

Creational design patterns

ALIN ZAMFIROIU

Design Patterns

► Margaret Heafield



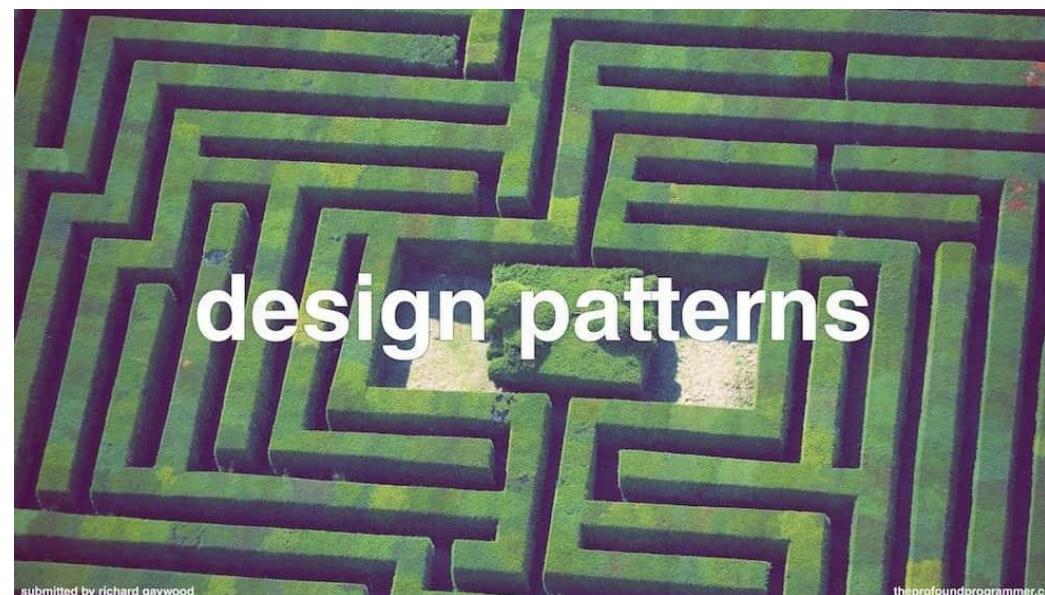
Ce sunt design patterns?

- ▶ O traducere în limba română ar fi: **șabloane de proiectare** sau **tipare de proiectare** și au scopul de a ajuta în rezolvarea unor probleme similare cu alte probleme pentru care au fost deja identificate rezolvări.



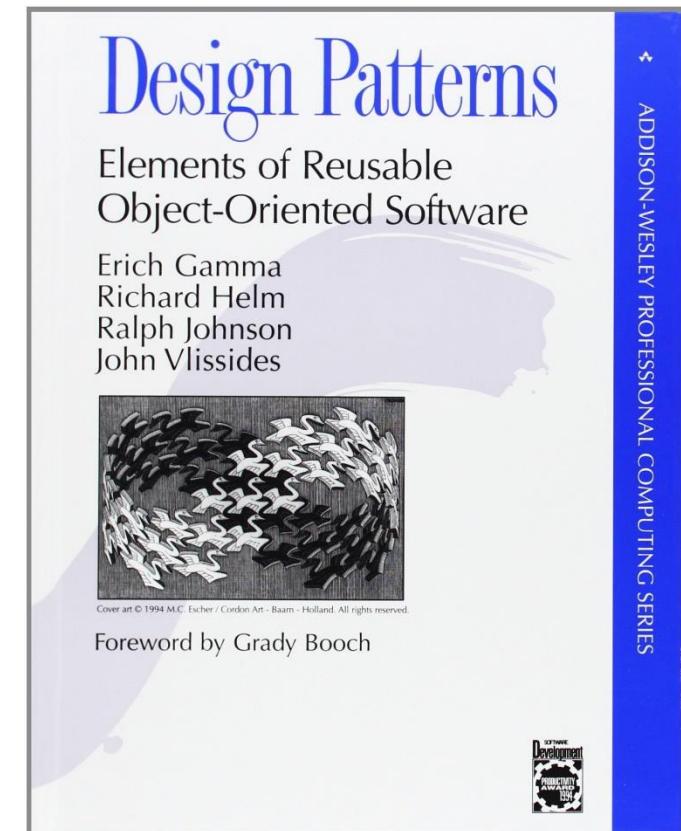
Ce sunt design patterns?

- ▶ Acestea au apărut datorită clasificării tipurilor de probleme. Prin această clasificare s-a observat că foarte multe probleme se rezolvă în același mod sau urmând aceeași serie de pași.



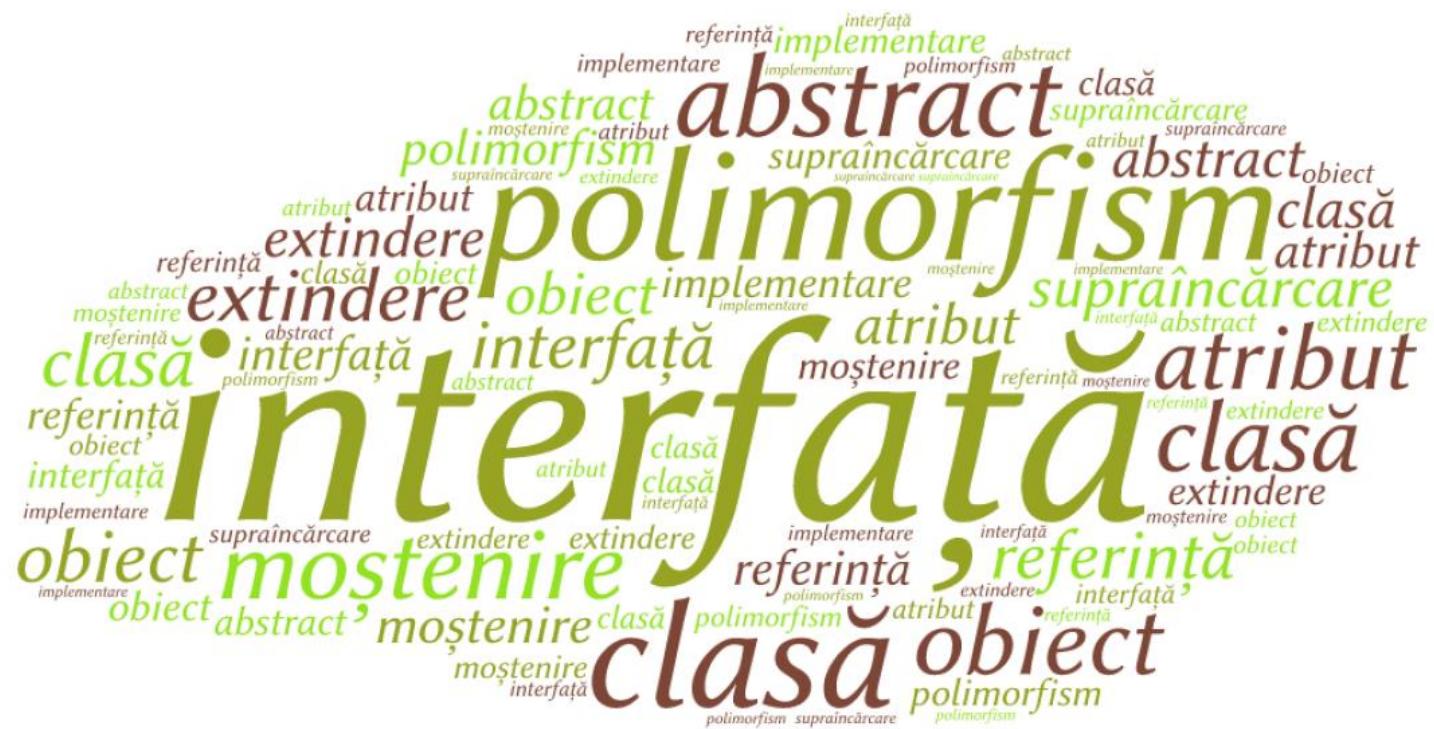
Design patterns - Gof

- ▶ Cartea reprezentativă pentru Design Patterns este **“Design Patterns: Elements of Reusable Object-Oriented Software”**, scrisă în anul 1994.
- ▶ Cartea este cunoscută și sub numele de: **“Gang of Four Book” (GoF)**.



GoF

- ▶ Pentru a citi această carte trebuie să știm programare orientată obiect.



GoF

- ▶ Cartea este împărțită în două părți:
 - ▶ Design patterns (2);
 - ▶ Catalog (3).
- ▶ Implementările prezentate sunt realizate în limbajul C++.

Calitate și testare software

Studii de caz

- ▶ Design patterns aplicate în două domenii:
 - ▶ Software pentru gestiunea activității unei firme de brokeraj de asigurări
 - ▶ Software pentru gestiunea unui cinematograf
- ▶ Implementările sunt realizate în Java.
- ▶ Subiecte propuse.



Avantaje design patterns

- ▶ sunt soluții **folosite și testate** de comunitate;
- ▶ există module deja dezvoltate, care pot fi **integrate** foarte ușor în proiectele de anvergură;
- ▶ reprezintă concepte universal cunoscute - definesc un **vocabulary comun**;
- ▶ ajută la **comunicarea** între programatori;
- ▶ permit **întelegerea mai facilă** a codului sursă/a arhitecturii;
- ▶ conduce la **evitarea rescrierii** codului sursă - refactoring.
- ▶ permit **reutilizarea soluțiilor** standard la nivel de cod sursă/arhitectură;
- ▶ permit **documentarea codului** sursa/arhitecturilor.

Dezavantaje design patterns

- ▶ Necunoașterea corectă a design pattern-ului conduce la ambiguitate;
- ▶ Din dorința de aplicare a design pattern-urilor se complică foarte mult codul sursă scris;
- ▶ “Irosirea timpului” cu etapa de analiză.

Utilizare Design Pattern

- ▶ Identificare problema;
- ▶ Mapare Design Pattern – Problemă;
- ▶ Identificare participanți;
- ▶ Alegerea numelor participanților;
- ▶ Implementarea interfețelor și claselor;
- ▶ Implementarea metodelor.

Componente design patterns

- ▶ Componentele unui design pattern sunt:
 - ▶ **nume** – care va ajuta la comunicarea între programatori, făcând parte din vocabularul acestora;
 - ▶ **problemă** – descrie contextul pentru care se folosește design pattern-ul respectiv;
 - ▶ **structură** – diagrama UML a claselor pentru realizarea design pattern-ului;
 - ▶ **participanți** – clasele ce fac parte din structura respectivului design pattern;
 - ▶ **implementare** – exemplu de cod sursă pentru o problemă rezolvată prin acel design pattern;
 - ▶ **utilizări** – folosiri concrete și practice ale design pattern-ului;
 - ▶ **corelații** – relația cu alte design pattern-uri.

Tipuri de design patterns – creaționale

- ▶ Design patternurile creaționale sunt:
 - ▶ **Singleton;**
 - ▶ **Builder;**
 - ▶ **Factory;**
 - ▶ **Factory Method;**
 - ▶ Abstract Factory;
 - ▶ **Prototype.**

Tipuri de design patterns – structurale

- ▶ Contribuie la compoziția claselor și obiectelor, realizând decuplarea interfețelor de clase:
 - ▶ **Adapter;**
 - ▶ Bridge;
 - ▶ **Composite;**
 - ▶ **Decorator;**
 - ▶ **Facade;**
 - ▶ **Flyweight;**
 - ▶ **Proxy.**

Tipuri de design patterns – comportamentale

- ▶ Permit distribuția responsabilităților pe clase și descrie interacțiunea între clase și obiecte:
 - ▶ **Chain of Responsibility;**
 - ▶ **Command;**
 - ▶ Iterator;
 - ▶ Interpreter;
 - ▶ Mediator;
 - ▶ **Memento;**
 - ▶ **Observer;**
 - ▶ **State;**
 - ▶ **Strategy;**
 - ▶ Visitor;
 - ▶ **Template.**

Tipuri de design patterns

the holy behaviors							
FM Factory Method	S Singleton	CD Command	MD Mediator	O Observer	CR Chain of Responsibility	CP Composite	D Decorator
PT Prototype							
AF Abstract Factory	TM Template Method				IN Interpreter	PX Proxy	FA Façade
BU Builder	SR Strategy	MM Memento	ST State	IT Iterator	V Visitor	FL Flyweight	BR Bridge

the holy origins

the holy structures

Tipuri de design patterns – creaționale

- ▶ Ajută la inițializarea și configurarea claselor și obiectelor;
- ▶ Design pattern-urile creaționale separă crearea obiectelor de utilizarea lor concretă. De aici și numele grupei de creaționale.

Tipuri de design patterns – creaționale

- ▶ Paternurile creaționale oferă o foarte mare flexibilitate în ceea ce privește:
 - ▶ Cine este creat;
 - ▶ Cine creaza obiectul;
 - ▶ Cum este creat;
 - ▶ Când este creat.

Singleton - nume

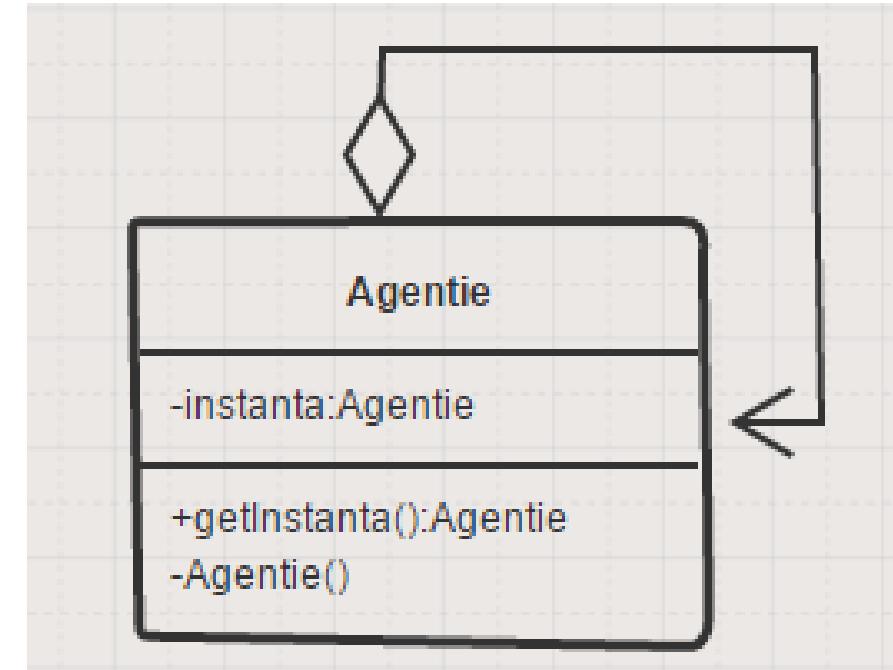
- ▶ Numele semnifică faptul că putem avea un singur obiect de tipul clasei respective: **SINGLE**.
- ▶ Clasa este cea care se ocupă de faptul că poate fi creat un singur obiect. Nu lasă acest lucru pentru utilizatori sau apelatori.
- ▶ Sintagma cheie pentru utilizarea acestui design pattern este: **instanță unică** sau **o singură instanță**.

Singleton - problemă

- ▶ Se consideră Agenția de turism: **AgeTur**;
- ▶ Să se realizeze o aplicație pentru vânzarea de pachete turistice. Aplicația trebuie să gestioneze cu foarte mare atenție vânzarea de pachete, astfel încât să nu se vândă de două ori același pachet.
- ▶ Pentru rezolvarea acestui lucru se recomandă existența unei singure instanțe a agenției.

Singleton - structură

Singleton
<u>- singleton : Singleton</u>
<u>- Singleton()</u>
<u>+ getInstance() : Singleton</u>



Singleton – participanți

- ▶ Clasa **SINGLETON - Agentie** – clasa care se ocupă de gestiunea singurei instanțe din acea clasă.
- ▶ Clasa Agentie are o instanță de tipul Agentie și are grijă să nu fie creată o altă instanță de acest tip.

Singleton - implementare

- ▶ Constructorul clasei Agentie este privat, astfel încât să nu poată fi accesat din afara clasei.
- ▶ Aceasta conține o instanță statică
- ▶ Crearea de obiecte se face prin intermediul unei metode statice, care în cazul în care instanța a fost inițializată o returnează. În sens contrar, apelează constructorul privat și inițializează această instanță, pe care o și returnează.

Singleton - tipuri

- ▶ Eager initialization
- ▶ Static block initialization
- ▶ Lazy initialization
- ▶ Thread Safe Singleton
- ▶ Inner static helper class
- ▶ Enum singleton
- ▶ Serializarea singletonului
- ▶ Singleton vs clasă statică
- ▶ Singleton Collection

Singleton – Eager Initialization

- ▶ Presupune inițializarea instanței chiar dacă aceasta nu este folosită.
- ▶ În cazul în care clasa Singleton nu este folosită, instanța tot este creată.
- ▶ De aceea, această variantă de Singleton nu este eficientă.

Singleton – Static block initialization

- ▶ Este asemănătoare cu *eager initialization* doar că această variantă furnizează posibilitatea de captare a posibilelor excepții generate de inițializarea instanței statice.
- ▶ Aceste două variante de Singleton inițializează instanța chiar dacă nu este folosită

Singleton – Lazy Initialization

- ▶ Este probabil cea mai implementată variantă de Singleton.
- ▶ Problema acestei variante apare atunci când este folosită multithreading, deoarece metoda poate fi apelată în același timp de pe două fire de execuție și astfel vor fi create două obiecte diferite.

Singleton – Thread safe Singleton

- ▶ Această variantă asigură faptul că metoda nu o să fie apelată de un alt fir de execuție până nu se termină metodă apelată deja pe un fir de execuție

Singleton – Inner static helper class

- ▶ Aceasă variantă a fost propusă de **Bill Pugh** și conține o clasă imbricată în clasa Singleton.
- ▶ Clasa Helper imbricată va fi încărcată doar când este apelată funcția de creare a instanței.
- ▶ Această variantă de Singleton îmbină *Eager initialization* cu *Lazy initialization*.

Enum singleton

- ▶ Aceasă variantă a fost propusă de **Joshua Bloch** și presupune utilizarea unei enumerări pentru crearea unică a instanței.
- ▶ Valorile enum-ului fiind accesibile la nivel global, poate fi considerat un singleton.
- ▶ Această variantă de Singleton nu permite Lazy initialization.

Singleton - Serializare

- ▶ Dacă avem serializare și apoi deserializăm o instanță singleton, se vor obține două instanțe: una creată și una deserializată.
- ▶ Pentru evitarea acestui aspect este necesară implementarea metodei **readResolve()**.
- ▶ Această metodă trebuie să returneze tot instanță creată în cadrul singletonului.

Singleton vs clasă statică

- ▶ Singleton respectă principiile de programare orientată obiect (**POO**);
- ▶ Un obiect Singleton poate fi trimis ca parametru unei funcții, în schimb o clasă statică nu poate fi transmisă ca parametru;
- ▶ O clasă Singleton poate implementa o interfață sau extinde o altă clasă;

Singleton Collection

- ▶ Cunoscut și sub numele de **Singleton registry**.
- ▶ Presupune gestiunea obiectelor unice într-o colecție.

Singleton - utilizări

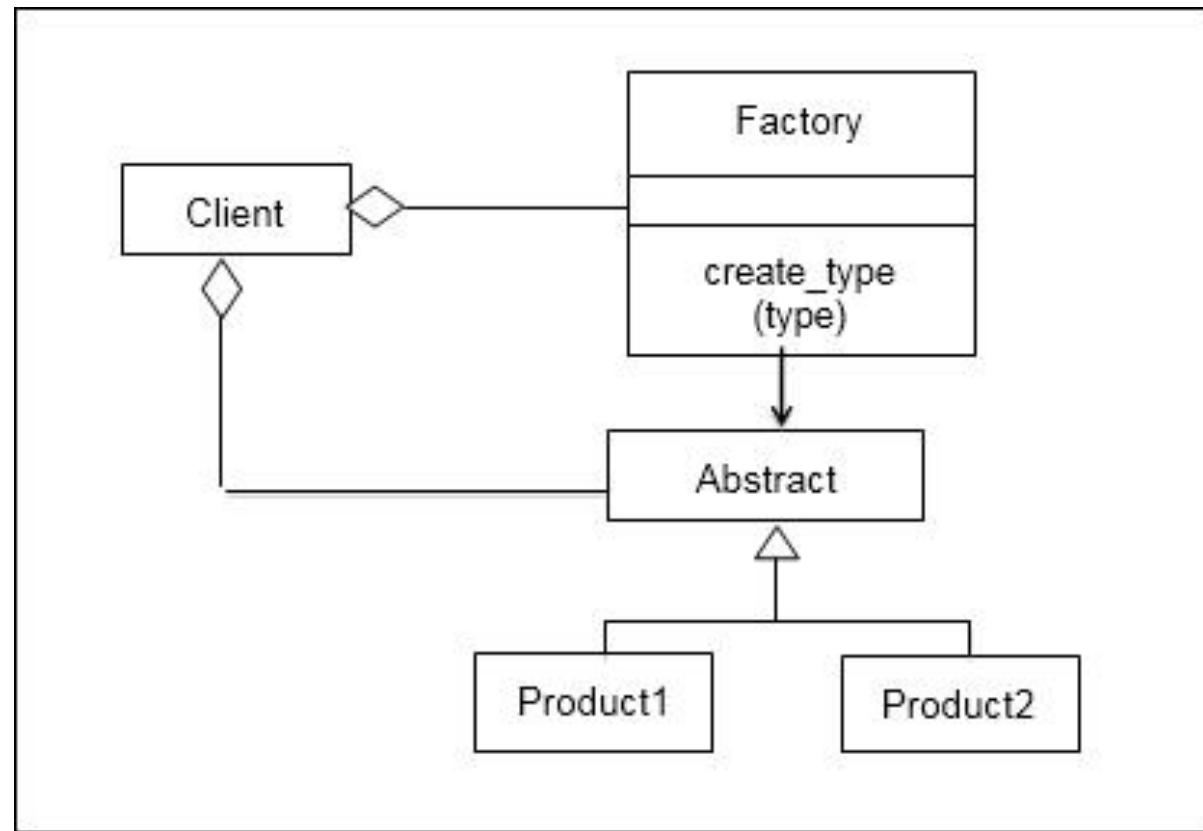
- ▶ Deschiderea unei singure instanțe ale unei aplicații.
- ▶ Conexiune unică la baza de date.
- ▶ Accesarea resurselor dispozitivelor mobile (SharedPreferences în Android).
- ▶ DocumentBuilderFactory în Android.

Singleton – corelații

- ▶ **Factory** – o singură fabrică de obiecte.
- ▶ **Builder** – un singur obiect de construit alte obiecte.

Factory - nume

- ▶ Oferă posibilitatea creării de obiecte concrete dintr-o familie de obiecte, fără să se știe exact tipul concret al obiectului.
- ▶ Sintagma după care este recunoscut este: **familie de obiecte sau obiecte din aceeași familie.**
- ▶ Simple factory este un design pattern care nu este prezentat în cartea GoF, însă este folosit în practică deoarece este foarte ușor de implementat.



