

Classe Problemes Setmana 10: Disseny: Intro Patrons de Disseny

Anna Puig

Enginyeria Informàtica

Facultat de Matemàtiques i Informàtica,

Universitat de Barcelona

Curs 2021/22

Temari

1	Introducció al procés de desenvolupament del software
2	Anàlisi de requisits i especificació
3	Disseny
4	Del disseny a la implementació
5	Ús de frameworks de testing

3.1	Introducció
3.2	Patrons arquitectònics
3.3	Criteris de Disseny: G.R.A.S.P.
3.4	Principis de Disseny: S.O.L.I.D.

3.5	Patrons de Disseny
-----	---------------------------

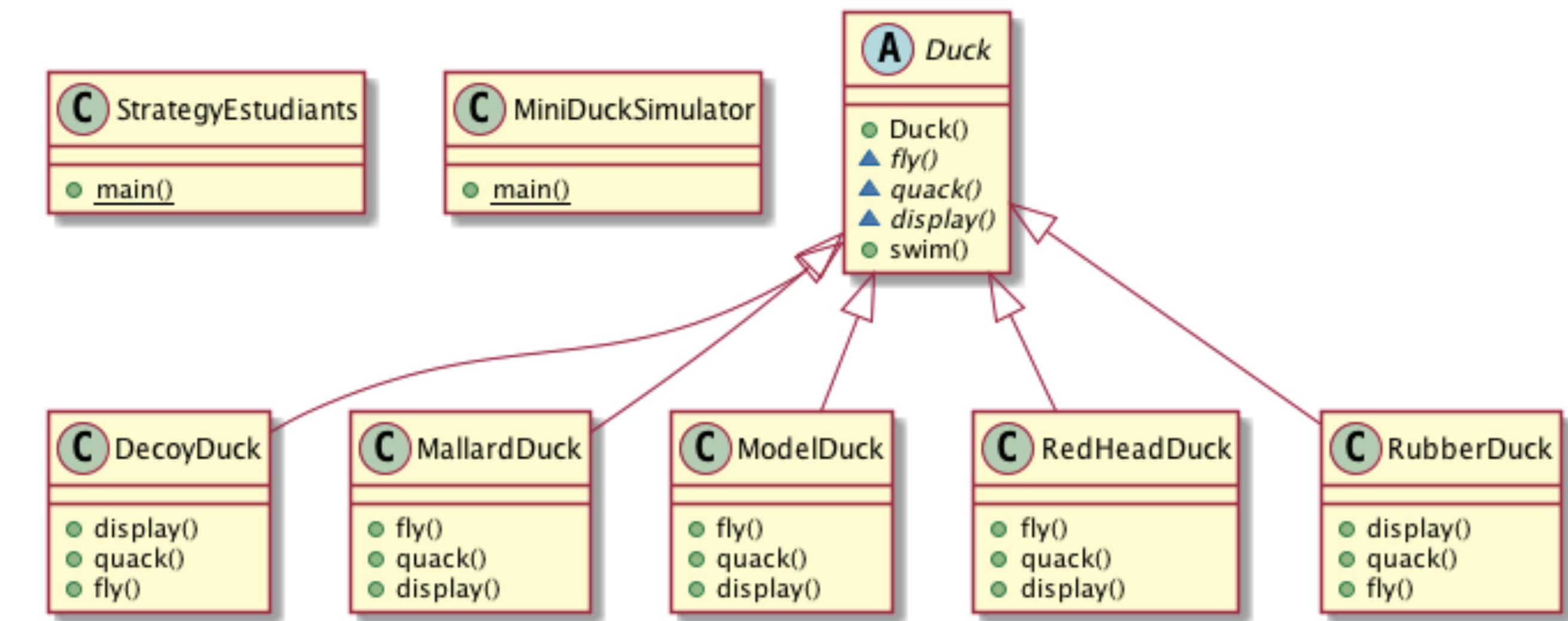
Passos per solucionar els problemes de patrons

Problema dels ànecs:

Es vol dissenyar una aplicació que simuli diferents tipus d'ànecs.

Els ànecs naden (swim), alguns parlen (quack) i alguns volen (fly). Com veureu hi ha una classe Duck, amb els mètodes quack(), swim(), fly() i display(). D'aquesta classe Duck, hereten les classes MallardDuck, ReadheadDuck, RubberDuck, DecoyDuck amb diferents implementacions dels mètodes fly() i quack().

Fixa't que quan volen, tots volen igual i, quan parlen, tots parlen igual. Identifiqueu quins principis S.O.L.I.D. vulnera. Com ho resoldríeu?



1. Problema a identificar

```
public abstract class Duck {  
    public Duck() {}  
    abstract void fly();  
    abstract void quack();  
    abstract void display();  
  
    public void swim() { System.out.println("All ducks float, even decoys!"); }  
}
```

```
public class MallardDuck extends Duck {  
    public void fly() { System.out.println("I'm flying!!"); }  
    public void quack() { System.out.println("Quack"); }  
    public void display() { System.out.println("I'm a real Mallard duck"); }  
}
```

```
public class DecoyDuck extends Duck {  
    public void display() { System.out.println("I'm a duck Decoy"); }  
    public void quack() { System.out.println("<< Silence >>"); }  
    public void fly() { System.out.println("I can't fly"); }  
}
```

1. Problema a identificar

```
public abstract class Duck {  
    public Duck() {  
    }  
    abstract void fly();  
    abstract void quack();  
    abstract void display();  
  
    public void swim() { System.out.println("All ducks float, even decoys!"); }  
}
```

```
public class MallardDuck extends Duck {  
    public void fly() { System.out.println("I'm flying!!"); }  
    public void quack() { System.out.println("Quack"); }  
    public void display() { System.out.println("I'm a real Mallard duck"); }  
}
```

```
public class DecoyDuck extends Duck {  
    public void display() { System.out.println("I'm a duck Decoy"); }  
    public void quack() { System.out.println("<> Silence ><"); }  
    public void fly() { System.out.println("I can't fly"); }  
}
```

1. Problema a identificar

```
public abstract class Duck {  
    public Duck() {  
    }  
    abstract void fly();  
    abstract void quack();  
    abstract void display();  
  
    public void swim() { System.out.println("All ducks float, even decoys!"); }  
}
```

```
public class MallardDuck extends Duck {  
    public void fly() { System.out.println("I'm flying!!"); }  
    public void quack() { System.out.println("Quack"); }  
    public void display() { System.out.println("I'm a real Mallard duck"); }  
}
```

```
public class DecoyDuck extends Duck {  
    public void display() { System.out.println("I'm a duck Decoy"); }  
    public void quack() { System.out.println("<< Silence >>"); }  
    public void fly() { System.out.println("I can't fly"); }  
}
```

Liskov!!

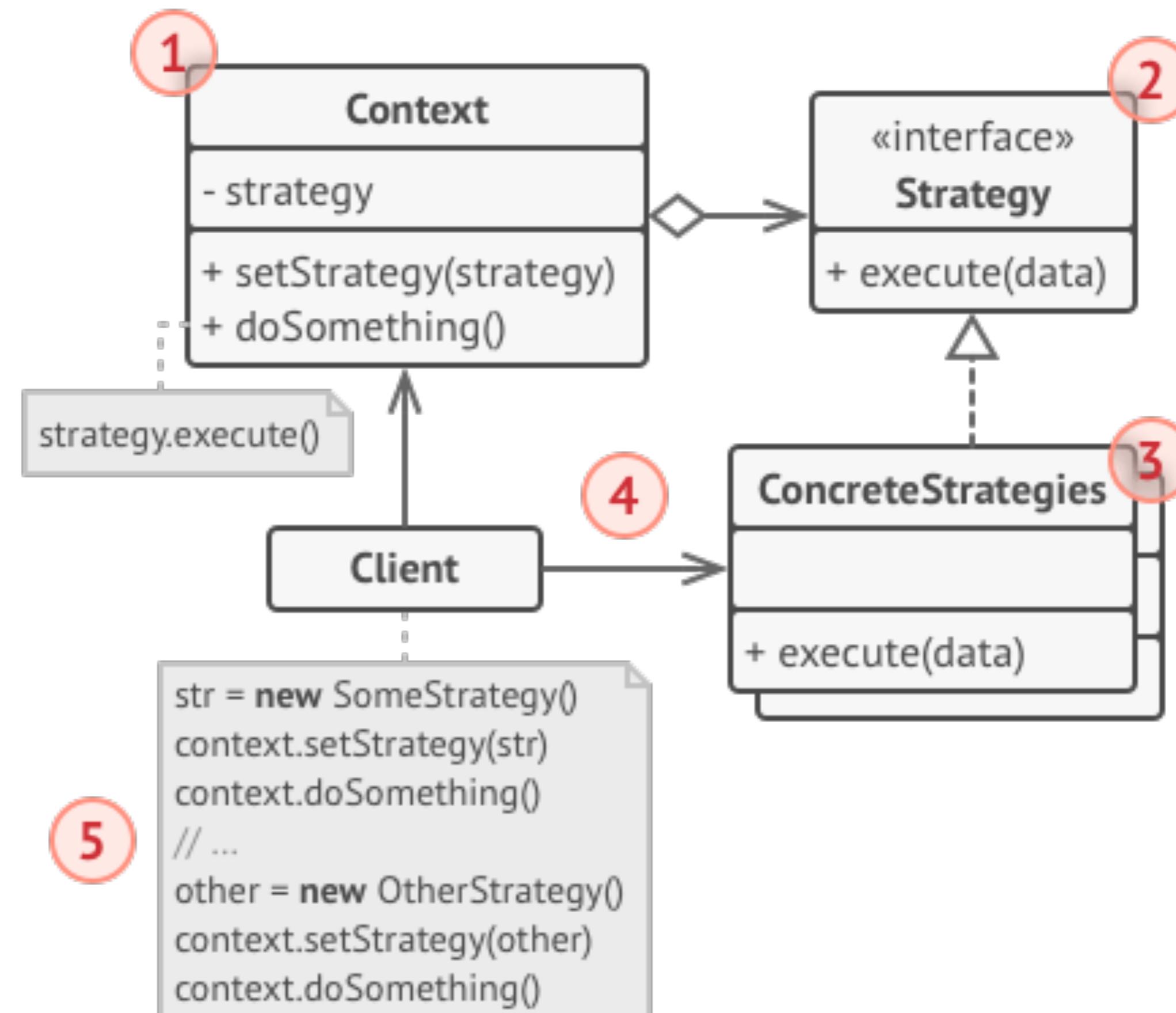


i repeticio de codi....!!

Patró Strategy

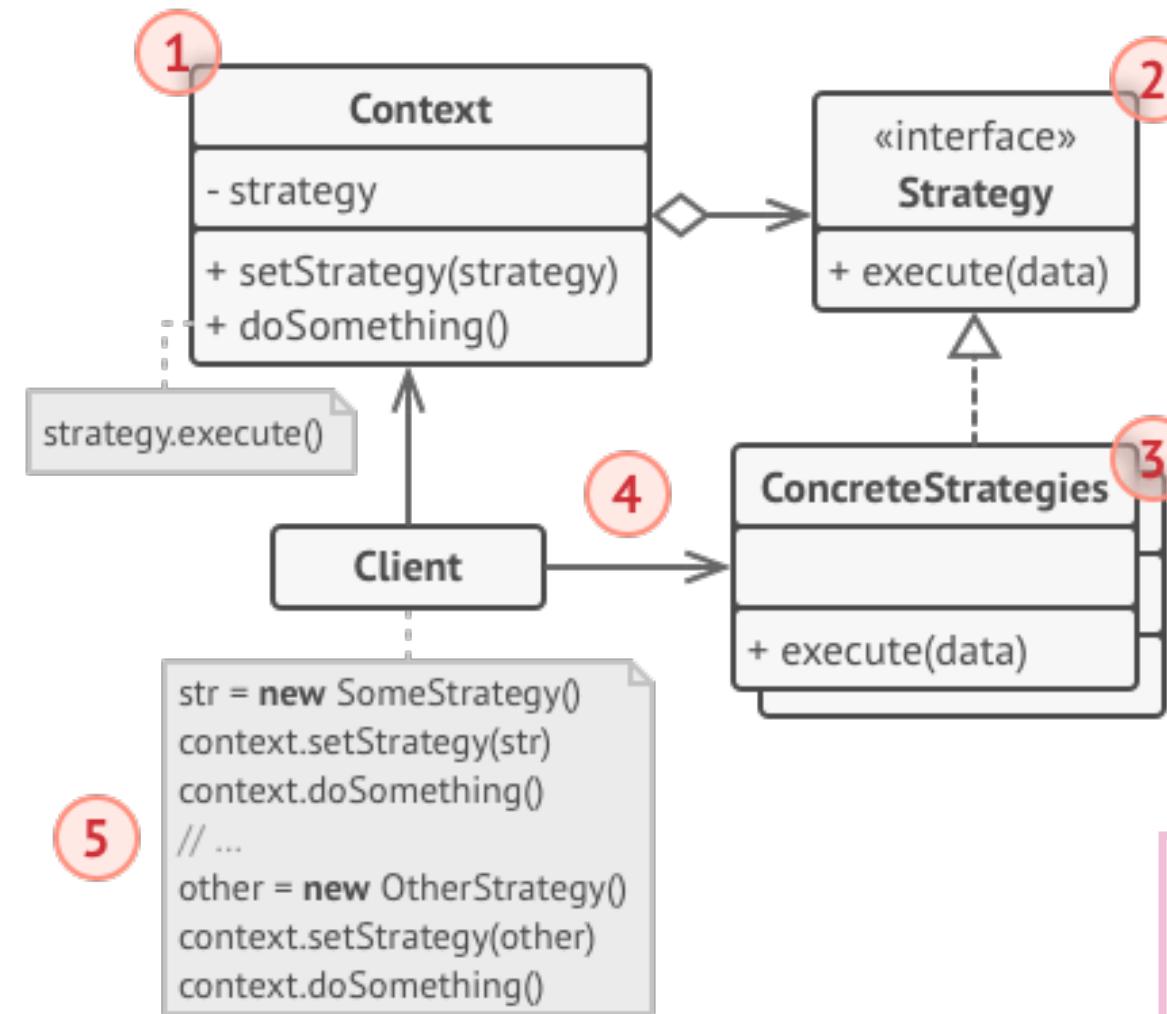
- **Strategy** – Defineix una família d'algorismes, els encapsula un a un i els fa intercanviables. Permet que l'algorisme canviï independentment del client que l'usa.

2. Patró a aplicar: Strategy



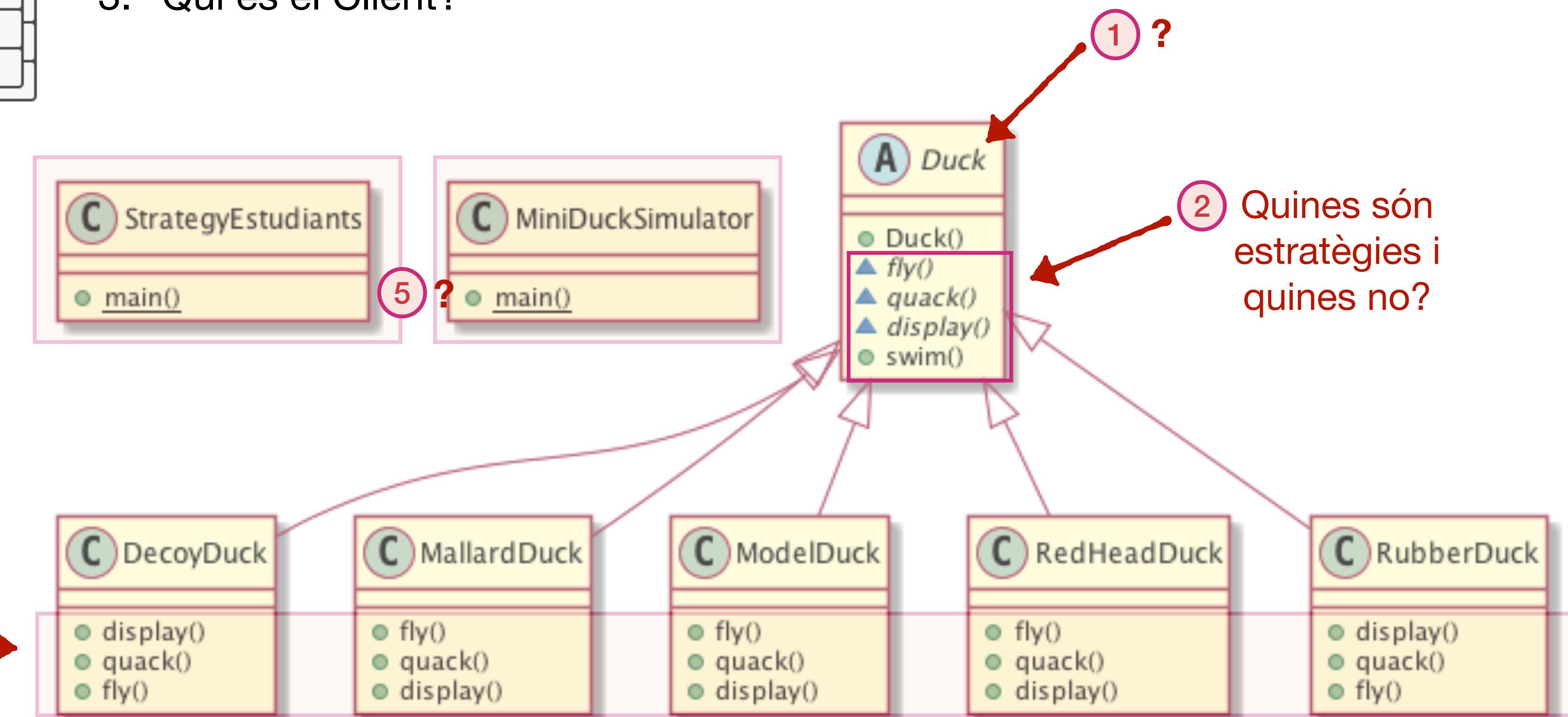
1. El **Context** no coneix res de les estratègies concretes, només les sap executar
2. La classe **Strategy** declara la interície comuna a tots els algorismes
3. Les implementacions concretes de les estratègies estan a les classes **ConcreteStrategies**
4. El **Client** crea l'estratègia concreta que vol fer servir
5. El **Client** passa al **Context** l'estratègia (`setStrategy`) i delega en el **Context** que l'executi.

3. Aplicar el patró: identificació de les classes del patró



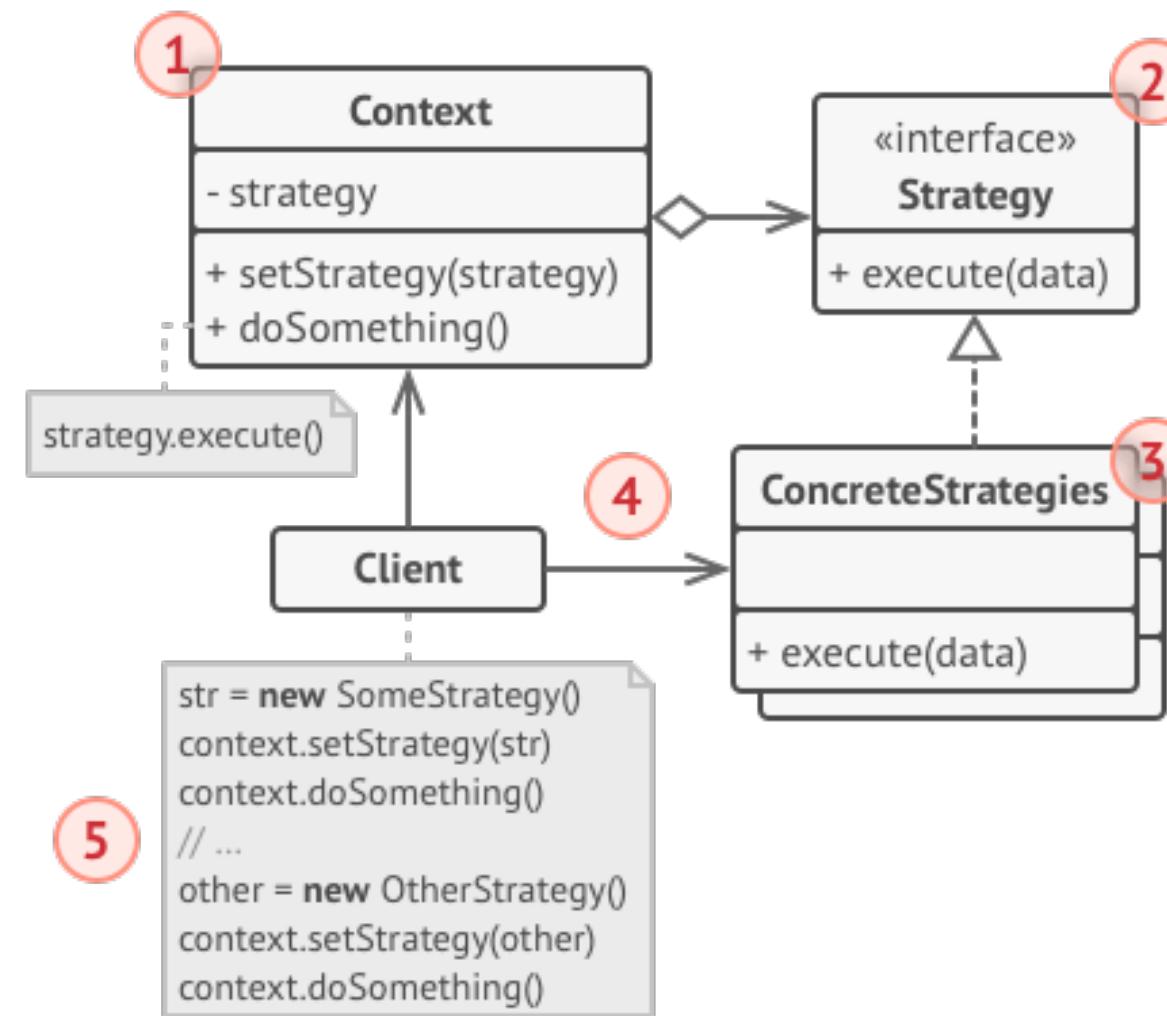
Contestar les preguntes:

1. Qui és el Context?
2. Quines són les Strategies?
3. Qui és el Client?



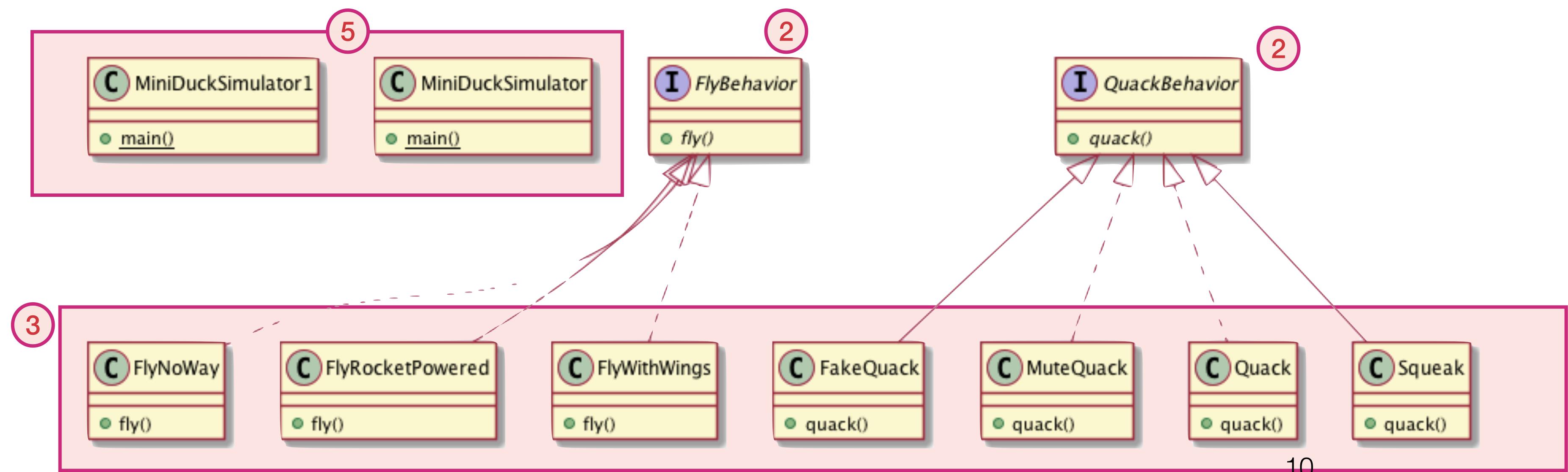
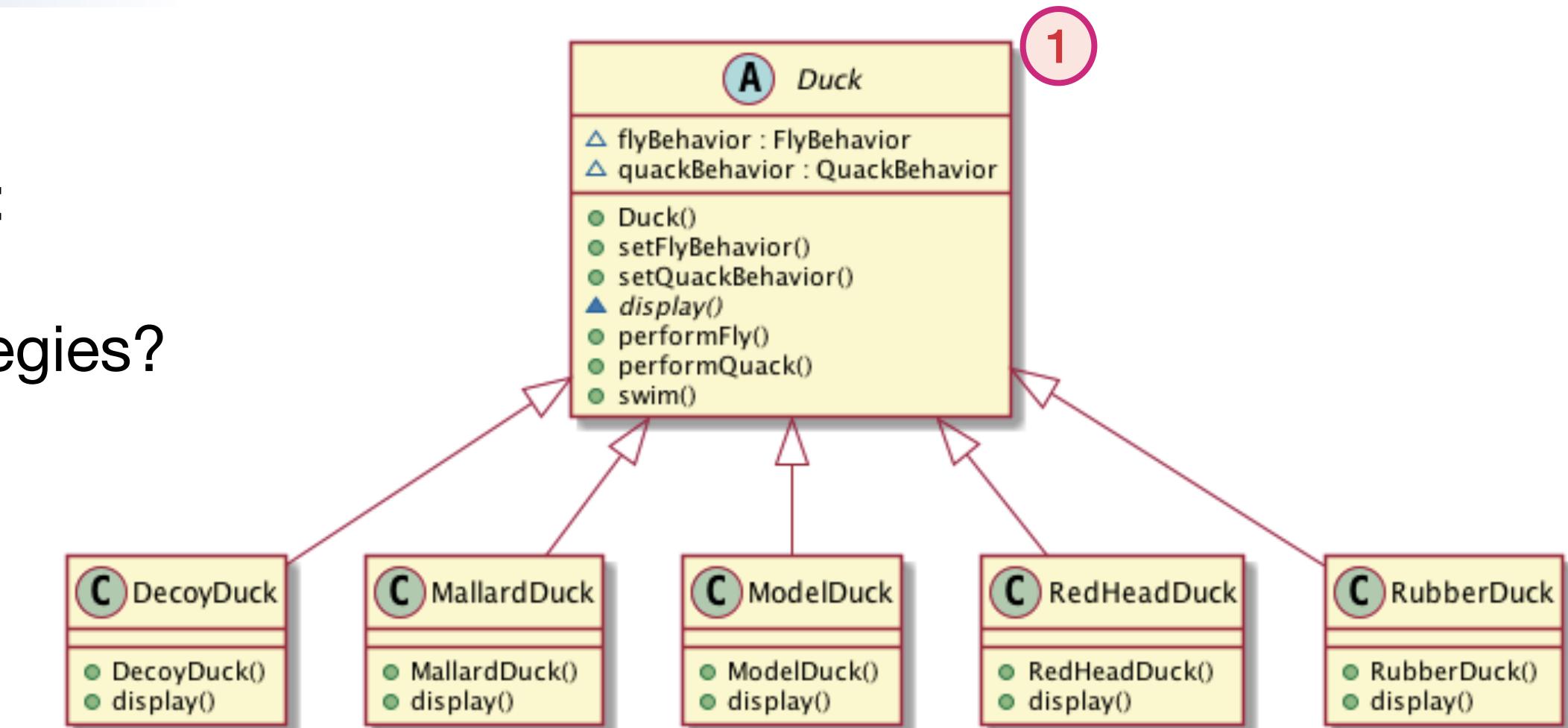
Quines són
estratègies i
quines no?

3. Aplicar el patró



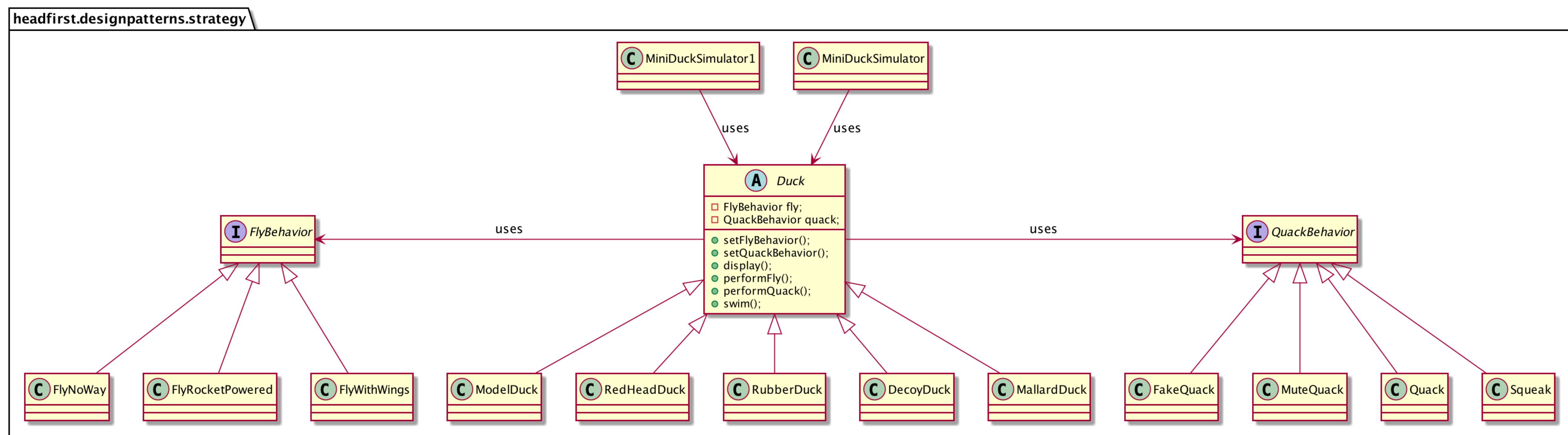
Contestar les preguntes:

1. Qui és el Context?
2. Quines són les Strategies?
3. Qui és el Client?



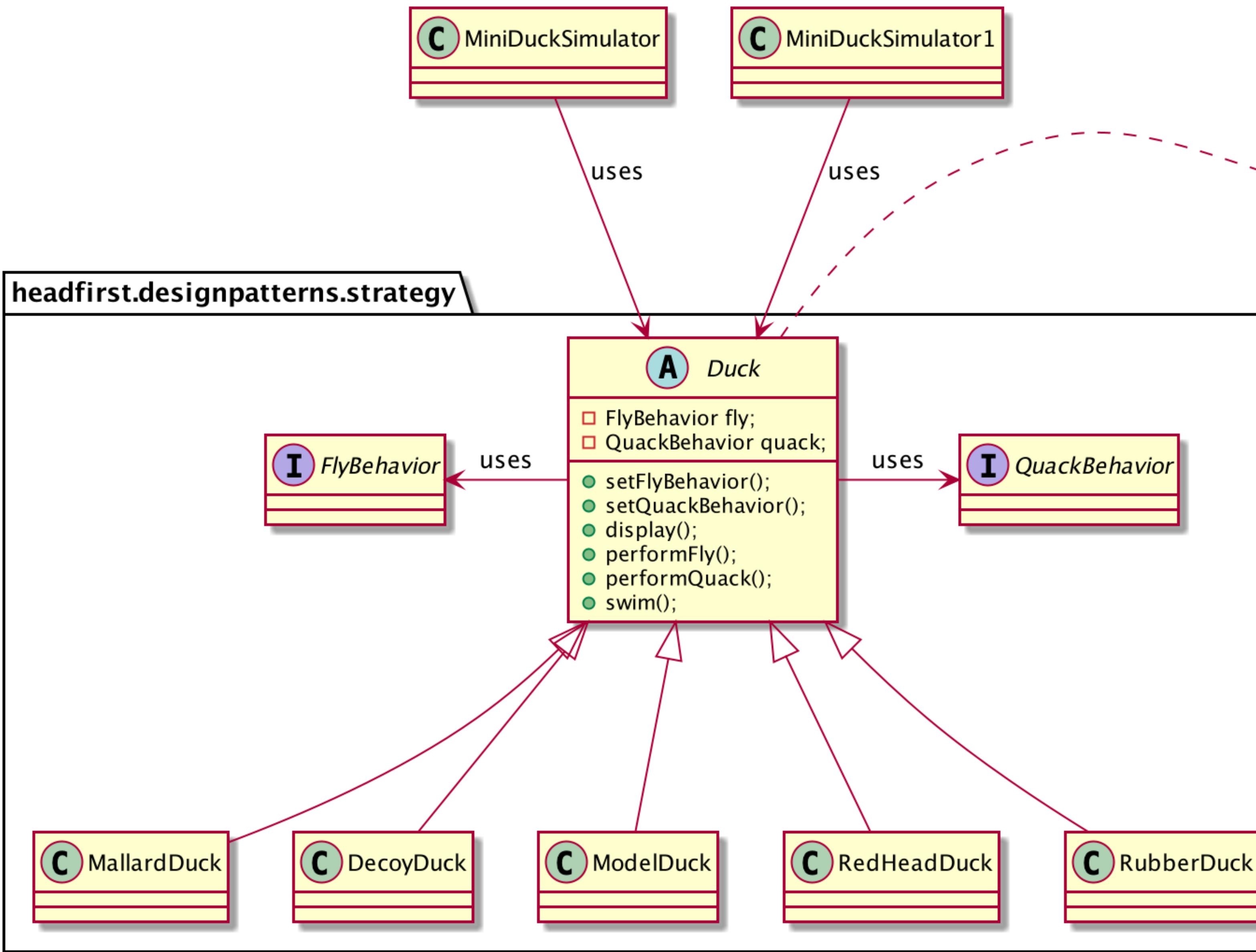
3. Aplicar el patrón

STRATEGYRESULTAT's Class Diagram



PlantUML diagram generated by SketchIt! (<https://bitbucket.org/pmesmeur/sketch.it>)
For more information about this tool, please contact philippe.mesmeur@gmail.com

STRATEGYRESULTAT's Class Diagram



```
public void setFlyBehavior(FlyBehavior fb) {  
    flyBehavior = fb;  
}  
public void setQuackBehavior(QuackBehavior qb) {  
    quackBehavior = qb;  
}  
abstract void display();  
public void performFly() {  
    flyBehavior.fly();  
}  
public void performQuack() {  
    quackBehavior.quack();  
}  
public void swim() {  
    System.out.println("All ducks float, even decoys!");  
}
```

4. Com funciona el main?

Abans d'aplicar el patró:

```
public static void main(String[] args) {  
  
    MallardDuck mallard = new MallardDuck();  
    RubberDuck rubberDuckie = new RubberDuck();  
    DecoyDuck decoy = new DecoyDuck();  
  
    Duck model = new ModelDuck();  
  
    mallard.display();  
    mallard.fly();  
    mallard.quack();  
  
    rubberDuckie.display();  
    rubberDuckie.fly();  
    rubberDuckie.quack();  
  
    decoy.display();  
    decoy.fly();  
    decoy.quack();  
  
    model.display();  
    model.fly();  
    model.quack();  
}
```

Després d'aplicar el patró:

```
public class MiniDuckSimulator {  
  
    public static void main(String[] args) {  
  
        MallardDuck mallard = new MallardDuck();  
        RubberDuck rubberDuckie = new RubberDuck();  
        DecoyDuck decoy = new DecoyDuck();  
        Duck model = new ModelDuck();  
  
        mallard.performQuack();  
        rubberDuckie.performQuack();  
        decoy.performQuack();  
  
        model.performFly();  
        model.setFlyBehavior(new FlyRocketPowered());  
        model.performFly();  
    }  
}
```

```
public class MallardDuck extends Duck {  
    public MallardDuck() {  
  
        quackBehavior = new Quack();  
        flyBehavior = new FlyWithWings();  
    }  
  
    public void display() { System.out.println("I'm a real Mallard duck"); }  
}
```

Patró Strategy

Nom del patró: **Strategy**

Consideracions:

- S'utilitza si es vol modificar l'estratègia a utilitzar en temps d'execució
- S'utilitza quan es tenen classes molt similars que només varien en la forma de comportar-se

Pros:

- S'aïllen els detalls de la implementació de la solució
- Es poden usar diferents estratègies i canviar-les en temps d'execució
- Open-Closed Principle ✓

Cons:

- Els Clients han de tenir clar en què es diferencien les diferents estratègies
- En el cas de tenir pocs algorismes que rarament canvien, no cal complicar més el codi usant aquest patró

Passos per solucionar els problemes de patrons: Projecte

1. **Problema** identificat a solucionar (principis que es vulneren...)
2. **Identificació del patró** a aplicar amb la seva solució genèrica
3. **Aplicació** del patró al problema
4. Programa principal o client que **usa** el patró
5. **Anàlisi** del patró utilitzat

Passos per solucionar els problemes de patrons

Problema de tarifes en vols:

<https://campusvirtual.ub.edu/mod/resource/view.php?id=3169990>

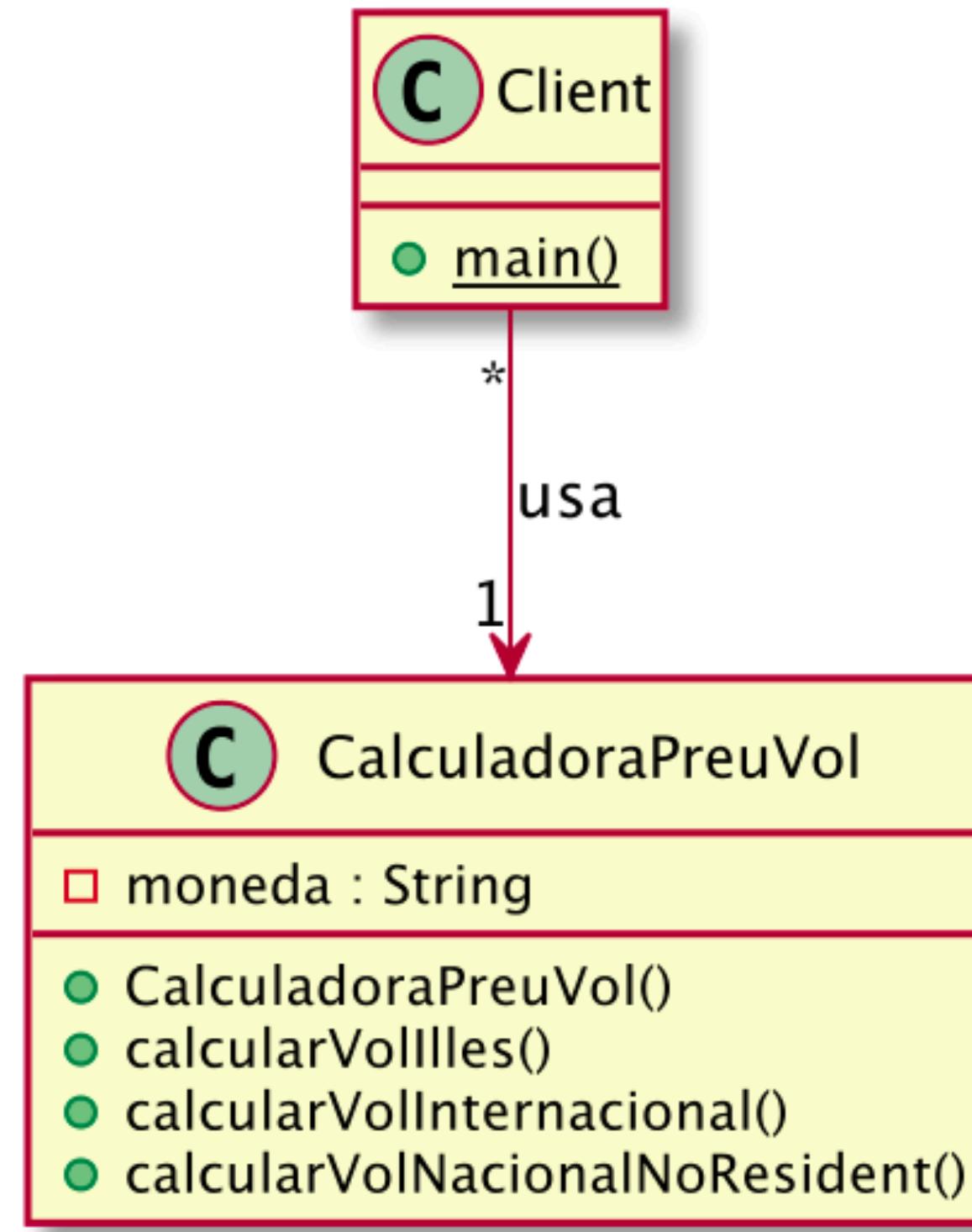
El nostre client és un resident de Mallorca que pertany a una família nombrosa, i cada cop que va comprar un vol té el següent problema:

No sap el preu que ha de pagar perquè depenen del vol se li apliquen descomptes diferents. El preu a pagar és el preu del vol aplicant un dels següents descomptes:

- Si és un vol amb origen i/o destí les Illes Balears, se li aplica el descompte "**Resident illes**" del 75%.
- Si és un vol nacional, però no és del cas anterior, se li aplica el descompte "**Família Nombrosa**" del 5%.
- Si és un vol internacional, **no té cap descompte**.

Per tant el nostre client vol una calculadora per saber sempre el preu que ha de pagar, que serà la nostra classe **CalculadoraPreuVol**.

Identifiqueu quins principis S.O.L.I.D. vulnera. Com ho resoldríeu?



