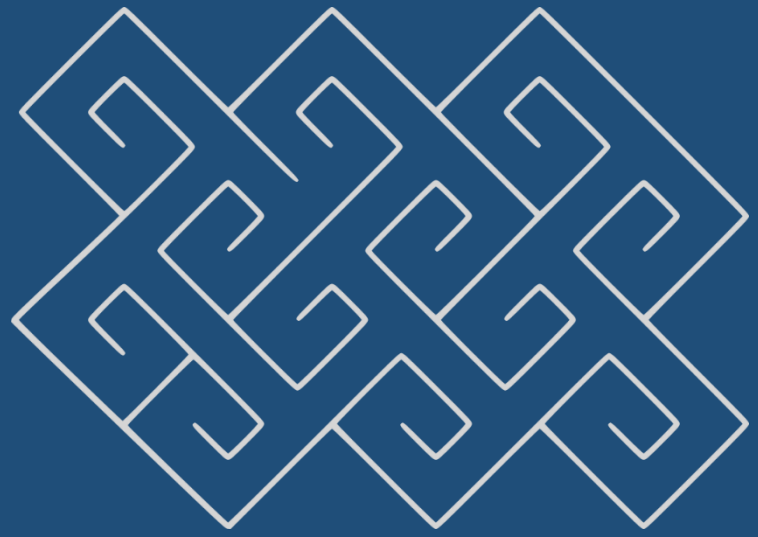


Design patterns

Conf. univ. dr. Catalin Boja

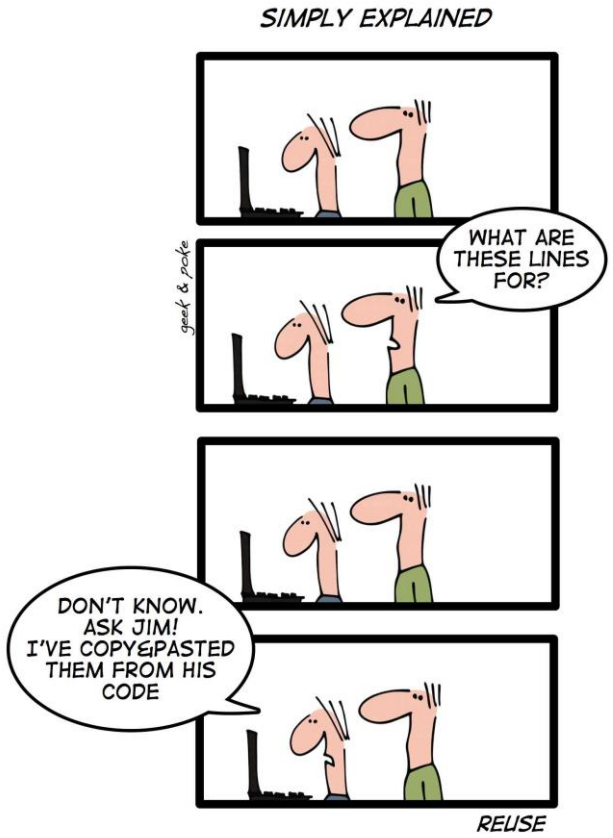
catalin.boja@ie.ase.ro

http://acs.ase.ro/software-quality-testing

Calitate cod sursă

Principii urmărite în scrierea codului:

- Ușor de citit/înțeles – clar
- Ușor de modificat – structurat
- Ușor de reutilizat
- Simplu (complexitate)
- Ușor de testat
- Implementează pattern-uri pentru problem standard



Left: Simply Explained: Code Reuse 2009-12-03.By Oliver Widder, Webcomics Geek Aad Poke.

Calitate cod sursă

Forte care o influențează:

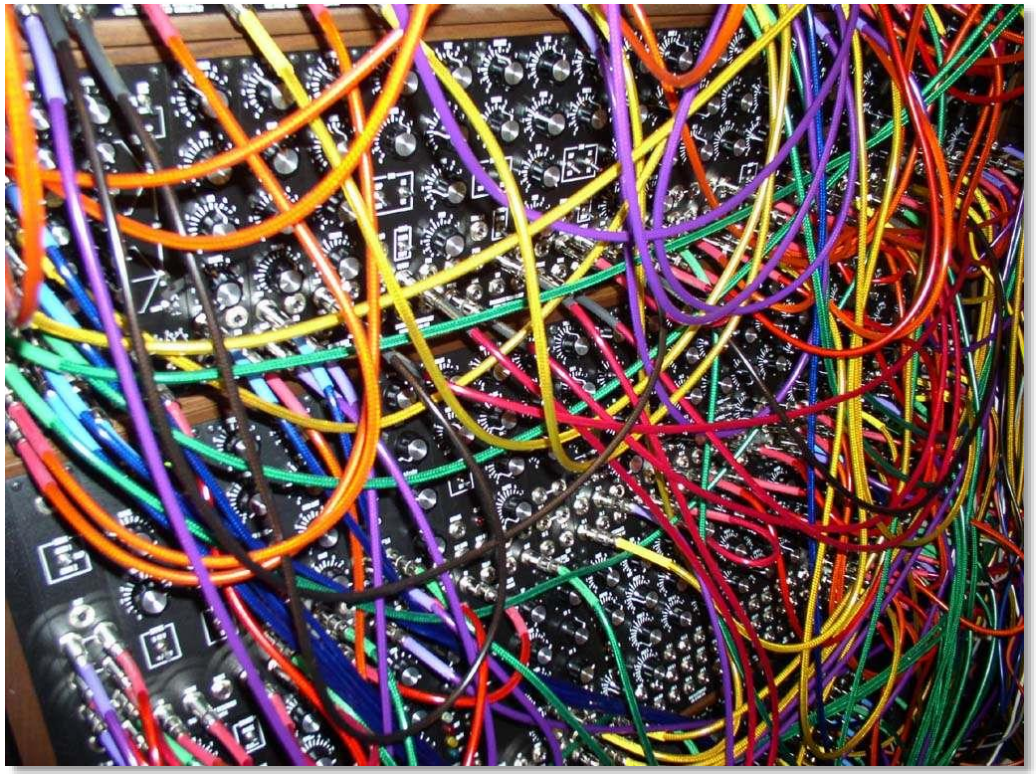
- Timpul disponibil (termene de predare)
- Costuri
- Experiența programatorului
- Competențele programatorului
- Claritate specificații
- Complexitate soluție
- Rata schimbări in specificații, cerințe, echipa, etc

http://khristianmcfadyen.com/

Anti-Pattern: Big ball of mud

"A Big Ball of Mud is a haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle."

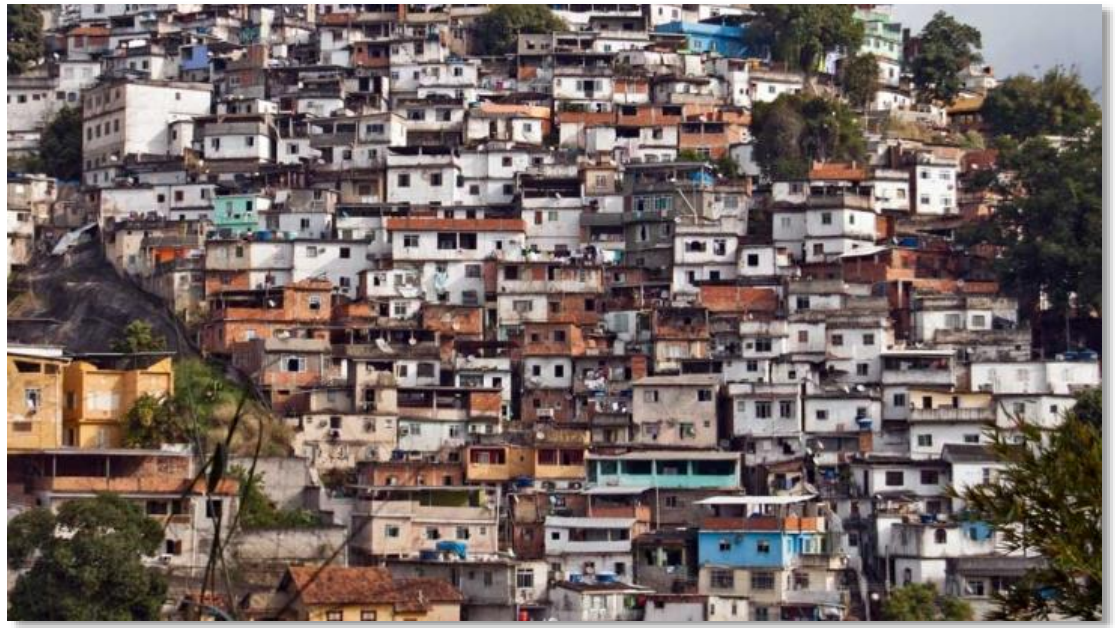
Brian Foote and Joseph Yoder, *Big Ball of Mud,* September 1997



Anti-Pattern: Big ball of mud

De unde ? De ce ?

- *Throwaway code* – soluții temporare (*Prototyping*) ce trebuie înlocuite/rescrise
- *Cut and Paste code* - ☺
- Adaptarea prin comentare/ștergere a altor soluții
- Termene limită foarte scurte sau nerealiste
- Lipsa de experiență
- Lipsa unor standarde/proceduri



Anti-Pattern: Big ball of mud

Cum eviți ?

- Rescrierea codului (***Refactoring***) pana la un nivel acceptabil de maturitate
- Utilizare Principii Clean Code
- Implementare Design Patterns

Design-pattern

- Un ***pattern*** reprezintă o *soluție reutilizabila* pentru o *problema standard*, într-un anumit *context*
- Facilitează reutilizarea arhitecturilor si a design-ului software
- NU sunt structuri de date

Design-pattern

“A pattern involves a general description of a recurring solution to a recurring problem with various goals and constraints. It identify more than a solution, it also explains why the solution is needed.“

James Coplien

“... describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice”

Cristopher Alexander

Avantajele unui Design-Pattern

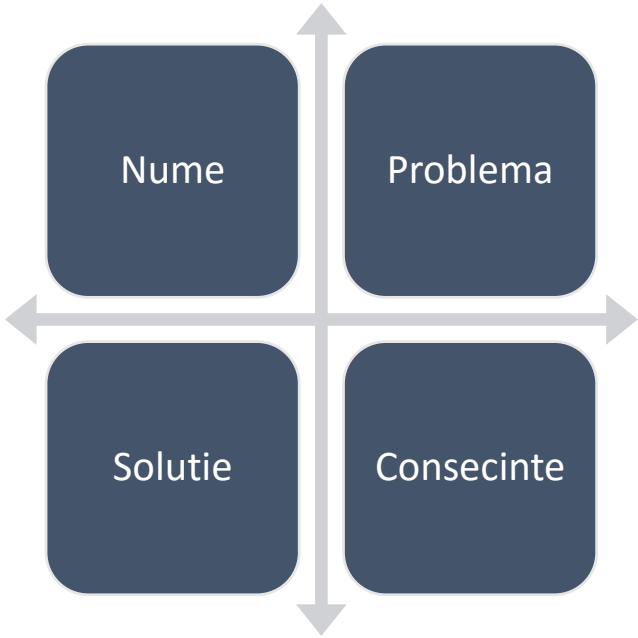
- Permit reutilizarea soluțiilor standard la nivel de cod sursa/arhitectura
- Permit documentarea codului sursa/arhitecturilor
- Permit înțelegerea mai facila a codului sursa/a arhitecturii
- Reprezintă concepte universal cunoscute - definesc un vocabular comun
- Sunt soluții testate si foarte bine documentate

Design-Pattern in Arhitecturi Software

Enterprise	
System	• OO Architecture
Application	• Subsystem
Macro	• Frameworks
Micro	• Design-Patterns
Objects	• OOP

Componentele unui Design-Pattern

- **Nume**:
  - Face parte din vocabularul unui programator/designer/arhitect software
  - Identifica in mod unic pattern-ul
- **Problema**:
  - Descrie scopul urmărit
  - Definește contextul
  - Stabilește când este aplicabil pattern-ul
- **Soluție**
  - Diagrama UML, pseudo-cod ce descrie elementele
- **Consecințe**
  - Rezultate
  - Avantaje si dezavantaje



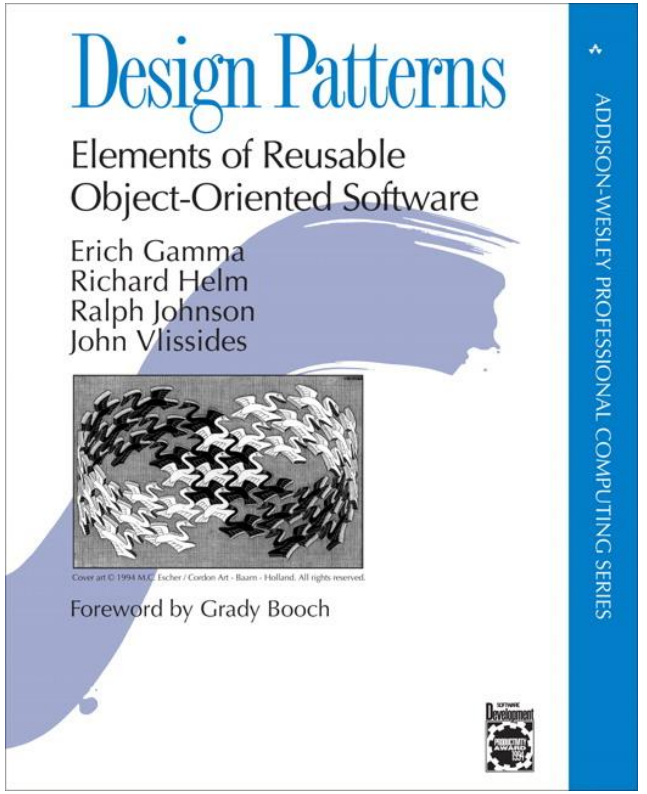
Istoric Design-Pattern

- 1970 – primele modele legate de conceptual de Window si Desktop (Smalltalk, Xerox Parc, Palo Alto)
- 1978 – MVC pattern (Goldberg and Reenskaug, Smalltalk, Xerox Parc)
- 1987 - Kent Beck and Ward Cunningham, "*Using Pattern Languages for Object-Oriented Programs*", OOPSLA-87 Workshop
- 1991 - Erich Gamma, an idea for a Ph.D. thesis about patterns
- 1993 - E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Abstraction and Reuse of Object-Oriented Design. ECOOP 97 LNCS 707, Springer, 1993

Istoric Design-Pattern

Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides (Addison-Wesley, 1995) - *Design Patterns*

- The Gang of Four (GOF)
- Cartea descrie 23 pattern-uri – probleme si soluții ce pot fi aplicate in numeroase scenarii
- Cea mai populara carte de Computer Software

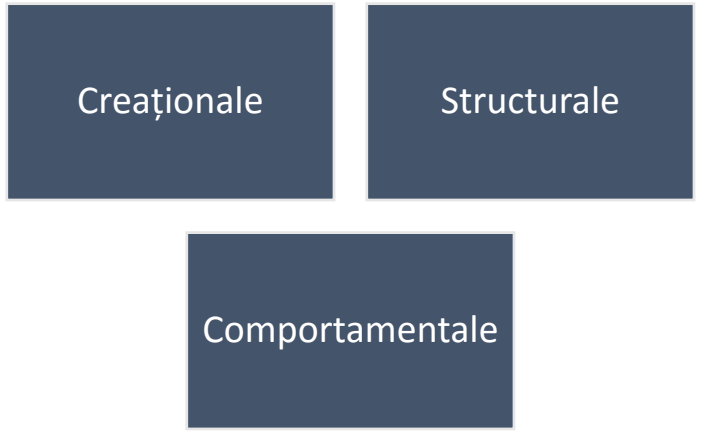


Utilizare Design-Patterns

- *Observer* in Java AWT si Swing pentru callback-uri
- *Iterator* in C++ STL si Java Collections
- *Façade* in multe librarii Open-Source pentru a ascunde complexitatea rutinelor interne
- *Bridge* si *Proxy* in framework-uri pentru aplicatii distribuite
- *Singleton* in Hybernate si NHybernate

Tipuri de Design-Pattern

- Creaționale
  - Inițializarea si configurarea claselor si obiectelor
- Structurale
  - Compoziția claselor si obiectelor
  - Decuplarea interfețelor si a claselor
- Comportamentale
  - Distribuția responsabilității
  - Interacțiunea intre clase si obiecte



Tipuri de Design-Pattern

Creaționale

- Factory Method
- Abstract Factory
- Builder
- Prototype
- Singleton

Structurale

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Comportamentale

- Interpreter
- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Visitor
- Template

Creational Design-Patterns

- Abstract Factory
  - Pattern pentru crearea de obiecte aflate intr-un anumit context
- Builder
  - Pattern pentru crearea in mod structurat (incremental) de obiecte complexe
- Factory Method
  - Pattern ce defineste o metoda pentru crearea de obiecte din aceeasi familie (interfata) in subclase
- Prototype
  - Pattern pentru clonarea unor noi instante (clone) ale unui prototip existent
- Singleton
  - Pattern pentru crearea unei singure instante (unica)

Structural Design-Patterns

- **Adapter:**
  - Adaptează interfața unui server/serviciu la client
- **Bridge**:
  - Decuplează modelul abstract de implementare
- **Composite**:
  - Agregarea a mai multor obiecte similare
- **Decorator**:
  - Extinde intr-un mod transparent un obiect
- **Facade**:
  - Simplifica interfața unui modul/subsistem
- **Flyweight**:
  - Partajare memorie intre obiecte similare.
- **Proxy**:
  - Interfață către alte obiecte/resurse

Behavioral Design-Patterns

- **Chain of Responsibility**:
  - Gestioneaza tratarea unui eveniment de catre mai multi furnizori de solutii
- **Command**:
  - Request or Action is first-class object, hence re-storable
- **Iterator**:
  - Gestioneaza parcurgerea unei colectii de elemente
- **Interpreter**:
  - Intepretor pentru un limbaj cu o gramatica simpla
- **Mediator**:
  - Coordoneaza interactiunea dintre mai multi asociati
- **Memento**:
  - Salvazeaza si restaureaza starea unui obiect

Behavioral Design-Patterns

- **Observer**:
  - Defineste un hadler pentru diferite evenimente
- **State**:
  - Gestioneaza obiecte al caror comportament depinde de starea lor
- **Strategy**:
  - Incapsuleaza diferiti algoritmi
- **Template Method**:
  - Incapsuleaza un algoritm ai carui pasi depend de o clasa derivate
- **Visitor**:
  - Descrie metode ce pot fi aplicate pe o structura neomogena

Creational Design-Patterns

Singleton, Abstract Factory, Factory Method, Simple Factory, Builder, Prototype

Modelul SINGLETON

Creational Design-Patterns

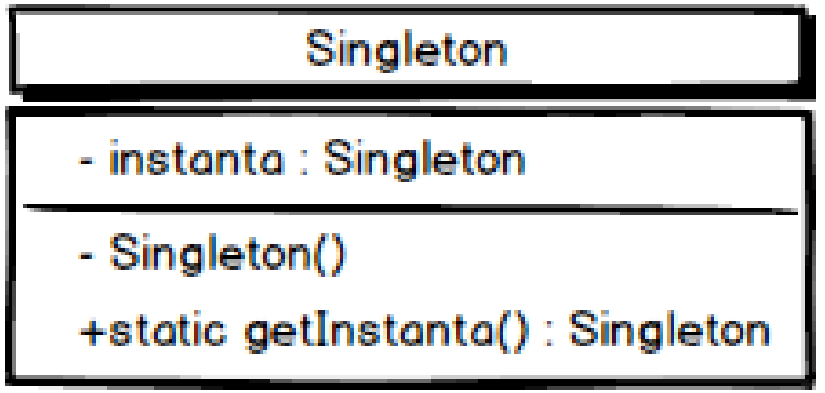
SINGLETON- Problema

- Se dorește crearea unei singure instanțe pentru o clasă prin care să fie gestionată o resursă/un eveniment în mod centralizat;
- Soluția se bazează pe existența unei singure instanțe ce poate fi creata o singură dată dar care poate fi referită de mai multe ori;
- Asigură un singur punct de acces, vizibil global, la unica instanță
- Exemple: gestiune conexiune baze de date sau alte resurse; mecanism de logging unic; manager evenimente; manager resurse vizuale; manager configurare.

