

<https://refactoring.guru/design-patterns/factory-method>



Tema 3: Disseny

Anna Puig

Enginyeria Informàtica

Facultat de Matemàtiques i Informàtica,

Universitat de Barcelona

Curs 2021/2022



UNIVERSITAT DE
BARCELONA

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

3.4. Patrons de disseny

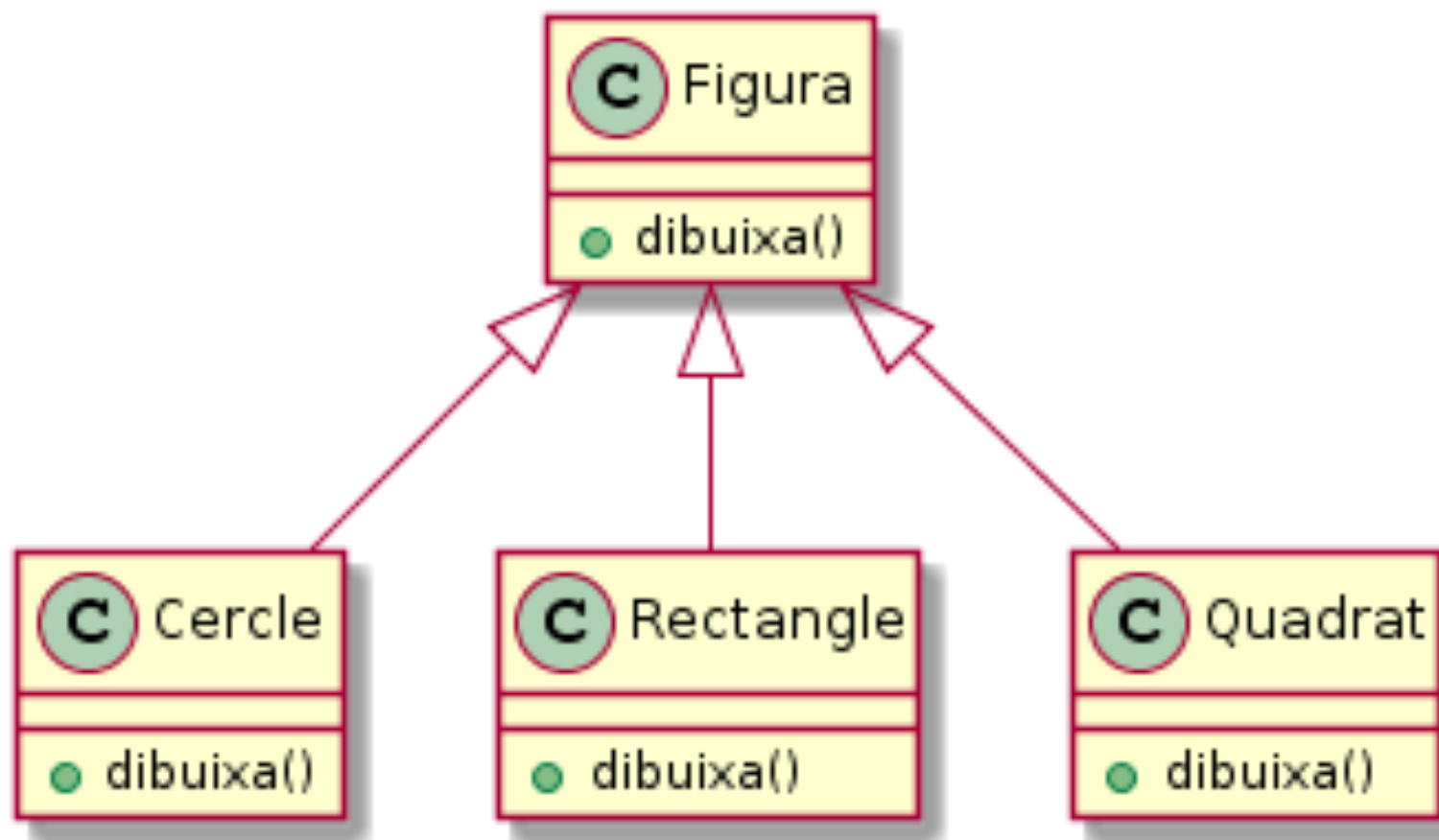
Propòsit → Àmbit ↓	CREACIÓ	ESTRUCTURA	COMPORTAMENT
CLASSE	<ul style="list-style-type: none"> • Factory method 	<ul style="list-style-type: none"> • class Adapter 	<ul style="list-style-type: none"> • Interpreter • Template method
OBJECTE	<ul style="list-style-type: none"> • Abstract Factory • Builder • 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