PAS 1. Identificar problema

Problema de tarifes en vols:

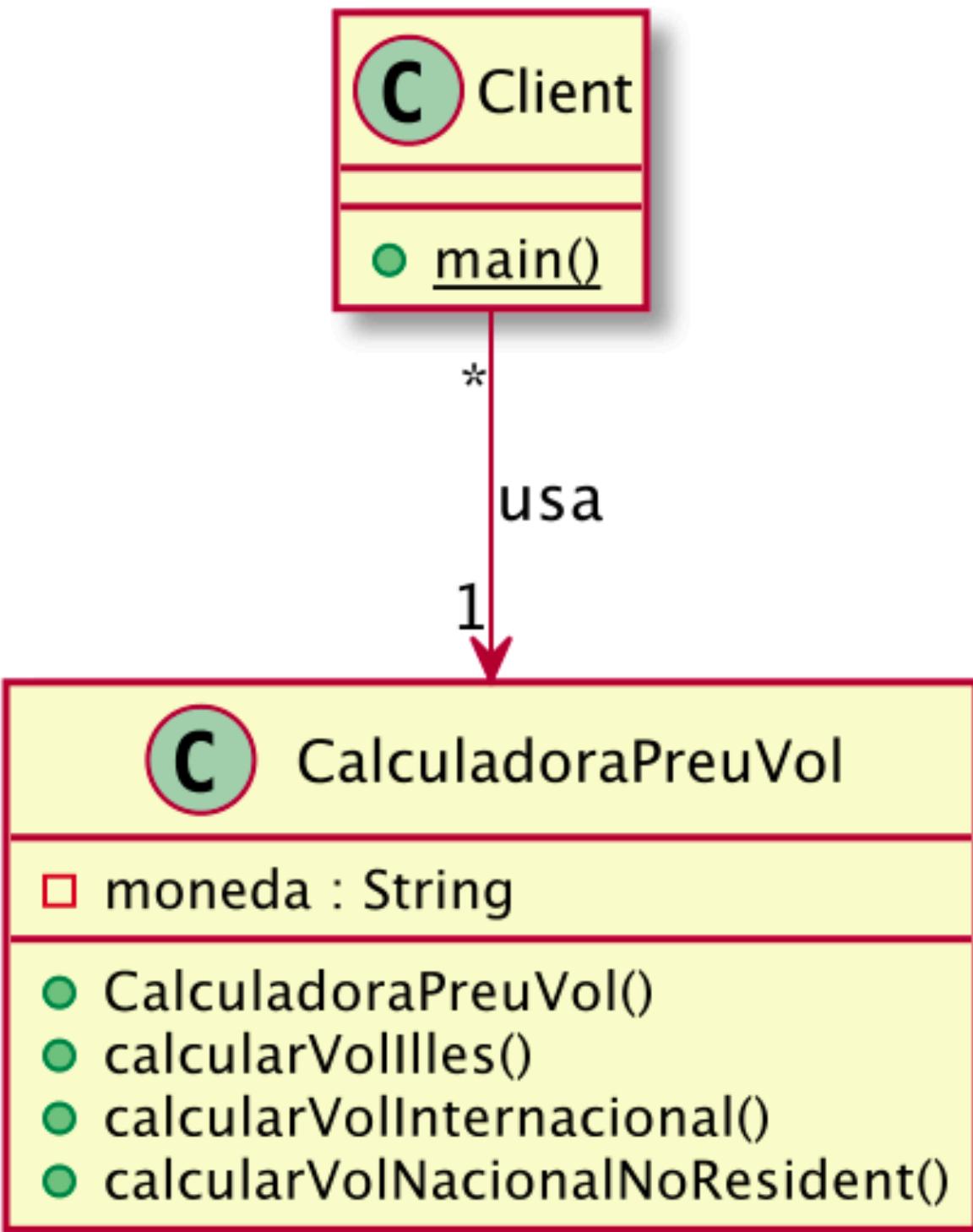
El nostre client és un resident de Mallorca que pertany a una família nombrosa, i cada cop que va comprar un vol té el següent problema:

No sap el preu que ha de pagar perquè depenen del vol se li apliquen descomptes diferents. El preu a pagar és el preu del vol aplicant un dels següents descomptes:

- Si és un vol amb origen i/o destí les Illes Balears, se li aplica el descompte "**Resident illes**" del 75%.
- Si és un vol nacional, però no és del cas anterior, se li aplica el descompte "**Família Nombrosa**" del 5%.
- Si és un vol internacional, **no té cap descompte**.

Per tant el nostre client vol una calculadora per saber sempre el preu que ha de pagar, que serà la nostra classe **CalculadoraPreuVol**.

Identifiqueu quins principis S.O.L.I.D. vulnera. Com ho resoldríeu?



OpenClosed



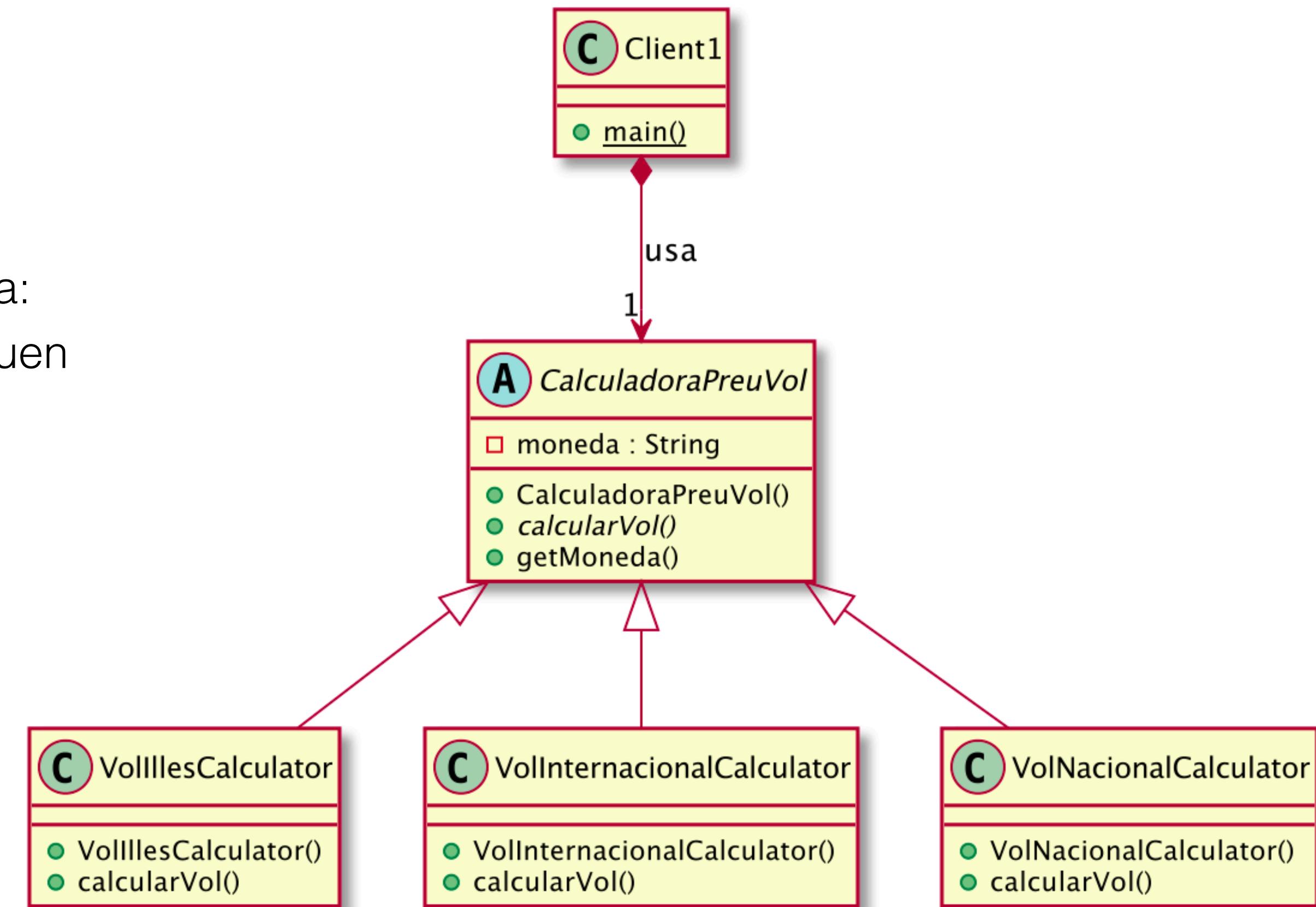
PAS 1. Identificar problema

Problema de tarifes en vols:

El nostre client és un resident de Mallorca que pertany a una família nombrosa, i cada cop que va comprar un vol té el següent problema:

No sap el preu que ha de pagar perquè depenen del vol se li apliquen descomptes diferents. El preu a pagar és el preu del vol aplicant un dels següents descomptes:

- Si és un vol amb origen i/o destí les Illes Balears, se li aplica el descompte "**Resident illes**" del 75%.
- Si és un vol nacional, però no és del cas anterior, se li aplica el descompte "**Família Nombrosa**" del 5%.
- Si és un vol internacional, **no té cap descompte**.



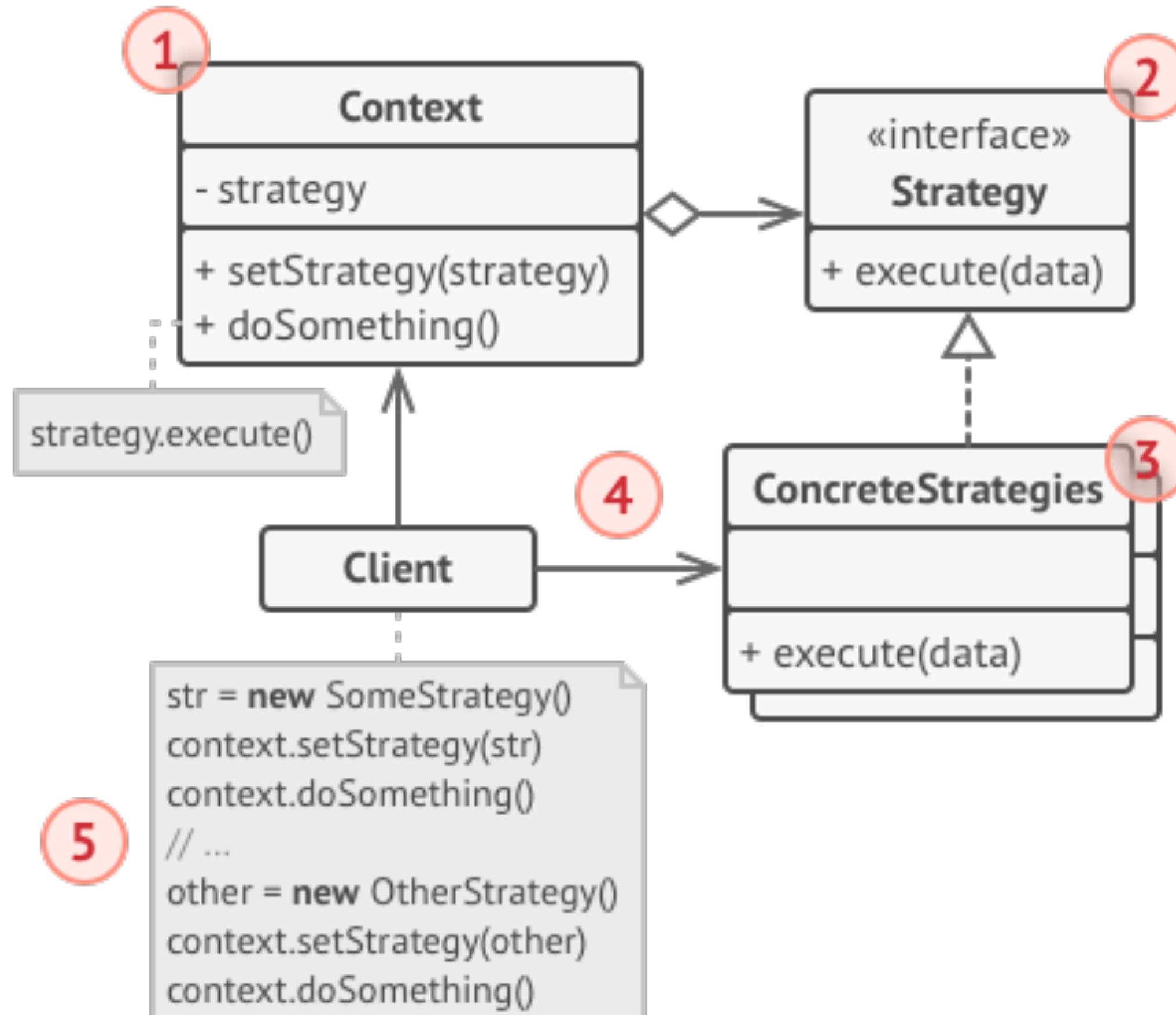
Per tant el nostre client vol una calculadora per saber sempre el preu que ha de pagar, que serà la nostra classe **CalculadoraPreuVol**.

Identifiqueu quins principis S.O.L.I.D. vulnera. Com ho resoldríeu?



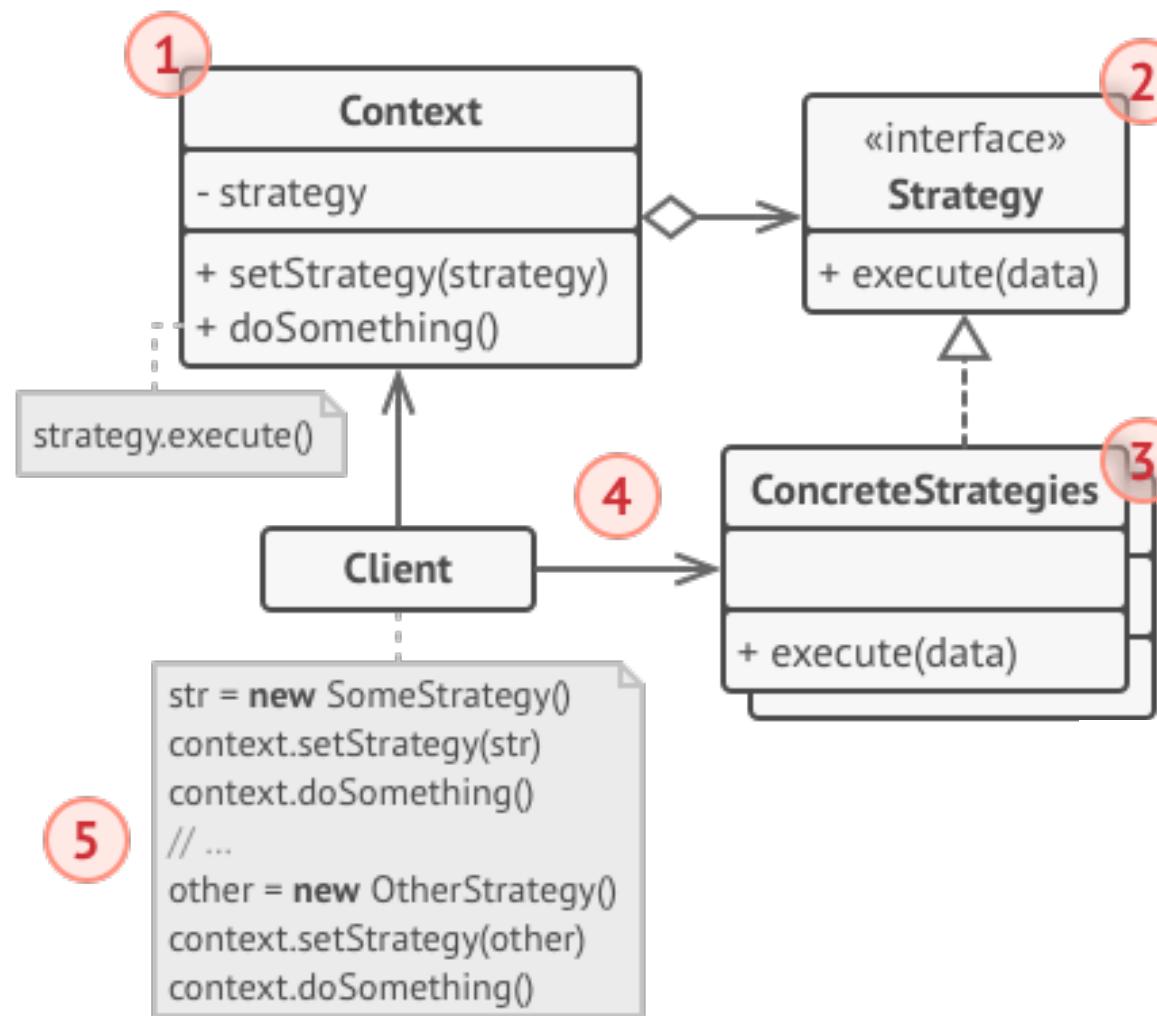
Repetició de codi!!

PAS 2. Patró a aplicar: Strategy



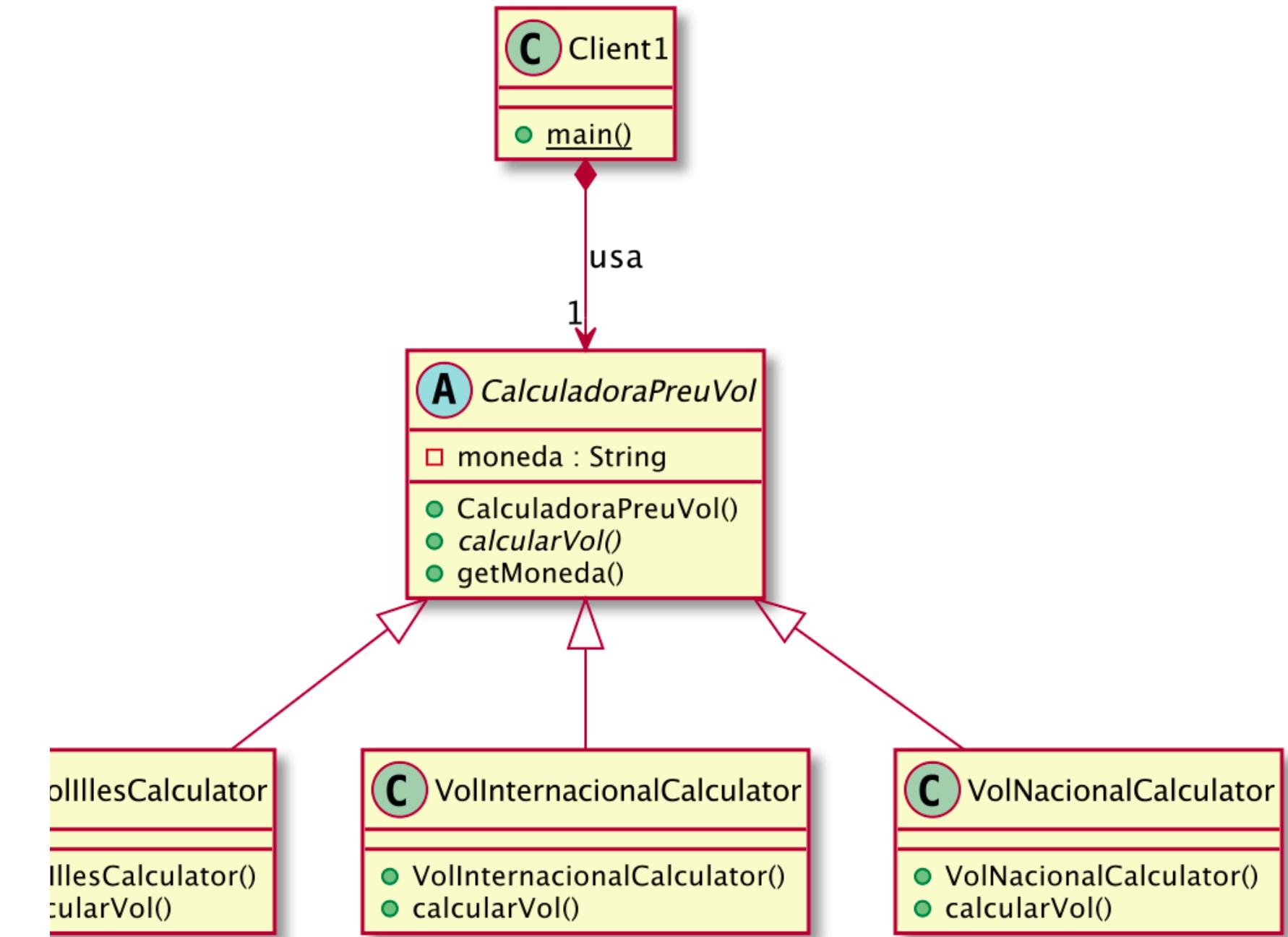
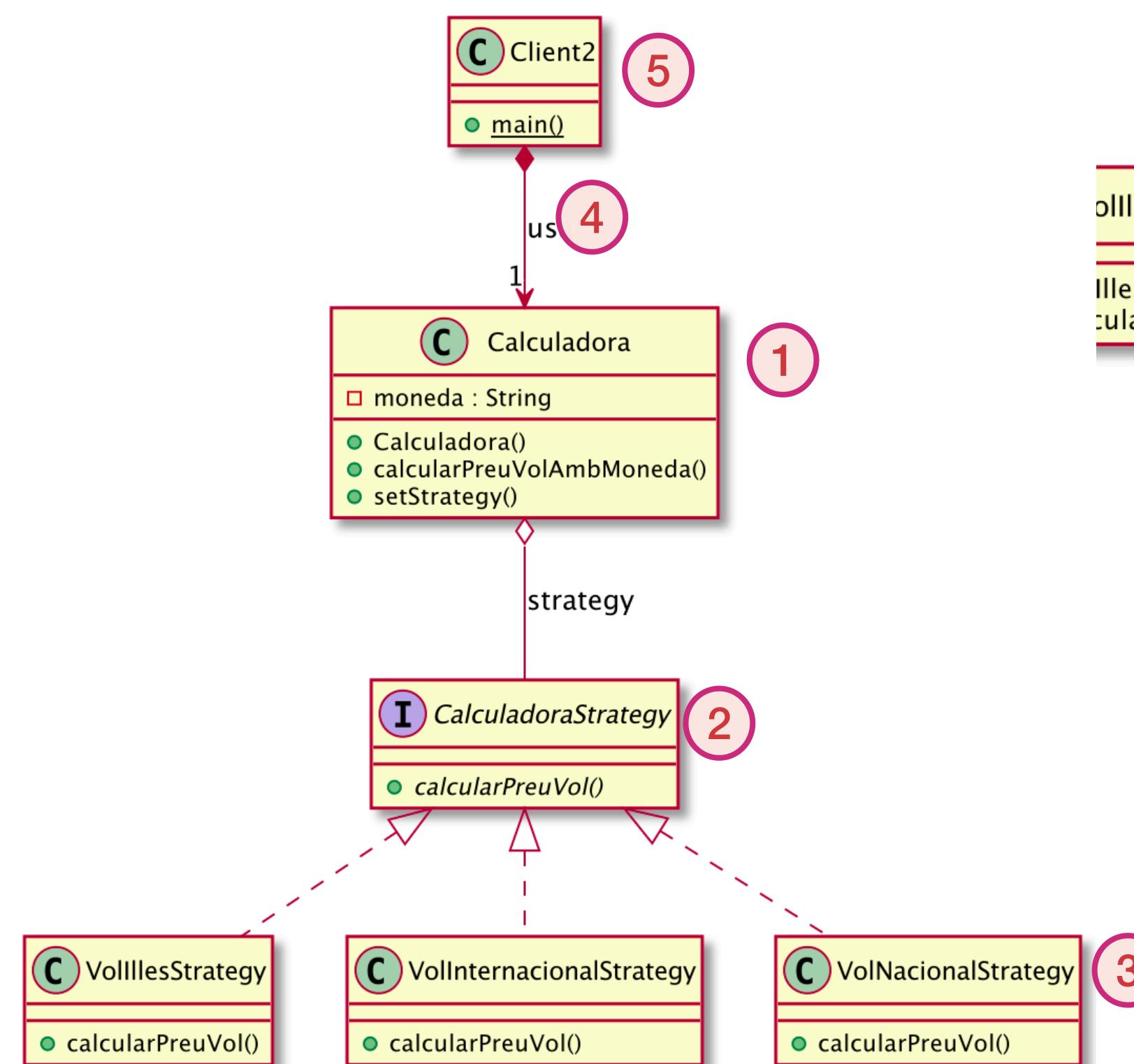
1. El **Context** no coneix res de les estratègies concretes, només les sap executar
2. La classe **Strategy** declara la interície comuna a tots els algorismes
3. Les implementacions concretes de les estratègies estan a les classes **ConcreteStrategies**
4. El **Client** crea l'estratègia concreta que vol fer servir
5. El **Client** passa al **Context** l'estratègia (`setStrategy`) i delega en el **Context** que l'executi.