SINGLETON - Diagrama

```
public class Singleton {
    private static Singleton instance = null;
    private Singleton() { }

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



SINGLETON - Componente

- **`Singleton()`**
  - un constructor privat (apelabil doar din clasa)
- **`private static Singleton instance`**
  - un atribut static, privat, de tipul clasei ce reprezinta instanta unica
- **`Singleton getInstance()`**
  - o metoda publica ce da acces la instanta unica
  - instanta unica este creata la primul apel al metodei

SINGLETON – Alte implementări

- Varianta 1 – atribut privat definit static
- Varianta 2 – atribut public static constant
- Varianta 3 – prin enumerare (Joshua Bloch în *Effective Java, 2nd Edition*)
- Varianta 4 – *Singleton collection* sau *Singleton registry* – obiectele unice sunt gestionate printr-o colecție

SINGLETON – Avantaje si Dezavantaje

Avantaje:

- Gestiune centralizata a unei resurse printr-o instanță unică
- Controlul strict al instanțierii unei clase – o singura data
- Nu permite duplicarea instanțelor
- Ușor de implementat
- ***Lazy instantiation*** – obiectul este creat atunci când este necesar

Dezavantaje:

- In multi-threading pot apărea probleme de sincronizare sau cooperare daca singleton-ul este partajat
- Poate deveni un *bottleneck* care sa afecteze performanta aplicației

SINGLETON – Scenarii

- Conexiune unica la baza de date;
- Gestiune unica fișiere/fișier de configurare;
- Gestiune unica preferințe pe platforma Android (SharedPreferences) sau gestiunea setărilor aplicației într-un context mai general;
- Gestiune unica conexiune rețea;
- Gestiune centralizată a accesului la anumite resurse utilizate de soluția software;
- Gestiune unică obiecte costisitoare, pe baza timpului și a resurselor necesare creării, ce trebuie să aibă o instanță unică. utilizată de mai multe ori.

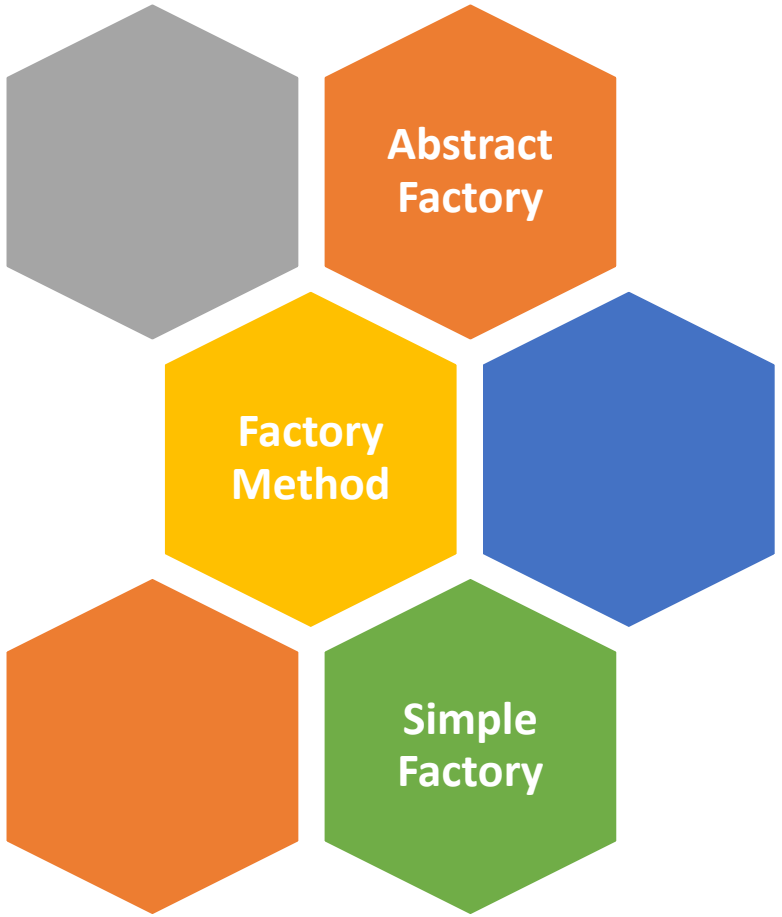
Modelul FACTORY

Creational Design-Patterns

FACTORY - Problema

- implementarea unui mecanism centralizat prin care crearea obiectelor este transparenta pentru client; prin interfața publică clientul știe cum să creeze obiecte însă nu știe cum este implementat acest lucru;
- soluția poate să fie extinsă prin adăugarea de noi tipuri concrete de obiecte fără a afecta codul existent;
- complexitatea creării obiectelor este ascunsa clientului;
- obiectele sunt referite printr-o interfață comună; clase concrete reprezintă o familie de obiecte definită în jurul interfeței comune;
- eliminarea dependenței codului clientului de crearea efectivă a obiectelor utilizate în soluție;

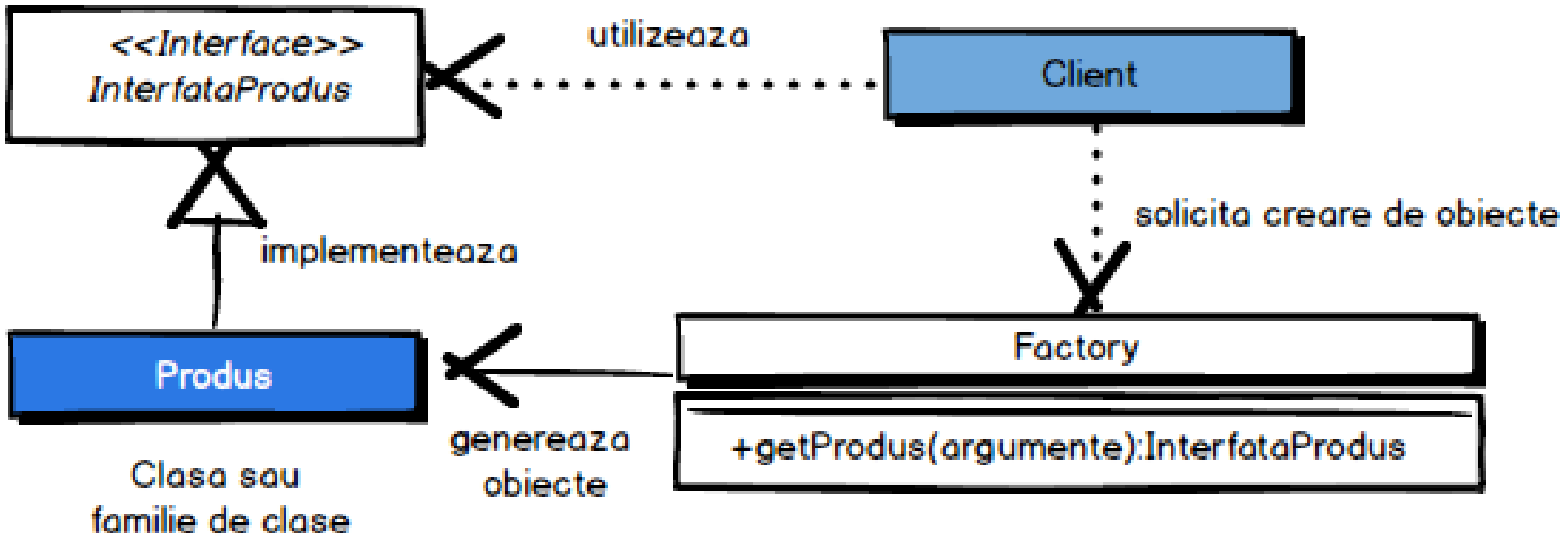
FACTORY



SIMPLE FACTORY

- este un caz particular al pattern-lui *Factory*
- unul dintre cele mai folosite în producție datorită simplității în implementare și a avantajelor oferite
- NU este descris în cartea celor 4 (GoF), el apărând natural, din practică

SIMPLE FACTORY - Diagrama



SIMPLE FACTORY - Componente

`InterfataProdus`

- interfața abstractă a obiectelor de tip Produs;

`Produs`

- tipurile concrete de clase ce implementează interfața ; instanțele acestei clase vor fi generate de către Factory;

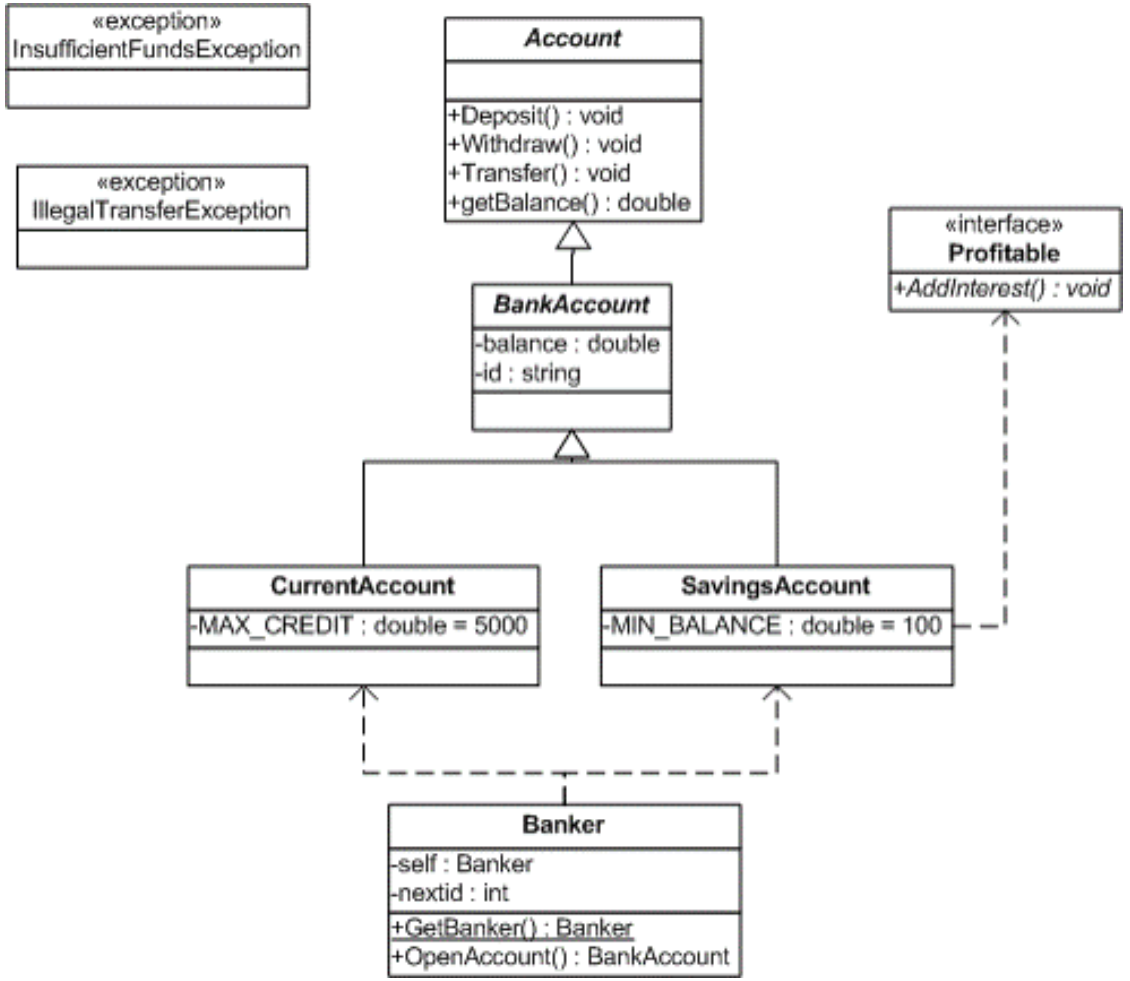
`Factory`

- clasa ce încapsulează procesul de creare a obiectelor de tip *InterfaceProdus*;

`Client`

- o altă clasă sau o metodă, care folosește interfața *Factory*-ului pentru a construi obiecte noi;

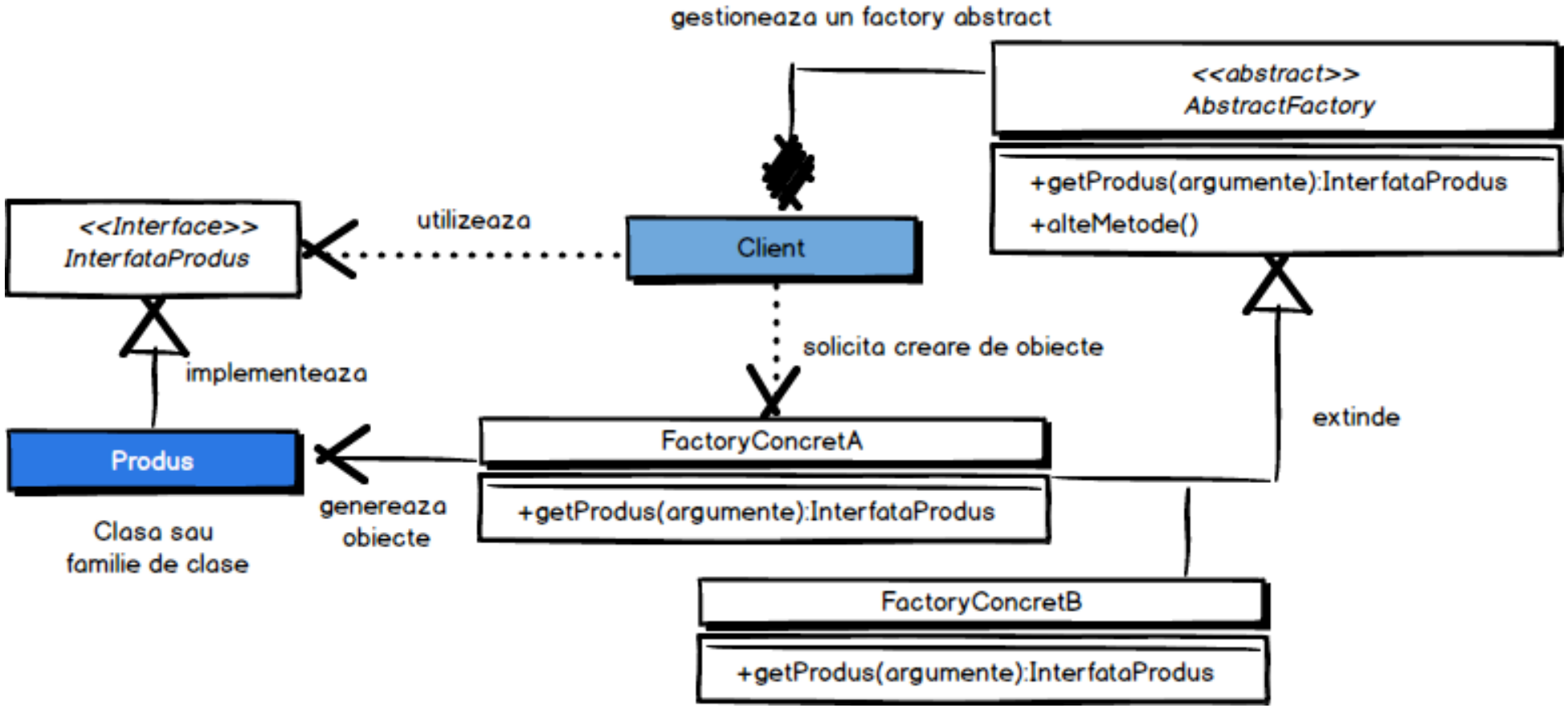
Exemplu de SINGLETON si SIMPLE FACTORY



Modelul FACTORY METHOD (Virtual Constructor)

Creational Design-Patterns

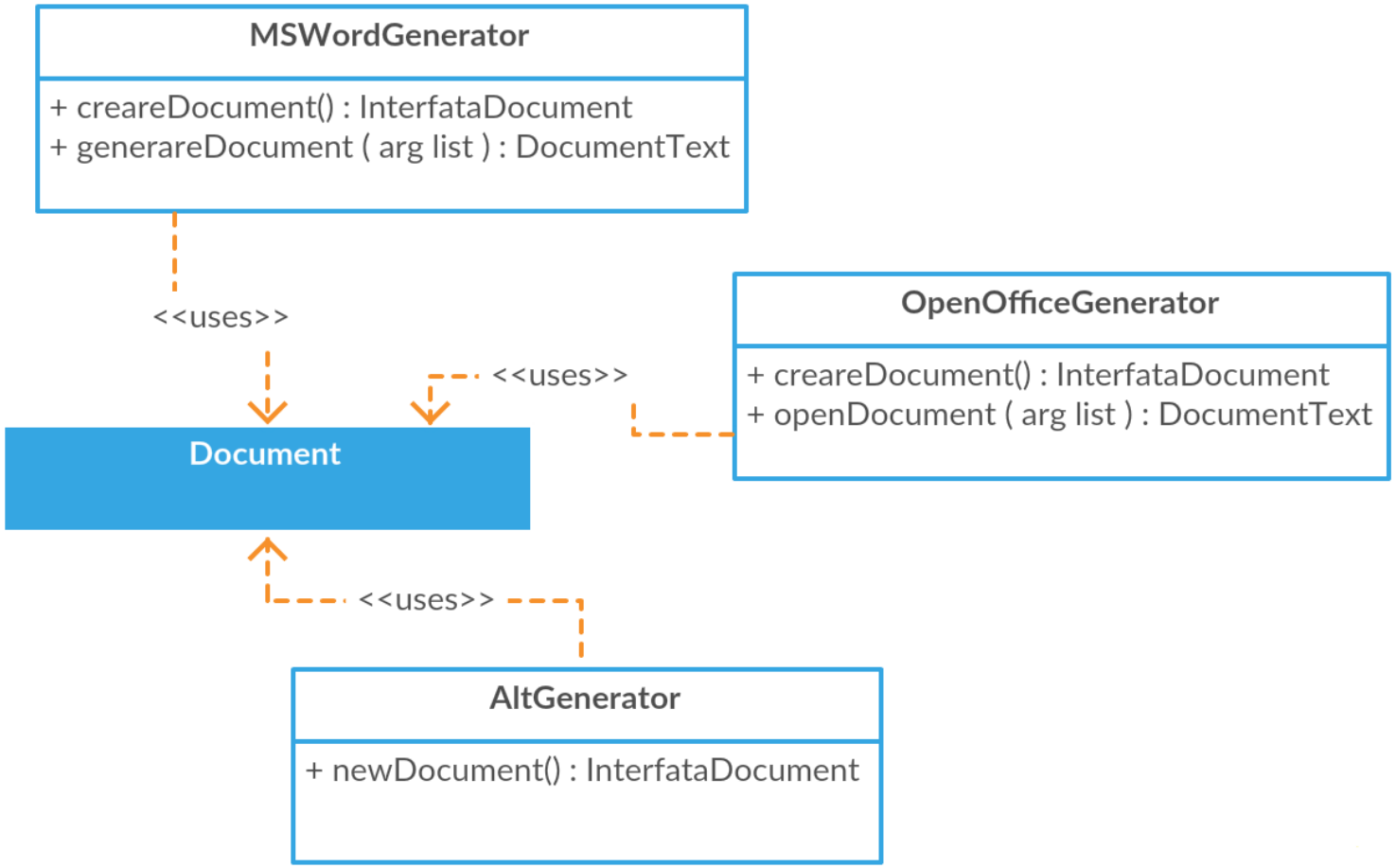
FACTORY METHOD - Diagrama



FACTORY METHOD - Componente

- **`InterfataProdus`**
  - interfață ce definește tipurile generice de obiecte ce pot fi create
- **`Produs`**
  - clasa concreta ce definește tipul de obiecte ce poate fi creat
- **`Factory`**
  - clasa abstracta ce definește interfața unui generator de obiecte
- **`FacgtoryConcretA, FactoryConcretB`** …
  - clasa concreta ce implementează generatorul de obiecte

FACTORY METHOD - Exemplu



FACTORY METHOD– Avantaje si Dezavantaje

Avantaje:

- Toate obiecte create au în comun interfața
- Controlul strict al instanțierii – obiectele nu sunt create direct prin constructori ci prin metoda de tip *factory*
- Diferitele tipuri de obiecte sunt gestionate unitar prin interfață comuna – noi tipuri, din aceeași familie, pot fi adăugate fără modificări
- Ușor de implementat
- Pot fi generate obiecte noi care aparțin aceleiași familii (interfață comună)
- Implementează principiul ***Dependency Inversion***

Dezavantaje:

- Nu pot fi generate obiecte “noi”
- Constructorii sunt privați – clasele nu pot fi extinse

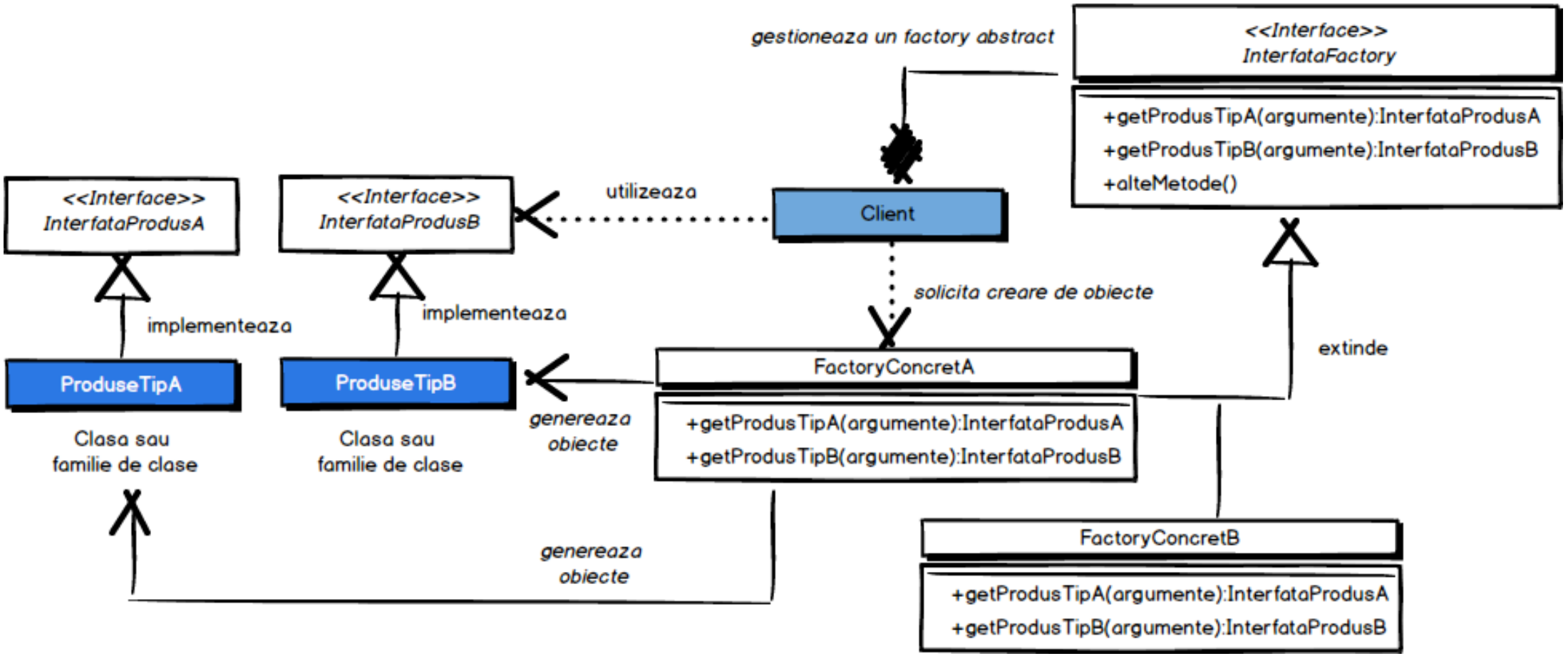