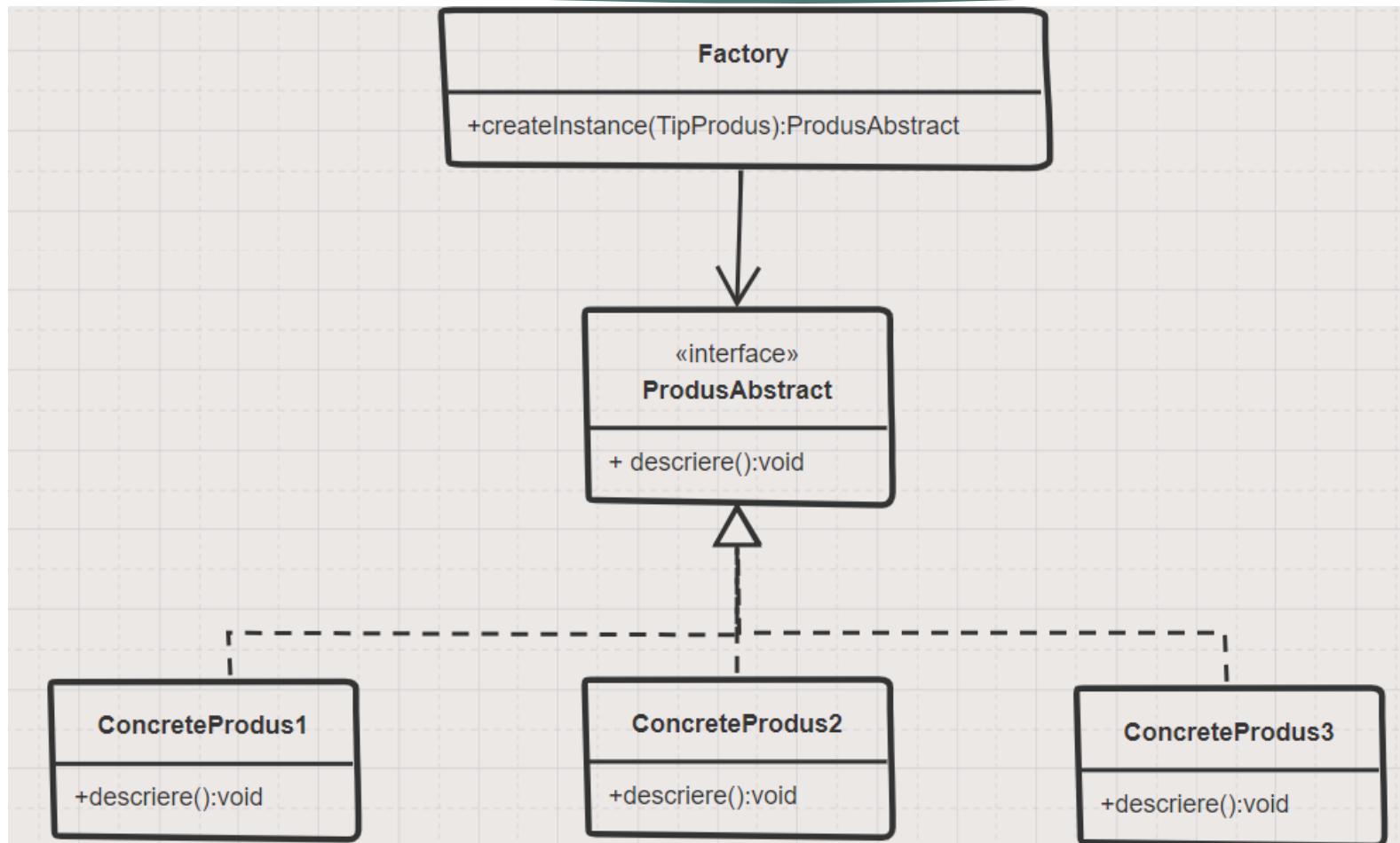
Factory – participanți

- ▶ Interfața – poate fi și clasă abstractă.
- ▶ Clasele concrete – implementează interfața sau clasa abstractă.
- ▶ Fabrica – este clasa care va crea obiectele concrete prin intermediul metodei `createInstance()`.

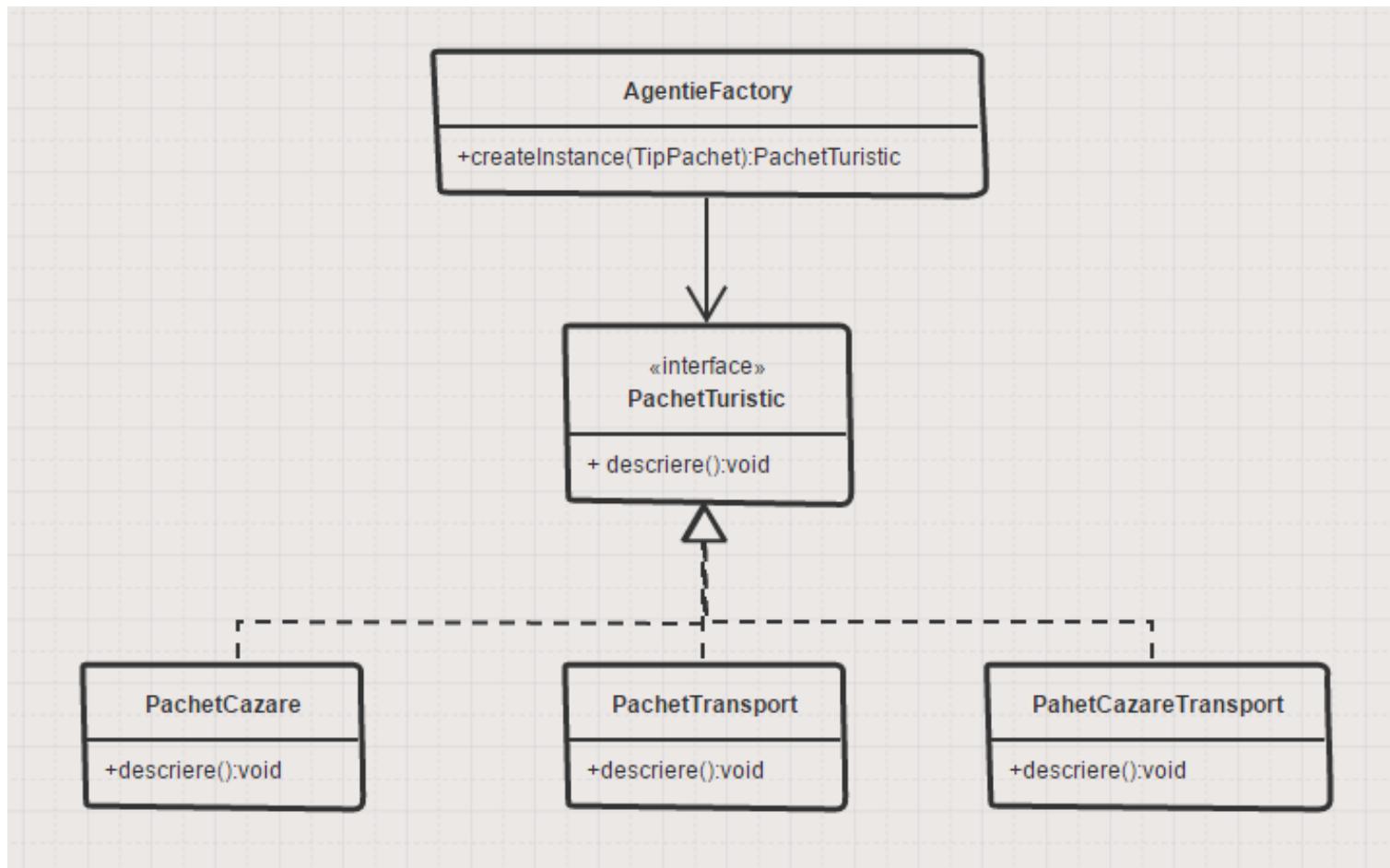
Factory - problemă

- ▶ Agentia de turism **AgeTur** ofera pachete turistice cu cazare si transport insa are in oferta si pachete turistice doar cu cazare sau doar cu transport. Toate ofertele fac parte din familia pachetelor turistice.
- ▶ Să se implementeze modulul de vânzare de pachete turistice pentru agentia **AgeTur**.

Factory - structură



Factory - structură



Factory – participanți

- ▶ Interfața **PachetTuristic** – poate fi și clasă abstractă.
- ▶ Clasele concrete **PachetCazare**, **PachetTransport**, **PachetCazareTransport** – implementează interfața PachetTuristic.
- ▶ Fabrica **AgentieFactory** – este clasa care va crea obiectele concrete prin intermediul metodei `createInstance()`.

Factory - implementare

- ▶ Metoda `createInstance()` din **AgentieFactory** primește tipul de pachet, care este un enum, și returnează tipul concret de obiect conform acestui parametru.
- ▶ Tipul returnat este abstract, însă sunt respectate principiile Liskov și Dependency Inversion din SOLID.

Factory - utilizări

- ▶ Crearea de view-uri pentru GUI.
- ▶ Existența unei familii de obiecte într-o aplicație

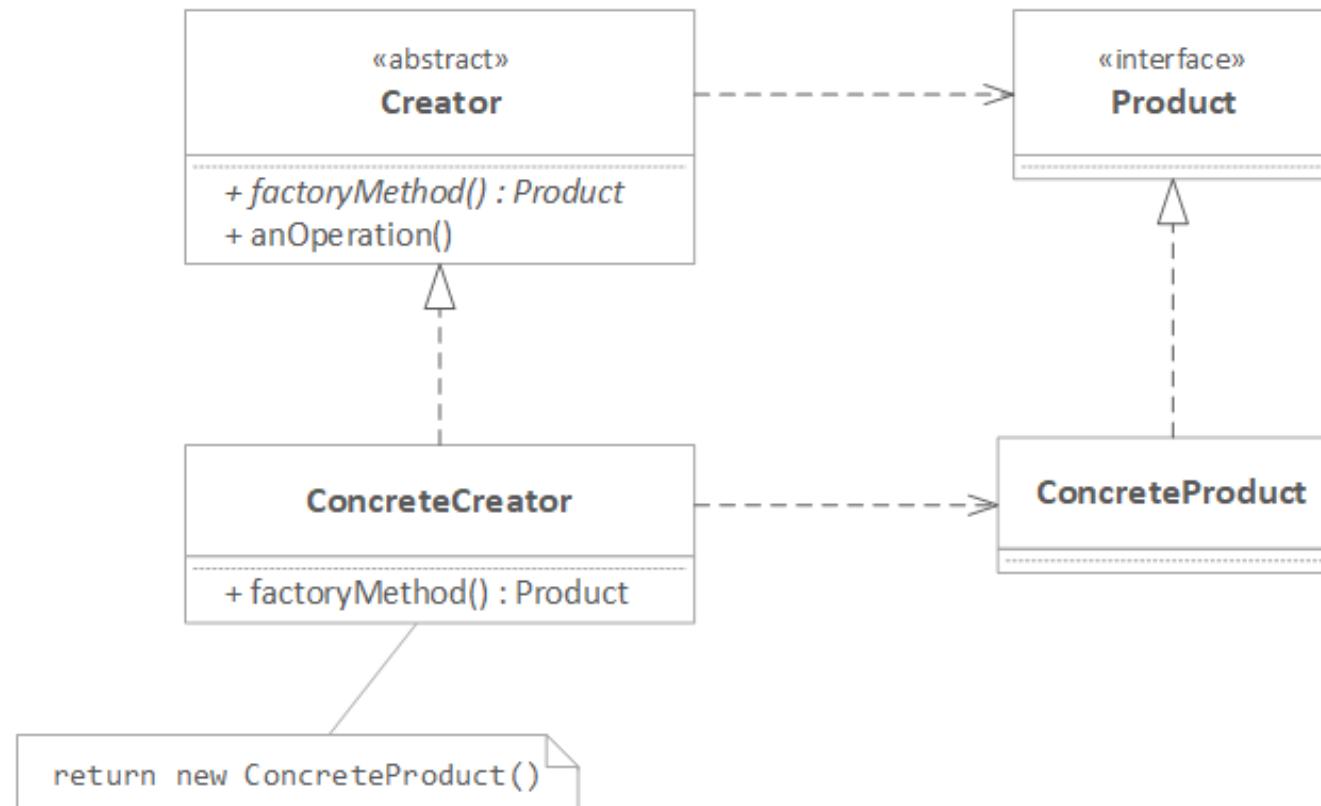
Factory – corelații

- ▶ **Singleton** – fabrica poate fi unică.
- ▶ **Composite** – prin intermediul fabricii sunt create nodurile container și nodurile frunză

Factory Method - nume

- ▶ Este asemănător cu *Simple Factory* doar că abstractizează nivelul de creare
- ▶ Se mai numește și **Virtual Constructor**.

Factory Method - Diagramă

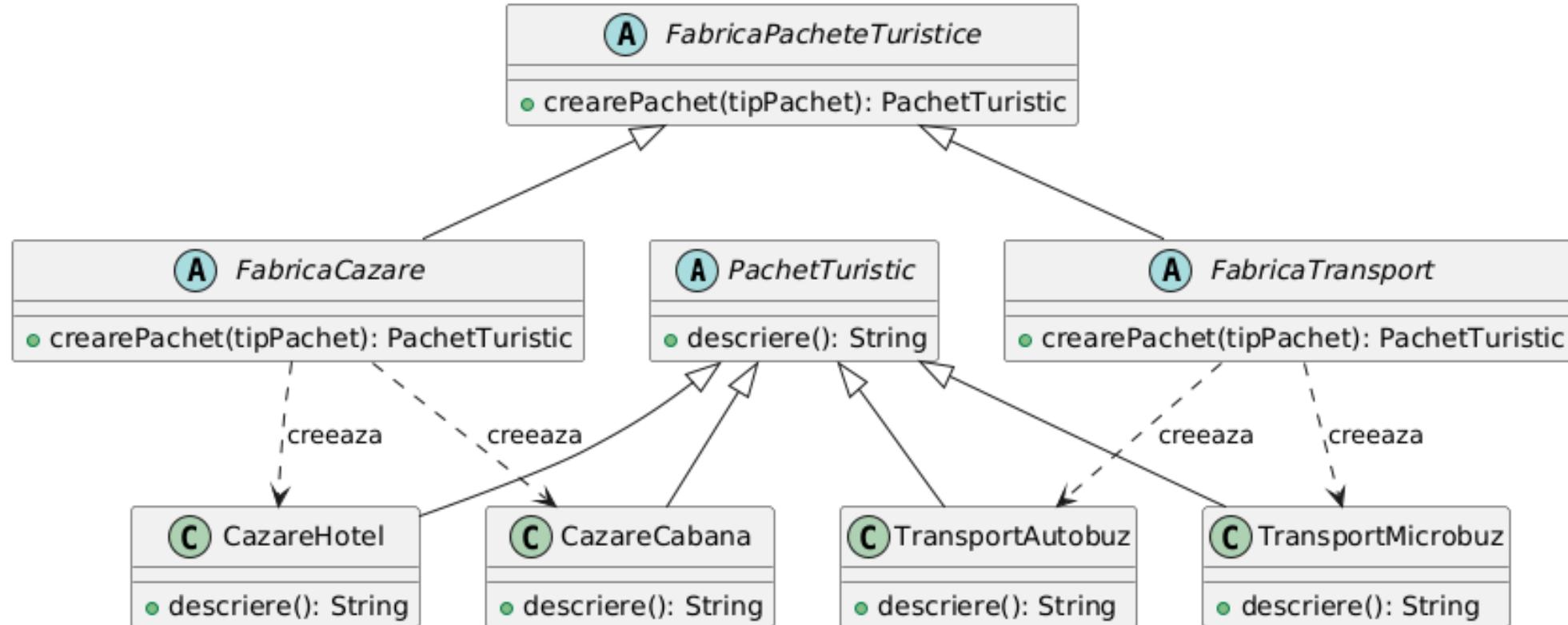


Factory Method - problemă

- ▶ Aceeași problemă ca la Simple Factory.
- ▶ Să se implementeze modulul de vânzare de pachete turistice pentru agenția **AgeTur**, știindu-se faptul că această oferă pachete din familia pachetelor turistice pentru cazare și pentru transport.

Factory Method

► Structură



Factory Method – participanți

- ▶ Interfața **PachetTuristic** – definește la nivel abstract obiectele ce pot fi create;
- ▶ Clasele concrete **CazareHotel**, **CazareCabana**, **TransportAutobuz**, **TransportMicrobuz** – implementează interfața *PachetTuristic*;

- ▶ Interfața *FabricaPacheteTuristice* – definește la nivel abstract generatoarele de obiecte
- ▶ Clasele concrete **FabricaCazare**, **FabricaTransport** – implementează interfața *Factory*, iar metoda *crearePachet(tipPachet)* va returna instanțe ale claselor concrete din familia *PachetTuristic*.

Abstract Factory - nume

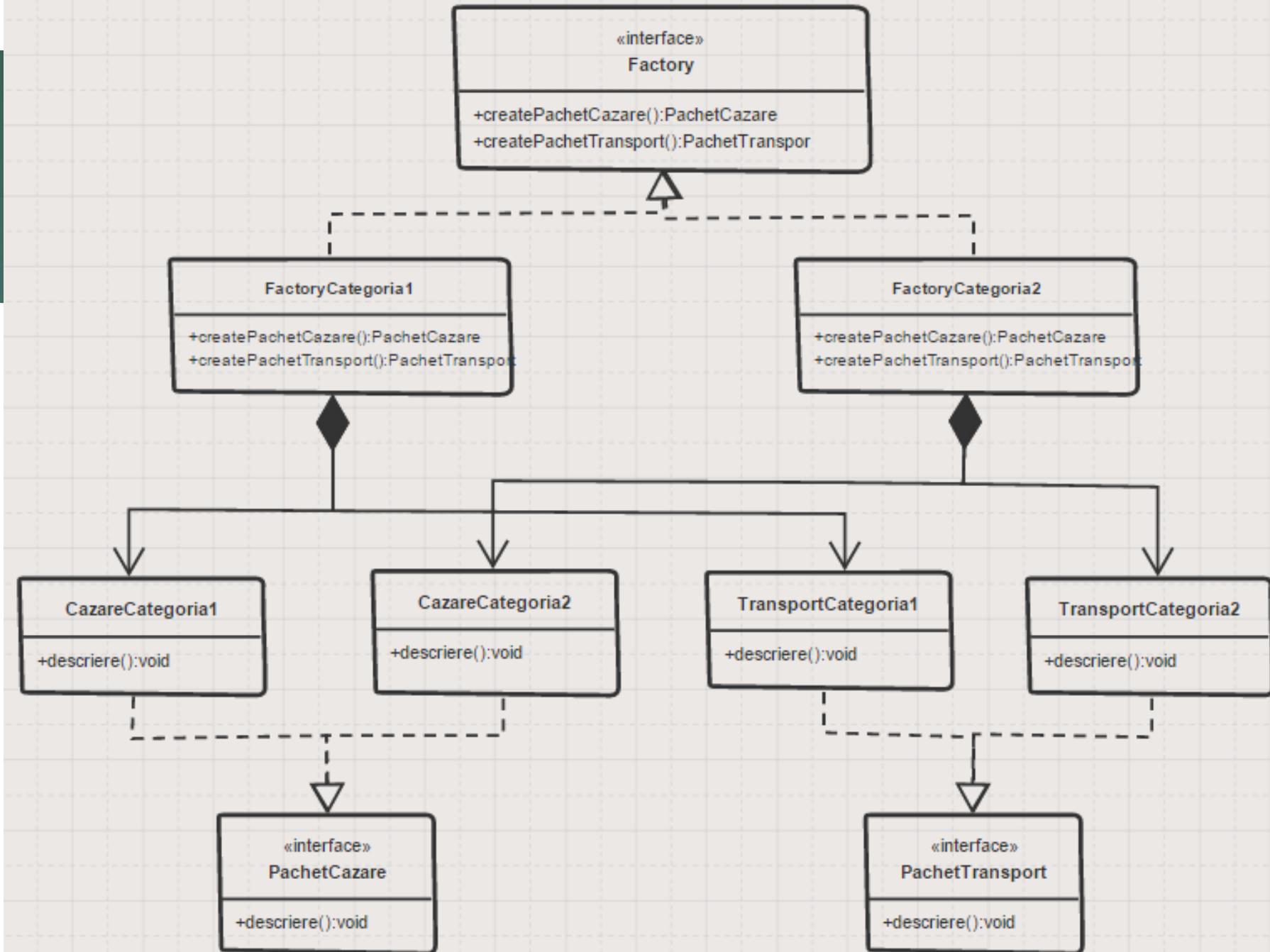
- ▶ **Abstract Factory** introduce un nou nivel de abstractizare.
- ▶ Fiecare fabrică de obiecte creează obiecte din două sau mai multe familii de obiecte

Abstract Factory - problemă

- ▶ Agentia de turism **AgeTur** ofera pachete de cazare si pachete de transport. Pentru cazare agentia are in oferta un pachet ieftin la hotel de 3 stele si un pachet scump la hotel de 5 stele. Pentru transport, agentia are in oferta pachet ieftin cu autocarul si pachet scump cu avionul.
- ▶ Sa se implementeze modulul de vanzare de pachete de cazare si transport pentru agentia **AgeTur**.

Abstract Factory

► Structură



Abstract Factory – participanți

- ▶ Abstract factory – interfața **Factory**;
- ▶ ConcreteFactory – Clasele **FactoryCategoria1** și **FactoryCategoria2**;
- ▶ AbstractProduct – Interfețele **PachetCazare** și **PachetTransport**;
- ▶ ConcreteProduct – Clasele concrete pentru Cazare și Transport pe cele două categorii.

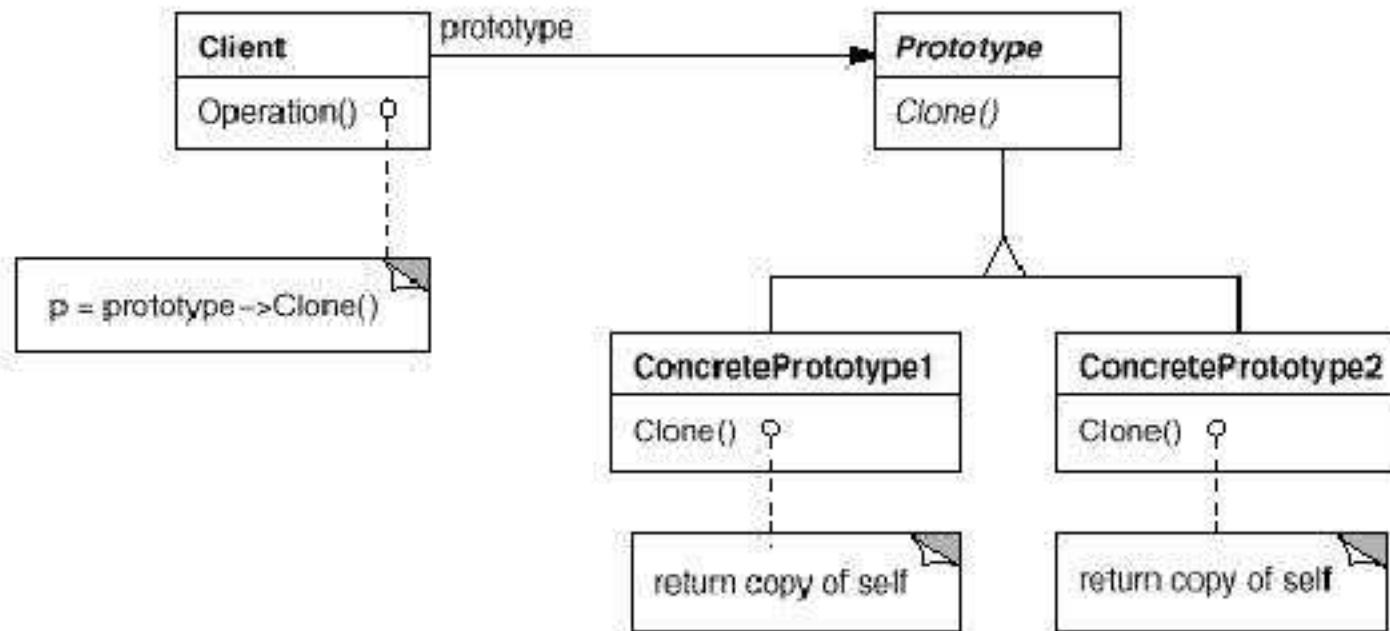
Abstract Factory - implementare

- ▶ Fiecare factory va crea două sau mai multe tipuri de obiecte. Pentru fiecare obiect există o metodă.
- ▶ Astfel avem o singură fabrică cu ajutorul căreia obținem și obiecte de tip cazare și obiecte de tip transport dintr-o anumită categorie.

Prototype - nume

- ▶ Ajută la crearea de clone pentru obiectele a căror construire durează foarte mult sau consumă foarte multe resurse.
- ▶ Prin intermediul acestui design pattern se creează un obiect considerat prototip. Acest prototip urmând a fi clonat pentru următoarele instanțe din acea clasă.