PAS 3. Aplicar el patró



Contestar les preguntes:

1. Qui és el Context?
2. Quines són les Strategies?
3. Qui és el Client?



PAS 4. Com funciona el main?

Després d'aplicar el patró:

```
package despres2;

public class Client2 {
    public static void main(String[] args){
        System.out.println("OUTPUT DESPRES2:");
        String moneda = "euros";

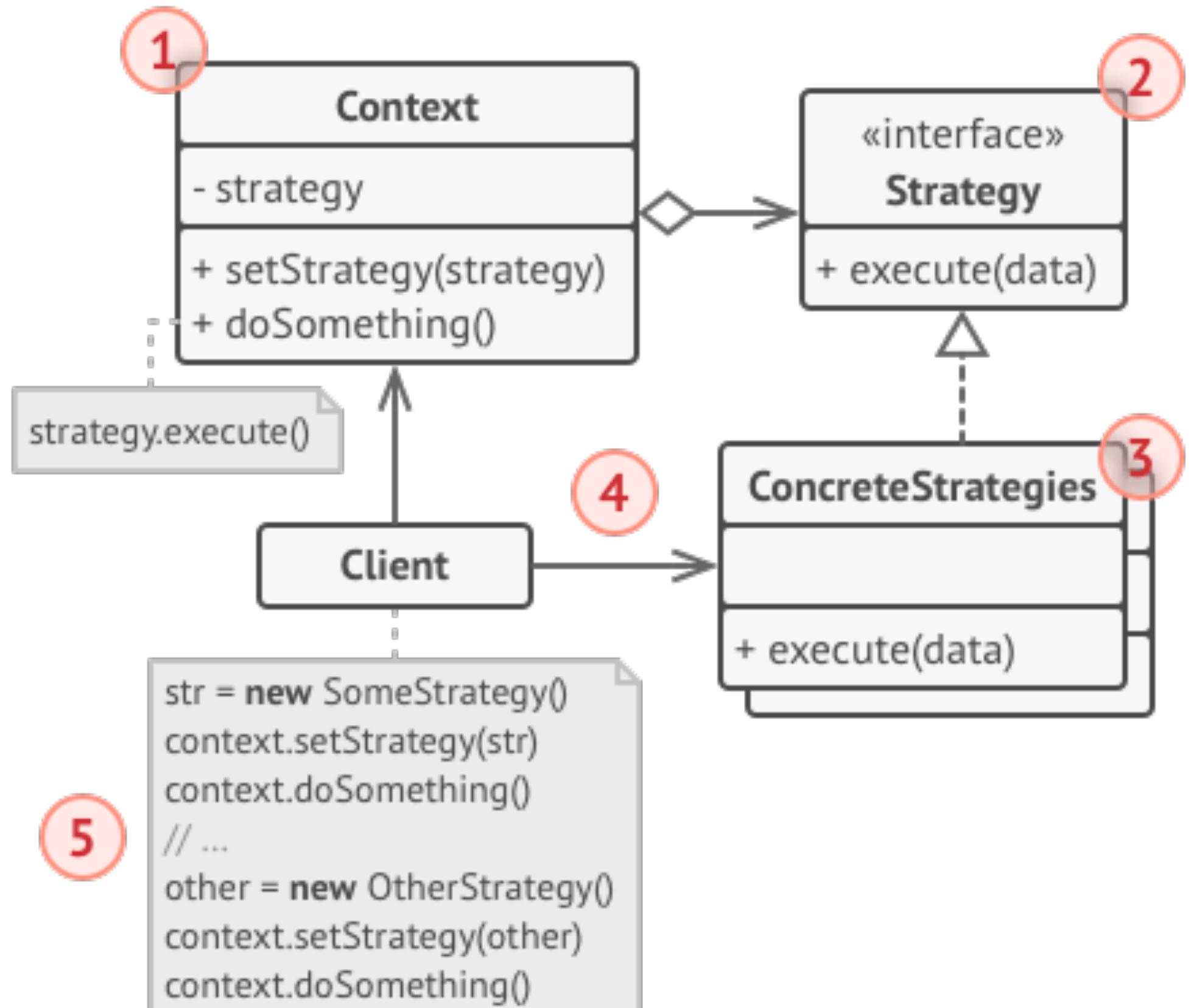
        CalculadoraStrategy str = new VolInternacionalStrategy();
        //obligo a inicialitzar el context amb una Strategy per assegurar-me que sempre podré fer el càcul
        //però puc canviar l'estrategia sempre que vulgui amb el setStrategy
        Calculadora calculadora = new Calculadora(str, moneda);

        System.out.println("Preu final vol 100 € internacional: " +
                           calculadora.calcularPreuVolAmbMoneda( preuVolSenseDescompte: 100));

        CalculadoraStrategy str2 = new VolIllesStrategy();
        calculadora.setStrategy(str2);
        System.out.println("Preu final vol 100 € Mallorca - Barcelona: " +
                           calculadora.calcularPreuVolAmbMoneda( preuVolSenseDescompte: 100));

        CalculadoraStrategy str3 = new VolNacionalStrategy();
        calculadora.setStrategy(str3);
        System.out.println("Preu final vol 100 € Barcelona - Madrid: " +
                           calculadora.calcularPreuVolAmbMoneda( preuVolSenseDescompte: 100));
    }
}
```

PAS 5. Com queden els principis?



S: Single Responsibility

O: Open Closed

L: Liskov

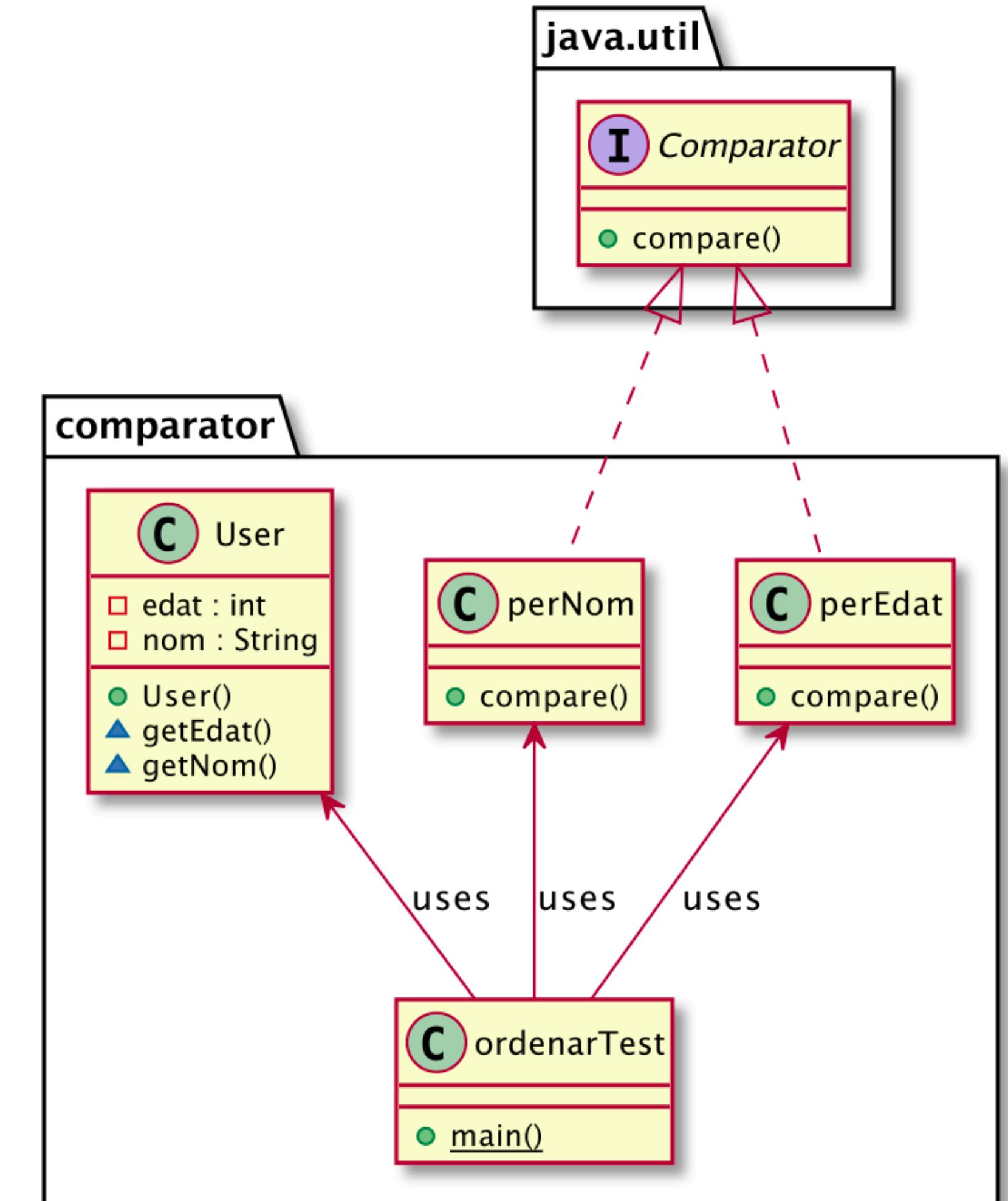
I: Interface Segregation

D: Dependency

Patró Strategy a Comparator

En aquest ús del Comparator de Java, quina classes són el client, el context i les estratègies concretes que corresponen al patró Strategy?

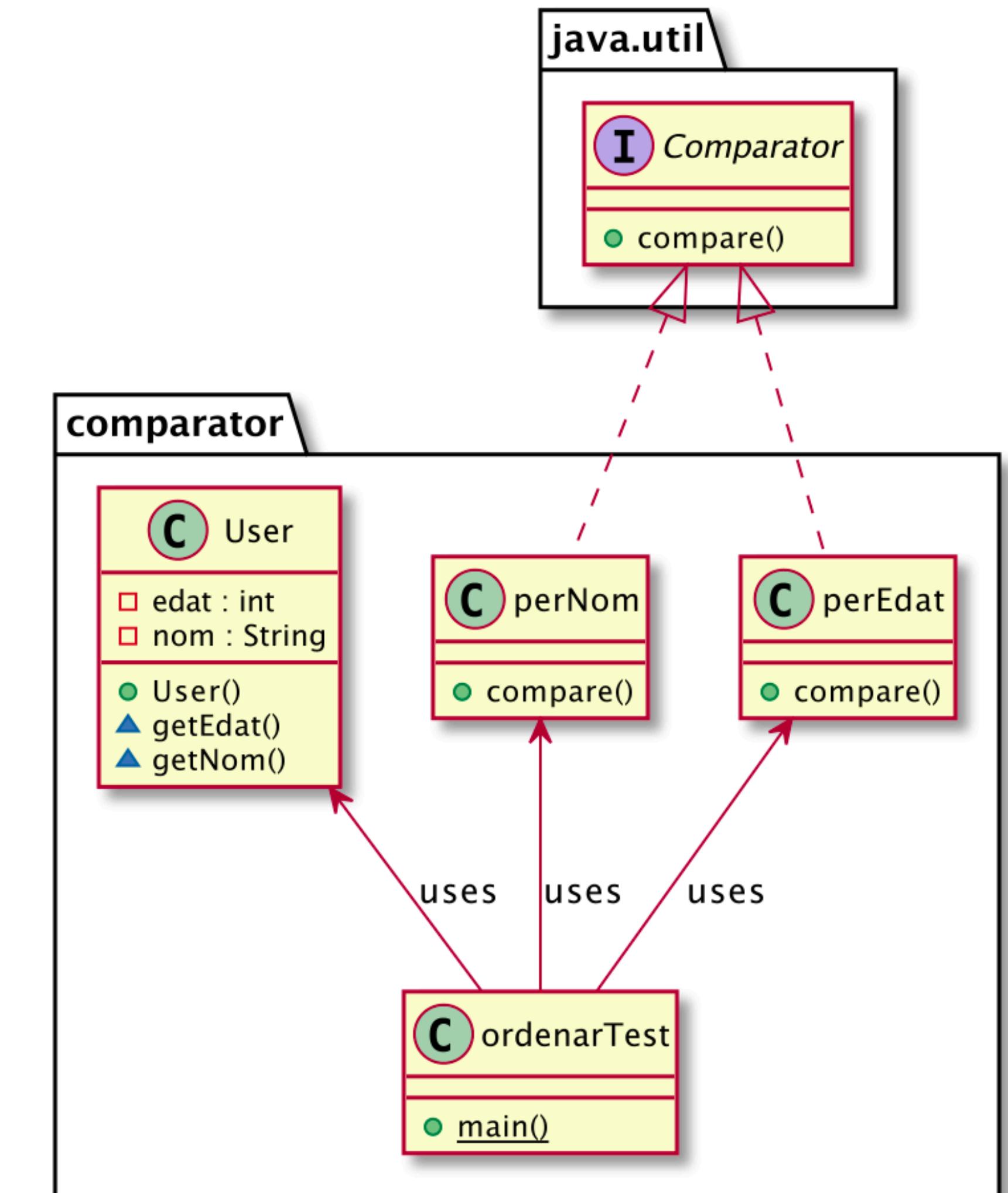
- a. No hi ha interfície Strategy, ja que les estratègies concretes ja s'injecten directament a la crida de fer el sort de la classe ordenarTest.
- b. El Context és la classe ordenarTest, el Client és la classe User, la interfície de strategy es la classe Comparator i les estratègies concretes són les classes perEdat i perNom.
- c. Aquest no és un exemple del patró Strategy.
- d. El Client i el Context són la mateixa classe ordenarTest, la interfície de strategy es la classe Comparator i les estratègies concretes són les classes perEdat i perNom.



Patró Strategy a Comparator

En aquest ús del Comparator de Java, quina classes són el client, el context i les estratègies concretes que corresponen al patró Strategy?

- a. No hi ha interfície Strategy, ja que les estratègies concretes ja s'injecten directament a la crida de fer el sort de la classe ordenarTest.
- b. El Context és la classe ordenarTest, el Client és la classe User, la interfície de strategy es la classe Comparator i les estratègies concretes són les classes perEdat i perNom.
- c. Aquest no és un exemple del patró Strategy.
- d. El Client i el Context són la mateixa classe ordenarTest, la interfície de strategy es la classe Comparator i les estratègies concretes són les classes perEdat i perNom.



Patró Strategy a Comparator

```
import java.util.ArrayList;
import java.util.List;

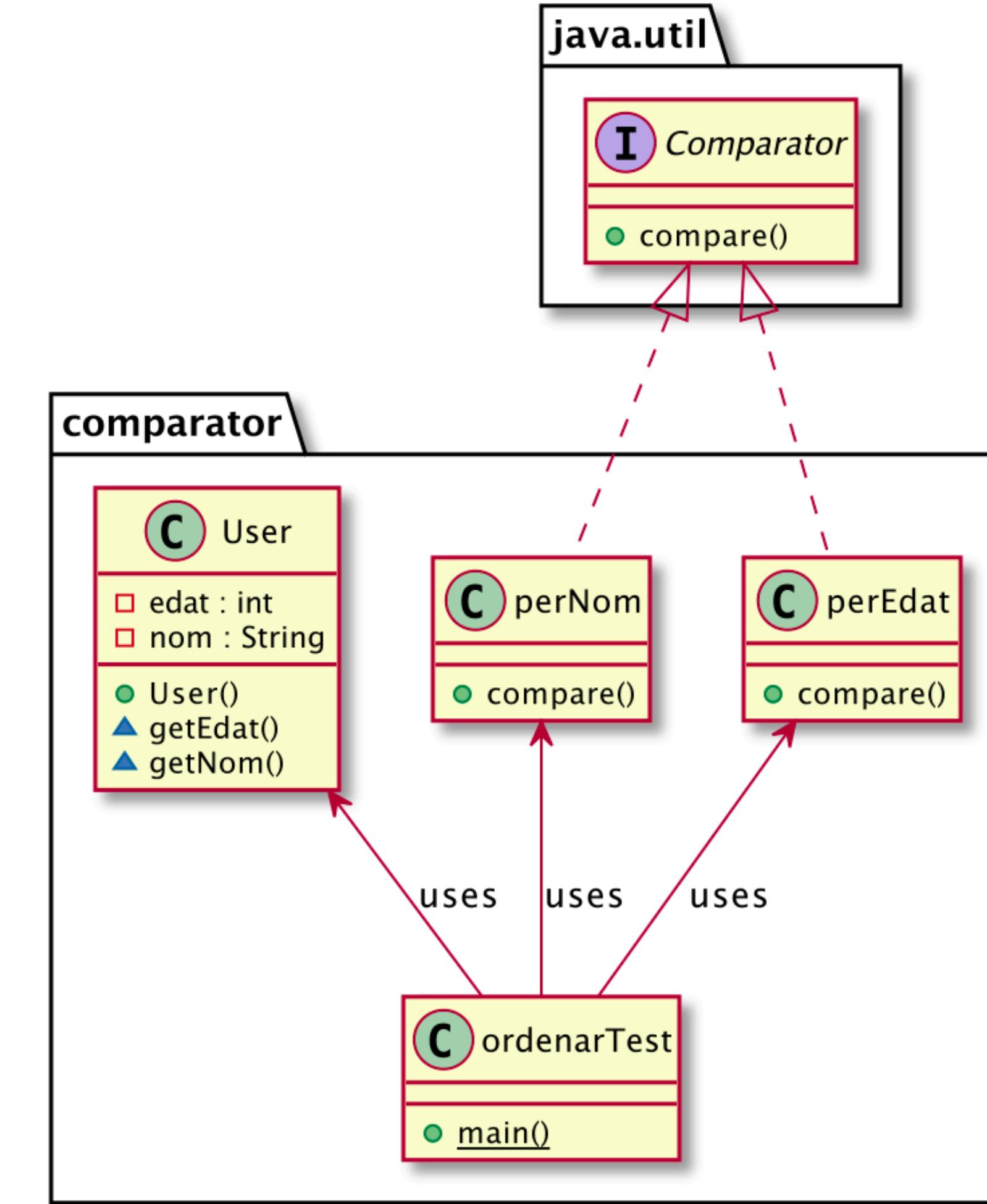
public class ordenarTest {

    public static void main(String[] args) {
        List<User> list = new ArrayList<>();
        list.add(new User( nom: "Maria", edat: 21));
        list.add(new User( nom: "David", edat: 23));
        list.add(new User( nom: "Raul", edat: 19));
        list.add(new User( nom: "Marta", edat: 20));
        list.add(new User( nom: "Cristina", edat: 21));

        list.sort(new perNom());

        System.out.println("Llista ordenada per noms:");
        for(User a: list) // printing the sorted list of names by names
            System.out.print(a.getNom() + ", ");

        System.out.println();
        System.out.println("Llista ordenada per edat:");
        list.sort(new perEdat());
        for(User a: list) // printing the sorted list of names by age
            System.out.print(a.getNom() + ": " + a.getEdat() + ", ");
    }
}
```

