Creational design patterns

ALIN ZAMFIROIU

Design Patterns

- **Margaret Heafield**

Ce sunt design patterns?

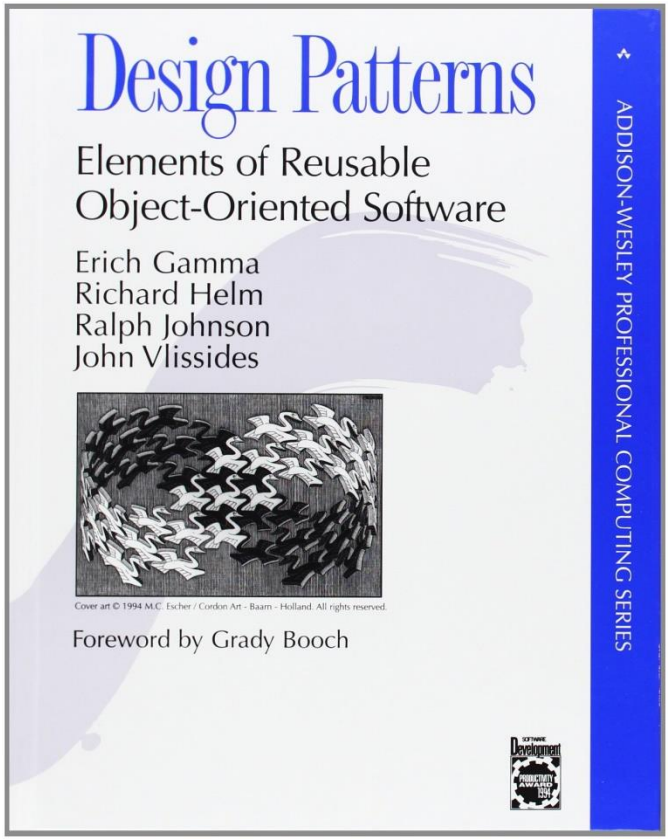
- O traducere în limba română ar fi: **șabloane de proiectare** sau **tipare de proiectare** și au scopul de a ajuta în rezolvarea unor probleme similare cu alte probleme pentru care au fost deja identificate rezolvări.

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.

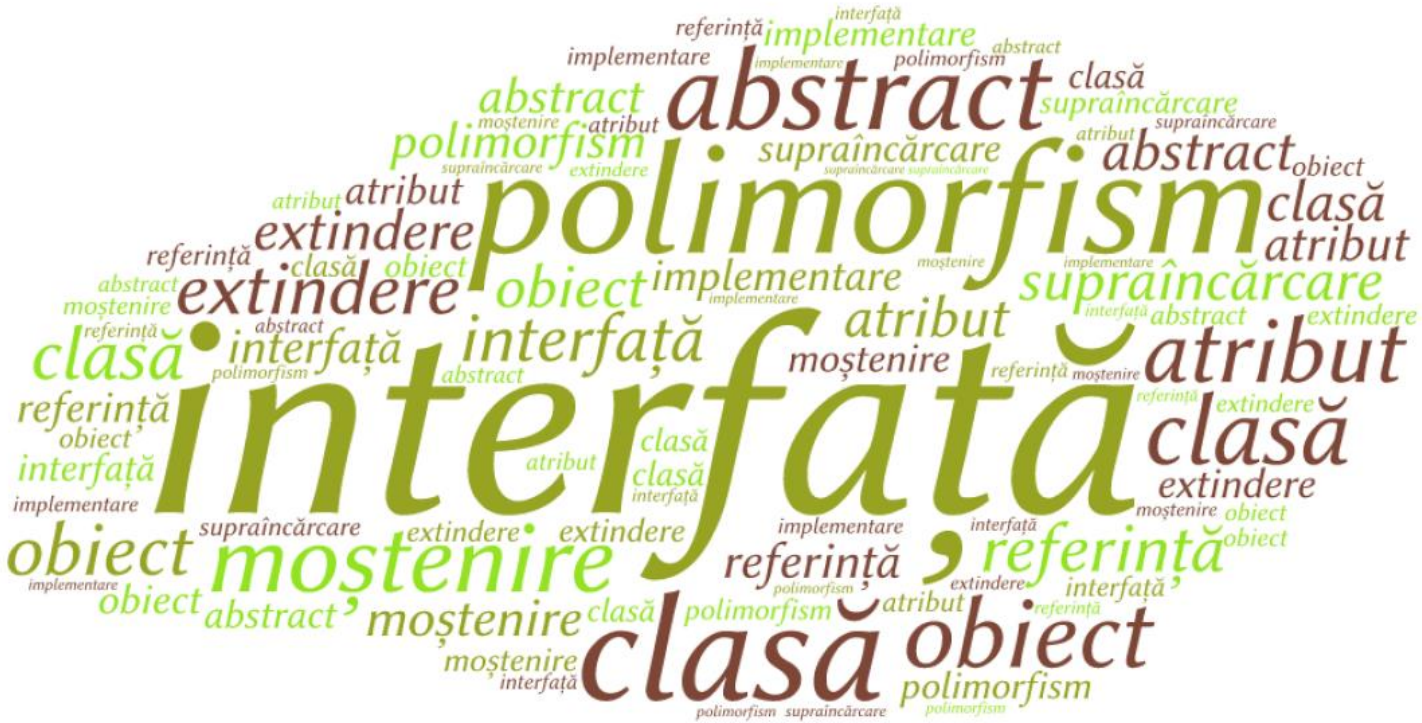
Design patterns - Gof

- 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.



https://tagul.com/create

GoF

- Cartea este împărțită în două părți:
  - Design patterns (2);
  - Catalog (3).

