul
 - Generează elemente
 - Stochează elementele într-o structură partajată
- Consumatorul
 - Preia elementele din structura partajată
 - Utilizează elementele

Implementare

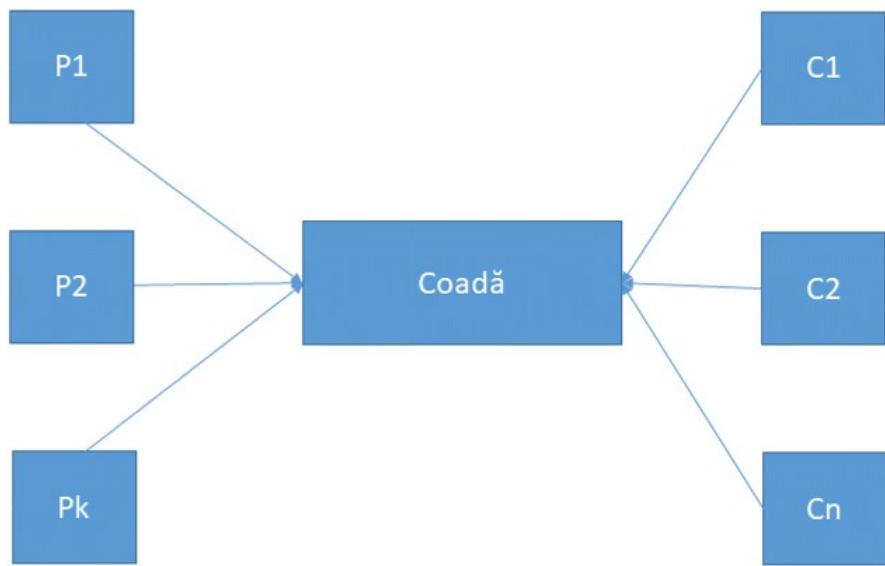
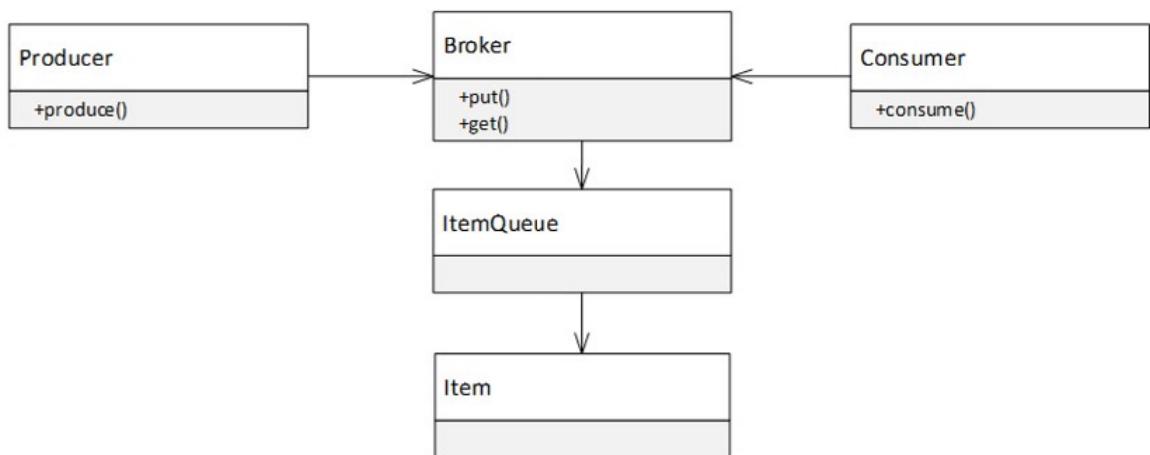


Diagrama de clase



Componente

- **Producer**

- Produce elemente și le adaugă în structura partajată

- **Consumer**

- Utilizează elementele și le elimină din structura partajată

- **Item**

- Elemente produse de *Producer* și utilizate de *Consumer*

- **ItemQueue**

- Stochează elementele produse de *Producer* și utilizate de *Consumer*

- **Broker**

- Gestioneaază accesul la structura partajată (*ItemQueue*)

Half-Sync/Half-Async

Problema

- Există sisteme care au următoarele caracteristici:
 - Trebuie efectuate sarcini ca răspuns la evenimentele externe care apar asincron (ex. întreruperi hardware)
 - Este ineficient să se dedice un fir separat de control pentru a efectua operații de I/O sincron pentru fiecare sursă externă de eveniment
 - Sarcinile de nivel mai înalt în sistem pot fi simplificate semnificativ dacă operațiile de I/O sunt efectuate sincron
- Una sau mai multe operații dintr-un sistem trebuie să ruleze într-un singur fir de control
 - Alte operații pot beneficia de mai multe fire de execuție

Problema

- Sistemele concurente includ atât servicii asincrone cât și sincrone
- Implementările asincrone sunt, în general, mai eficiente
 - anumite servicii pot fi asociate direct pe mecanisme asincrone (manipulatoare de întreruperi hardware sau de semnale software)
- Implementările sincrone simplifică efortul de programare
 - anumite servicii pot fi constrânsă să ruleze la puncte bine definite în secvență de prelucrare

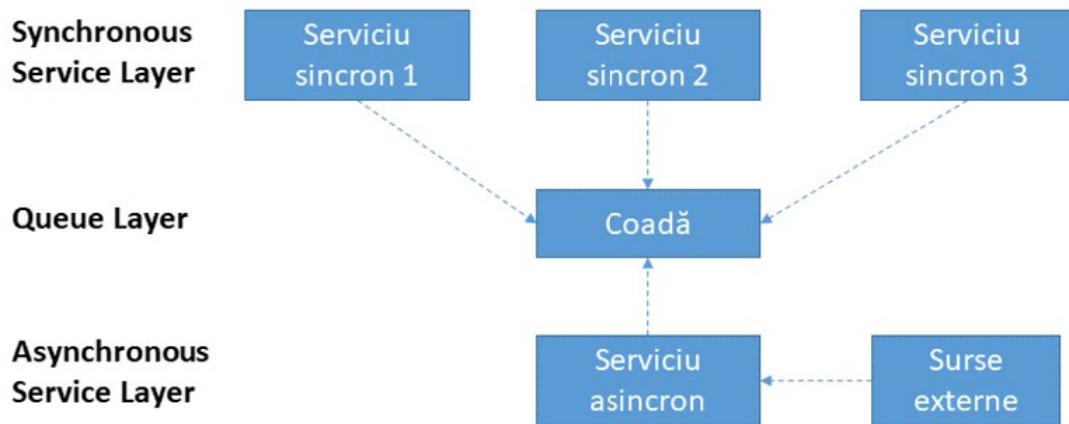
Scop

- Deconectarea procesărilor asincrone și sincrone în sistemele concurente
- Introducerea a două niveluri de comunicare (unul pentru asincron, altul pentru sincron)
 - Simplificarea scrierii programelor
 - Fără reducerea necorespunzătoare a performanței

Componente

- **Synchronous Service Layer (SSL)**
 - Execută procesarea de nivel înalt în mod sincron
- **Asynchronous Service Layer (ASL)**
 - Execută procesarea de nivel scăzut în mod asincron
- **Queue Layer (QL)**
 - Asigură o zonă tampon între nivelurile sincron și cel asincron
- **External Events Source**
 - Generează evenimente recepționate și procesate de ASL

Diagrama de componentă



Two-Phase Terminator

Problema

- Existența unui proces sau fir de execuție care rulează nedefinit
- În cazul apariției unor situații excepționale, terminarea forțată ar conduce la efecte neprevăzute
- Este necesar ca firul de execuție sau procesul să aibă timp, înainte de terminare, să elibereze resursele și să finalizeze anumite operații

Scop

- Permite închiderea ordonată a unui fir de execuție sau a unui proces în cadrul unei aplicații concurentă
- Firul de execuție poate elibera resursele înainte de încheierea execuției

Diagrama de clase

Terminator
-solicitareIncheiereExecutie
esteSolicitataIncheiereaExecutiei incheieExecutia

- **esteSolicitataIncheiereaExecutiei**
 - Dacă returnează true, firul eliberează resursele și își încheie execuția
- **incheieExecutia**
 - firul își încheie execuția după eliberarea resurselor

Double Buffering

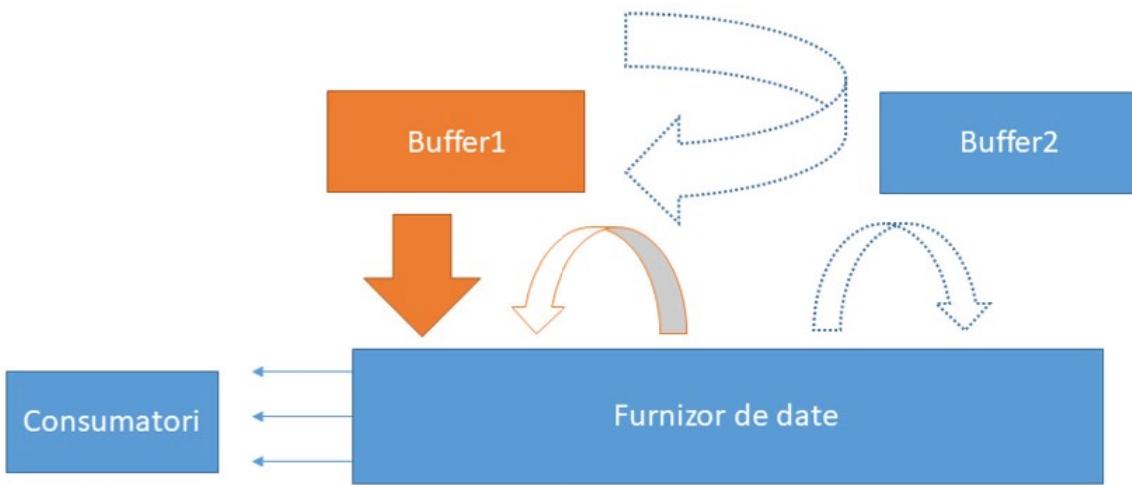
Problema

- Un obiect utilizează foarte multe date la un moment dat
- Există un obiect care produce date; acesta nu are control asupra momentului în care datele vor fi utilizate
- Pot apărea probleme legate de performanță, dacă datele nu sunt disponibile pentru prelucrare

Scop

- Evitarea întârzierilor în prelucrarea datelor prin producerea în mod asincron a datelor
- Asigurarea datelor înainte de necesitatea prelucrării acestora
- Formă specializată a modelului de proiectare **Producer-Consumer**

Diagrama



Balking

Problema

- Un obiect poate avea o stare care nu permite apelul unei metode
- Execuția metodei nu poate fi amînată, aceasta trebuind să fie executată imediat
- Dacă obiectul nu se află într-o anumită stare, apelul unei metode nu este necesar

Scop

- Permite unui fir de execuție să finalizeze prelucrările dacă o condiție nu a fost îndeplinită
- O metodă nu va fi apelată, dacă nu sunt îndeplinite anumite condiții

Diagrama de clase

ObiectProjejat	
stare	
actualizeazaStare {guarded} metodaProtejata {guarded}	<ul style="list-style-type: none">• actualizeazaStare<ul style="list-style-type: none">• este utilizată pentru modificarea stării obiectului• se apelează doar dacă sănt îndeplinite anumite condiții• metodaProtejata<ul style="list-style-type: none">• este apelată dintr-un fir, iar execuția continuă doar dacă este îndeplinită o anumită condiție

Implementare

- Se testează starea unui obiect (în mod sincron), iar dacă aceasta nu corespunde unei condiții, se ieșe din metodă
- Ieșirea din metodă se poate realiza prin
 - Returnarea unei valori
 - Lansarea unei excepții

Guarded Suspension

Problema

- Metodele unei clase sănătate marcate ca fiind sincronizate
- Starea unui obiect face imposibilă execuția completă a unei anumite metode sincronizate
- Dacă s-ar executa metoda în starea curentă a obiectului ar putea apărea un blocaj (deadlock)