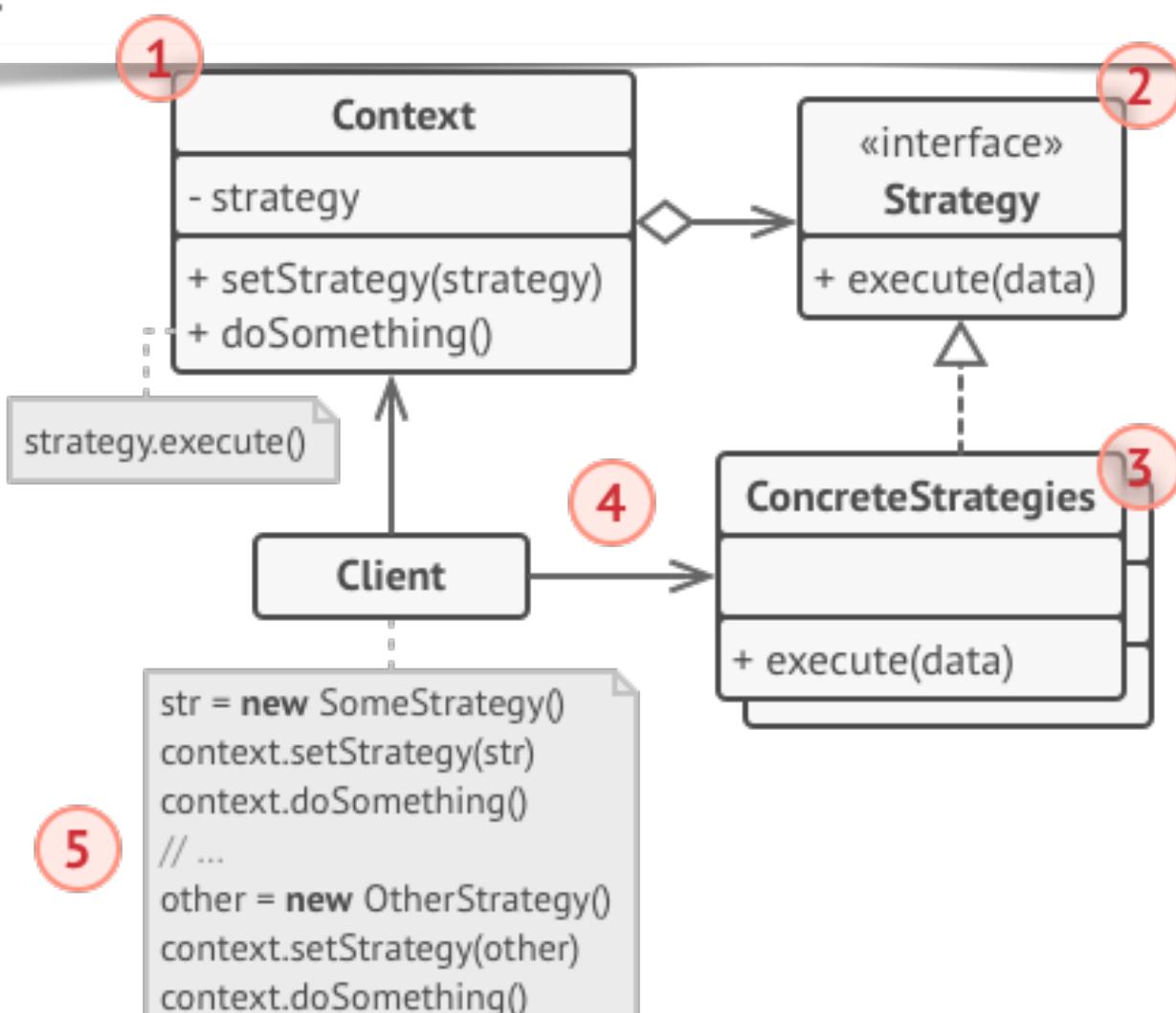


Patrón Strategy a Comparator

```
public class User {
    private String nom;
    private int edat;

    public User (String nom, int edat) {
        this.nom = nom;
        this.edat = edat;
    }

    String getNom (){ return nom;}
    int getEdat() { return edat;}
}
```



```
import java.util.Comparator;

public class perEdat implements Comparator<User> {

    @Override
    public int compare(User o1, User o2) {
        return o1.getEdat()-o2.getEdat();
    }
}
```

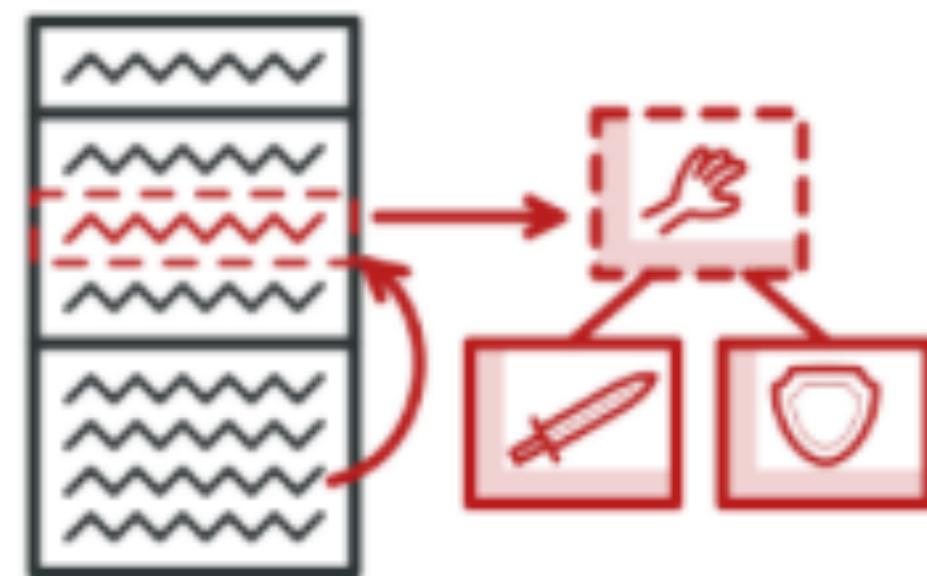
```
import java.util.Comparator;

public class perNom implements Comparator<User> {

    @Override
    public int compare(User o1, User o2) {
        return o1.getNom().compareToIgnoreCase(o2.getNom());
    }
}
```

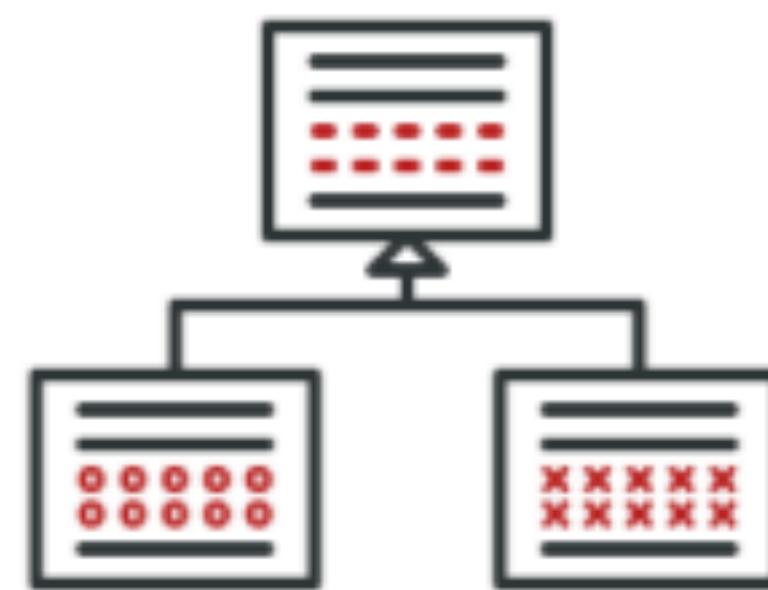
3.5. Patrons de disseny: Patró Strategy

- Patrons relacionats



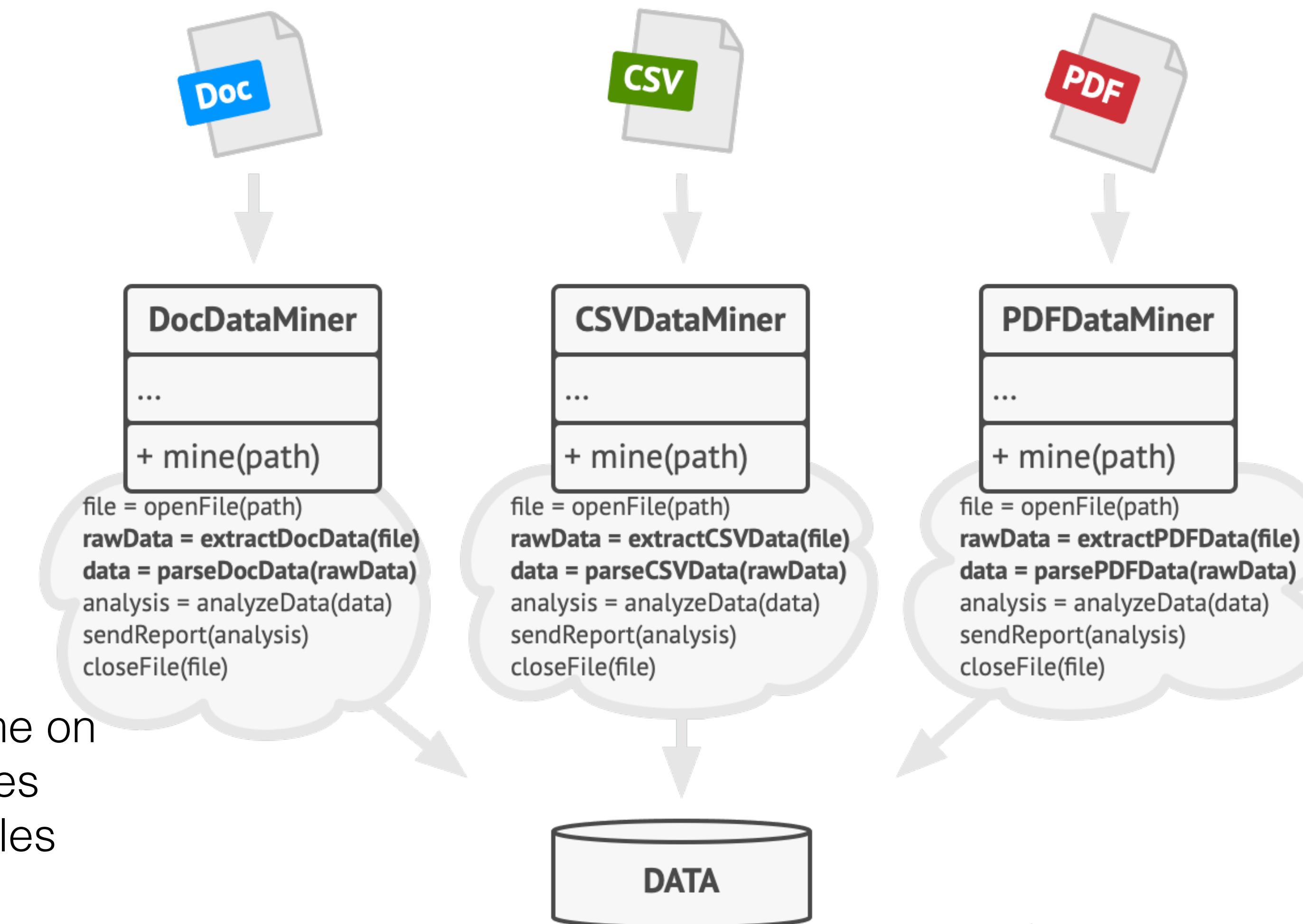
Strategy

Diferents mètodes posats
cadascun d'ells en una
classe diferent



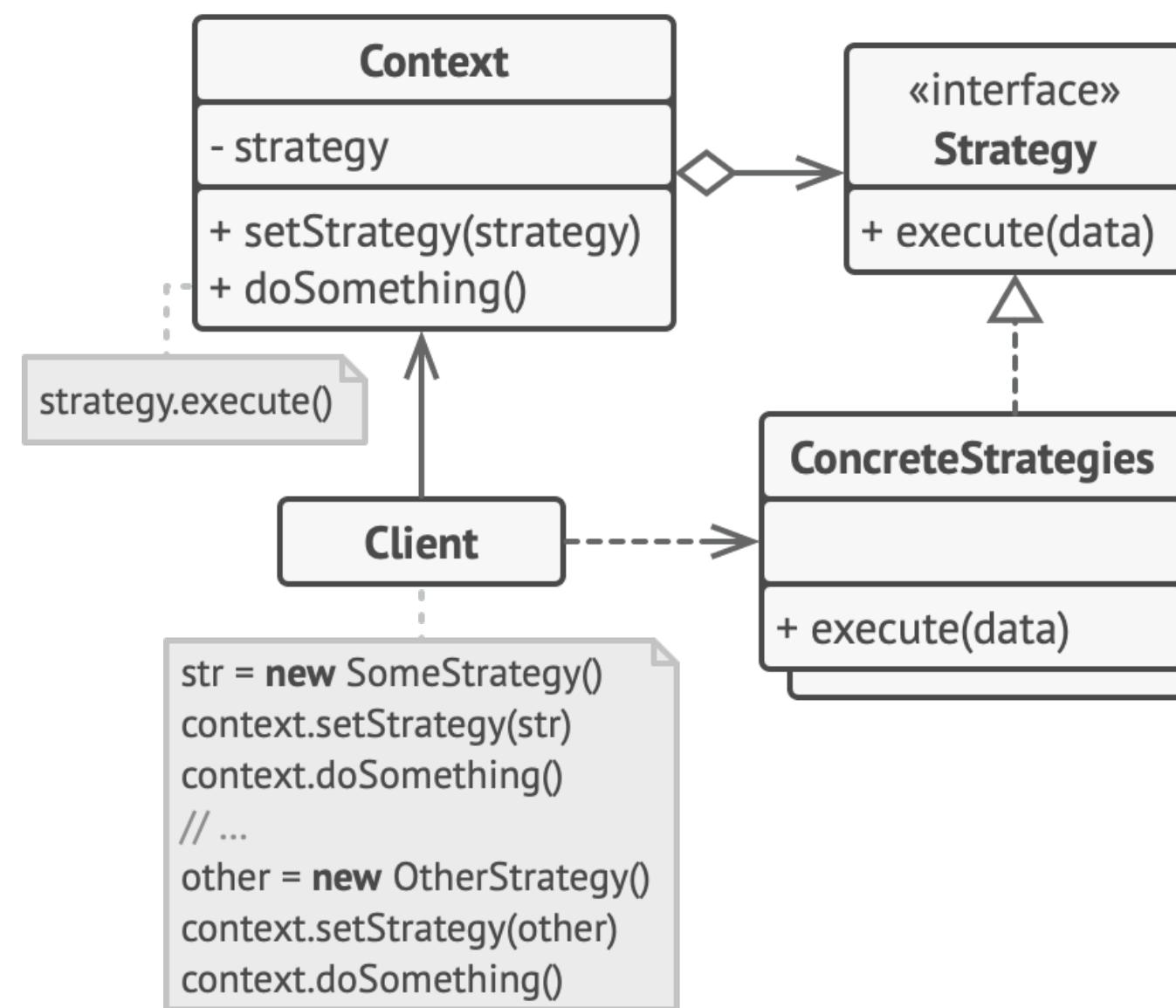
Template Method

Esquelet d'un algorisme on
hi ha alguna part que es
canvia a les classes filles



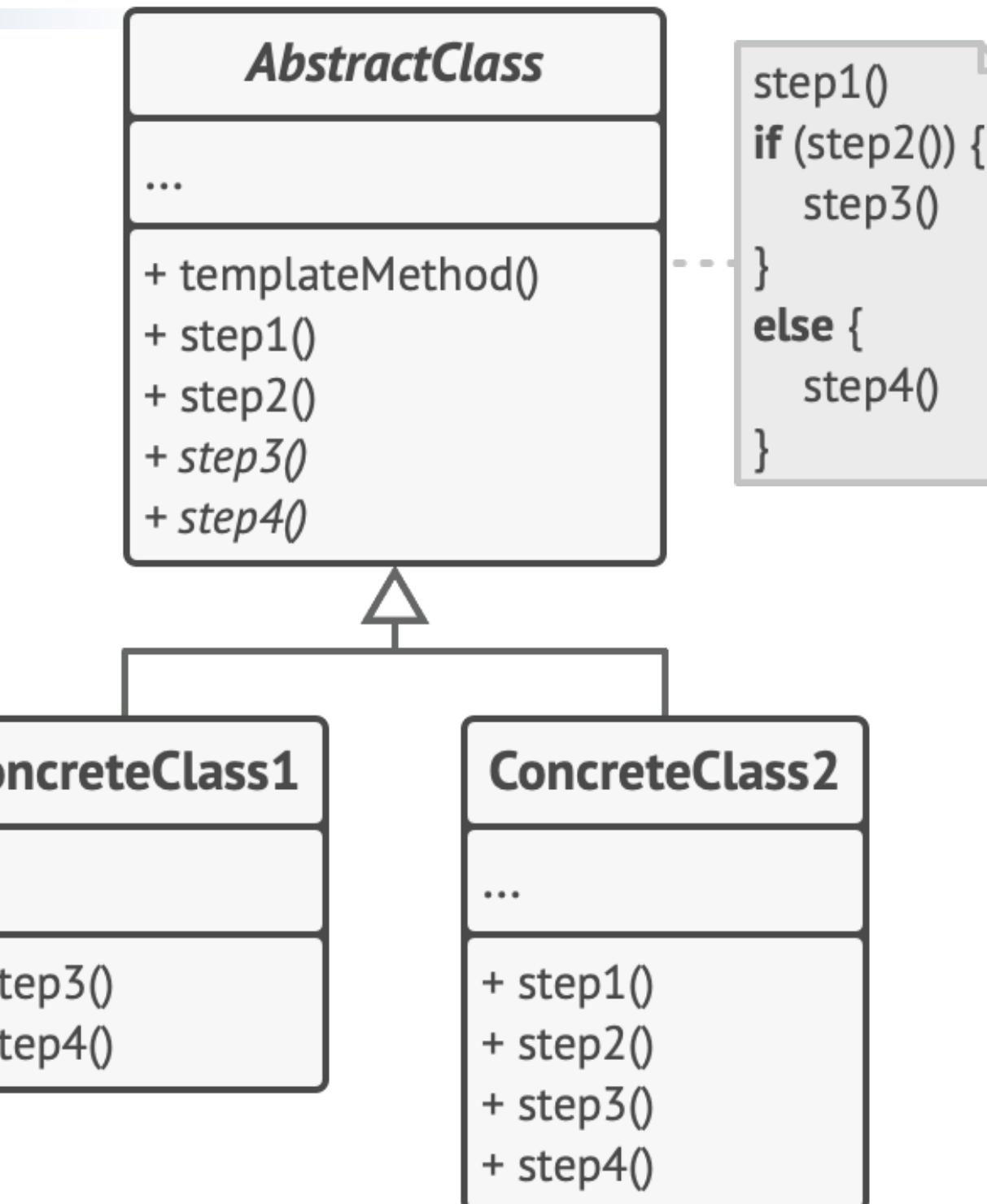
3.5. Patrons de disseny: Patró Strategy

- Patrons relacionats



Strategy

Diferents mètodes posats
cadascun d'ells en una
classe diferent



Template Method

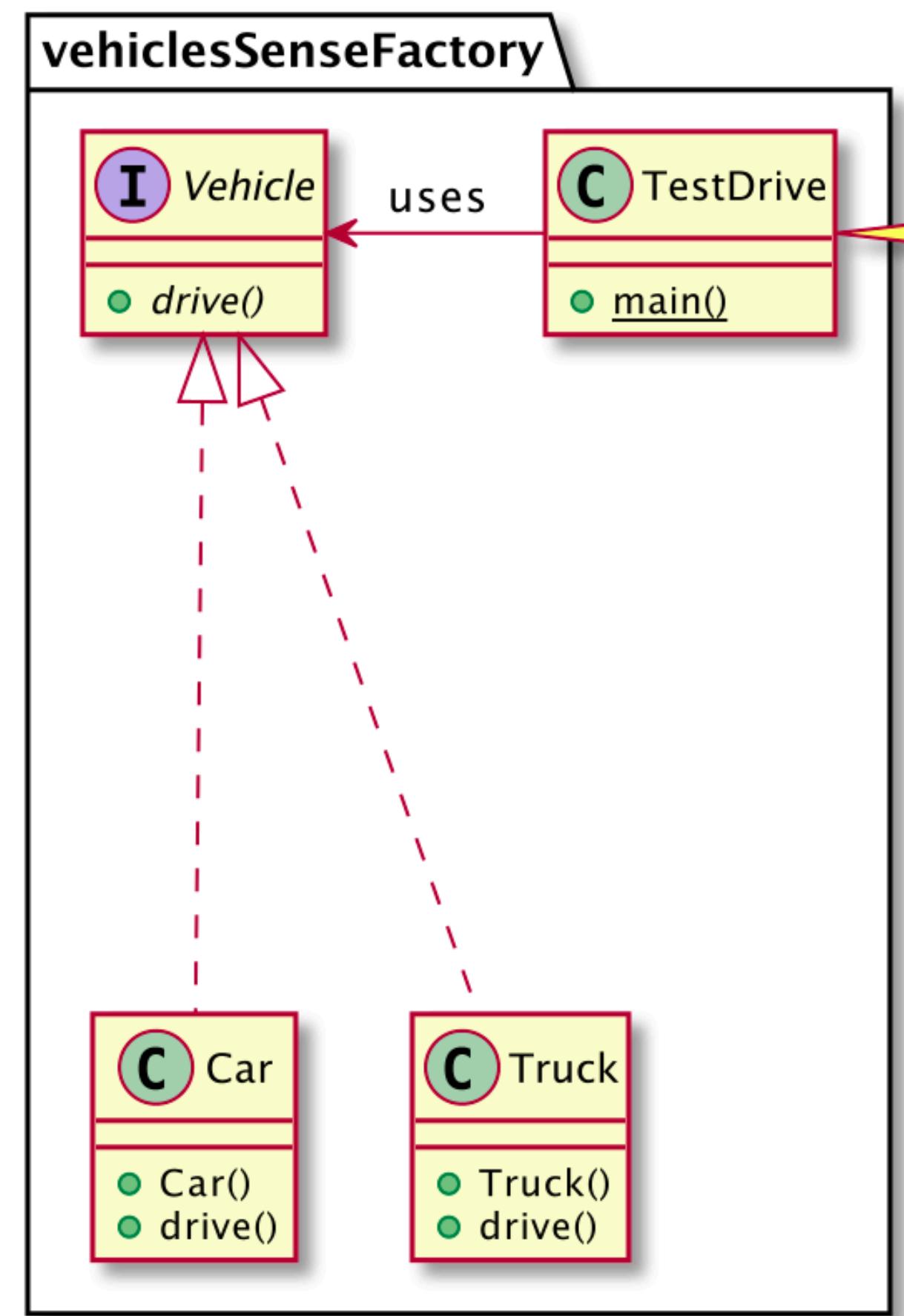
Esquelet d'un algorisme on
hi ha alguna part que es
canvia a les classes filles