Modelul ABSTRACT FACTORY

Creational Design-Patterns

ABSTRACT FACTORY- Problema

- Se dorește implementarea unui mecanism prin care crearea obiectelor este transparenta pentru client
- Soluția poate să fie extinsă prin adăugarea de noi tipuri concrete de obiecte fără a afecta codul scris
- Complexitatea creării obiectelor este ascunsa clientului. Acesta știe cum sa le creeze însă nu știe cum acestea sunt efectiv create
- Crearea obiectelor este decuplata de soluție si generatorul poate fi înlocuit fără prea mare efort
- Obiectele sunt referite printr-o interfață comună și nu direct. Ele formează o familie de obiecte in jurul interfeței comune

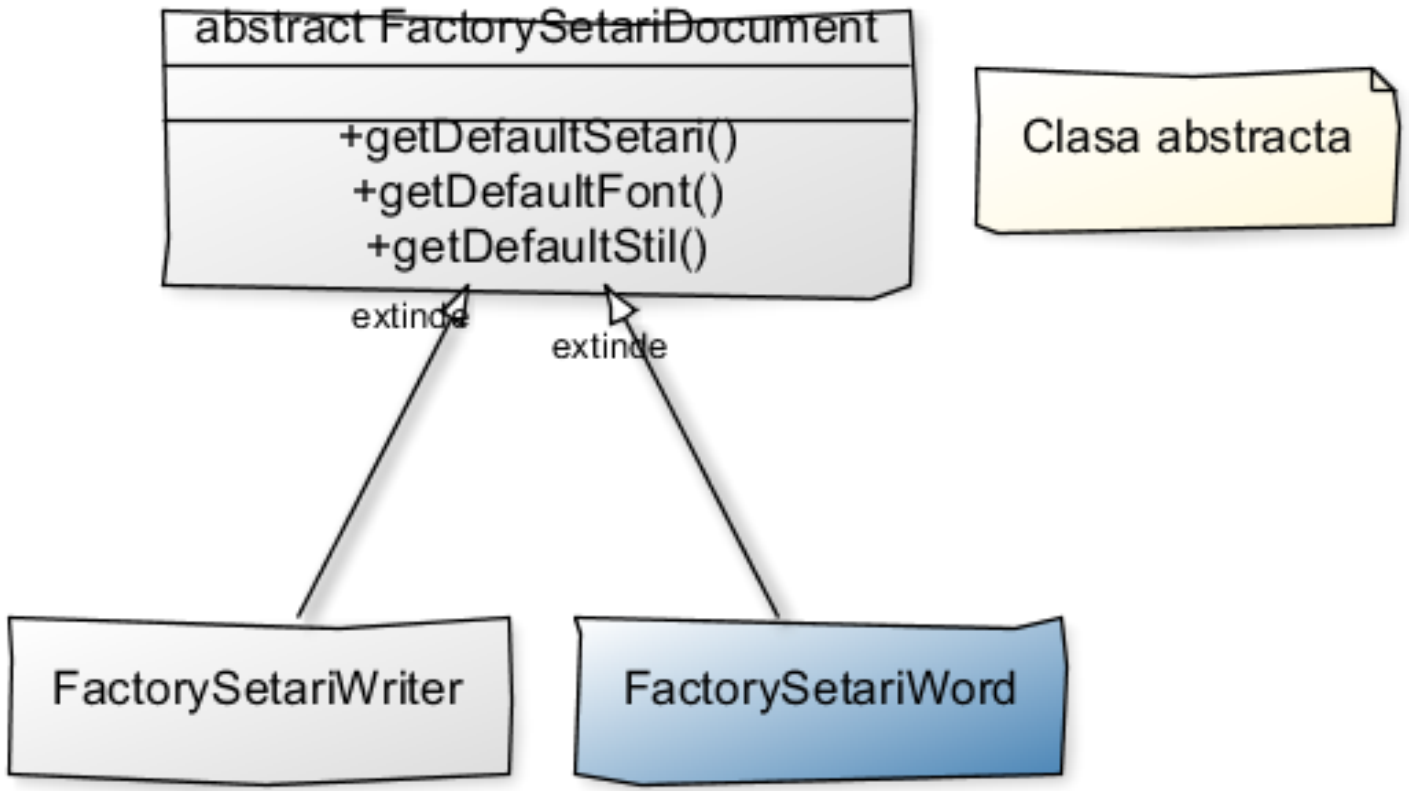
ABSTRACT FACTORY - Diagrama



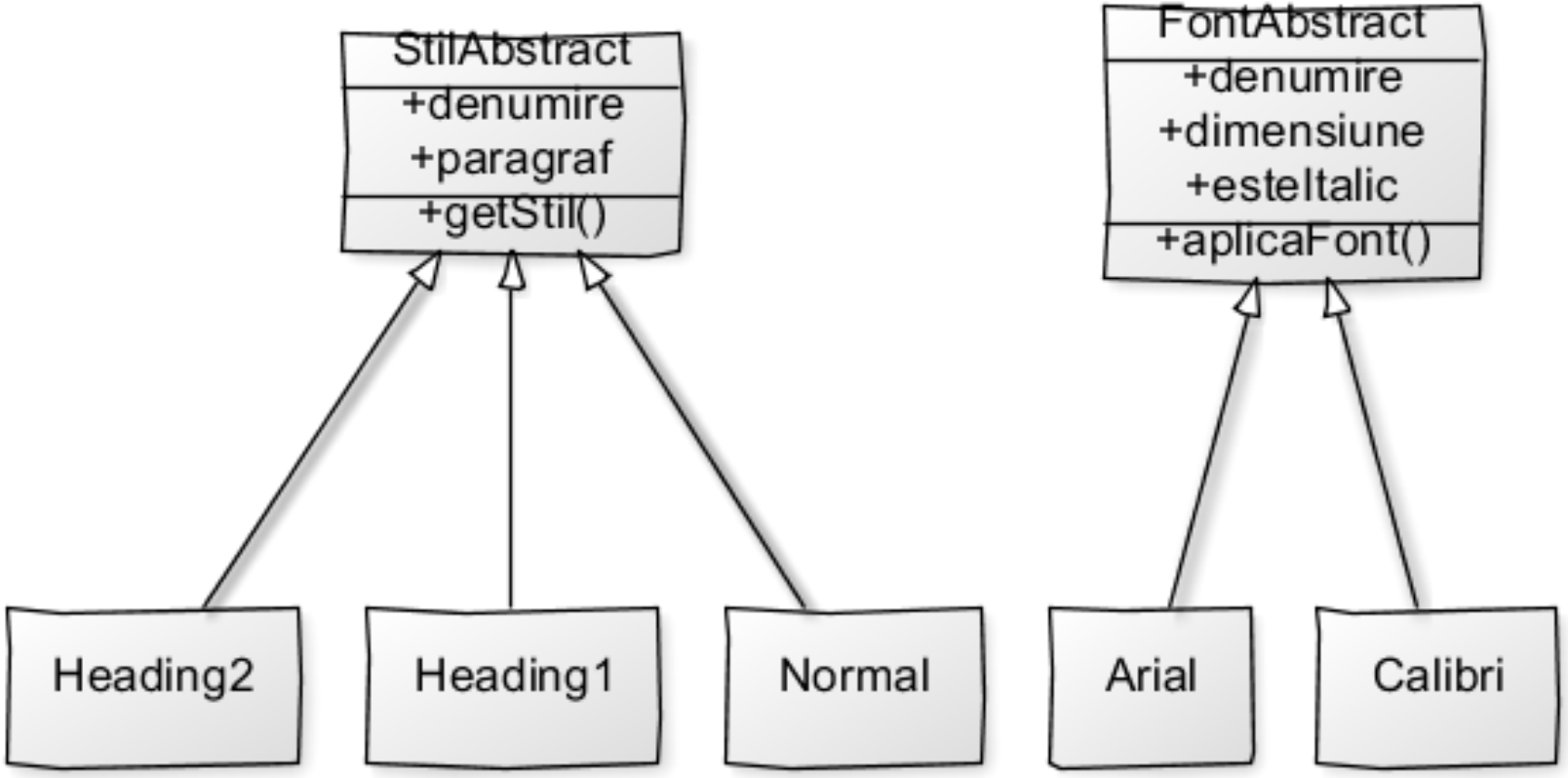
ABSTRACT FACTORY - Componente

- **`InterfataFactory`**
  - interfață ce definește metodele abstracte pentru crearea de instanțe
- **`InterfataProdusA/InterfataProdusB`**
  - interfețe ce definesc tipurile abstracte de obiecte ce pot fi create
- **`FactoryConcretA/FactoryConcretB`**
  - clase concrete ce implementează interfața si metodele prin care sunt create obiecte de tipul ***`Product`***
- **`ProduseTipA, ProduseTipB,`**…
  - clase concrete ce definesc diferitele tipuri de obiecte ce pot fi create

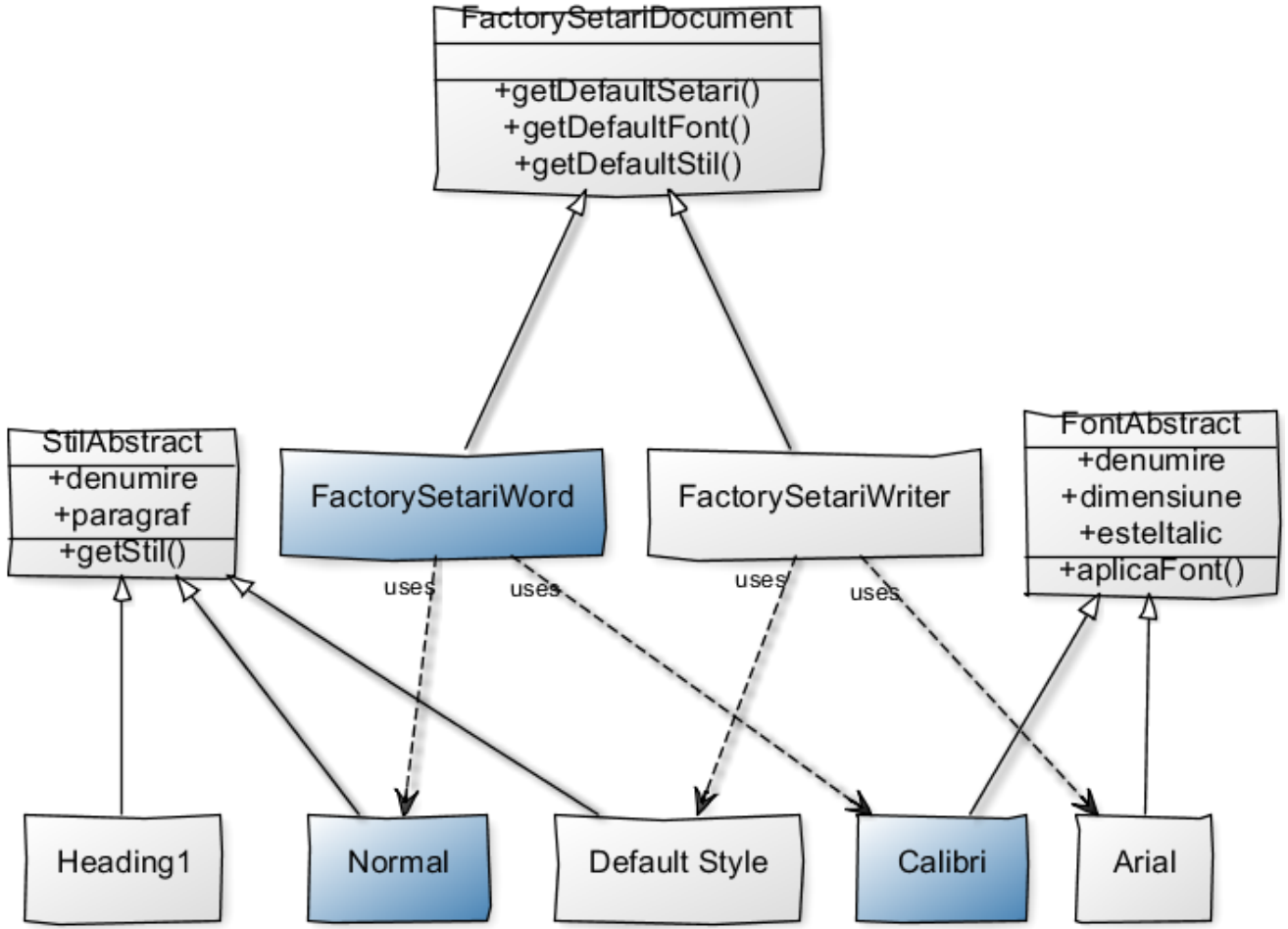
ABSTRACT FACTORY - Exemplu



ABSTRACT FACTORY - Exemplu



ABSTRACT FACTORY - Exemplu



ABSTRACT FACTORY– Avantaje si Dezavantaje

Avantaje:

- decuplează generatorul de instanțe de clientul care le utilizează;
- controlul strict al instanțierii – obiectele nu sunt create direct prin constructori ci prin metodele de tip *factory*;
- diferitele tipuri de obiecte sunt gestionate unitar prin interfață comuna – noi tipuri pot fi adăugate fără modificări

Dezavantaje:

- număr mare de clase implicate
- Nivel de complexitate ridicat

ABSTRACT FACTORY– Scenarii

- Gestiune creare produse catalog magazine virtual
- Gestiune creare tipuri diferite de client
- Gestiune creare tipuri diferite de rapoarte
- Gestiune creare tipuri de numere de telefon
- Gestiune creare tipuri de conturi bancare
- Gestiune creare tipuri de pizza
- Gestiune creare meniuri într-un restaurant

FACTORY

- Toate pattern-urile Factory promovează ***loose coopling*** si sunt bazate pe ***Dependency Inversion Principle***
- Factory Method
  - bazata pe moștenire – crearea obiectelor este realizata de subclase ce implementează metoda *factory*
  - Are scopul de a delega crearea obiectelor către subclase
- Abstract Factory
  - bazata pe compunere – crearea obiectelor este realizata de metode publicate de interfață
  - Are scopul de a crea familii de obiecte fără a depinde de implementarea lor concreta

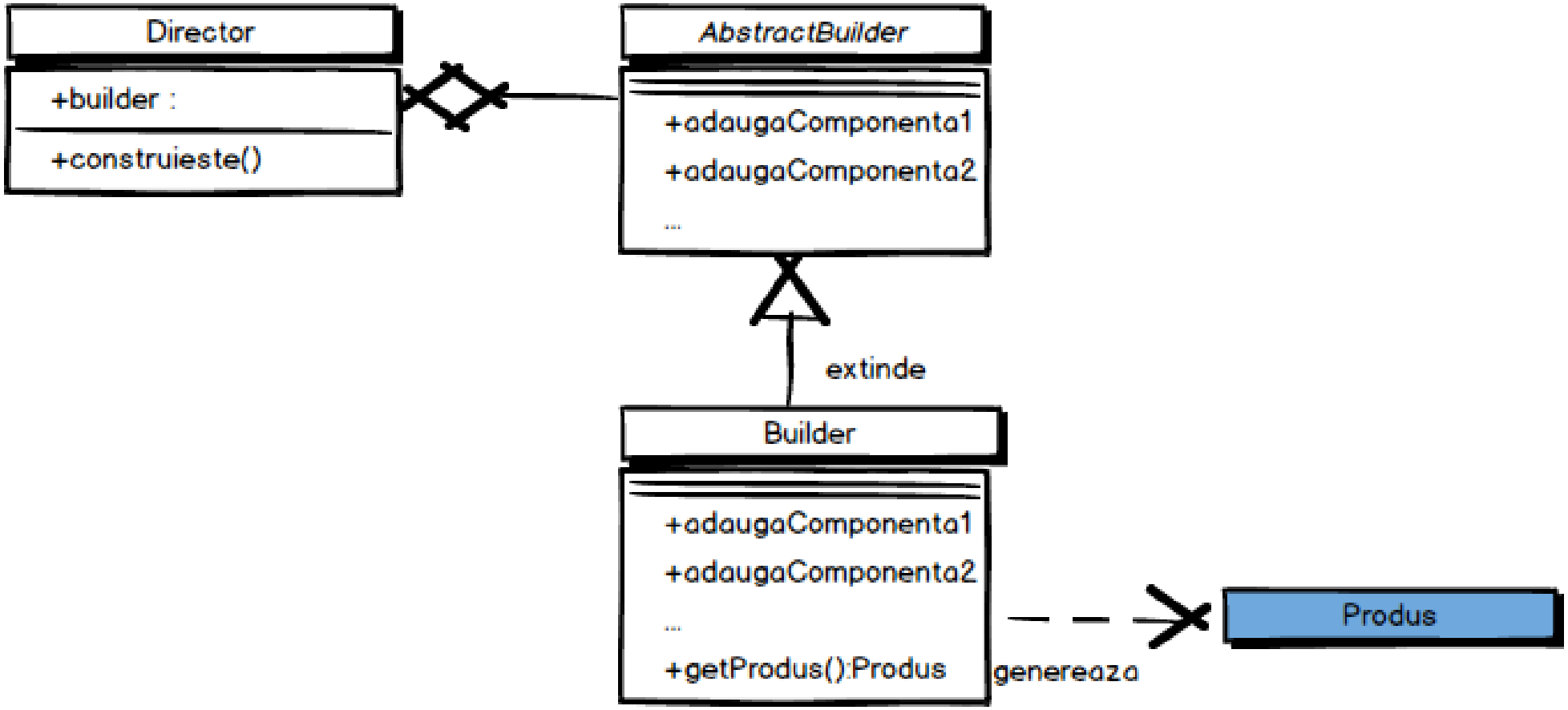
Modelul BUILDER (Adaptive Builder)

Creational Design-Patterns

BUILDER - Problema

- Soluția trebuie sa construiască obiecte complexe printr-un mecanism care este independent de procesul de realizare a obiectelor
- Clientul construiește obiectele complexe specificând doar tipul si valoarea sa, fără a cunoaște detaliile interne ale obiectului (cum stochează si reprezintă valorile)
- Procesul de construire a obiectelor trebuie sa poată fi utilizat pentru a defini obiecte diferite din aceeași familie
- Obiectele sunt gestionate prin interfața comuna
- Instanța de tip *Builder* construiește obiectul însă tipul acestuia este definit de subclase

BUILDER- Diagrama



http://en.wikipedia.org/wiki/Builder_pattern

BUILDER - Componente

- **`AbstractBuilder`**
  - interfața abstracta ce definește metodele prin care sunt construite parti ale obiectului complex
- **`Builder`**
  - clasa concreta ce construiește părțile si pe baza lor obiectul final
- **`Produs`**
  - clasa abstracta ce definește obiectul complex ce este construit
- **`Director`**
  - clasa concreta ce construiește obiectul complex utilizând interfața de tip *Builder*

BUILDER - Avantaje si Dezavantaje

Avantaje:

- Obiectele complexe pot fi create independent de părțile care îl compun (un obiect poate sa le conțină pe toate sau doar o parte)
- Sistemul permite reprezentarea diferita a obiectelor create printr-o interfață comună
- Algoritmul de creare a obiectului este flexibil deoarece clientul alege ce părți sa fie create

Dezavantaje:

- **Atenție la crearea de obiecte – pot fi omise atribute**

Modelul PROTOTYPE

Creational Design-Patterns

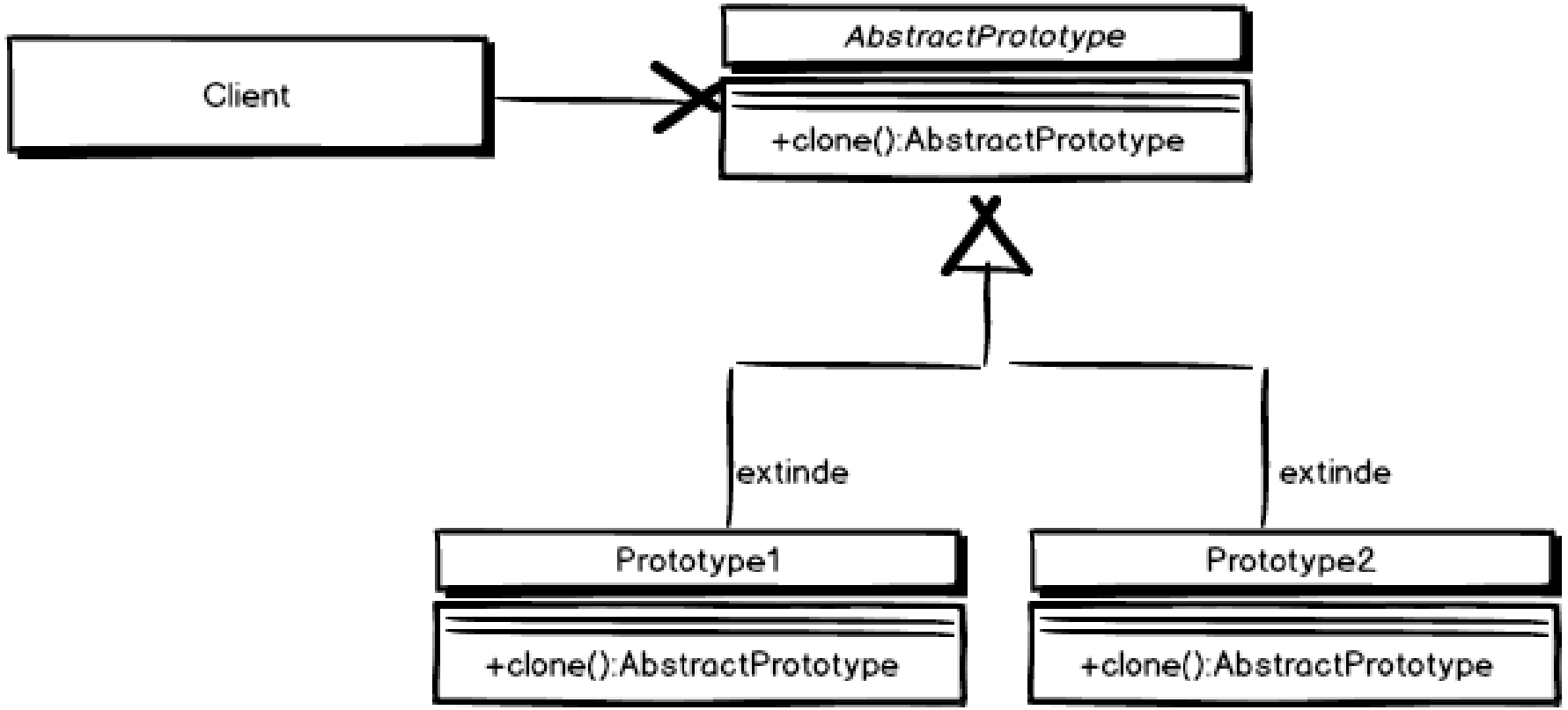
PROTOTYPE – Scenariu

ACME Inc. dorește să dezvolte un joc 3D pentru dispozitive Android utilizând un engine propriu. Cele 2 modele 3D pentru caractere sunt destul de complexe și generarea lor are impact asupra timpului de procesare și implicit asupra duratei de viață a acumulatorului. Același model este utilizat de mai multe ori pentru a popula o scena cu personaje. Trebuie găsită o soluție eficientă prin care scenele să fie încărcate rapid.