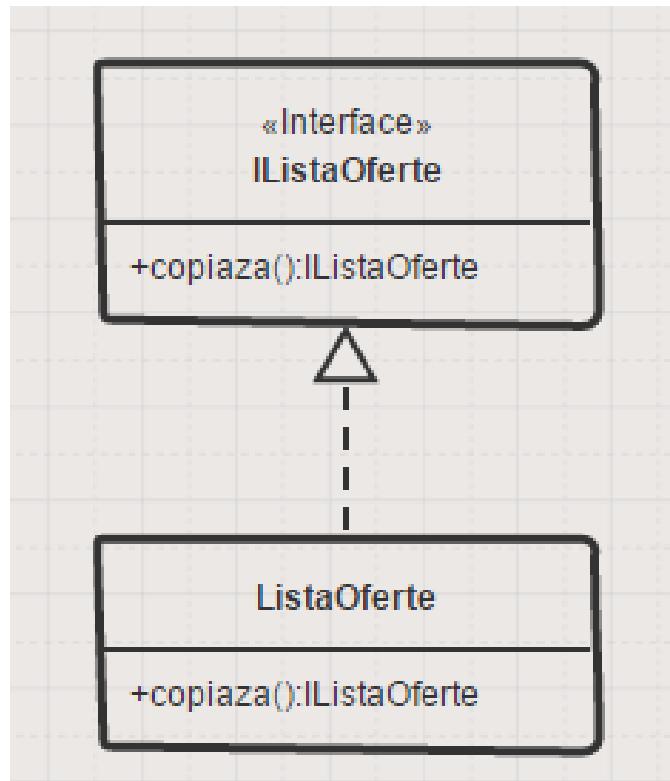
Prototype - Diagramă



Prototype - problemă

- ▶ Agenția **AgeTur** dorește să trimită lista de oferte existentă în baza de date, tuturor clienților.
- ▶ Trimiterea ofertelor tuturor clienților durează foarte mult deoarece la fiecare client este creat un obiect de tipul ListaOferte. La crearea obiectului se citesc din fișier toate ofertele.
- ▶ Să se găsească o soluție eficientă prin care listele de oferte să fie încărcate mai repede.

Prototype - structură



Prototype – participanți

- ▶ **Prototype** (IListaOferte) – interfață care anunță metoda de copiere;
- ▶ **ConcretePrototype** (ListaOferte) – implementează metoda de copiere sau de clonare.

Prototype - implementare

- ▶ Foarte mare atenție la clonarea obiectelor.
- ▶ Metoda clone din *Cloneable* nu face deep copy.
- ▶ Metoda de copiere trebuie implementată astfel încât să realizeze deep copy, și să nu mai fie nevoie de interogarea bazei de date.

Prototype - utilizări

- ▶ Atunci când obiectele create seamănă între ele, iar crearea unui obiect durează foarte mult sau consumă resurse foarte multe.
- ▶ Aplicabil ori de câte ori folosim clone()

Prototype – corelații

- ▶ **Factory** – se poate stoca o serie de obiecte și să fie clonate atunci când sunt cerute obiectele de acel tip;
- ▶ **Decorator** – se clonează obiectele și apoi se modifică;
- ▶ **Composite** – elementele de pe același nivel pot fi clonate.

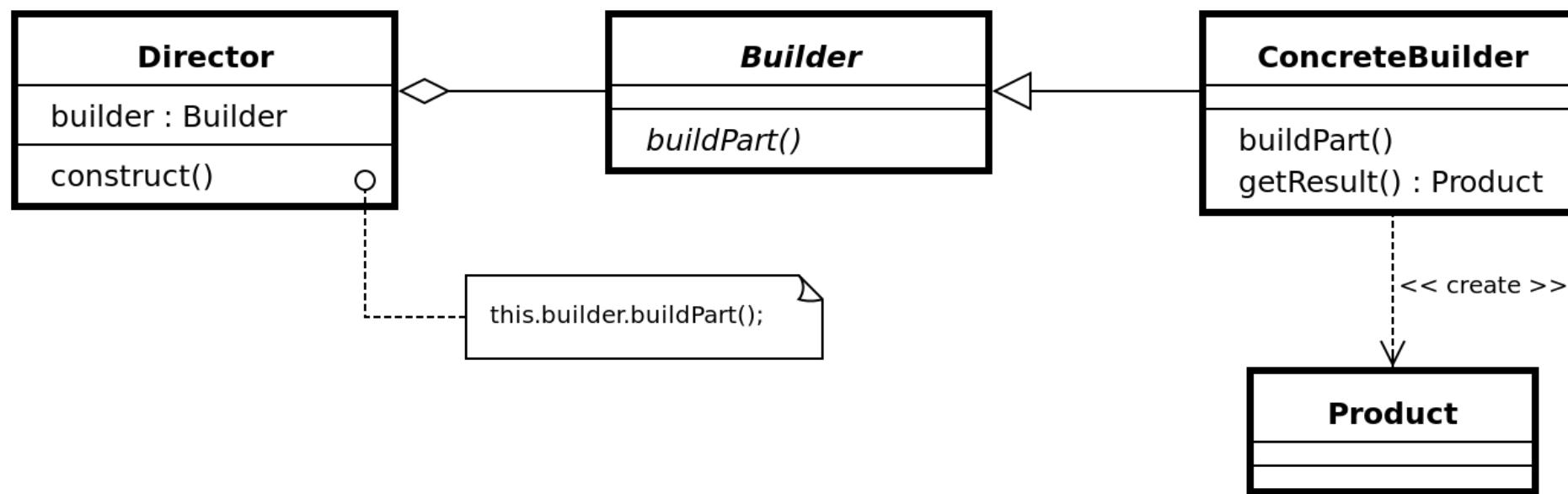
Prototype - extra

- ▶ Se dorește realizarea unui joc de curse auto. Pentru acest joc trebuie generat publicul din tribune. Publicul este reprezentat de foarte multe persoane.
- ▶ Este creată clasa Persoana. Constructorul clasei Persoana construiește un obiect de tip Persoana, care va fi desenat în tribună.
- ▶ Construirea unui obiect este costisitoare și durează foarte mult deoarece trebuie calculate raporturile dintre dimensiunile capului, gâtului și a umerilor persoanei construite.
- ▶ Trebuie să se găsească o soluție de implementare prin care persoanele să fie create mai repede și desenate în joc.

Builder - nume

- ▶ Ajută la crearea de obiecte concrete.
- ▶ Numele vine de la faptul că ajută la construirea de obiecte (build).
- ▶ Se construiesc obiecte complexe prin specificarea anumitor proprietăți dorite din multitudinea existentă.
- ▶ Obiectele create sunt finale.

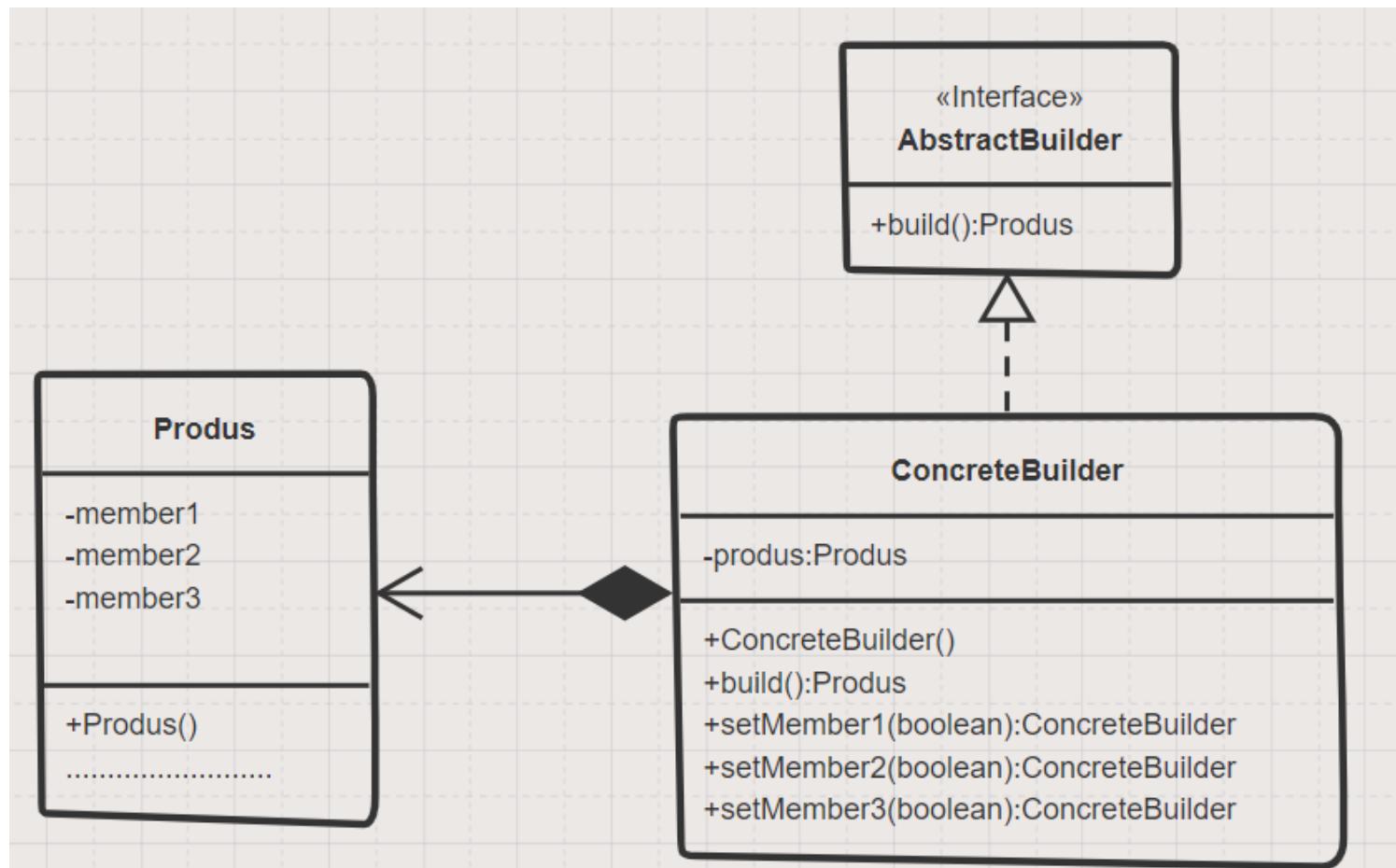
Builder - Diagrama



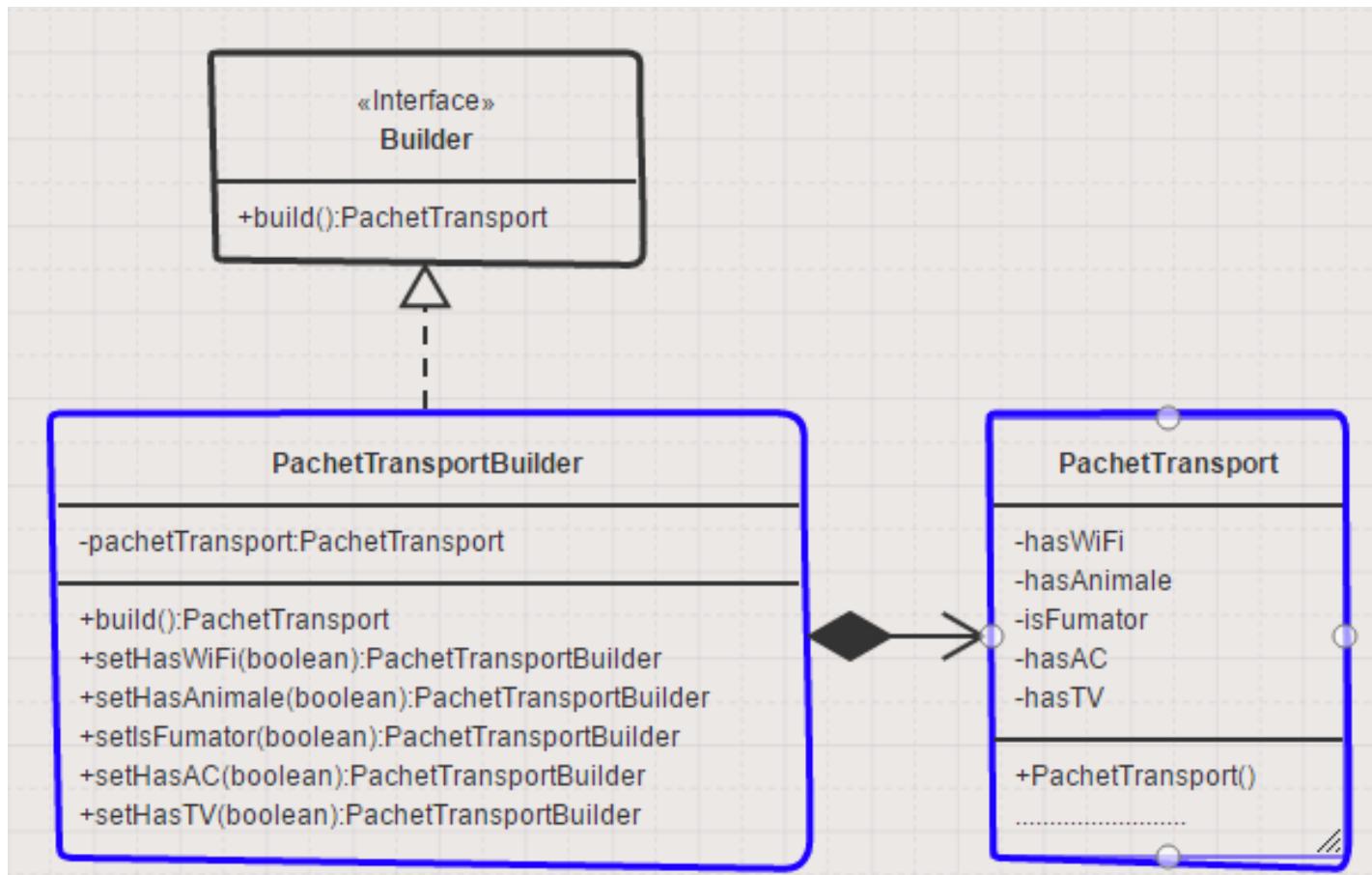
Builder - problemă

- ▶ Agenția **AgeTur** oferă pachete de transport cu facilități extra precum: WiFi, animale de companie, locuri pentru fumători, aer condiționat, televizor.
- ▶ Aceste facilități sunt adăugate doar dacă un client le dorește, nefiind incluse în pachetul de bază.
- ▶ Să se implementeze modulul de creare pachete de transport pentru agentia **AgeTur**.

Builder - structură



Builder - structură



Builder – participanți

- ▶ **AbstractBuilder** – interfață care definește metoda build pentru construirea obiectelor complexe.
- ▶ **ConcreteBuilder** – clasa concretă care implementează interfața și construiește obiectele complexe.
- ▶ **Produs** – clasa concretă a obiectelor complexe, pentru care urmează să se creeze Builderul.

Builder - implementare

- ▶ Există două variante de implementare ale acestui Design Pattern:
 - ▶ Crearea obiectului complex în constructorul clasei Builder și modificarea atributelor conform cerințelor. În această situație **Builder** (PachetTransportBuilder) are un singur atribut de tipul **Produs** (PachetTransport);
 - ▶ Crearea obiectului complex se realizează în metoda build() pe baza setărilor realizate. În această situație clasa **Builder** (PachetTransportBuilder) conține aceleași atribute ca și clasa **Produs** (PachetTransport).

Builder - implementare

- ▶ Există și o a treia variantă de implementare care presupune utilizarea unei clase imbricate (Inner class).
- ▶ Pentru această situație se folosește una dintre variantele prezentate anterior

Builder - utilizări

- ▶ În general pentru construirea de obiecte complexe cu foarte multe atribută.
- ▶ StringBuilder
- ▶ DocumentBuilder în Android.

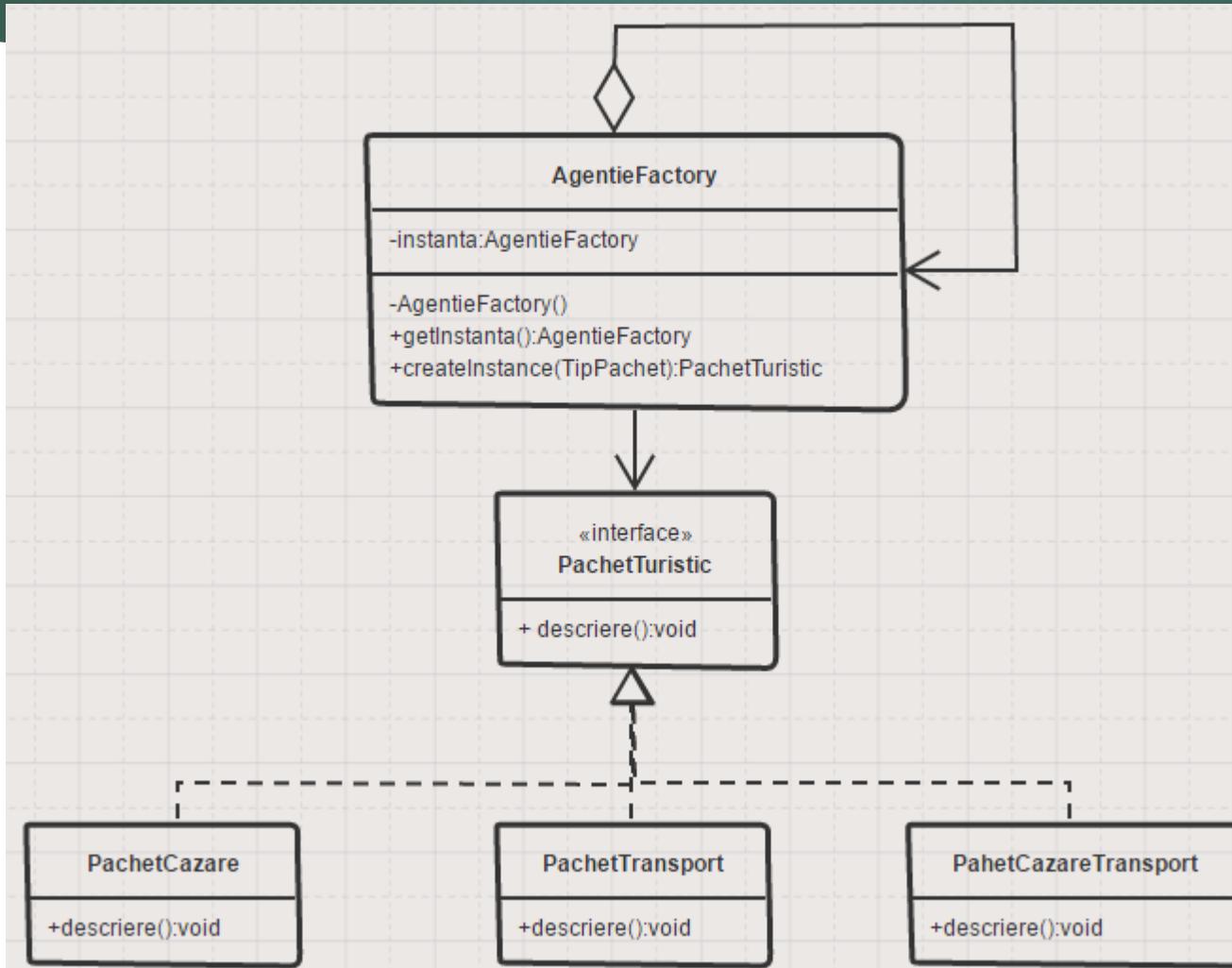
Builder – corelații

- ▶ **Singleton** – Clasa Builder poate fi singleton, astfel încât, construirea de obiecte să fie centralizată.

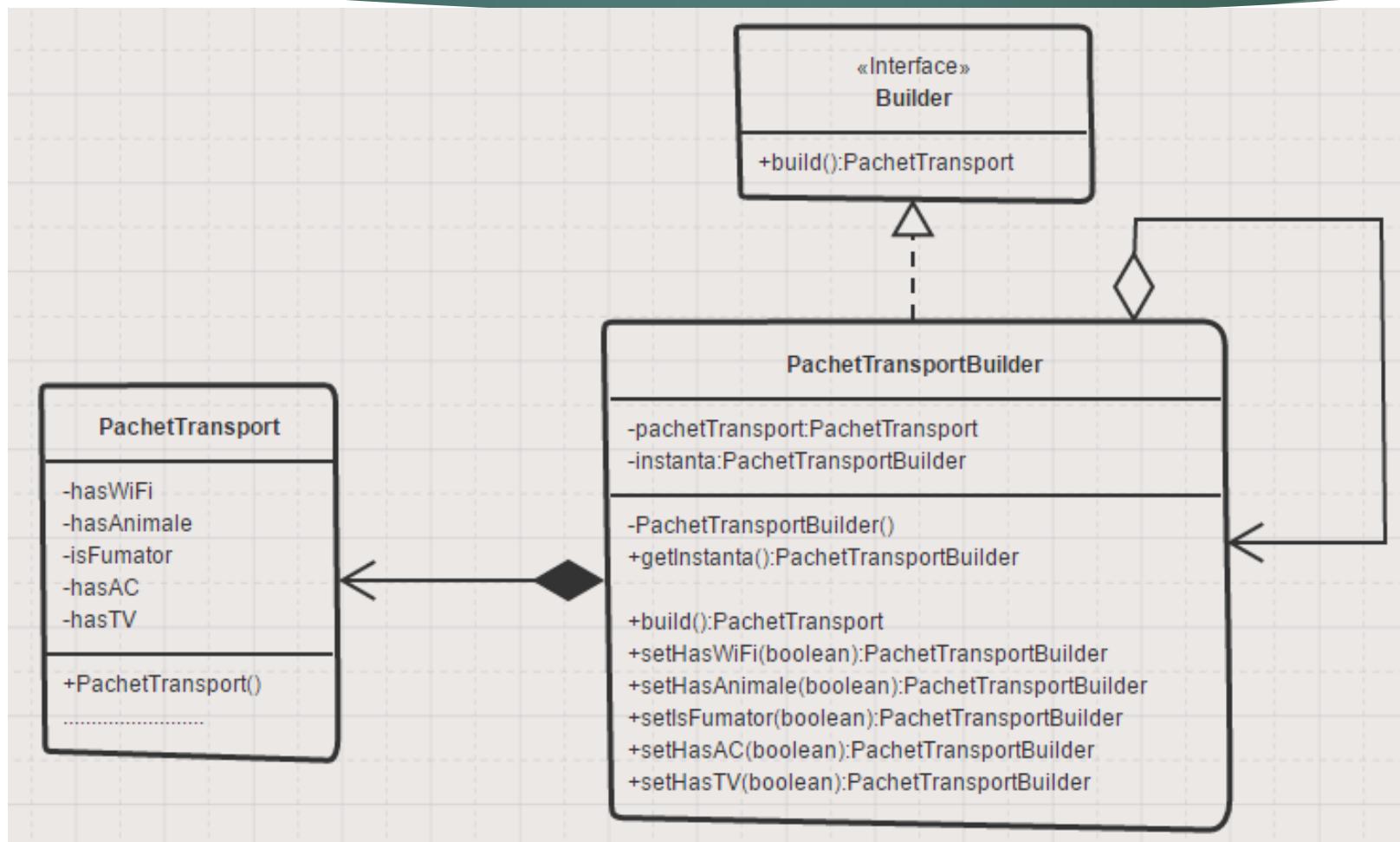
Builder - Alte exemple

- ▶ O banca pune la dispoziția clienților săi posibilitatea de creare de conturi pentru care dacă un client dorește poate să seteze să fie contul în care să primească salariul, să fie cu card atașat sau să aibă internet banking. În cazul în care clientul nu setează aceste informații, contul este creat fără aceste facilități.

Extra – Singleton - Factory



Extra – Singleton - Builder





Structural design patterns

ALIN ZAMFIROIU

Design pattern-urile structurale

- ▶ Ajută la compunerea și configurarea claselor și obiectelor;
- ▶ Design patern-urile structurale sunt concentrate pe cum sunt compuse clasele și obiectele pentru formarea de structuri complexe (obiecte complexe).