Scop

- Permite unui proces să aștepte pînă când au fost îndeplinite anumite precondiții înainte de procesare
- Similar cu **Balking**, cu diferența că se așteptă îndeplinirea condiției și nu se ieșe din funcție

Diagrama de clase

ObiectProjejat
stare
actualizeazaStare {guarded} metodaProtejata {guarded}

- **actualizeazaStare**
 - este utilizată pentru modificarea stării obiectului
 - se apelează doar dacă sînt îndeplinite anumite condiții
- **metodaProtejata**
 - este apelată dintr-un fir, iar execuția este suspendată pînă cînd este îndeplinită o anumită condiție

Implementare

- Suspendarea metodei, dacă nu este îndeplinită condiția
- La îndeplinirea condiției, firele suspendate își pot relua execuția
- Utilizarea metodelor de tipul wait() și notify()

Read/Write Lock

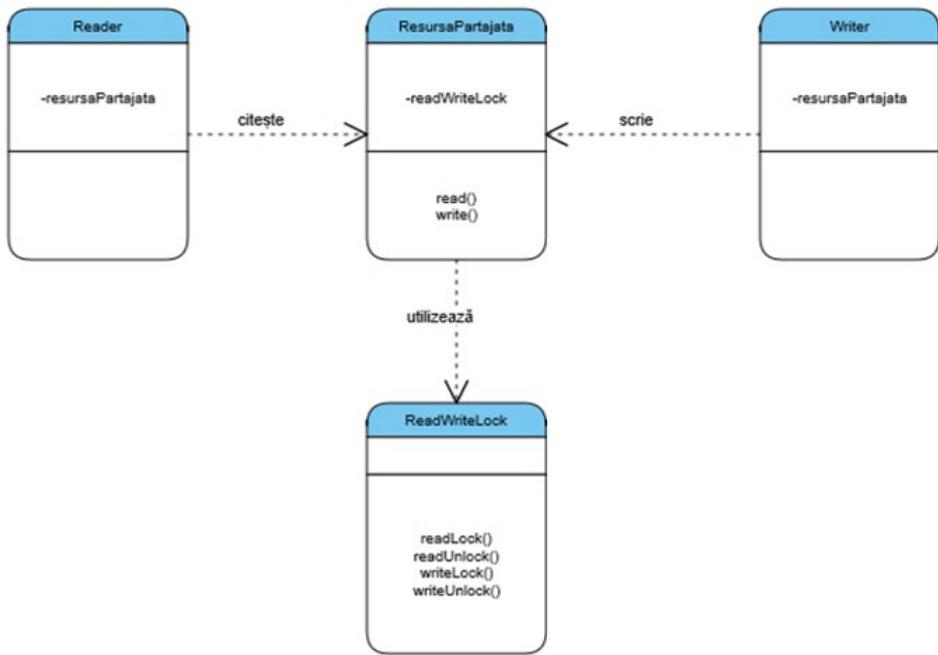
Problema

- Într-un mediu concurențial, citirea stării unui obiect în timpul modificării acesteia poate conduce la inconsistență
- Frecvența citirilor este mai mare decât frecvența scrierilor

Scop

- Permite citiri simultane și scrieri exclusive în cadrul unei aplicații concurente
- Nu se poate scrie în momentul efectuării unei citiri
- Formă specializată de **Scheduler**

Diagrama de clasă



Thread Pool

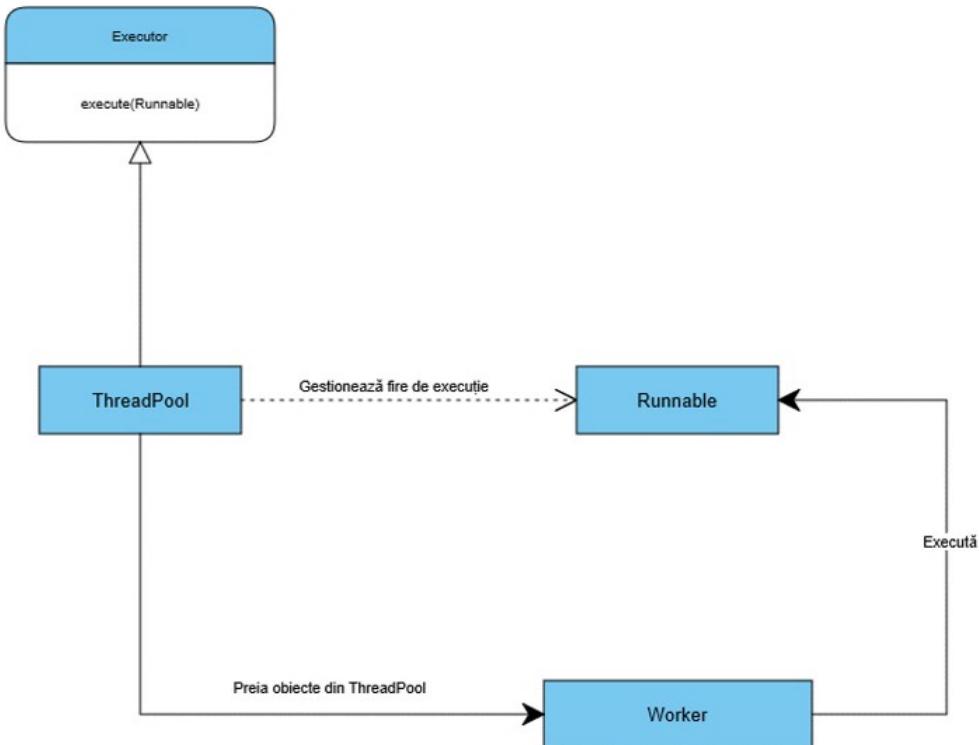
Problema

- Necesitatea efectuării unor operații independente, în fire de execuție distincte
- Costul creării unor fire noi de execuție este ridicat
- Există un număr optim de fire de execuție care pot rula la un moment dat
 - Prea multe fire -> probleme de performanță
 - Prea puține fire -> resurse neutilizate

Scop

- Reutilizarea firelor de execuție
- Evitarea efortului de creare de noi fire de execuție
- Gestiunea numărului de fire existente la un moment dat

Diagrama de clase



Implementare

• Executor

- Interfața definește o metodă care primește ca parametru un obiect de tip **Runnable**, în vederea execuției acestuia

• ThreadPool

- Implementează interfața **Executor**
- Gestioneză o listă de fire de execuție pentru procesare sarcinilor transmise
- Limitează numărul de obiecte de tip **Worker**

• Runnable

- Interfață ce permite execuția într-un fir distinct

• Worker

- Au asociat un fir de execuție
- Primesc obiecte din **ThreadPool** și le execută

Asynchronous Processing

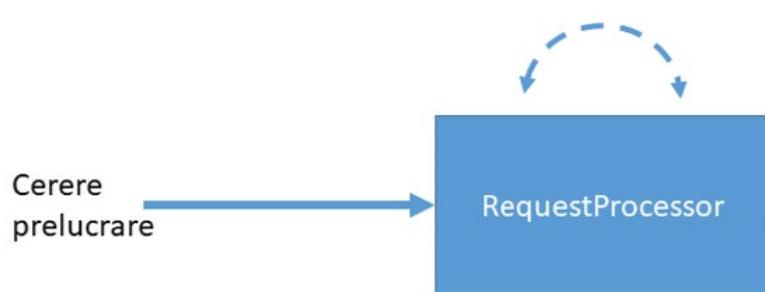
Problema

- Un obiect primește solicitări pentru anumite prelucrări
- Solicitările sunt asincrone, iar clientii nu trebuie să aștepte rezultatul
- Este posibil ca solicitările să fie primite în timpul efectuării unor prelucrări

Scop

- Prelucrarea asincronă a unor solicitări prin secvențializarea acestora
- Se pot utiliza pentru implementare modelele
 - Thread Pool
 - Scheduler
 - Producer-Consumer

Diagrama

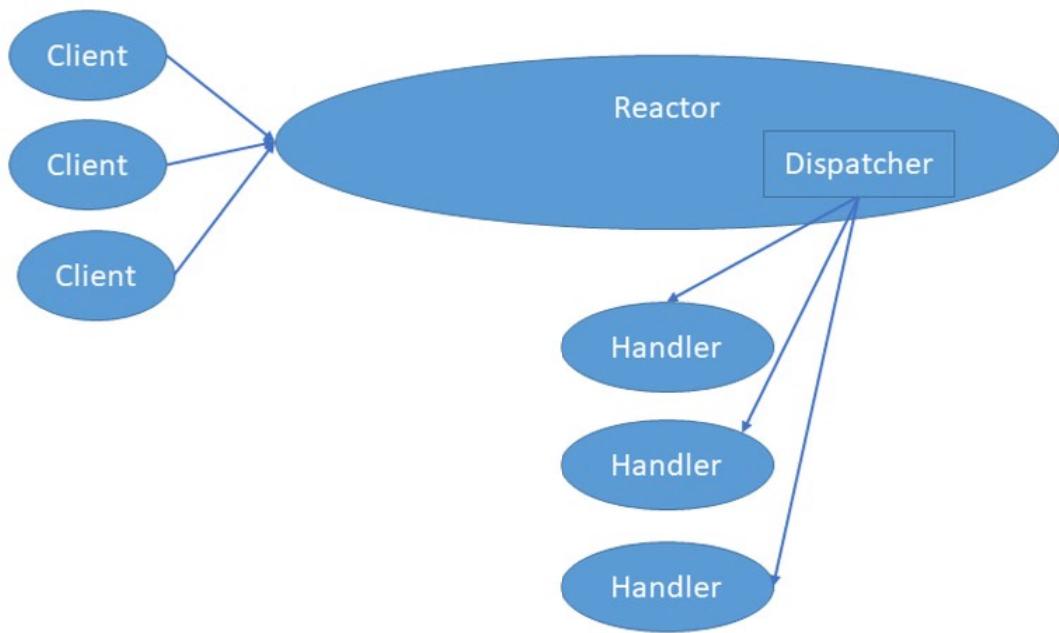


Reactor

Problema

- Aplicațiile bazate pe evenimente trebuie să gestioneze diferite cereri, destinate unor servicii specifice, în mod asincron
- Cererile recepționate în mod asincron trebuie distribuite către serviciile specifice
- Este necesară menținerea scalabilității

Diagrama



Implementare

- **Reactor**

- Rulează într-un fir de execuție distinct
- Reacționează la solicitările de prelucrare (I/O) prin distribuirea acestora către obiectul de tip **Handler** potrivit

- **Handler**

- Prelucrează operațiile de I/O
- Prelucrările se realizează operații fără blocare

Thread-Specific Storage

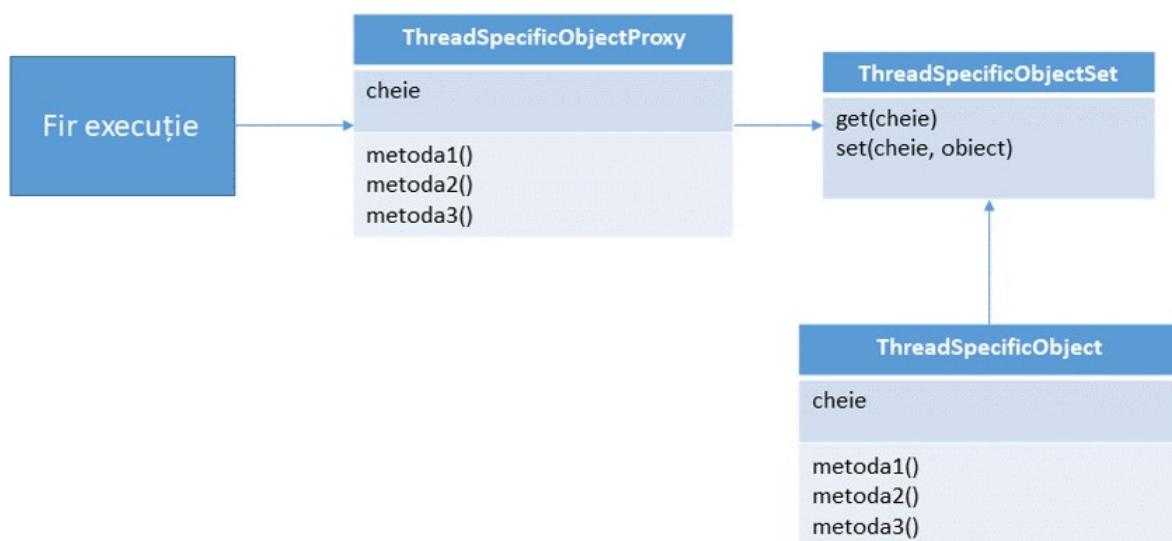
Problema

- Partajarea resurselor într-un context concurențial
- Evitarea blocajelor prin utilizarea obiectelor de sincronizare

