



UNIVERSITÉ
DE MONTPELLIER



Design Pattern

Présentation

M3105, ACDA, CPOA, GL, toto,...

- Production d'une conception détaillée (**StarUML**) en appliquant des patrons de conception (**design pattern**)
- Réalisation (**Java**) et application des bonnes pratiques POO

Conception Orientée Objet

Concepts de base

- **Classes/Objets** : Système est un ensemble d'objets produits par des classes, des objets qui communiquent entre eux par appels de méthodes
- **Encapsulation** : accès privée de la structure de l'objet, accès publique des services (méthodes)
- **Héritage** : Spécialisation/Généralisation de classes organisées en arborescence
- **Substitution** : Une sous-classe qui prend le rôle d'une super-classe
- **Surcharge** : Différentes versions d'une même méthode selon le nombre et le type des paramètres fournis
- **Polymorphisme** : Une méthode d'une sous-classe peut modifier le comportement de la même méthode de la super-classe.

Conception Orientée Objet

Why COO?

- **Sécurité** : accès privé d'une partie d'un objet
- **Souplesse** : Les méthodes polymorphes permettent de modifier le comportement des sous-classes sans modifier le comportement des super-classes
- **Factorisation** : Réutilisation du code des super-classes
- **Réutilisation** : Faire appel aux services des objets sans avoir à comprendre comment le service est réalisé

Conception Orientée Objet

Maintenance et évolutivité

- **Rigidité** : Effet avalanche suite à une petite modification dans la conception / code
- **Fragilité** : Conception / code en cristal sensible aux modifications
- **Immobilisme** : Conception / code impossible à réutiliser
- **Viscosité** : Conception / code à réviser au lieu de le réutiliser
- **Opacité** : Conception / code difficile à comprendre

Conception Orientée Objet

Maintenance et évolutivité

- **Rigidité** : Effet avalanche suite à une petite modification dans la conception / code
- **Fragilité** : Conception / code en cristal sensible aux modifications
- **Immobilisme** : Conception / code impossible à réutiliser
- **Viscosité** : Conception / code à réviser au lieu de le réutiliser
- **Opacité** : Conception / code difficile à comprendre

Principes SOLID !

Conception Orientée Objet

Principes SOLID

- Agile Software Development, Principles, Patterns and Practices.

Robert C. Martin, 2002

- Single responsibility principle
- Open close principle
- Liskov principle
- Interface segregation principle
- Dependency inversion principle

Principes **SOLID**

Single responsibility principle

Principe : Si une classe a plus d'une responsabilité, ces dernières seront couplées. Les modifications apportées à une responsabilité impacteront les autres, augmentant la **rigidité** et la **fragilité** de la conception / du code.

Principes SOLID

Single responsibility principle

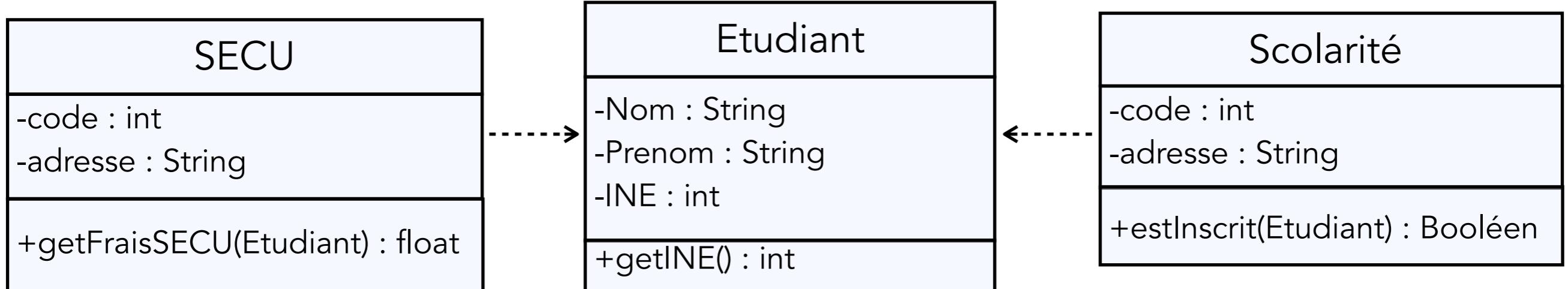
Principe : Si une classe a plus d'une responsabilité, ces dernières seront couplées. Les modifications apportées à une responsabilité impacteront les autres, augmentant la **rigidité** et la **fragilité** de la conception / du code.

Etudiant
-Nom : String -Prenom : String -INE : int
+getINE() : int +estInscrit() : Booléen +getFraisSECU() : float

Principes SOLID

Single responsibility principle

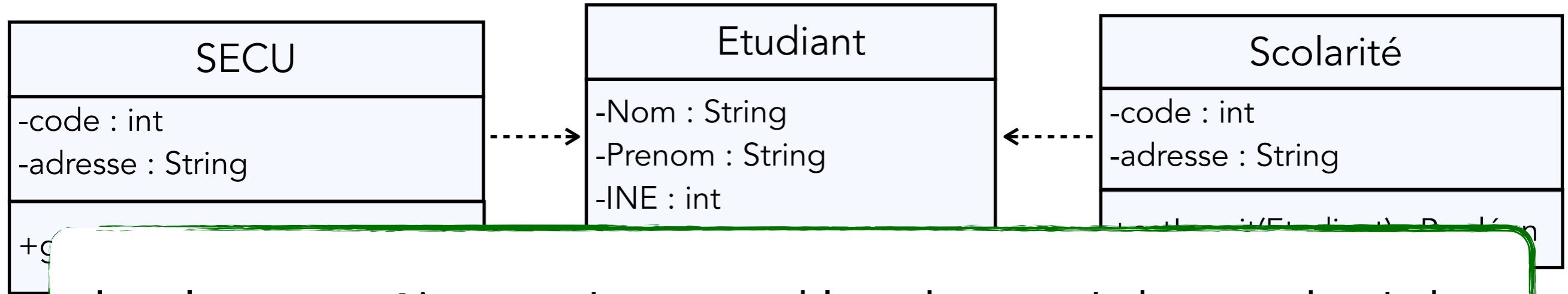
Principe : Si une classe a plus d'une responsabilité, ces dernières seront couplées. Les modifications apportées à une responsabilité impacteront les autres, augmentant la **rigidité** et la **fragilité** de la conception / du code.



Principes SOLID

Single responsibility principle

Principe : Si une classe a plus d'une responsabilité, ces dernières seront couplées. Les modifications apportées à une responsabilité impacteront les autres, augmentant la **rigidité** et la **fragilité** de la conception / du code.



La classe Etudiant est responsable uniquement de ce qui est du ressort de l'étudiant.

L'inscription est de la responsabilité de la scolarité

Le calcul des frais de sécurité sociale géré par une classe dédiée

Principes SOLID

Open close principle

Principe : Ouvert aux extensions, fermé aux modifications. Une classe doit être extensible sans être modifiée.

(Conception et programmation orientées objet, 2000, B. Meyer)

Principes SOLID

Open close principle

Principe : Ouvert aux extensions, fermé aux modifications. Une classe doit être extensible sans être modifiée.

(Conception et programmation orientées objet, 2000, B. Meyer)

```
public Shape(ShapeTypeEnum ShapeType) {  
    if (ShapeType == ShapeTypeEnum.CIRCLE) {  
        Shape = new Circle() ;  
    } else if (...) {  
        ...  
    }  
}
```



Principes SOLID

Open close principle

Principe : Ouvert aux extensions, fermé aux modifications. Une classe doit être extensible sans être modifiée.

(Conception et programmation orientées objet, 2000, B. Meyer)

```
public Shape (ShapeType shapeType) {  
    shape = ShapeFactory.getShape(shapeType) ;  
}
```



Principes SOLID

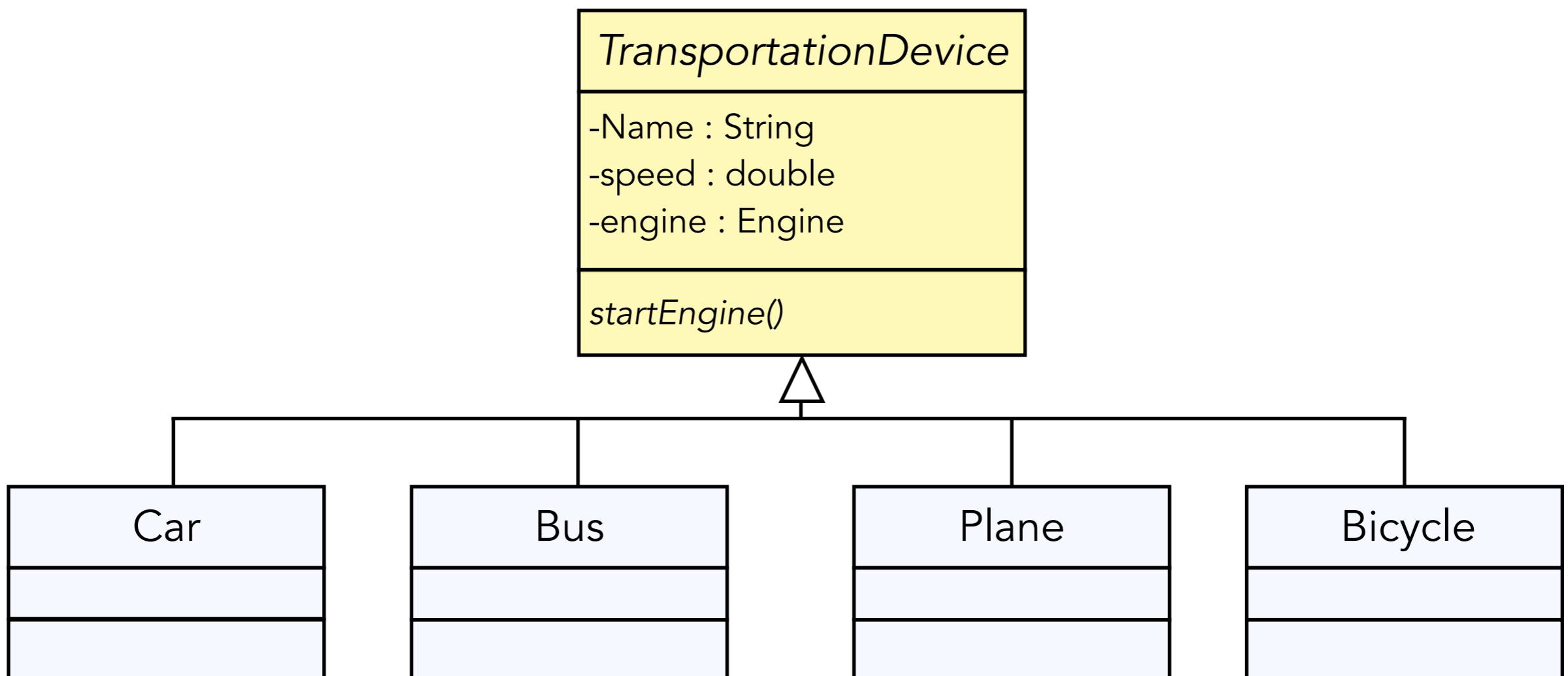
Liskov substitution principle

Principe : Les sous-classes doivent pouvoir jouer le rôle de leur super-classe sans aucun problème

Principes SOLID

Liskov substitution principle

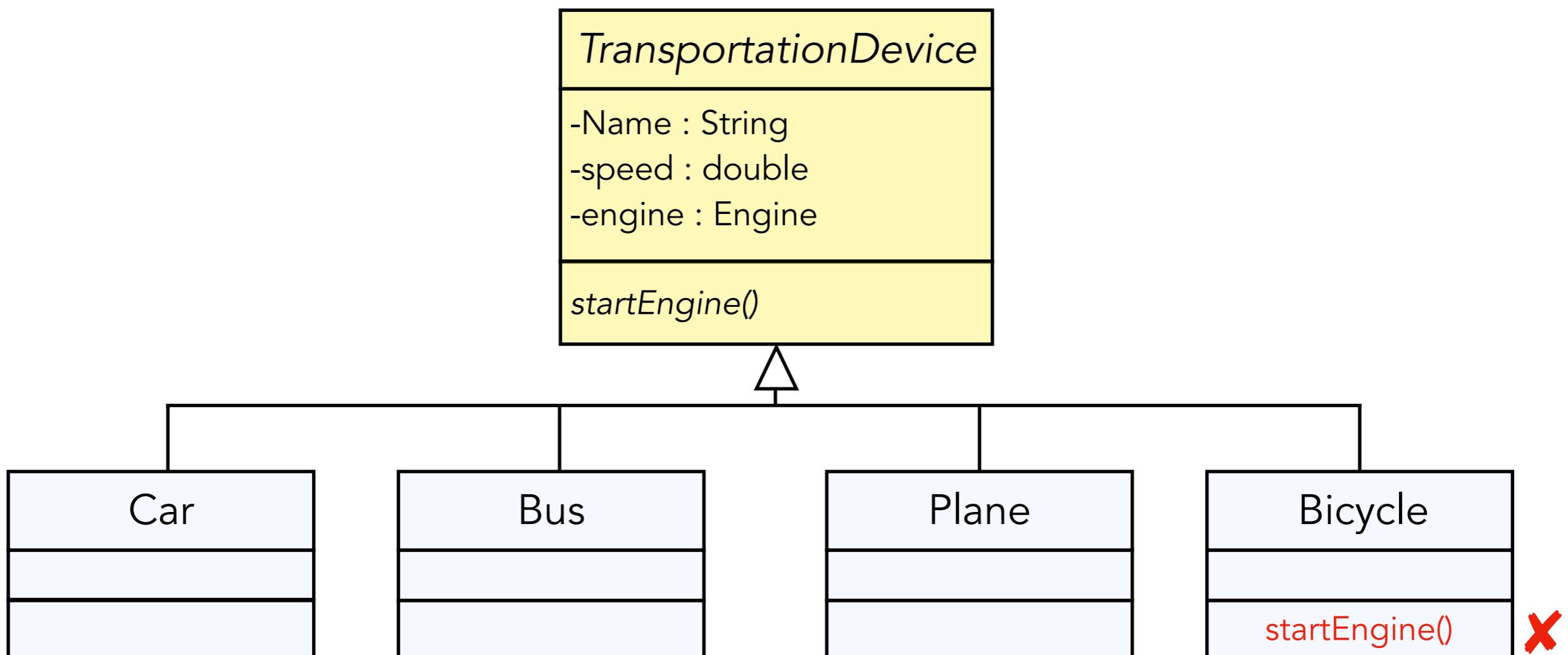
Principe : Les sous-classes doivent pouvoir jouer le rôle de leur super-classe sans aucun problème



Principes SOLID

Liskov substitution principle

Principe : Les sous-classes doivent pouvoir jouer le rôle de leur super-classe sans aucun problème