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

El problema

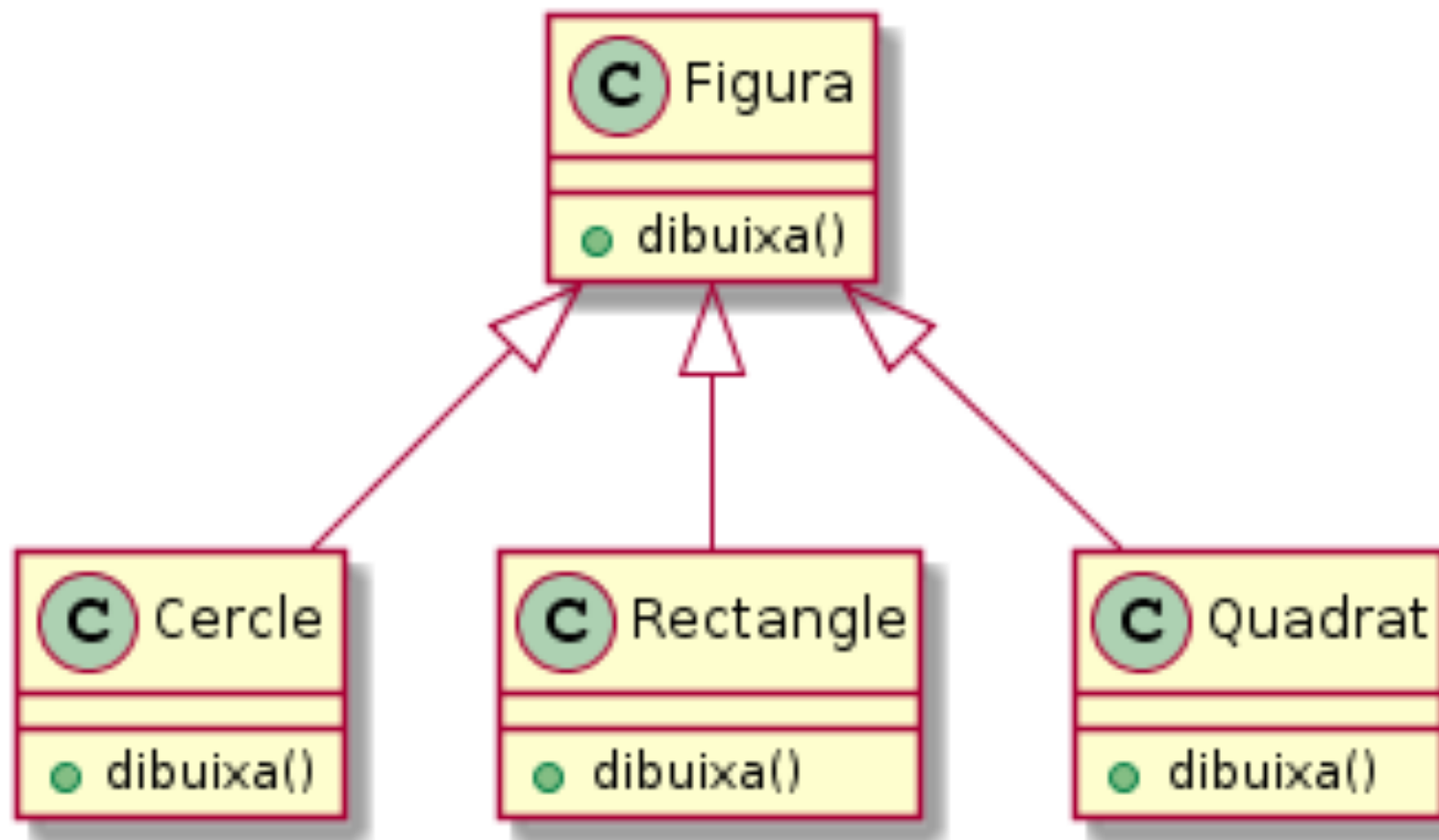


PROBLEMA

- Es vol tenir el codi per a crear les instàncies dels diferents tipus de figures

Patró Factory Method

El problema



Dibuixador

```
if (tipus == 1)
    f = crea Cercle( )
else if (tipus == 2)
    f = crea Rectangle( )
else if (tipus == 3 )
    f = crea Quadrat( )

f.dibuixa()
```

PROBLEMA

- Què passa si ara afegim un Triangle?

- Vulnera el principi S.O.L.I.D.