- Implementările prezentate sunt realizate în limbajul C++.

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 vocabular 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;
  - Factory;
  - Factory Method;
  - Abstract Factory;
  - Builder;
  - Prototype.

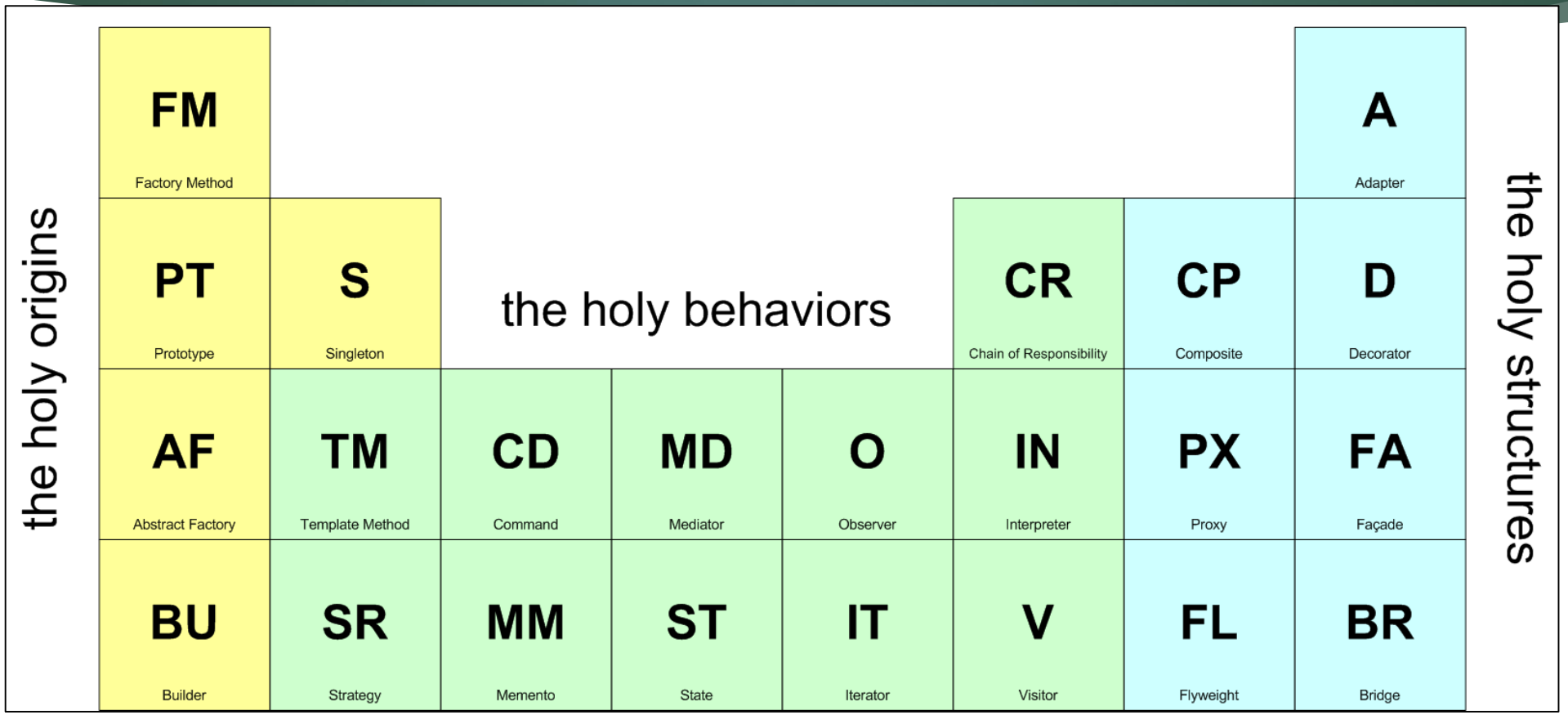
Tipuri de design patterns – structurale

- 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 Responsability;
  - Command;
  - Iterator;
  - Interpreter;
  - Mediator;
  - Memento;
  - Observer;
  - State;
  - Strategy;
  - Visitor;
  - Template.

Tipuri de design patterns



Tipuri de design patterns – creaționale

- Ajută la inițializarea și configurarea claselor și obiectelor;
- Design paternurile 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 creeat;
  - Când este creeat.

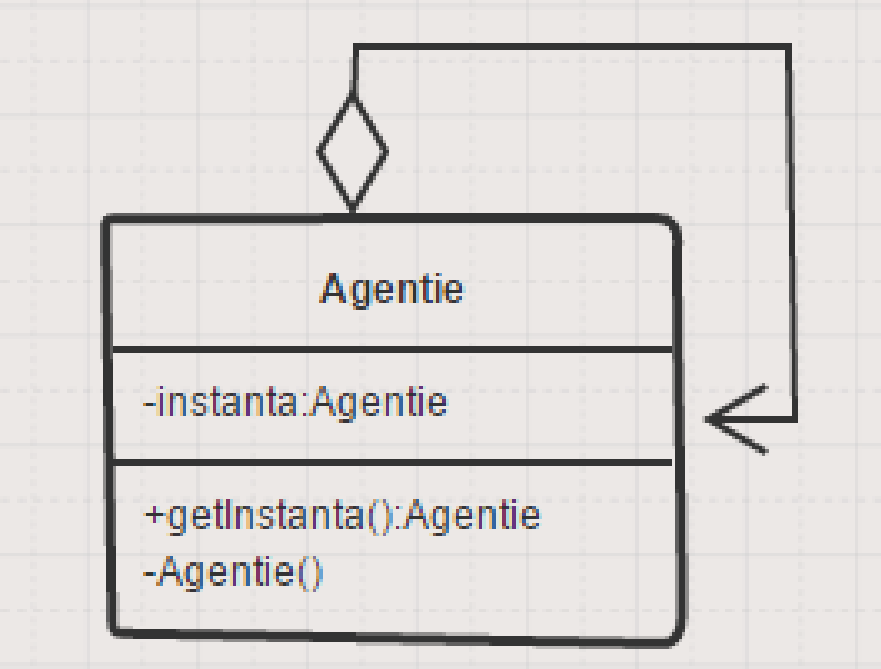
Singleton - nume

- 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 – 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 private, 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 initilization
- Thread Safe Singleton
- Inner static helper class
- Enum singleton
- Serializarea singletonului
- Singleton vs clasă statică