Principes SOLID

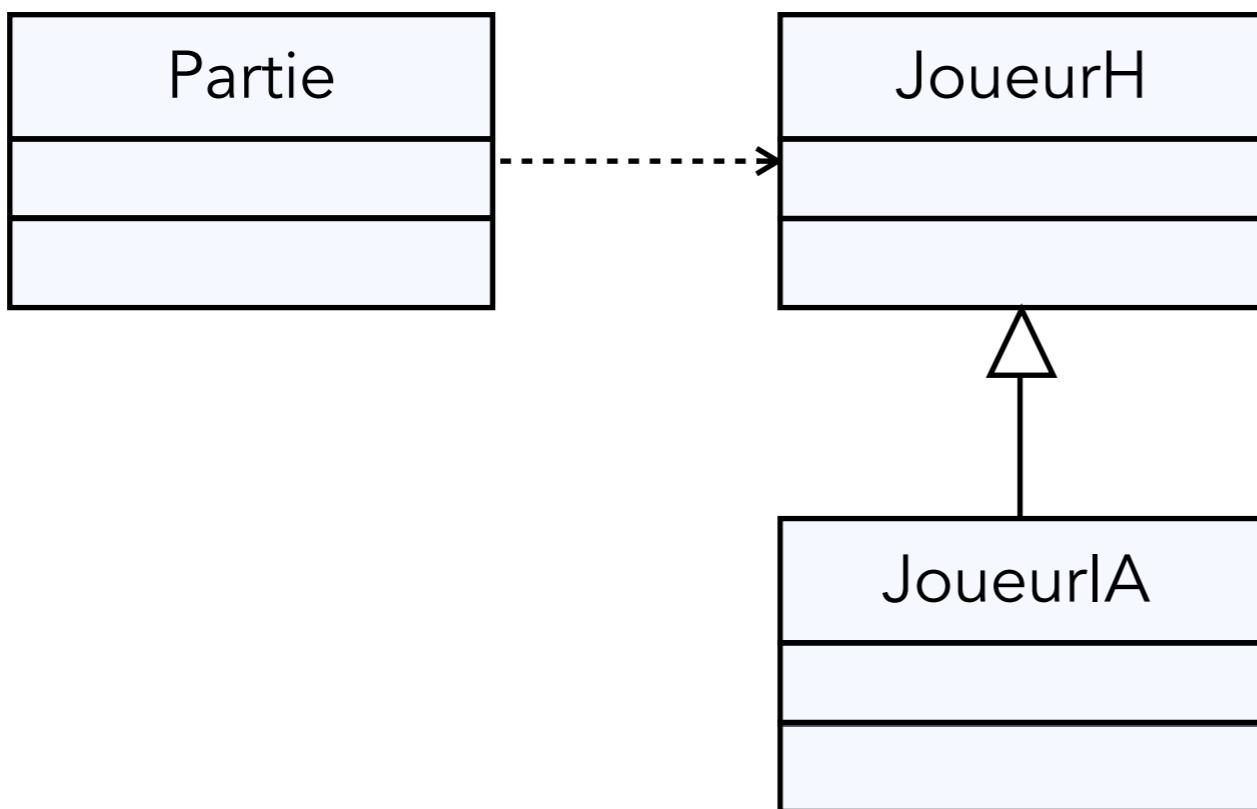
Interface segregation principle

Principe : Utiliser les interfaces pour définir les contrats

Principes SOLID

Interface segregation principle

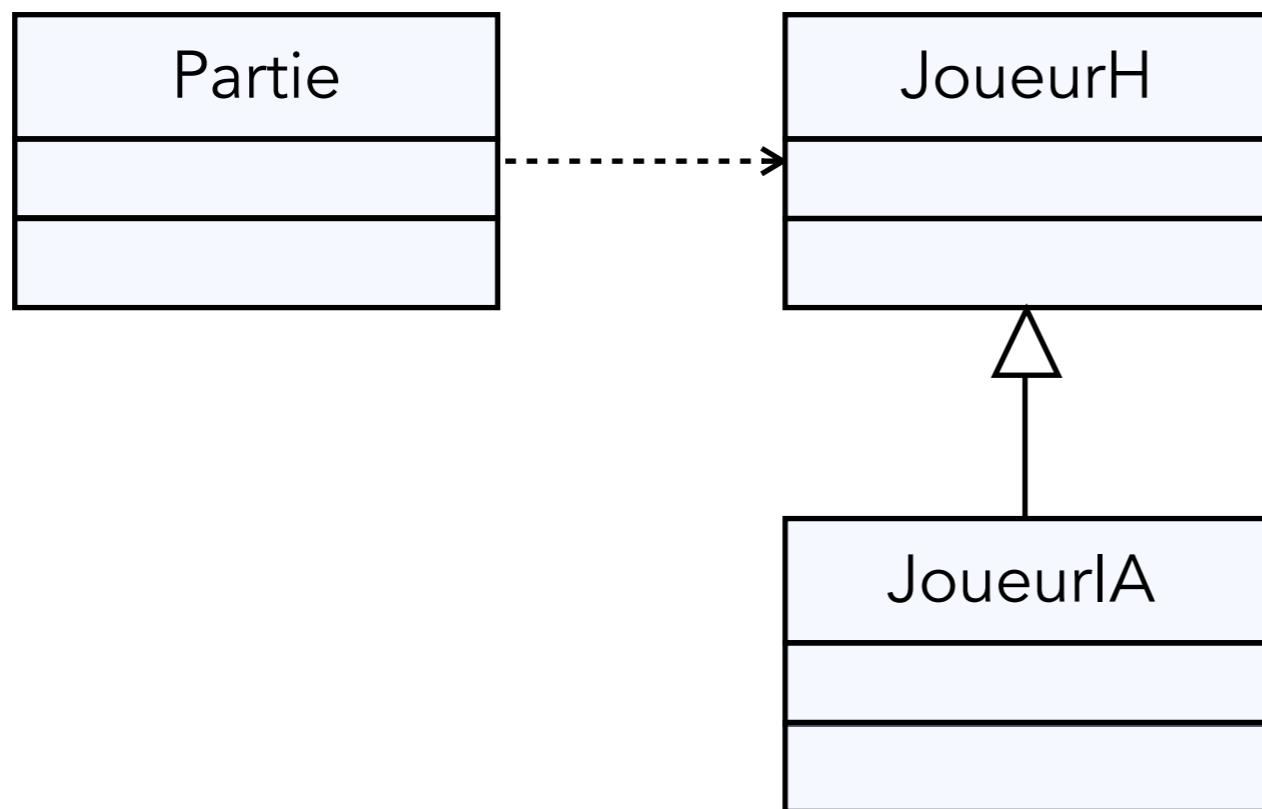
Principe : Utiliser les interfaces pour définir les contrats



Principes SOLID

Interface segregation principle

Principe : Utiliser les interfaces pour définir les contrats

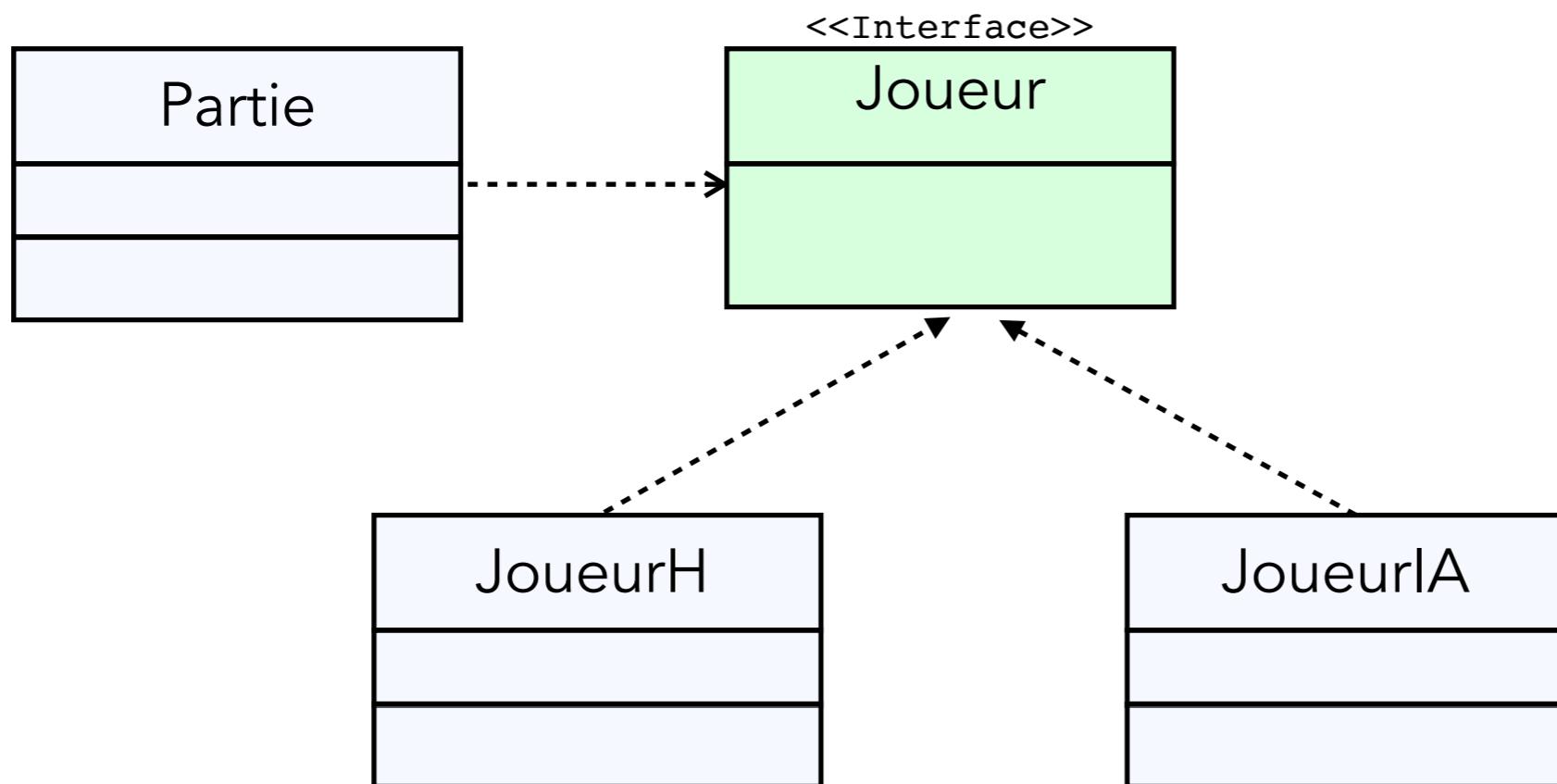


Le joueur IA hérite de tout le code du joueur humain

Principes SOLID

Interface segregation principle

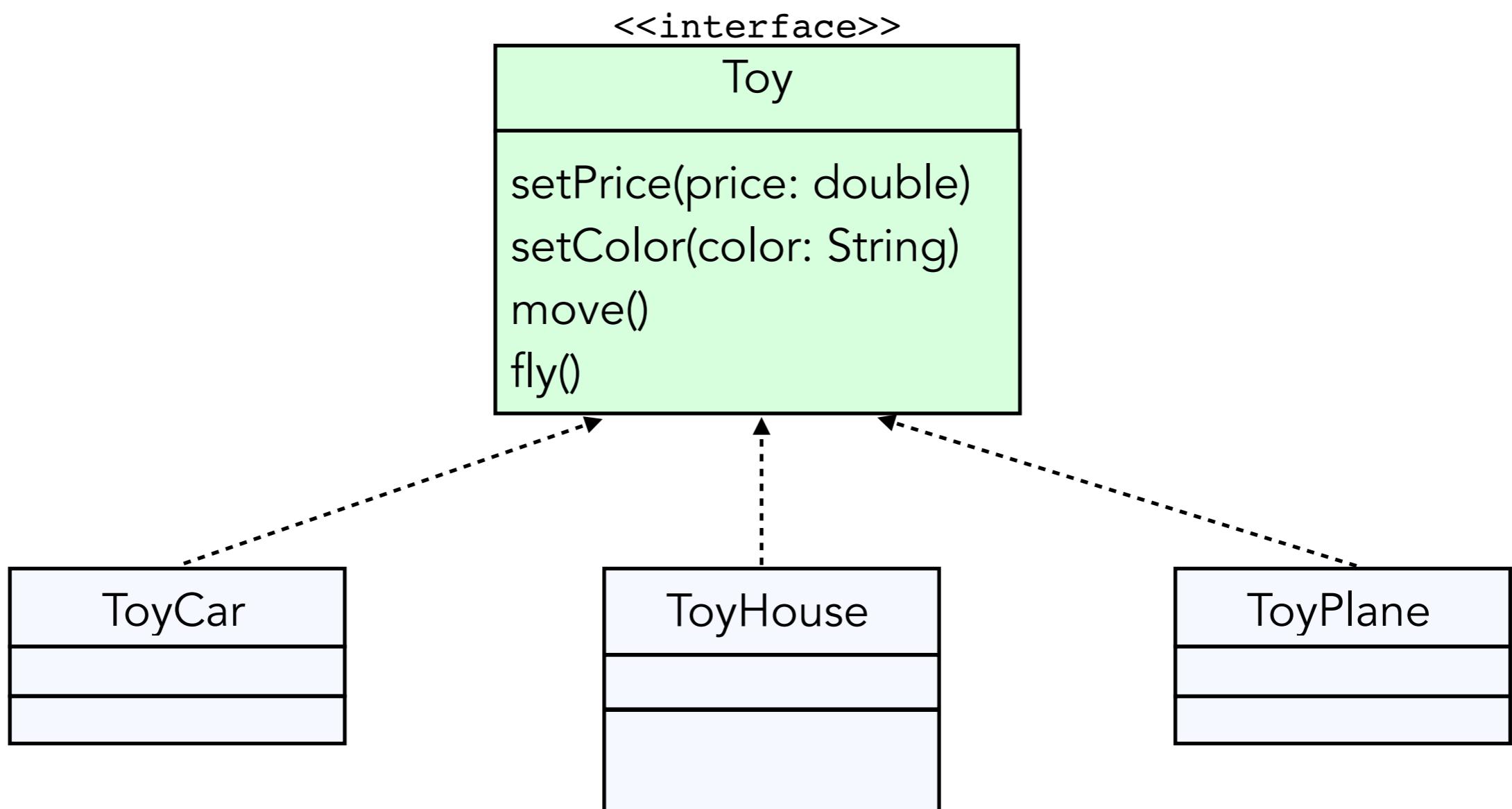
Principe : Utiliser les interfaces pour définir les contrats



Principes SOLID

Interface segregation principle

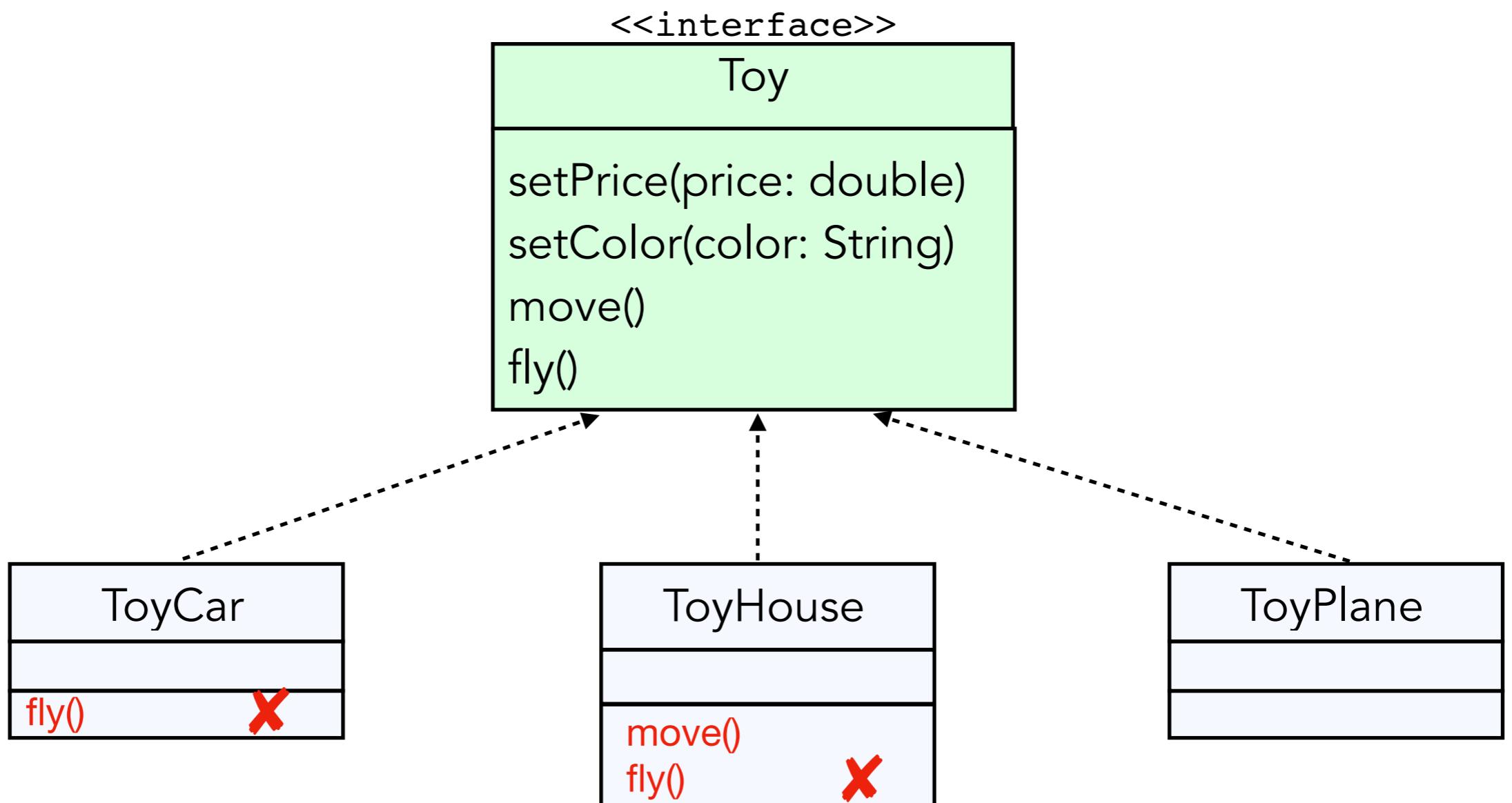
Principe : Utiliser les interfaces pour définir les contrats



Principes SOLID

Interface segregation principle

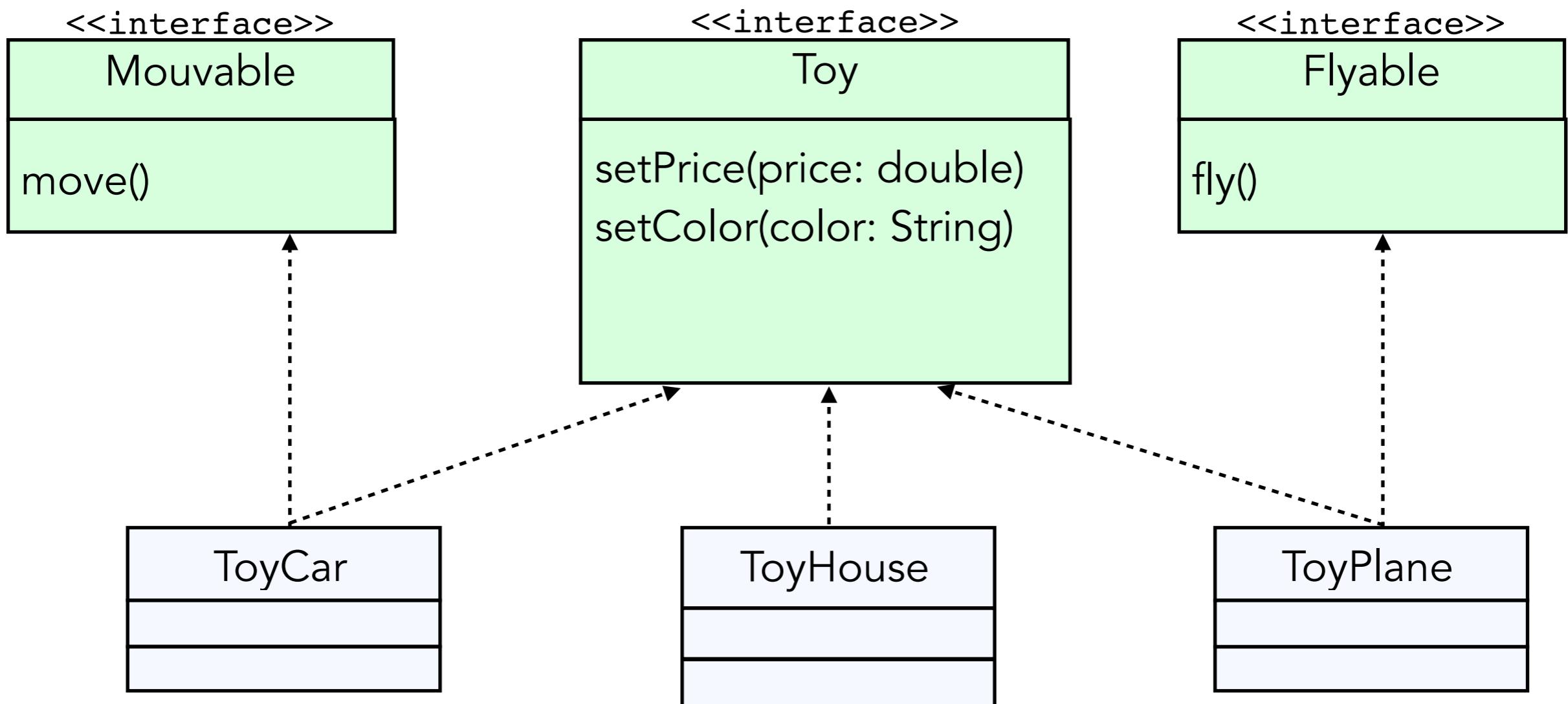
Principe : Utiliser les interfaces pour définir les contrats