PROTOTYPE - Problema

- Soluția generează obiecte costisitoare (timp creare si memorie ocupata) cu durata de viată lungă
- Pentru eficienta, Soluția reutilizează obiectul prin clonarea acestuia (se creează o instant noua a obiectului)
- Implementat printr-o metoda clone()

PROTOTYPE - Diagrama



PROTOTYPE - Implementare

SHALLOW COPY VS. DEEP COPY

In Java implementarea implicită pentru clone() este ?

PROTOTYPE - Avantaje si Dezavantaje

Avantaje:

- Creare rapidă de obiecte identice (valori) prin clonare
- Evită apelul explicit al constructorului
- Poate fi construita o colectie de prototipuri care sa fie utilizata pentru a genera obiecte noi

Dezavantaje:

- **Atenție la crearea de obiecte ce partajează aceleași resurse – shallow copy**

Structural Design-Patterns

Adapter, Facade, Decorator, Composite, Flyweight, Proxy

Modelul ADAPTER (Wrapper)

Structural Design-Patterns

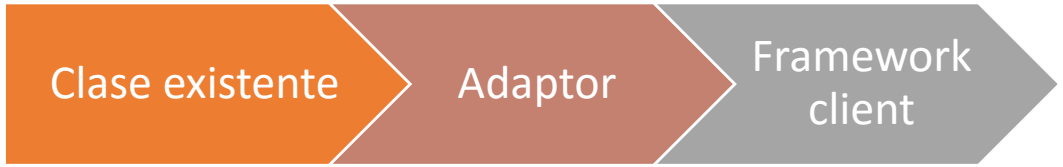
ADAPTER – Scenariu

ACME Inc. dorește să cumpere un nou framework pentru serviciile din back-end. Interfața pentru aceste servicii gestionează datele prin intermediul obiectelor de tip ACME, iar noul framework procesează datele prin intermediul obiectelor de tip MICRO. Programatorii companiei trebuie sa găsească o soluție de a integra cele doua framework-uri fără a le modifica.

ADAPTER - Problema

- Utilizarea împreună a unor clase ce nu au o interfață comuna
- Clasele **nu se modifica** însă se construiește o interfață ce permite utilizarea lor in alt context
- Clasele sunt adaptate la un nou context
- Apelurile către interfața clasei sunt mascate de interfața adaptorului
- Transformarea datelor dintr-un format în altul

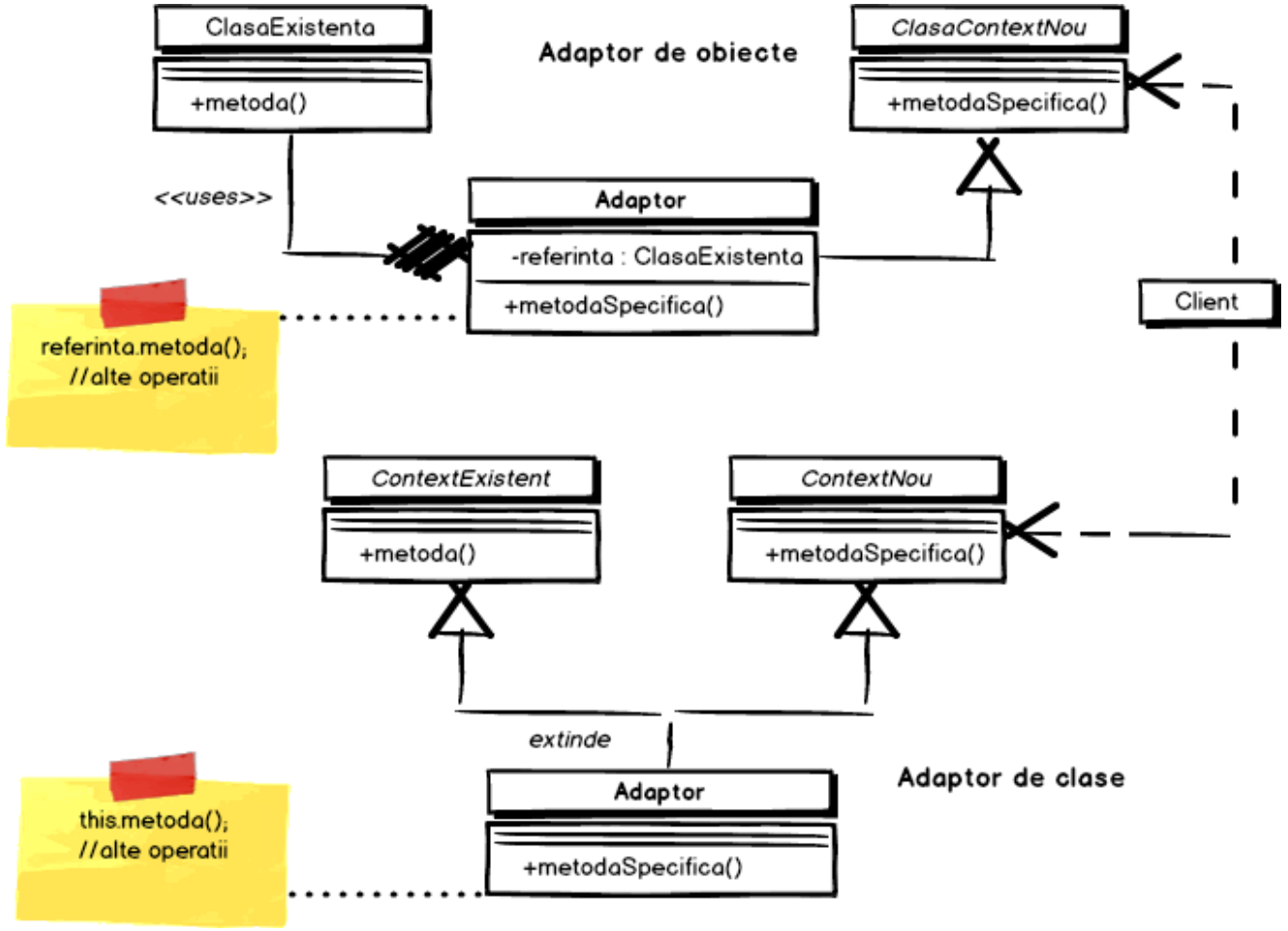
ADAPTER - Scenariu



Codul nu se schimbă

Codul nu se schimbă

ADAPTER - Diagrama



ADAPTER - Componente

- **`ClasaExistenta`**
  - clasa existenta ce trebuie adaptata la o noua interfață;
- **`ClasaContextNou`**
  - definește interfața specifică noului domeniu;
- **`Adaptor`**
  - adaptează interfața clasei existente la cea a clasei din noul context;
  - contine (composition) o referinta catre clasa/obiectul ce trebuie adaptat
- **`Client`**
  - Reprezintă framework-ul care apelează interfața specifică noului domeniu

ADAPTER – Avantaje si Dezavantaje

Avantaje:

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

Dezavantaje:

Adaptorul de clase se bazează pe derivare multipla, lucru care nu este posibil in Java. Alternativa este prin interfețe si compunere

Modelul FAÇADE (Wrapper)

Structural Design-Patterns

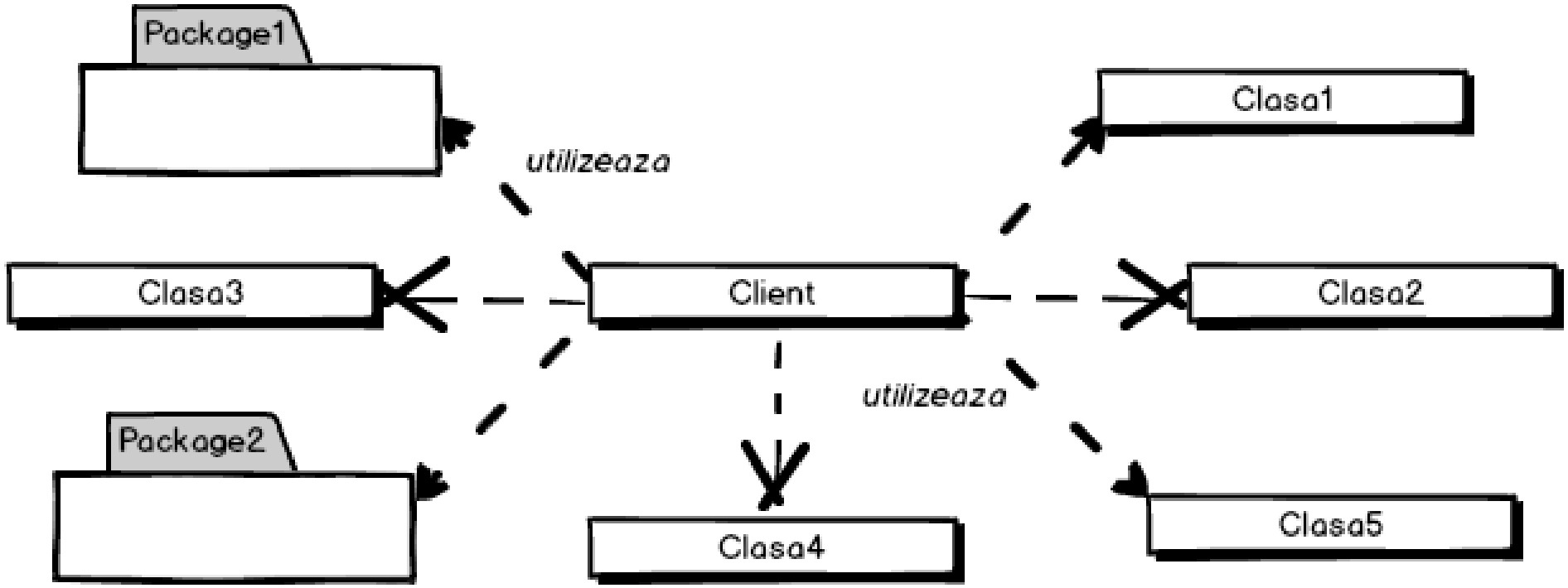
FAÇADE - Scenariu

ACME Inc. dezvoltă o soluție software pentru managementul unei locuințe inteligente. Includerea în framework a tuturor componentelor controlabile dintr-o astfel de locuință (ferestre, încălzire, alarma, etc) a generat un număr mare de clase. Departamentul care dezvolta interfața Web a soluției oferă un set minim de funcții ce pot fi controlate de la distanta. Deși funcționalitatea este simpla, numărul mare de clase ce se instanțiază și a metodelor apelate îngreunează dezvoltarea si testarea. In acest sens, o interfață mai simpla ar ajuta acest departament.

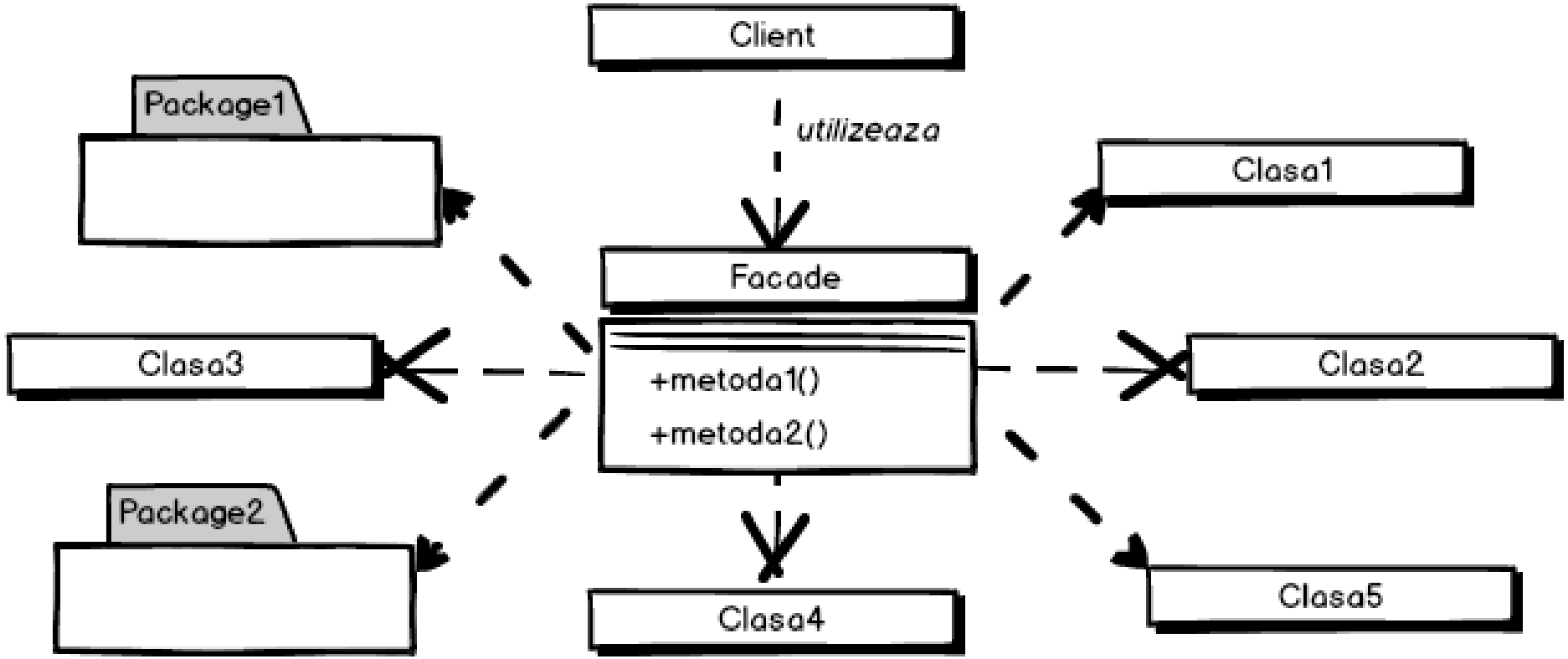
FAÇADE - Problema

- Soluția conține o mulțime de clase iar execuția unei funcții presupune apeluri multiple de metode aflate in aceste clase
- Clasele nu se modifica însă se construiește un layer intermediar ce permite apelul/gestiunea facila a metodelor din mai multe interfețe
- Utilă in situația in care framework-ul creste in complexitate si nu este posibila rescrierea lui pentru simplificare
- Apelurile către multiplele interfețe sunt mascate de aceasta interfață comună

FAÇADE - Scenariu



FAÇADE - Diagrama



FAÇADE - Componente

- **`Clasa1, Clasa2, …, Package1, Package2, …`**
  - clase existente ce pun la dispoziție diferite interfețe;
- **`Facade`**
  - Definește o interfața simplificata pentru contextul existent;
- **`Client`**
  - Reprezintă framework-ul care apelează interfața specifica noului domeniu

FAÇADE - Avantaje si Dezavantaje

Avantaje:

- Framework-ul nu se rescrie
- Se adaugă doar un layer intermediar ce ascunde complexitatea framework-ului din spate
- Pot fi definite cu ușurință metode care sa simplifice orice situație
- Implementează principiul ***Least Knowledge*** – reducerea interacțiunilor intre obiect la nivel de “prieteni apropriați”

Dezavantaje:

- Creste numărul de clase wrapper
- Creste complexitatea codului prin ascunderea unor metode