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

- Această 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

- Această variantă a fost propusă de **Joshua Bloch** și 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 doua instanțe: una creată și una deserializată.
- Pentur evitarea acestui aspect este necesară implementarea metodei **readResolve().**
- Această metodă trebuie să returneze tot instanță creată în cadrul singletonului.

Singleton vs clasă statică

- Singleton respecta 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 - utilizări

- Deschiderea unei singure instanțe ale unei aplicații.
- Conexiune unică la baza de date.
- SharedPreferences în Android.

Singleton – corelații

- **Factory** – o singură fabrică de o 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 obiete** sau **obiecte din aceeași familie**.

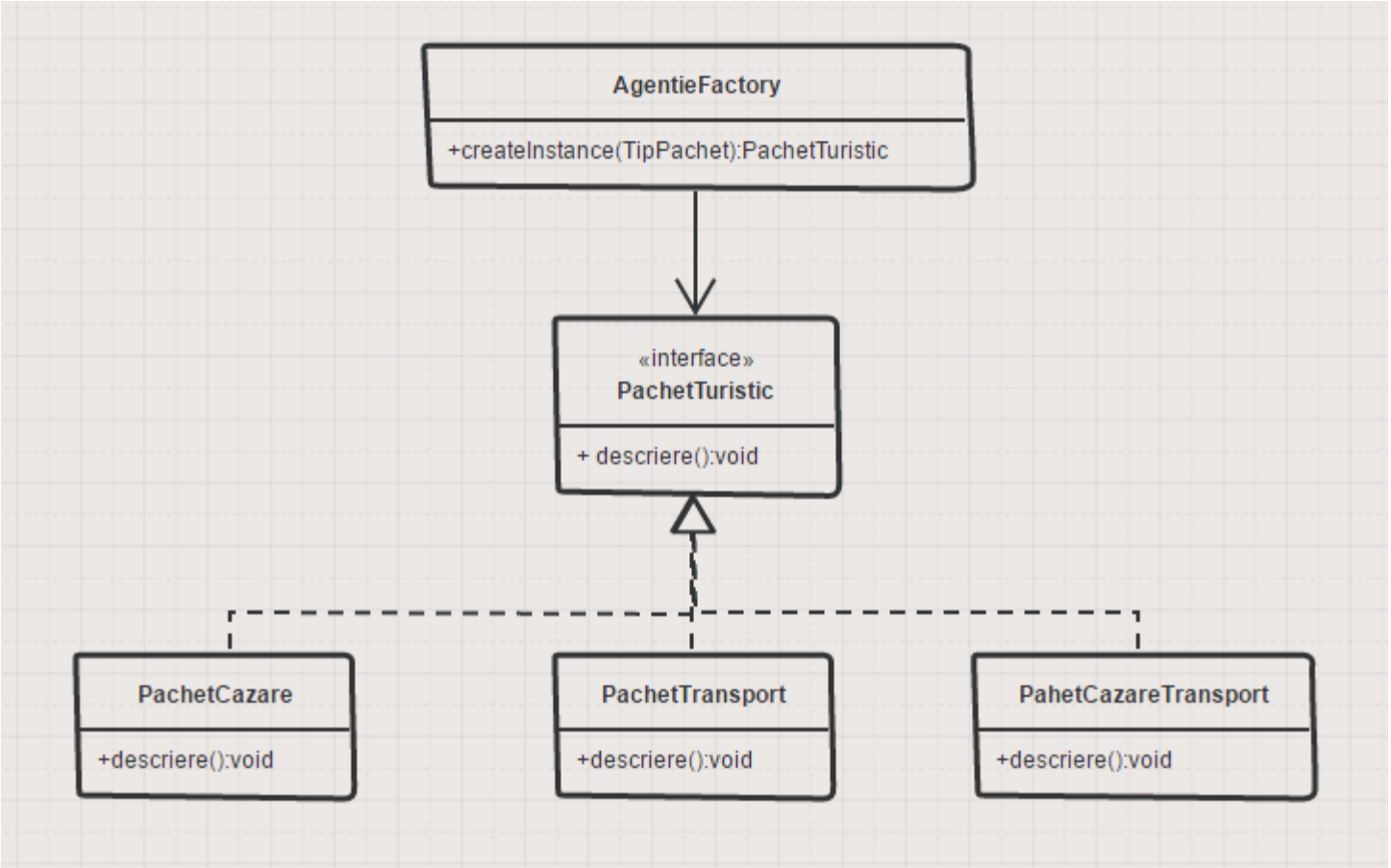
- Simple factory este un design pattern care nu este prezentat în cartea GoF, însă este folosit în practică deoarece este foarte ușor de implementat.

Factory - problemă

- Agenția de turism **AgeTur** oferă pahete turistice cu cazare și transport însă are în ofertă și 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 agenția **AgeTur**.