Principes SOLID

Interface segregation principle

Principe : Utiliser les interfaces pour définir les contrats



Principes SOLID

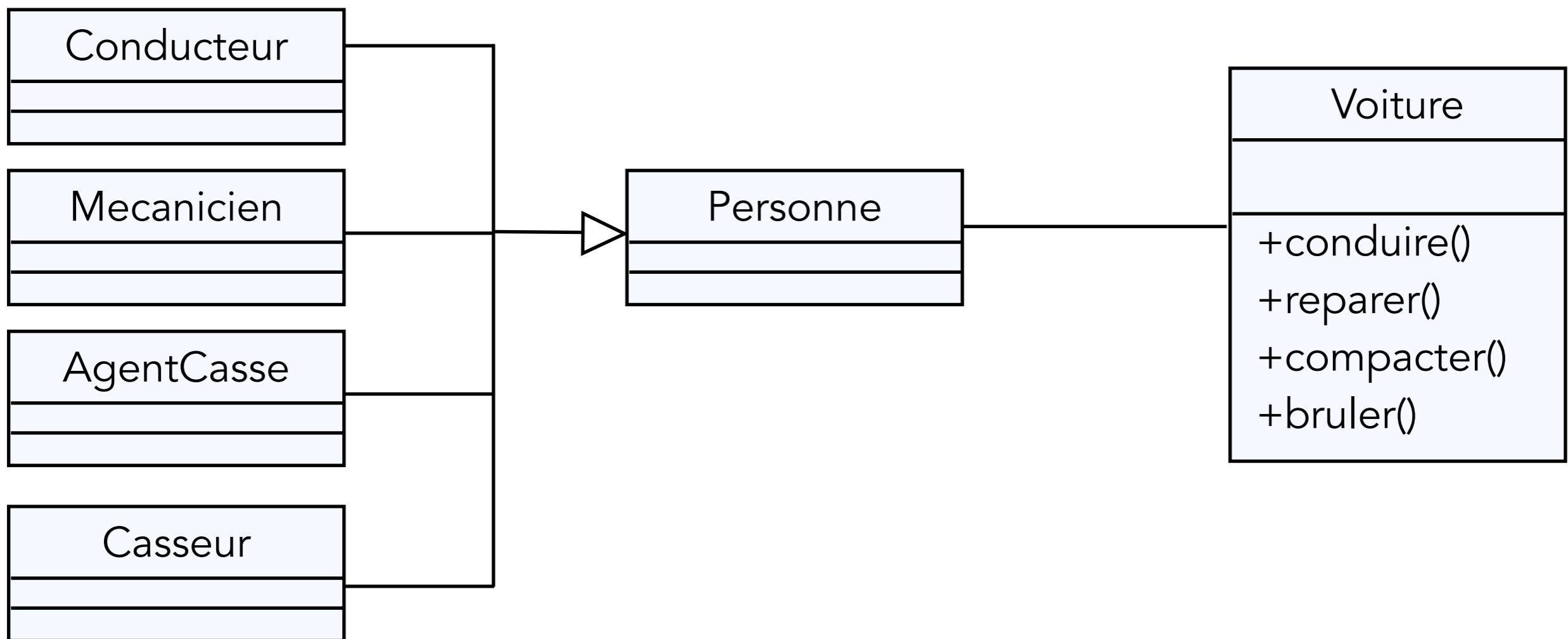
Dependency inversion principle

Principe : Non-dépendant à l'inutile => Multiplier les interfaces, plus simples, thématiquement cohérentes

Principes SOLID

Dependency inversion principle

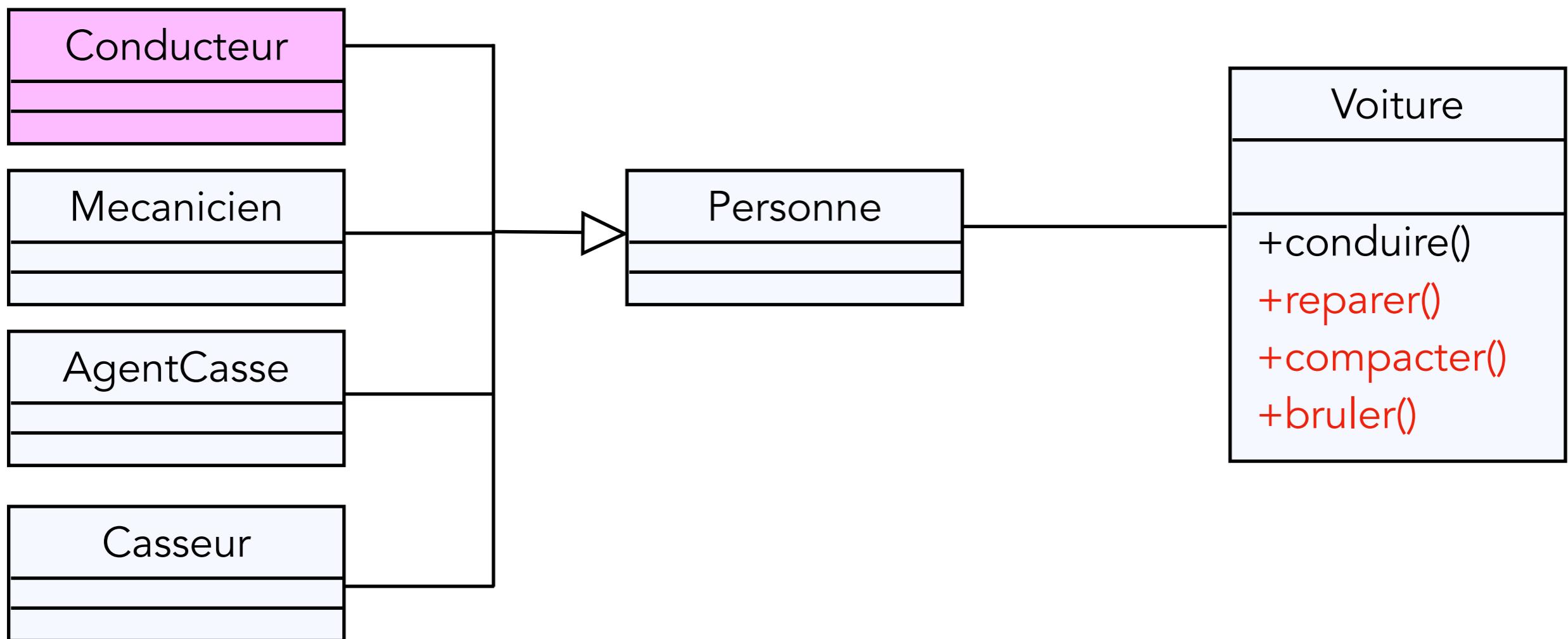
Principe : Non-dépendant à l'inutile => Multiplier les interfaces, plus simples, thématiquement cohérentes



Principes SOLID

Dependency inversion principle

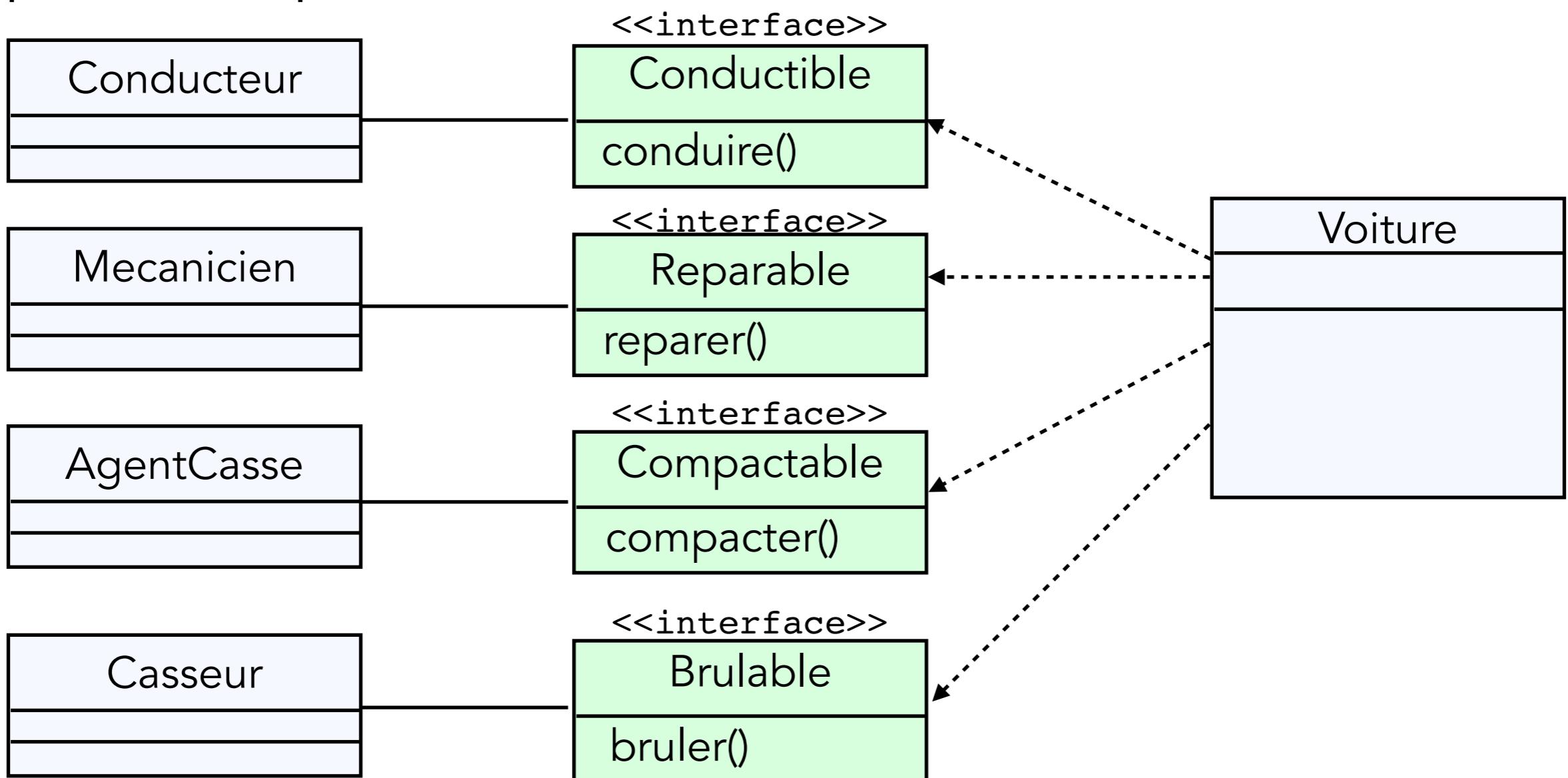
Principe : Non-dépendant à l'inutile => Multiplier les interfaces, plus simples, thématiquement cohérentes



Principes SOLID

Dependency inversion principle

Principe : Non-dépendant à l'inutile => Multiplier les interfaces, plus simples, thématiquement cohérentes



Principes SOLID

Dependency inversion principle

Principe : Non-dépendant à l'inutile => Multiplier les interfaces, plus simples, thématiquement cohérentes

```
/** Une voiture */
public class Voiture implements
    Conductible, Reparable, Compactable, Brulable{ ... }

/** Un conducteur lambda */
public class Conducteur {
    private Conductible vehicule;
    /** Crée un nouveau conducteur */
    public Conducteur(Conductible v) {
        vehicule = v;
        vehicule.conduire(); // Fait un tour d'essai
    }
}
```

Pattern

Patron ou Modèle

Un pattern décrit à la fois un **problème** qui se produit très fréquemment dans l'environnement et l'architecture de la **solution** à ce problème de telle façon que l'on puisse **utiliser** cette solution des milliers de fois sans jamais **l'adapter** deux fois de la même manière.

C. Alexander 1977

-

Pattern

Patron ou Modèle

Un pattern décrit à la fois un **problème** qui se produit très fréquemment dans l'environnement et l'architecture de la **solution** à ce problème de telle façon que l'on puisse **utiliser** cette solution des milliers de fois sans jamais **l'adapter** deux fois de la même manière.

C. Alexander 1977

- Pattern Language : Towns, Buildings Construction (Alexander, Ishikouwa, et Silverstein 1977)

Pattern

Patron ou Modèle

- **Coad [Coad92]** – Une abstraction d'un doublet, triplet ou d'un ensemble de classes qui peut être réutilisé encore et encore pour le développement d'applications
- **Appleton[Appleton97]** – Une règle tripartite exprimant une relation entre un certain contexte, un certain problème qui apparaît répétitivement dans ce contexte et une certaine configuration logicielle qui permet la résolution de ce problème
- **Aarsten [Aarsten96]** – Un groupe d'objets coopérants liés par des relations et des règles qui expriment les liens entre un contexte, un problème de conception et sa solution

Pattern

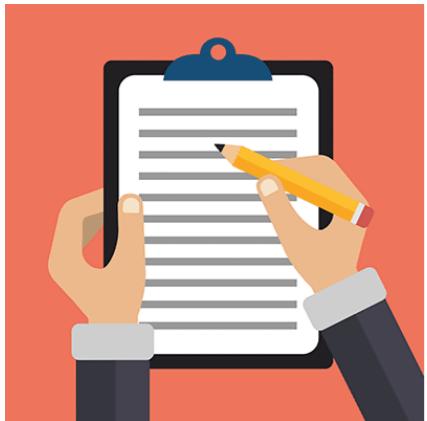
Patron ou Modèle

- **Coad [Coad92]** – Une abstraction d'un doublet, triplet ou d'un ensemble de classes qui peut être réutilisé encore et encore pour le développement d'applications
- **Appleton[Appleton97]** – Une règle tripartite exprimant une relation entre un certain contexte, un certain problème qui apparaît répétitivement dans ce contexte et une certaine configuration logicielle qui permet la résolution de ce problème
- **Aarsten [Aarsten96]** – Un groupe d'objets coopérants liés par des relations et des règles qui expriment les liens entre un contexte, un problème de conception et sa solution

Pattern : Une solution standard, utilisable dans la conception de différents logiciels

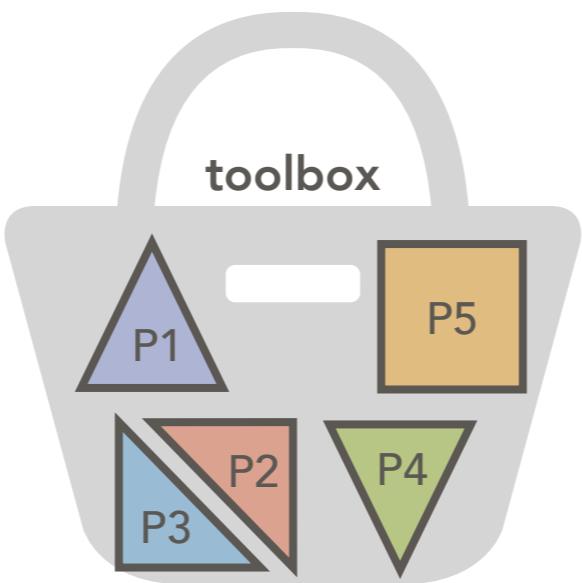
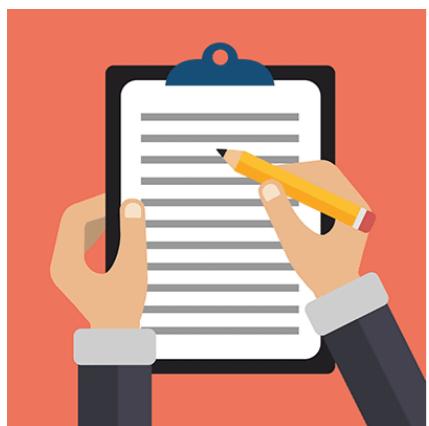
Design Patterns

Modèles ou Patrons de Conception



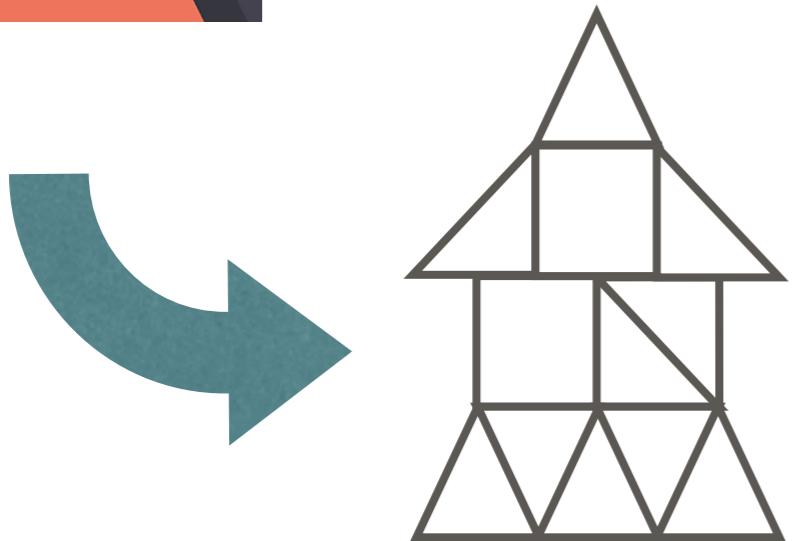
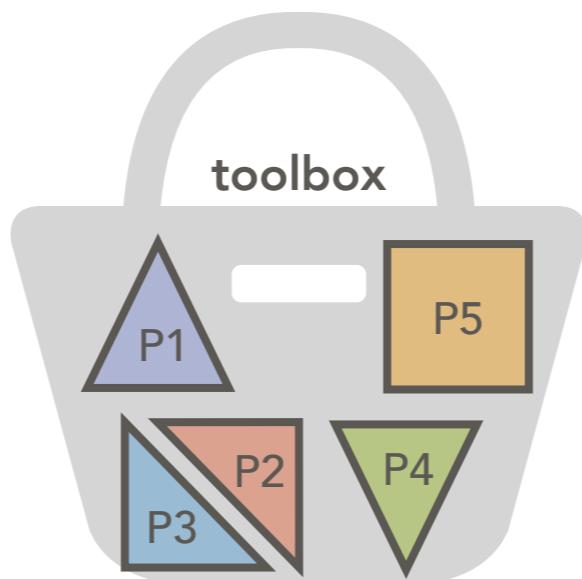
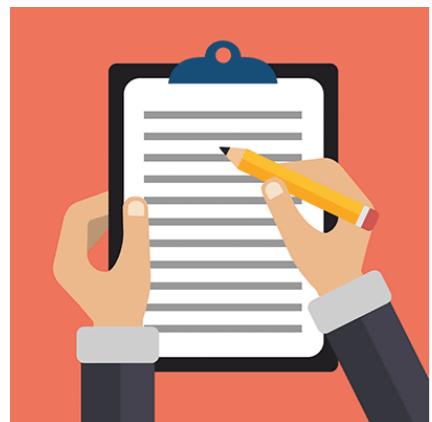
Design Patterns