- Impact negativ asupra performantei aplicației

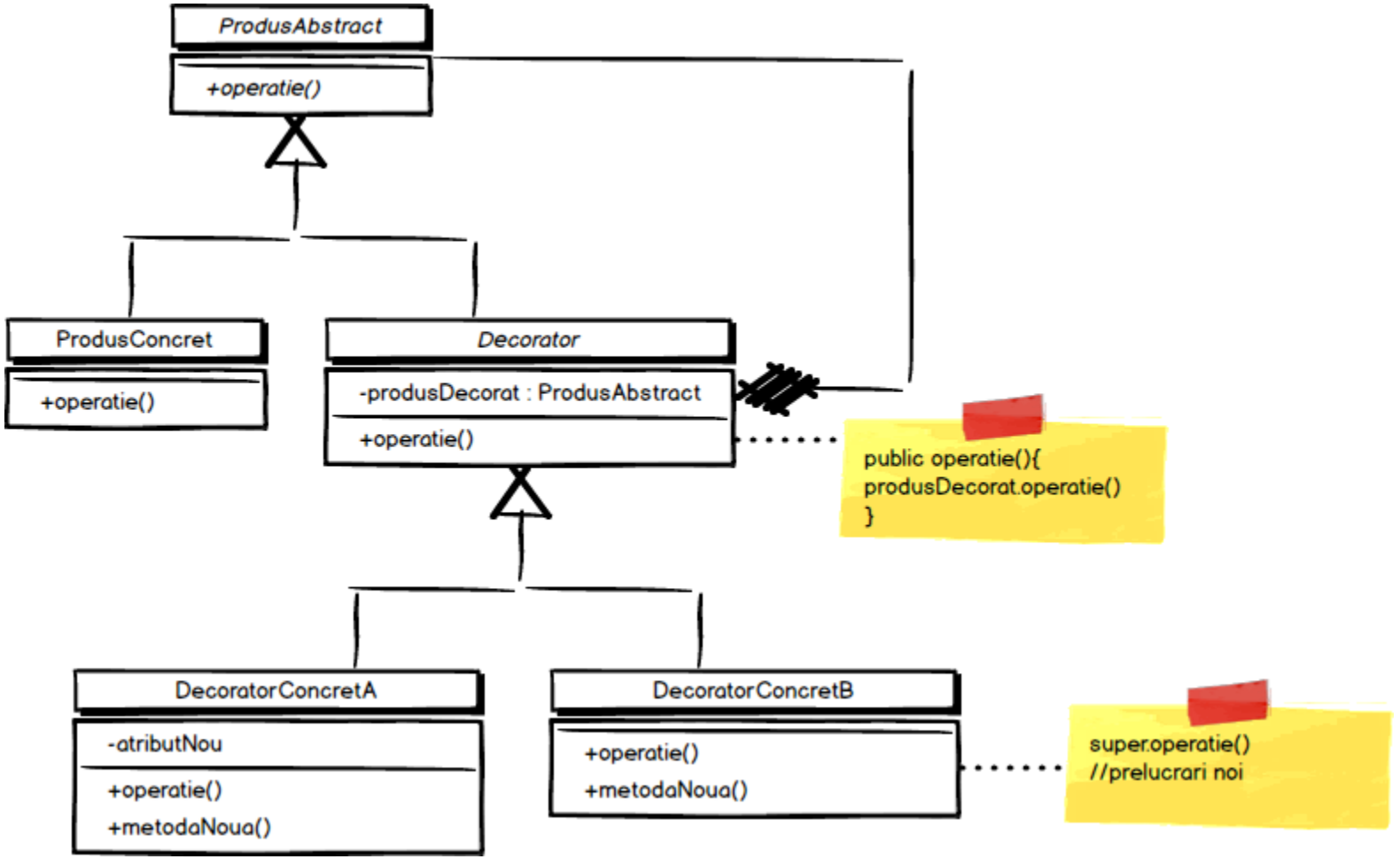
Modelul DECORATOR (Wrapper)

Structural Design-Patterns

DECORATOR - Problema

- Extinderea (decorarea) statica sau la *run-time* a funcționalității sau stării unor obiecte, independent de alte instanțe ale aceleiași clase
- Obiectul poate sa fie extins prin aplicarea mai multor decoratori
- Clasa existenta nu trebuie sa fie modificata
- Utilizarea unei abordări tradiționale, prin derivarea clasei, duce la ierarhii complexe ce sunt greu de gestionat. Derivarea adaugă comportament nou doar la compilare

DECORATOR - Diagrama



DECORATOR - Componente

- **`AbstractProduct`**
  - clasa abstracta ce definește interfața obiectelor ce pot fi decorate cu noi funcții;
- **`ConcreteProduct`**
  - definește obiecte ce pot fi decorate;
- **`Decorator`**
  - gestionează o referința de tip *AbstractProduct* către obiectul decorat;
  - metodele moștenite din *AbstractProduct* apelează implementările specifice din clasa obiectului referit;
  - Poate defini o interfață comună claselor decorator;
- **`ConcreteDecorator`**
  - Adaugă funcții noi obiectului referit;

DECORATOR - Avantaje

- Extinderea funcționalității a unui obiect particular se face dinamic, *la run-time*
- Decorarea este transparenta pentru utilizator deoarece clasa moștenește interfața specifica obiectului
- Decorarea se face pe mai multe niveluri, însă transparent pentru utilizator
- Nu impune limite privind un număr maxim de decorări

DECORATOR - Dezavantaje

- Un decorator este un *wrapper* pentru obiectul inițial. El nu este identic cu obiectul încapsulat
- Utilizarea excesiva generează o mulțime de obiecte care arata la fel dar care se comporta diferit –> dificil de înțeles si verificat codul
- Situația trebuie analizata cu atenție deoarece in unele situații pattern-ul *Strategy* este mai indicat

Adapter vs. Facade vs. Decorator

Adapter – asigură o interfață diferită obiectului

Facade – asigură o interfață simplificată obiectului

Decorator – asigură o interfață îmbunătățită obiectului

Modelul COMPOSITE

Structural Design-Patterns

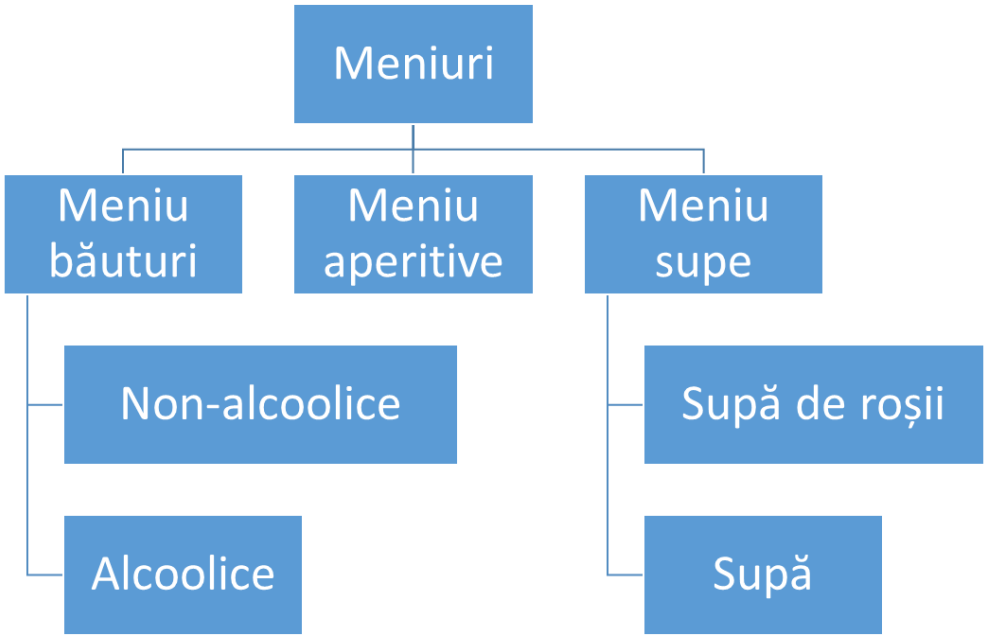
COMPOSITE - Scenariu

ACME Inc. dezvoltă o soluție software pentru managementul resurselor umane dintr-o companie. Soluția trebuie să ofere un mecanism unitar care să centralizeze angajații companiei și care să țină cont de:

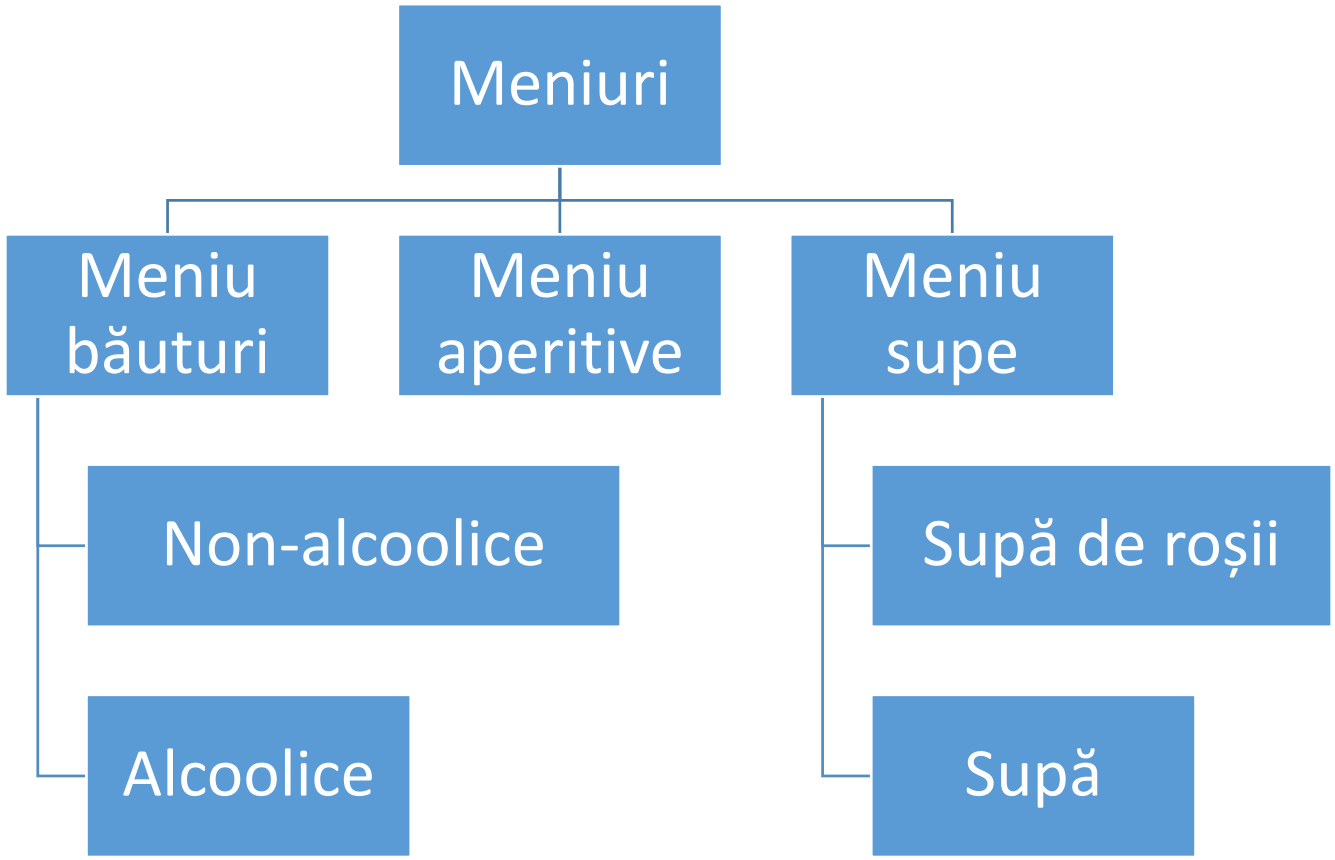
- relațiile ierarhice
- apartenența angajaților la un departament
- rolurile diferite ale angajaților
- setul comun de funcții pe care un angajat le poate îndeplini

COMPOSITE - Problema

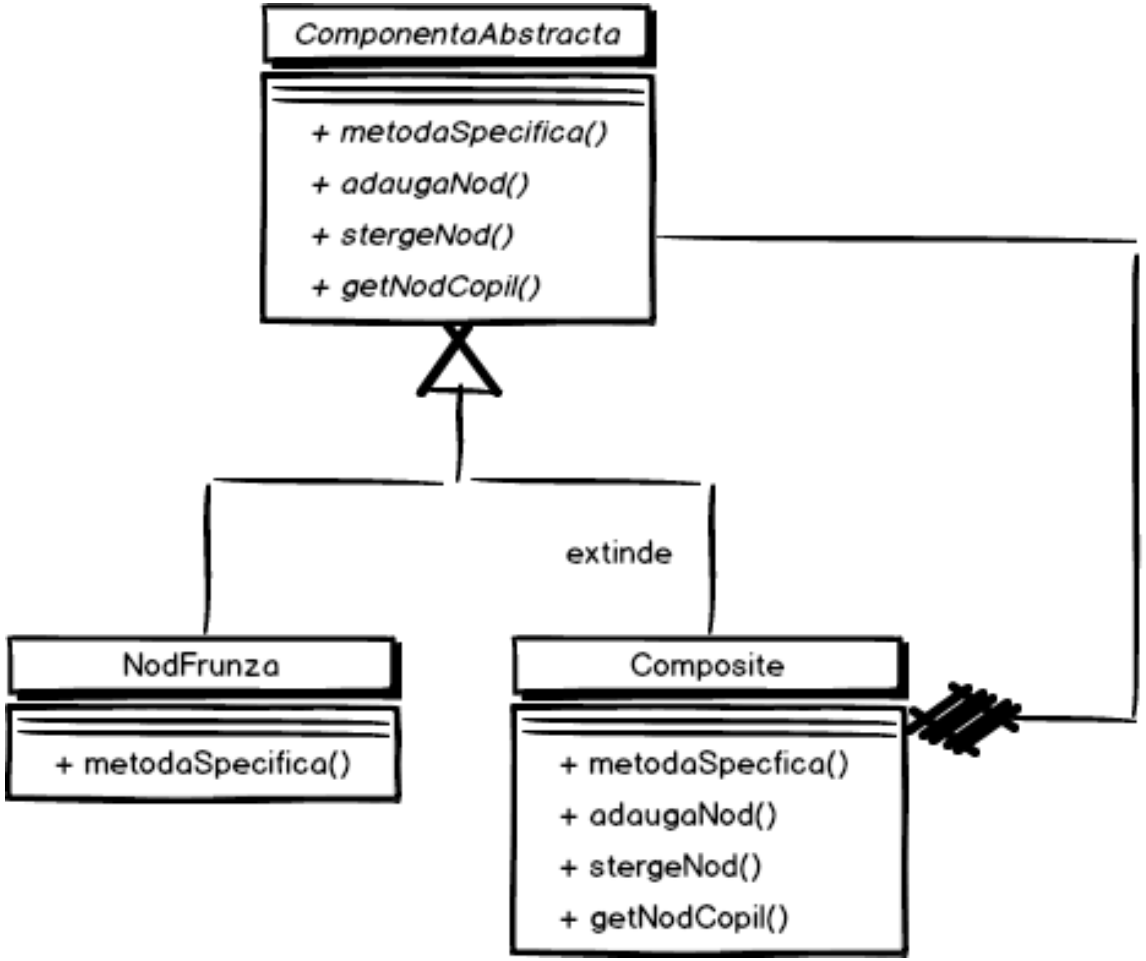
- Soluția conține o mulțime de clase aflate in relație ierarhică ce trebuie tratate unitar
- Se construiesc structuri arborescente în care nodurile intermediare și cele frunză sunt tratate unitar



COMPOSITE - Scenariu



COMPOSITE - Diagrama



COMPOSITE - Componente

- **`Componenta`**
  - conține descrierea abstractă a tuturor componentelor din ierarhie
  - Descrie interfața obiectelor aflate in compoziție
- **`NodFrunza`**
  - Reprezintă nodurile frunza din compoziție
  - Implementează toate metodele
- **`Composite`**
  - Reprezintă o componentă compusa – are noduri fiu
  - Implementează metode prin care sunt gestionate nodurile fiu

COMPOSITE - Avantaje

- Framework-ul nu se rescrie
- Permite gestiunea facila a unor ierarhii de clase ce conțin atât primitive cat si obiecte compuse
- Codul devine mai simplu deoarece obiectele din ierarhie sunt tratate unitar
- Adăugarea de noi componente care respecta interfața comună nu ridica probleme suplimentare

Modelul FLYWEIGHT

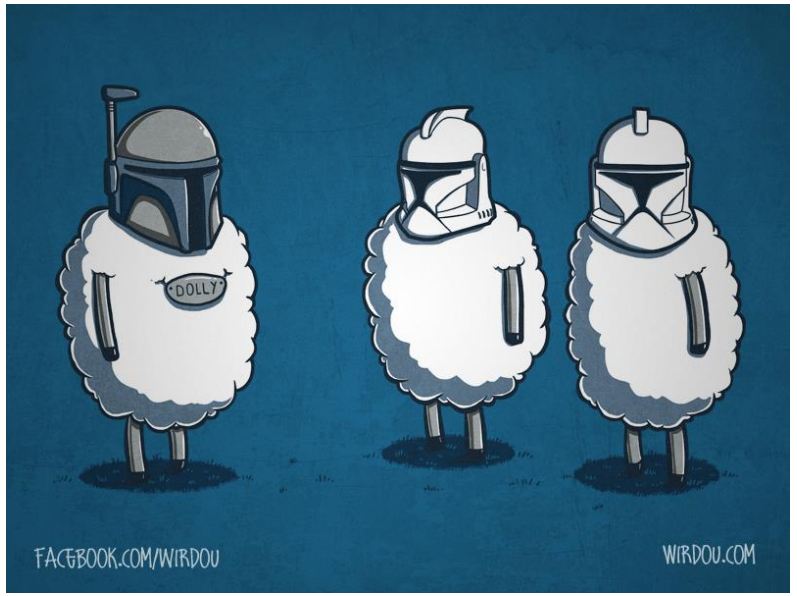
Modele structurale

FLYWEIGHT – Scenariu problemă

ACME Inc. dezvolta un editor de texte ca soluție alternativa la soluțiile cunoscute. In faza de testare s-a observat ca pe măsura ce crește dimensiunea textului, crește și memoria ocupată de această aplicația. Ritmul de creștere este unul anormal, destul de rapid , iar in final generează întârzieri între momentul tastării unui caracter si cel al afișării. Teste pe aceasta zona au arătat ca exista o legătura între numărul de caractere tastate si numărul de obiecte.

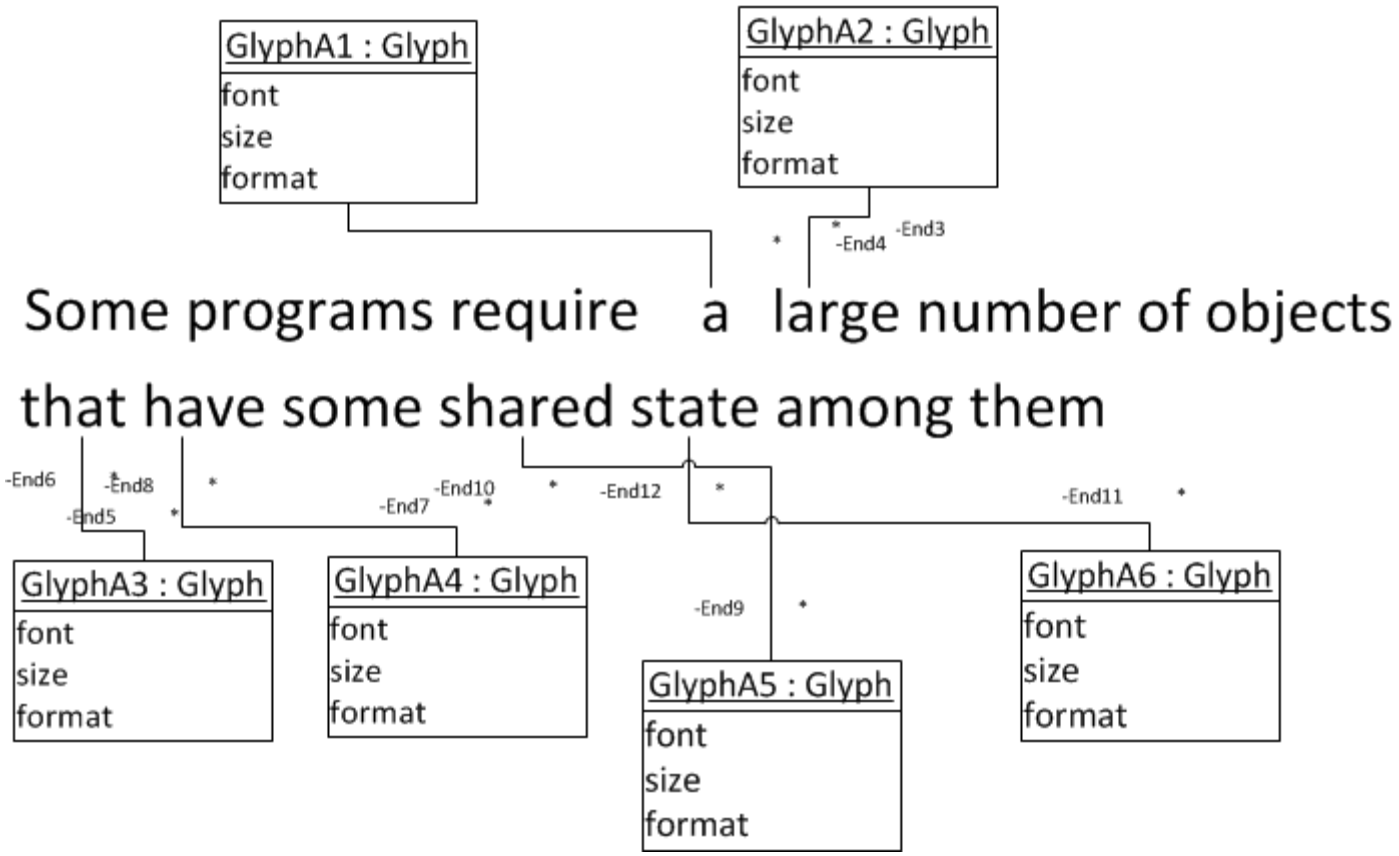
FLYWEIGHT - Problema

- Soluția generează o mulțime de obiecte cu o structura interna complexa si care ocupa un volum mare de memorie
- Obiectele au atribute comune însă o parte din starea lor variază; memoria ocupata de ele poate fi minimizata prin partajarea stării fixe intre ele
- Starea obiectelor poate fi gestionat prin structuri externe iar numărul de obiecte efectiv create poate fi minimizat