Scop

- Accesul firelor de execuție la un obiect local acestora, prin intermediul unui singur punct, global
- Nu implică un mecanism de sincronizare la fiecare acces
- Se realizează copii locale al obiectelor

Diagrama de clase



Implementare

- **ThreadSpecificObject**

- Obiecte accesibile firelor de execuție
- Accesul la aceste obiect se realizează prin intermediul unei chei

- **ThreadSpecificObjectProxy**

- Permite accesul clientilor la obiectele de tip **ThreadSpecificObject**
- Simulează obiectele reale

- **ThreadSpecificObjectSet**

- Gestionează obiectele **ThreadSpecificObject** asociate unui fir de execuție

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 alții, *Head First Design Patterns*, O'Reilly, 2004
- M. Grand, Patterns in Java, Volume 1—A Catalog of Reusable Design Patterns Illustrated with UML, Second Edition, John Wiley & Sons, 2002
- M. Grand, Patterns in Java, Volume 3 - Java Enterprise Design Patterns, John Wiley & Sons, Inc., 2002
- 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

Modele de proiectare a aplicațiilor de întreprindere

Cursul 8 - 9

Sumar

- Modele arhitecturale
- Modele de proiectare pentru aplicații Web

Modele arhitecturale

Modele arhitecturale

- Organizarea sistemelor informaticе
- Abordare la nivel înalt, global
 - Componente
 - Mecanisme specifice sistemului
 - Comunicare
- Aspecte urmărite
 - Scalabilitate
 - Partiționarea sistemului
 - Protocole
 - Interfețe

Modele arhitecturale (exemple)

- Arhitectură stratificată
- Arhitectură condusă de evenimente
- Arhitectură de tip micro-kernel
- Arhitectură bazată de microservicii
- Arhitectură bazată pe cloud

Arhitectura stratificată

Prezentare (interfață utilizator)

Logica organizațională

Servicii

Persistență

Baze de date

Modele de proiectare pentru aplicații Web

- Model View Controller (MVC)
- Model View Presenter (MVP)
- Model View ViewModel (MVVM)
- Model View Template (MVT)
- Front Controller
- Page Controller
- Template View
- Intercepting Filter
- Transform View
- Application Controller
- Two Step View

Model View Controller (MVC)

Necesitate

- Separarea rolurilor
- Testabilitate
- Modularitate

Model View Controller

- Model arhitectural
- Nivelul de prezentare
- Interacțiunea cu interfața este distribuită în trei roluri distincte
- Este aplicată separarea rolurilor

Componente

- **Model**

- Încapsulează datele și accesul la acestea
- Gestionează starea aplicației
- Notifică *View-ul* cu privire la modificările apărute

- **View**

- Interfața utilizator
- Logica prezentării
- Gestionează interacțiunea cu utilizatorul

- **Controller**

- Gestionează interacțiunile
- Coreleză acțiunile utilizatorului cu datele asociate
- Conectează modelul și interfața

Tipuri

- **Model pasiv**

- Modelul nu include suport pentru notificarea schimbărilor

- **Model activ**

- Modelul permite notificarea componentelor abonate

Diagrama (model pasiv)

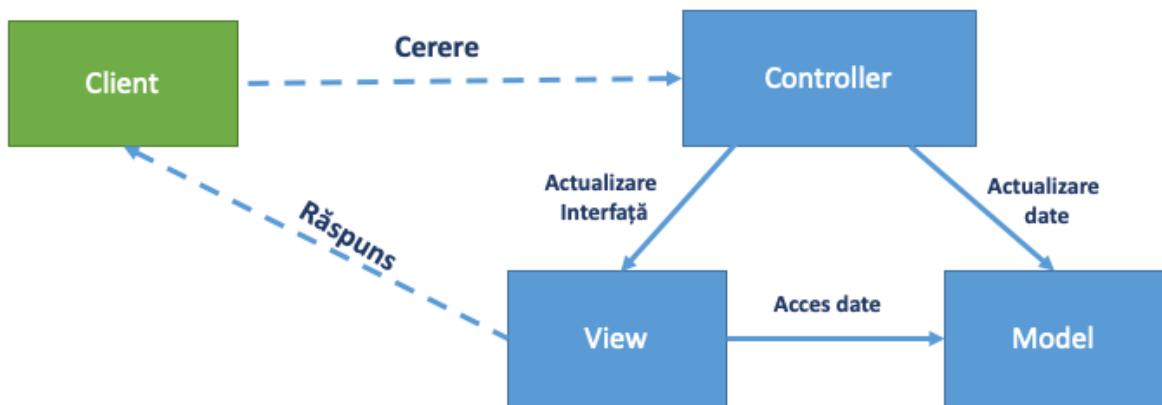
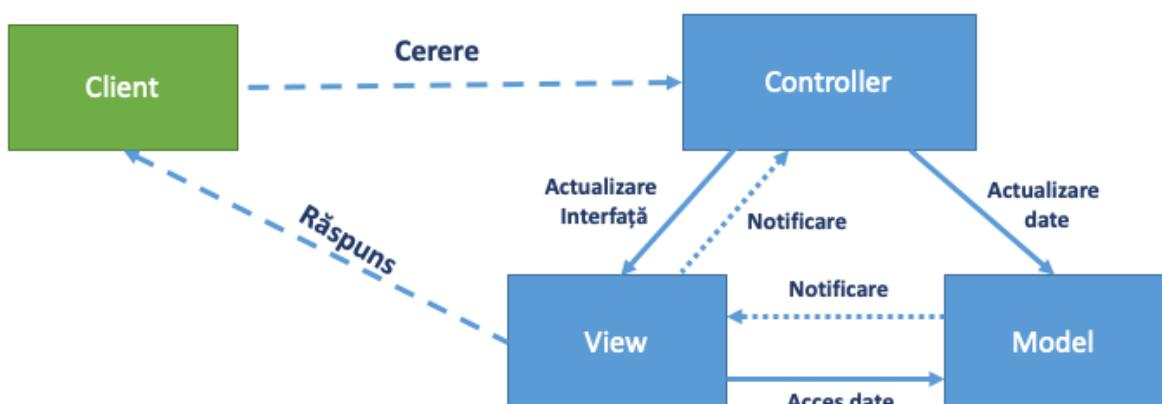


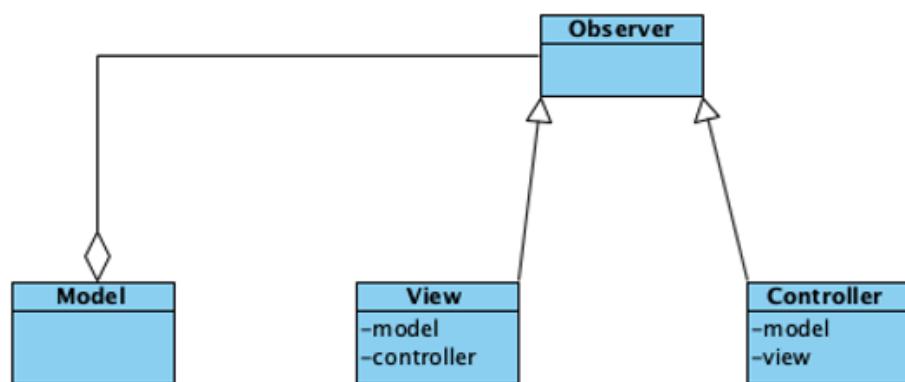
Diagrama (model activ)



Implementare

- View-ul recepționează un eveniment (ex. clic pe un buton)
- Este invocată metoda corespunzătoare din controler
- Controlerul accesează clase/metode din model
- Dacă modelul a fost modificat, se trimit notificări
- În urma notificărilor, view-ul este actualizat

Diagrama de clase (model activ)



Utilizare

- Se instanțiază modelul
- Se instanțiază view-ul, pe baza modelului
- Se adaugă obiectele de tip view la lista de notificări a modelului
- Se instanțiază controlerul, cu referințe la model și view

Implementări cunoscute

- ASP.NET MVC
- Rails
- Spring MVC

MVC

Avantaje

- Separarea interfeței utilizator de model
- Modelul poate fi testat independent
- Existența mai multor componente de interfață pentru același model

Dezavantaje

- Controlerul și view-ul sunt legate destul de puternic
- Complexitate ridicată pentru aplicațiile simple

Model-View-Presenter (MVP)

Model-View-Presenter (MVP)

- Model arhitectural
- Suport pentru automatizarea testării
- Componența de tip View este punctul de intrare al aplicației

Tipuri

- **View pasiv**
 - View-ul nu se actualizează singur
 - View-ul nu este informat cu privire la modificările modelului
 - Presenter-ul gestionează legătura cu modelul
- **Controler supervisor**
 - View-ul poate interacționa direct cu modelul
 - Presenter-ul actualizează modelul
 - Modelul notifică view-ul

Diagrama (View Pasiv)

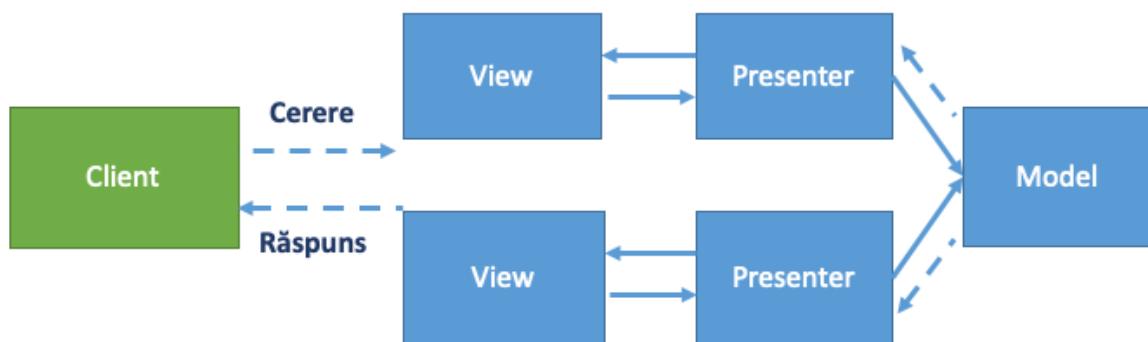
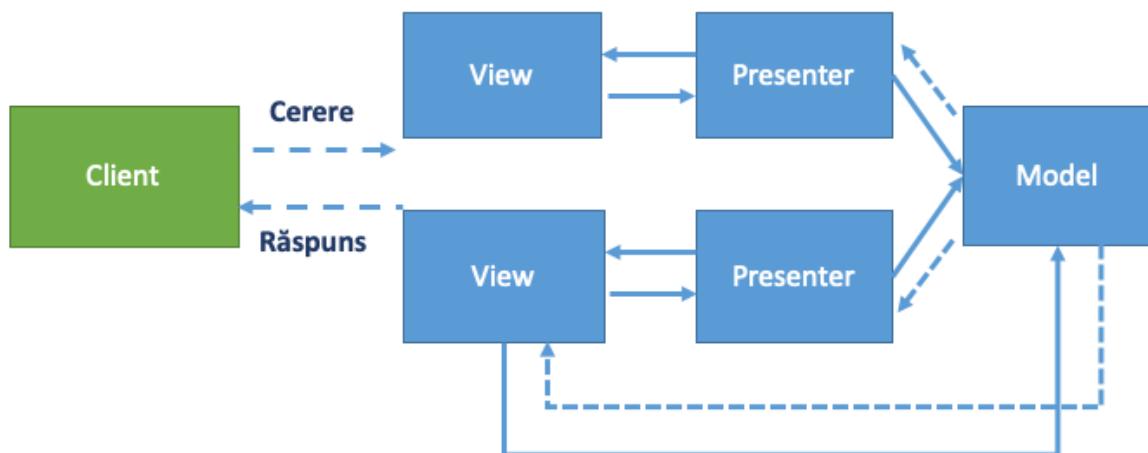


Diagrama (Controler supervisor)



Componente

- **Model**

- Nivelul datelor aplicației (acces la baze de date, preluare rețea etc.)