Modèles ou Patrons de Conception



Design Patterns

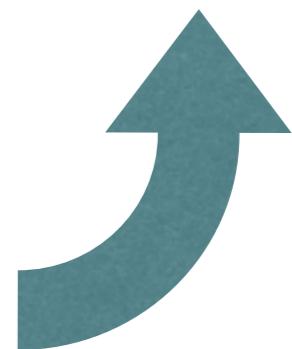
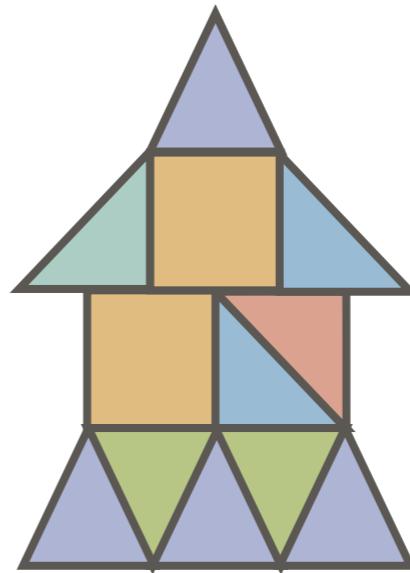
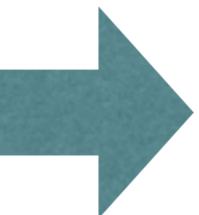
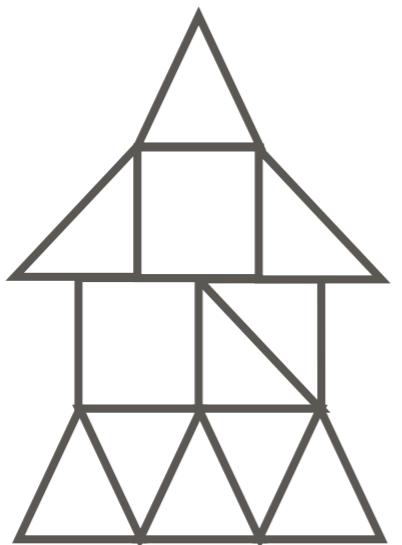
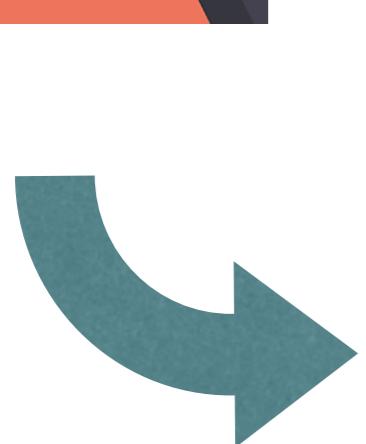
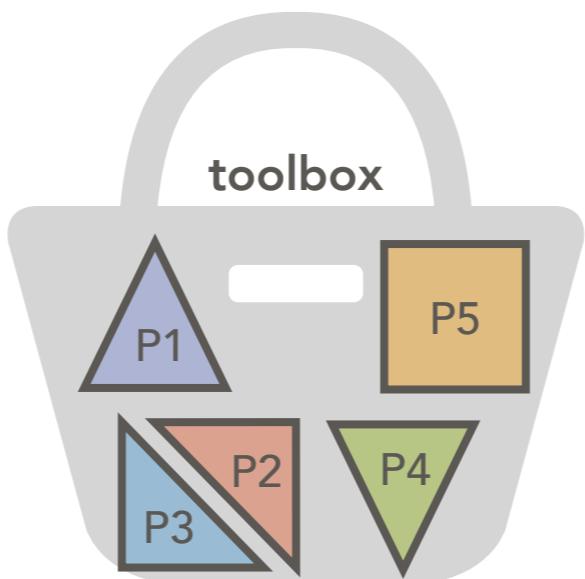
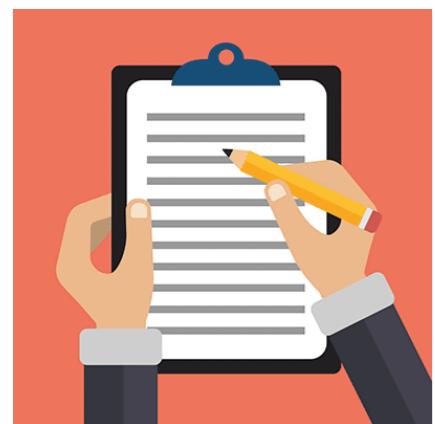
Modèles ou Patrons de Conception



Règles de découpage

Design Patterns

Modèles ou Patrons de Conception



Patterns Categories

- **Patterns de construction** : Ces patterns sont très courants pour déléguer à d'autres classes la construction des objets
- **Patterns de structure** : Ces patterns tendent à concevoir des agglomérations de classes avec des macro-composants
- **Patterns de comportement** : Ces patterns tentent de répartir les responsabilités entre chaque classe (l'usage est plutôt dynamique)

Patterns Categories

- **Patterns de construction** : Ces patterns sont très courants pour déléguer à d'autres classes la construction des objets
- **Patterns de structure** : Ces patterns tendent à concevoir des agglomérations de classes avec des macro-composants
- **Patterns de comportement** : Ces patterns tentent de répartir les responsabilités entre chaque classe (l'usage est plutôt dynamique)

Creational Patterns

Les Patrons de Constructions

Creational Patterns

- Factory Pattern / Le Patron de Fabrique
- Builder Pattern / Le Patron Monteur
- Singleton Pattern / Le Patron Singleton

Creational Patterns

Factory Pattern / Patron de Fabrique

Isoler la création des objets

Découpler les classes concrètes de leur utilisateur

Définition d'un constructeur "virtuel"

Principe SOLID

- **Open/Closed:** Ouvert aux extensions, fermé aux modifications
- **Dependency Inversion:** Non-dépendant à l'inutile

Factory Pattern

Cas1 : Concessionnaire

```
/** Un vendeur de voitures */
public class Concessionnaire {

    /** Les marques vendues par le concessionnaire */
    public static enum Marque { Mercedes, Volkswagen, BMW};

    /** Commande la voiture demandée.
     * @param type la marque du véhicule à commander
     * @return la voiture demandée. */
    public Voiture Commande(Marque type){
        Voiture v ;
        if (type instanceof Mercedes) v = new ClasseA() ;
        else if (type instanceof Volkswagen) v = new Polo() ;
        else if (type instanceof BMW) v = new Serie1() ;
        v.immatriculer() ;
        return v ;
    }
}
```

Factory Pattern

Cas1 : Concessionnaire

```
/** Un vendeur de voitures */
public class Concessionnaire {

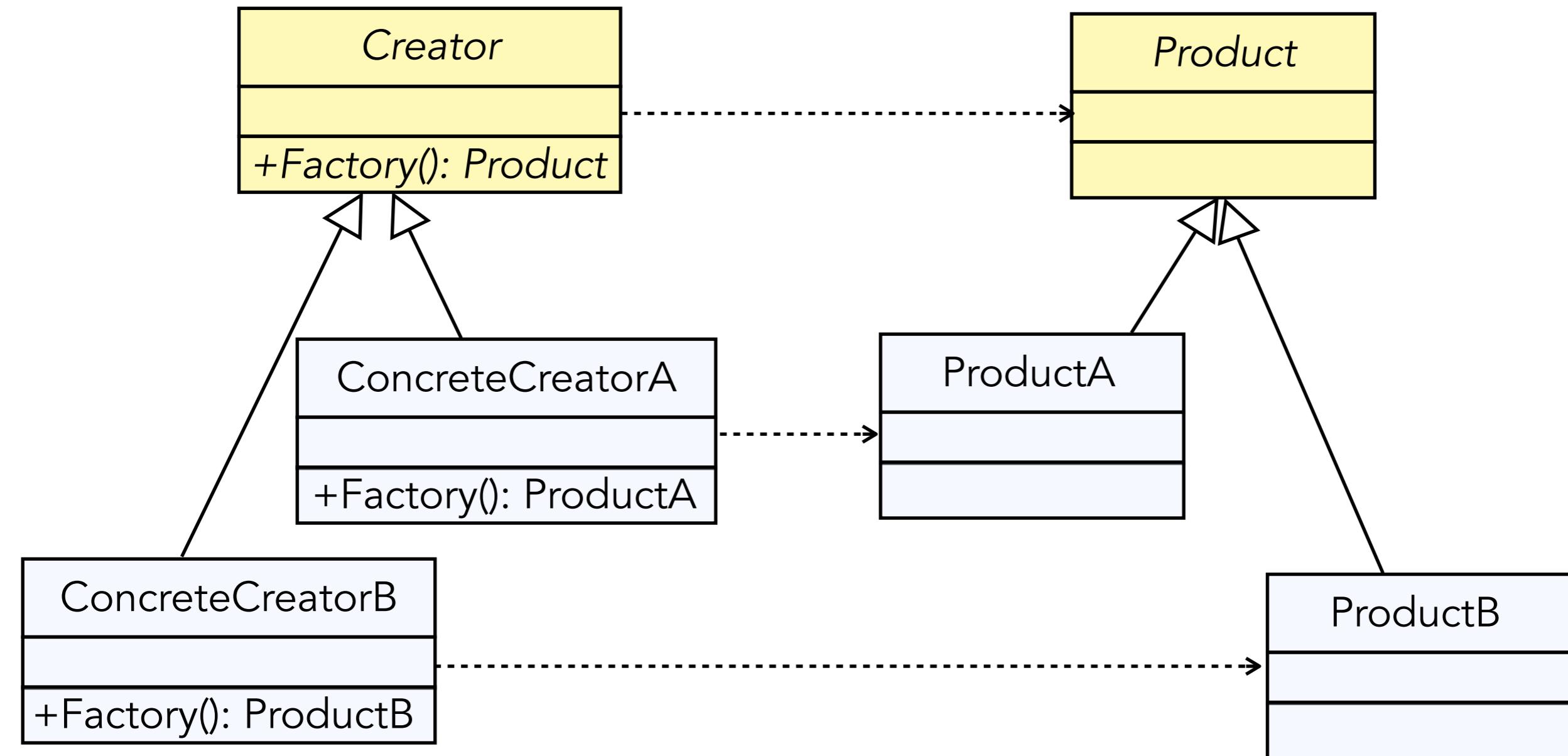
    /** Les marques vendues par le concessionnaire */
    public static enum Marque { Mercedes, Volkswagen, BMW};

    /** Commande la voiture demandée.
     * @param type la marque du véhicule à commander
     * @return la voiture demandée. */
    public Voiture Commande(Marque type){
        Voiture v ;
        if (type instanceof Mercedes) v = new ClasseA() ;
        else if (type instanceof Volkswagen) v = new Polo() ;
        else if (type instanceof BMW) v = new Serie1() ;
        v.immatriculer() ;
        return v ;
    }
}
```

Ajout de véhicule Audi =>
Violation du principe Open/closed

Cas1 : Concessionnaire

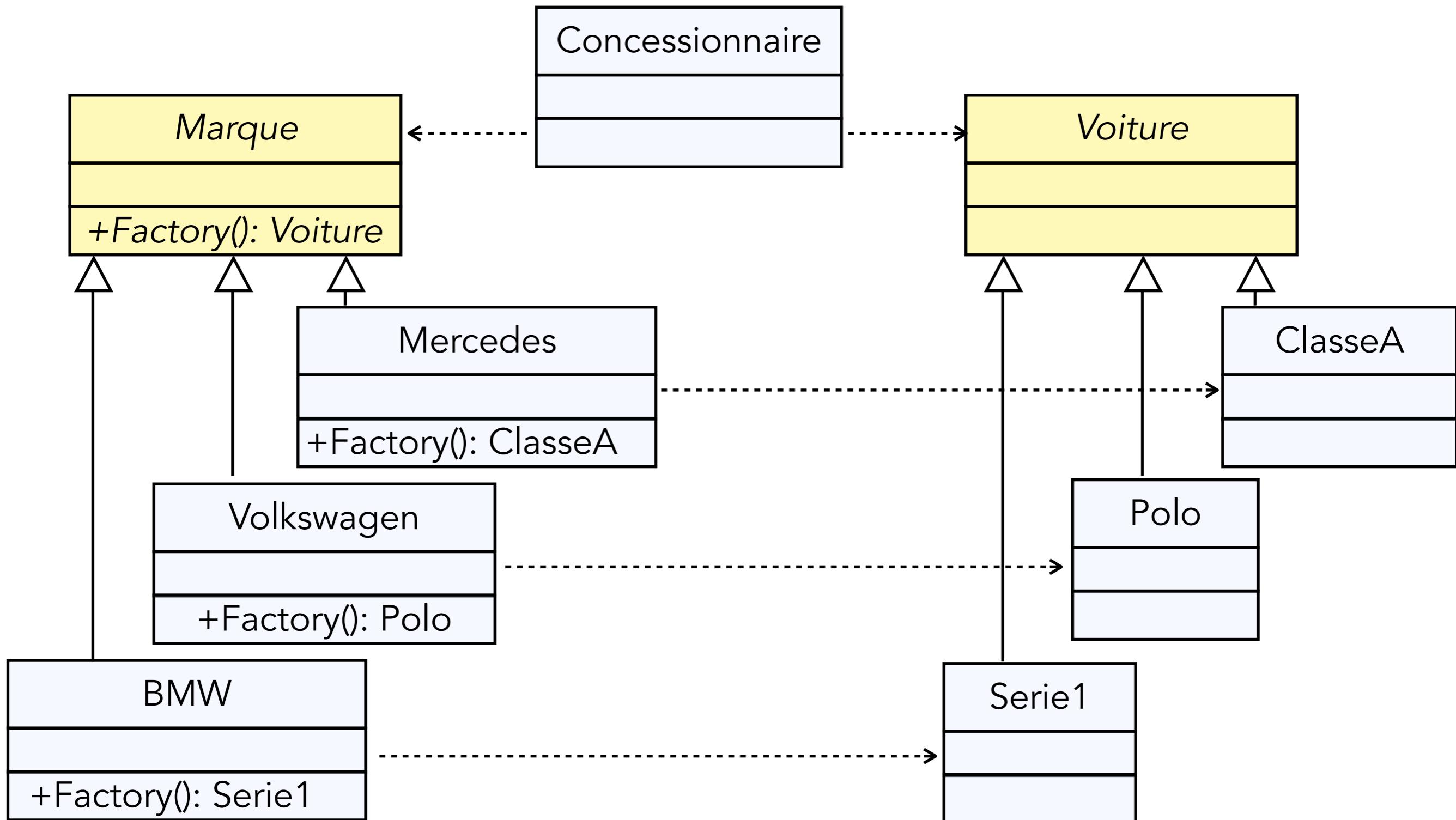
Pattern to use



Un créateur => Un type d'objet

Cas1 : Concessionnaire

UML



Cas1 : Concessionnaire

JAVA

```
/** Les marques vendues par le concessionnaire */
public class Marque {
    /** Fabrique une voiture de cette marque */
    public abstract Voiture fabrique();
};

public class Mercedes extends Marque {
    @Override
    public Voiture fabrique() { return new ClasseA(); }
};

public class Volkswagen extends Marque {
    @Override
    public Voiture fabrique() { return new Polo(); }
};

public class BMW extends Marque {
    @Override
    public Voiture fabrique() { return new Serie1(); }
};
```

Cas1 : Concessionnaire

JAVA

```
/** Les
public c
    /**
     publi
};

public c
    @over
    publi
};

public class Volkswagen extends Marque {
    @override
    public Voiture fabrique() { return new Polo(); }
};

public class BMW extends Marque {
    @override
    public Voiture fabrique() { return new Serie1(); }
};

    /**
     Un vendeur de voitures */
public class Concessionnaire {
    /**
     Commande la voiture demandée.
     @param type la marque du véhicule à commander
     @return la voiture demandée. */
    public Voiture Commande(Marque type){
        Voiture v = type.fabrique();
        v.immatriculer();
        return v;
    }

    /**
     La concessionnaire
     @param type la marque à commander
     @return la voiture demandée. */
    public Voiture Commande(String type) {
        Marque m = null;
        if (type.equals("VW"))
            m = new Volkswagen();
        else
            m = new BMW();
        return m.Commande(type);
    }
}
```

Cas1 : Concessionnaire

JAVA

```
/** Les
public c
    /**
     publi
};

public c
    @over
    publi
};

public class Volkswagen extends Marque {
    @override
    public Voiture fabrique() { return new Polo(); }

    public class Citroen extends Marque {
        @Override
        public Voiture fabrique() { return new Spacetourer(); }
    }
},
```