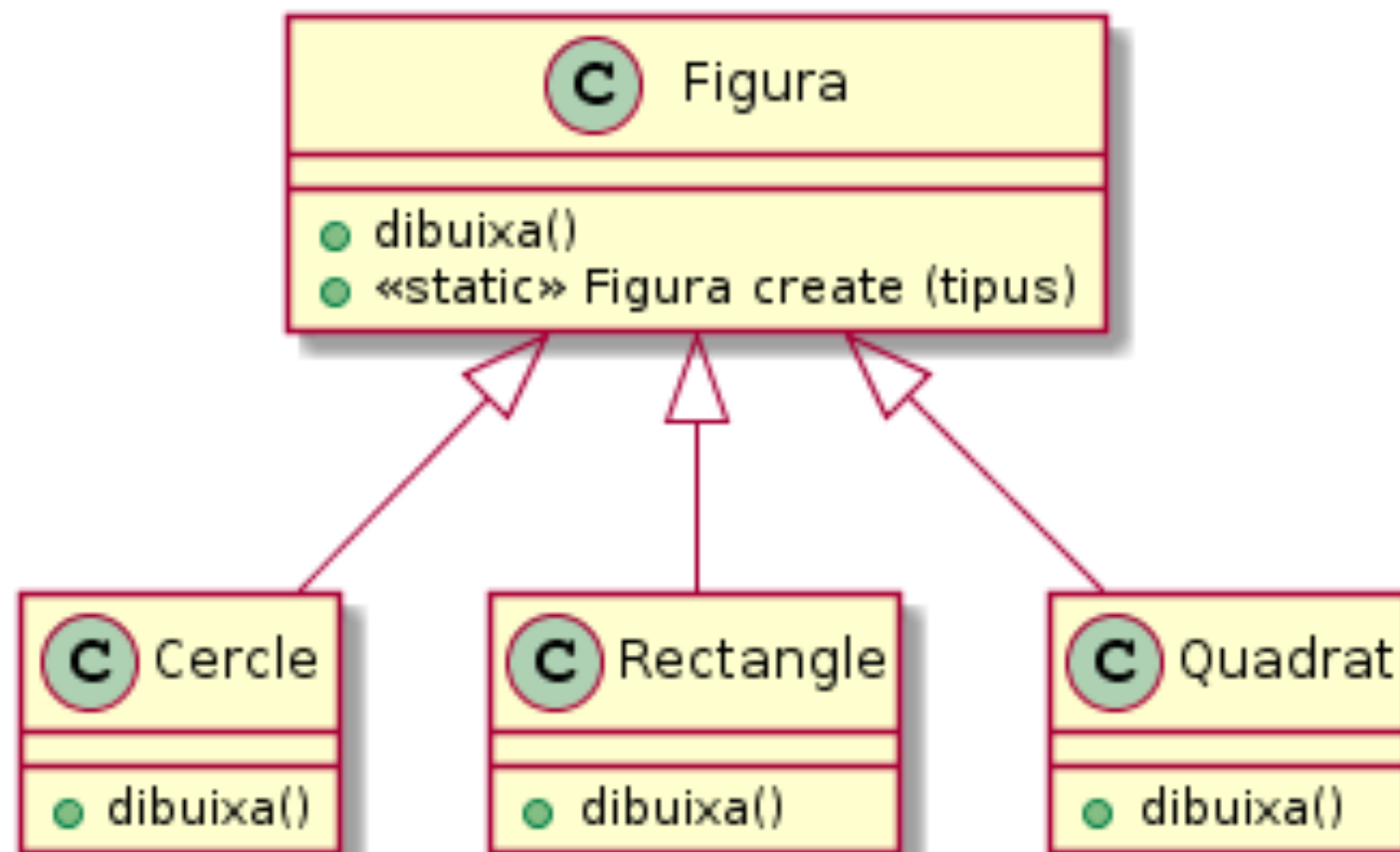
Open-Closed

Patró Factory

Una primera solució?

Dibuixador

```
f = Figura.create (1)  
f.dibuixa()
```

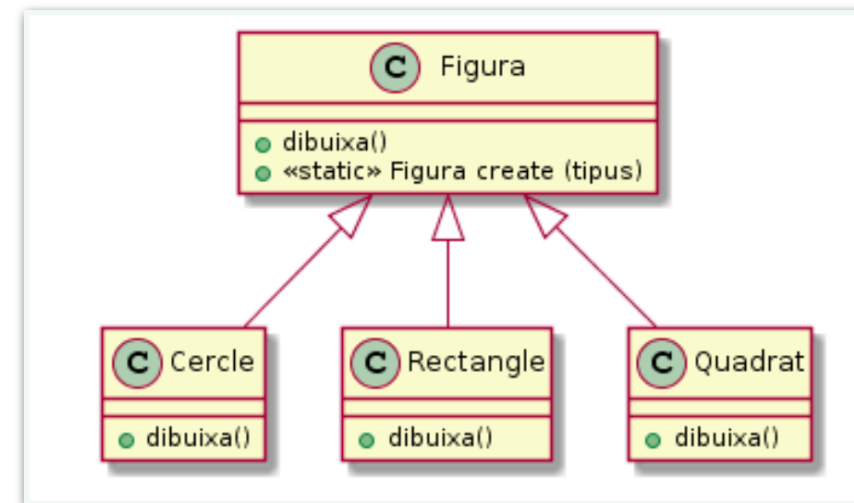


SOLUCIÓ

- Es pot fer un mètode **static** a la classe Figura que faci el switch segons el tipus.
- No es podrà sobrecarregar el mètode **create** amb l'herència

Patró Factory Method

```
public class Figura{  
    public static Figura create(int tipus)  
    {  
        if (tipus == 1) { return new Cercle(); }  
        else if (tipus == 2) { return new Rectangle(); }  
        else if (tipus == 3 ) { return new Quadrat(); }  
  
        return null;  
    }  
}
```



- Vulnera el principi ? de SOLID.

OCP

LSP

Patró Factory Method

Nom del patró: Factory Method

Context: Creació

Problema:

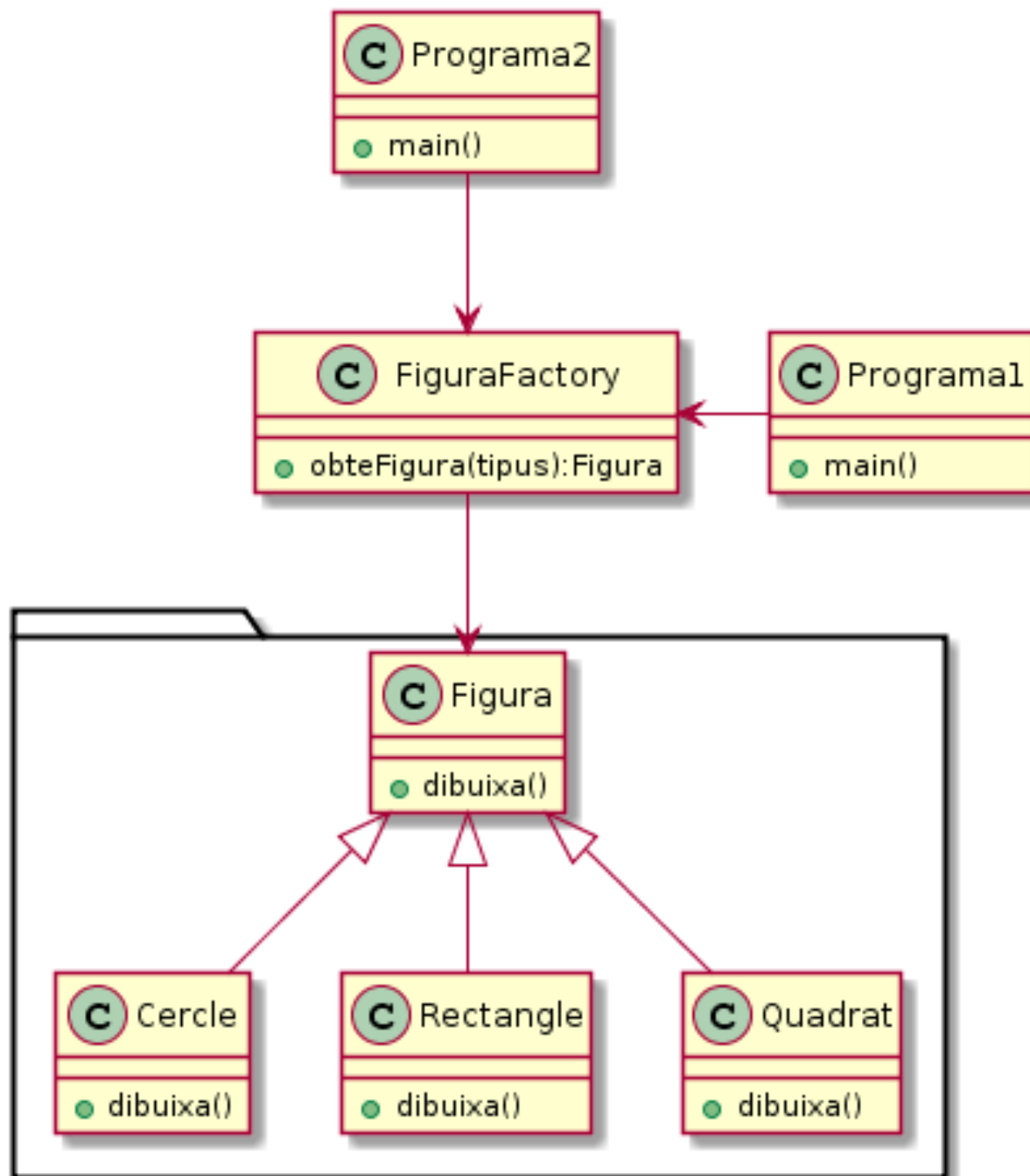
- **Separar la classe que crea els objectes, de la jerarquia d'objectes a instanciar**