- **View**

- Nivelul interfață utilizator
- Afisează datele
- Notifică **Presenter**-ul cu privire la acțiunile utilizatorilor

- **Presenter**

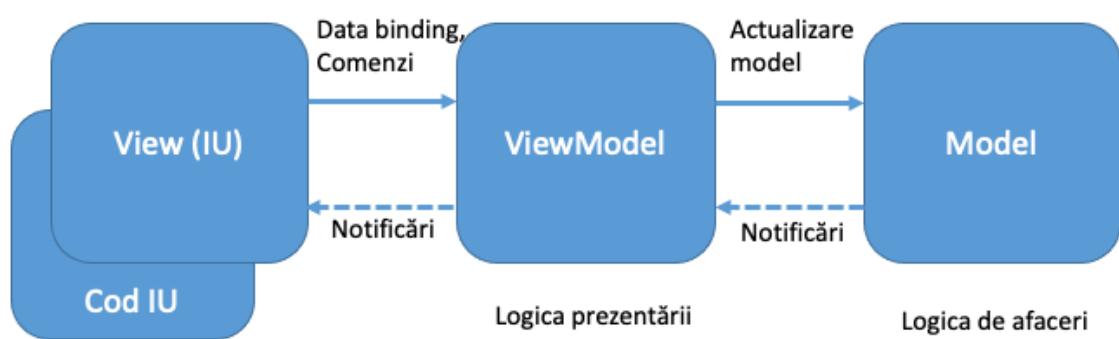
- Legătura între date și interfață utilizator
- Preia datele din model
- Aplică logica interfeței utilizator
- Gestionează starea View-ului
- Reacționează la interacțiunea utilizatorilor cu View-ul

MVP

- Față de MVC, testabilitate mai mare la nivelul componentelor
- Cuplarea mai scăzută a componentelor

Model-View-ViewModel (MVVM)

Diagrama



Componente

- **Model**

- Datele aplicației (acces la baze de date, preluare rețea etc.)

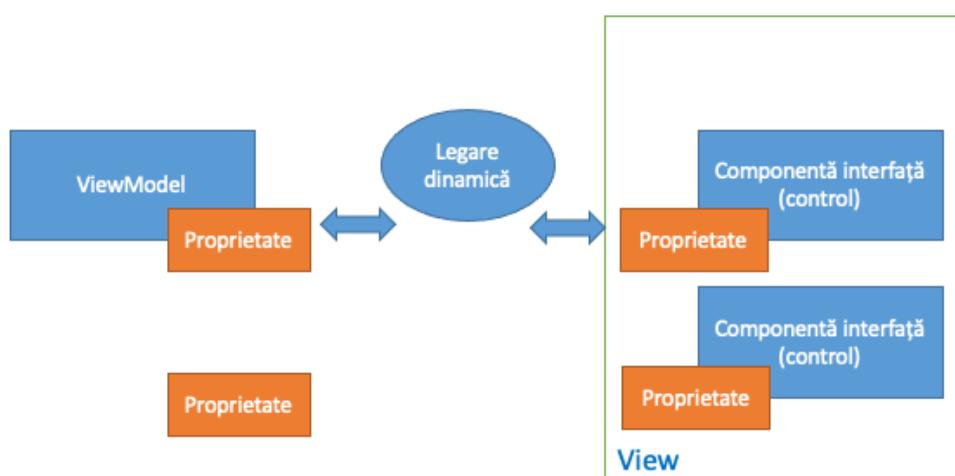
- **View**

- Interfața utilizator

- **ViewModel**

- Legătura între date și interfață
- Uzual, se utilizează legarea dinamică a datelor (*data binding*)
 - Proprietăți ViewModel
 - Proprietăți componente interfață utilizator
- Generează evenimente la modificări ale datelor
 - Sursa de date sau din interfață

Legarea dinamică a datelor



Implementări cunoscute

- Silverlight
- Windows Presentation Foundation (WPF)
- Xamarin.Forms
- Android Data Binding

Implementări cunoscute (Web)

- Knockout

Model View Template (MVT)

Model View Template

- Model arhitectural
- Interacțiunea cu interfața este distribuită în trei roluri distincte
- Framework-ul gestionează interacțiunea dintre componente
- Este aplicată separarea rolurilor

Componente

- **Model**

- Încapsulează datele și accesul la acestea
- Notifică *View-ul* cu privire la modificările apărute

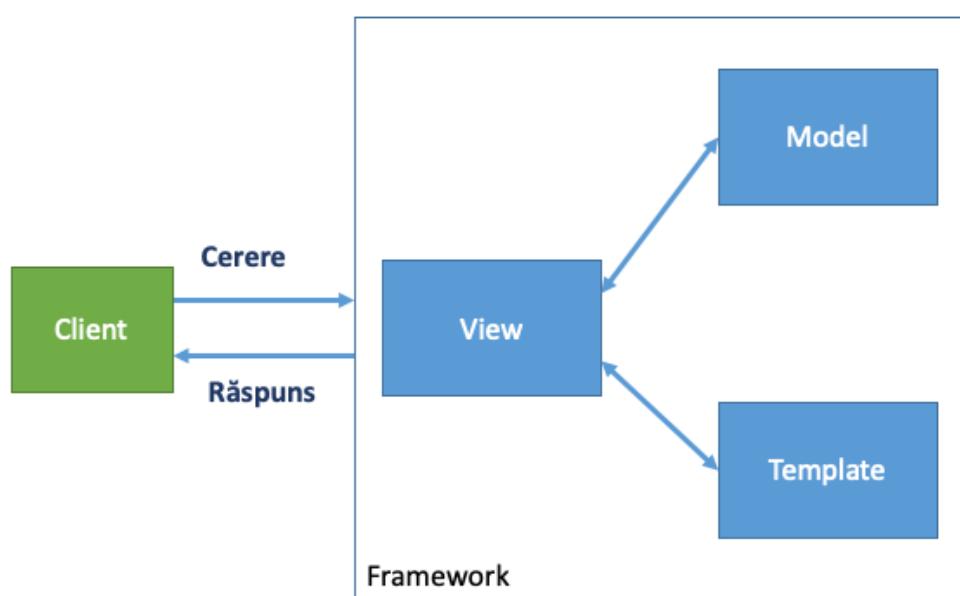
- **View**

- Logica aplicației
- Interacționează cu modelul și

- **Template**

- Nivelul de prezentare
- Gestionează interacțiunea cu interfața aplicației

Diagrama



Implementări cunoscute

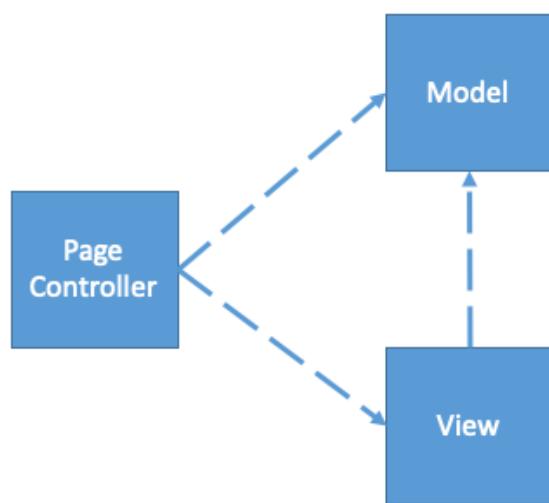
- Django

Page Controller

Page Controller

- Obiect desemnat să trateze o cerere pentru o anumită pagină sau acțiune într-un site Web
- Pentru fiecare pagină există un controller
- Controlerul poate fi implementat
 - script pe partea de server
 - pagină dinamică
- Utilizare pentru aplicații Web de complexitate medie

Diagrama



Componente

- **Page Controller**
 - Gestionează metodele HTTP GET și POST
 - Decide modelul și pagina care vor fi utilizate
- **Model**
 - Implementează logica domeniului
- **View**
 - Afisează conținutul HTML

Front Controller

Front Controller

- Acționează ca punct unic de intrare al unei aplicații Web
- Redirecționează cererile către diferite controlere de pagini
- Acționează în două etape
 - Gestiona cererilor
 - Gestiona comenziilor
- Variantă modificată a modelului de proiectare Mediator

Diagrama

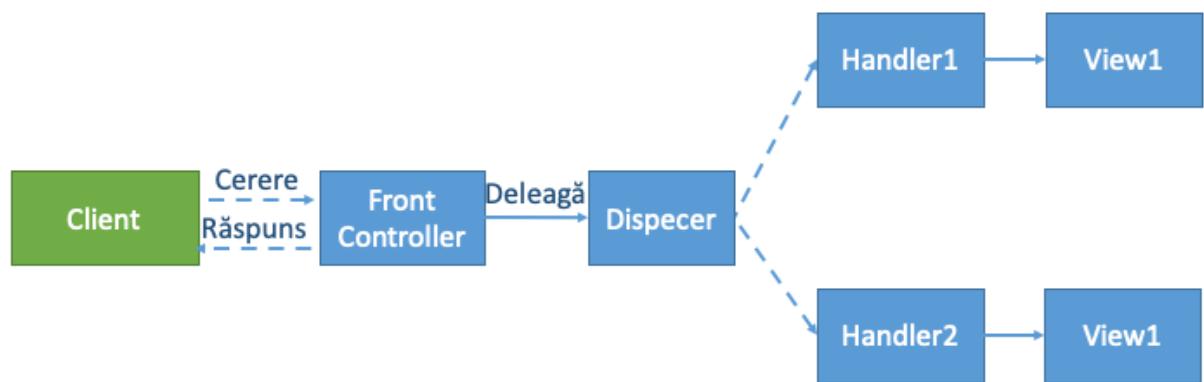
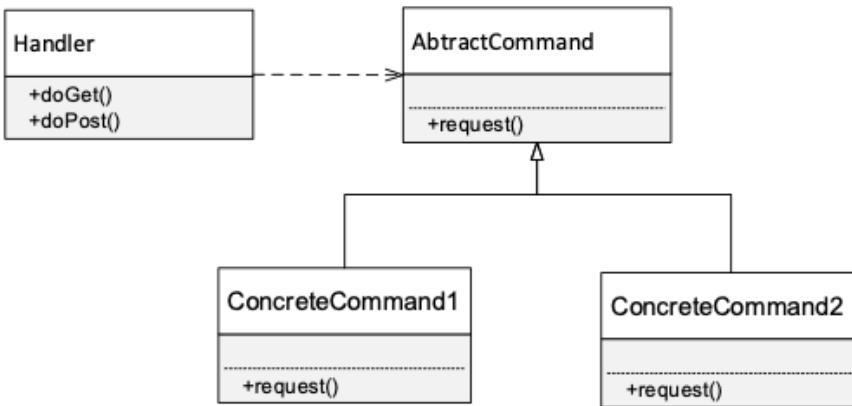