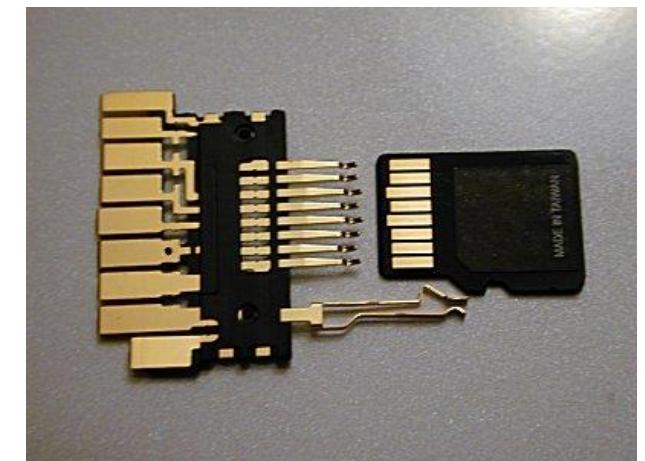
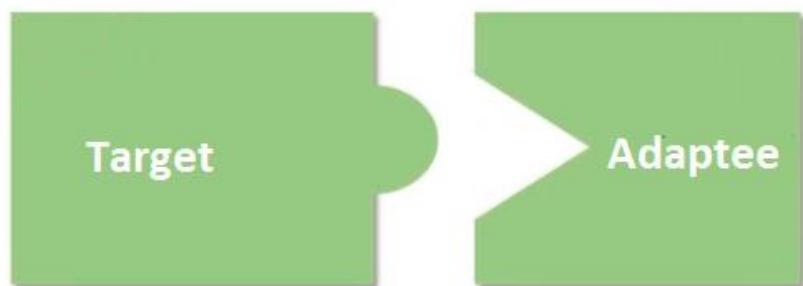
Structural design patterns

- ▶ Contribuie la compoziția claselor și obiectelor, realizând decuplarea interfețelor de clase:
 - ▶ **Adapter;**
 - ▶ **Facade;**
 - ▶ **Decorator;**
 - ▶ **Composite;**
 - ▶ **Flyweight;**
 - ▶ **Proxy;**
 - ▶ Bridge.

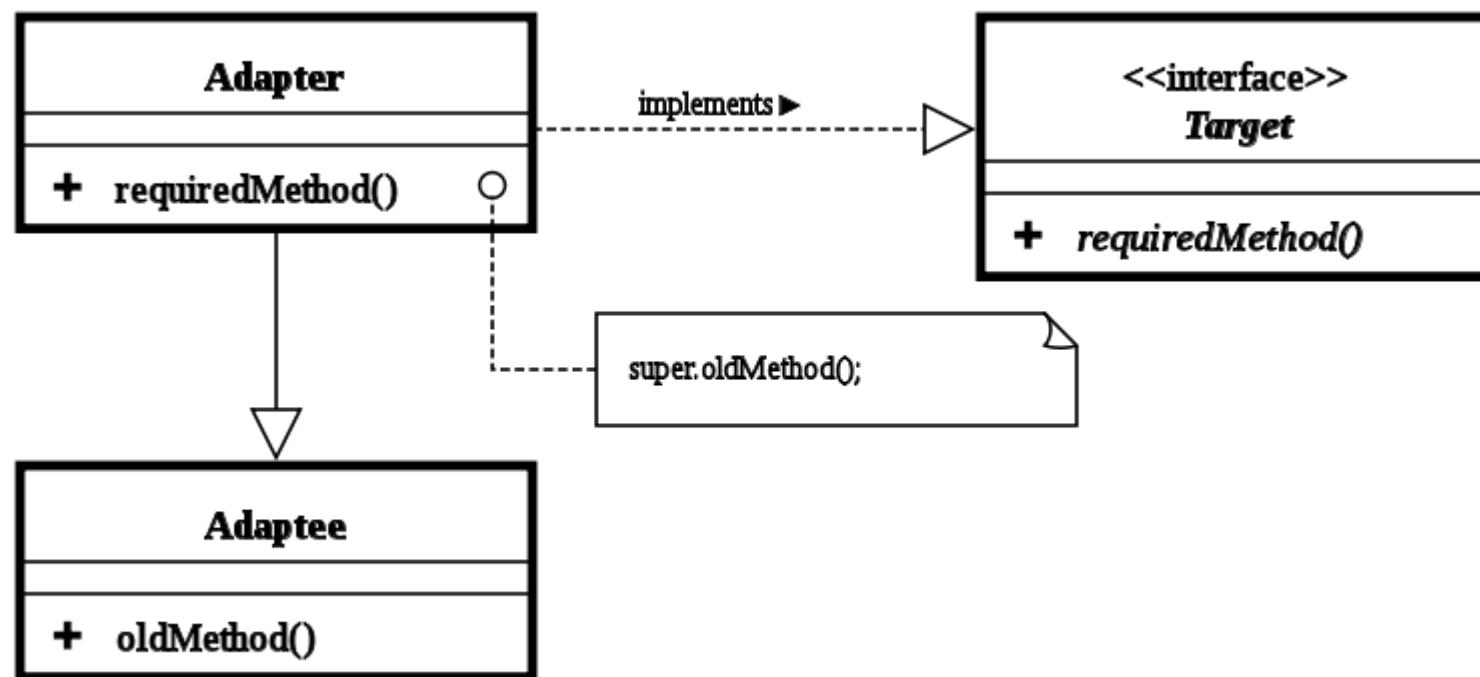
Adapter - nume

- ▶ Design pattern-ul **Adapter** rezolva problema utilizării anumitor clase din framework-uri diferite, care nu au o interfață comună.
- ▶ Clasele existente nu se vor modifica ci se va adăuga noi clase pentru realizarea unui **Adapter** între acestea. Clasa **Wrapper**.
- ▶ Utilizarea claselor existente se va face mascat prin intermediul adapterului creat.
- ▶ **Important:** Adapterul nu adaugă funcționalitate. Funcționalitatea este realizată de clasele existente.

Adapter - structură



Adapter - Diagramă



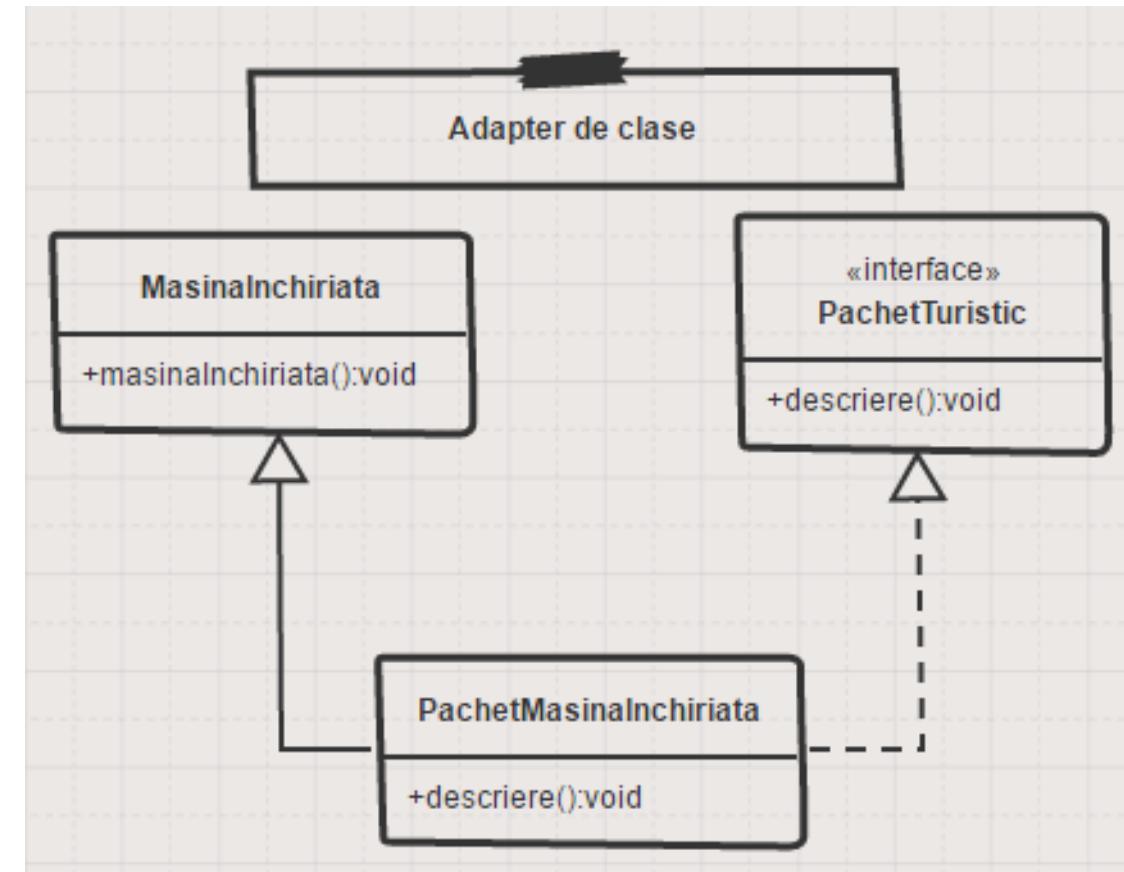
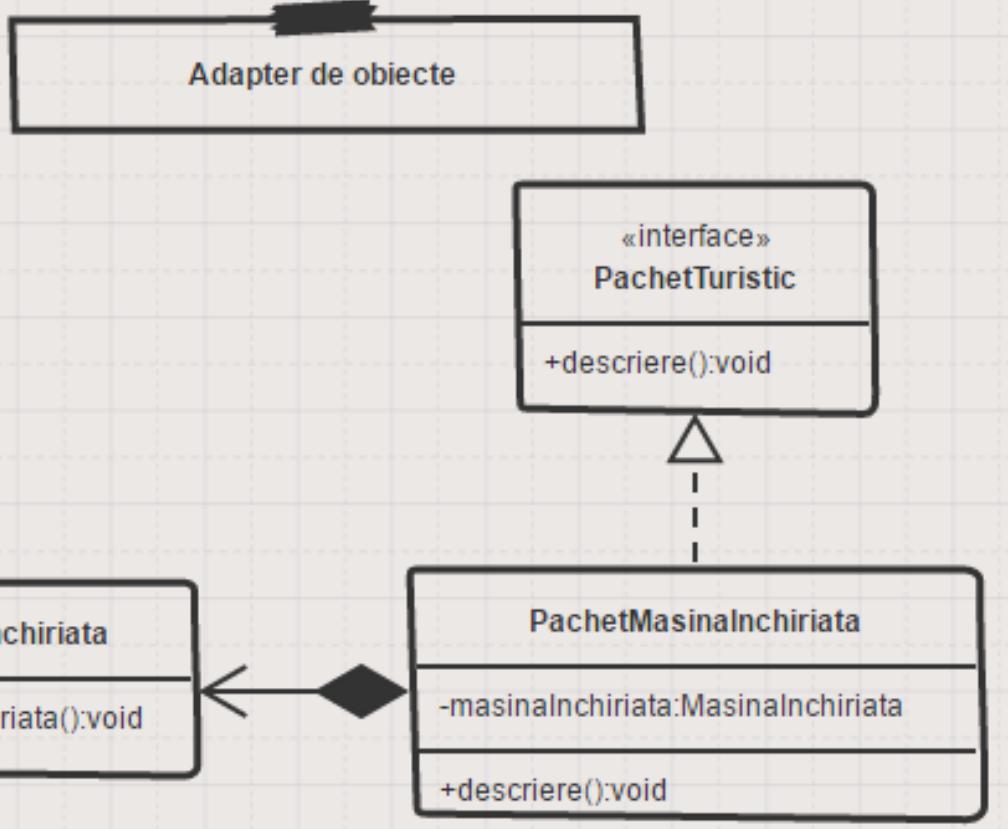
Adapter - problemă

- ▶ Agenția de turism **AgeTur** dorește să se ocupe și de închiriat mașini pentru clienții săi. De aceea cumpără un soft pentru închiriat mașini, însă operatorul trebuie să intre în fiecare aplicație pentru rezervarea unui pachet complet de cazare, transport și mașină închiriată.
- ▶ Clasele din softul achiziționat nu sunt asemănătoare cu clasele din softul deținut de agenție.
- ▶ Să se rezolve problema, astfel încât, cele două soft-uri să poată fi folosite împreună, însă fără a modifica clasele din cele două soft-uri.

Adapter - structură

- ▶ Există două tipuri de Adapter.
- ▶ Aceste tipuri diferă prin modul de implementare.
- ▶ **Adapter de obiecte**
- ▶ **Adapter de clase**

Adapter - structură



Adapter – participanți

- ▶ Clasa existentă – **MasinaInchiriată** – este clasa achiziționată ce trebuie adaptată.
- ▶ Clasa utilizată – **PachetTuristic** – este clasa sau interfața utilizată în cadrul aplicației. La această clasă/interfață trebuie adaptată clasa existentă.
- ▶ Adapter – **PachetMasinaInchiriată** – este adaptorul creat astfel încât clasa existentă să poată fi folosită ca și o clasă utilizată.

Adapter - implementare

► Adapter de obiecte:

- Clasa **Adapter** conține o **instanță** a clasei existente și implementează interfața la care trebuie să facă adaptarea.

- Prin implementarea interfeței se asigură implementarea unui set de metode. Aceste metode vor face apeluri/call-uri ale metodelor specifice clasei existente prin intermediul instanței.

Adapter - implementare

► Adapter de clase:

- Clasa **Adapter** moștenește clasa existente și implementează interfața la care trebuie să facă adaptarea.
- Prin implementarea interfeței se asigură implementarea unui set de metode. Aceste metode vor face apeluri/call-uri ale metodelor specifice clasei existente prin intermediul părintelui (**super**).
- Atenție: **Java nu permite moștenire multiplă.**

Adapter - utilizări

- ▶ Se folosește ori de câte ori este necesară conlucrarea mai multor framework-uri și nu se dorește modificarea codului existent.

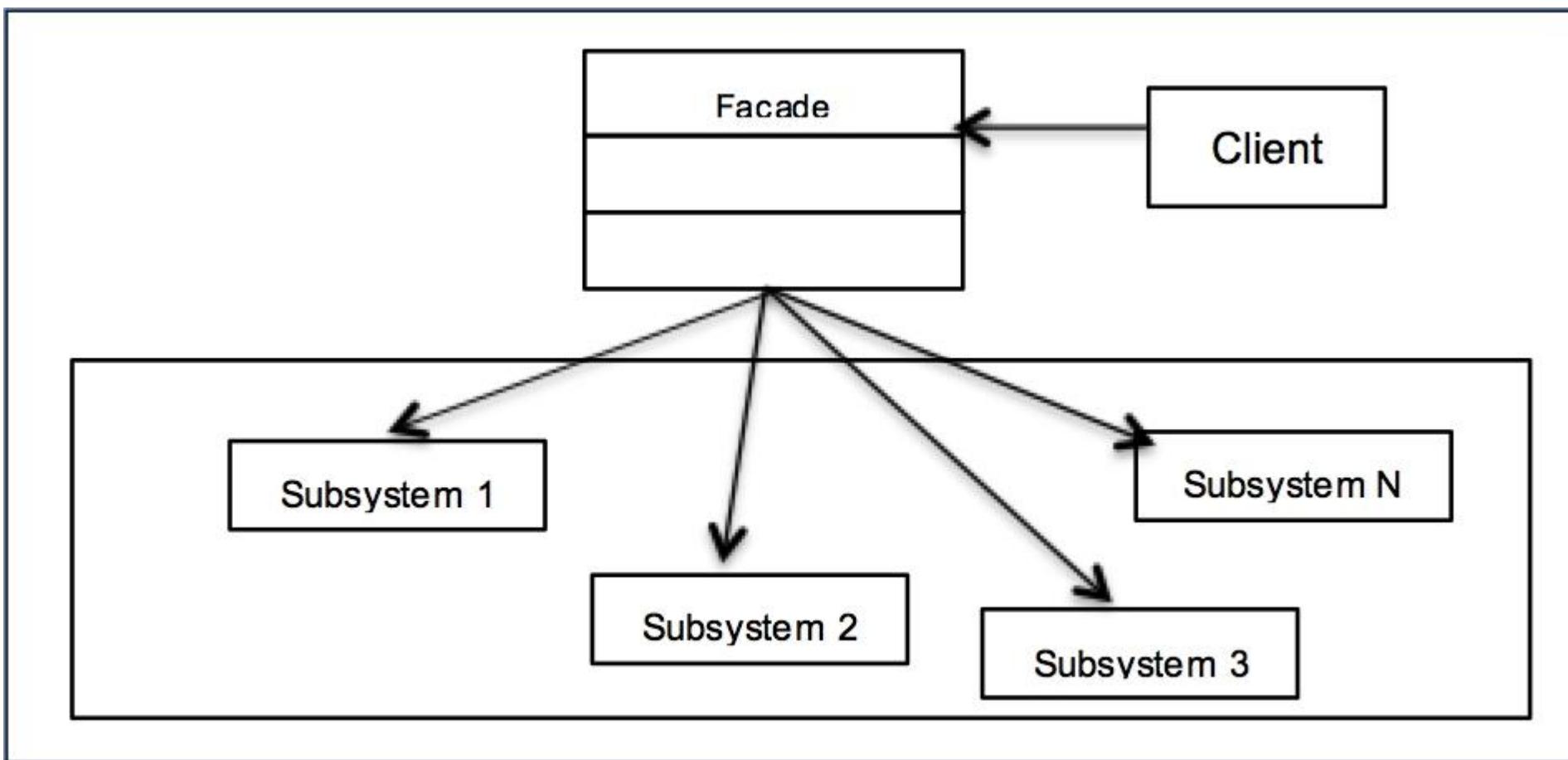
Adapter – corelații

- ▶ **Fațade** – ambele sunt wrappere;
- ▶ **Decorator** – fac același lucru, cu specificarea faptului că decorator adaugă și funcționalități noi;
- ▶ **Proxy** – ambele ascund într-un fel clasa existentă;
- ▶ **Bridge** – sunt asemănătoare ca și structură.

Façade - nume

- ▶ **Ușurează** lucrul cu framework-uri foarte complexe.
- ▶ Realizează o **fațadă** pentru aceste framework-uri, iar cine dorește să utilizeze acele framework-uri, poate folosi această fațadă, fără a fi necesară cunoașterea tuturor claselor, metodelor și atributelor din cadrul framework-ului

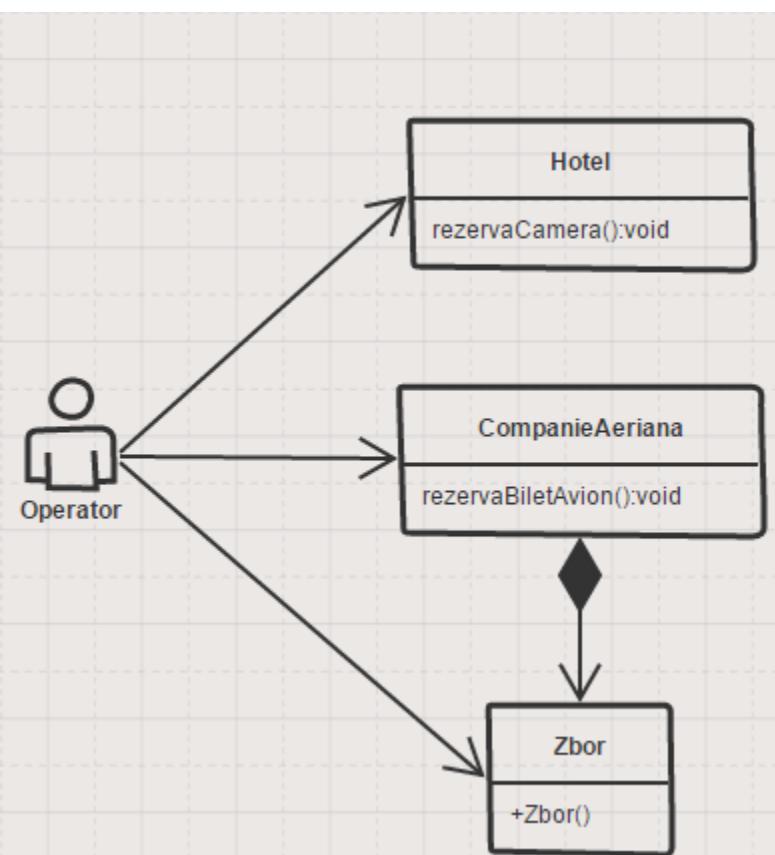
Façade - Diagramă



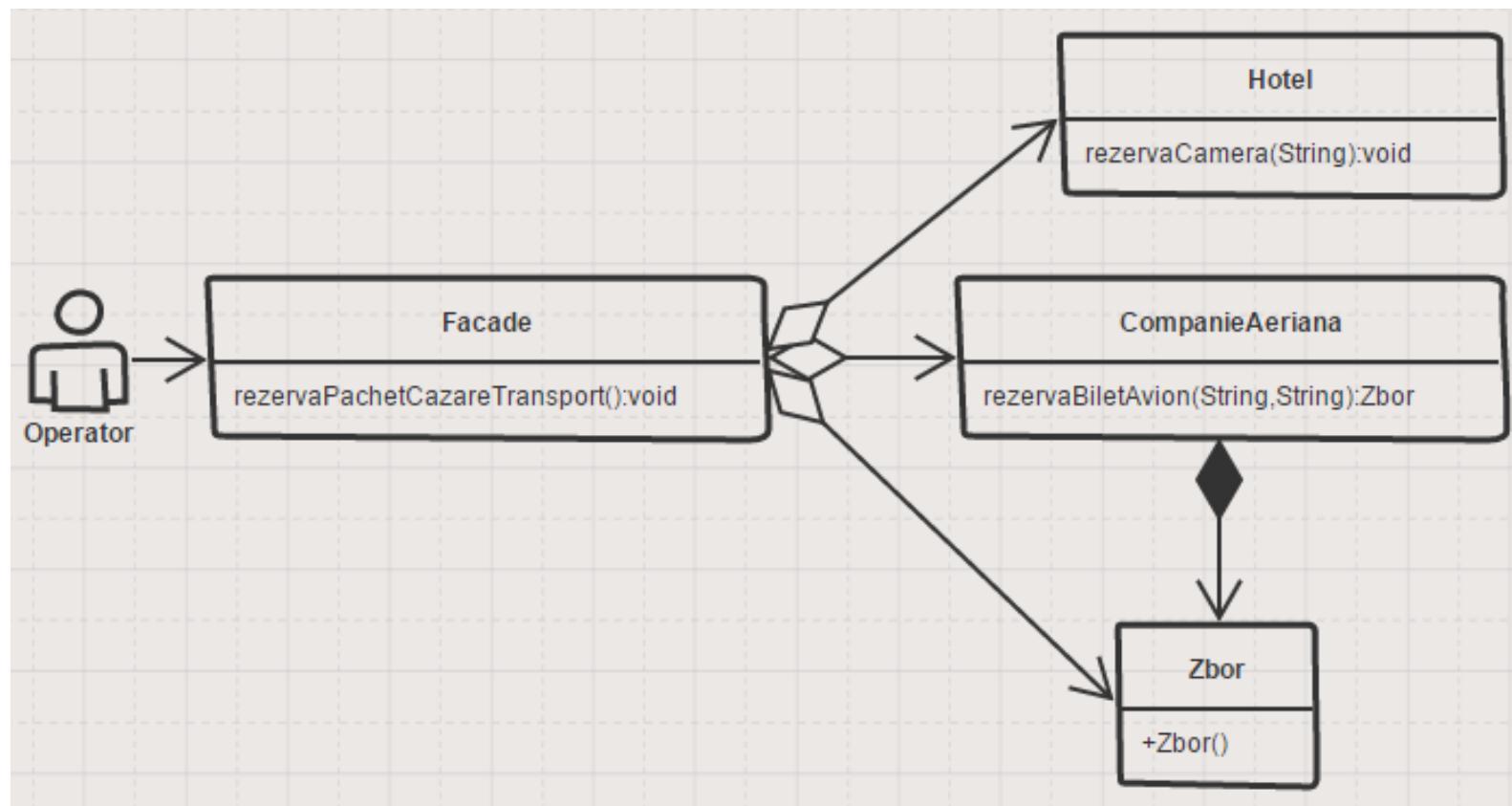
Façade - problemă

- ▶ Pentru realizarea rezervărilor pachetelor turistice un operator trebuie să creeze pachetul turistic ideal pentru client. Apoi trebuie să rezerve cazarea la hotelul dorit, să caute zboruri pentru clienți și să salveze noul pachet rezervat.
- ▶ Să se realizeze un modul care să simplifice procesul de rezervare pentru operator.