Solució:

- Es separa la classe que crea els objectes, de la jerarquia d'objectes a instanciar
- Permet que una classe postposi la instanciació a les subclasses (són aquestes les que decideixen quina classe instanciar)
- L'aplicació client no sap la lògica per crear l'objecte i crea l'objecte a partir d'una interfície comuna

Patró Factory Method

Primera aproximació (versió simplificada del Factory Method)



SOLUCIÓ:

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

Patrón Factory Method

```
public class Programa1{  
    public static void main (string[] args) {  
        FiguraFactory factory = new FiguraFactory();  
        Figura fig1 = factory.obteFigura(1);  
        fig1.dibuixa(); // pintarà un Cercle  
        Figura fig2 = factory.obteFigura(2);  
        fig2.dibuixa(); // pintarà un Rectangle  
        Figura fig3 = factory.obteFigura(3);  
        fig3.dibuixa(); // pintarà un Quadrat  
    }  
}
```

Patró Factory Method

```
public class FiguraFactory{  
    public Figura obteFigura(int tipus)  
    {  
        if (tipus == 1) { return new Cercle(); }  
        else if (tipus == 2) { return new Rectangle(); }  
        else if (tipus == 3 ) { return new Quadrat(); }  
  
        return null;  
    }  
}
```

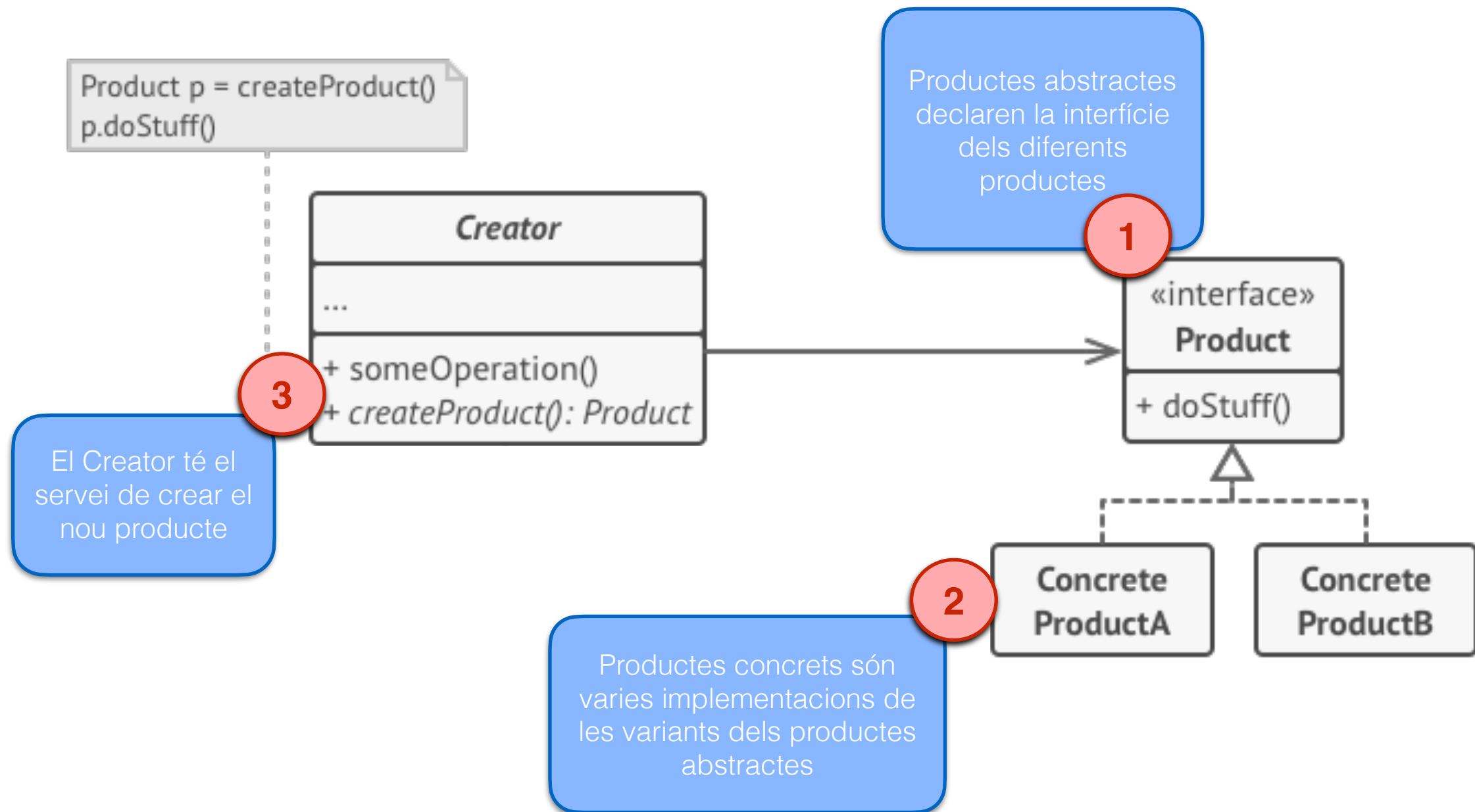
Es creen les diferents instàncies derivades de Figura aquí, no en els diferents programes o aplicacions