3.4. Patrons de disseny

Propòsit → Àmbit ↓	CREACIÓ	ESTRUCTURA	COMPORTAMENT
CLASSE	<ul style="list-style-type: none">• Factory method	<ul style="list-style-type: none">• class Adapter	<ul style="list-style-type: none">• Interpreter• Template method
OBJECTE	<ul style="list-style-type: none">• Abstract Factory• Builder• Prototype• Singleton• Object pool	<ul style="list-style-type: none">• Object Adapter• Bridge• Composite• Decorator• Facade• Flyweight• Proxy	<ul style="list-style-type: none">• Chain of Responsibility• Command• Iterator• Mediator• Memento• Observer• State• Strategy• Visitor

Patró Simple Factory

El problema



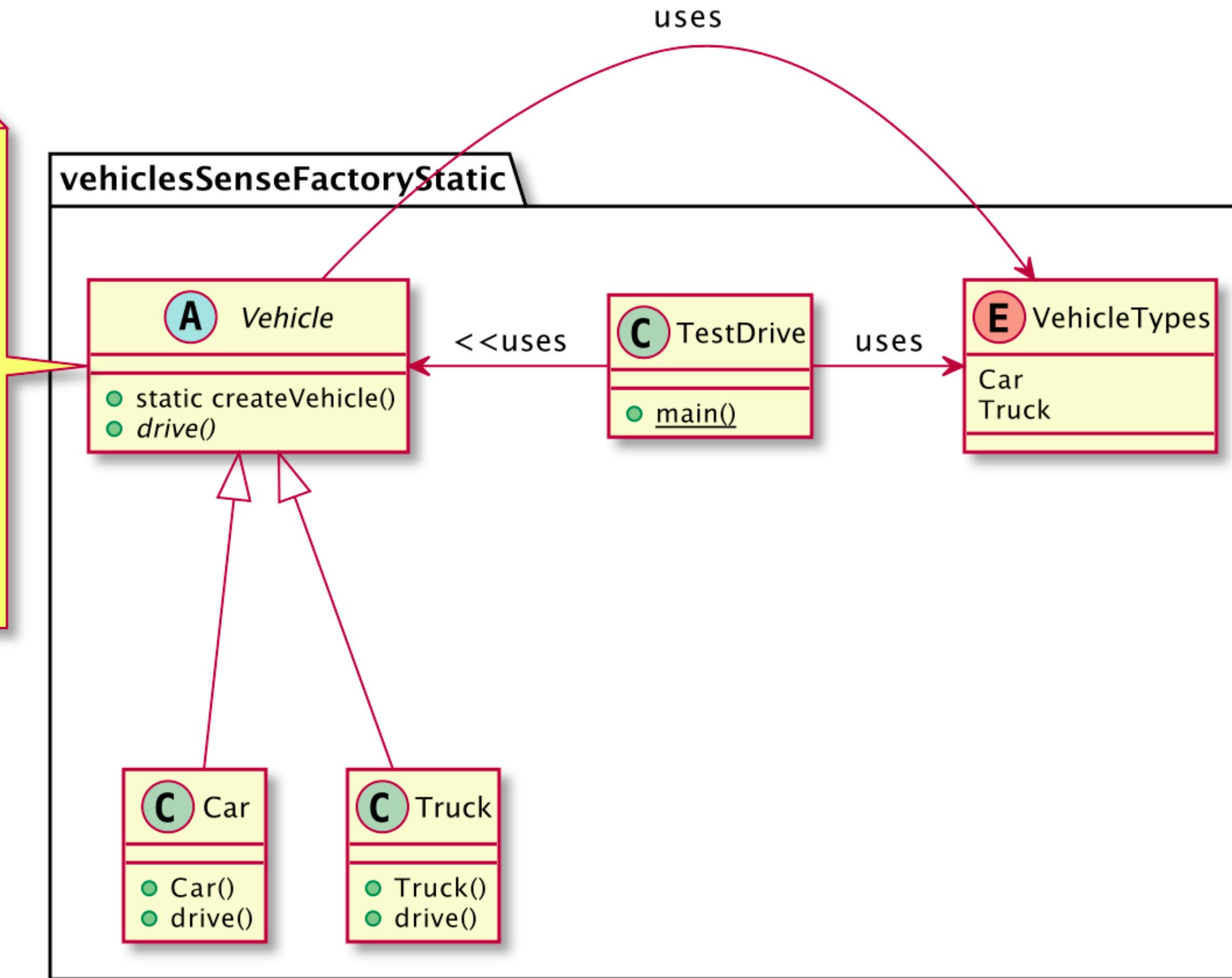
```
public class TestDrive {  
    public static void main(String[] args) {  
  
        System.out.println("Quin tipus de vehicle vols?");  
  
        String name;  
        Scanner teclado = new Scanner(System.in);  
        name = teclado.nextLine();  
        Vehicle vehicle = null;  
  
        switch (name) {  
            case "CAR":  
                vehicle = new Car();  
                break;  
            case "TRUCK":  
                vehicle = new Truck();  
                break;  
        }  
        vehicle.drive();  
    }  
}
```

Open-Closed

Patrón Factory

Una primera solución?

```
public static Vehicle createVehicle (VehicleTypes name) {  
    Vehicle vehicle = null;  
    switch (name) {  
        case Car:  
            vehicle = new Car();  
            break;  
        case Truck:  
            vehicle = new Truck();  
            break;  
    }  
    return vehicle;  
  
}  
public abstract void drive();
```



- Vulnera el principio **?** de S.O.L.I.D.

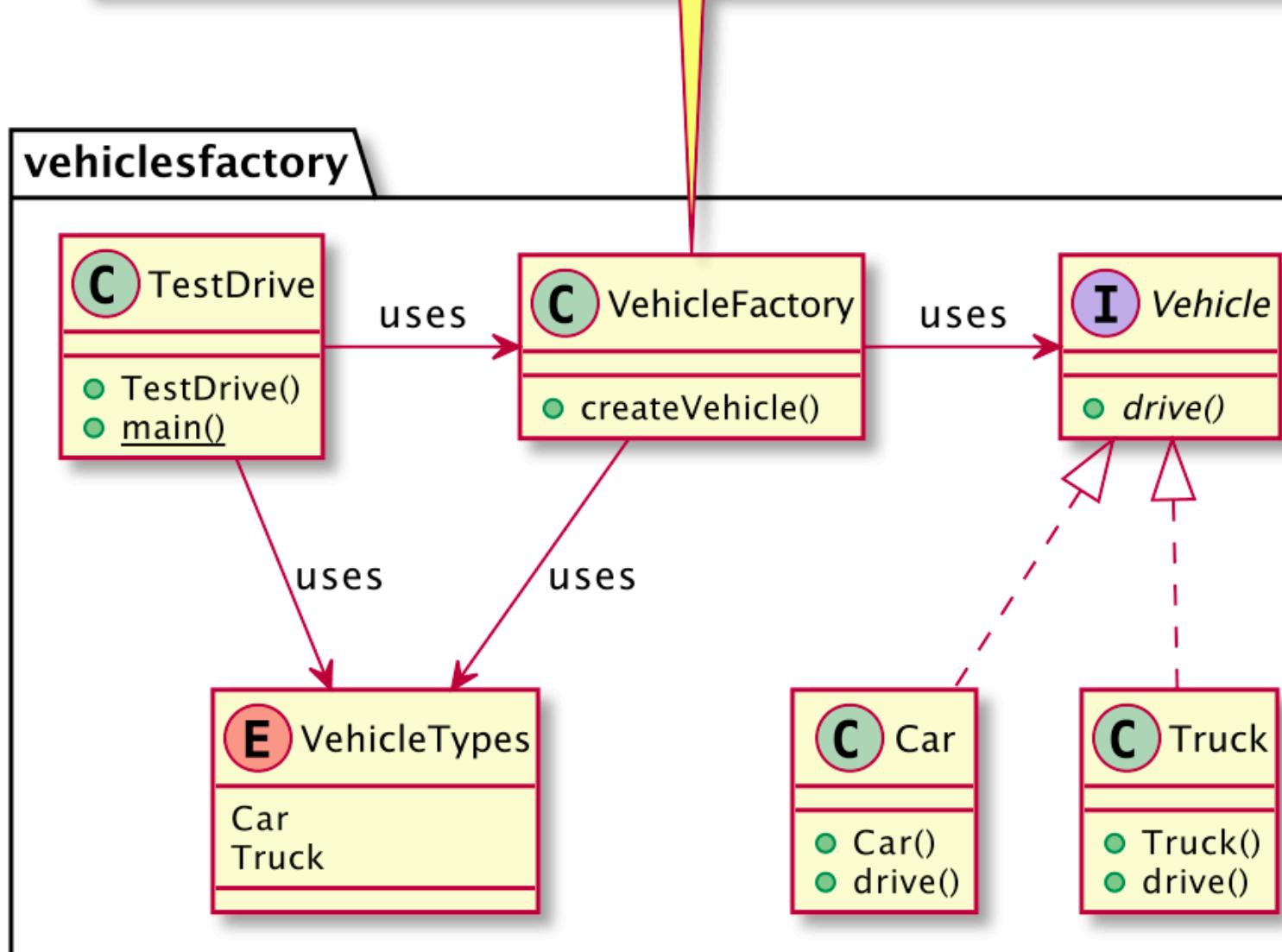
Patrons Factory

- **Simple Factory** – Defineix una classe per crear objectes i consulta el nou objecte creat a través d'una interfície comú dels objectes creats

Patró Factory Method

Primera aproximació (versió simplificada del Factory Method)

```
public class VehicleFactory {  
    /**  
     * Method to create vehicle types  
     * @param vehicleType  
     * @return Vehicle  
     */  
    public Vehicle createVehicle(VehicleTypes vehicleType) {  
        Vehicle vehicle = null;  
        switch (vehicleType) {  
            case Car:  
                vehicle = new Car();  
                break;  
            case Truck:  
                vehicle = new Truck();  
                break;  
        }  
        return vehicle;  
    }  
}
```



SOLUCIÓ:

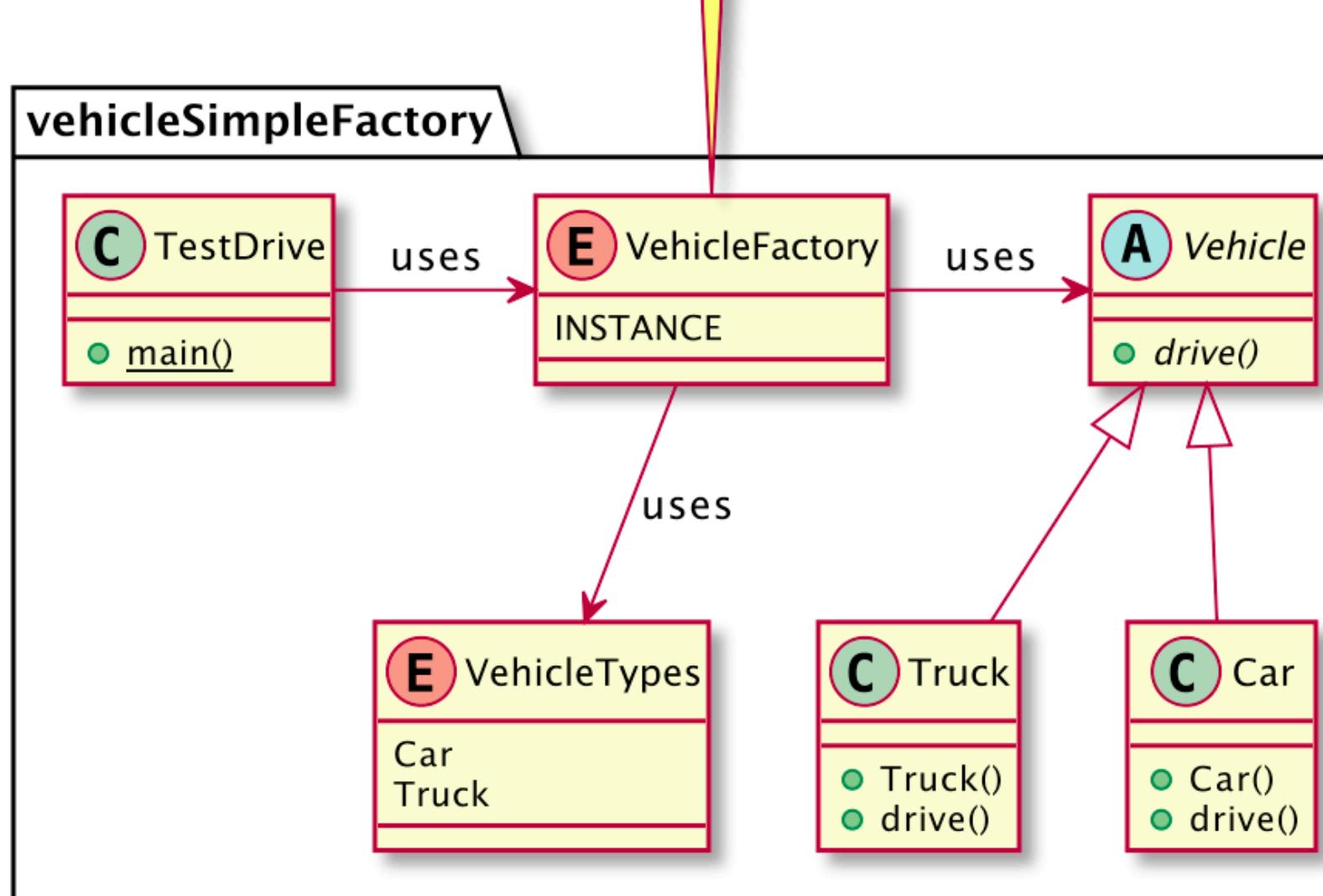
- Es separa el creador de les instàncies de la pròpia classe
- Les instàncies es creen en una classe Factoria, en aquest cas **VehicleFactory**

Patró Factory Method

Primera aproximació (versió simplificada del Factory Method)

```
public enum VehicleFactory {  
    INSTANCE;  
  
    public Vehicle createVehicle (VehicleTypes name) {  
        Vehicle vehicle = null;  
        switch (name) {  
            case Car:  
                vehicle = new Car();  
                break;  
            case Truck:  
                vehicle = new Truck();  
                break;  
        }  
        return vehicle;  
    }  
}
```

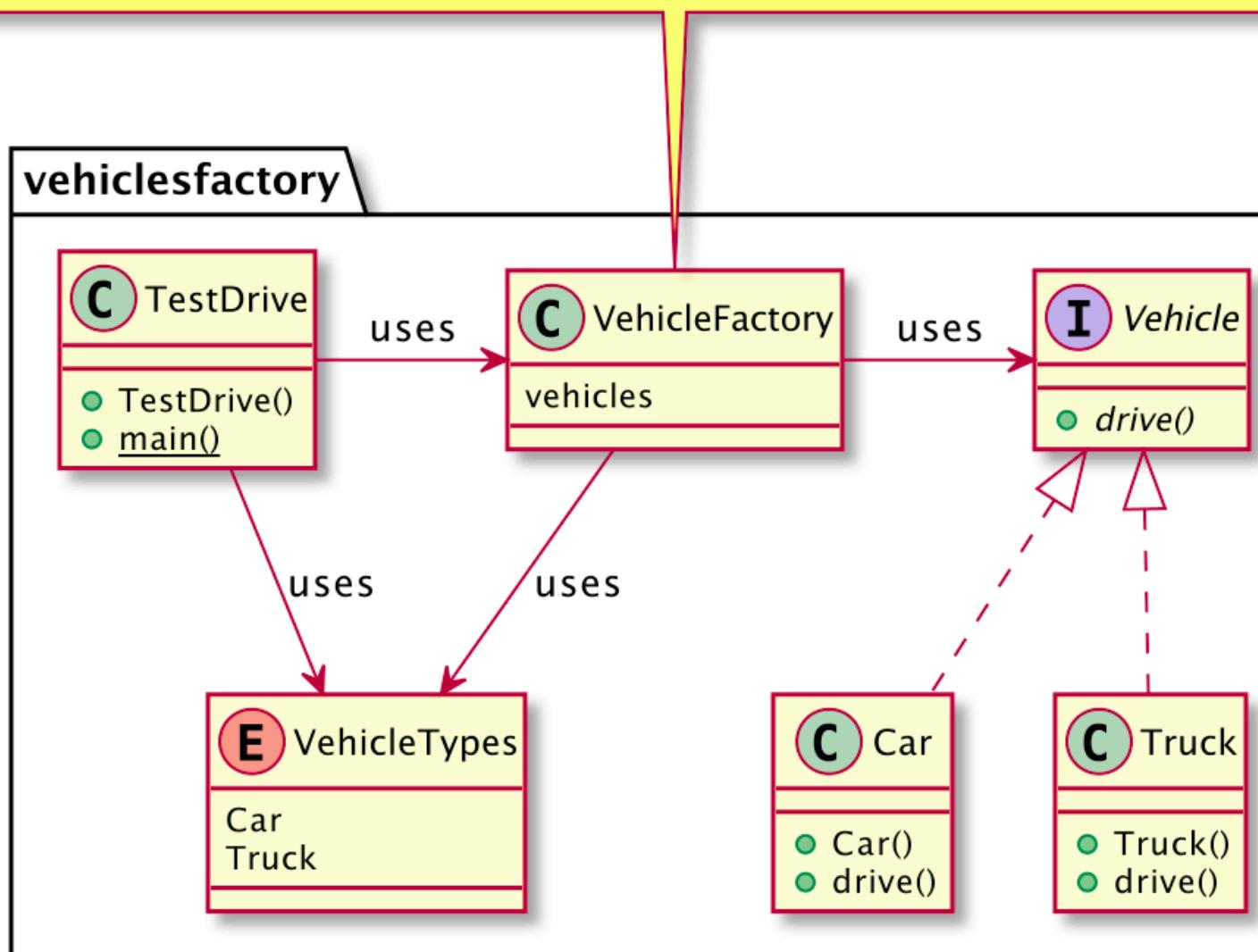
- Pot ser un Singleton?
- com solucionar el Open-Closed?



Patró Factory Method

Primera aproximació (versió simplificada del Factory Method)

```
private Map<String, Vehicle> vehicles = new HashMap<String, Vehicle>();  
/**  
 * Method to create vehicle types  
 * @param vehicleType  
 * @return Vehicle  
 * @throws Exception  
 */  
public Vehicle createVehicle(String vehicleType)  
    throws Exception {  
    Vehicle vehicle = vehicles.get(vehicleType);  
    if (vehicle != null) {  
        return vehicle;  
    } else {  
        try {  
            String name = Vehicle.class.getPackage().getName();  
            vehicle = (Vehicle) Class.forName(name + "." + vehicleType).newInstance();  
            vehicles.put(vehicleType, vehicle);  
            return vehicle;  
        } catch (Exception e) {  
            throw new Exception("The vehicle type is unknown!");  
        }  
    }  
}
```



- Com solucionar el Open Closed?

ús de reflexivitat

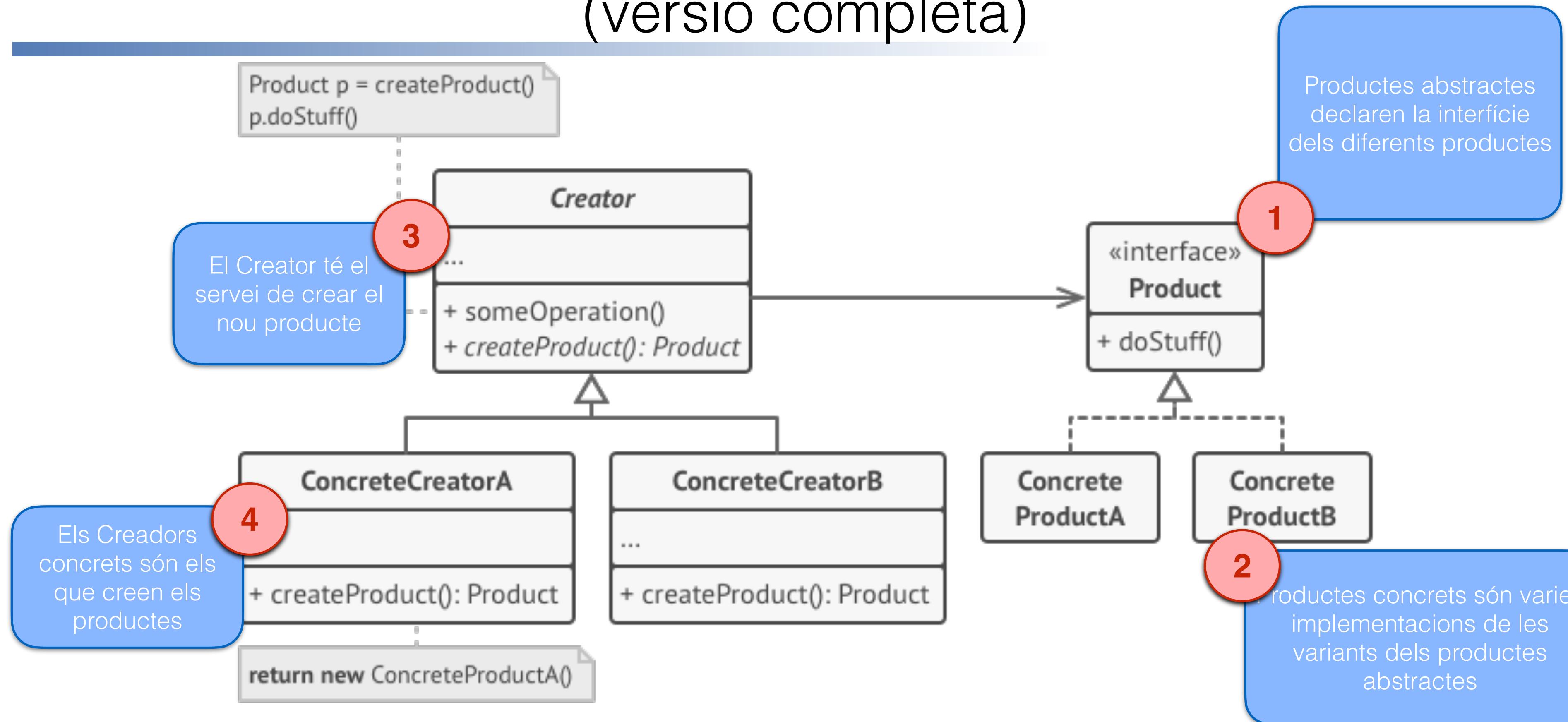
- com són les instàncies dels vehicles concrets?