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 nu mai folosește *enum* ci abstractizează nivelul de creare
- Se mai numește și **Virtual Constructor**.

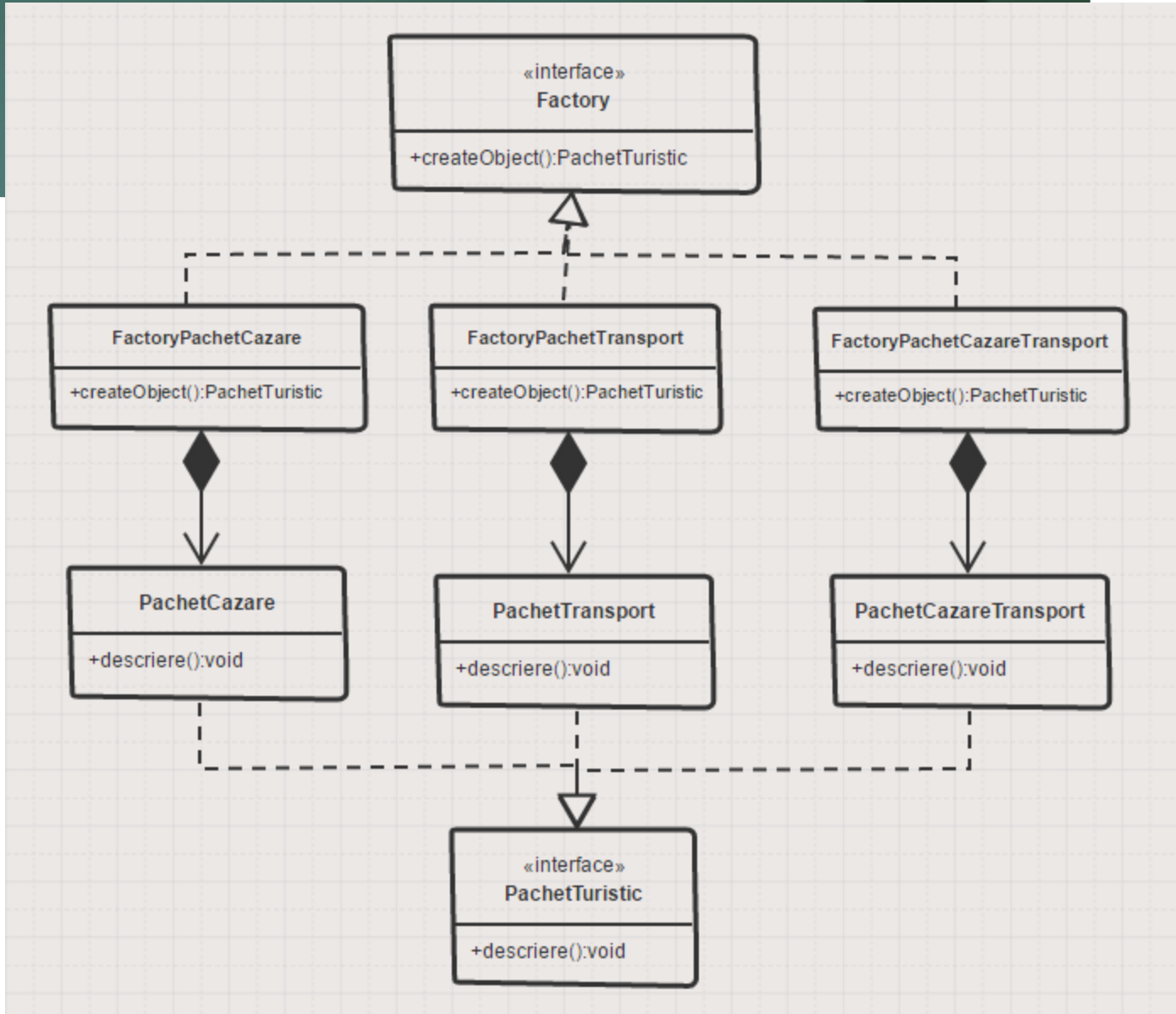
Factory Method - problemă

- 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. Să nu se utilizeze tipuri enum.

Factory Method

- **Structură**



Factory Method – participanți

- Interfața **PachetTuristic** – definește la nivel abstract obiectele ce pot fi create;
- Clasele concrete **PachetCazare**, **PachetTransport**, **PachetCazareTransport** – implementează interfața *PachetTuristic;*

- Interfața Factory – definește la nivel abstract generatoarele de obiecte
- Clasele concrete **FactoryPachetCazare**, **FactoryPachetTransport**, **FactoryPachetCazareTransport** – implementează interfața *Factory*, iar metoda *createObject()* va returna instanțe ale claselor concrete *PAchetCazare*, *PachetTransport*, *PachetCazareTransport*

Factory Method - implementare

- Pentru Factory Method nu mai avem nevoie de switch sau if-else, deoarece fiecare fabrica va produce doar un anumit tip de obiect și îl va returna

- Pentru apel se folosește abstractizări ci nu obiectele concrete.

Abstract Factory - nume

- **Abstract Factory** introduce un nou nivel de abstractizare.