Façade - structură



VS



Façade – participanți

- ▶ Clasele din cadrul framework-ului folosit – **Hotel, CompanieAeriana, Zbor**;
- ▶ **Facade** – clasa care ascunde complexitatea framework-ului.

Façade - implementare

- ▶ Clasa **Facade** cuprinde metode care să utilizeze metodele din clasele framework-ului.
- ▶ Clasa **Facade** ascunde complexitatea prin apelurile sale.

Façade - utilizări

- ▶ Pentru framework-urile open-source disponibile.

Façade – corelații

- ▶ **Singleton** – Fațada poate fi o unică instanță.

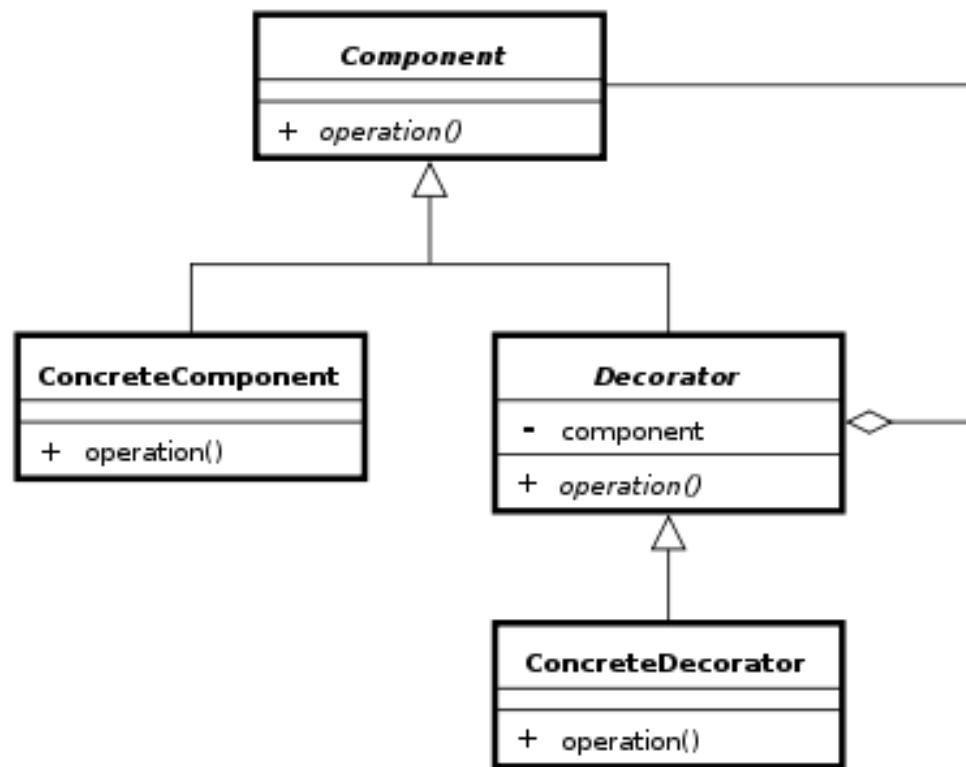
Façade – un alt exemplu

- ▶ În momentul în care un client vine la recepție pentru o cameră, recepționistul trebuie să verifice dacă are camera liberă, apoi să verifice dacă acea cameră a fost curățată de la plecarea ultimului client, de asemenea trebuie să verifice dacă au fost puse prosoape noi în cameră. Managerul hotelului dorește realizarea unui modul care să simplifice munca recepționistului și să nu mai fie nevoie să verifice în toate locurile ci să verifice într-un singur loc.
- ▶ Să se implementeze modulul care permite acest lucru.

Decorator - nume

- ▶ Este folosit pentru modificarea funcționalității unui obiect la runtime.
- ▶ Este folosit de asemenea pentru adăugarea de noi funcționalități sau noi caracteristici unui obiect la runtime.
- ▶ Se pot face decorări multiple prin moștenire continuă.

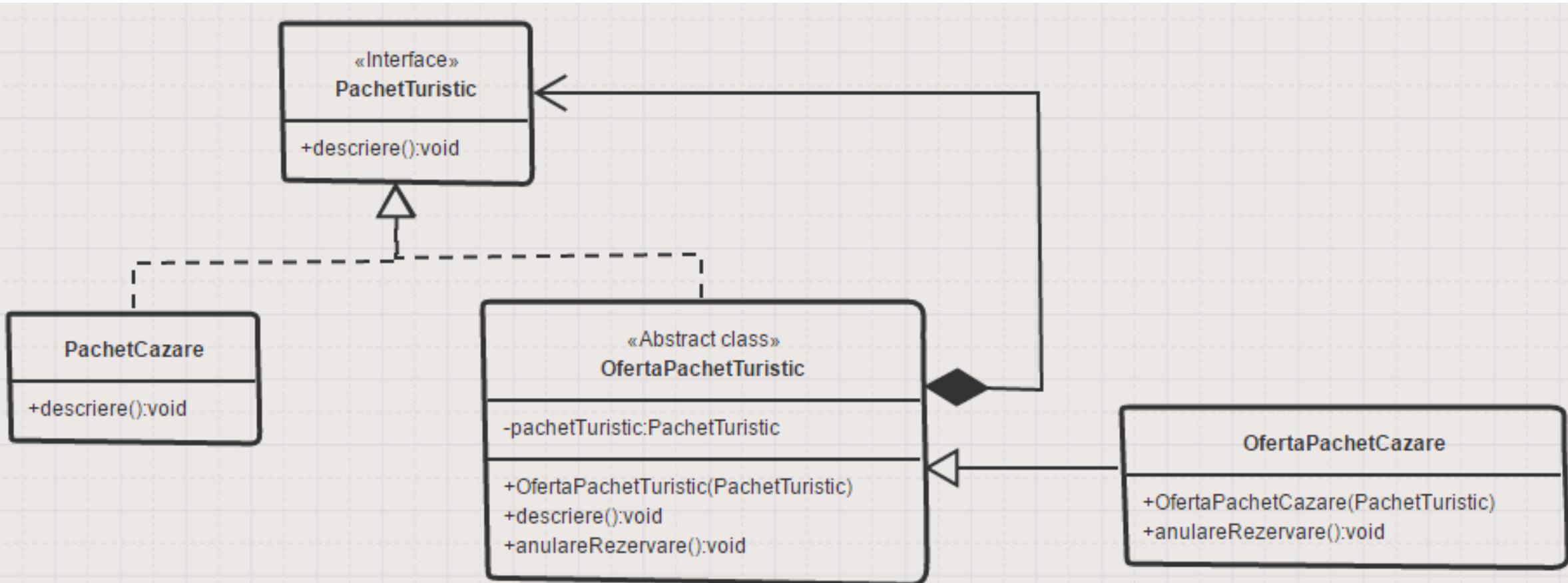
Decorator - Diagramă



Decorator - problemă

- ▶ Agenția de turism dorește să ofere pentru o perioadă de trei luni posibilitatea de anulare a pachetelor rezervate. Această ofertă expiră în trei luni. Managerul agenției nu dorește să modifice codul, deoarece după trei luni trebuie să modifice iar codul sursă.
- ▶ Să se implementeze modulul care asigură oferta agenției fără a se modifica codul existent.

Decorator - structură



Decorator – participanți

- ▶ AbstractProduct – **PachetTuristic** – clasa abstractă sau interfața care definește familia de obiecte existente în codul sursă;
- ▶ ConcreteProduct – **PachetCazare** – clasa concretă existentă în codul sursă;

- ▶ AbstractDecorator – **OfertaPachetTuristic** - clasa abstractă care definește decoratorul;
- ▶ ConcreteDecorator – **OfertaPachetCazare** – clasa concretă care decorează ConcreteProduct.

Decorator - implementare

- ▶ În cadrul clasei abstracte OfertaPachetTuristic se implementează interfața care definește familia de obiecte și se creează o instanță de tipul acelei interfețe.
- ▶ Pentru metoda din interfață se furnizează o implementare și se adaugă noi funcții abstractive.
- ▶ În cadrul clasei decorator concret se implementează și noile metode din decoratorul abstract.

Decorator - utilizări

- ▶ Pentru adăugarea de noi funcționalități claselor existente.
- ▶ Pentru îmbunătățirea claselor existente fără a modifica codul existent și fără a face extindere sau moștenire

Decorator – corelații

- ▶ **Adapter** - fac același lucru, cu specificarea faptului că decorator adaugă și funcționalități noi;
- ▶ **Strategy** – Decorator modifică întreaga clasa, iar Strategy modifică comportamentul.

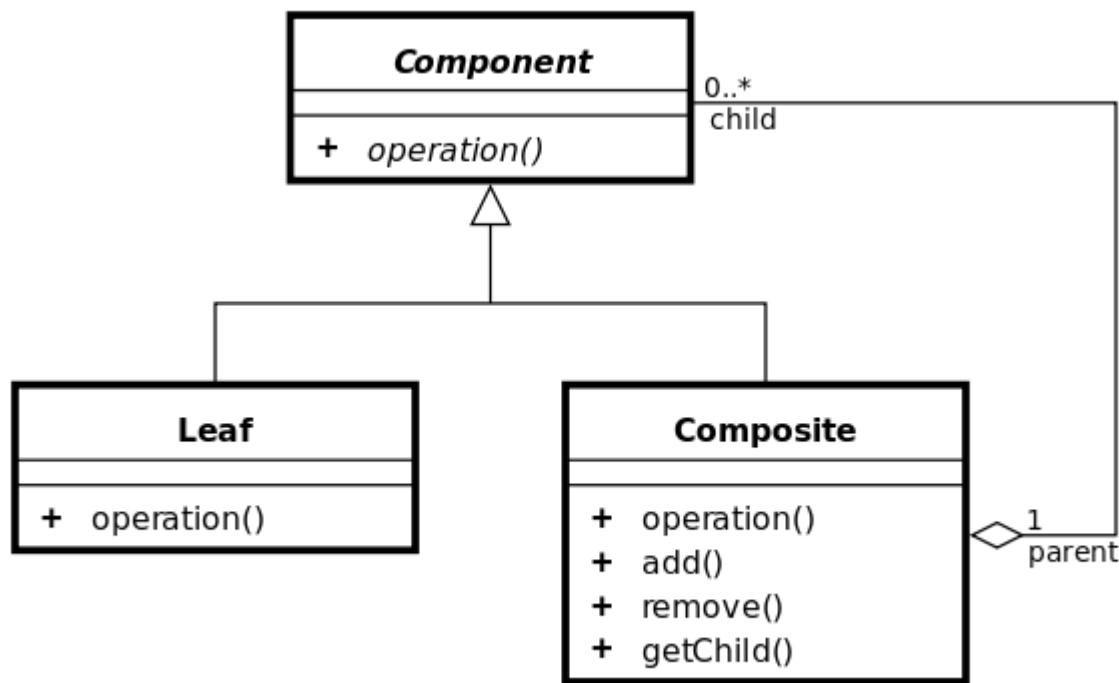
Decorator - Alt exemplu

- ▶ Cu ocazia sărbătorilor de sfârșit de an managerul hotelului dorește ca atunci când este printată o factură să se printeze și o felicitare cu mesajul La mulți ani pentru client și de asemenea să acorde un discount cu un procent primit ca parametru de funcție.
- ▶ Se dorește adăugarea acestei noi funcționalități pentru clasa Factură. Funcționalitatea va fi adăugată obiectului la momentul apelului de printare.

Composite - nume

- ▶ Este un design patterns structural folosit atunci când este necesară crearea unei structuri ierarhice sau o structură arborescentă prin compunerea de obiecte.
- ▶ **ATENȚIE:** Composite nu este o structură de date (arbore). Composite este un design pattern.

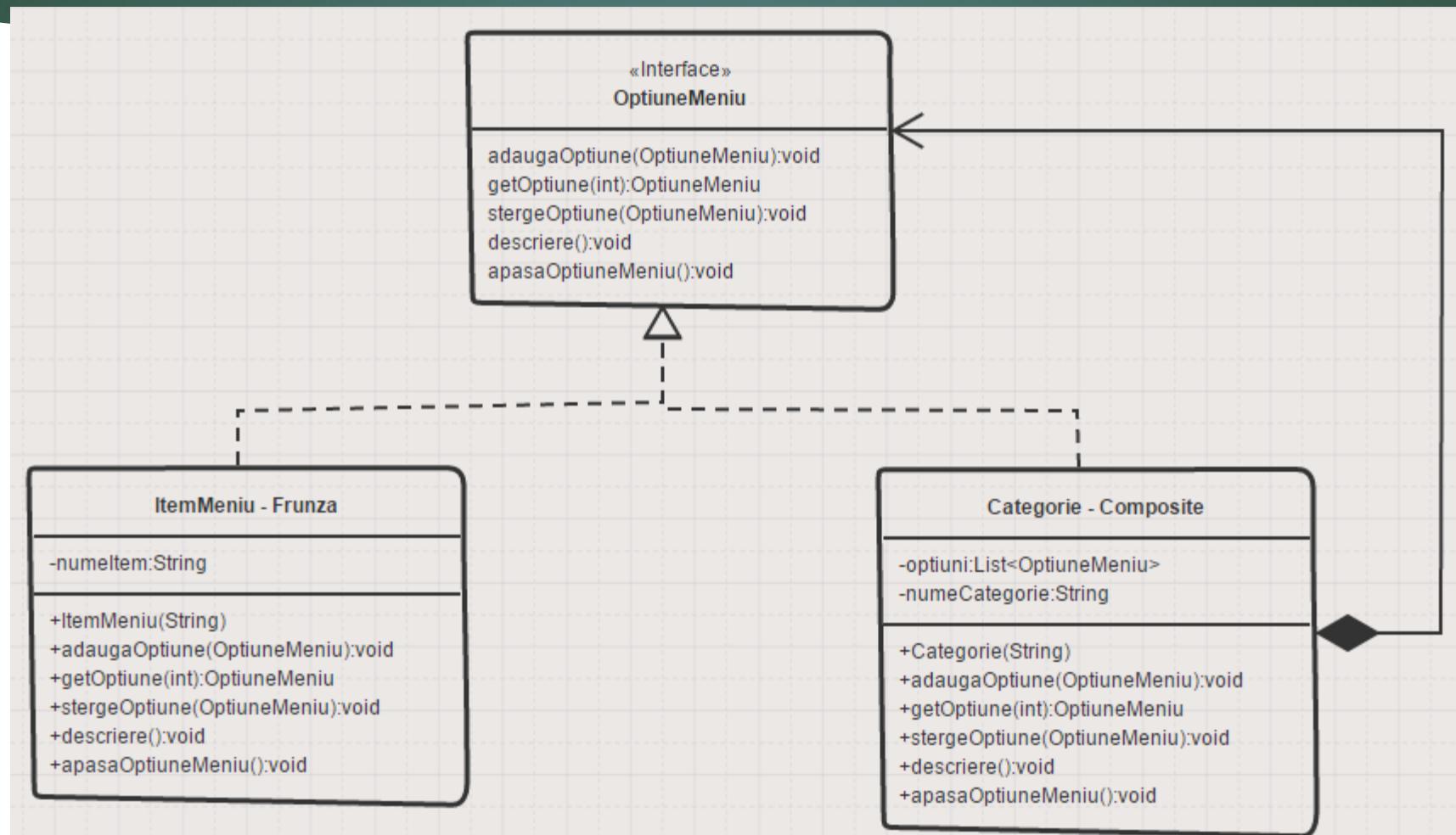
Composite - Diagramă



Composite - problemă

- ▶ Agenția dorește realizarea unei aplicații software în care meniul să fie organizat într-o structură arborescentă pe categorii. Fiecare categorie poate conține alte subcategorii cu opțiuni, pe care operatorul să le acceseze. Astfel rezultă o structură ierarhică cu toate opțiunile.
- ▶ Să se implementeze modulul pentru generarea meniului aplicației software dorite de agenția de turism.

Composite - structură



Composite – participanți

- ▶ Componenta abstractă – **OptiuneMeniu** – clasă abstractă sau interfață care descrie toate componentele arborescenței;
- ▶ Composite – **Categorie** – reprezintă componenta care poate să conțină fii;
- ▶ NodFrunza – **ItemMeniu** – reprezintă componentele care nu au fii.

Composite - implementare

- ▶ Clasele **Composite** conțin o listă cu elemente de tip ComponentAbstracta. În această listă se adauga și se sterg noduri noi.
- ▶ Clasele **NodFrunza** implementează metodele de adaugare nod sau stergere nod, chiar dacă nu au nicio implementare pentru ele.

Composite - utilizări

- ▶ Meniurile aplicațiilor;
- ▶ Meniurile de la restaurant;
- ▶ Organizarea studentilor în grupe și serii. (Seria E - 2024)
- ▶ Pentru reprezentarea oricărei arborescențe.

Composite – corelații

- ▶ **Decorator** – Nodurile Composite pot fi privite ca Noduri frunza decorate.
- ▶ **Chain of Responsibility** – legăturile dintre clase.

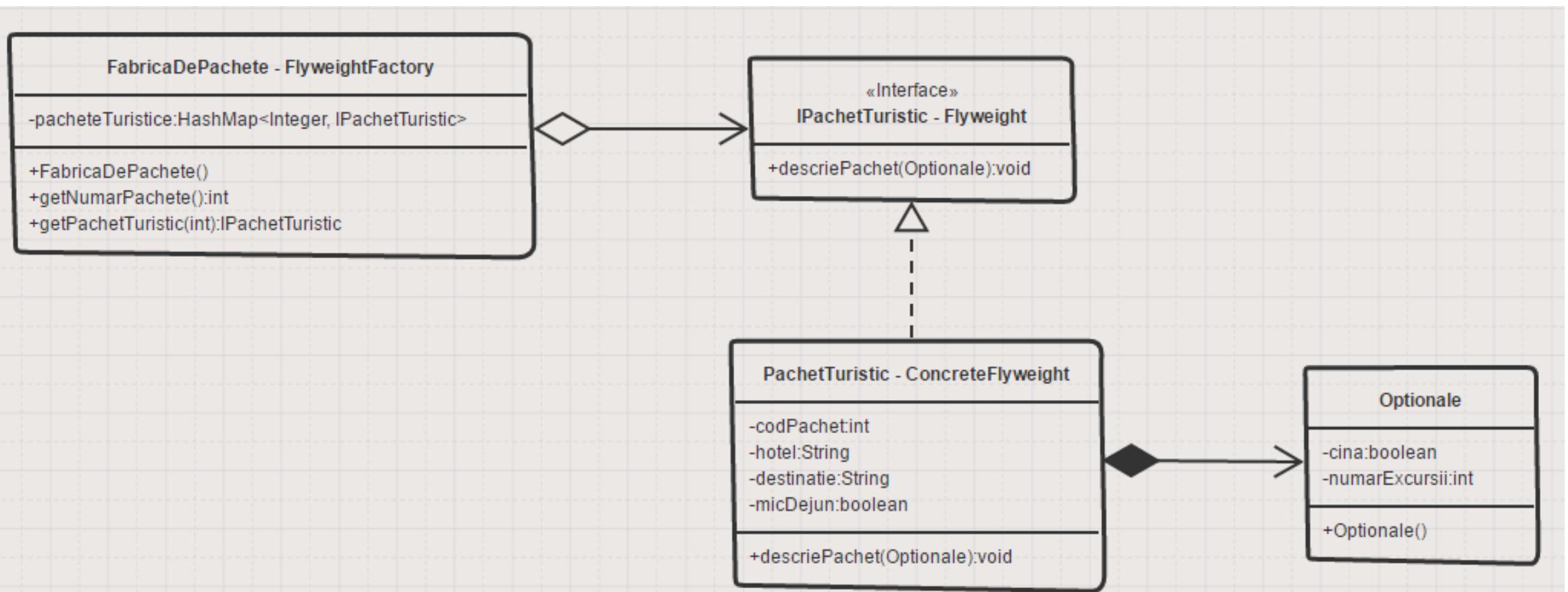
Flyweight - nume

- ▶ Este utilizat atunci când trebuie să construim foarte multe obiecte/instanțe ale unei clase, însă majoritatea obiectelor au o parte comună, sau permanentă.
- ▶ Astfel prin utilizarea design pattern-ului **Flyweight** se reduce consumul de memorie, păstrându-se într-o singură instanță partea comună.
- ▶ Partea care diferă de la un obiect la altul este salvată într-o altă clasă și este adăugat după construirea obiectelor.

Flyweight - problemă

- ▶ Agenția organizează excursii și cu grupuri. Toate pachetele celor din grup vor avea același hotel de cazare, aceeași destinație și toți vor avea sau nu mic dejun inclus. Identificarea realizându-se după codul pachetului. Opțional fiecare persoană, dacă dorește, poate să aibă cina inclusă în pachet și numărul de excursii în care să meargă în zilele libere.
- ▶ Să se implementeze modulul care asigură gestiunea de pachete turistice pentru cienții agenției, ținându-se cont de faptul că memoria trebuie folosită în mod optim.

Flyweight - structură



Flyweight – participanți

- ▶ Flyweight – **IPachetTuristic** – interfață care leagă partea extrinsecă de partea intrinsecă a obiectelor (partea permanentă și partea temporară);
- ▶ ConcreteFlyweight – **PachetTuristic** – clasa care extinde interfața Flyweight și realizează implementarea pentru partea permanentă a obiectului;
- ▶ UnsharedConcreteFlyweight – **Optionale** – clasa care definește starea temporară sau partea extrinsecă a obiectelor;
- ▶ FlyweightFactory – **FabricaDePachete** – clasa care gestionează obiecte de tip ConcreteFlyweight într-o colecție.

Flyweight - implementare

- ▶ Clasa **FabricaDePachete** conține un *HashMap* pentru reținerea pachetelor asemănătoare.
- ▶ Clasa **PachetTuristic** reține codul, hotelul, destinația și daca are sau nu mic dejun, deoarece aceste atrbute au aceleași valori pentru mai mulți turiști.
- ▶ Clasa **Optionale** retine doar numărul de excursii și daca are sau nu cina, deoarece aceste atrbute diferă foarte mult.

Flyweight - utilizări

- ▶ În jocuri, atunci când foarte multe modele seamănă, însă diferă prin culoare sau prin poziție (copaci, mașini, oameni, etc).