Diagrama de clase



Componente

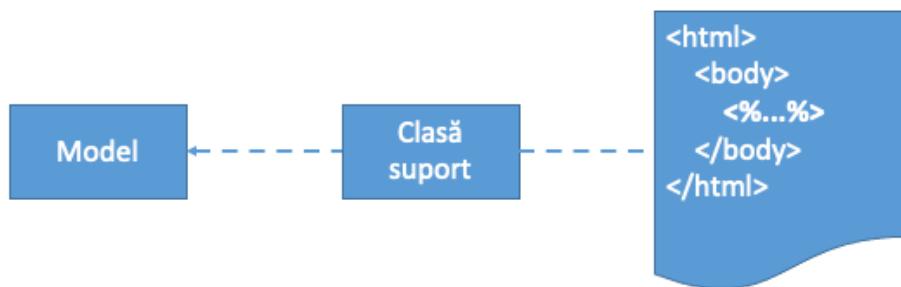
- **Handler**
 - Recepționează comenzile GET și POST
 - Analizează URL-ul și decide, static sau dinamic, care va fi comanda invocată
- **AbstractCommand**
 - Declară interfața pentru comenziile invocate de către Handler
- **ConcreteCommand1, ConcreteCommand2**
 - Comenzi concrete, implementate pe baza interfeței AbstractCommand

Template View

Template View

- Generează conținut HTML prin intermediul marcajelor specifice
- Suport pentru execuția condiționată și repetitivă
- Utilizare
 - ASP/ASP.NET
 - JSP
 - PHP
- Uzual, se utilizează împreună cu Page Controller în aceeași pagină
- Marcajele sunt interpretate de o componentă dedicată
 - este generat conținutul HTML

Diagrama



Componente

- Pagini care includ marcaje
 - Marcajele sunt asociate datelor dinamice
 - Datele dinamice sunt recepționate la invocarea paginii
- Obiecte suport
 - Preiau și furnizează valorile specifice paginilor
 - Încapsulează codul în cadrul claselor

Intercepting Filter

Probleme

- Clientul a fost autentificat?
- Clientul are o sesiune validă?
- Adresa IP a clientului este dintr-o rețea de încredere?
- Calea cererii încalcă vreo constrângere?
- Ce codificare folosește clientul pentru a trimite datele?
- Este suportat navigatorul Web?
- Trebuie decomprimat/decodificat conținutul?

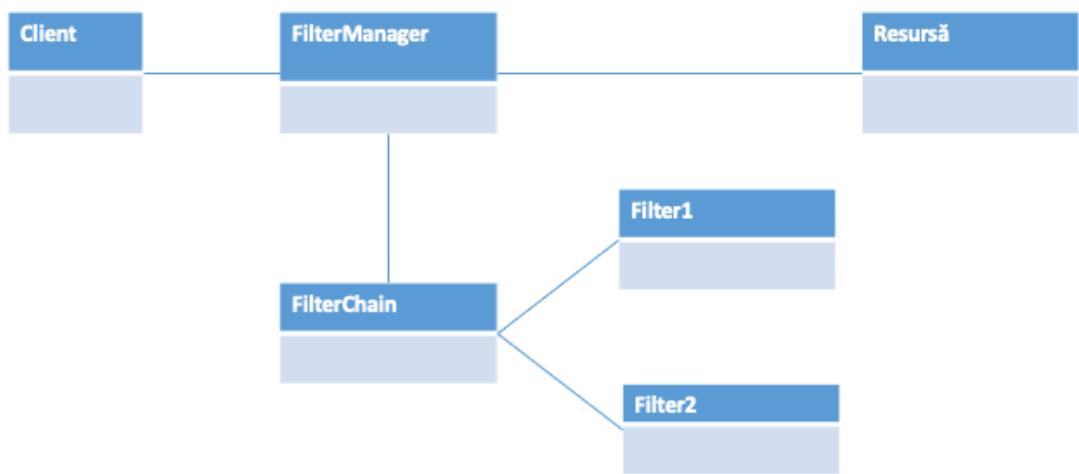
Intercepting Filter

- Anumite cereri necesită o serie de prelucrări în vederea utilizării ulterioare a acestora
 - Modificări
 - Verificări
 - Decompresie
- Operații
 - Pre-prelucrare
 - Post-prelucrare
- Se aplică cererilor și răspunsurilor

Implementare

- Definirea de filtre bazate pe o interfață comună
- Filtele se activează automat la
 - Apariția unei cereri
 - Furnizarea unui răspuns

Diagrama de clase



Componențe

- **FilterManager**
 - Gestionează prelucrările filtrelor
 - Creează **FilterChain** cu filtrele corespunzătoare, în ordinea stabilită
 - Inițiază prelucrările
- **FilterChain**
 - Colecție ordonată de filtre independente
- **Filter1, Filter2**
 - Filtre asociate **Resursei**
 - Coordonate de **FilterChain**
- **Resursa**
 - Resursa solicitată de client

Transform View

Transform View

- Procesează un element al domeniului (model) și îl transformă în format HTML
- Intrare: date
- Ieșire: HTML

Diagrama



Componente

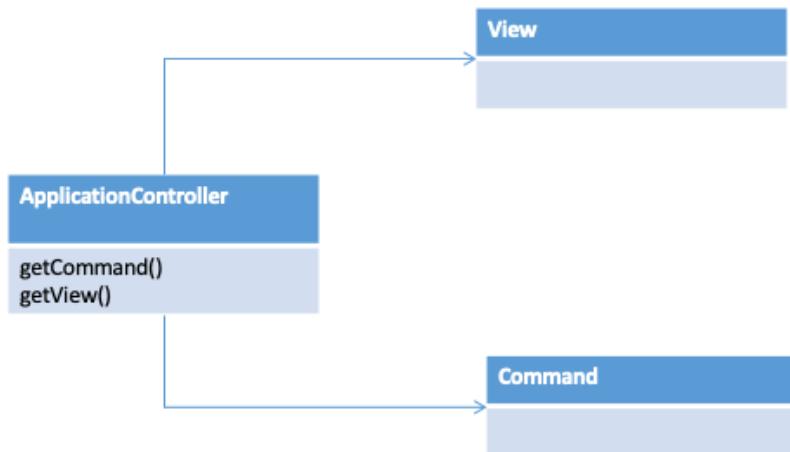
- Obiectele modelului care trebuie afișate într-o pagină Web
- Componenta care efectuează transformarea
- Pagina HTML asociată rezultatului

Application Controller

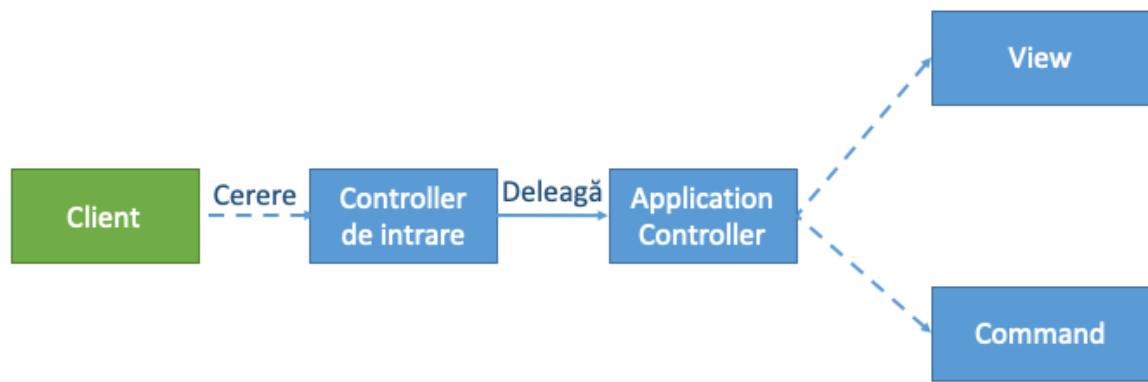
Application Controller

- Componentă centralizată pentru gestiunea:
 - Navigației în cadrul interfeței
 - Fluxului aplicației
- Degrevează controllerul de intrare de anumite activități
- Responsabilități:
 - Decide ce comandă se execută
 - Decide ce view se afișează

Diagrama de clase



Diagrama

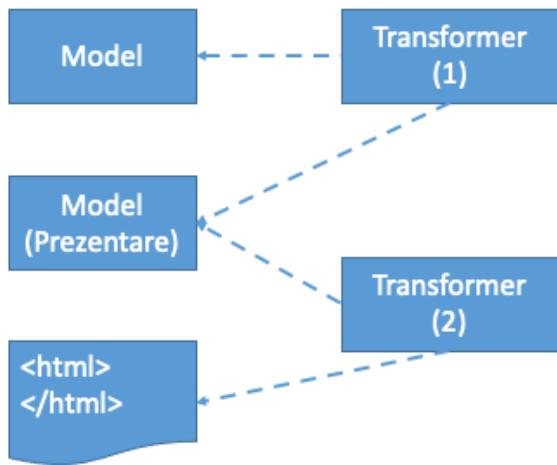


Two Step View

Two Step View

- Procesează un element al domeniului (model) și îl transformă în format HTML în două etape
- Transformarea este împărțită în două etape:
 - Datele modelului -> Logica prezentării (fără formatare)
 - Logica prezentării (fără formatare) -> HTML
- Suport pentru diferite platforme

Diagrama



Componente

- Obiect asociat modelului care trebuie afișate într-o pagină Web
- Componenta care efectuează prima transformare
- Rezultat 1
 - Obiect de interfață, fără formatare
- Componenta care efectuează transformarea finală
- Rezultat final
 - Pagina HTML asociată obiectului care trebuie reprezentat

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/>
- R. Fadatare, *Core J2EE Patterns: Best Practices and Design Strategies*, Prentice Hall, 2013
- 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
- M. Richards, Software Architecture Patterns, O'Reilly, 2015
- 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. Shrivets, *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

Modele de proiectare a aplicațiilor de întreprindere

Cursul 10 - 14

Sumar

- Alte modele
 - Blackboard
 - Dependency Injection
 - Value Object
- Arhitecturi pentru aplicații de întreprindere
- Modele specifice nivelului de persistență a datelor (1)

Blackboard

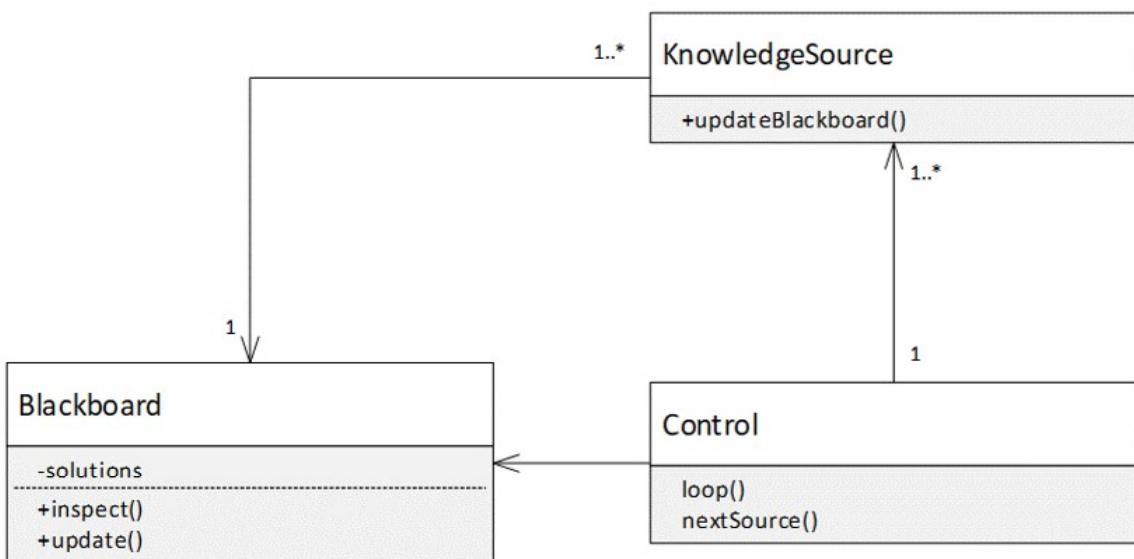
Problema

- 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

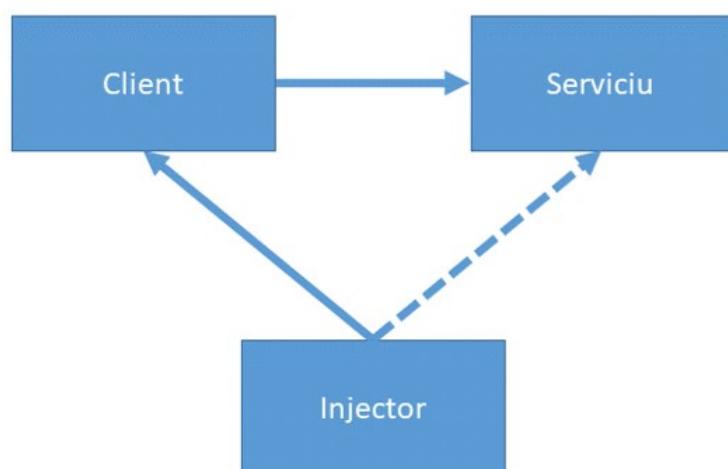
- **Blackboard**
 - Include obiectele din spațiul soluțiilor
- **KnowledgeSource**
 - Module specializate cu reprezentări specifice
- **Control**
 - Responsabil cu selectarea, configurarea și execuția modulelor