- Fiecare fabrica de obiecte creează obiecte din doua sau mai multe familii de obiecte

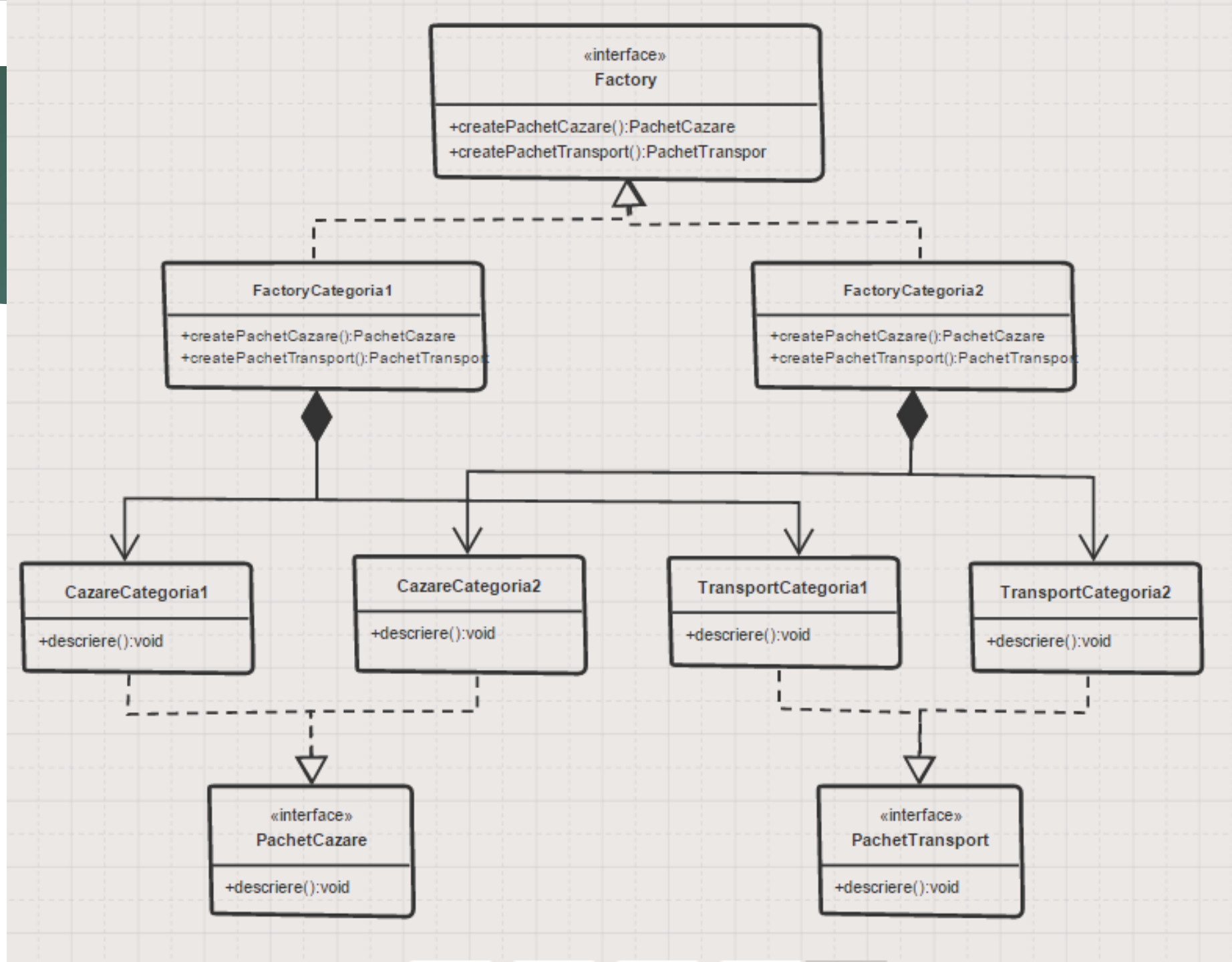
Abstract Factory - problemă

- Agenția de turism **AgeTur** oferă pachete de cazare și pachete de transport. Pentru cazare agenția are în ofertă un pachet ieftin la hotel de 3 stele și un pachet scump la hotel de 5 stele. Pentru transport, agenția are în ofertă pachet ieftin cu autocarul și pachet scump cu avionul.

- Să se implementeze modulul de vânzare de pachete de cazare și transport pentru agenția **AgeTur**.