Flyweight – corelații

- ▶ **Composite** – nodurile frunză pot reprezenta aceeași instanță;
- ▶ **Factory** – construirea de obiecte este gestionată de o clasă

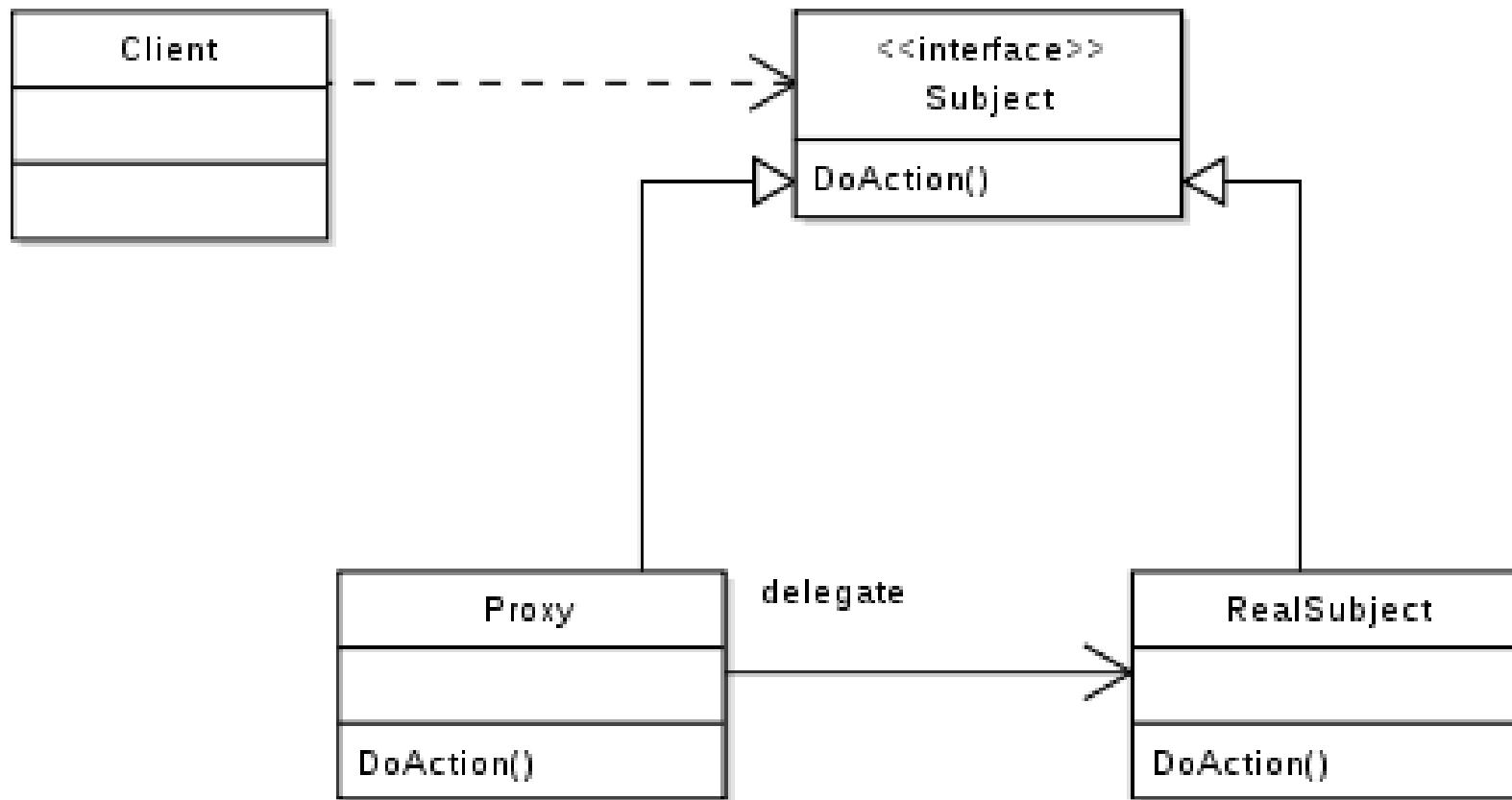
Flyweight – Alt exemplu

- ▶ Pentru crearea unui obiect de tip Cameră trebuie reținute informații cu privire la inventarul camerei: numărul de prosoape, numărul de cearșafuri, numărul de perne, numărul de umerașe, numărul camerei, numele clientului găzduit. Managerul hotelului dorește ca aceste lucruri să fie tipărite pentru fiecare client, însă se dorește ca acest lucru să fie realizat optim din punct de vedere al utilizării memoriei.

Proxy - nume

- ▶ Este utilizat atunci când se dorește păstrarea funcționalității unei clase, însă aceasta se va realiza doar în anumite condiții.
- ▶ Prin Proxy se controlează comportamentul și accesul la un obiect.

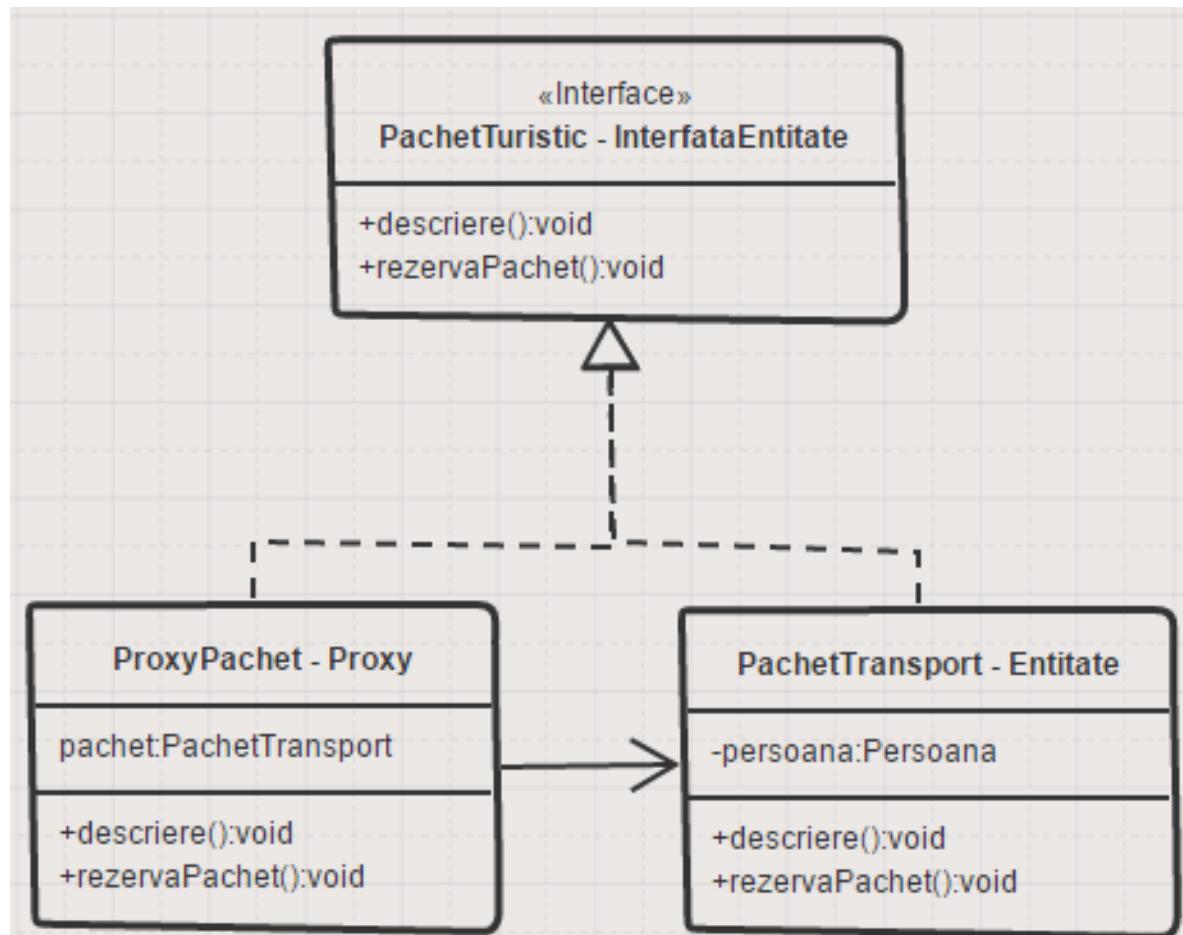
Proxy - Diagramă



Proxy - problemă

- ▶ Agentia ia hotărârea că pachetele de transport din ofertă pot fi rezervate doar de către pensionari – persoane cu vîrstă peste 65 de ani.
- ▶ Să se implementeze un nivel intermediar care să permită realizarea rezervării doar pentru persoanele cu vîrstă peste 65 de ani.

Proxy - structură



Proxy – participanți

- ▶ Interfața Entitate – **PachetTuristic** – interfață obiectului real;
- ▶ Entitate – **PachetTransport** – clasa obiectului real. Obiectele acestei clase vor fi controlate de către proxy.
- ▶ Proxy – **ProxyPachet** – clasa care gestionează referința către obiectul real.

Proxy - implementare

- ▶ Clasa **ProxyPachet** implementează interfața **PachetTuristic**, astfel încât să poată fi folosit ca un pachet turistic.
- ▶ Și are un atribut de tipul **PachetTransport**, acesta fiind obiectul gestionat, la care da acces în mod controlat.
- ▶ Metoda *rezervaPachet()* a instanței va fi apelată doar în cazul în care condițiile sunt îndeplinite.

Proxy - utilizări

- ▶ De fiecare dată când se dorește realizarea de permisiuni pentru anumite obiecte sau pentru accesul la anumite modificări ale obiectelor.

Proxy – corelații

- ▶ **Adapter** - ambele ascund într-un fel clasa existentă;
- ▶ **Decorator** – Proxy permite accesul la anumite funcționalități, iar Decorator adaugă noi funcționalități.
- ▶ **Facade** – este tot o interfață pentru obiectul real;

Proxy – alt exemplu

- ▶ Camerele rezervate la hotel au anularea rezervării în mod gratuit. Managerul ia decizia ca această anulare să fie gratuită doar pentru rezervările de maxim o noapte. Să se implementeze un modul prin care să se permită anularea rezervării doar pentru rezervările de cel mult o noapte.



Behavioral design patterns (comportamentale)

ALIN ZAMFIROIU

Behavioral design patterns

- ▶ Furnizează soluții pentru o mai bună interacțiune între obiecte și clase.
- ▶ Aceste design pattern-uri controlează relațiile complexe dintre clase.
- ▶ Permit distribuția responsabilităților pe clase și descrie interacțiunea între clase și obiecte

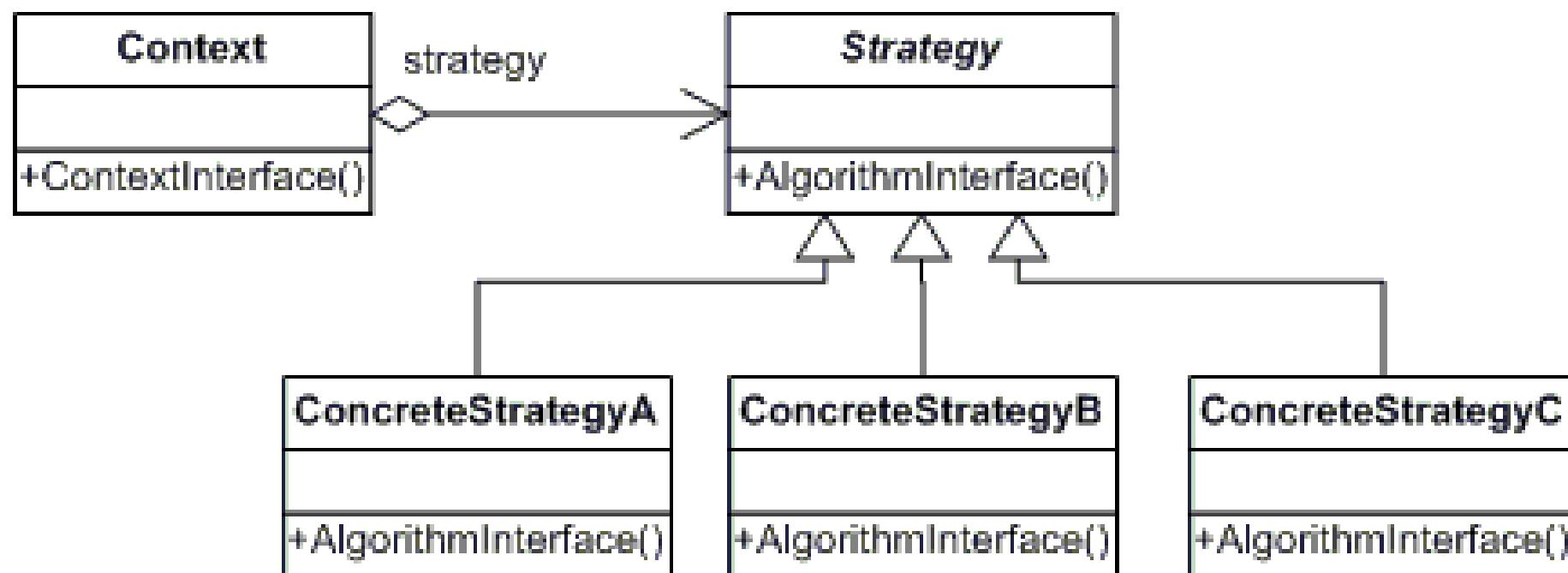
Behavioral design patterns

- ▶ **Strategy;**
- ▶ **Observer;**
- ▶ **Chain of Responsibility;**
- ▶ **State;**
- ▶ **Command;**
- ▶ **Template Method;**
- ▶ **Memento;**
- ▶ Interpreter;
- ▶ Mediator;
- ▶ Visitor;
- ▶ Iterator.

Strategy - nume

- ▶ Este folosit atunci când avem mai mulți algoritmi pentru rezolvarea unei probleme, iar alegerea implementării se face la run-time.
- ▶ Fiecare comportament este dat de o clasă.
- ▶ Definește strategia adoptată la run-time.

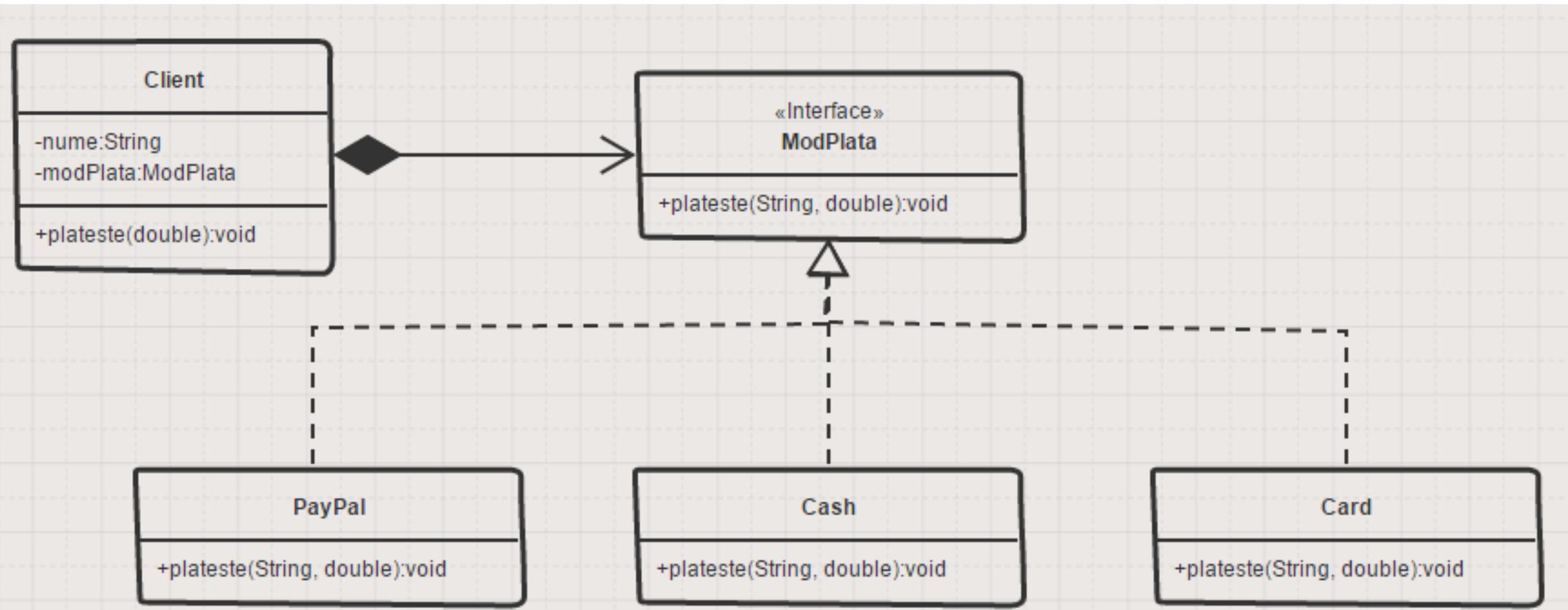
Strategy - Diagramă



Strategy - problemă

- ▶ Agenția de Turism dorește implementarea modului de plată al pachetelor turistice rezervate. Modul de plată îl decide clientul în momentul în care trebuie să facă plata. Plata se poate realiza cu cardul, cash sau prin PayPal.
- ▶ Să se implementeze modulul de plată

Strategy - structură



Strategy – participanți

- ▶ IStrategy – **ModPlata** – interfață sau clasă abstractă care definește la nivel abstract obiectele ce oferă noi moduri de prelucrare;
- ▶ Strategy – **Cash, Card, PayPal** – clasele concrete care implementează algoritmii de prelucrare.
- ▶ Obiect – **Client** – clasa care gestionează o referință de tipul IStrategy.

Strategy - implementare

- ▶ În cadrul fiecărei clase Strategy concret se implementează metoda de procesare.
- ▶ În cadrul clasei care gestionează instanța Strategy, se pune la dispoziție un mecanism de schimbare a metodei, și se are grija ca implementarea să folosească implementarea furnizată de instanța concretă a obiectului Strategy.

Strategy - utilizări

- ▶ Furnizarea metodei de comparare pentru metodele de sortare.
- ▶ Utilizarea validatoarelor pentru anumite controale.

Strategy – corelații

- ▶ **State** – cele două design pattern-uri sunt foarte asemănătoare, diferență fiind dată de faptul că la strategy, strategia este dată ca parametru, iar la state trecerea de la o stare la alta se face controlat.

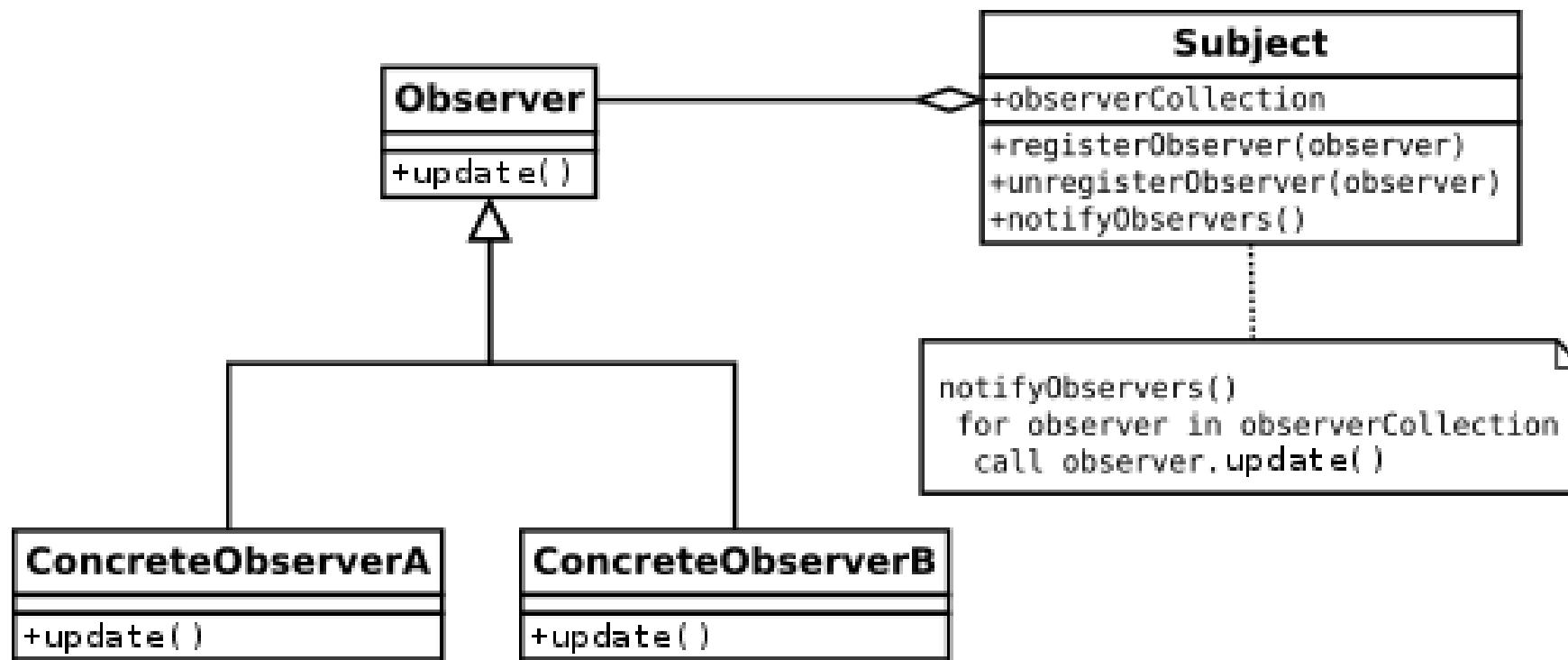
Strategy – Alt exemplu

- ▶ Pentru fiecare client, recepționerul hotelului trebuie să realizeze verificarea actelor în momentul cazării. Pentru cetățenii europeni din UE, verificarea constă în controlul buletinului și scanarea acestuia. Pentru cetățenii europeni non-UE, verificarea constă în controlul pașaportului și scanarea acestuia, iar pentru cetățenii americanii constă în controlul vizei și a scanării acesteia. Această verificare se face în momentul în care turistul ajunge la recepție, deci modul de verificare se stabilește la run-time. Să se implementeze modulul care face verificarea actelor turiștilor cazați la hotel.

Observer - nume

- ▶ Observer definește o relație de “1 : n”, în care un subiect notifică mai mulți observeri.
- ▶ Acest design pattern este folosit atunci când anumite elemente (obiecte) trebuie să fie anunțate de schimbarea stării altor obiecte.

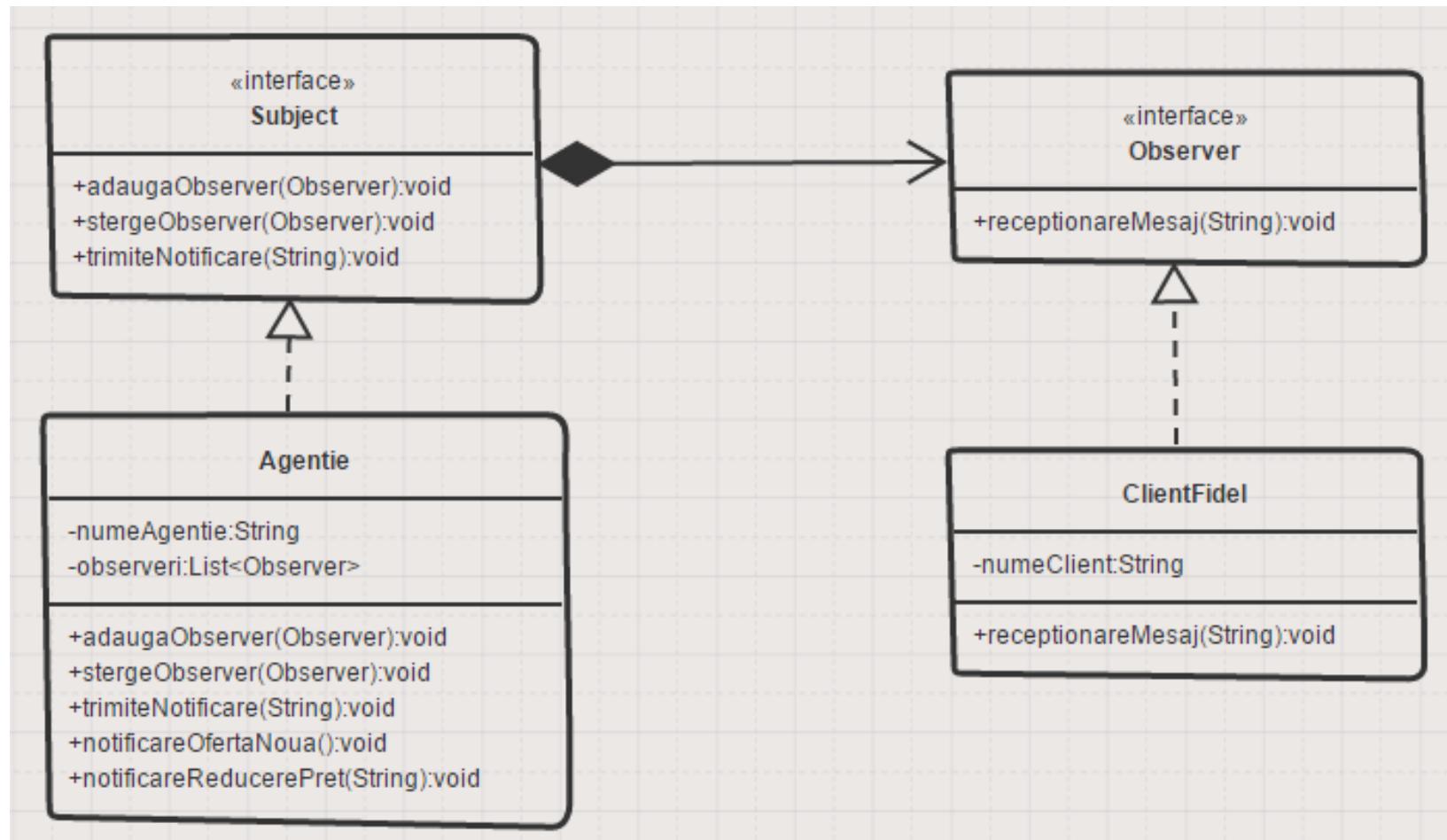
Observer - Diagramă



Observer - problemă

- ▶ Agentia de turism dorește să anunțe clienții fideli ori de câte ori apar noi oferte. Astfel se dorește implementarea unui modul care atunci când se realizează o ofertă de preț sau se introduce un nou pachet să se trimită notificări tuturor clienților abonați la notificările agenției de turism.