- Utilizarea unui obiect înseamnă reîncărcarea stării lui variabile într-un obiect existent



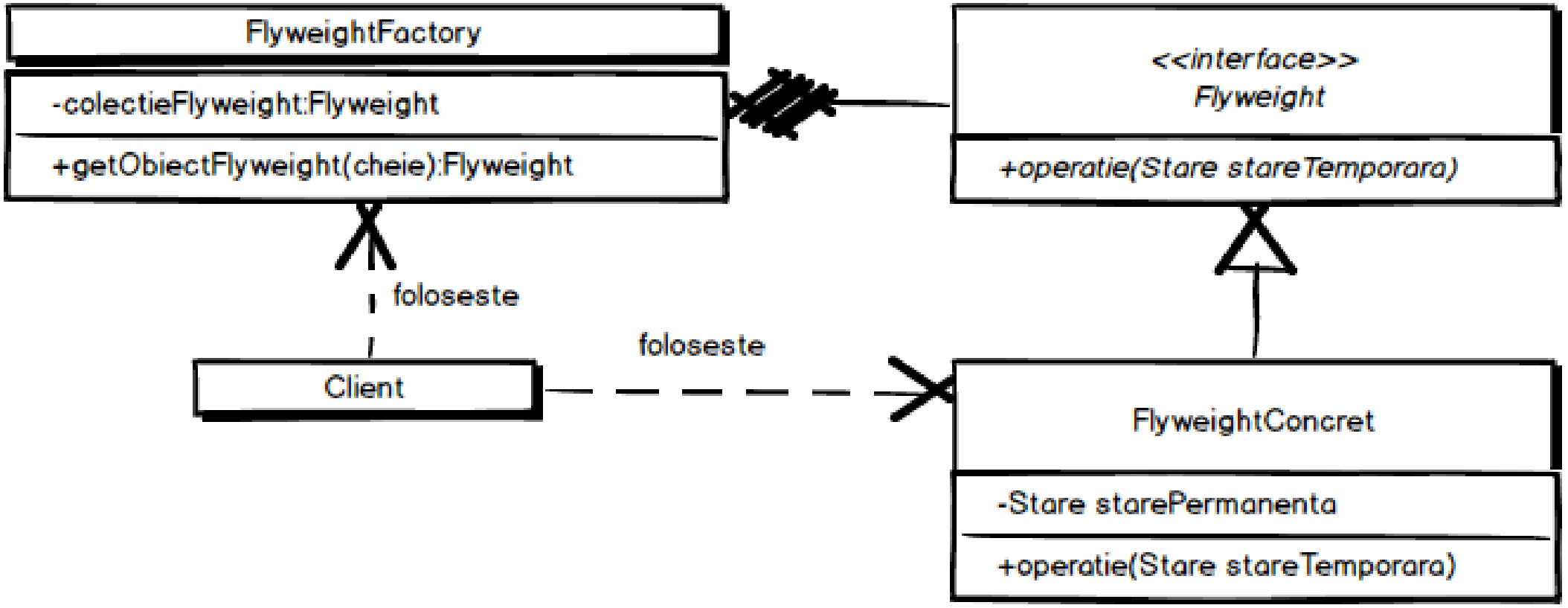
Attack of Clones

FLYWEIGHT - Scenariu



Not known author source

FLYWEIGHT - Diagrama



FLYWEIGHT - Componente

- **`Flyweight`**
  - Interfață ce permite obiectelor să primească valori ce fac parte din starea lor si să facă diferite procesări pe baza acesteia;
- **`FlyweightFactory`**
  - Un pattern de tip *factory* ce construiește si gestionează obiecte de tip flyweight;
  - Menține o colecție de obiecte diferite astfel încât ele sa fie create o singură dată
- **`FlyweightConcret`**
  - Clasa implementează interfața de tip Flyweight si permite stocarea stării permanente (ce nu poate fi partajat) a obiectelor
  - Valori ce reprezintă o stare temporara partajata între obiecte sunt primite si procesate prin intermediul metodelor din interfață
- **`Client`**
  - Gestionează obiectele de tip *flyweight* si starea lor temporara

FLYWEIGHT - Avantaje si Dezavantaje

Avantaje:

- Se reduce memoria ocupata de obiecte prin partajarea lor intre clienți sau a stării lor intre obiecte de același tip
- Pentru a gestiona corect partajarea obiectelor de tip *Flyweight* între clienți și fire de execuție, acestea trebuie sa *immutable*

Dezavantaje:

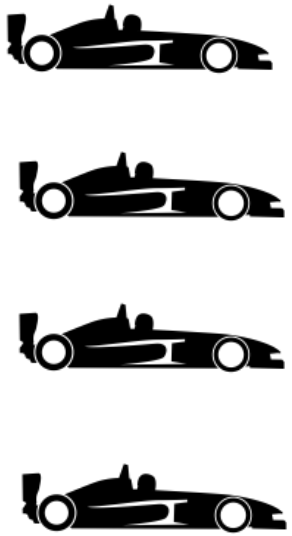
- Trebuie analizate clasele si contextul pentru a se determina ce reprezintă stare variabilă ce poate fi internalizată
- Efectele sunt vizibile pentru soluții in care numărul de obiecte este mare
- Nivelul de memorie redusa depinde de numărul de categorii de obiecte de tip *Flyweight*

FLYWEIGTH vs DECORATOR vs PROTOTYPE

- Prototype creează obiecte identice prin clonarea unui obiect existent. Obiectele create sunt identice (spațiu de memorie, atribute)
- Decorator permite modificarea la run-time a funcționalității obiectului fără a-i schimba definiția. Obiectele sunt create în mod normal și apoi *decorate* la execuție
- Flyweight permite stocarea și crearea obiectelor prin partajarea unui obiect parțial ce conține doar partea comună (atribute și metode). Obiectele flyweight partajează partea comună însă își gestionează partea specifică

FLYWEIGTH vs DECORATOR vs PROTOTYPE

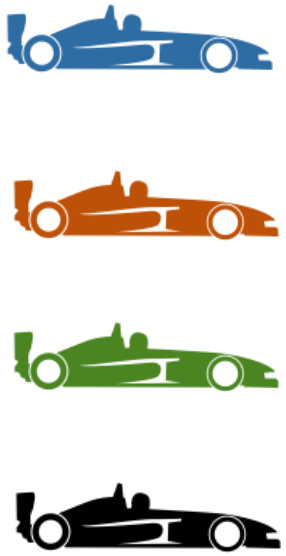
PROTOTYPE



Prototip inițial

DECORATOR

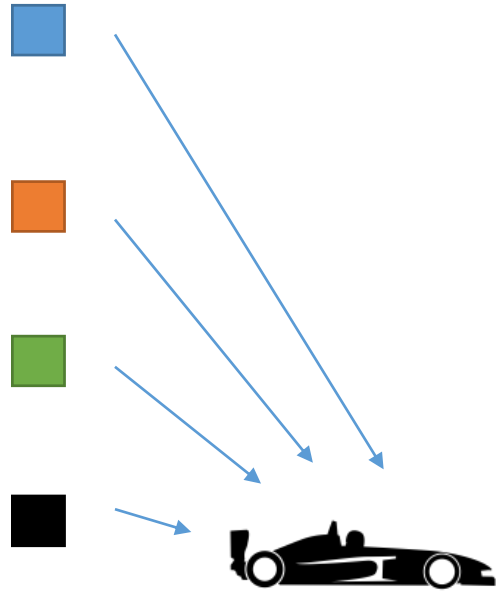
Decorate la run-time cu altă culoare



Obiect default

FLYWEIGTH

Setări particulare pentru instanțe



Flyweight

FLYWEIGHT - Șabloane asemănătoare

- ***Adapter*** – modifica interfața obiectului
- ***Decorator*** – adaugă dinamic funcții noi la comportamentul obiectului

- ***Façade*** – asigura o interfață simplificata

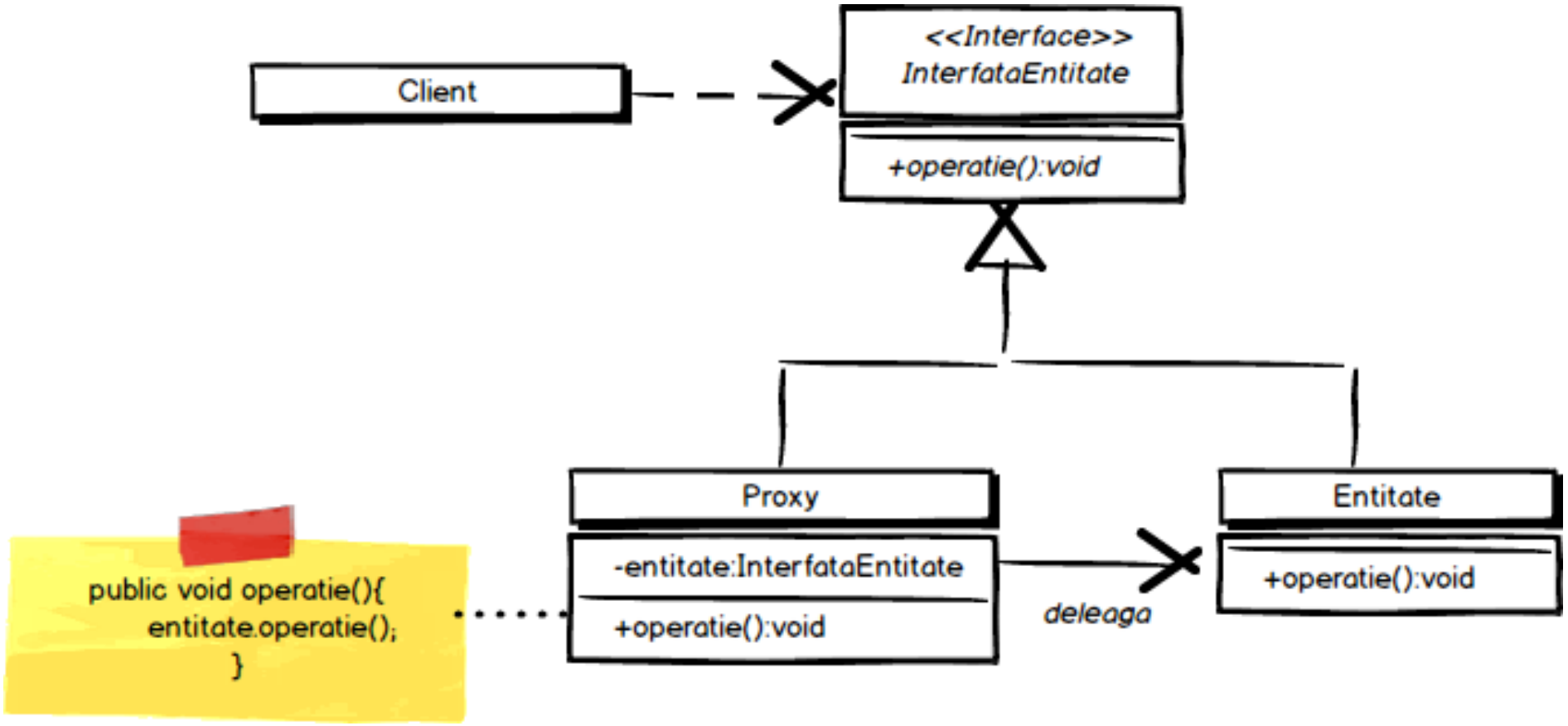
Modelul PROXY

Structural Design-Patterns

PROXY - Problema

- Interconectarea de API-uri diferite aflate pe aceeași mașină sau in rețea
- Definirea unei interfețe între framework-uri diferite

PROXY - Diagrama



PROXY - COMPONENTE

- **`InterfataEntitate`**
  - Definește interfața obiectului real la care se face conectarea
  - Interfața este implementata si de proxy astfel încât să se poată conecta la obiecte
- **`Proxy`**
  - Gestionează referința către obiectul real
  - Implementează interfața obiectului real
  - Controlează accesul la obiectul real
- **`Entitate`**
  - Obiectul real către care proxy-ul are legătura

PROXY - Tipuri de PROXY

- **Virtual Proxies**: gestionează crearea si inițializarea unor obiecte costisitoare; acestea pot fi create doar atunci când e nevoie sau partajează o singura instanță intre mai mulți client;
- **Remote Proxies**: asigura o instanță virtuala locala pentru un obiect aflat la distanta – Java RMI.
- **Protection Proxies**: controlează accesul la metodele unui obiect sau la anumite obiecte.
- **Smart References**: gestionează referințele către un obiect

Șabloane asemănătoare

- ***Adapter*** – modifică interfața obiectului
- ***Decorator*** – adaugă dinamic funcții noi la comportamentul obiectului
- ***Strategy (Behavioral)*** – modifică comportamentul obiectului
- ***Façade*** – asigura o interfață simplificată
- ***Composite*** – agregă mai multe obiecte asemănătoare pentru o gestiune unitară

Behavioral Design-Patterns

Strategy, Observer, Chain of Responsability, Template, State, Command, *Iterator*, Memento

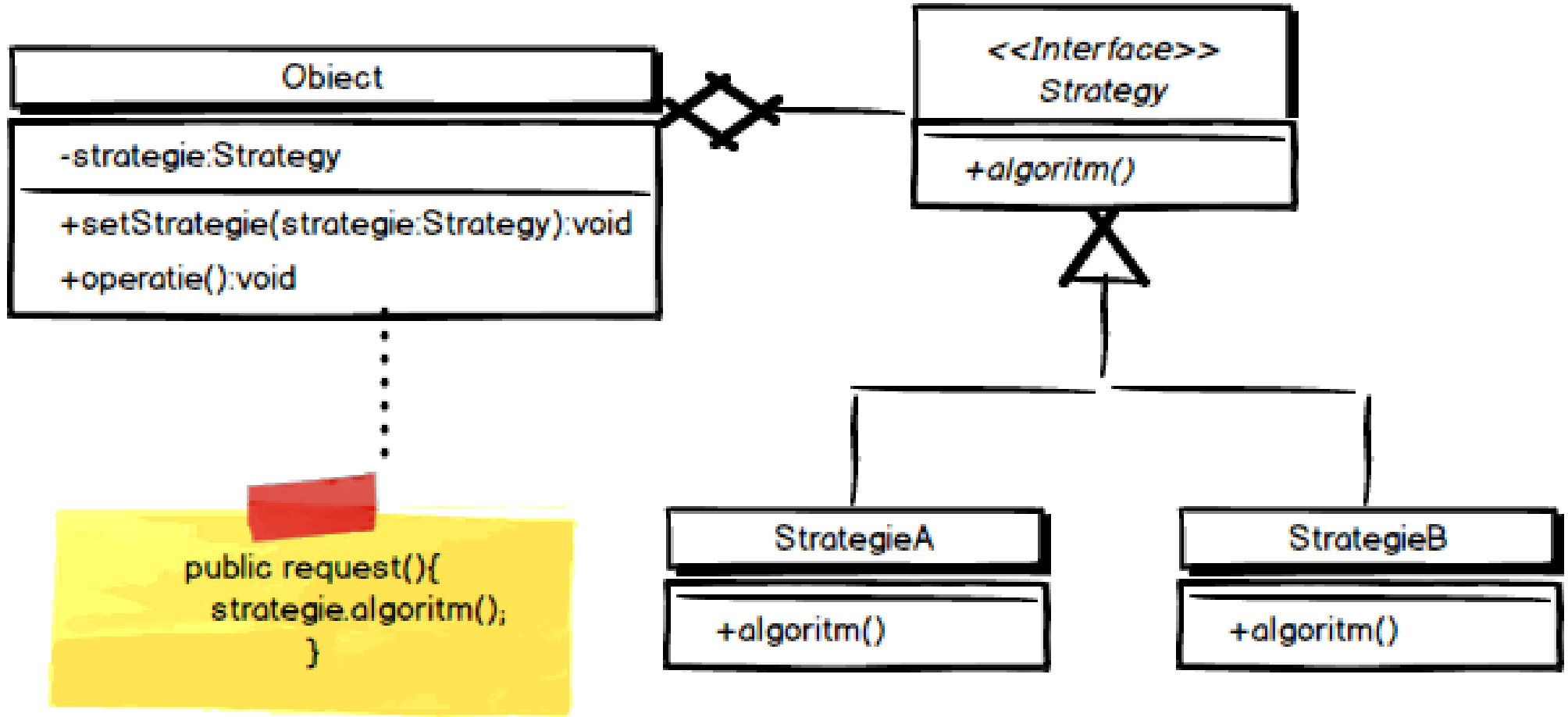
Modelul STRATEGY

Modele comportamentale

STRATEGY - Problema

- Alegerea la *run-time* a algoritmului/funcției care sa fie utilizata pentru procesarea unor date;
- Algoritmul se poate alege pe baza unor condiții descrise la execuție in funcție de contextul datelor de intrare
- Clasa existenta nu trebuie sa fie modificata