Abstract Factory

- **Structură**



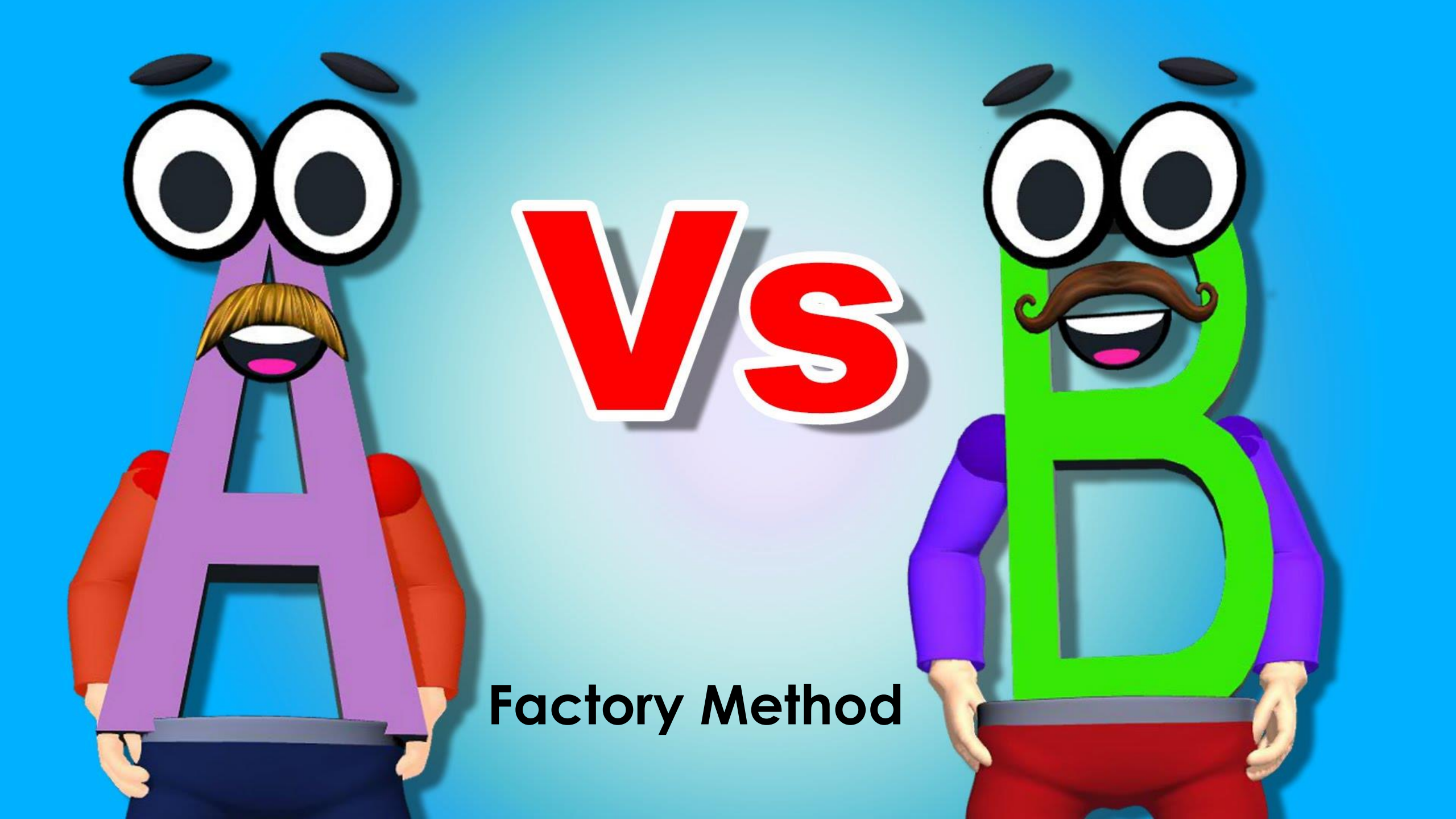
Abstract Factory – participanți

- 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.


Vs
Factory Method

Builder

WEBSITE BUILDER

Add A Theme

Builder - nume

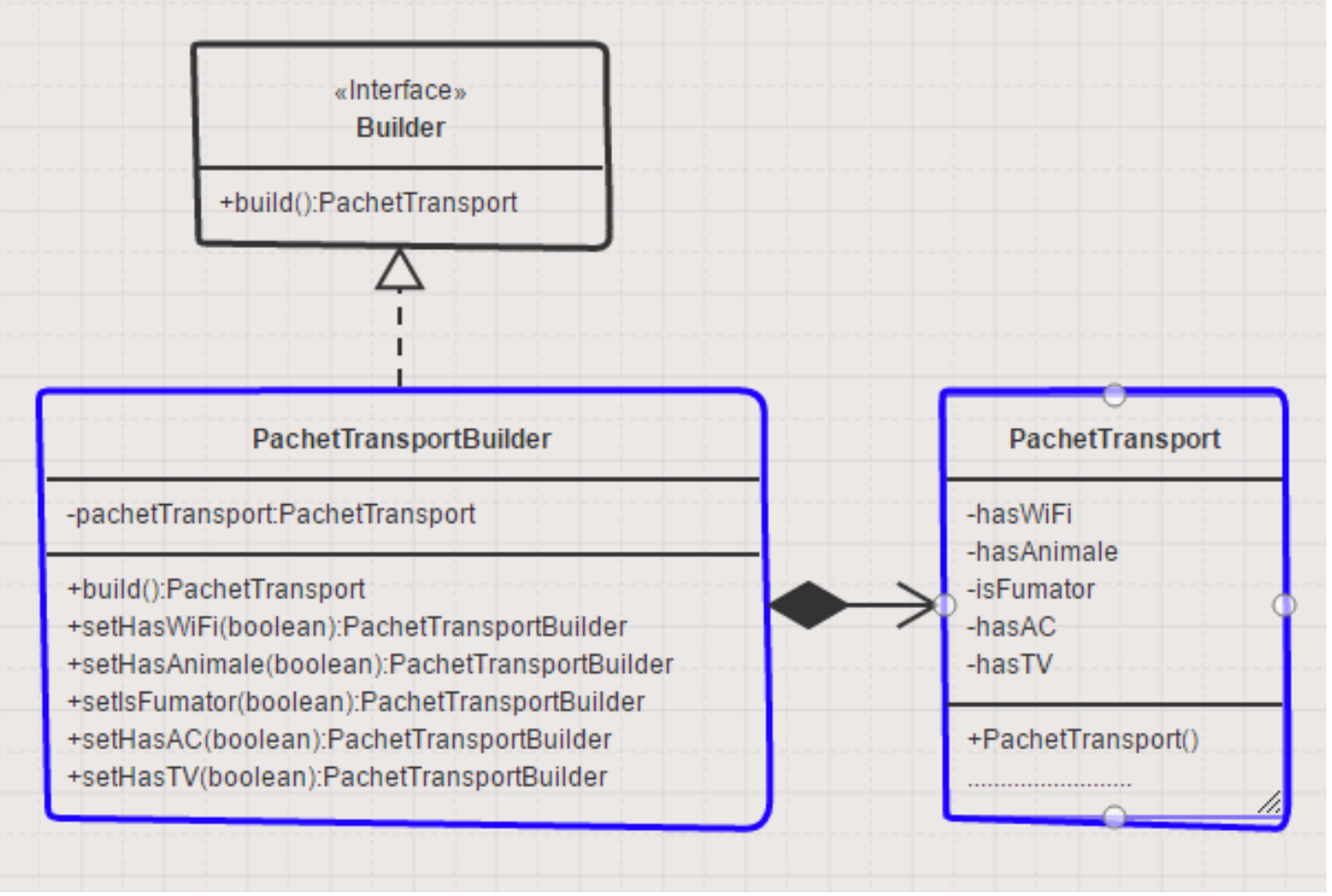
- Ajută la crearea de obiecte concrete.
- Numele vine de la faptul că ajută la construirea de obiecte (build).
- Se construiesc obiecte complexe prin specificarea anumitor proprietăți dorite din multitudinea existentă.