Es poden afegir més figures sense haver de modificar els programes. Només cal modificar FiguraFactory.

OCP a nivell de Factory

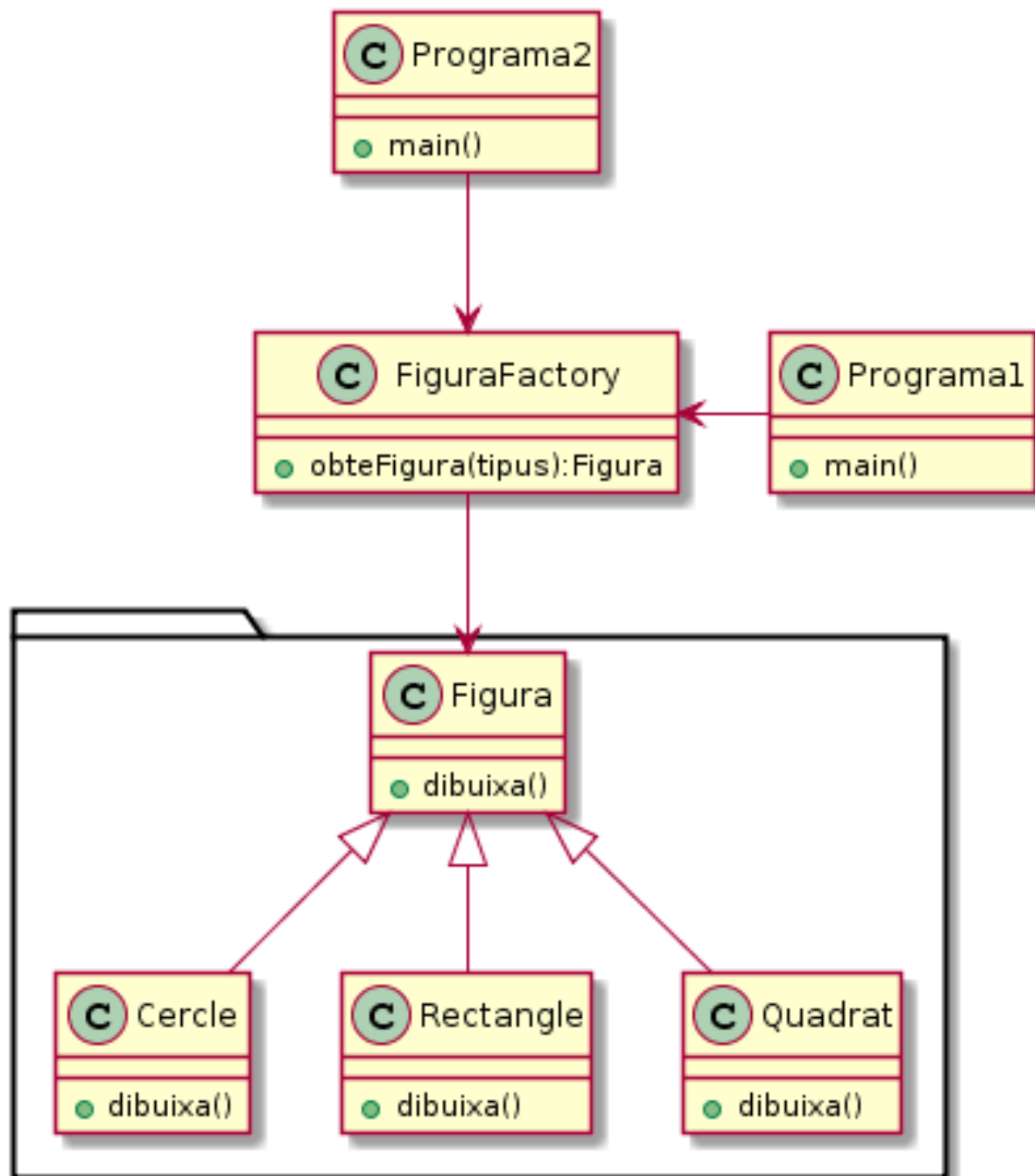
Patrón Simple Factory

Primera aproximació
(versió simplificada del Factory Method)



Patró Factory Method

Primera aproximació (versió simplificada del Factory Method)



Anàlisi d'aquesta aproximació:

- Què passa si es vol tenir diferents representacions de les figures (imatges o punts o línies), segons l'aplicació?
- Vulnera el principi **D** de S.O.L.I.D. (la Factory depèn de totes les filles concretes de la jerarquia)

DIP

Patró Factory Method

(versió completa)

Es poden tenir dues jerarquies paral·leles: **una per les Factories** de diferents tipus (imatges, punts, línies) i **una altra per les figures**

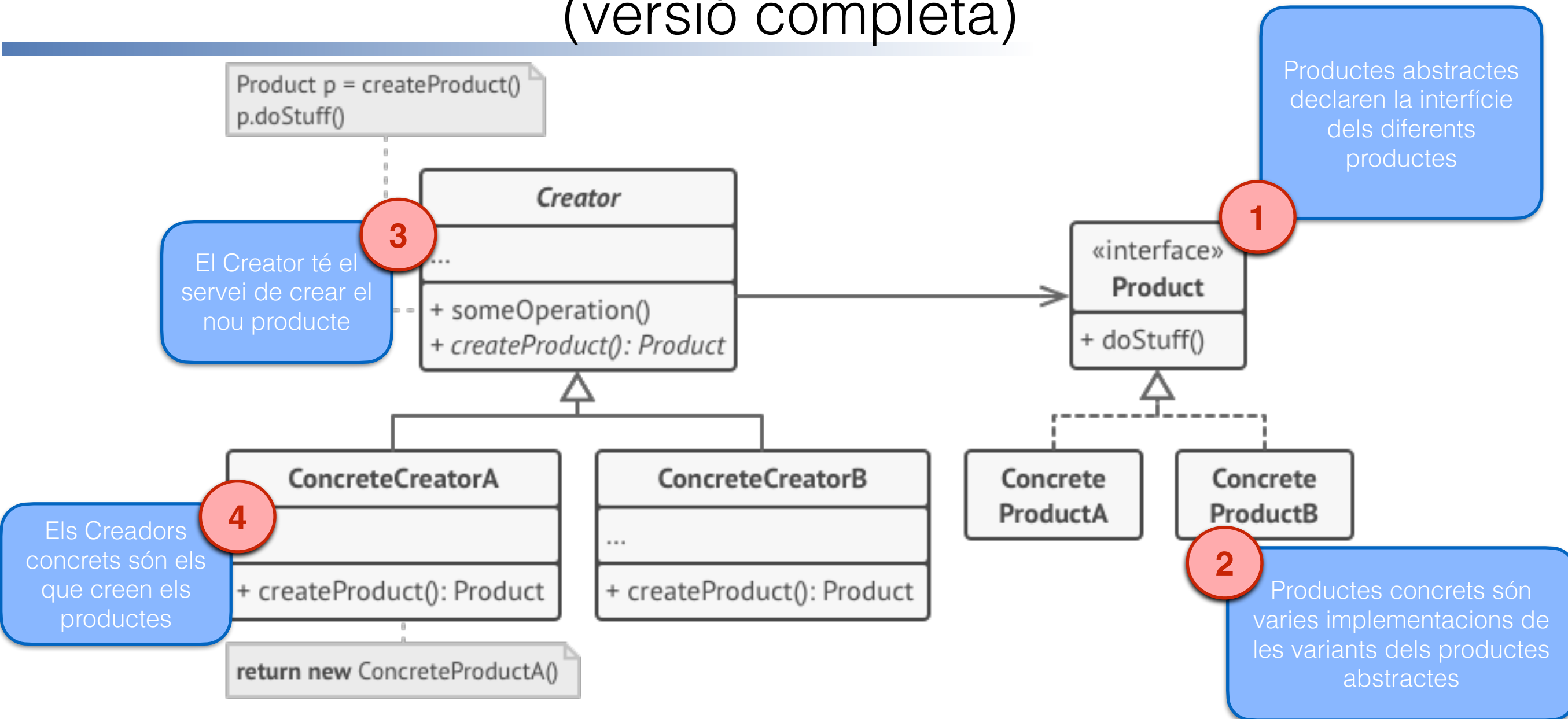
```
public abstract FiguresFactory {  
    public Figura display(String tipus) {  
        Figura f;  
        f = createFigura(tipus);  
        f.dibuixa();  
    }  
    public abstract Figura createFigura(String tipus)  
}
```

```
public FiguresImatgeFactory extends FiguresFactory{  
    Figura createFigura (String tipus) {  
        Figura f;  
        switch (tipus) {  
            case "Quadrat": f = new QuadratImatge();  
                            break;  
            case "Cercle": f = new CercleImatge();  
                            break;  
            case "Rectangle": f = new RectangleImatge();  
                              break;  
        }  
        return f;  
    }  
}
```

```
public FiguresPuntsFactory extends FiguresFactory{  
    Figura createFigura (String tipus) {  
        Figura f;  
        switch (tipus) {  
            case "Quadrat": f = new QuadratPunts();  
                            break;  
            case "Cercle": f = new CerclePunts();  
                            break;  
            case "Rectangle": f = new RectanglePunts();  
                              break;  
        }  
        return f;  
    }  
}
```


Patrón Factory Method

(versió completa)