I si vull tenir diferents criteris per a construir les classes concretes?

Patrons Factory

- **Factory Method** – Defineix una classe abstracte per crear objectes, però deixa a les subclasses decidir quina classe ha d'instanciar i consulta el nou objecte creat a través d'una interfície comú dels objectes creats

Patró Factory Method (versió completa)



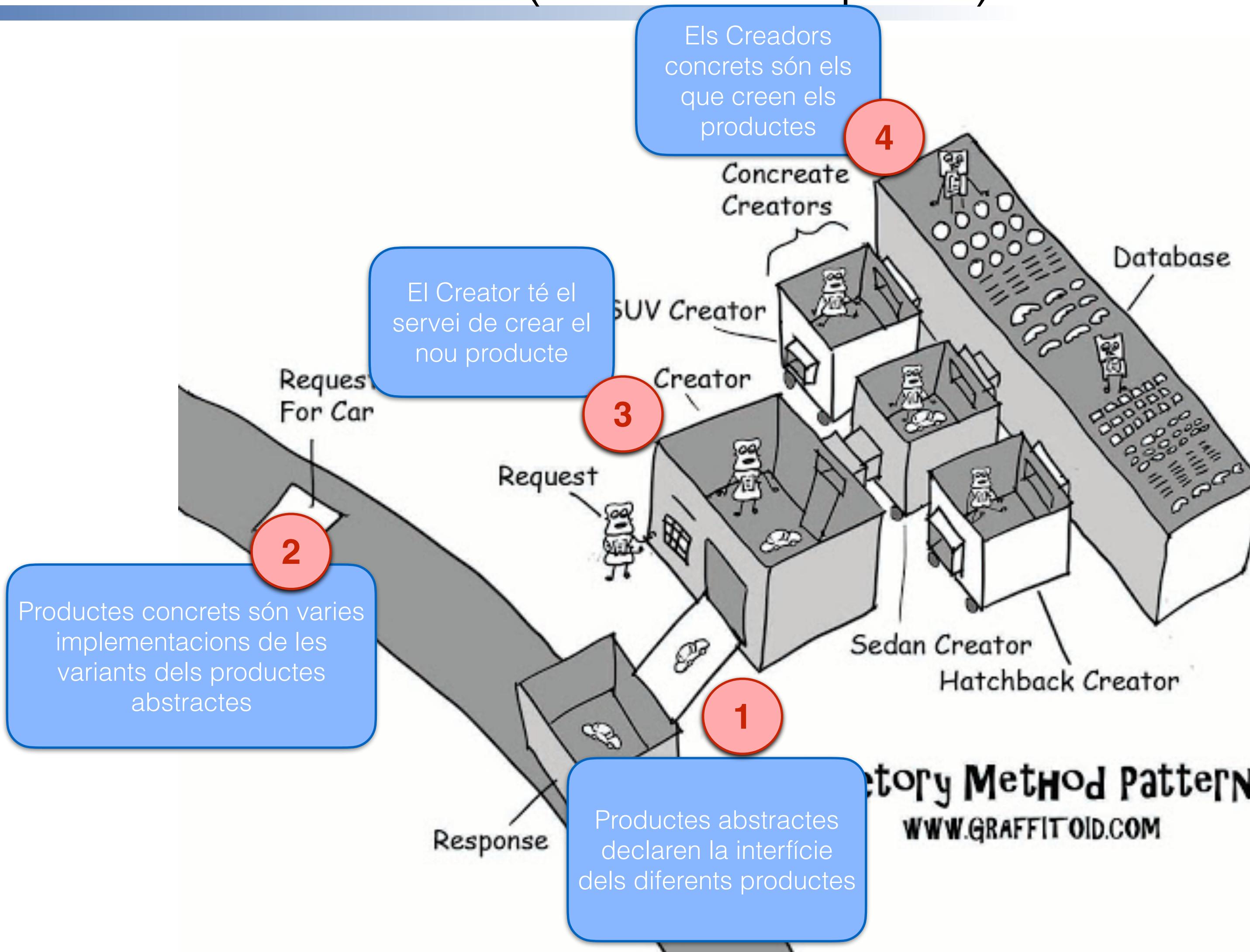
Creator proporciona la signatura d'un mètode per crear els objectes.

La resta de mètodes a la classe Creator són per operar amb els productes creats en el ConcreteCreator

Creator **NO crea els objectes**

ConcreteCreators creen els objectes de la jerarquia **Product**

Patró Factory Method (versió completa)



Patró Factory Method

Nom del patró: Factory method

Context: Creació

On s'usa en la realitat?

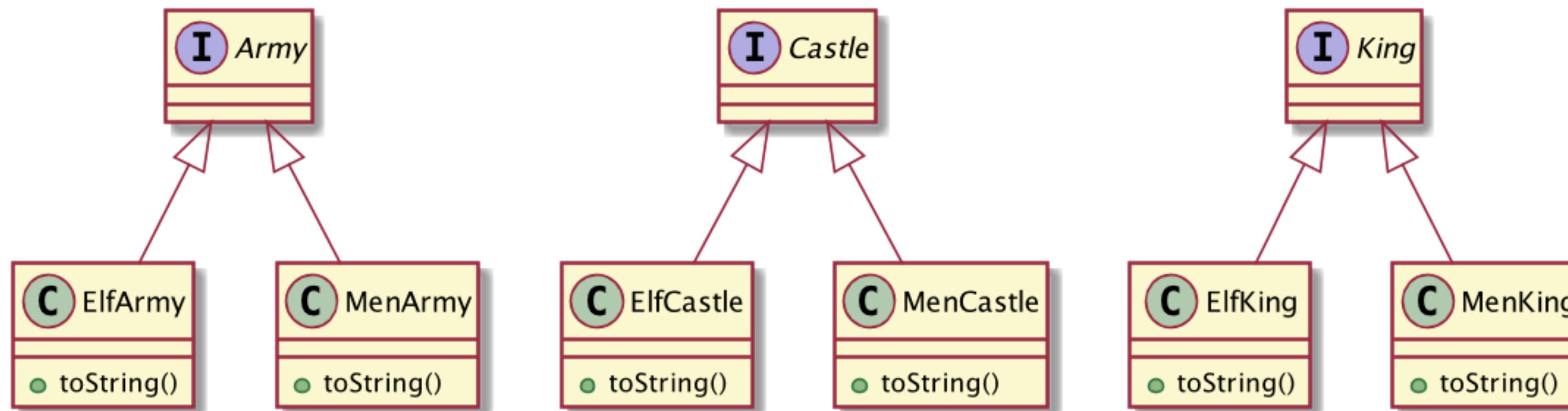
- A la JDK per exemple:
 - getInstance() de java.util.NumberFormat o ResourceBundle
 - wrapper classes com Integer, Boolean, etc. per a retornar valors en usar el mètode valueOf()
 - java.nio.charset.Charset.forName(),
java.sql.DriverManager#getConnection(),
java.net.URL.openConnection(), java.lang.Class.newInstance(),
java.lang.Class.forName()

Patrons Factory

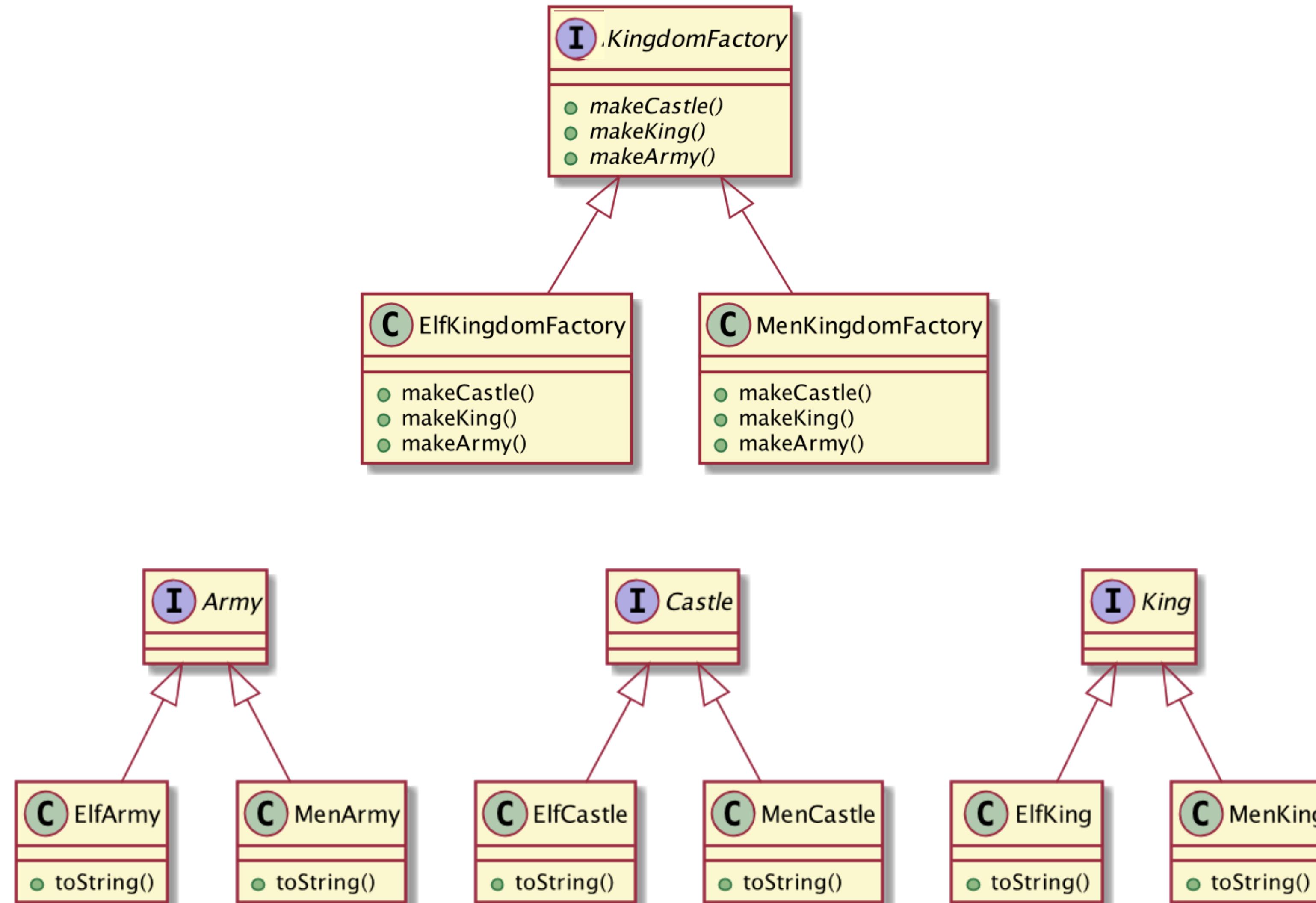
- **Factory Method** – Defineix una classe abstracte per crear objectes, però deixa a les subclasses decidir quina classe ha d'instanciar i consulta el nou objecte creat a través d'una interfície comú dels objectes creats
- **Abstract Factory** – Ofereix una interfície per crear una **família d'objectes** relacionats, sense explícitament especificar les seves classes

Exemple Patró Abstract Factory

- Volem crear dos regnes (els dels elfs i els dels homes). Cada regne té un castell, un rei i una armada. Per a cada un dels elements d'un regne es dissenya una interfície
- Com solucionem la seva creació “coordinada”?

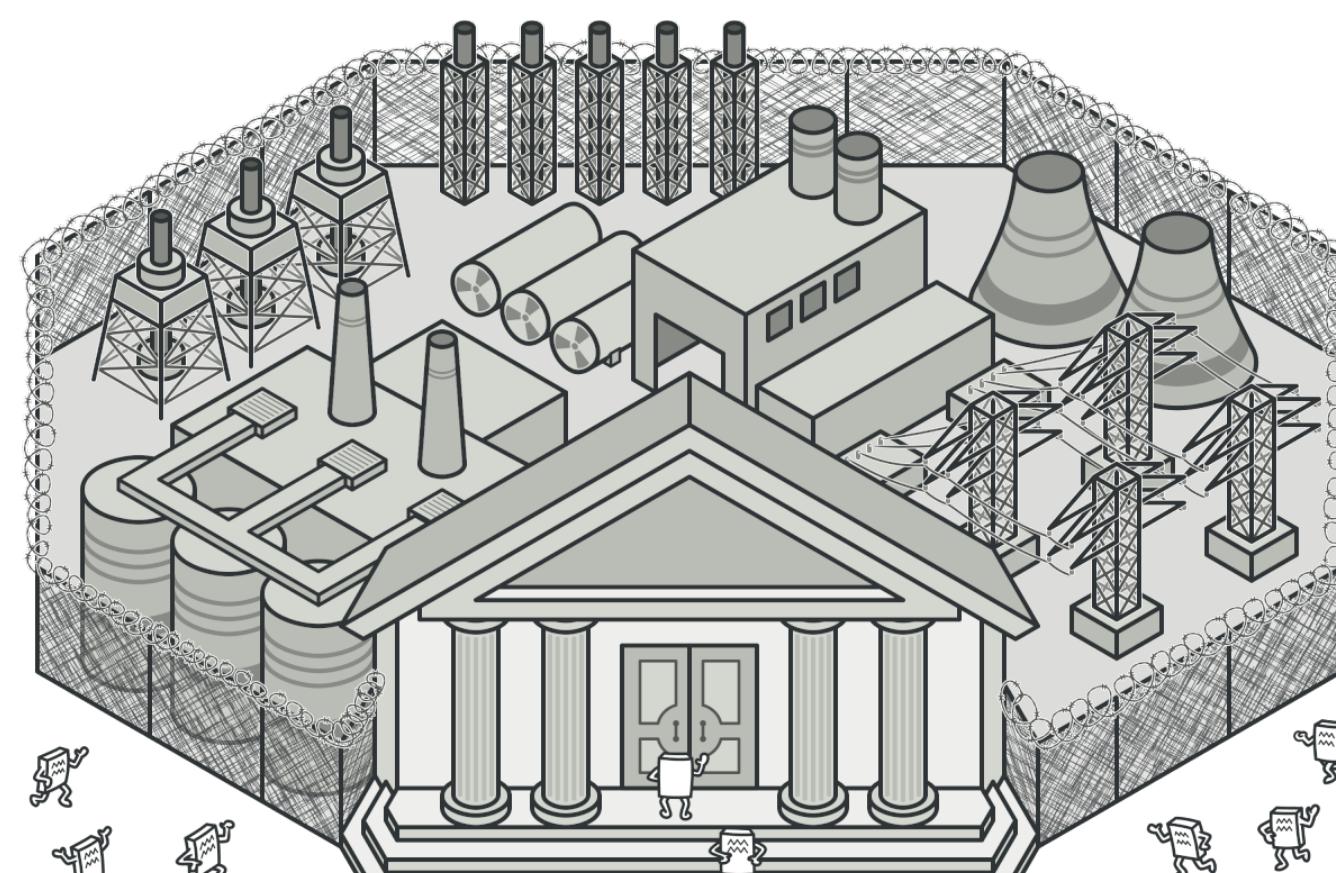


Exemple Patró Abstract Factory



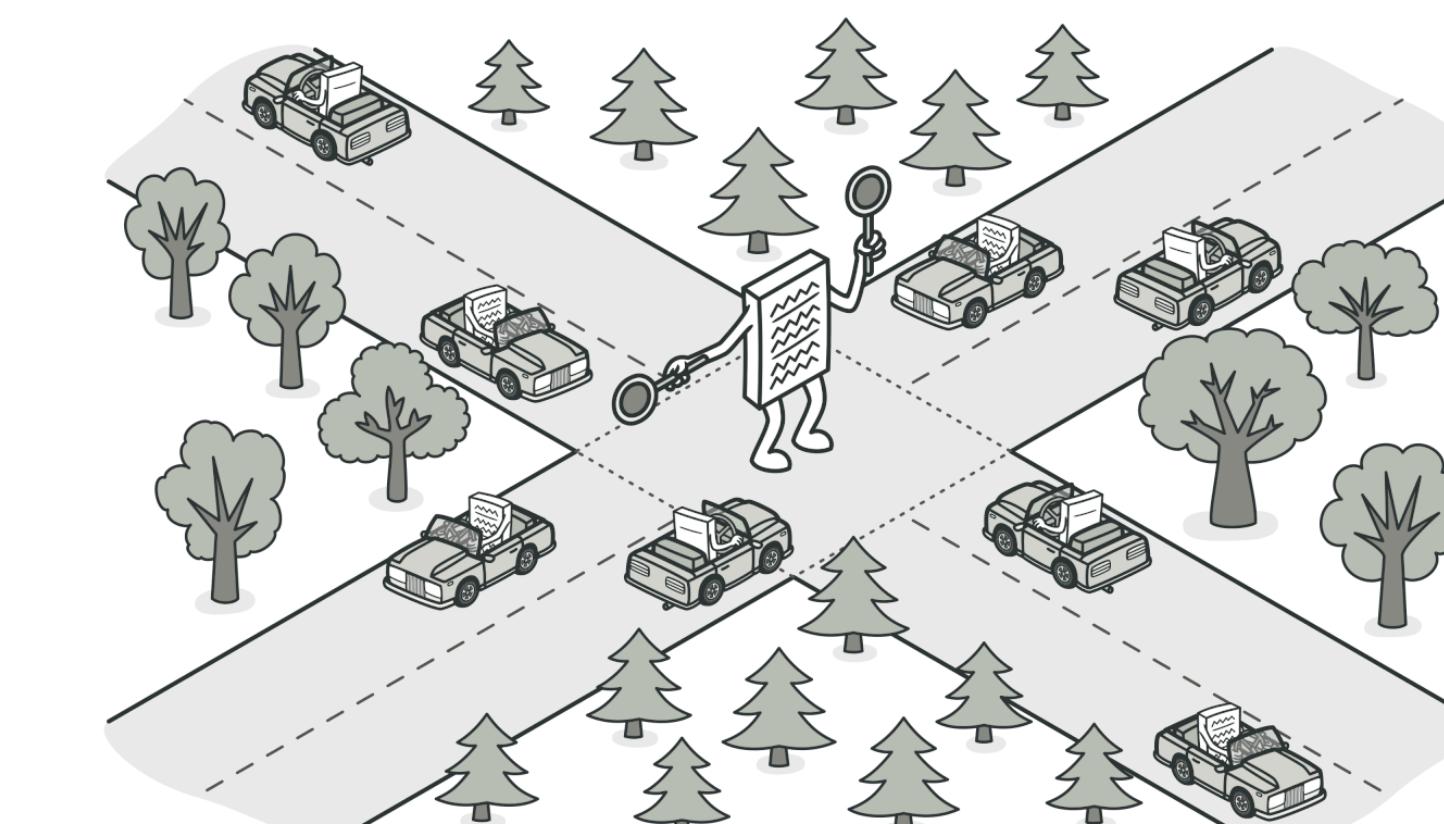
3.5. Patrons de disseny: Patró Façana (Facade)

- Patrons relacionats
 - **Mediator** té una feina similar d'organitzar tasques de classes acoblades.
 - “Normalment” la Façana es transforma en una classe única usant el patró **Singleton**