Dependency Injection

Scop

- Asigurarea unei dependențe reduse între obiecte
- Obiectele nu trebuie să fie responsabile pentru crearea propriilor dependențe
- Mecanism pentru crearea și transmiterea obiectelor (dependențelor) către un alt obiect
- Se bazează pe principiile
 - Dependency Inversion (SOLID)
 - Clasele trebuie să depindă de abstractizări și nu de implementări concrete
 - Inversarea controlului (IoC)
 - Inversarea controlului asupra obiectelor în scopul obținerii unei cuplări scăzute

alte abordări



Componente

- **Client**
 - Clasa dependentă de **Serviciu**
- **Serviciu**
 - Furnizează servicii **Clientului**
- **Injector**
 - Creează un **Serviciu** și îl transmise clasei **Client**

Implementare

- Modalități diferite de transmitere a obiectelor
- Opțiuni
 - Constructor
 - Proprietăți (set)
 - Interfețe
 - Metode de tip set

Implementare

```
class Serviciu {  
    ...  
}  
  
class Client {  
    Serviciu serviciu;  
  
    Client() {  
        serviciu = new Serviciu();  
    }  
    ...  
}  
  
class Client {  
    Serviciu serviciu;  
  
    Client(Serviciu serviciu) {  
        this.serviciu = serviciu;  
    }  
    ...  
}  
  
class Injector {  
    void inj() {  
        Serviciu s = new Serviciu();  
        Client c = new Client(s);  
        ...  
    }  
}
```

Avantaje

- Suport pentru testare
- Cuplare scăzută a componentelor
- Extinderea cu ușurință a aplicațiilor
- Scăderea complexității codului

Dezavantaje

- Identificarea dinamică a tipurilor (la execuție)
- Complexitatea învățării
- Apariția excepțiilor (la execuție)
 - de la compilare

Implementări existente

- Autofac
- Dagger
- Guice
- LightInject
- Ninject
- Spring
- Unity

Value Object

Value Object

- Obiecte simple, a căror egalitate nu se bazează pe identitate
- Similar tipurilor primitive simple
- Valori monetare, date, constante enumerative etc.
- Uzual, se transmit prin valoare și nu prin referință
- Uzual, aceste obiecte sunt imutabile

Modele arhitecturale (cont.)

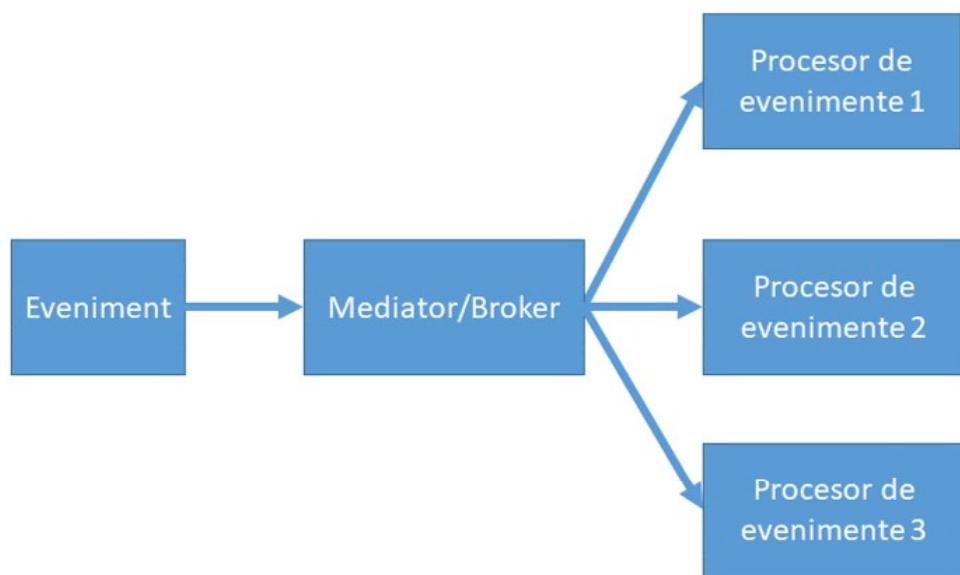
Modele arhitecturale (cont.)

- Arhitectură stratificată
- Arhitectură condusă de evenimente
- Arhitectură de tip micro-kernel
- Arhitectură bazată de microservicii
- Arhitectură bazată pe cloud (space-based)

Arhitectură stratificată



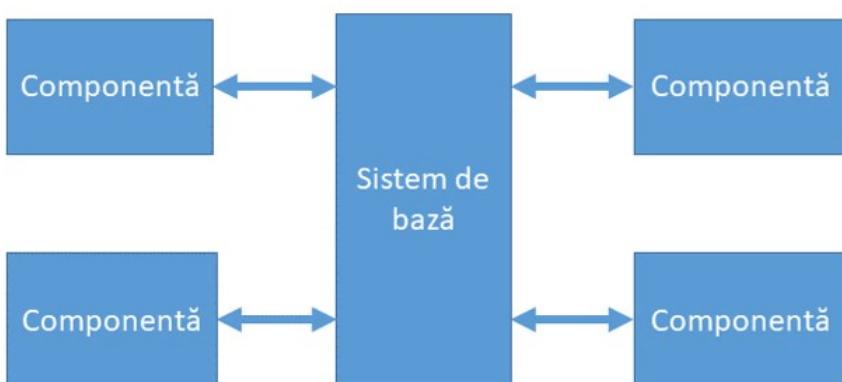
Arhitectură condusă de evenimente



Componente

- **Eveniment**
 - Inițial
 - De procesare
- **Mediator**
 - Evenimentele sunt gestionate de o singură entitate
 - Util în situațiile în care este necesară gestionarea secvenței de prelucrare a evenimentului
- **Broker**
 - Mesajele sunt distribuite într-o manieră înlănțuită
 - Procesări simple, care nu necesită un control global
- **Canale**
 - Permit transmiterea mesajelor asincrone către procesor
 - Uzual, cozi de mesaje
- **Procesor de evenimente**
 - Include logica aplicației (business)
 - Componente decuplate
 - Prelucrază evenimentele

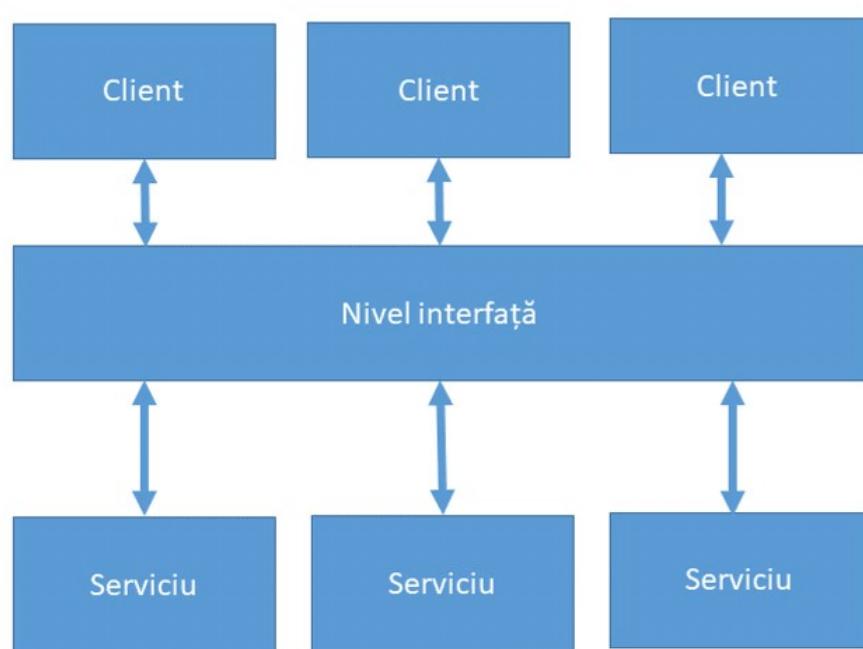
Arhitectură de tip micro-kernel



Componente

- Sistem de bază
 - Include logica de bază a aplicației
 - Pun la dispoziție un mecanism de conectare și identificare a componentelor adiționale
- Componete
 - Module de sine-stătătoare, independente
 - Includ prelucrări specializate
 - Extind funcționalitățile sistemului de bază

Arhitectură bazată de microservicii



Componente

- **Client**

- Inițiază cereri la nivelul aplicației

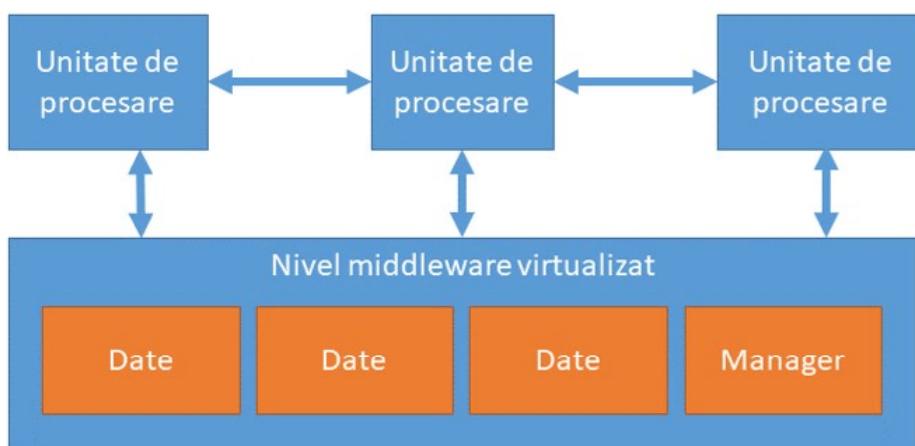
- **Nivel interfață**

- Asigură comunicarea dintre clienți și aplicație
- Interfață de programare a aplicațiilor (API)
- Aplicație (interfață utilizator)
- Broker de mesaje

- **Servicii**

- Componente dedicate, independente
- Nivel de granularitate diferit
 - o singură acțiune
 - implementări parțiale ale logicii aplicației

Arhitectură bazată pe cloud (space-based)



Componente

- **Unități de procesare**
 - Componente ale aplicației
 - Granularitate diferită
 - Aplicații de complexitate redusă
 - Componente care implementează diferite funcționalități ale aplicației
 - Includ
 - modulele aplicatiei
 - date în memorie asociate
 - suport pentru persistență/replicarea datelor
- **Nivel middleware virtualizat**
 - Asigură comunicarea între componente
 - Suport pentru sincronizare și prelucrarea cererilor
 - Includ componente pentru
 - Mesagerie (gestiunea cererilor și a sesiunilor)
 - Date (asigură replicarea datelor la nivelul aplicației)
 - Procesare (prelucrarea și coordonarea cererilor)
 - Managementul unităților de proiectare (initializare, terminare, monitorizare etc.)