Observer - structură



Observer – participanți

- ▶ Observator – **Observer** – clasa abstractă sau interfață care definește la nivel abstract obiectele ce vor fi notificate.
- ▶ ObservatorConcret – **ClientFidel** – clasele care definesc la nivel concret observatorii sau obiectele care vor fi notificate.
- ▶ Observabil – **Subject** – clasa abstractă sau interfață care definește la nivel abstract obiectele care gestionează lista de observatori și care atunci când își modifică starea notifică observatorii din listă.
- ▶ ObservabilConcret – **Agentie** – clasa concretă care gestionează lista de observatori.

Observer - implementare

- ▶ În cadrul clasei concrete a subiectului observabil se gestionează o listă de obiecte care observă.
- ▶ În metoda de trimitere notificare se parcurge lista de observatori și se trimitе un mesaj fiecărui prin apelul funcției specifice.

Observer - utilizări

- ▶ **Model View Controller**

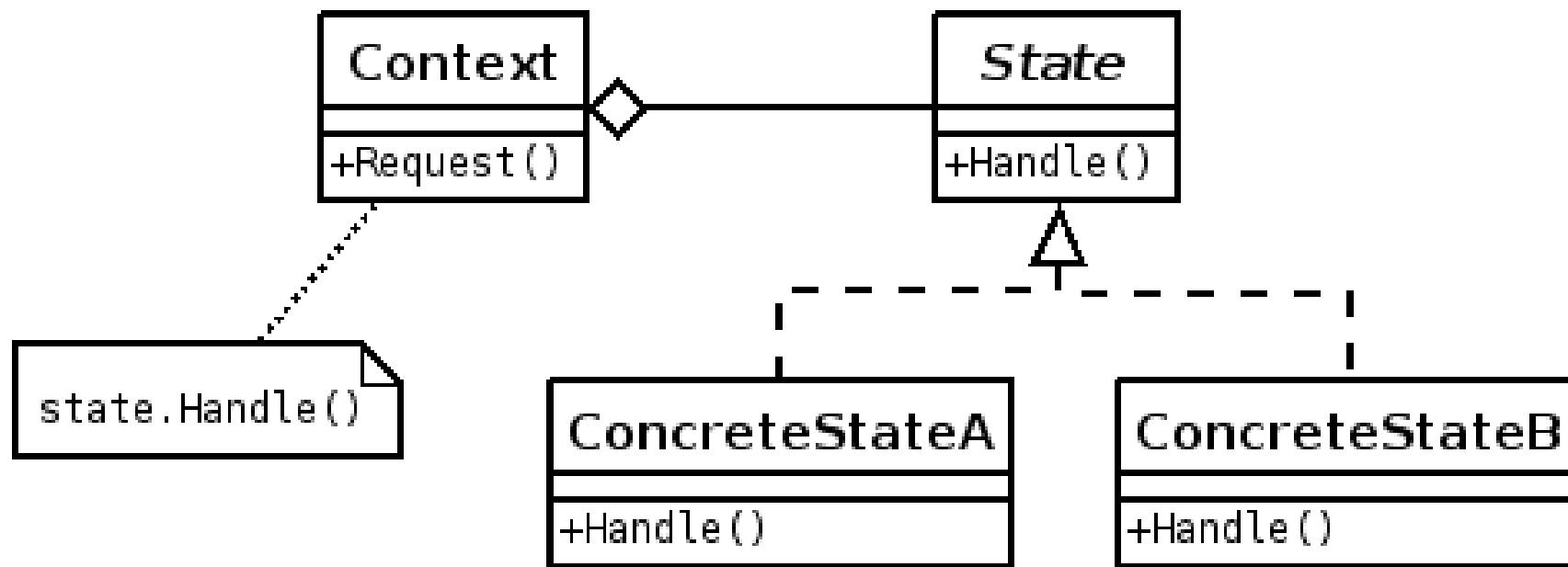
Observer – corelații

- ▶ **Singleton** – Subiectul poate fi unic.

State - nume

- ▶ **State** este un design pattern comportamental folosit atunci când un obiect își schimbă comportamentul pe baza stării în care se află.
- ▶ Este foarte asemănător cu **Strategy**, diferența constă în modul de schimbare a stării respectiv a strategiei.

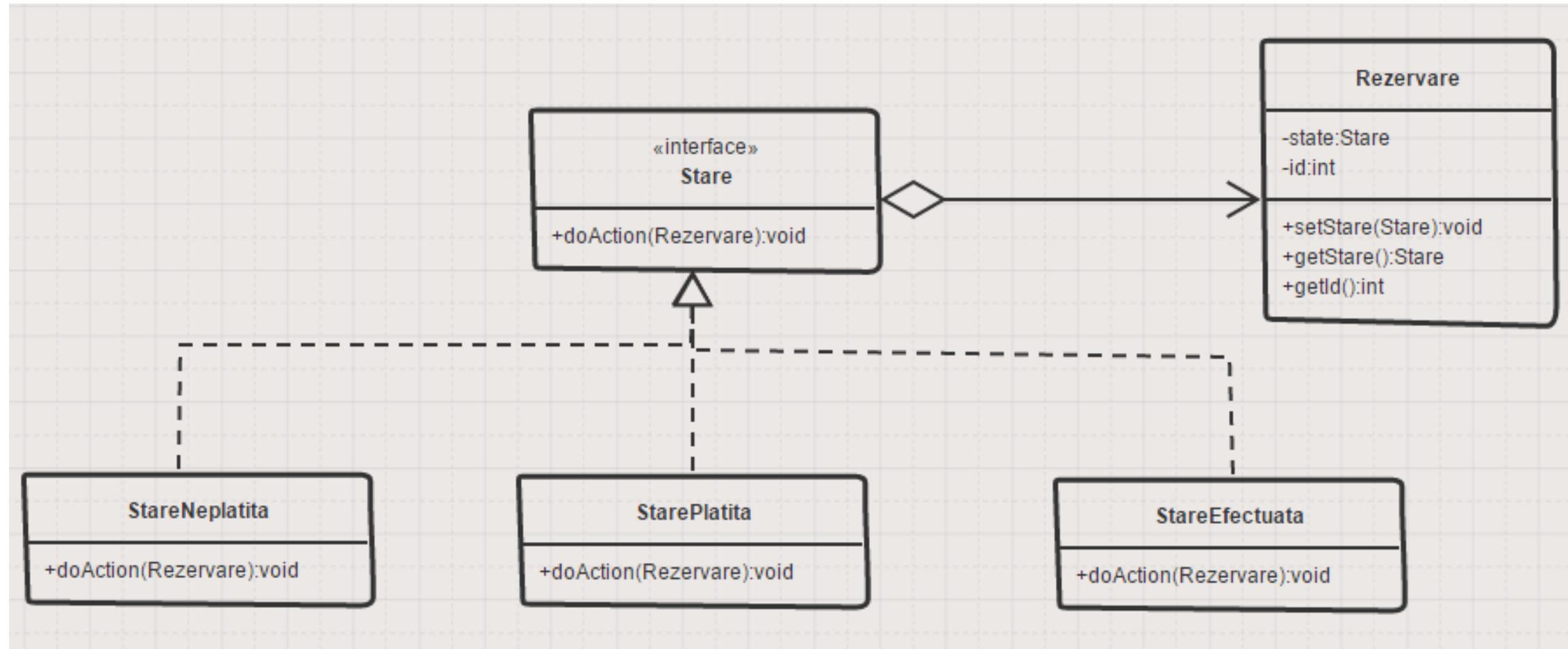
State - Diagramă



State - problemă

- ▶ Agentia de turism dorește implementare a unui modul pentru gestiunea rezervărilor realizate pentru pachetele din oferta sa. Rezervările pot fi într-un din următoarele stări: neplatita, platita, efectuata.
- ▶ Sa se implementeze modulul dorit de către agentia de turism.

State - structură



State – participanți

- ▶ State – **Stare** – interfață sau clasa abstractă care definește la nivel abstract stările în care poate fi un obiect.
- ▶ ConcreteState – **StareNeplatita**, **StarePlatita**, **StareEfectuata** – stările concrete în care poate fi un obiect.
- ▶ Context – **Rezervare** – clasa care definește obiectul care va trece prin stările create.

State - implementare

- ▶ În cadrul fiecărei clase de stare există metoda de setare a stării prin care se modifică starea contextului sau a rezervării, în cazul de față.
- ▶ Modificarea stării nu se face prin setare din programul apelator ca la Strategy, ci prin apelul acestei stări.

State - utilizări

- ▶ Stările rezervărilor sau a comenziilor în orice aplicație.
- ▶ Folosit pentru evitarea utilizării structurii **switch** sau **if-else**.

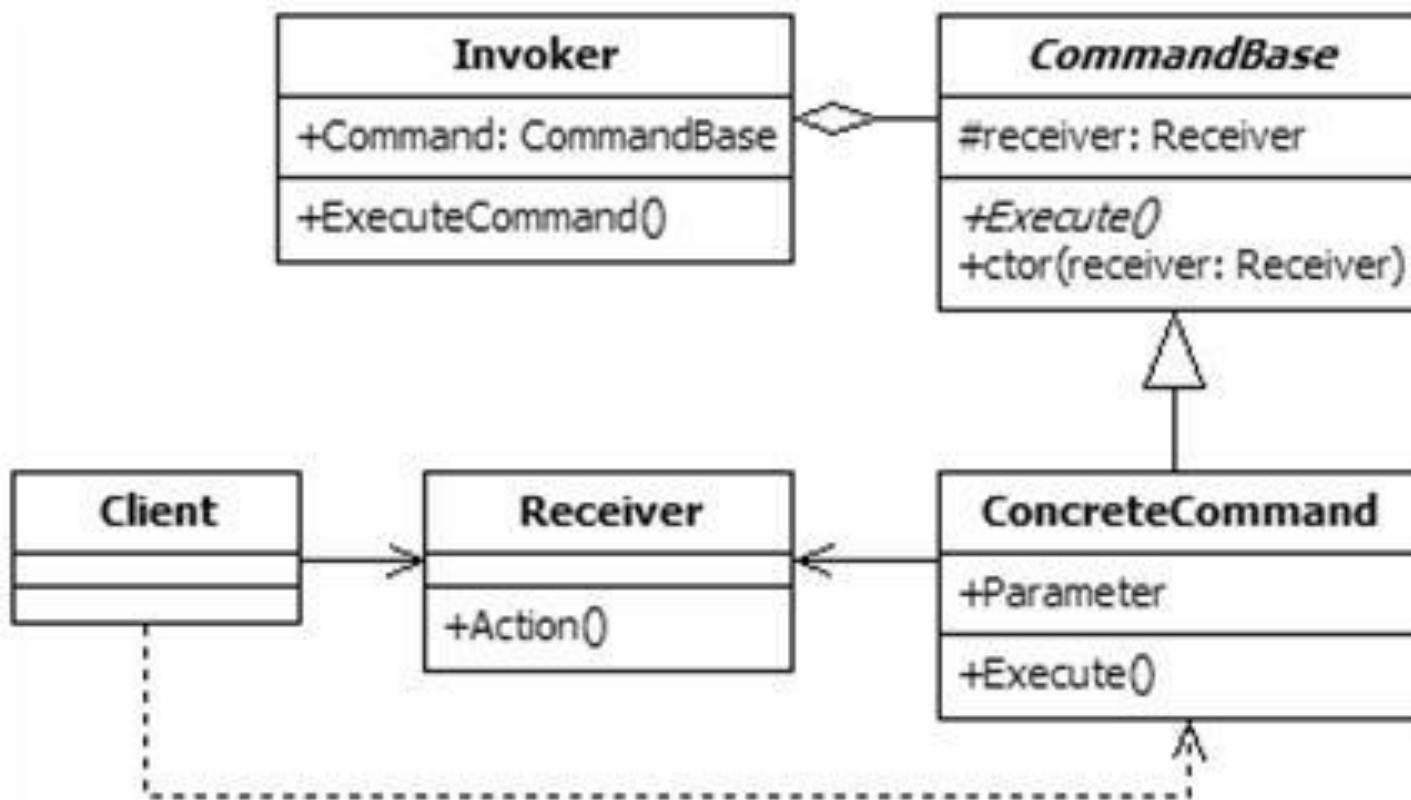
State – corelații

- ▶ Strategy – sunt total asemănătoare
- ▶ Toate design pattern-urile comportamentale.

Command - nume

- ▶ Este folosit pentru implementarea **loose coupling**.
- ▶ În acest mod ascunde aplicarea de comenzi, fără se știe concret ce presupune acea comandă.
- ▶ Astfel clientul este decuplat de cel ce execută acțiunea.

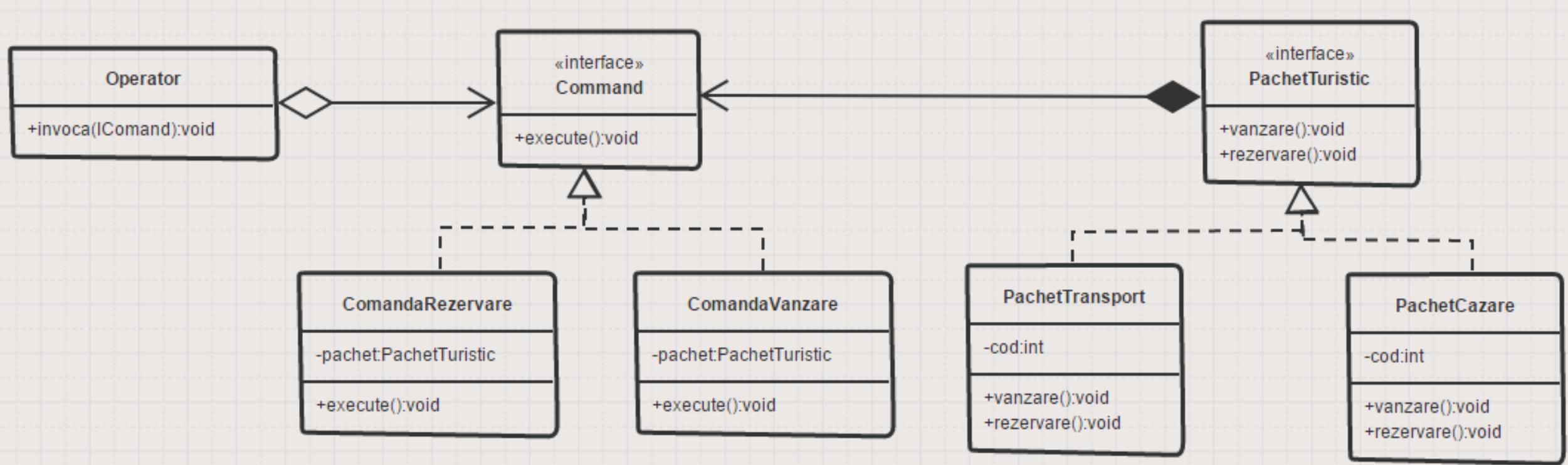
Command - Diagramă



Command - problemă

- ▶ Managerul agenției dorește ca în cadrul aplicației operatorul să poată da comenzi de rezervare sau vânzare pentru pachetele de cazare sau pentru pachetele de transport.
- ▶ Aceste comenzi vor fi realizate prin intermediul clasei Operator.

Command - structură



Command – participanți

- ▶ Command – **Command** – interfață care definește la nivel abstract comenzi sau acțiunile;
- ▶ Comenzi concrete – **ComandaVanzare**, **ComandaRezervare** – clasele concrete pentru fiecare comandă.
- ▶ Receiver – **PachetTuristic** (**PachetCazare**, **PachetTransport**) – obiectul responsabil cu execuția acțiunilor. În cazul de față avem o familie de obiecte.
- ▶ Invoker – **Operator** – Clasa care se ocupă cu gestiunea comenzielor.

Command - implementare

- ▶ Pentru implementare există două situații:
 - ▶ Invoker-ul să fie folosit doar pentru a invoca comenzi;
 - ▶ Invoker-ul poate salva comenzi invocate, și astfel se poate face undo pe comenzi executate sau invocate.
- ▶ Pentru a două situație avem nevoie de o listă de comenzi în Invoker și de metode apelate pe undo() în toate clasele.

Command - utilizări

- ▶ Macro-urile
- ▶ Lucrul cu fișiere
- ▶ Oriunde se dorește revenirea la o stare anterioară prin intermediuul comenziilor

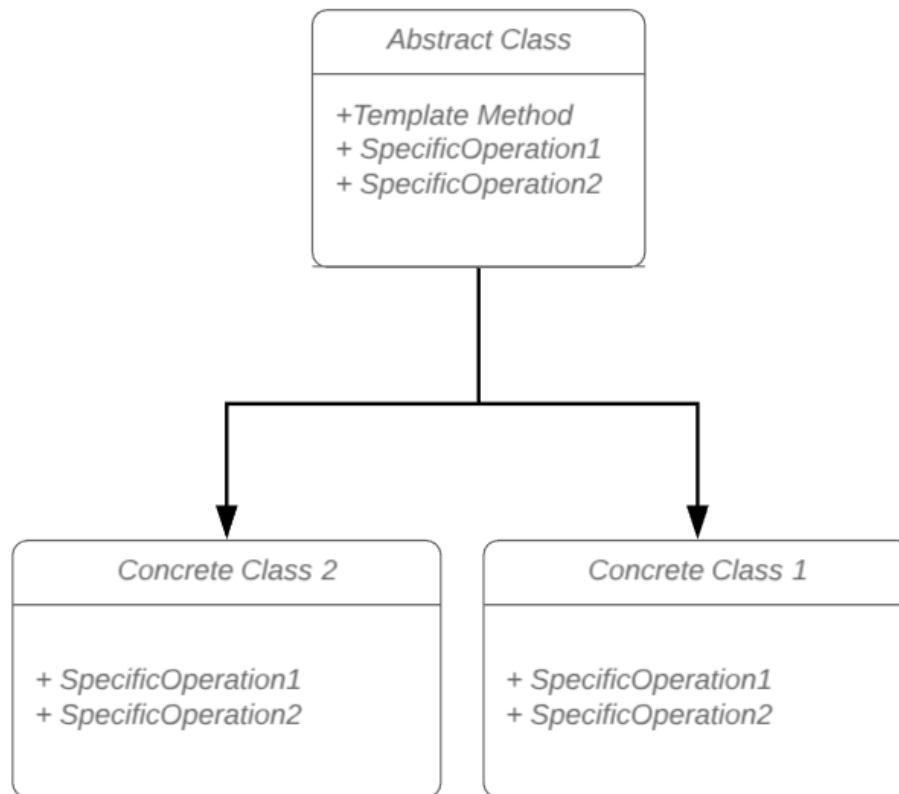
Command – corelații

- ▶ **Memento** – Pentru revenirea la stările anterioare.
- ▶ **Adapter** – se folosește funcționalitatea deja existentă

Template Method - nume

- ▶ Folosit atunci când un algoritm este cunoscut și urmează anumiți pași precisi.
- ▶ Fiecare pas este realizat de căte o metodă.
- ▶ Există o metodă care implementează algoritmul și apelează toate celelalte metode.

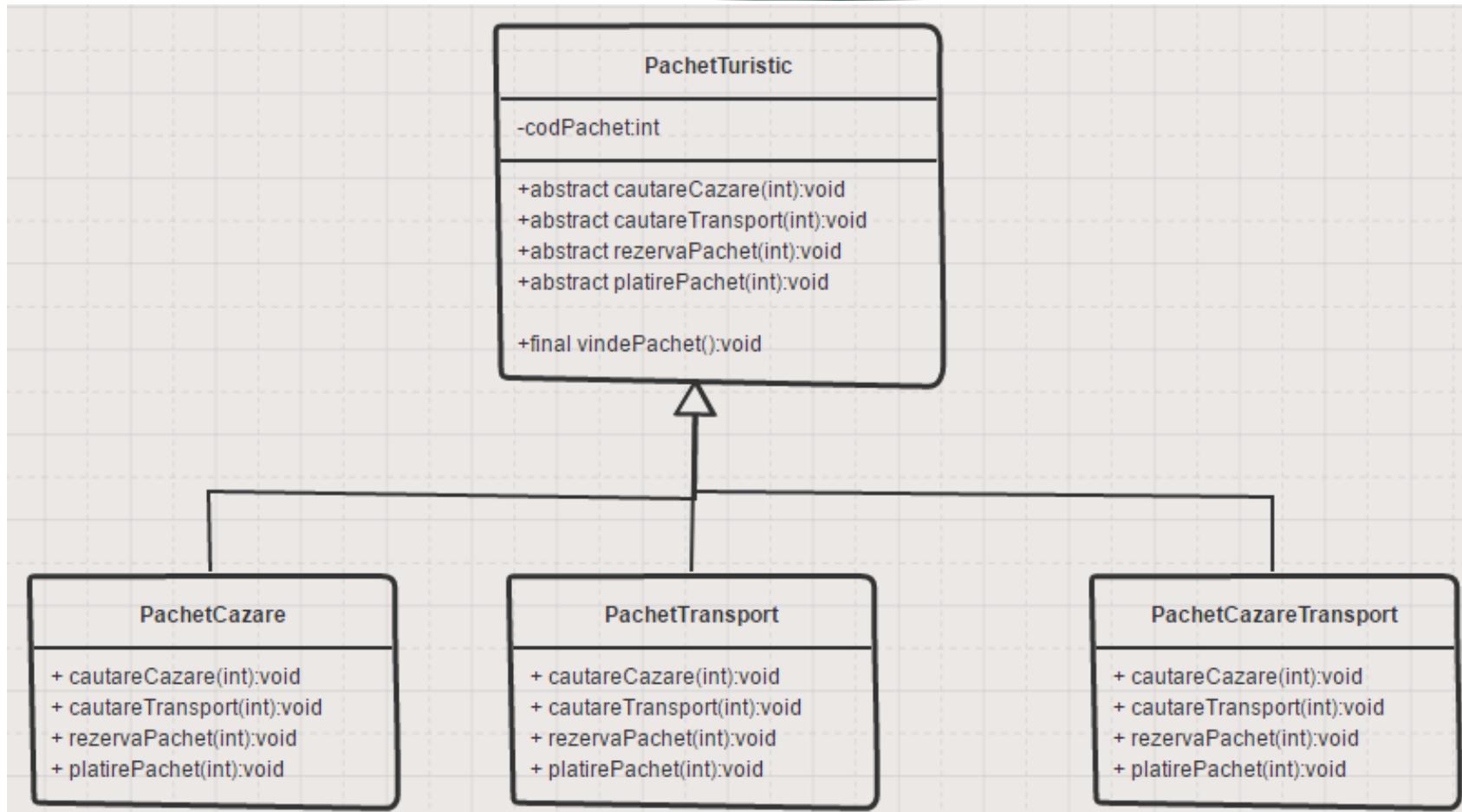
Template Method - Diagramă



Template Method - problemă

- ▶ Vânzarea unui pachet turistic se realizează de fiecare dată după un pattern bine stabilit:
 - ▶ Se caută cazare;
 - ▶ Se caută transport;
 - ▶ Se rezervă întreg pachetul;
 - ▶ Se vinde pachetul, prin realizarea plății.
- ▶ Să se implementeze modulul care realizează vânzarea de pachete turistice.

Template Method - structură



Template Method– participanți

- ▶ Template – **PachetTuristic** – clasa abstractă care implementează metoda template și anunță celelalte metode, care vor fi folosite în cadrul metodei template.
- ▶ ConcreteTemplate – **PachetCazare**, **PachetTransport**, **PachetCazareTransport** – clasele concrete care vor implementa metodele abstractive, determinând astfel modul de realizare a template-ului.