Facade

les classes internes no coneixen la façana

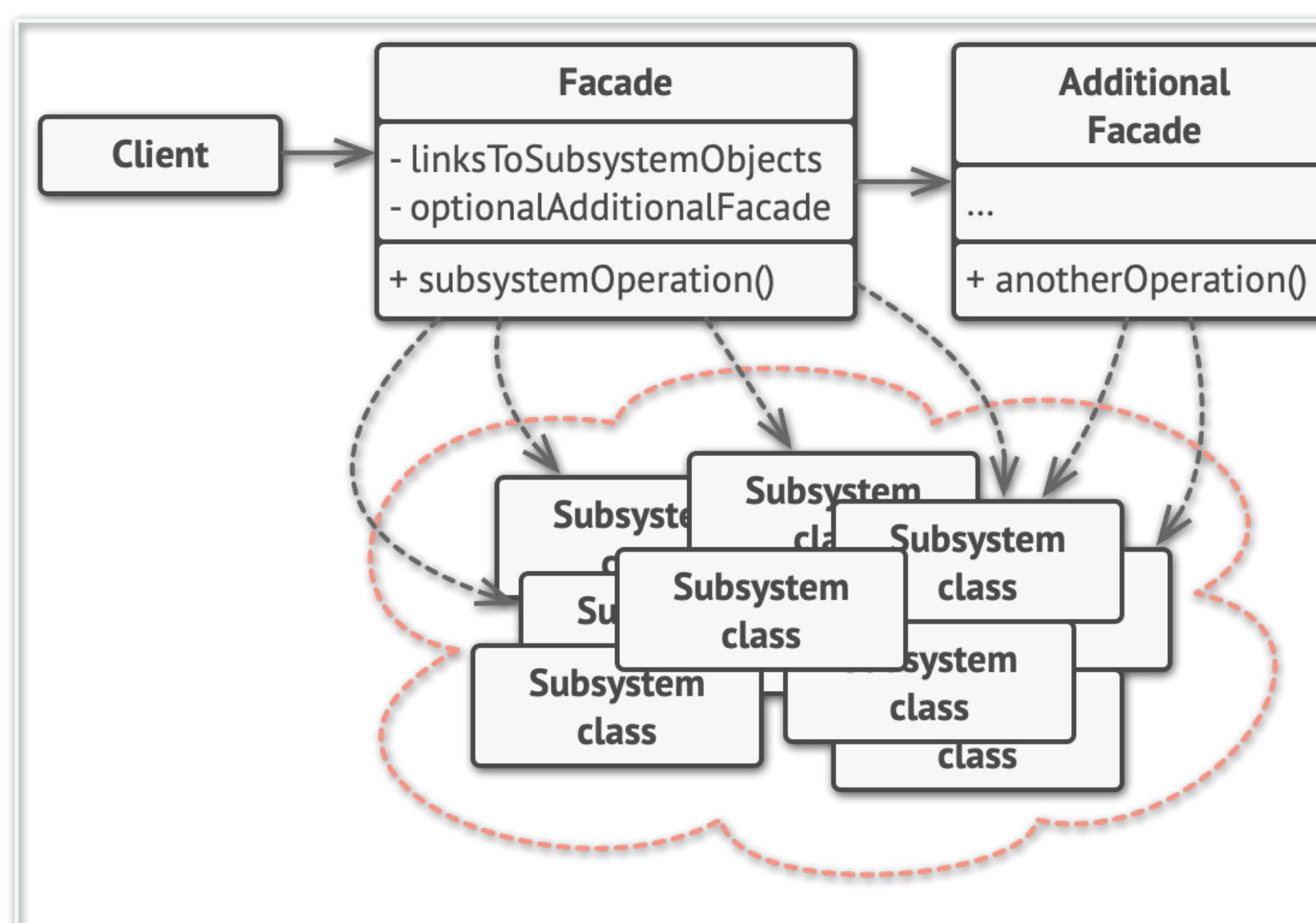


Mediator

les classes comparteixen el mediador per comunicar-se entre elles

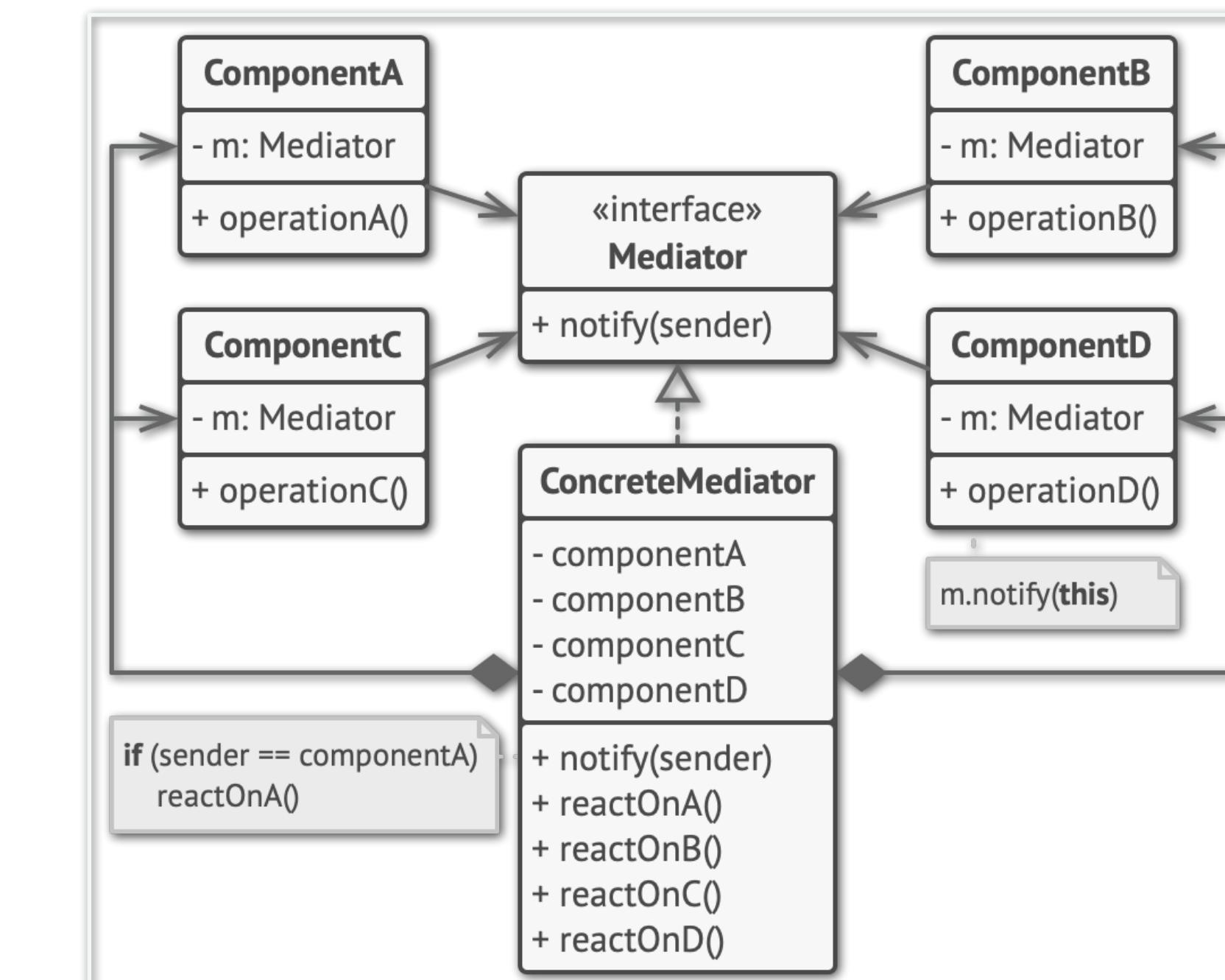
3.5. Patrons de disseny: Patró Façana (Facade)

- Patrons relacionats
 - **Mediator** té una feina similar d'organitzar tasques de classes acoblades.



Facade

les classes internes no coneixen la façana

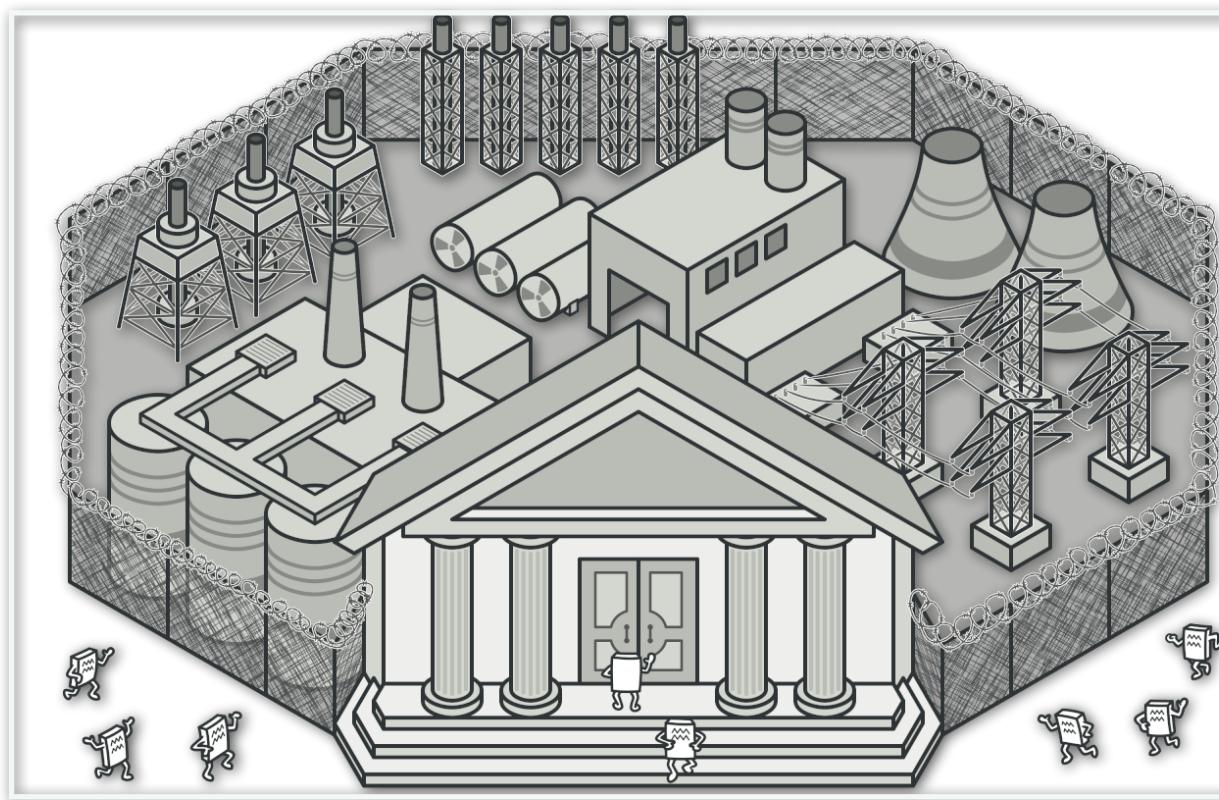


Mediator

les classes comparteixen el mediador per comunicar-se entre elles

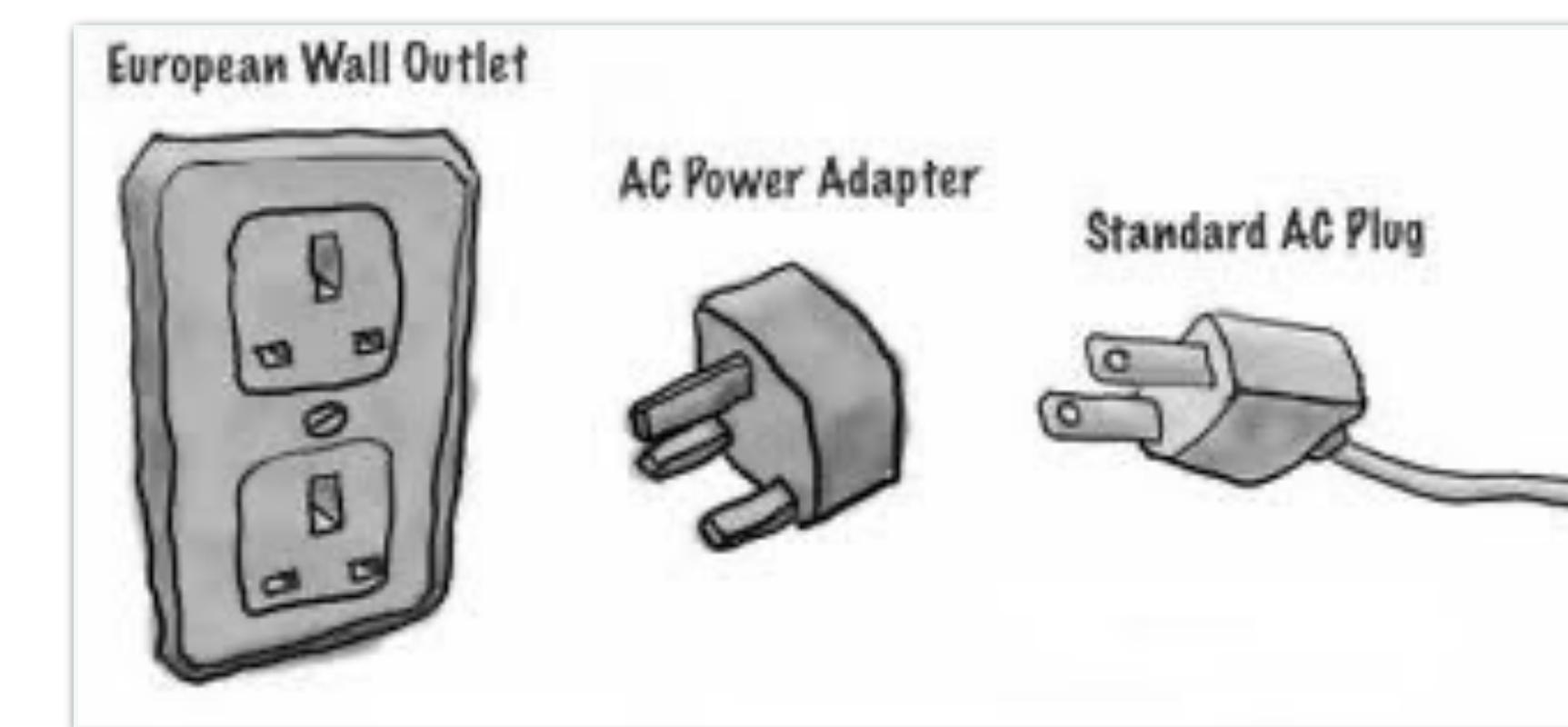
3.5. Patrons de disseny: Patró Façana (Facade)

- Patrons relacionats
 - **Adapter** té una feina de comunicar diferents interfícies (o signatures) - No afegeix utilitats al sistema



Facade

Pot combinar utilitats



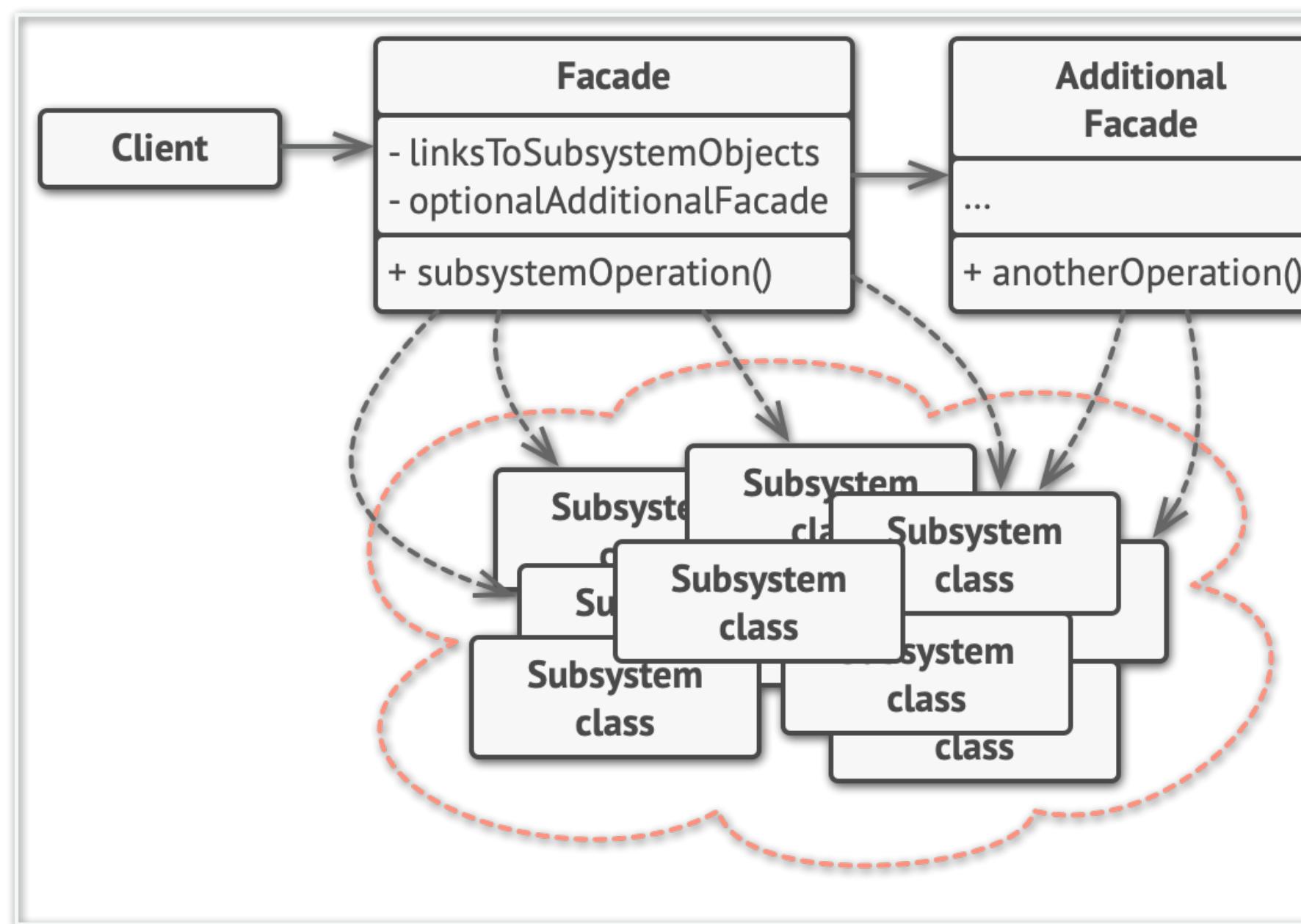
Adapter

Només converteix interfícies.

No afegeix lògica o comportament nous

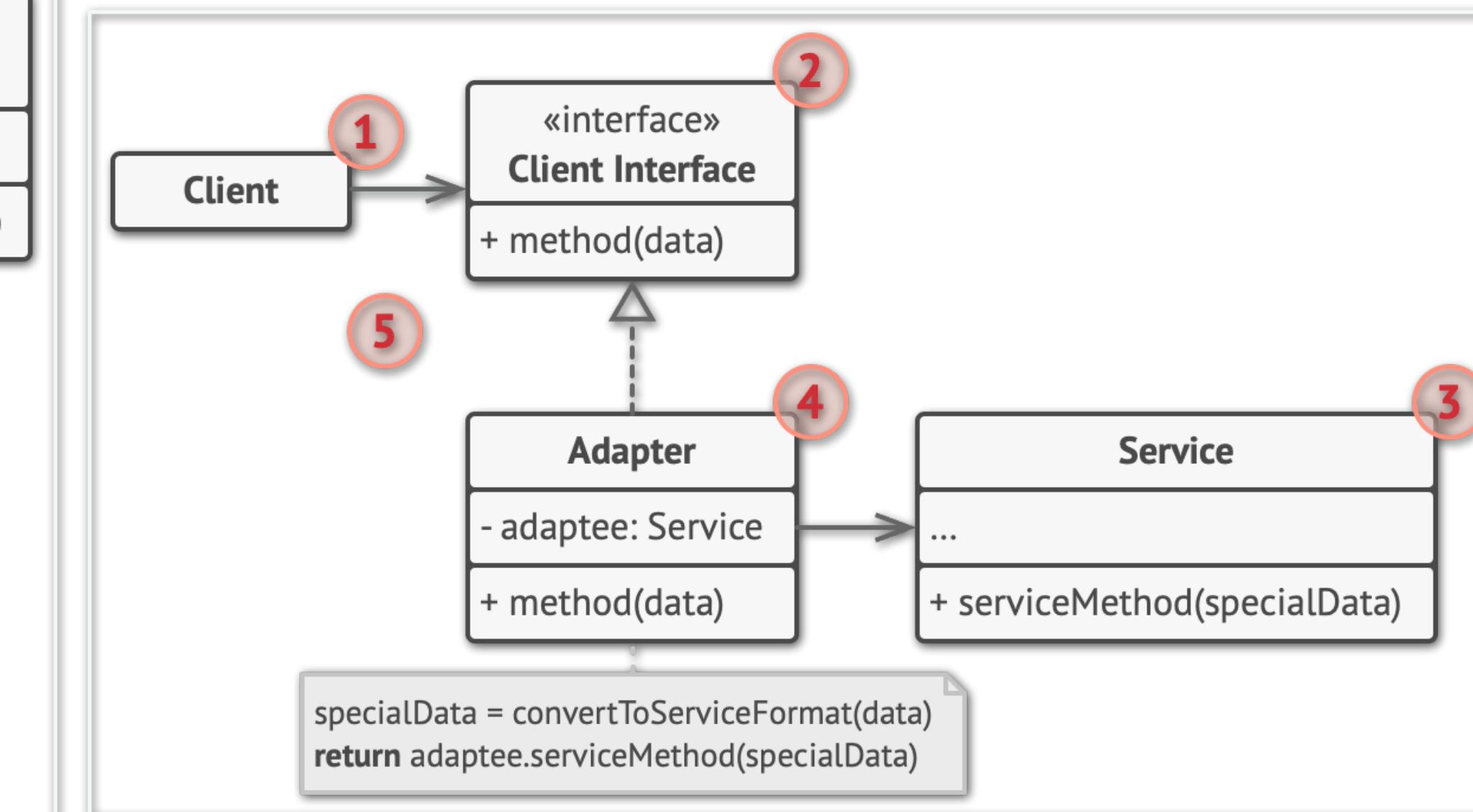
3.5. Patrons de disseny: Patró Façana (Facade)

- Patrons relacionats
 - **Adapter** té una feina de comunicar diferents interfícies (o signatures) - No afegeix utilitats al sistema



Facade

Pot combinar utilitats



Adapter

Només converteix interfícies. No afageix lògica ni comportament addicional