- Utilizarea unei abordări tradiționale, prin includerea in clasa a tuturor metodelor posibile, duce la ierarhii complexe ce sunt greu de gestionat. Derivarea adaugă comportament nou doar la compilare

STRATEGY - Diagrama



STRATEGY - Componente

- **`IStrategy`**
  - clasa abstracta ce definește interfața obiectelor ce pot oferi noi funcții/algoritmi de prelucrare;
- **`StrategyA`**
  - definește obiecte ce furnizează soluții pentru prelucrarea datelor;
- **`Obiect`**
  - gestionează o referința de tip **IStrategy** către obiectul care va oferi funcția/algoritmul;
  - Gestionează datele/contextual ce necesită prelucrare;

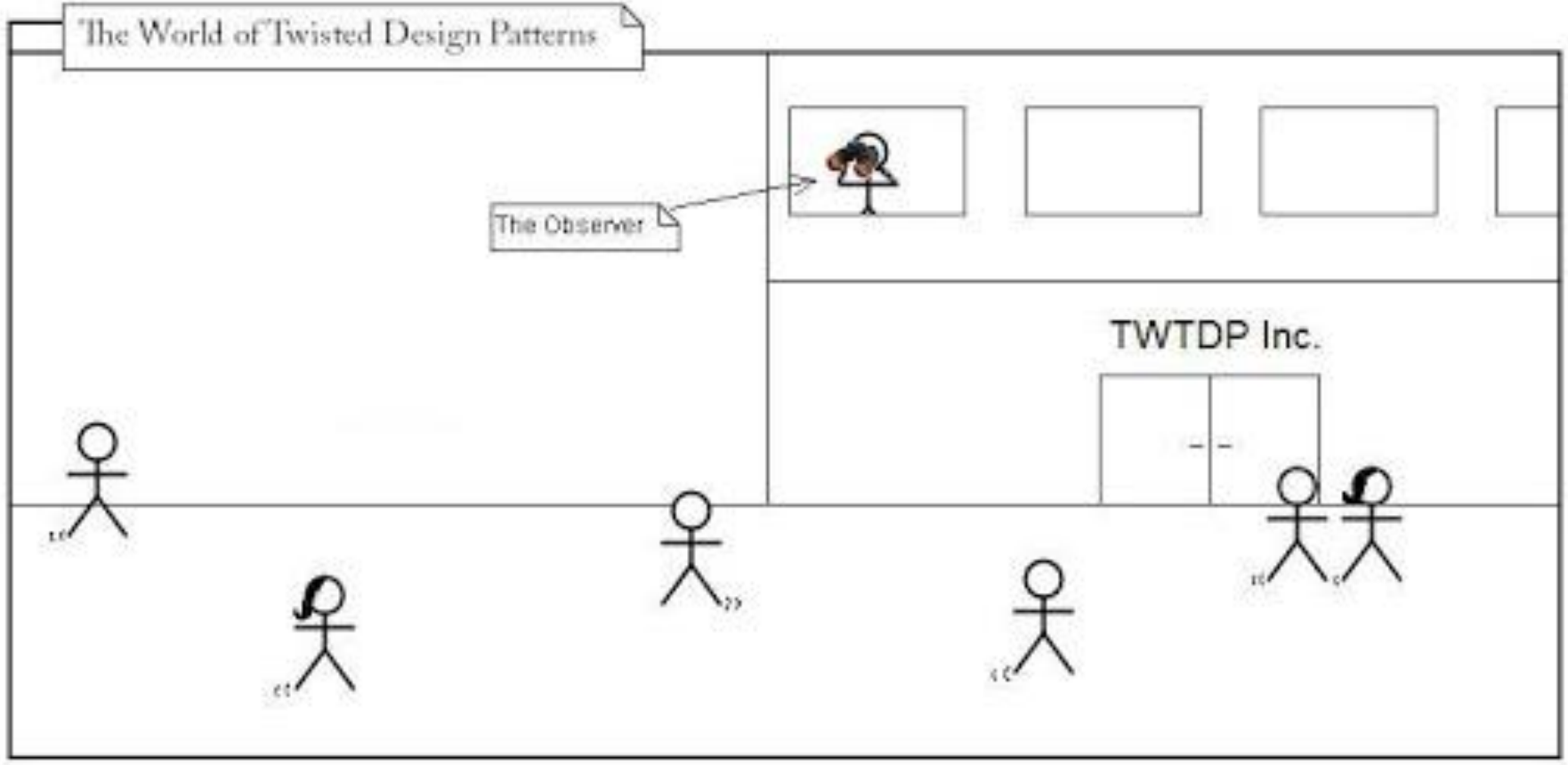
STRATEGY - Avantaje

- Alegerea metodei de prelucrare a datelor se face dinamic, la run-time
- Este permisa definirea de noi algoritmi independent de modificarea clasei ce gestioneaza datele
- Nu impune limite privind un numar maxim de functii/algoritmi ve pot fi folositi

Modelul OBSERVER

Modele comportamentale

OBSERVER



http://umlcomics.blogspot.ro/2010/03/world-of-twisted-design-patterns.html

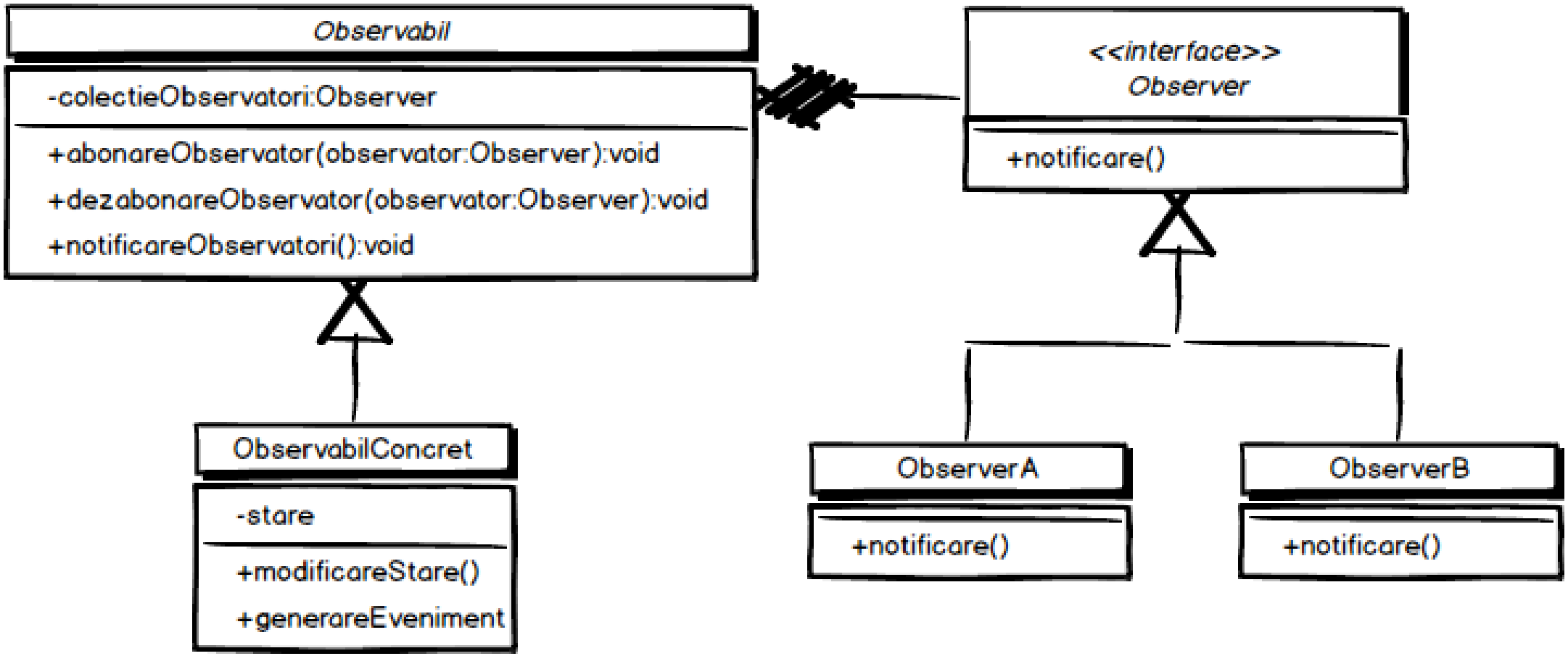
OBSERVER - Problema

- Exista componente care trebuie sa fie notificate la producerea unui eveniment
- Gestiunea evenimentelor la nivel de interfață
- Componentele se abonează/înregistrează la acel eveniment – modificare de stare/acțiune
- La producerea unui eveniment pot fi notificate mai multe componente

OBSERVER - Avantaje

- Externalizarea/delegarea funcțiilor către componente “observator” care dau soluții la anumite evenimente independent de proprietarul evenimentului
- Conceptul integrat în pattern-ul arhitectural *Model View Controller* (MVC)
- Implementează conceptul POO de *loose coupling* – obiectele sunt interconectate prin notificări și nu prin instanțieri de clase și apeluri de metode

OBSERVER - Diagrama



OBSERVER - Componente

- **`Observabil`**
  - clasa abstracta ce definește interfața obiectelor gestionează evenimente si care sunt observabile;
- **`ObservabilConcret`**
  - definește obiecte ce permit abonarea unor observatori;
- **`Observator`**
  - Interfață ce definește modalitatea in care sunt notificați observatorii;
  - Permite gestiunea mai multor observatori
- **`ConcreteDecorator`**
  - Implementează funcții concrete care sunt executate in urma notificării;

OBSERVER - Metode de comunicare

2 modele de notificare a observatorului la modificarea stării:

- **Push** – obiectul trimite toate detaliile observatorului
- **Pull** – obiectul doar notifică observatorul și acesta cere datele când are nevoie de ele

Modelul CHAIN OF RESPONSABILITY

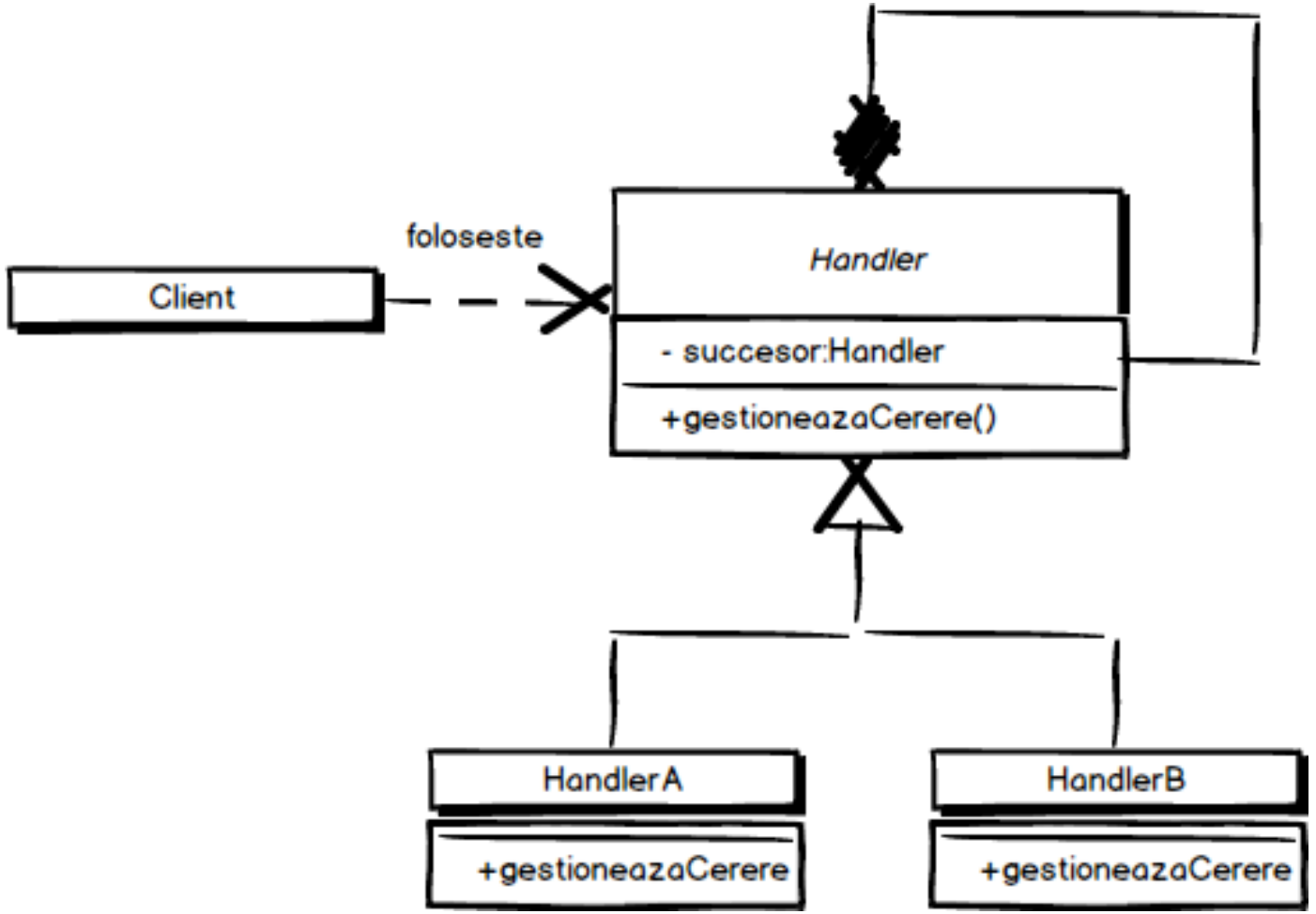
Modele comportamentale

CHAIN OF RESPONSABILITY - Problema

- Tratarea unui eveniment sau a unui obiect se face diferit in funcție de starea acestuia
- Gestiunea tuturor cazurilor ar implica o structură complexă care să verifice toate cazurile particulare
- Există legături de dependență între cazurile de utilizare: execuția unui caz poate implica ignorarea celorlalte sau tratarea următorului caz

http://www.leahy.com.au/

CHAIN OF RESPONSABILITY - Diagrama



CHAIN OF RESPONSABILITY - Componente

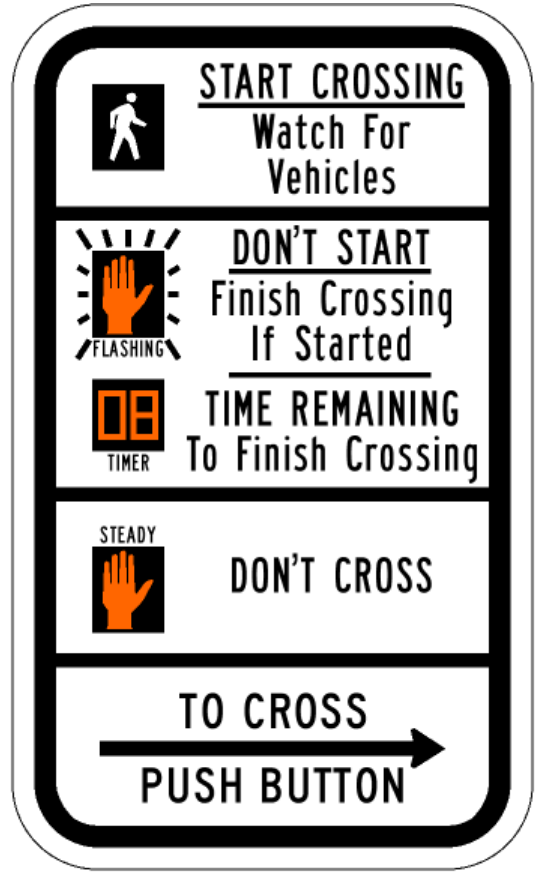
- **`Handler`**
  - clasa abstracta ce definește interfața obiectelor ce gestionează cererea de procesare a evenimentului;
- **`HandlerA`**
  - definește obiecte concrete ce formează secvența de tratare a notificării;
- **`Client`**
  - Generează evenimentul sau notifica primul obiect din secvența de obiecte;

Modelul STATE

Modele comportamentale

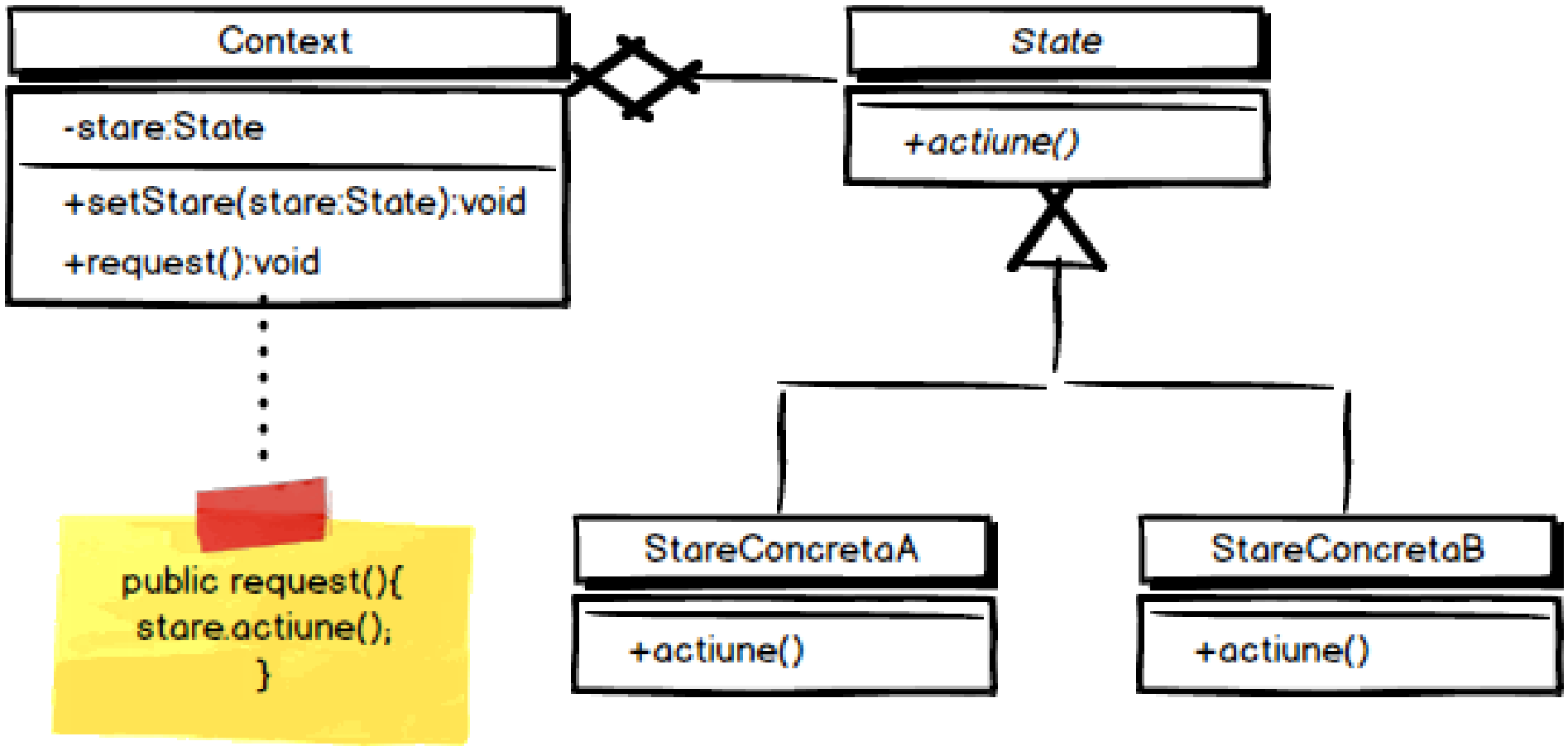
STATE - Problema

- Aplicația tratează un anumit eveniment diferit în funcție de starea unui obiect
- Numărul de stări posibile poate să crească și tratarea unitară a acestora poate să influențeze complexitatea soluției
- Modul de tratare a acțiunii este asociat unei anumite stări și este încapsulat într-un obiect de stare



R10-3e

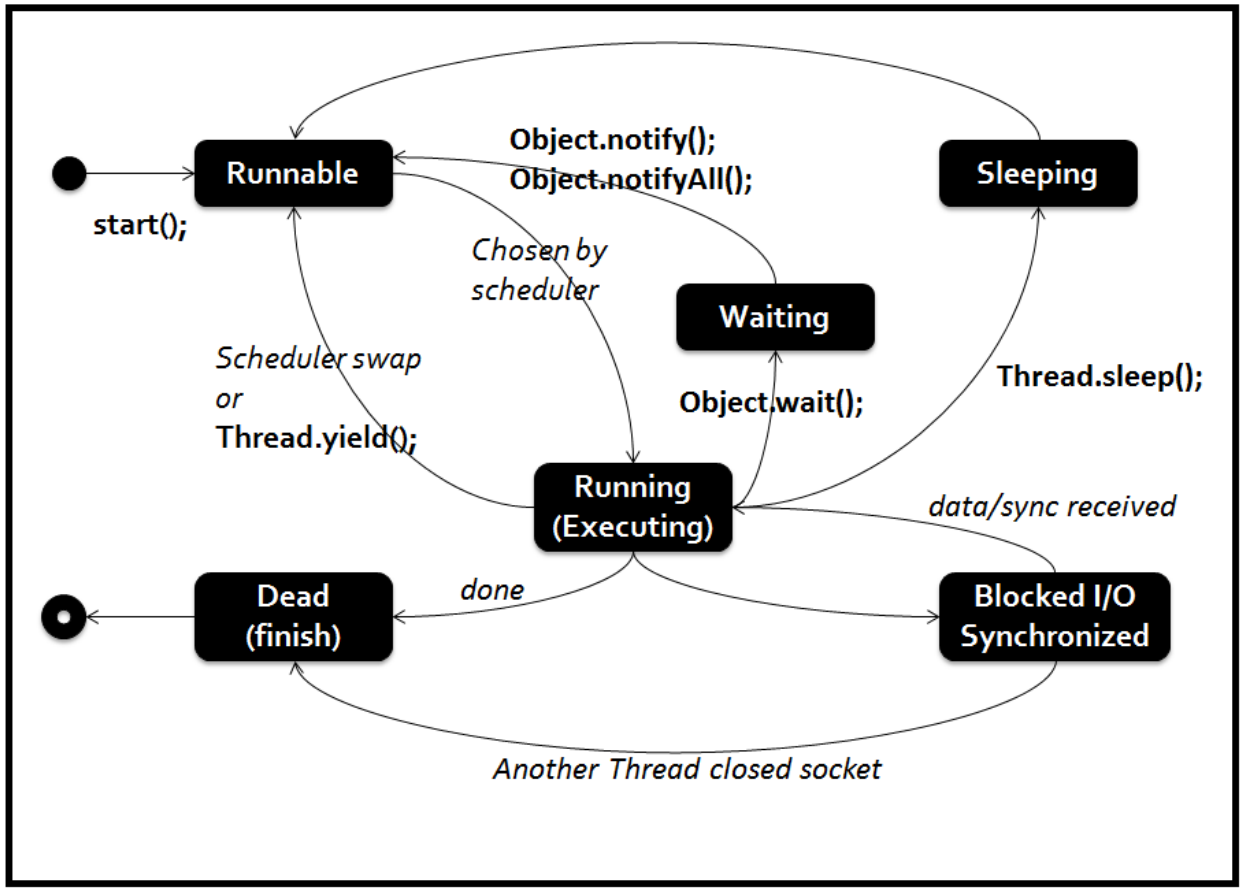
STATE - Diagrama



STATE - Componente

- **`State`**
  - clasa abstracta ce definește interfața obiectelor ce gestionează implementarea unor acțiuni în funcție de stare;
- **`StareConcretaA, StareConcretaB`**
  - definește obiecte concrete ce implementează metodele de acțiunii aferente stării respective;
- **`Context`**
  - gestionează referința de tip State și folosește metodele acesteia pentru a implementa diferite prelucrări în funcție de stare;

STATE Machine



Ciclu de viata Java Thread

https://codexplo.wordpress.com/2012/10/20/state-diagram-of-java-thread/

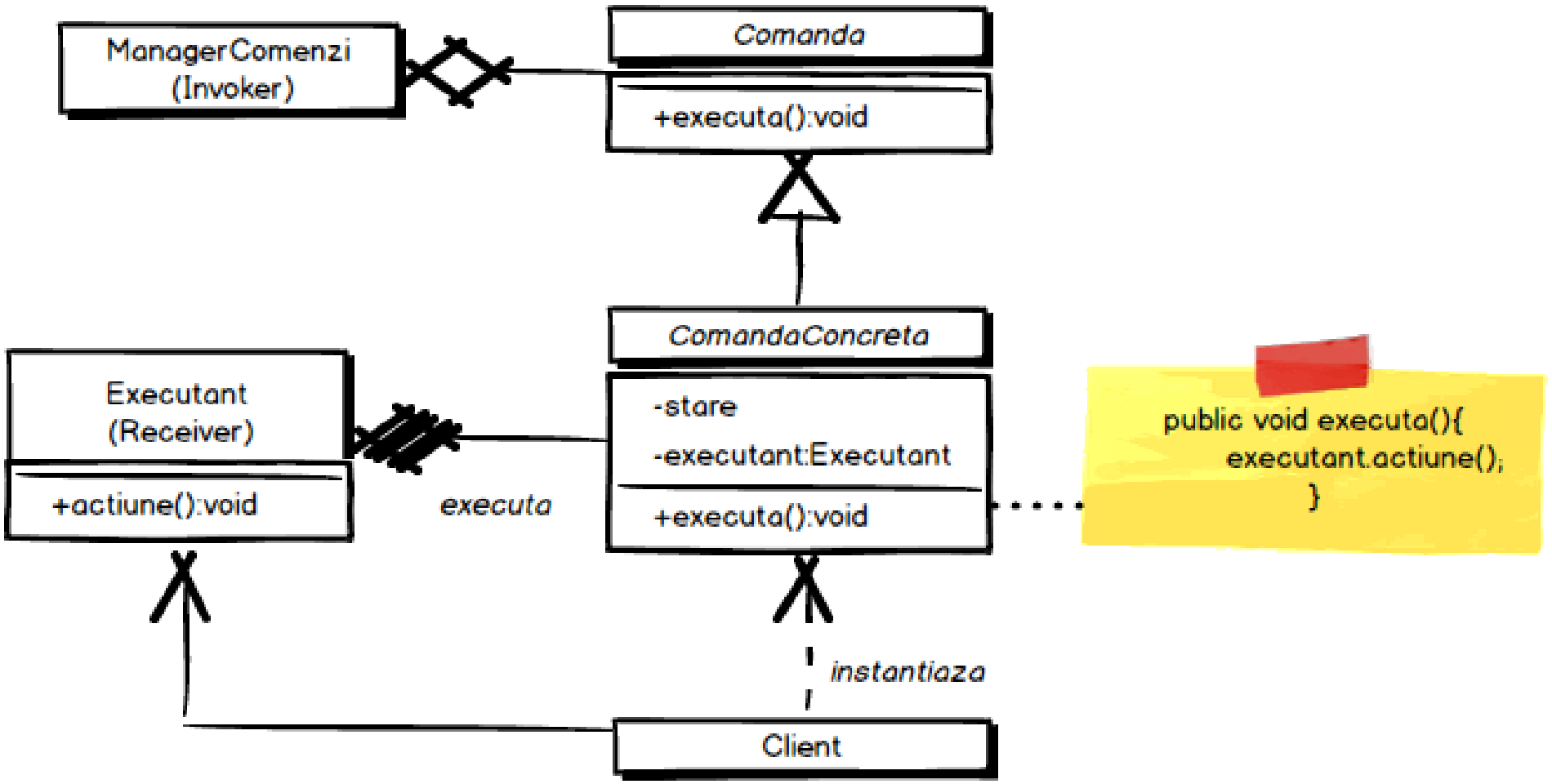
Modelul COMMAND

Modele comportamentale

COMMAND - Problema

- Aplicația definește acțiuni parametrizabile ce pot fi executate mai târziu fără a solicita clientului cunoașterea detaliile interne necesare execuției.