Factory Pattern

Cas2 : Pizzeria

```
/** Une pizzeria */
public class PizzaStore{

    /** Prépare la pizza dont le type est passé en argument.
     * @return la pizza préparée. */
    public PizzaStore getPizza(String type){

        PizzaStore pizza ;

        if (type.equals("cheese")) pizza = new CheesePizza() ;

        else if (type.equals("pepperoni")) pizza = new PepperoniPizza() ;

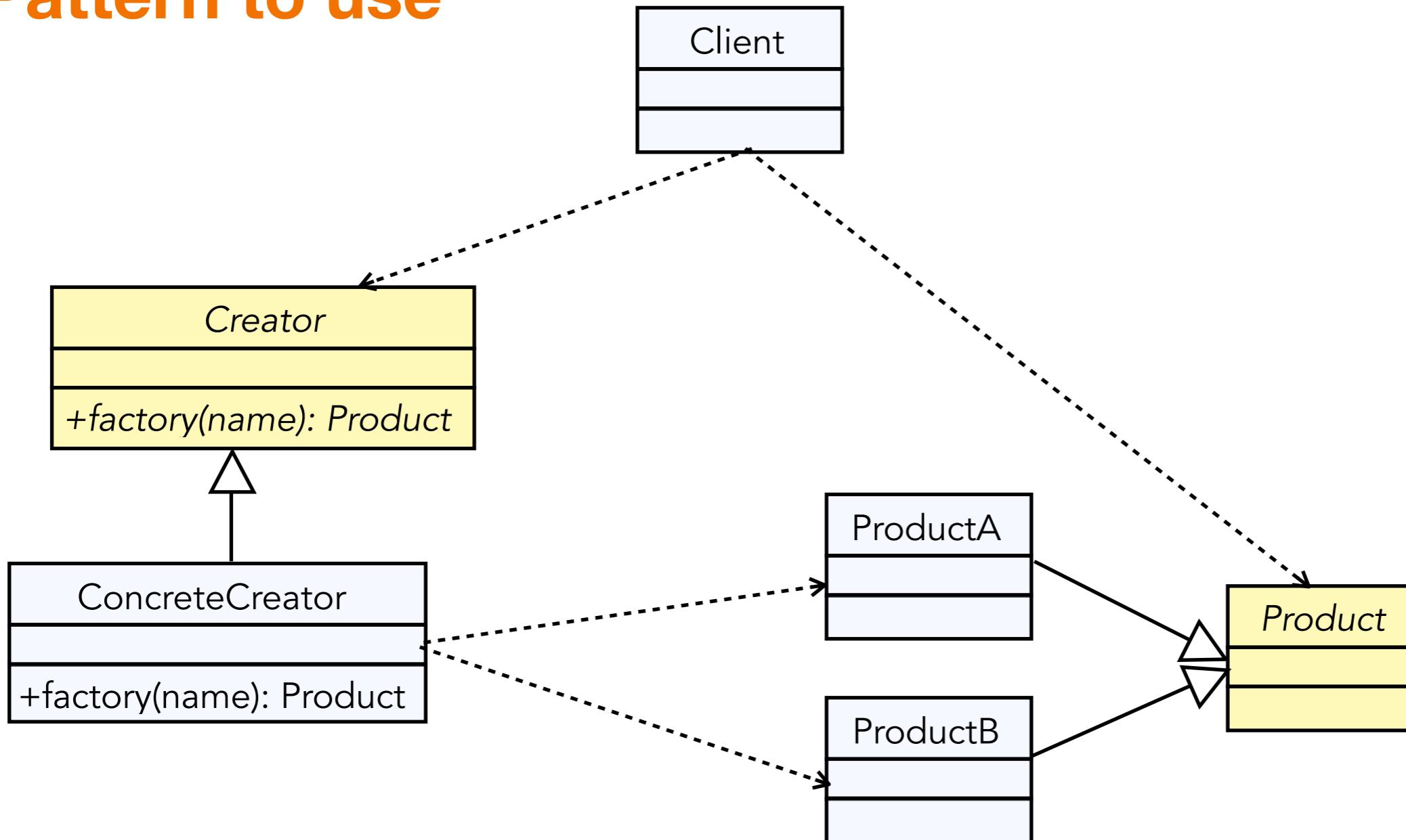
        else if (type.equals("clam")) pizza = new ClamPizza() ;

        pizza.prepare() ;
        pizza.bake() ;
        pizza.cut() ;
        pizza.box() ;

        return pizza ;
    }
}
```

Cas2 : Pizzeria

Pattern to use



Un créateur => différents types de produits

Cas2 : Pizzeria

JAVA

```
/** La fabrique à pizzas */
public class PizzaFactory{

    /** Crée la pizza dont le type est passé en argument. */
    public static Pizza getPizza(String type){

        Pizza pizza ;

        if (type.equals("cheese")) pizza = new CheesePizza() ;
        else if (type.equals("pepperoni")) pizza = new PepperoniPizza() ;
        else if (type.equals("clam")) pizza = new ClamPizza() ;
        return pizza ;
    }

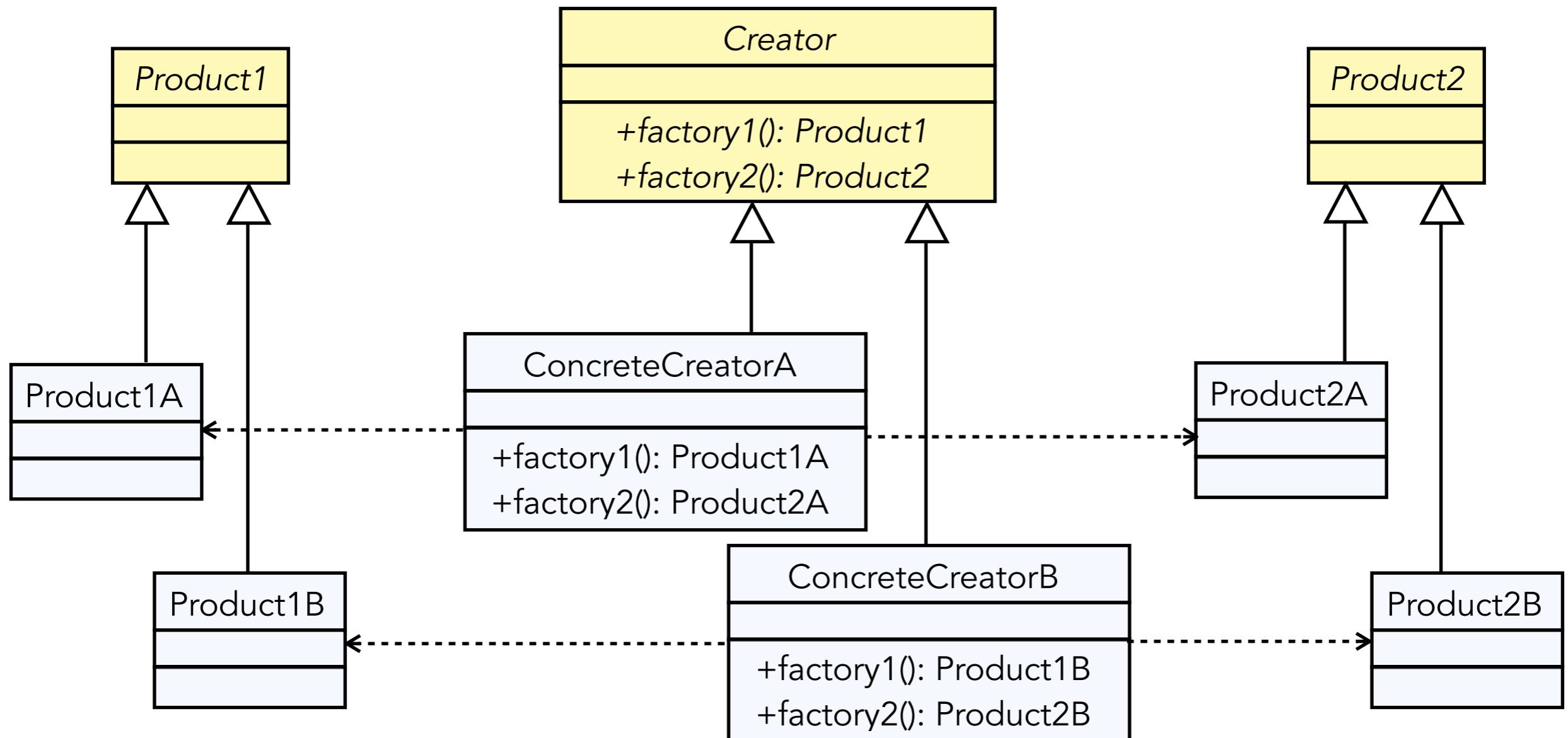
    /** Une pizzeria */
    public class PizzaStore{

        /** Prépare la pizza dont le type est passé en argument. */
        public Pizza getPizza(String type){
            Pizza pizza = PizzaFactory.createPizza(type);
            pizza.prepare() ;
            pizza.bake() ;
            pizza.cut() ;
            pizza.box() ;

            return pizza ;
        }
    }
}
```

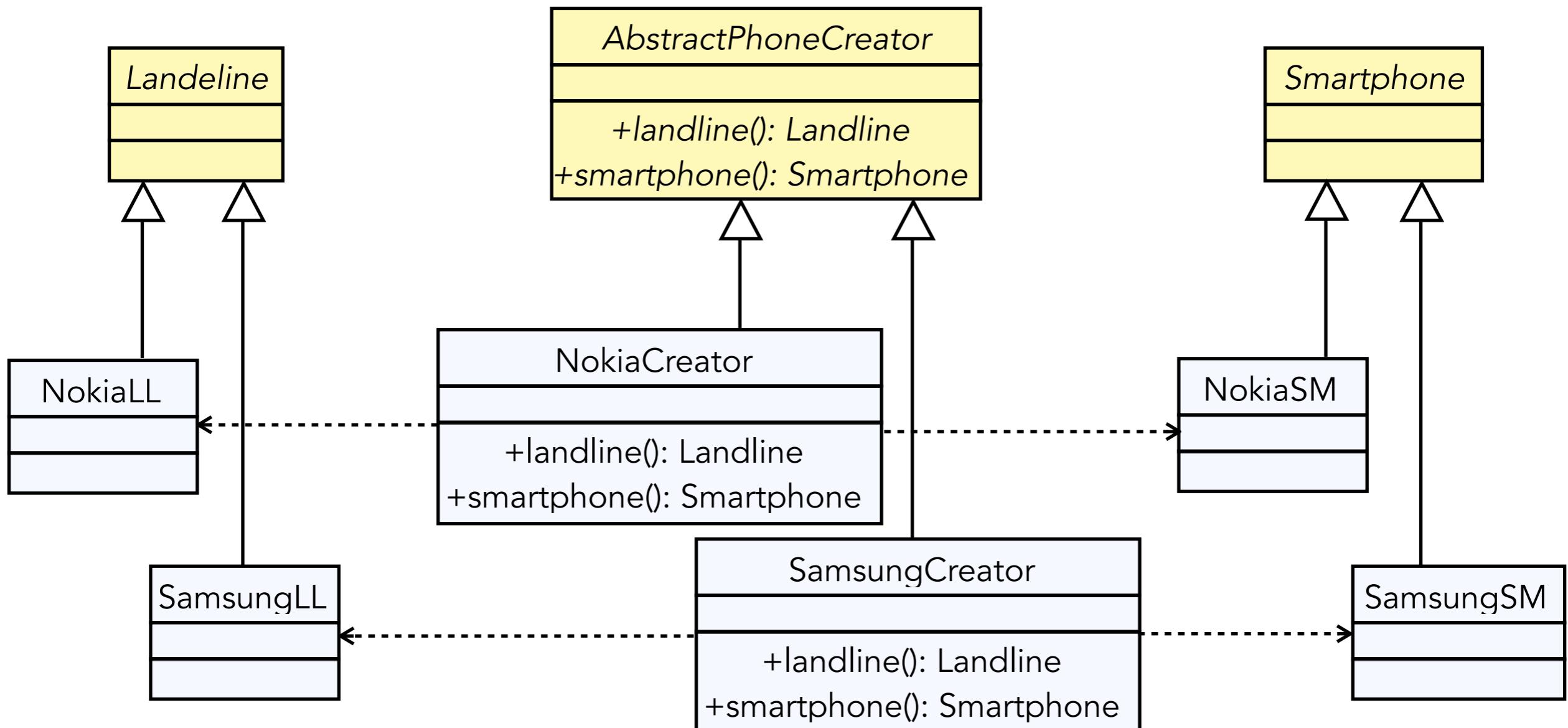
Creational Patterns

Abstract Factory Pattern / Patron Fabrique Abstraite



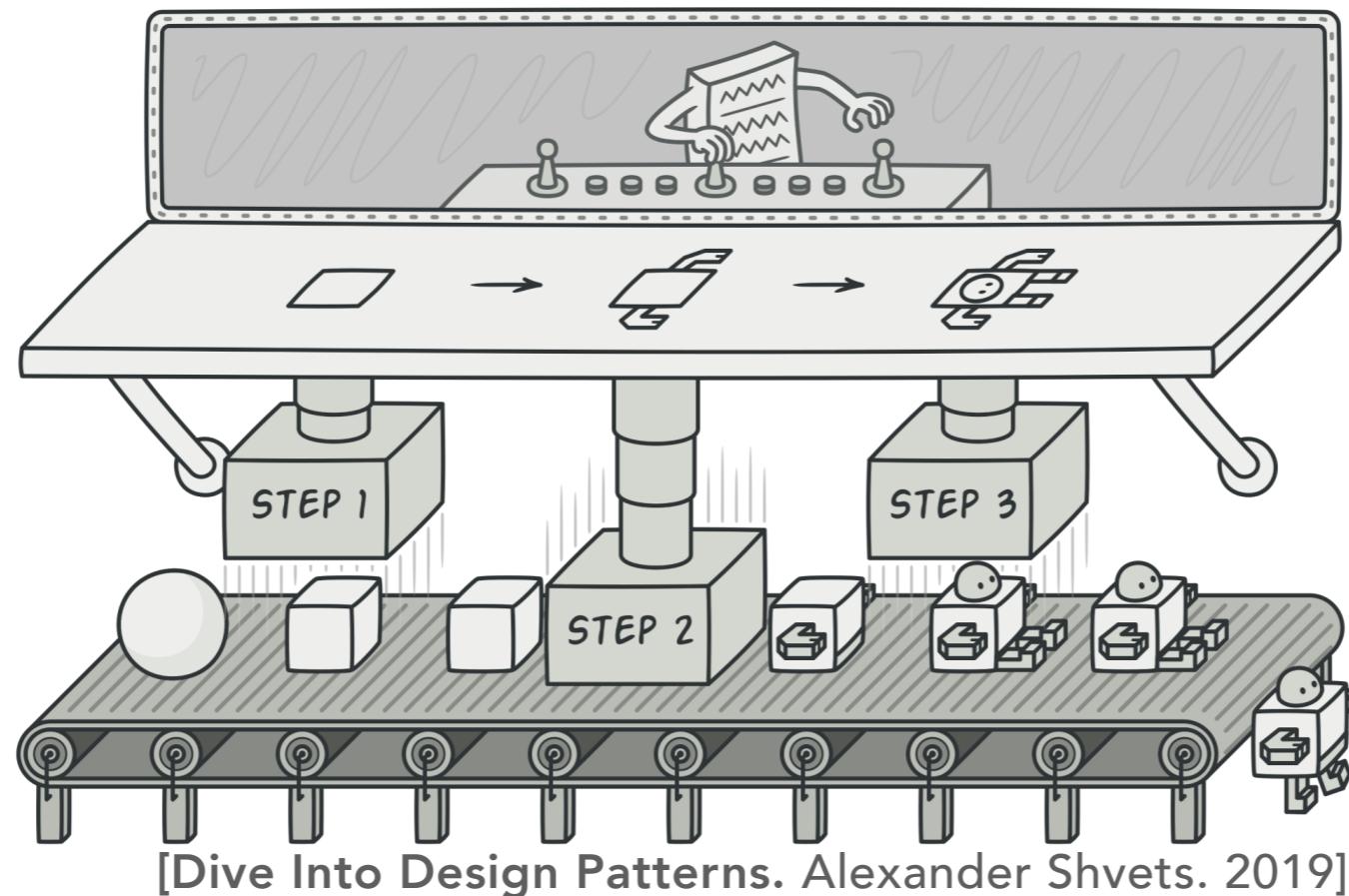
Abstract Factory Pattern

Exemple



Builder Pattern

Le Patron Monteur



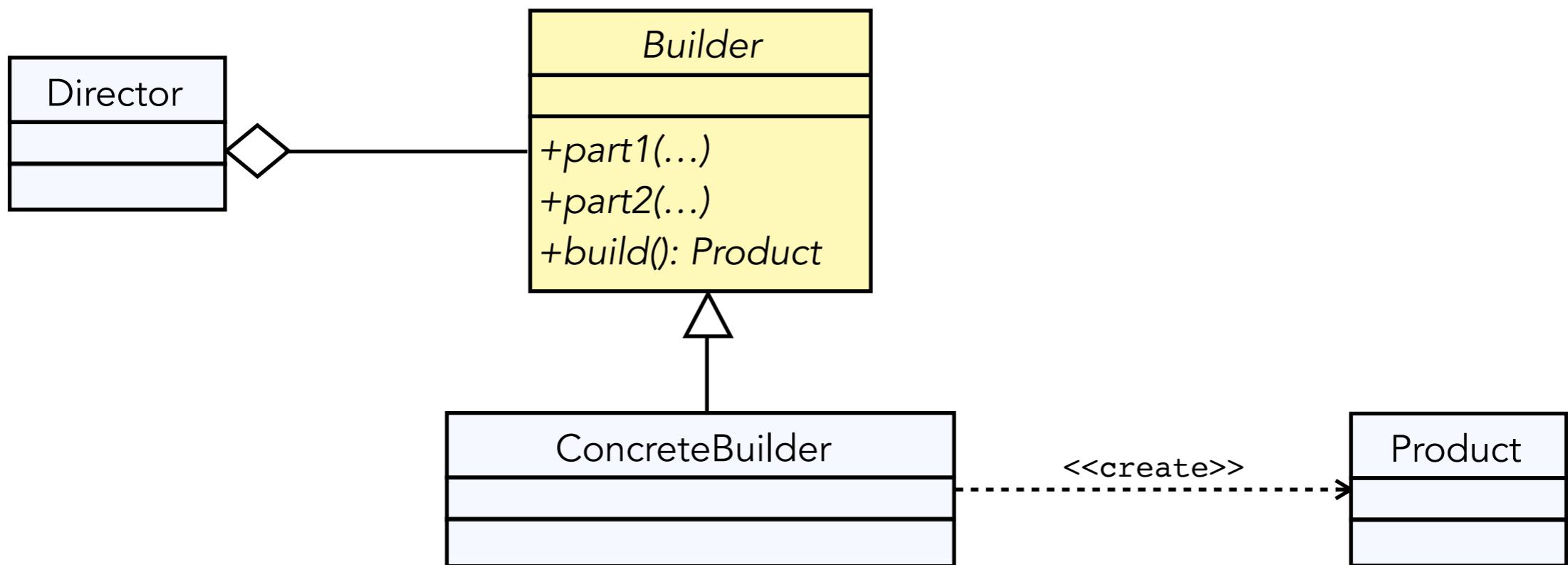
Builder Pattern

Le Patron Monteur

- Rendre facile la construction d'un objet complexe, avec beaucoup d'arguments dont certains sont optionnels.
- Encapsuler la façon dont un objet complexe est construit.
- Permet aux objets d'être construits en plusieurs et différentes étapes (contrairement à la Factory).
- Le client n'a pas accès à la représentation interne du produit.
- Souvent utilisé pour la construction des composites.
- La construction d'objets nécessite plus de connaissances comparant à la Factory.