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 agenția **AgeTur**.

Builder - structură



Builder – participanți

- **AbstractBuilder** – interfața care definește metoda build pentru construirea obiectelor complexe.
- **Builder** – clasa concretă care implementează interfața ci 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 - utilizări

- În general pentru construirea de obiecte complexe cu foarte multe atribute.
- StringBuilder

Builder – corelații

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

Prototype - nume

- Ajută la crearea de clone pentru obiectele a căror construire durează foarte mult sau consumă foarte multe resurse.

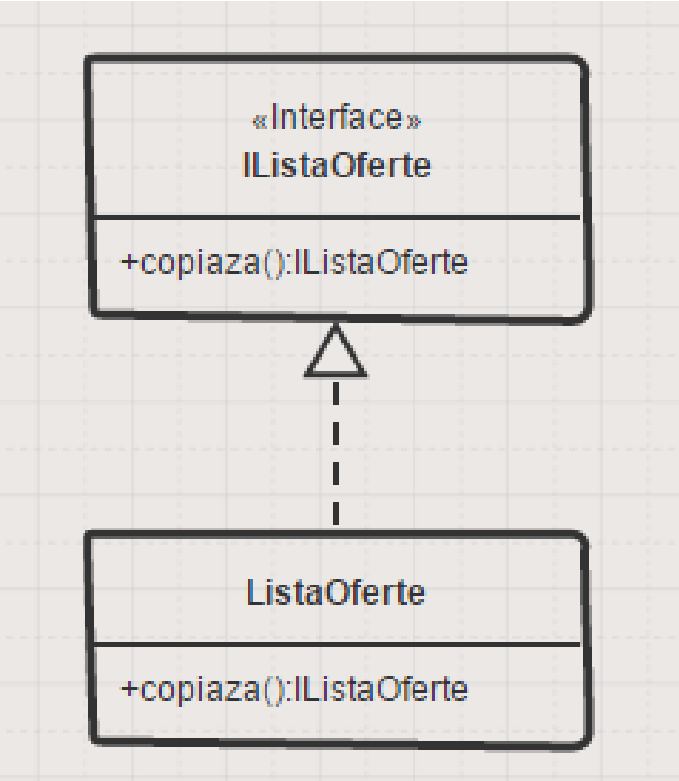
- Prin intermediul acestui design pattern se creează un obiect considerat prototip. Acest prototip urmând a fi clonat pentru următoarele instanțe din acea clasă.

Prototype - 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 se creează un obiect de tipul ListaOferte. La crearea obiectului se citesc din baza de date toate ofertele.

- Să se găsească o soluție eficientă prin care listele de oferte să fie încărcate mai repede.

Prototype - structură



Prototype – participanți

- **Prototype** (IListaOferte) – interfața 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()