Criterii de comparație

- Testabilitatea
- Scalabilitatea
- Implementarea
- Performanțe
- Agilitatea
- Instalarea

Modele pentru aplicații de întreprindere

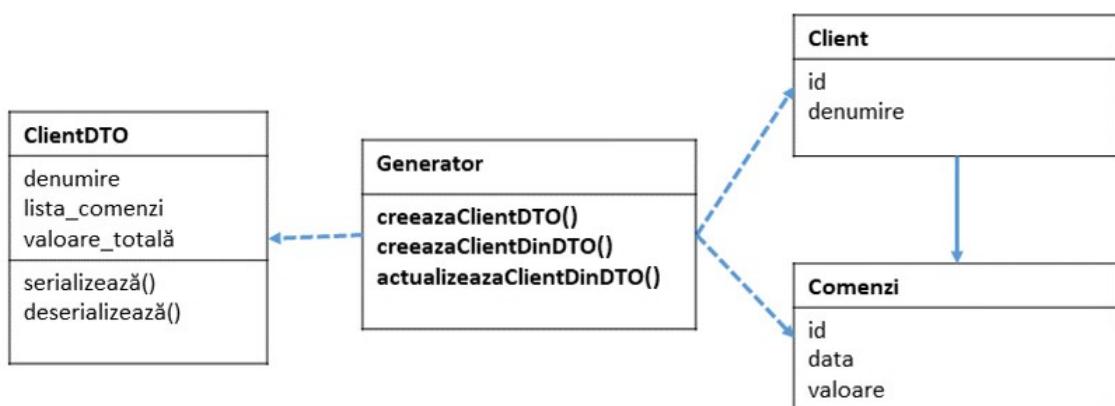
- Transferul obiectelor
 - Data Transfer Object
- Specifică sursei de date
 - Active Record
 - Data Mapper
 - Table Data Gateway
 - Row Data Gateway
 - Data Access Object
- Asociere metadate Obiect/Sursă de date
 - Query Object
 - Repository
 - Object Relational Mapping
- Obiect/Sursă de date
 - Unit of Work
 - Identity Map
 - Lazy Load

Data Transfer Object

Data Transfer Object

- Transferul datelor între diferite subsisteme/niveluri ale aplicației
- Reducerea numărului de parametri ai metodelor
- Returnarea mai multor valori dintr-o metodă
- Reducerea numărului de apeluri/cereri
- Datele sănt încapsulate într-un obiect
- Uzual, obiectele includ doar date, fără logică
- Obiecte serializabile

Diagrama de clase



Componente

- ClientDTO
 - Obiectul de tip DTO obținut pe baza obiectelor din model Client și Comanda
- Generator
 - Generează obiect de tip DTO pe baza modelului
 - Creează obiecte din model pe baza DTO
 - Bazat pe modelul Mapper
- Client, Comanda
 - Obiecte ale modelului

Discuții

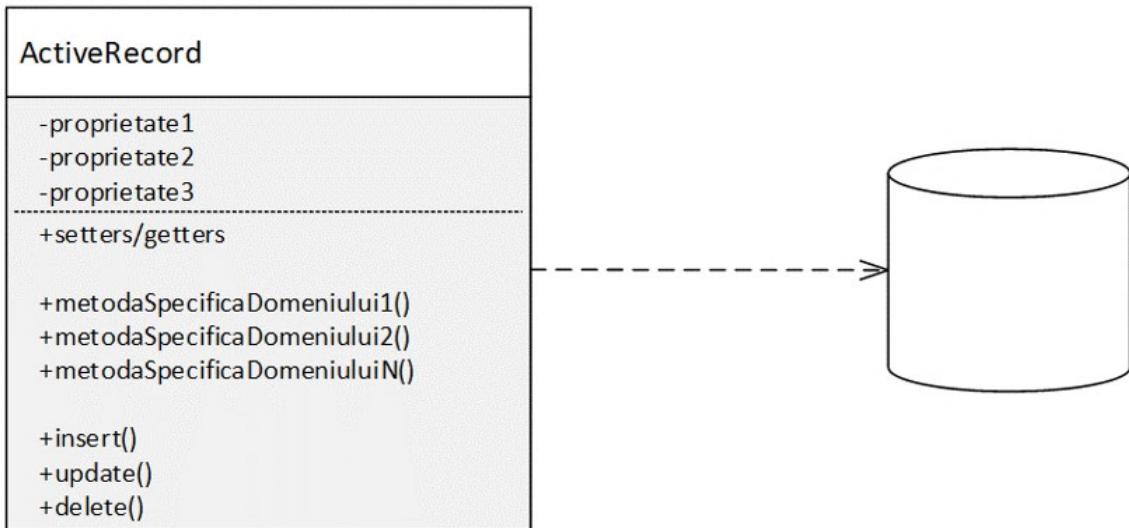
- Formă simplificată a obiectelor domeniului
- Tipuri simple de date
- Date agregate din mai multe surse
- Un caz particular este Result Set

Active Record

Active Record

- Obiect asociat unei înregistrări dintr-o tabelă/view a unei baze de date
- Încapsulează accesul la baza de date
- Include și logica domeniului pentru datele reprezentate

Diagrama de clase



Discuții

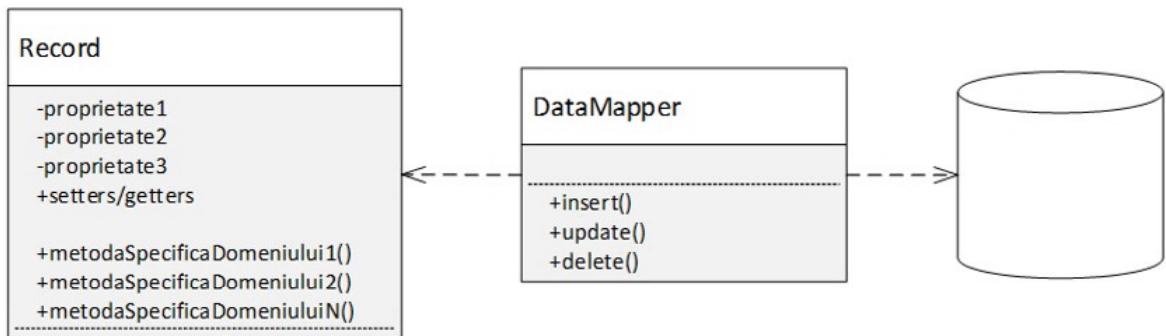
- Suport pentru aplicații de complexitate mică/medie
- Util pentru aplicații de tip CRUD
- Încalcă principiul SRP (Single Responsibility Principle)
- Nivelul asociat logicii domeniului este strîns legat de nivelul de persistență

Data Mapper

Data Mapper

- *Nivel intermediar* care separă obiectele în memorie de baza de date
- Izolează cele două niveluri (aplicație și persistență) între ele și asigură transferul de date între acestea
- Cele două niveluri (aplicație și persistență) sunt independente între acestea, dar și față de obiectul de tip Data Mapper
- Decuplează clasele modelului de nivelul de persistență
- Permite transferul obiectelor între aplicație și baza de date

Diagrama de clase



Discuții

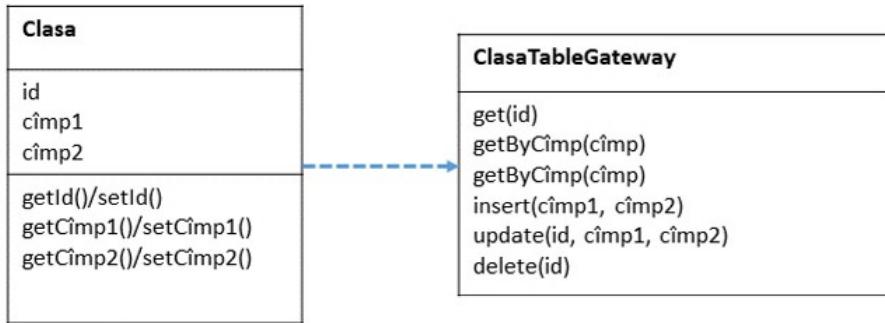
- Este respectat Priniciul SRP (Single Responsibility Principle)
- Nivelul asociat logicii domeniului nu este legat de nivelul de persistență
- În forma de bază, include referințe la biblioteci specifice nivelului de persistență

Table Data Gateway

Table Data Gateway

- Obiect care gestionează accesul la o tabelă dintr-o bază de date
- Pune la dispoziție metode pentru:
 - Adăugare
 - Regăsire
 - Modificare
 - Ștergere
- Metodele operează cu parametrii asociați membrilor clasei
- O singură instanță gestionează toate înregistrările dintr-o tabelă

Diagrama de clase



Discuții

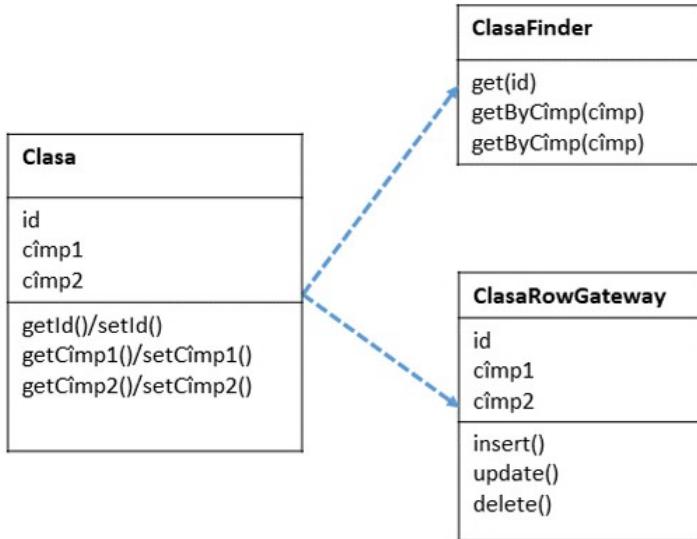
- La nivelul TDG nu există legătură cu entitățile din domeniu
- Include comenzi SQL necesare operațiilor la nivelul tabelei

Row Data Gateway

Row Data Gateway

- Obiect care gestionează accesul la o înregistrare dintr-o tabelă
- O singură instanță pe înregistrare
- Include membri asociați cîmpurilor clasei
- Pune la dispoziție metode pentru
 - Adăugare
 - Modificare
 - Ștergere

Diagrama de clase

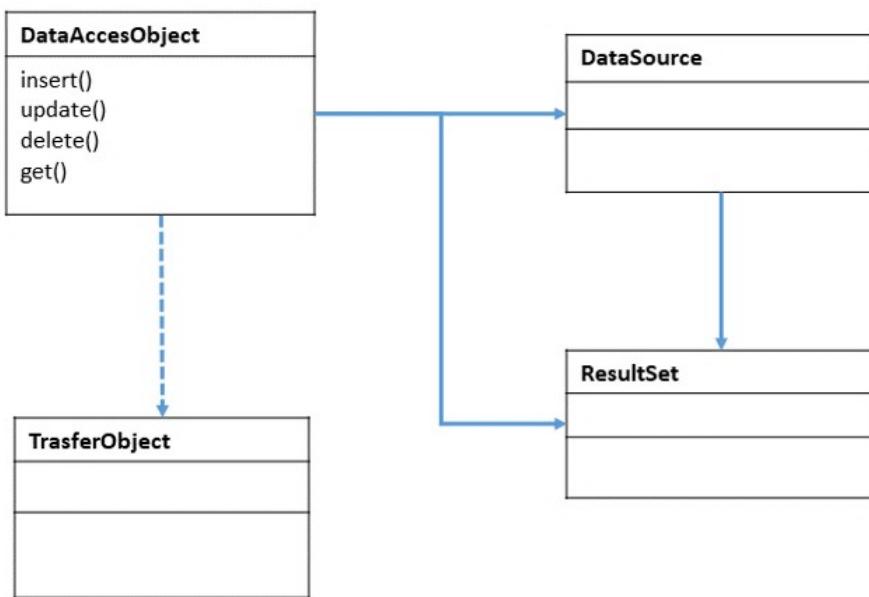


Data Access Object

Data Access Object

- Similar modelelor anterioare de acces la date
- Uzual, gestionează obiect de tipul DTO (Data Transfer Object)
- Acționează ca un adaptor între sursa de date și componente