Builder Pattern

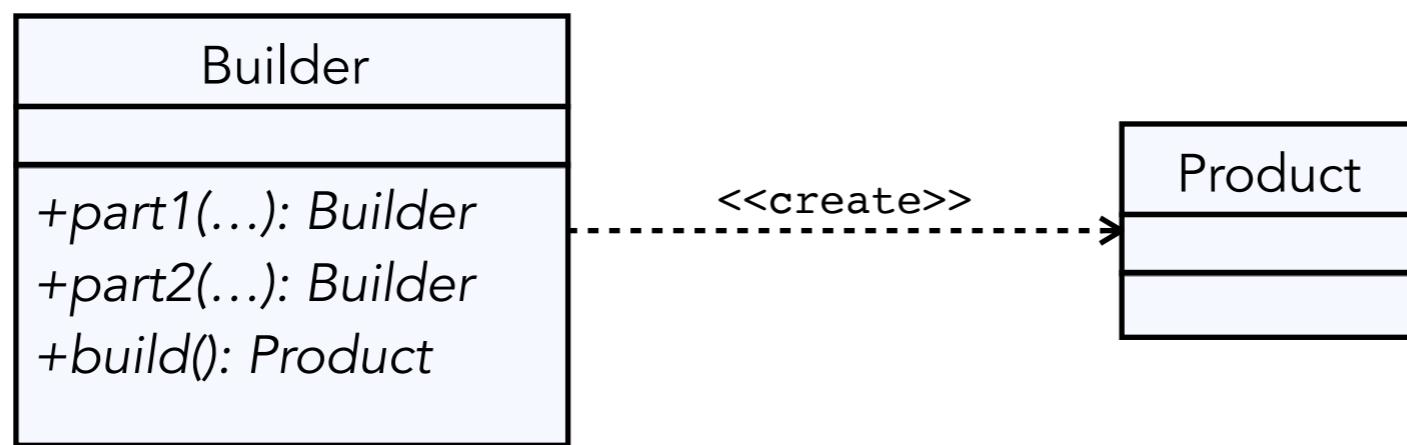
Le Patron Monteur



Fluent Builder Pattern

Le Patron Monteur Fluent

Une variante Fluent sans Directeur ni classe abstraite est beaucoup plus utilisée.



Fluent Builder Pattern

Le Patron Monteur Fluente

```
Pizza custom = new Pizza.Builder(Size.Large)
    .withMeat(Meat.Chicken)
    .withVegetable(Vegetable.Tomato)
    .withOlives(Olive.Black)
    .withCheese(Cheese.Mozzarella)
    .withCheese(Cheese.Mozzarella) // extra topping!
    .bake();
```

Singleton Pattern

Le Patron Singleton

- Pattern qui permet l'unicité d'un objet
- Assure qu'une classe ne peut avoir qu'un seul objet
- Assure un accès global à une unique et même instance

Singleton Pattern

Le Patron Singleton

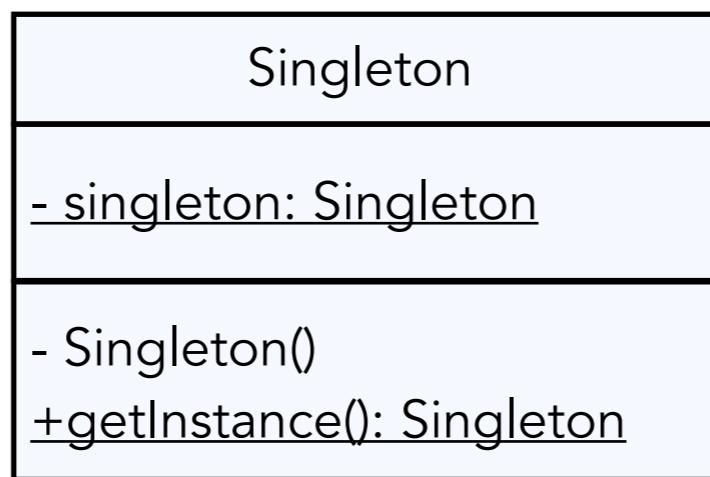
- Pattern qui permet l'unicité d'un objet
- Assure qu'une classe ne peut avoir qu'un seul objet
- Assure un accès global à une unique et même instance

Comment ?

- Pour empêcher d'autres objets d'utiliser le constructeur de la classe Singleton, il faut rendre le constructeur de la classe par défaut privé.
- Ajouter une méthode statique qui agit comme un constructeur. Cette méthode doit appeler le constructeur privé pour créer un objet et le mettre dans un attribut statique.

Singleton Pattern

Le Patron Singleton (UML)



Singleton Pattern

Le Patron Singleton (JAVA)

```
// Java program implementing Singleton class
// with getInstance() method
class Singleton
{
    // static variable single_instance of type Singleton
    private static Singleton single_instance = null;

    // private constructor restricted to this class itself
    private Singleton()
    {
        ...
    }

    // static method to create instance of Singleton class
    public static Singleton getInstance()
    {
        if (single_instance == null)
            single_instance = new Singleton();

        return single_instance;
    }
}
```

Singleton Pattern

Le Patron Singleton (exemple)

```
class Captain
{
    private static Captain captain;
    //We make the constructor private to prevent the use of "new"
    private Captain() { }
    public static synchronized Captain getCaptain(){
        // Lazy initialization
        if (captain == null)
        {
            captain = new Captain();
            System.out.println("New captain is elected for your team.");
        }
        else
        {
            System.out.print("You already have a captain for your team.");
            System.out.println("Send him for the toss.");
        }
        return captain;
    }
}
// We cannot extend Captain class. The constructor is private in this case.
//class B extends Captain{}// error
```

Singleton Pattern

Le Patron Singleton (exemple)

```
public class SingletonPatternExample {  
    public static void main(String[] args) {  
  
        System.out.println("/**Singleton Pattern Demo**\n");  
        System.out.println("Trying to make a captain for your team:");  
  
        //Constructor is private. We cannot use "new" here.  
        //Captain c3 = new Captain();//error  
  
        Captain captain1 = Captain.getCaptain();  
  
        System.out.println("Trying to make another captain for your team:");  
        Captain captain2 = Captain.getCaptain();  
  
        if (captain1 == captain2)  
        {  
            System.out.println("captain1 and captain2 are same instance.");  
        }  
    }  
}
```

Structural Patterns

Les Patrons de structuration

Structural Patterns

- Decorator (Wrapper) Pattern / Le Patron Décorateur
- Adapter Pattern / Le Patron Adaptateur
- Composite Pattern / Le Patron Composite
- Proxy Pattern / Le Patron Proxy

Structural Patterns

Decorator (Wrapper) Pattern / Le Patron Décorateur

- **Decorator** est un pattern de conception structurel qui vous permet d'ajouter de nouveaux comportements (des responsabilités).
- Principes SOLID :
 - **Single responsibility** : un décorateur => une responsabilité
 - **Open/Closed** : Ajouter un décorateur sans modification de l'existant
 - **Interface segregation** : Objets simples, déléguer les options aux décorateurs

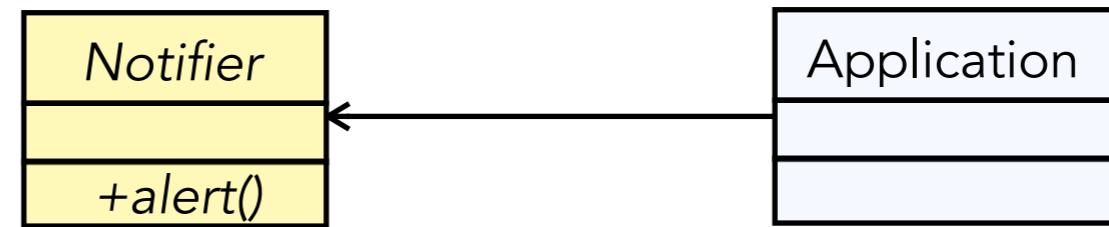
Structural Patterns

Decorator (Wrapper) Pattern / Le Patron Décorateur

- **Decorator** est un pattern de conception structurel qui vous permet d'ajouter de nouveaux comportements (des responsabilités).
- Principes SOLID :
 - **Single responsibility** : un décorateur => une responsabilité
 - **Open/Closed** : Ajouter un décorateur sans modification de l'existant
 - **Interface segregation** : Objets simples, déléguer les options aux décorateurs