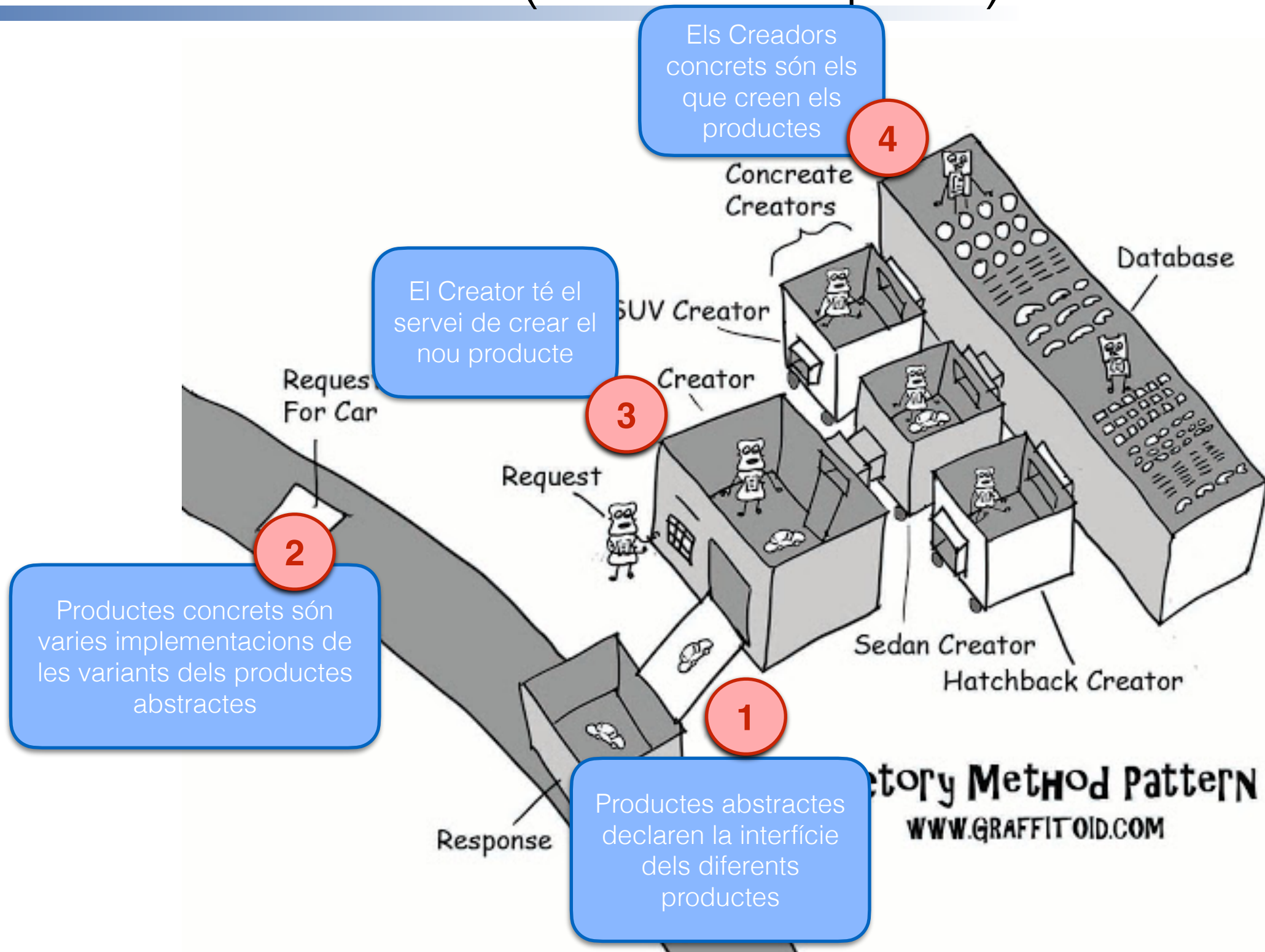
Creator proporciona la signatura d'un mètode per crear els objectes.
La resta de mètodes a la classe Creator són per operar amb els productes creats en el ConcreteCreator

Creator NO crea els objectes

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

Patrón Factory Method

(versió completa)



Patró Factory Method

Nom del patró: Factory method

Context: Creació

Pros:

- Centralització en la creació d'objectes
- Facilita l'escalabilitat del sistema
- L'usuari s'abstrau de la instància a crear

És un dels patrons de disseny més usats i més robustos

Cons:

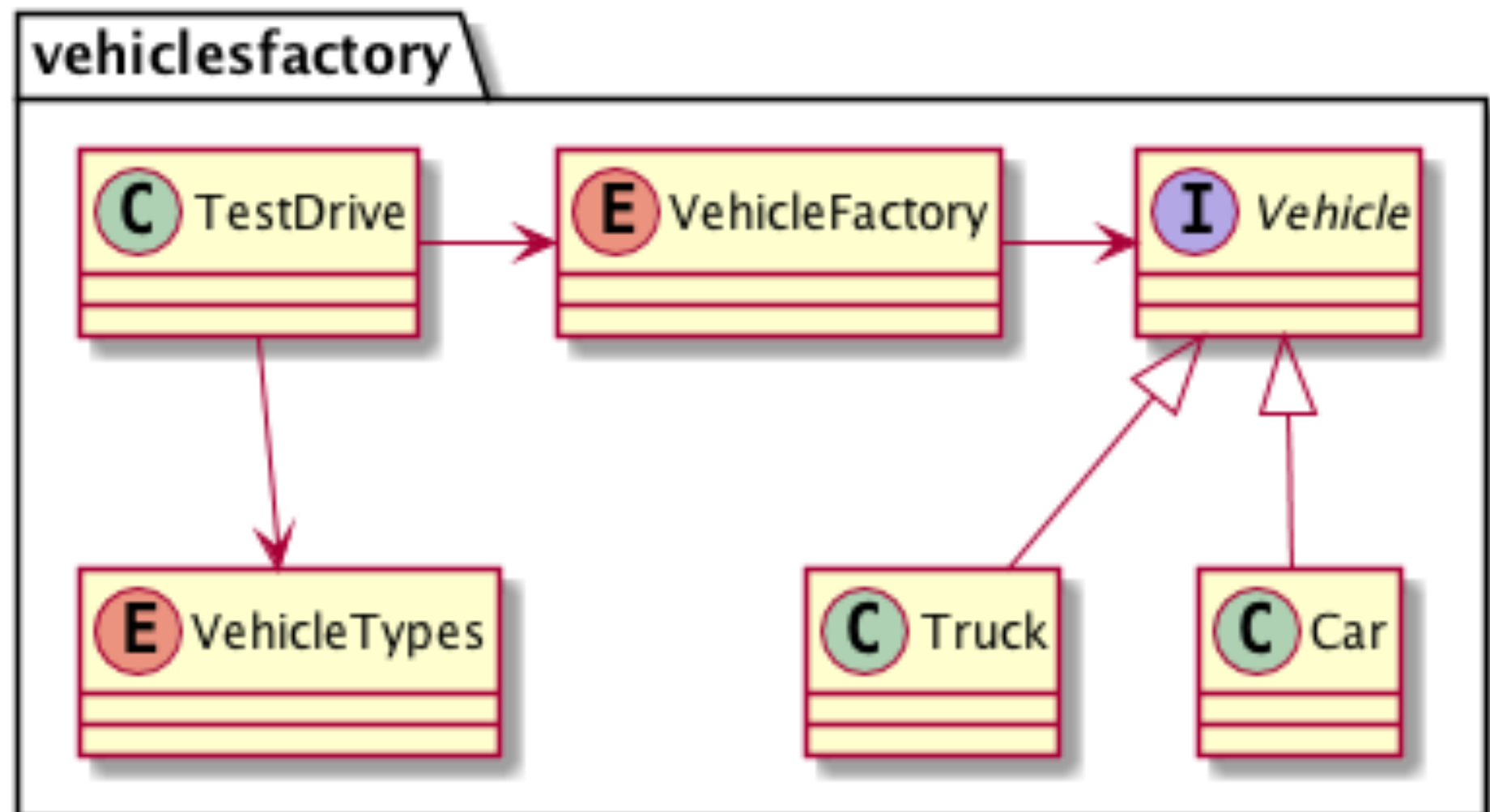
- Potser afegeix una complexitat innecessària en el codi de la vostra aplicació
- Els constructors poden vulnerar el Open-Closed Principle (veure transp.)

En tot cas, si es fa molt sovint la creació de molts objectes del mateix tipus base i necessiteu manipular-los com objectes abstractes, segurament necessiteu una Factory

OCP a nivell de Creador

Sobre el Principi Obert-Tancat als patrons Factory i Factory Method

Vulnera el principi obert-tancat? Com evitar-ho?
<http://java.globinch.com/patterns/design-patterns/factory-design-patterns-and-open-closed-principle-ocp-in-solid/>



Sobre el Principi Obert-Tancat als patrons Factory

```
public static void main(String[] args) {  
    try {  
        VehicleFactory factory = VehicleFactory.INSTANCE;  
        Vehicle vehicle = factory.createVehicle("car");  
        vehicle.drive();  
        vehicle = factory.createVehicle("truck");  
        vehicle.drive();  
        vehicle = factory.createVehicle("truck1");  
        vehicle.drive();  
    } catch (Exception e) {  
        System.out.println(e.getMessage());  
    }  
}
```

Sobre el Principi Obert-Tancat als patrons Factory

```
public enum VehicleFactory {  
    INSTANCE;  
    /**  
     * @author Binu George  
     */  
    public Vehicle createVehicle(String vehicleType) throws Exception {  
        if (vehicleType.equalsIgnoreCase("car")) {  
            return new Car();  
        } else if (vehicleType.equalsIgnoreCase("truck")) {  
            return new Truck();  
        }  
        throw new Exception("The vehicle type is unknown!");  
    }  
}
```

Vulnera el principi obert-tancat? Com evitar-ho?

Sobre el Principi Obert-Tancat als patrons Factory: ús de reflexivitat

```
public enum VehicleTypes {  
    Car, Truck  
}
```

```
public static void main(String[] args) {  
    try {  
        VehicleFactory factory = VehicleFactory.INSTANCE;  
        Vehicle vehicle = factory.createVehicle(VehicleTypes.Car.name());  
        vehicle.drive();  
        vehicle = factory.createVehicle(VehicleTypes.Truck.name());  
        vehicle.drive();  
        vehicle = factory.createVehicle(VehicleTypes.Car.name());  
        vehicle.drive();  
        vehicle = factory.createVehicle("Truck2");  
        vehicle.drive();  
    } catch (Exception e) {  
        System.out.println(e.getMessage());  
    }  
}
```


Sobre el Principi Obert-Tancat als patrons Factory: ús de reflexivitat

```
public enum VehicleFactory {  
    INSTANCE;  
  
    public Vehicle createVehicle(String vehicleType) throws Exception {  
        Vehicle vehicle = null;  
        String name = Vehicle.class.getPackage().getName();  
        try {  
            vehicle = (Vehicle) Class.forName(name + "." + vehicleType).newInstance();  
            return vehicle;  
        } catch (InstantiationException e) {  
            throw new Exception("The vehicle type is not valid as a object!");  
        } catch (IllegalAccessException e) {  
            throw new Exception("The vehicle type is not found!");  
        } catch (ClassNotFoundException e) {  
            throw new Exception("The vehicle class is unknown!");  
        }  
    }  
}
```

Quantes instàncies diferents de cada vehicle es tenen?

Sobre el Principi Obert-Tancat als patrons Factory

```
public enum VehicleFactory {  
    INSTANCE;  
}
```

```
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!");  
        }  
    }  
}
```

Solució de Factory només amb la possibilitat de fer una única instància de cada tipus de vehicle

Patrón 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()