Template Method - implementare

- ▶ În clasa abstractă, metoda template se declară finală, astfel încât să nu poată fi suprascrisă.
- ▶ În cadrul claselor concrete sunt implementate doar metodele folosite în metoda template.

Template Method - utilizări

- ▶ Atunci când modul de procesare sau de rezolvare a unei probleme urmează un număr finit și cunoscut de pași.

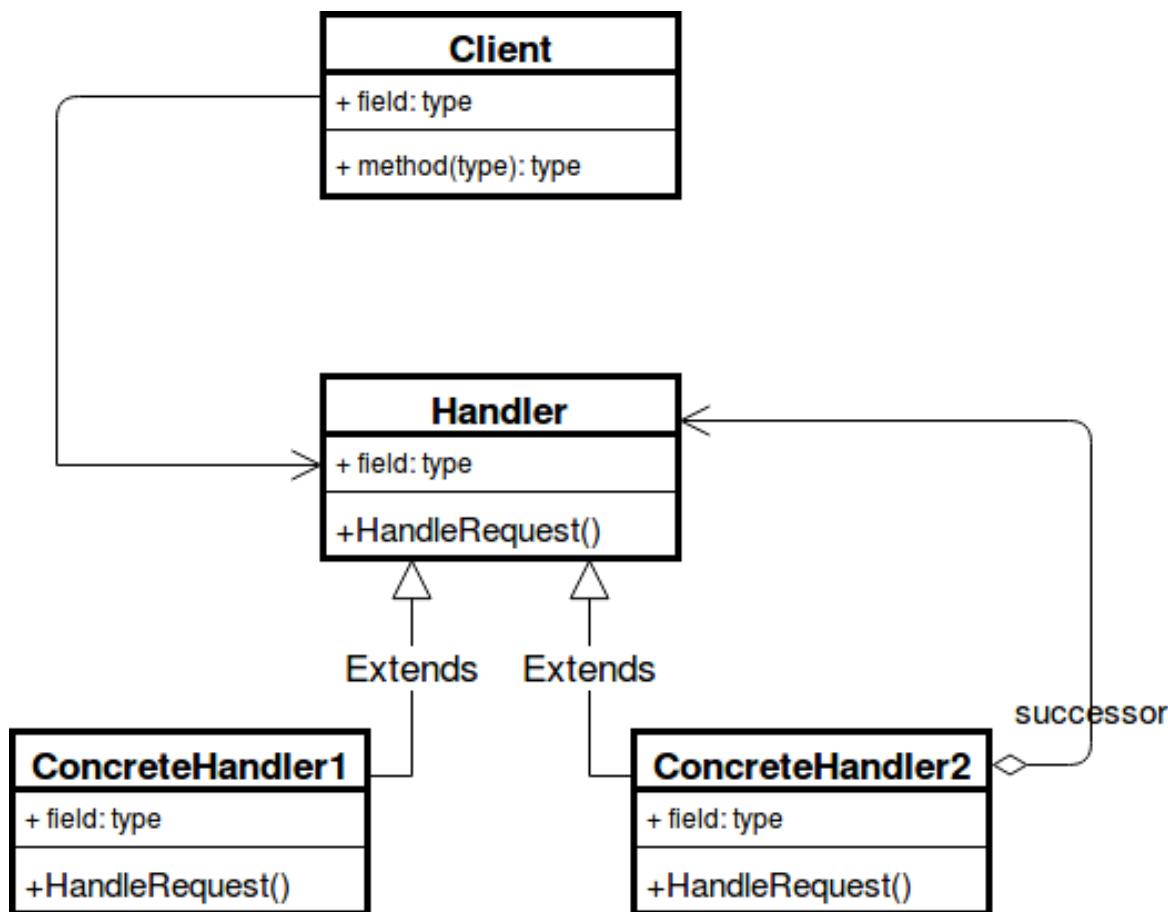
Template Method – corelații

- ▶ **Factory** – avem o familie de obiecte.

Chain of Responsibility - nume

- ▶ Acest design pattern este folosit atunci când cel care are nevoie de rezolvarea unei probleme nu știe exact cine poate să rezolve problema, însă are o listă de posibile obiecte ce pot rezolva problema.
- ▶ Aceste obiecte posibile se ordonează într-un lanț, apoi cel care are problema de rezolvat apelează pentru prima za din lanț.

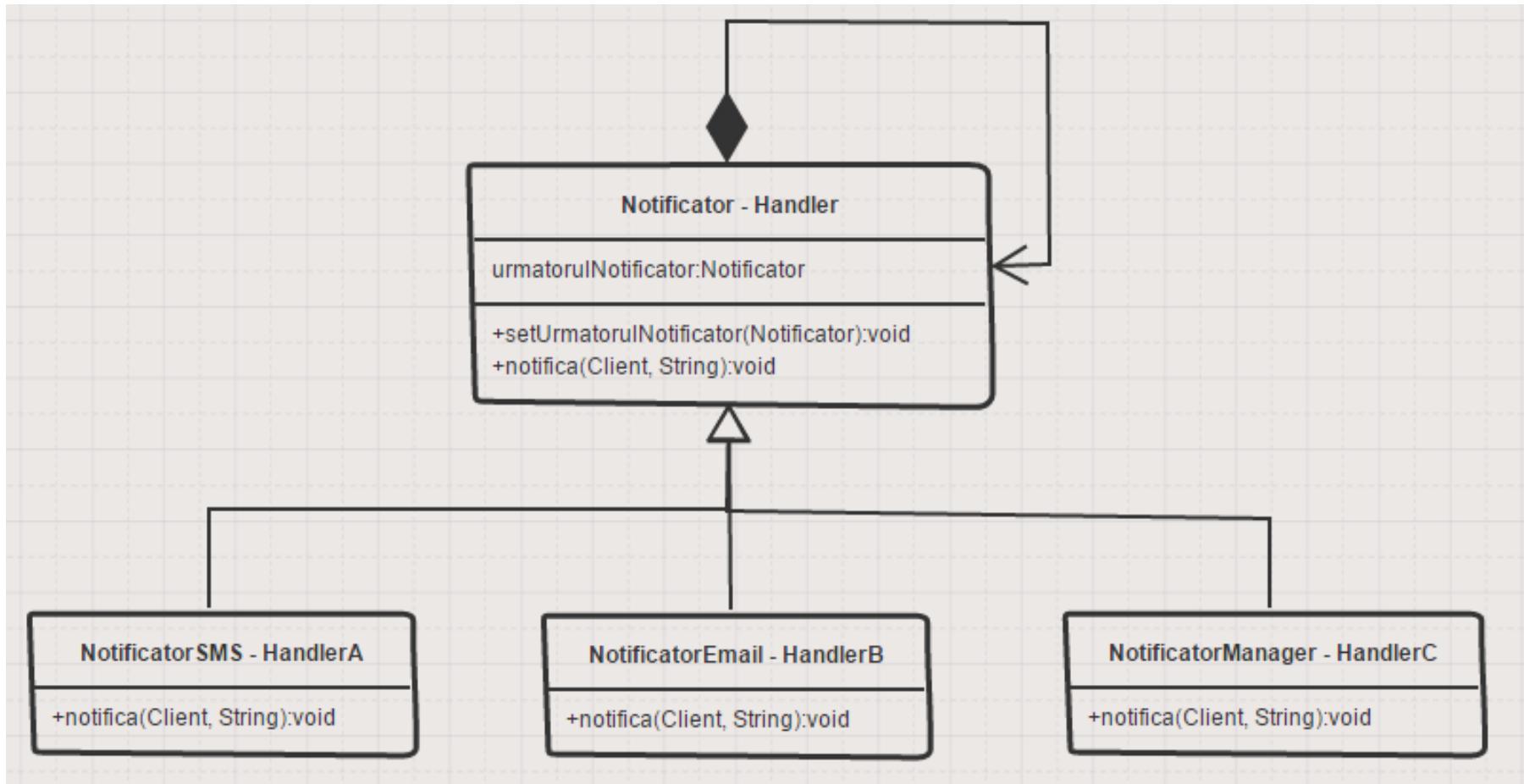
Chain of Responsibility - Diagramă



Chain of Responsibility - problemă

- ▶ Transmiterea notificărilor către clienții fideli se realizează prin mesaje SMS sau prin e-mail.
- ▶ Problema este că agenția deține pentru anumiți clienți numărul de telefon, iar pentru alți clienți doar adresa de mail. Să se implementeze funcționalitatea de a trimite notificări clienților prin SMS, iar în cazul în care pentru anumiți clienți agenția nu are în baza de date numărul de telefon, să se trimită notificarea prin email. În cazul clienților pentru care nu există nici numărul de telefon, nici adresa de mail, se trimite managerului agenției o notificare cu numele clientului pentru care nu există date de contact.

Chain of Responsibility - structură



Chain of Responsibility – participanți

- ▶ Handler – **Notificator** – clasa abstractă sau interfață care definește interfața obiectelor ce vor gestiona cererea de procesare și de rezolvare a problemei.
- ▶ ConcreteHandler – **NotificatorSMS**, **NotificatorEmail**, **NotificatorManager** – clasele concrete ale căror obiecte vor forma lantul.

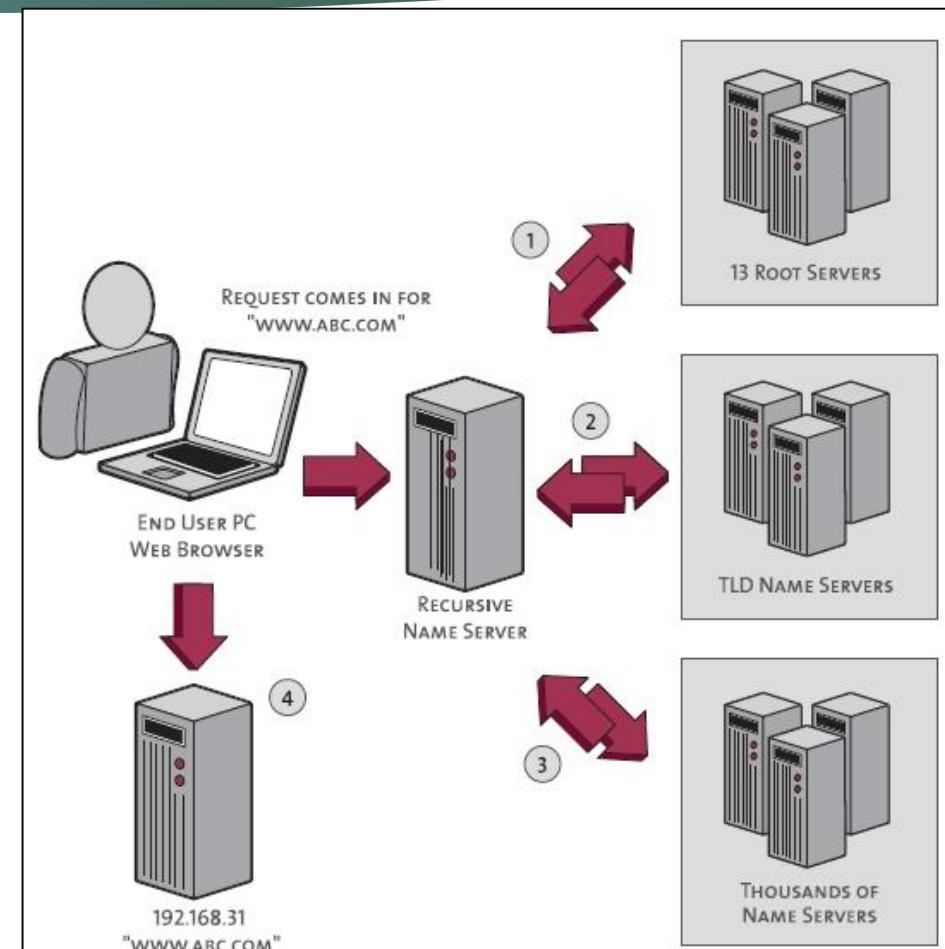
Chain of Responsibility - implementare

- ▶ Gestirea următorului handler se face în clasa abstractă.
- ▶ În cazul în care un handler concrete nu poate rezolva problema apelează următorul handler.
- ▶ Ultimul handler din lanț trebuie implementat astfel încât să nu apeleze la un următor handler, deoarece acesta nu există.

Chain of Responsibility - utilizări

► DNS Resolver

► <https://gieseanw.wordpress.com/2010/03/25/building-a-dns-resolver/>



Chain of Responsibility – corelații

- ▶ **Composite** – asemănătoare la structură – în cazul ambelor design pattern- uri parintele poate acționa ca și successorul.

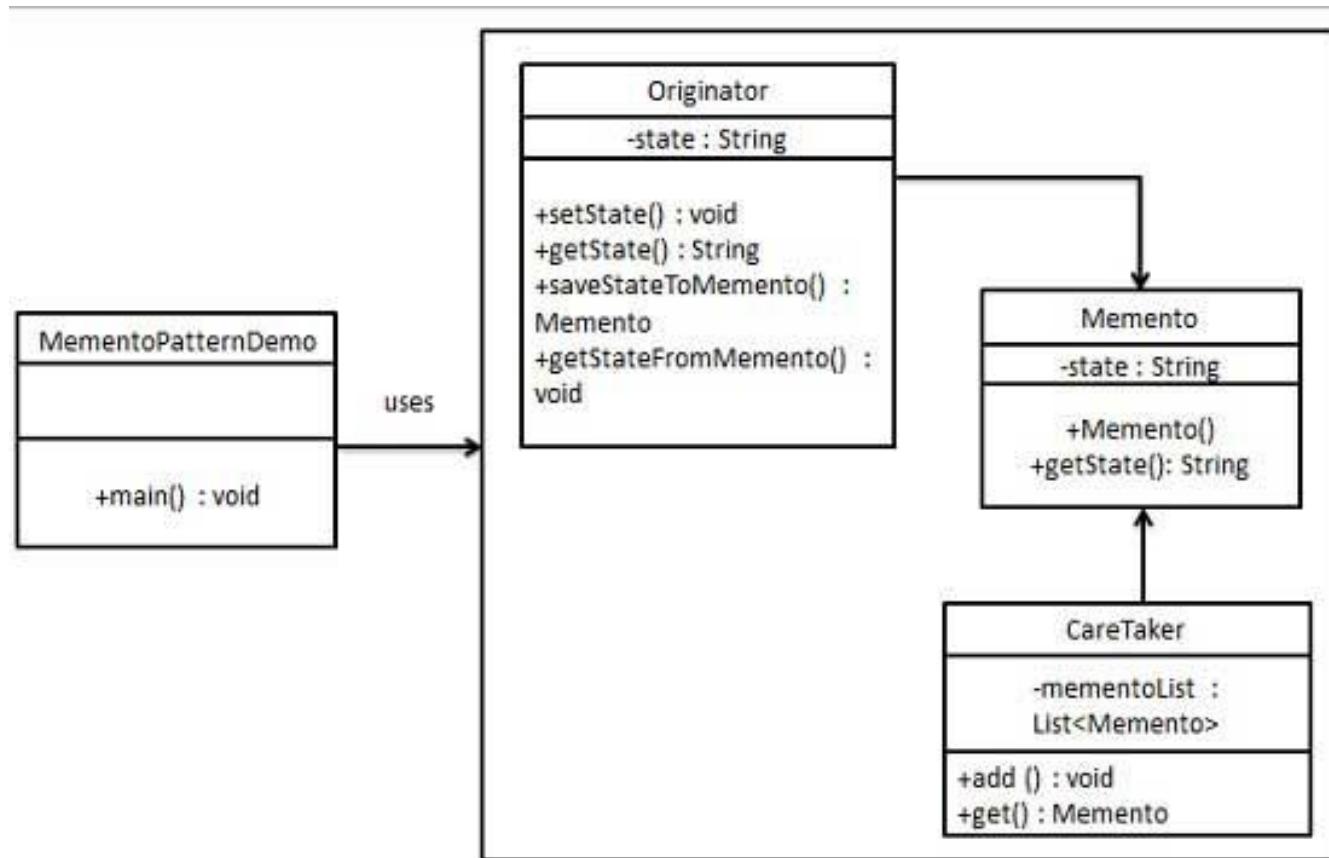
Design patterns

- ▶ Hotelul dorește să anunțe toți clienții abonați la notificări, considerați clienți fideli. Pentru fiecare client hotelul deține numărul de telefon sau/și adresa de mail. Să se implementeze gestiunea clienților fideli și funcționalitatea de a trimite notificări acestora prin SMS, iar în cazul în care pentru anumiți clienți hotelul nu are în baza de date numărul de telefon, să se trimită notificarea prin email. În cazul clienților pentru care nu există nici numărul de telefon, nici adresa de mail, se trimit managerului hotelului o notificare cu numele clientului pentru care nu există date de contact.

Memento - nume

- ▶ Folosit atunci când se dorește salvarea anumitor stări pentru obiectele unei clase.
- ▶ Permite salvarea și revenirea la stările salvate ori de câte ori acest lucru este dorit.
- ▶ Backup.

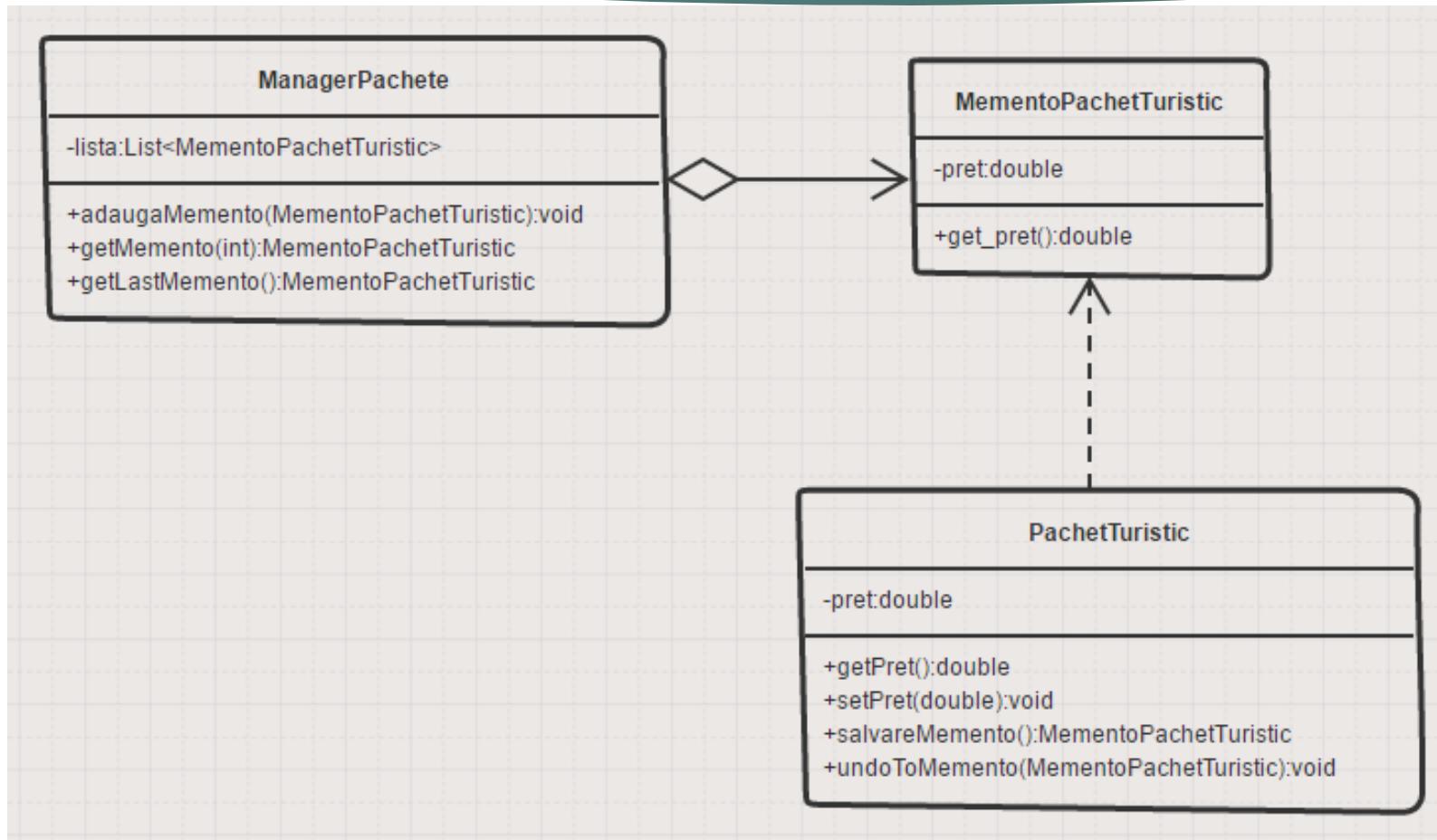
Memento - Diagramă



Memento - problemă

- ▶ Pachetele din oferta agenției își modifică des prețul, în funcție de ofertele zilei sau de cererea din acea zi pentru pachete turistice.
- ▶ Agenția dorește implementarea unui modul prin care aceste prețuri să fie salvate, astfel încât să fie permisă revenirea la un anumit preț folosit anterior.
- ▶ Să se implementeze acest modul

Memento - structură



Memento – participanți

- ▶ Memento – **MementoPachetTuristic** – clasa care gestionează starea internă a obiectului. Clasa care realizează imaginile sau stările intermediiare.
- ▶ Originator – **PachetTuristic** – Clasa care are obiecte pentru care se vor salva stări intermediiare. Salvarea stărilor se realizează în cadrul acestei clase.
- ▶ CareTaker – **ManagerPachete** – Clasa care gestionează obiectele de tip Memento.

Memento - implementare

- ▶ Clasa **Memento** poate fi una externă și se ocupă de atributele pentru care trebuie realizată imaginea intermediară.
- ▶ Deoarece această clasă este folosită doar de către obiectele de tip PachetTuristic, clasa **Memento** poate fi inclusă în cadrul clasei **PachetTuristic**.

Memento - utilizări

- ▶ Salvarea fișierelor
- ▶ Realizarea de backup-uri.

Memento – corelații

- ▶ **Command** - Pentru revenirea la stările anterioare.

Recapitulare

- ▶ Design pattern-uri Creaționale:
 - ▶ **Singleton** – se poate crea o singură instanță pentru o clasă, are constructorul privat;
 - ▶ **Factory** – crează obiecte dintr-o familie de clase. Simple Factory, Factory Method, Abstract Factory;
 - ▶ **Builder** – ajută la crearea obiectelor complexe cu foarte multe atribute;
 - ▶ **Prototype** – folosit atunci când crearea unui obiect consumă foarte multe resurse. Se creează un prototip și este folosit pentru clonare.

Recapitulare

- ▶ Design pattern-uri Structurale:
 - ▶ **Adapter** – adaptează un framework, astfel încât să lucreze cu un alt framework. Nu adaugă funcționalitate;
 - ▶ **Facade** – simplifică lucrul cu framework-uri complexe;
 - ▶ **Decorator** – adaugă noi funcționalități la run-time;
 - ▶ **Composite** – folosit pentru crearea și gestiunea ierarhiilor;
 - ▶ **Flyweight** – gestionează eficient obiectele pentru o utilizare optimă a memoriei;
 - ▶ **Proxy** – controlarea comportamentului și impunerea de condiții;

Recapitulare

- ▶ Design pattern-uri Comportamentale:
 - ▶ **Strategy** – schimbă la run-time metoda apelată;
 - ▶ **Observer** – când un subiect își schimbă starea – n observatori sunt notificați;
 - ▶ **Chain of Responsibility** – se formează un lanț pentru rezolvarea unei probleme. În cazul în care o verigă a lanțului nu poate rezolva problema, apelează la veriga succesoare;
 - ▶ **State** – schimbă comportamentul pe baza stării în care se află obiectul;
 - ▶ **Command** – gestiunea de comenzi aplicate asupra obiectelor;
 - ▶ **Template Method** – pentru gestiunea unui pattern de pași;
 - ▶ **Memento** – salvarea și gestiunea stărilor anterioare ale obiectului;

Behavioral Design Patterns