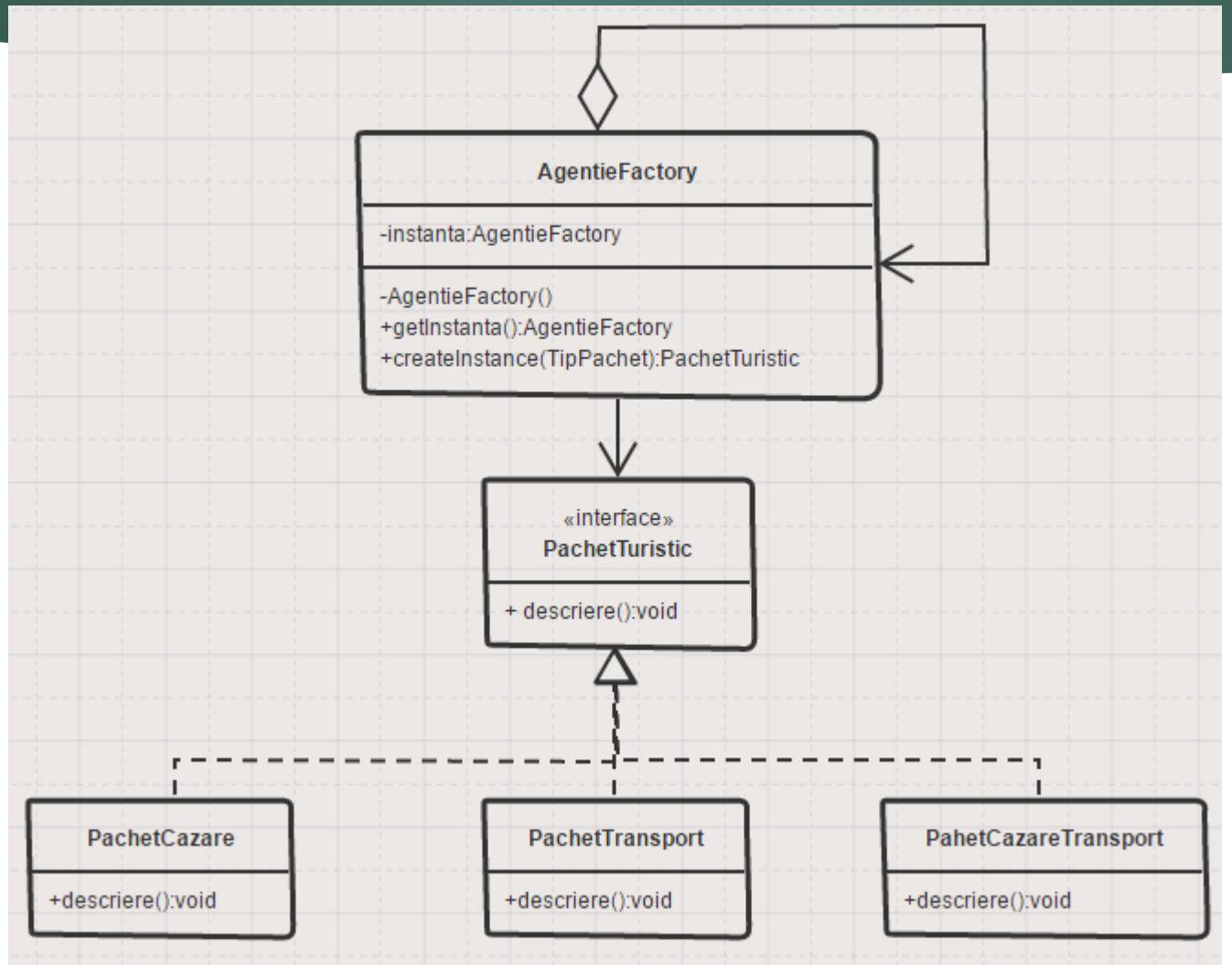
Prototype – corelații

- **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.

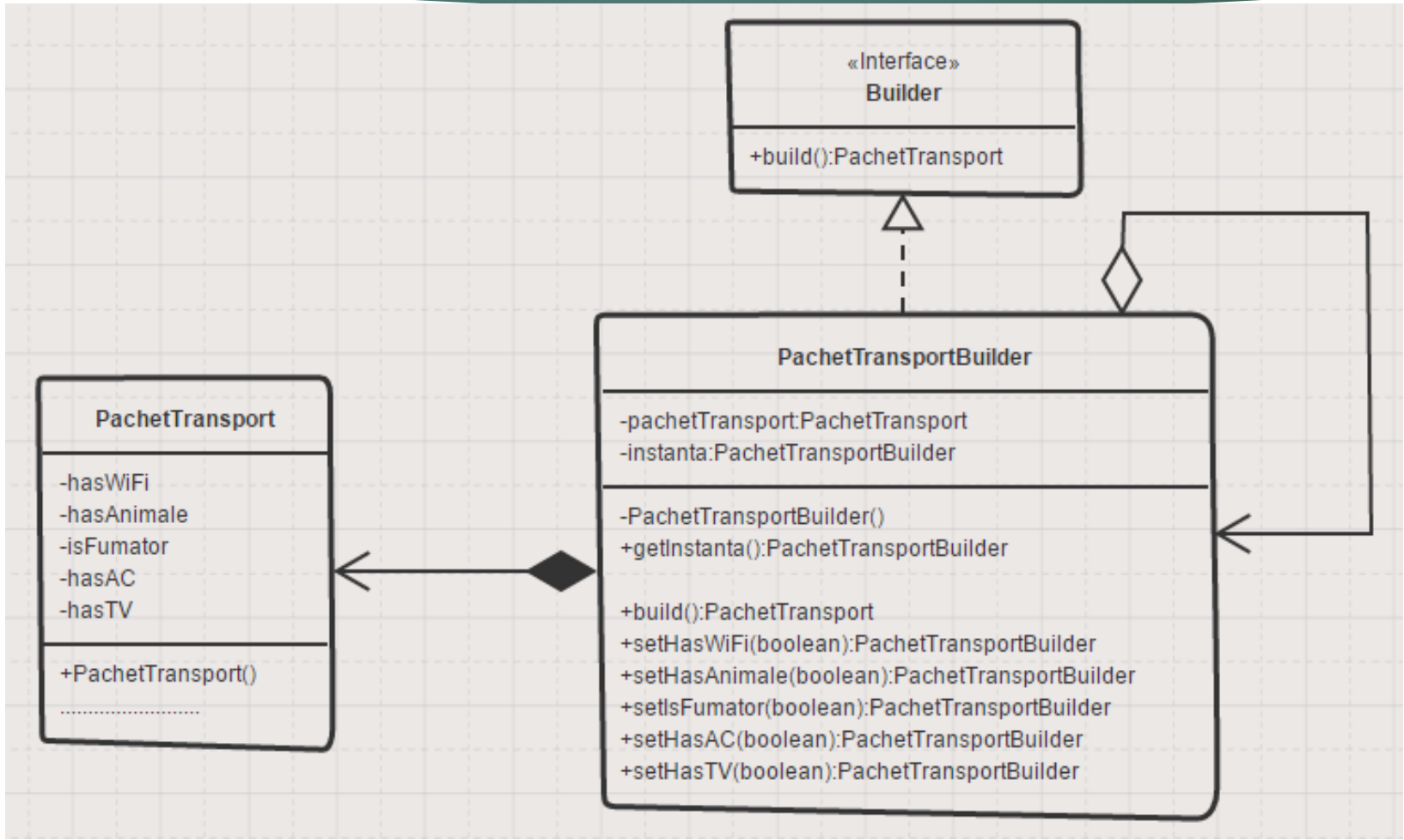
Prototype - extra

- 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.

Extra – Singleton - Factory



Extra – Singleton - Builder



Creational Design Patterns