- Pentru a nu bloca clientul, se dorește ca aceste acțiuni să fie definite și trimise spre execuție fără a mai fi gestionate de client
- Se decuplează execuția întârziată (ulterioara) a unei acțiuni de proprietar. Din punctul acestuia de vedere, acțiunea a fost deja trimisa spre execuție.
- Concept echivalent cu macro-urile. Obiectul de tip *command* încapsulează toate informațiile necesare execuției acțiunii mai târziu de către responsabil
- Clientul este decuplat de cel ce executa acțiunea

COMMAND - Diagrama



COMMAND - Componente

- **`Comanda`**
  - Definește interfața necesară execuției acțiunii + alte detalii;
- **`ComandaConcreta`**
  - Extinde interfața comenzii si implementează metoda prin care este controlat *Receiver-ul*
  - Reprezintă legătura dintre *Receiver* si acțiune
- **`Client`**
  - Creează un obiect *ComandaConcreta* si setează *Receiver-ul* acestuia;
- **`Invoker (ManagerComenzi)`**
  - Creează comanda și cere îndeplinirea acțiunii;
- **`Receiver (Executant)`**
  - Obiectul care este responsabil cu execuția acțiunii;

COMMAND - Scenariu

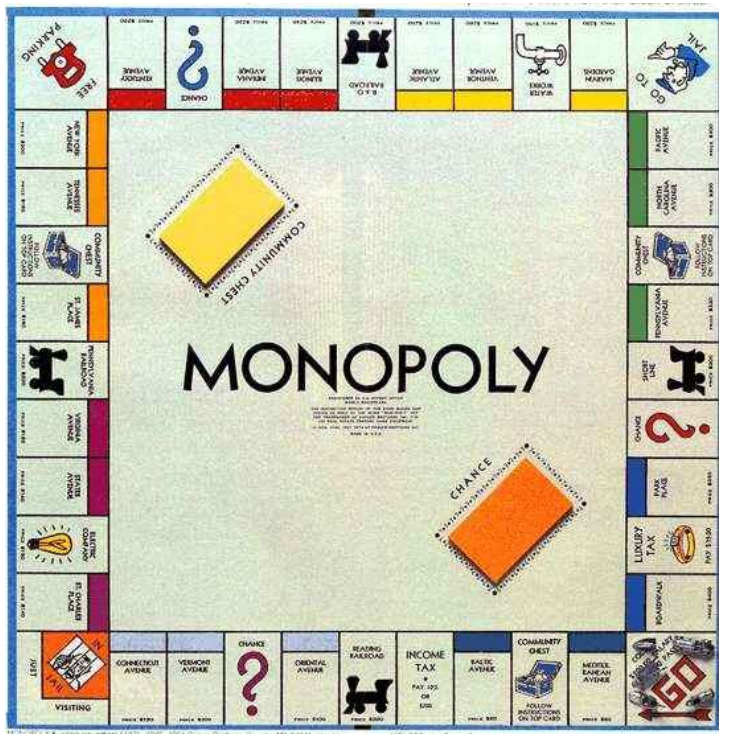
ACME Inc. dezvolta o soluție software pentru un restaurant, astfel încât chelnerul să poată prelua comenzile direct pe telefonul mobil. Comenzile sunt preluate de la client și ele sunt create pe loc, fiind automat alocat bucătarul specializat pe acel fel de mâncare, ingredientele folosite si alte cerințe speciale ale clientului. Aceste detalii sunt puse de aplicație, fără a fi necesara intervenția chelnerului care doar selectează felul de mâncare solicitat. Comenzile sunt trimise bucătarilor la finalizarea comenzii pentru masa respectiva, urmând să fie executate in funcție de gradul de încărcare al fiecărui bucătar.

Modelul TEMPLATE METHOD

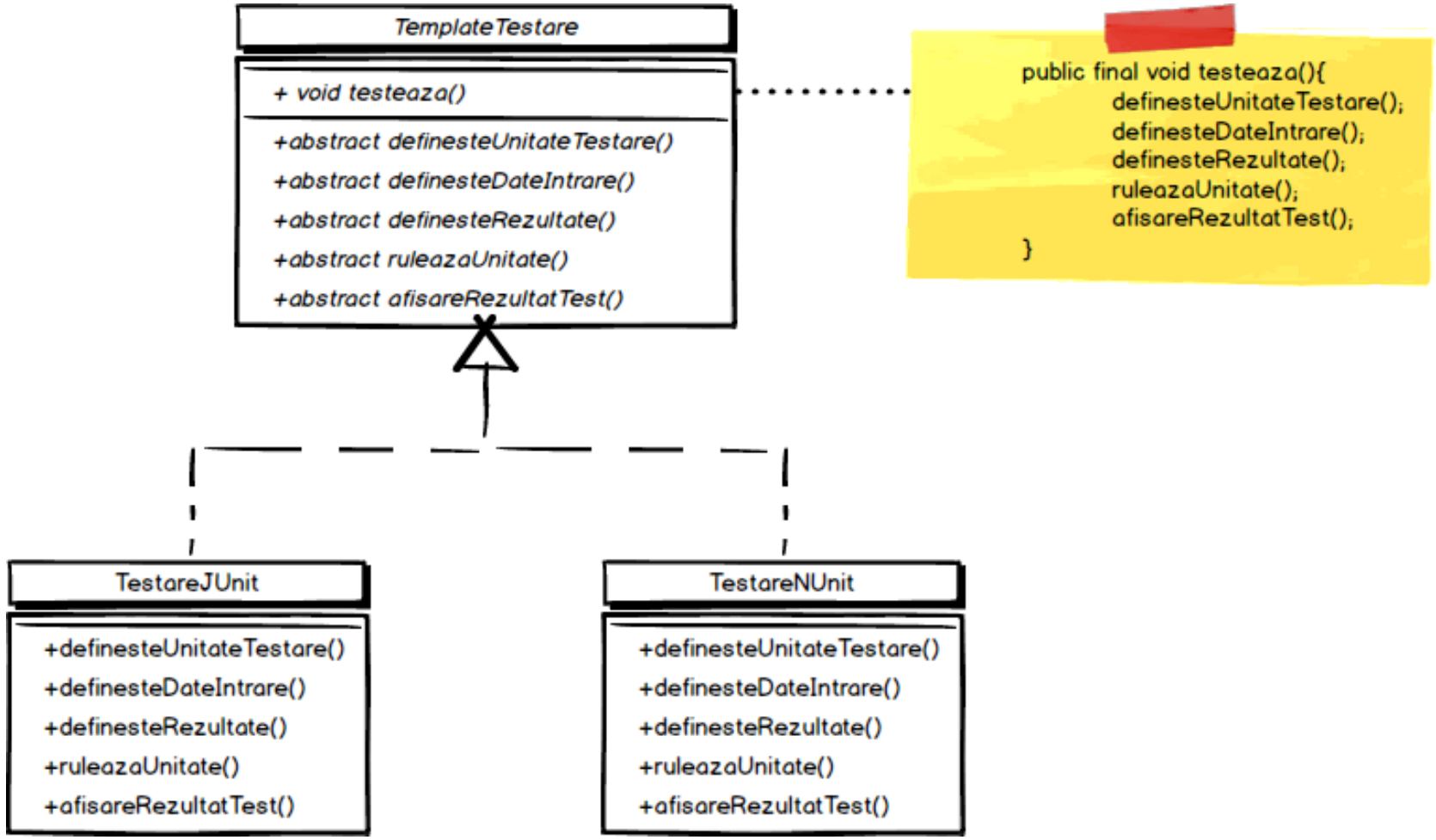
Modele comportamentale

TEMPLATE METHOD - Problema

- Implementarea unui algoritm presupune o secvență predefinita si fixa de pași
- Metoda ce definește schema algoritmului – metoda template
- Pot fi extinse/modificate metodele care implementează fiecare pas însă schema algoritmului nu este modificabilă
- Implementează principiul Hollywood: "*Don't call us, we'll call you.*“
- Metodele concrete de definesc pașii algoritmului sunt apelate de metoda template



TEMPLATE METHOD - Diagrama



TEMPLATE - Componente

- **`TemplateTestare`**
  - Clasa abstracta ce definește interfața unui obiect de tip TemplateTestare si modalitatea în care sunt executate metodele
  - Conține o metod template ce implementează șablonul de execuție a interfeței și care nu se supradefinește
- **`TestareJUnit`**
  - Definește interfața obiectului într-o situație concretă
- **`TestareNUnit`**
  - Definește interfața obiectului într-o situație concretă

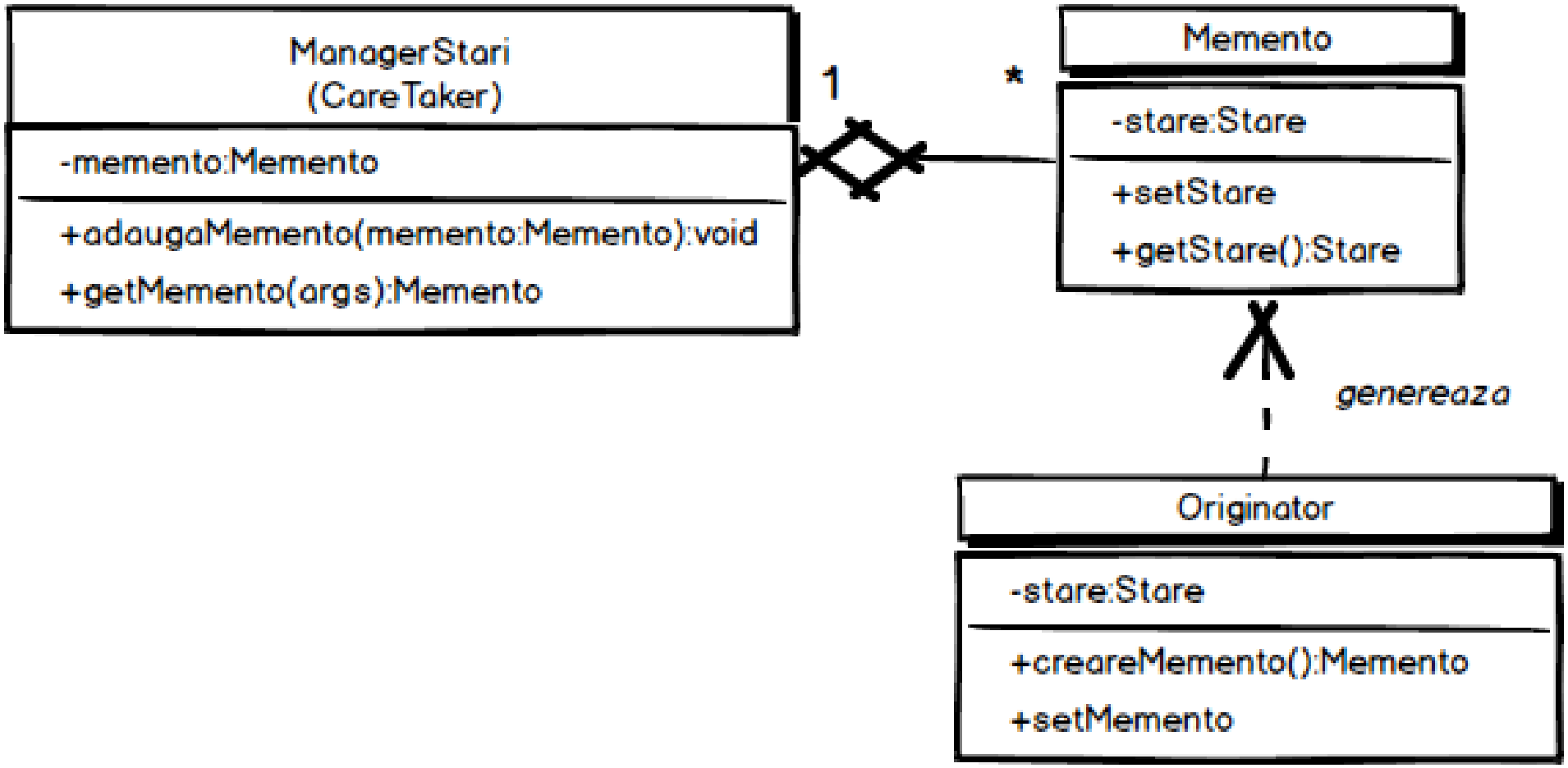
Modelul MEMENTO

Modele comportamentale

MEMENTO - Problema

- Aplicația trebuie sa permită salvarea stării unui obiect
- Imaginile stării obiectului pentru diferite momente sunt gestionate separat
- Obiectul își poate restaura starea pe baza unei imagini anterioare

MEMENTO - Diagrama



MEMENTO - Componente

- **`Memento`**
  - Gestionează starea interna a obiectului *Originator* pentru un anumit moment; Este creat de *Originator* si este gestionat de *Caretaker*
- **`Originator`**
  - Obiectul a cărui stare este urmărita; Poate genera un Memento cu starea lui la momentul respective. Poate sa își reface starea pe baza unui Memento
- **`ManagerStari (Caretaker)`**
  - Gestionează obiectele de tip Memento fără a avea acces pe conținutul acestora;

Recapitulare

- **Creaționale**:
  - *Factory* – creează obiecte dintr-o familie
  - *Builder* – creează obiecte setând anumite atribute
  - *Singleton* – creează o unică instanță
- **Structurale:**
  - *Adapter* – adaptează un API (o interfață) la altul
  - *Composite* – gestionează o ierarhie de obiecte
  - *Decorator* – atribuie la run-time funcționalitate nouă unui obiect existent
  - *Façade* – simplifica execuția (apelarea) unui scenariu complex
  - *Flyweight* – gestionează eficient mai multe instanțe (clone) ale unui set redus de modele

Recapitulare

- **Comportamentale**:
  - *Strategy* – schimba la run-time funcția executată
  - *Observer* – execută o acțiune când are loc un eveniment sau un observabil își schimba starea
  - *Chain of Responsibility* – gestionează o secvență de acțiuni ce pot procesa un eveniment sau un obiect
  - *Command* – gestionează realizarea întârziată a unei acțiuni
  - *Memento* – gestionează stările anterioare ale unui obiect
  - *State* – stabilește tipul acțiunii în funcție de starea obiectului
  - *Template* – gestionează un șablon fix de acțiuni