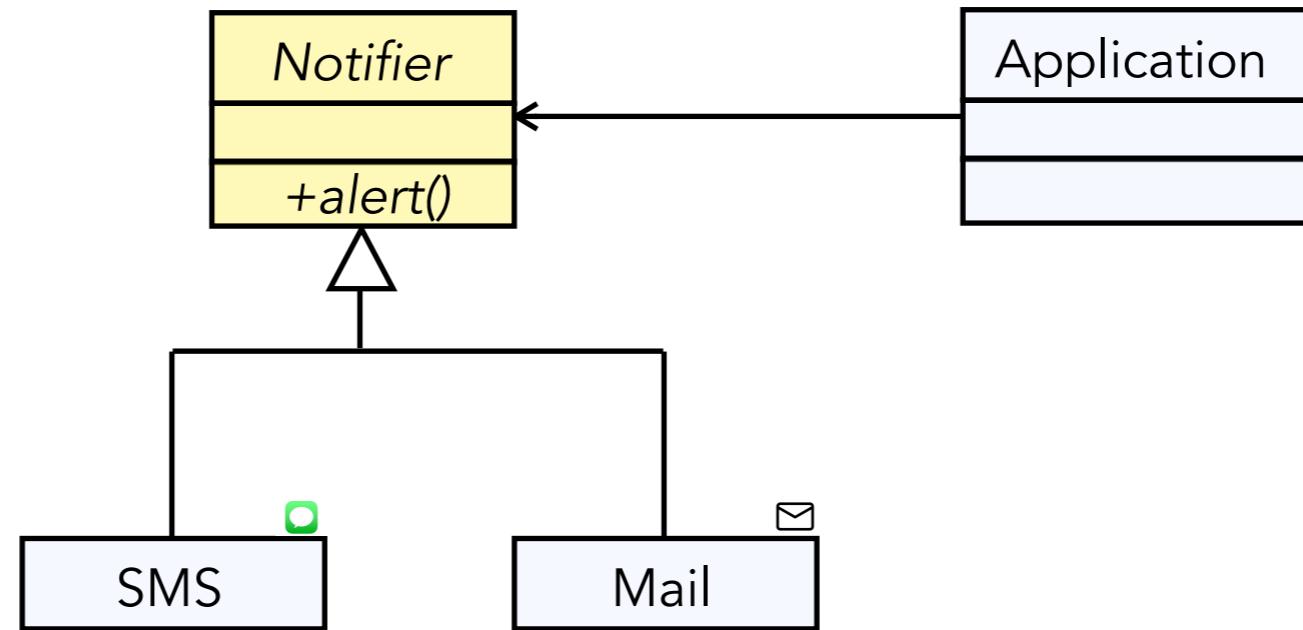
Decorator

Notifier example



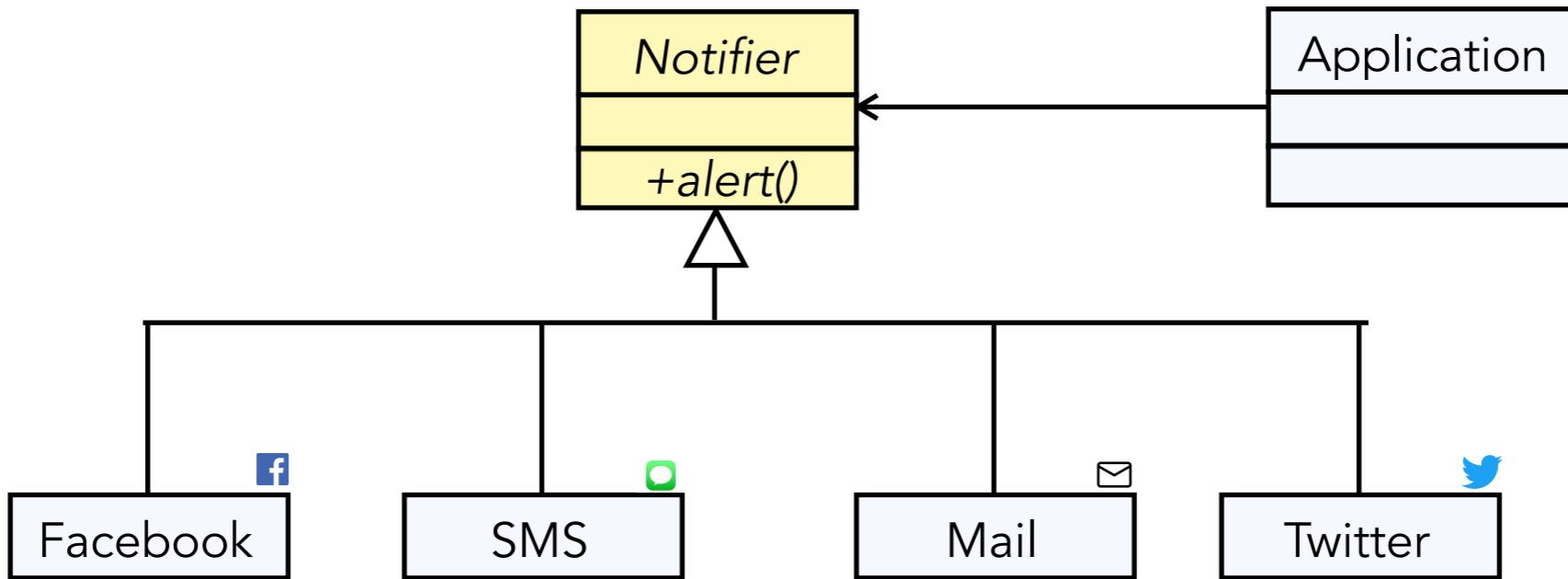
Decorator

Notifier example



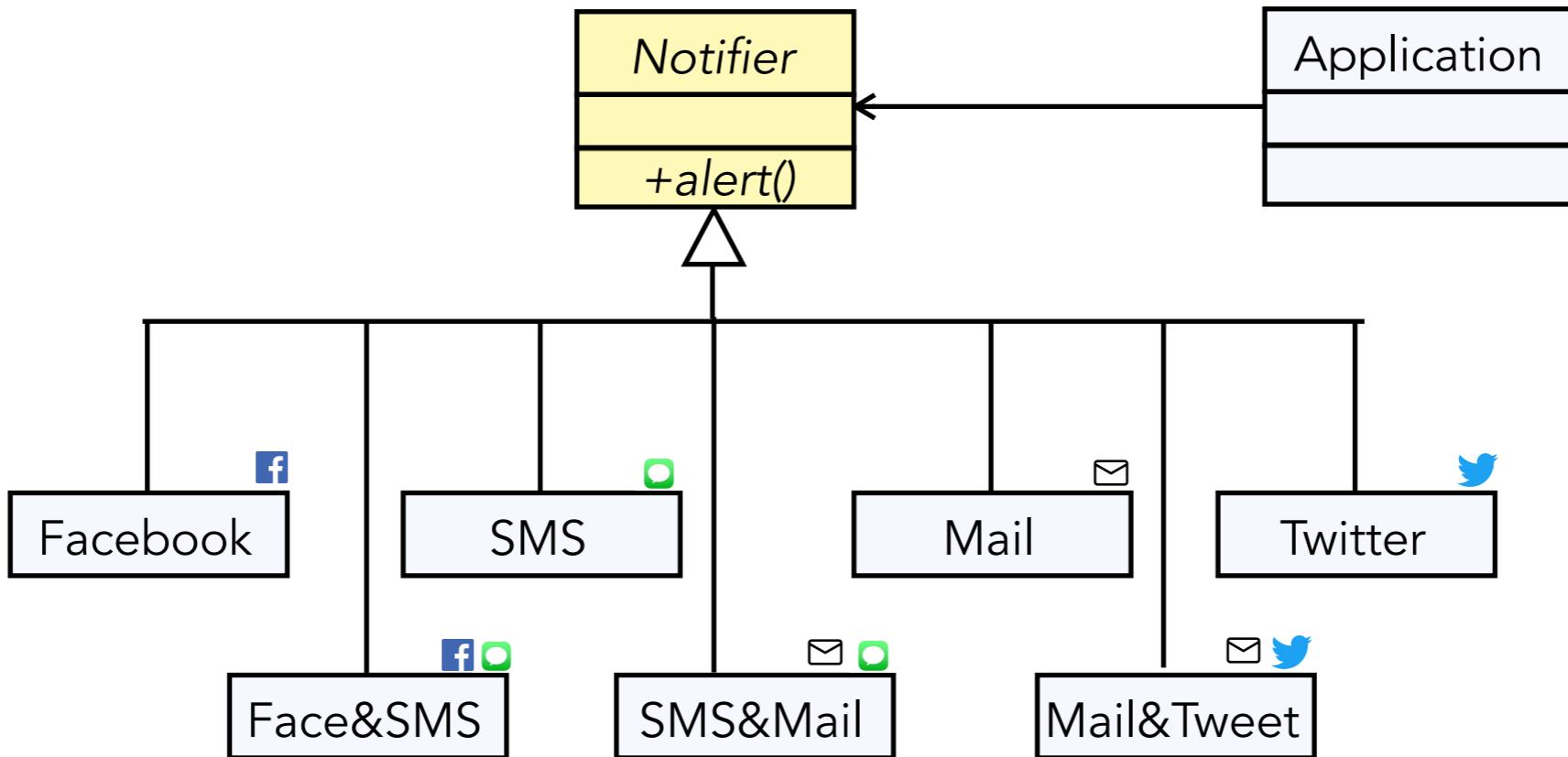
Decorator

Notifier example



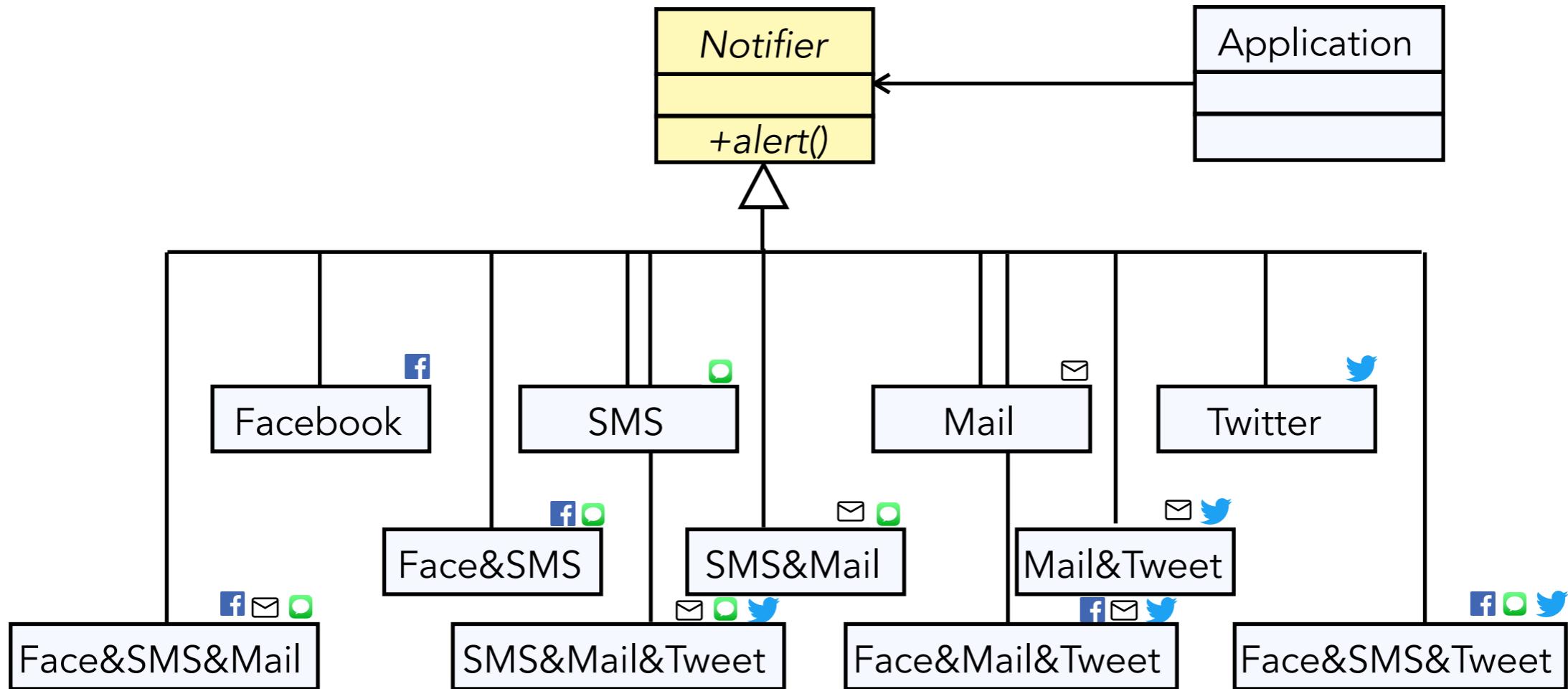
Decorator

Notifier example



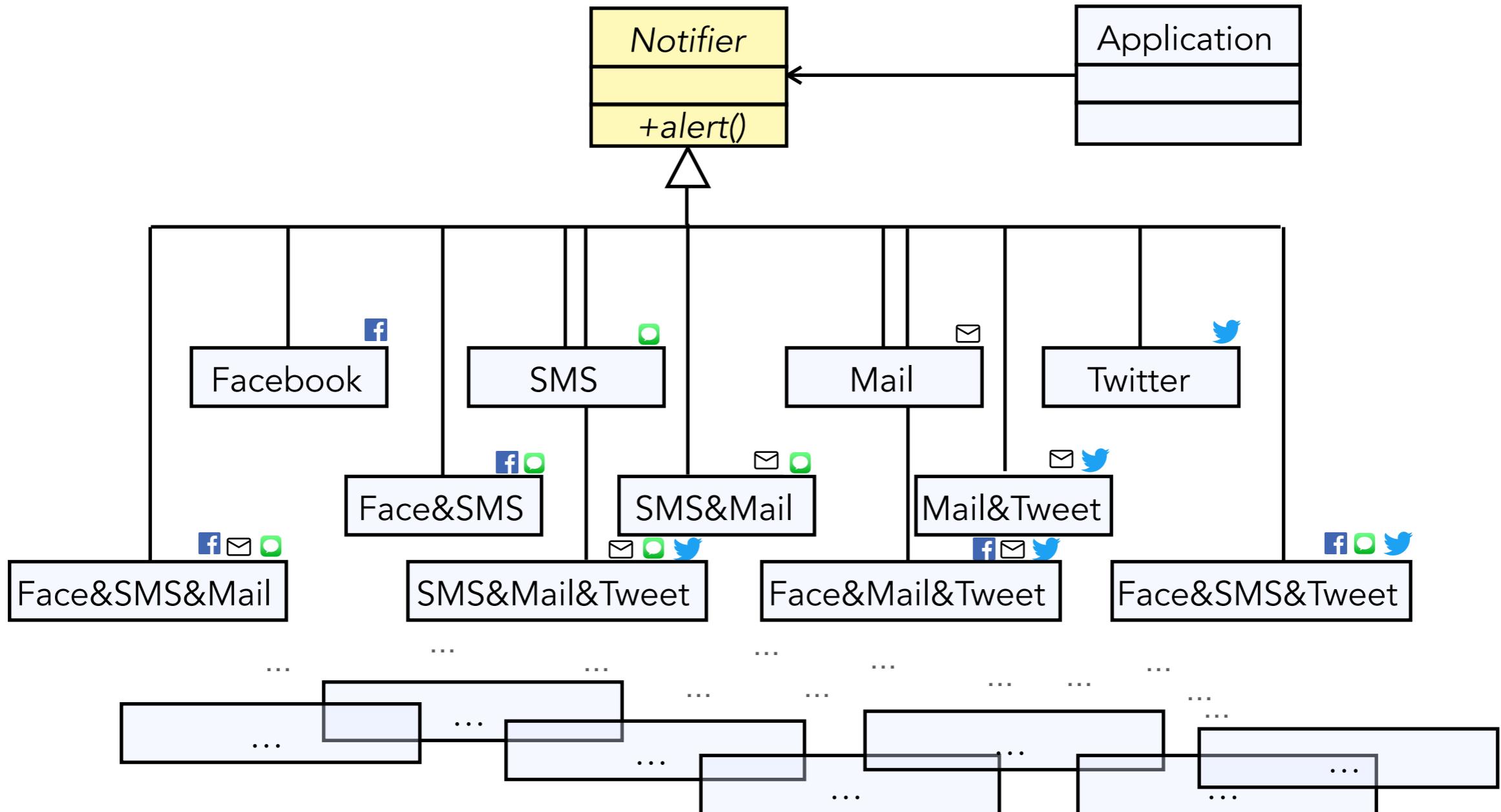
Decorator

Notifier example



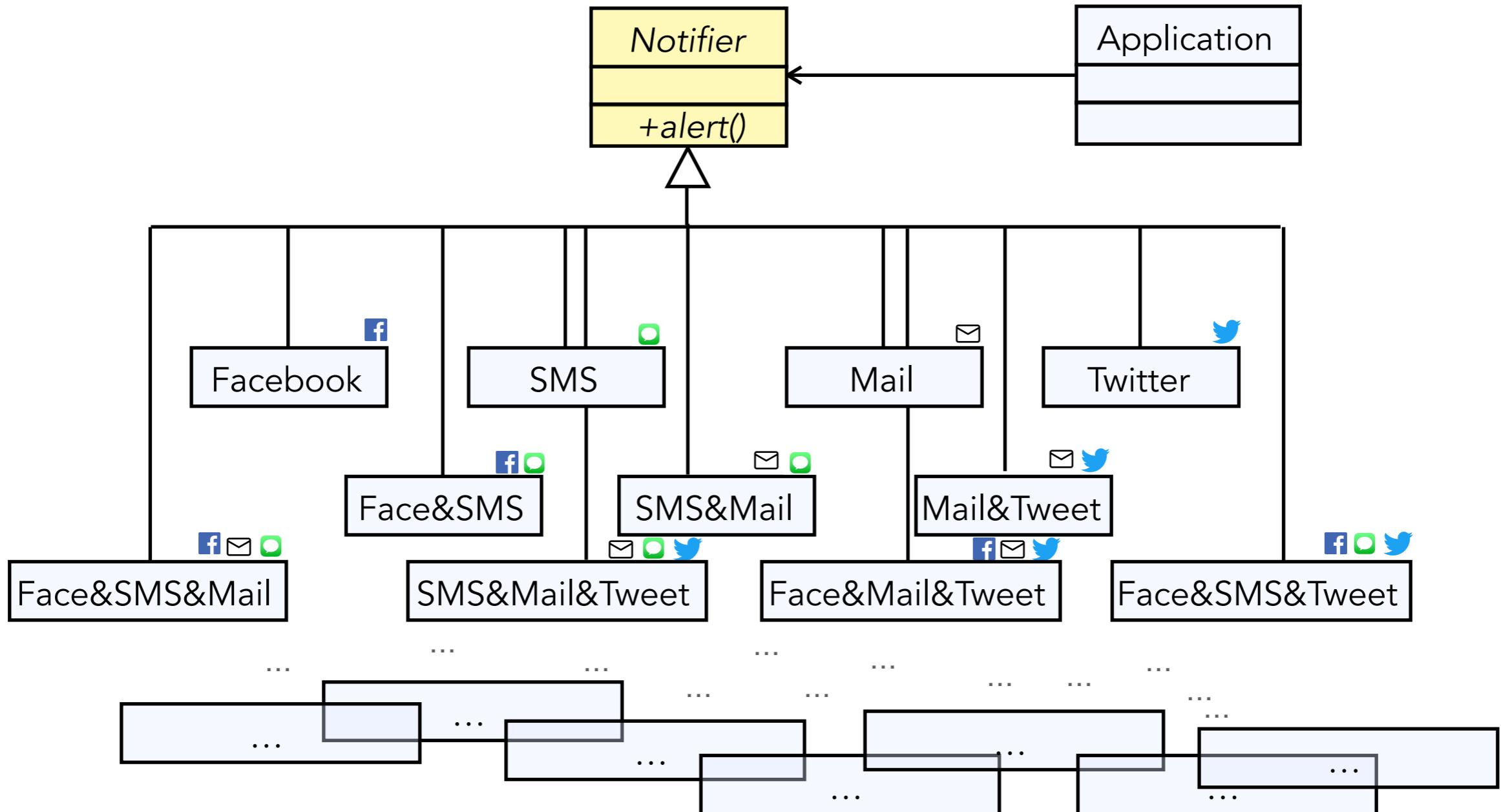
Decorator

Notifier example



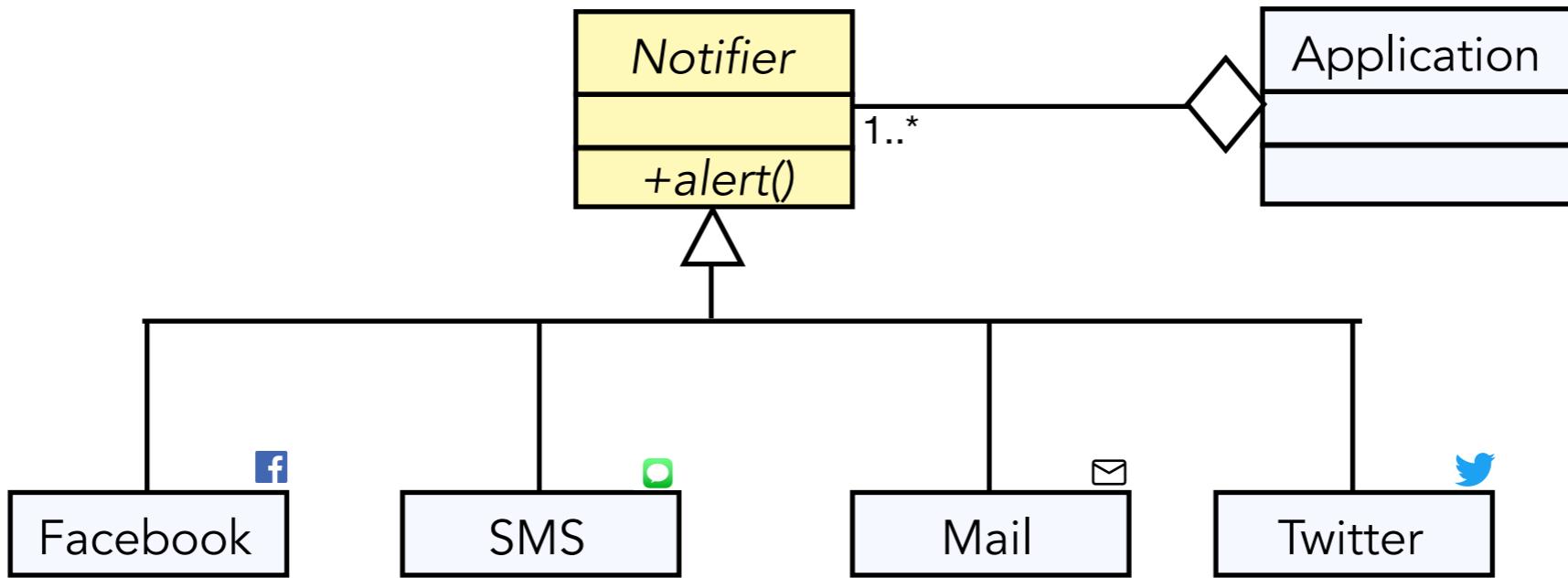
Decorator

Notifier example



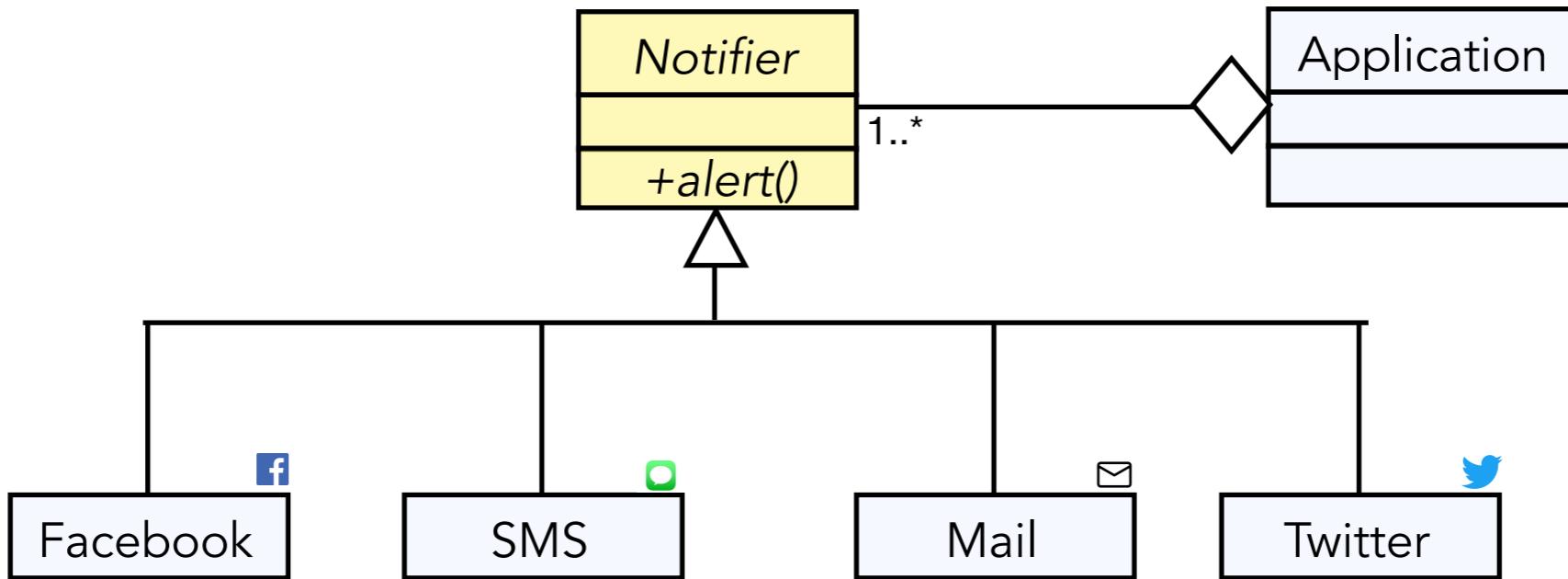
Decorator

Notifier example



Decorator

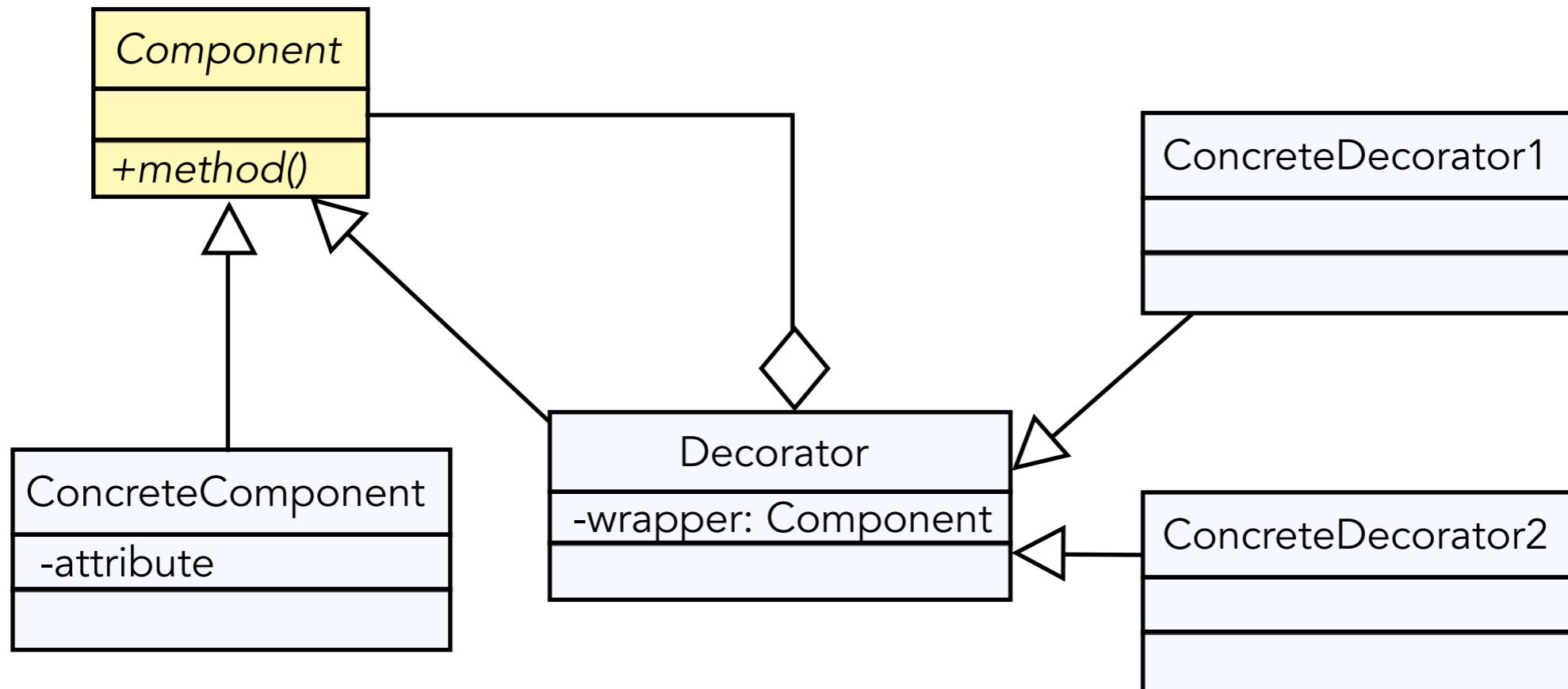
Notifier example with an acceptable solution