Diagrama de clase



Componente

- **DataAccessObject**

- Este creat de către **Client**

- **TransferObject**

- Obiectul utilizat în tranzacții
- Utilizat de către **Client**

- **DataSource**

- Sursa de date
 - Bază de date
 - Fișiere
 - Servicii etc.
- Este accesată de către **DataAccessObject**, utilizând-se un **TransferObject**

- **ResultSet**

- Setul de date din memorie
- Asociat rezultatului unei interogări a sursei de date
- Pe baza acesteia se creează obiectul de tip **TransferObject**

Discuții

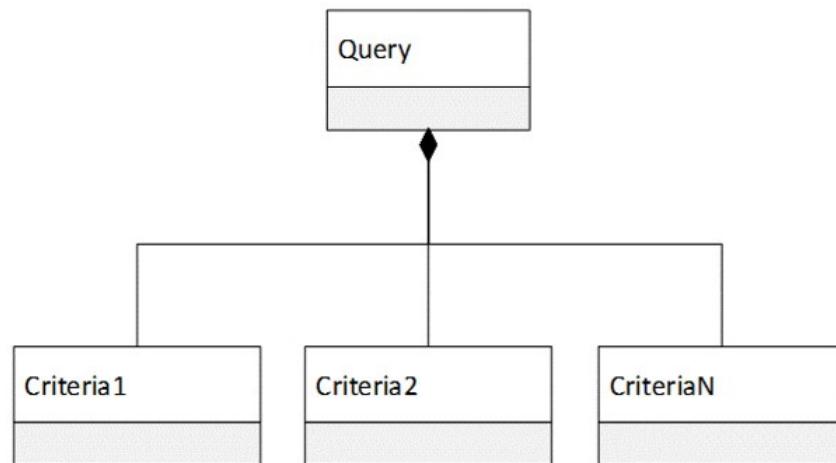
- La nivelul TDG nu există legătură cu entitățile din domeniu
- Include comenzi SQL necesare operațiilor la nivelul tabelei

Query Object

Query Object

- Obiecte asociate interogărilor dintr-o sursă de date
- Implementate în limbajul domeniului
- Interfață independentă de limbajul de interogare al bazei de date
- Uzual, se utilizează împreună cu modelul *Repository*
- Modelele de proiectare *Interpreter, Specification*
- Traducere din limbajul domeniului în limbajul de interogare al bazei de date

Diagrama



Componente

- Query
 - Clasa asociată obiectului de tip cerere
 - Generează comenzi specifice sursei de date pe baza criteriile furnizate
 - Utilizează obiecte din dimeniu
- Criteria1, Criteria2, CriteriaN
 - Criteriile specifice interogării
 - Includ cîmpuri prin intermediul cărora se pot genera criterii de selecție
 - operator, cîmp, valoare

Sumar

- Modele specifice nivelului de persistență a datelor (2)

Repository

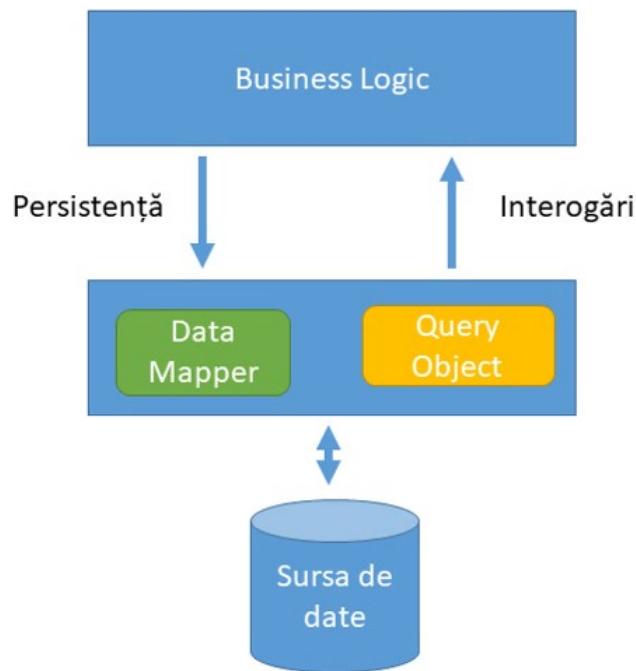
Repository

- Colecție de obiecte, în memorie, de tipul domeniului
- Nivel intermediar între domeniu și nivelul de asociere a datelor (data mapping)
- Suport pentru
 - Adăugare
 - Modificare
 - Ștergere
 - Selectie
- Suport pentru Interrogări complexe
- Suport pentru diferite surse de date

Utilizare

- Număr ridicat de obiecte ale domeniului
- Surse multiple de date
- Controlul codului pentru selecția datelor (interrogări)
- Optimizarea regăsirii datelor

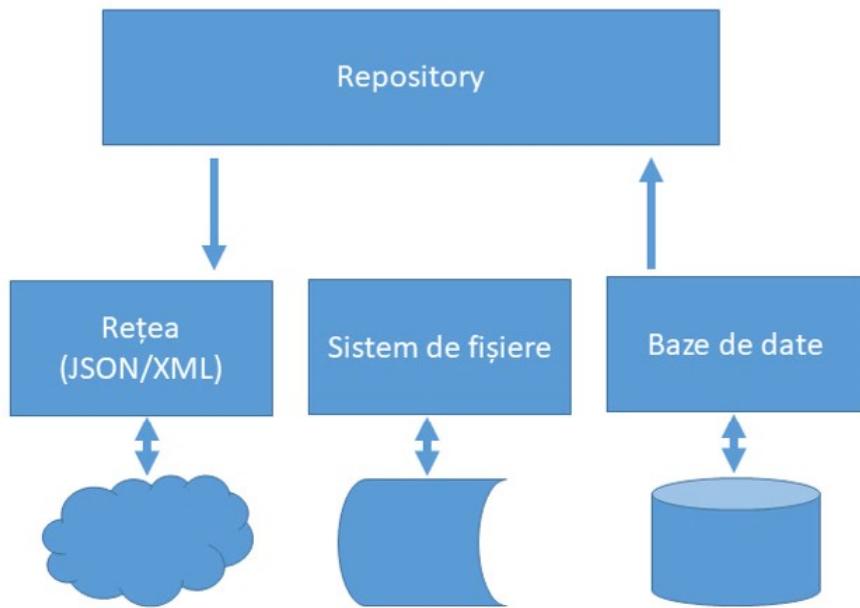
Diagrama



Implementare

- Repository per entitate
 - O clasă specializată de tip Repository per entitate
- Implementare generică
 - O singură clasă de tip Repository
 - Poate fi utilizată pentru orice tip de obiect din domeniu

alte abordări



Object Relational Mapping

ORM (Object Relational Mapping)

- Nivel intermediar dintre *baza de date relatională* și aplicație
- Include și un nivel de abstractizare dedicat interogării datelor
- Accesul la baza de date este implementat prin diferite mecanisme

Diagrama

Nivel de abstractizare a bazei
de date + limbaj de selecție

Nivel de acces la date

Implementări existente

- Django ORM
- OpenJPA
- Hibernate
- Doctrine
- Yii

Unit of Work

Unit of Work

- Gestionează o listă de obiecte cu modificările intervenite în utilizarea acestora
- Toate modificările aplicate obiectelor se vor reflecta, la un moment dat, în baza de date, printr-o singură tranzacție
- În caz de nereușită la scrierea în baza de date, se revine la starea inițială
 - Este necesară urmărirea operațiilor efectuate în baza de date
- Uzual, este utilizat împreună cu Repository

Diagrama de clase

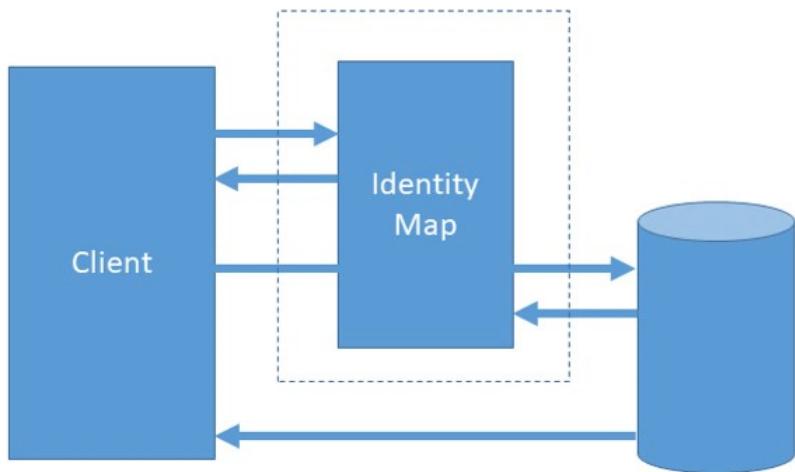
Unit of Work
registerNew(object) registerDeleted(object) registerDirty(object) registerClean(object) commit() rollback()

Identity Map

Identity Map

- Operațiile de selecție din baza de date sănătoare din punct de vedere al performanțelor
- În acest sens, se va reține fiecare obiect încărcat într-o structură ce permite regăsirea rapidă (Map)
- La solicitarea unui obiect, mai întâi se verifică în structura de date dacă a fost încărcat în prealabil
- Dacă este identificat obiectul în structură, acesta este returnat
- Dacă nu este identificat obiect, acesta se preia din baza de date și se salvează în structura de date corespunzătoare

Diagrama dinamică



Implementări

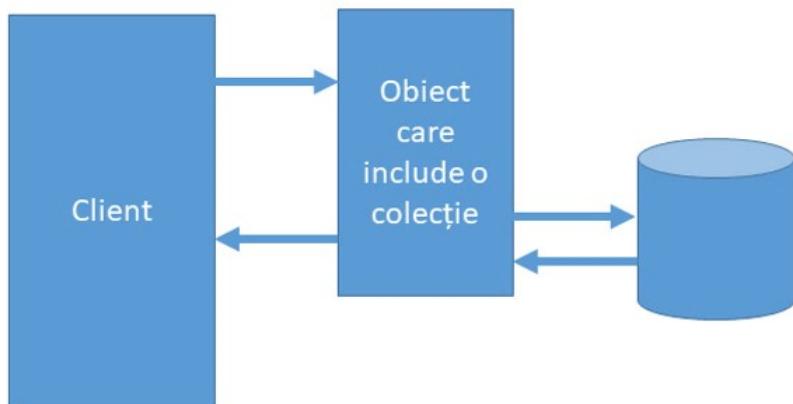
- Selectarea cheii
- Asociate unei tabele sau mai multor tabele
- Pot fi generice sau concrete
- Asociate componentelor de acces la date

Lazy Load

Lazy Load

- Operațiile de selecție din baza de date sănătătoare din punct de vedere al performanțelor
- Obiectele sănătătoare încărcate în memorie în momentul în care aceste sănătătoare sunt accesate
- Se bazează pe modelul Lazy Initialization

Diagrama dinamică



Implementare

- Un obiect care conține o colecție de obiecte asociate
- Inițial, lista nu este încărcată și este nulă
- La solicitarea listei, aceasta este inițializată prin preluarea înregistrărilor corespunzătoare din baza de date și returnată clientului
- Soluții
 - Lazy Initialization
 - Virtual Proxy

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/>
- R. Fadatare, *Core J2EE Patterns: Best Practices and Design Strategies*, Prentice Hall, 2013
- 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
- M. Richards, Software Architecture Patterns, O'Reilly, 2015
- 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. Shrivastava, *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