Ex: Face&SMS&Mail

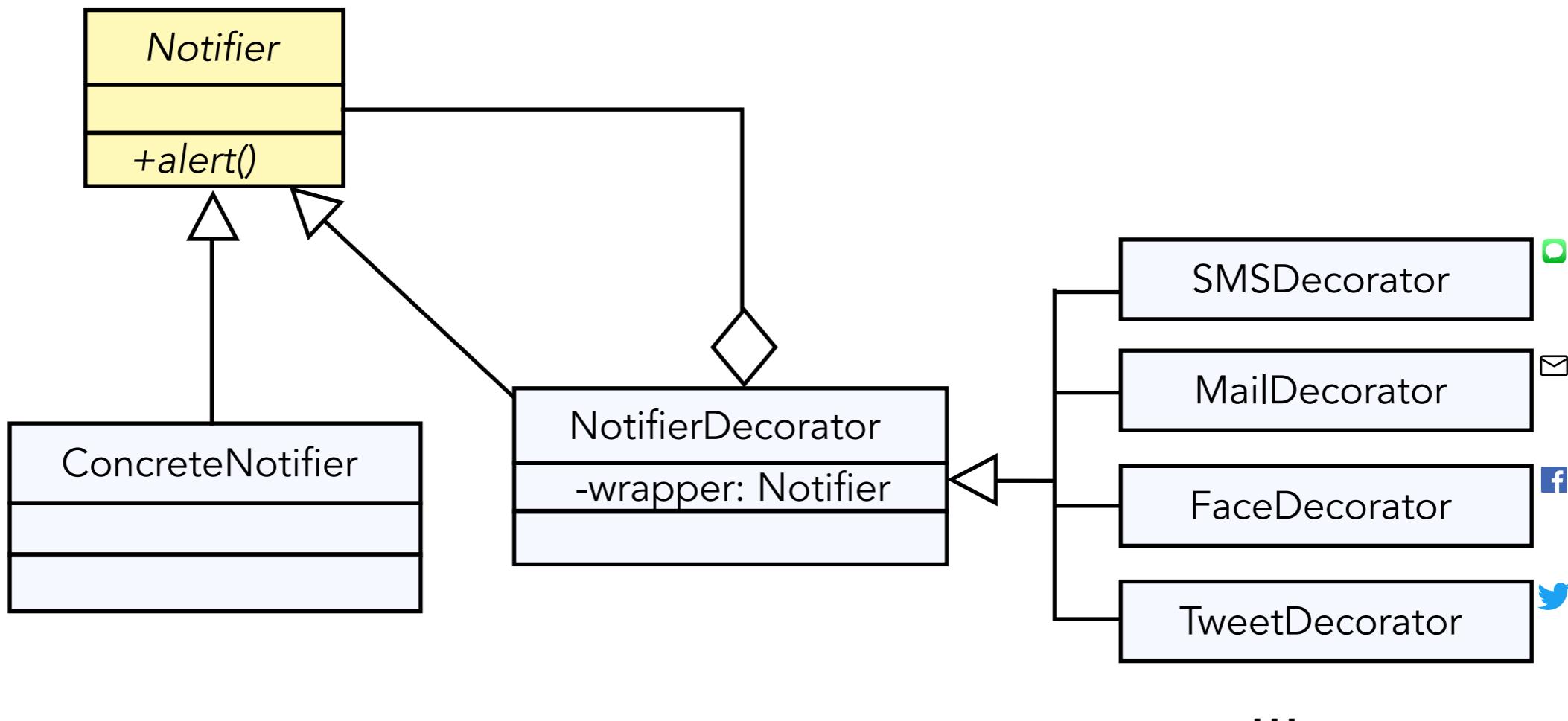
Decorator

UML



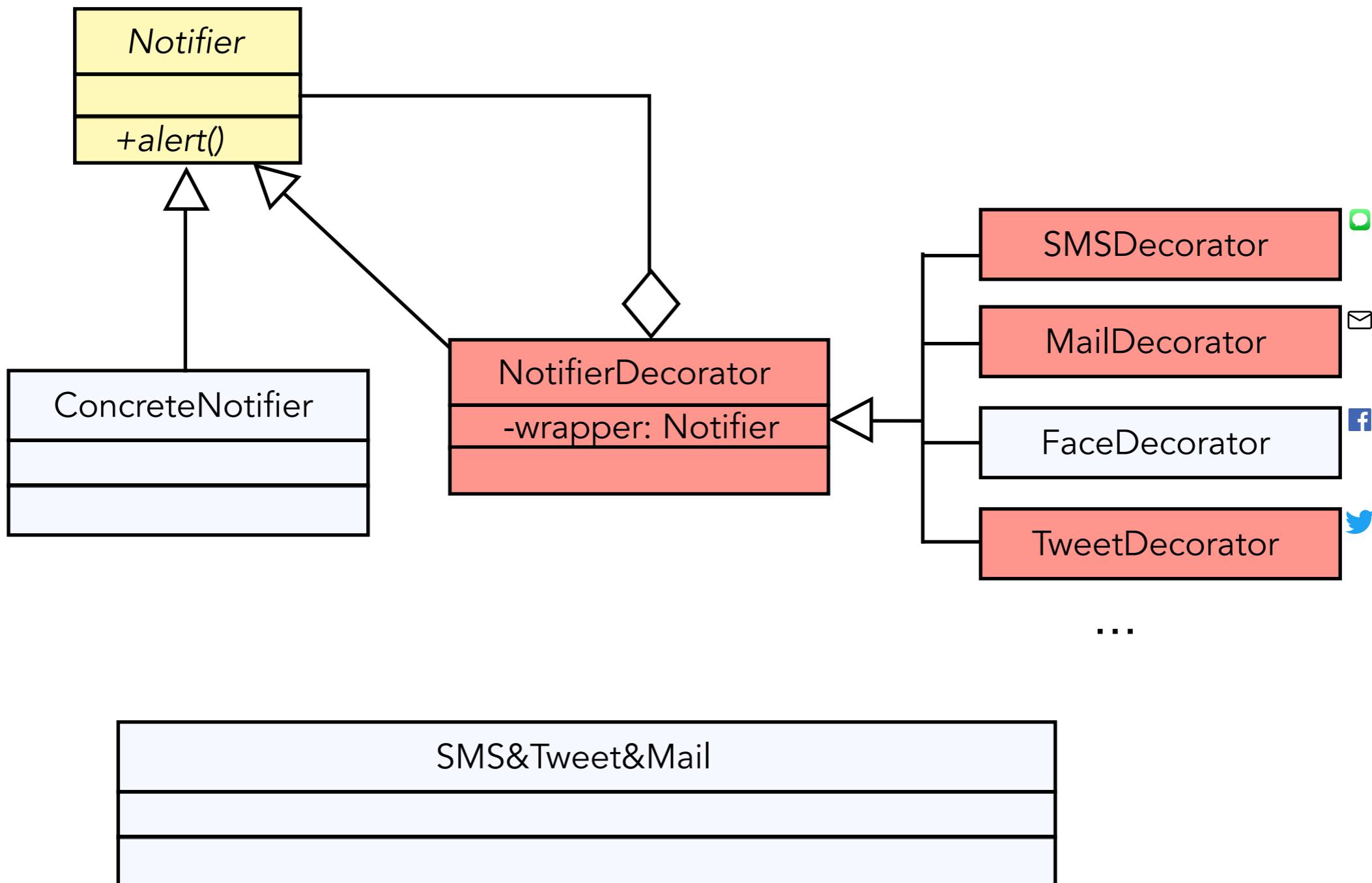
Decorator

Notifier example with Decorator Pattern (UML)



Decorator

Notifier example with Decorator Pattern (UML)



Decorator

Notifier example with Decorator Pattern (JAVA)

```
public class Notifier {  
    public String alert() {  
        return "Alert!!";  
    }  
}
```

Decorator

Notifier example with Decorator Pattern (JAVA)

```
public class Notifier {  
    public String alert() {  
        return "Alert!!";  
    }  
}
```

```
public abstract class NotifierDecorator extends Notifier {  
    private Notifier wrapper;  
  
    public NotifierDecorator(Notifier notifier) {  
        super("decorator");  
        wrapper = notifier; }  
  
    @Override  
    public String alert() { return wrapper.alert(); }  
}
```

Decorator

Notifier example with Decorator Pattern (JAVA)

```
public class Notifier {  
    public String alert() {  
        return "Alert!!";  
    }  
}
```

```
public abstract class NotifierDecorator extends Notifier {  
    private Notifier wrapper;  
  
    public NotifierDecorator(Notifier notifier) {  
        super("decorator");  
        wrapper = notifier; }  
  
    @Override  
    public String alert() { return wrapper.alert(); }  
}
```

```
public class Mail extends NotifierDecorator {  
    @Override  
    public String alert() { return ":mail:"+super.alert(); }  
  
    public Mail(Notifier notifier) { super(notifier); }  
}
```

```
public class SMS extends NotifierDecorator {  
    @Override  
    public String alert() { return ":sms:"+super.alert(); }  
  
    public SMS(Notifier notifier) { super(notifier); }  
}
```

```
public class Tweet extends NotifierDecorator {  
    @Override  
    public String alert() { return ":tweet:"+super.alert(); }  
  
    public Tweet(Notifier notifier) { super(notifier); }  
}
```

Decorator

Notifier example with Decorator Pattern (JAVA)

```
public class Notifier {  
    public String alert() {  
        return "Alert!!";  
    }  
}
```

```
public class Mail extends NotifierDecorator {  
    @Override  
    public String alert() { return ":mail:"+super.alert(); }  
  
    public Mail(Notifier notifier) { super(notifier); }  
}
```

```
public abstract class NotifierDecorator extends Notifier {  
    private Notifier wrapper;  
  
    public NotifierDecorator(Notifier notifier) {  
        super("decorator");  
        wrapper = notifier; }  
  
    @Override  
    public String alert() { return wrapper.alert(); }  
}
```

```
public class SMS extends NotifierDecorator {  
    @Override  
    public String alert() { return ":sms:"+super.alert(); }  
  
    public SMS(Notifier notifier) { super(notifier); }  
}
```

```
public static void main (String args[]) {  
    Notifier notifier = new SMS(new Tweet(new Mail()));  
    notifier.alert();  
}
```

```
public class Tweet extends NotifierDecorator {  
    @Override  
    public String alert() { return ":tweet:"+super.alert(); }  
  
    public Tweet(Notifier notifier) { super(notifier); }  
}
```

Decorator

Notifier example with Decorator Pattern (JAVA)

```
public class Notifier {  
    public String alert() {  
        return "Alert!!";  
    }  
}
```

```
public class Mail extends NotifierDecorator {  
    @Override  
    public String alert() { return ":mail:"+super.alert(); }  
  
    public Mail(Notifier notifier) { super(notifier); }  
}
```

```
public abstract class NotifierDecorator extends Notifier {  
    private Notifier wrapper;  
  
    public NotifierDecorator(Notifier notifier) {  
        super("decorator");  
        wrapper = notifier; }  
  
    @Override  
    public String alert() { return wrapper.alert(); }  
}
```

```
public class SMS extends NotifierDecorator {  
    @Override  
    public String alert() { return ":sms:"+super.alert(); }  
  
    public SMS(Notifier notifier) { super(notifier); }  
}
```

```
public static void main (String args[]) {  
    Notifier notifier = new SMS(new Tweet(new Mail()));  
    notifier.alert();  
}
```

```
public class Tweet extends NotifierDecorator {  
    @Override  
    public String alert() { return ":tweet:"+super.alert(); }  
  
    public Tweet(Notifier notifier) { super(notifier); }  
}
```

```
>>  
>>  
>>  
>> :sms::tweet::mail::Alert!!
```

Structural Patterns

- Decorator (Wrapper) Pattern / Le Patron Décorateur
- Adapter Pattern / Le Patron Adaptateur
- Composite Pattern / Le Patron Composite
- Proxy Pattern / Le Patron Proxy

Structural Patterns

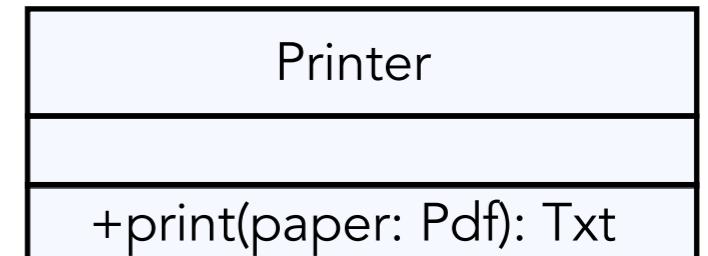
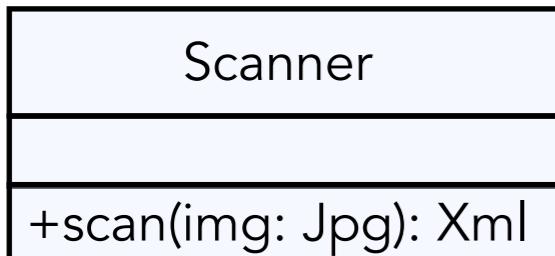
Adapter Pattern / Le Patron Adaptateur

- **Adapter** est un pattern qui permet aux objets avec des interfaces incompatibles de collaborer.
- Principes SOLID :
 - **Liskov Substitution** : Adapter remplace la cible de façon transparente.

Adapter Pattern

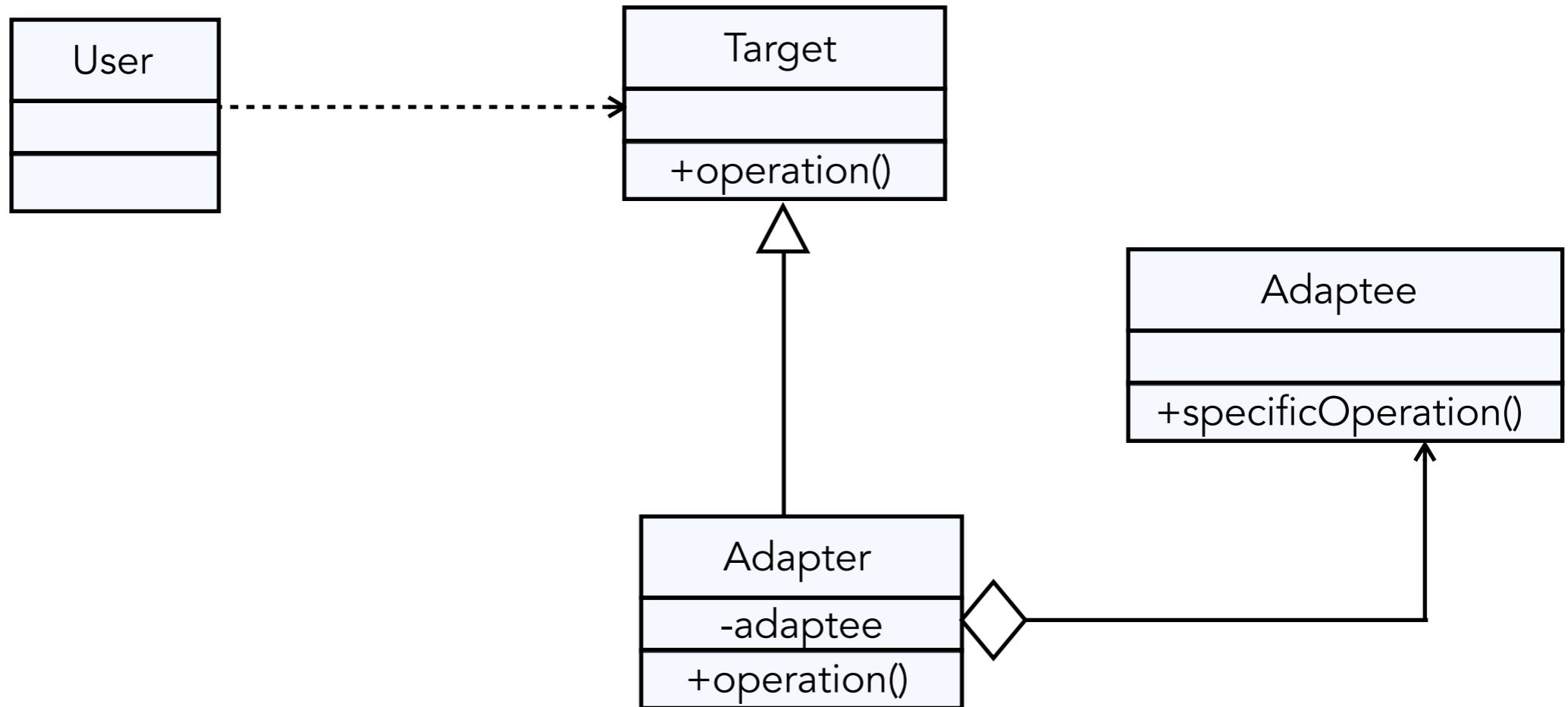
Photocopier Example

- **Scanner** : scanne une feuille, produit un fichier JPG et retourne un compte-rendu de l'opération en format XML
- **Printer** : prend un PDF, imprime sur papier et retourne un compte-rendu format txt



Adapter Pattern

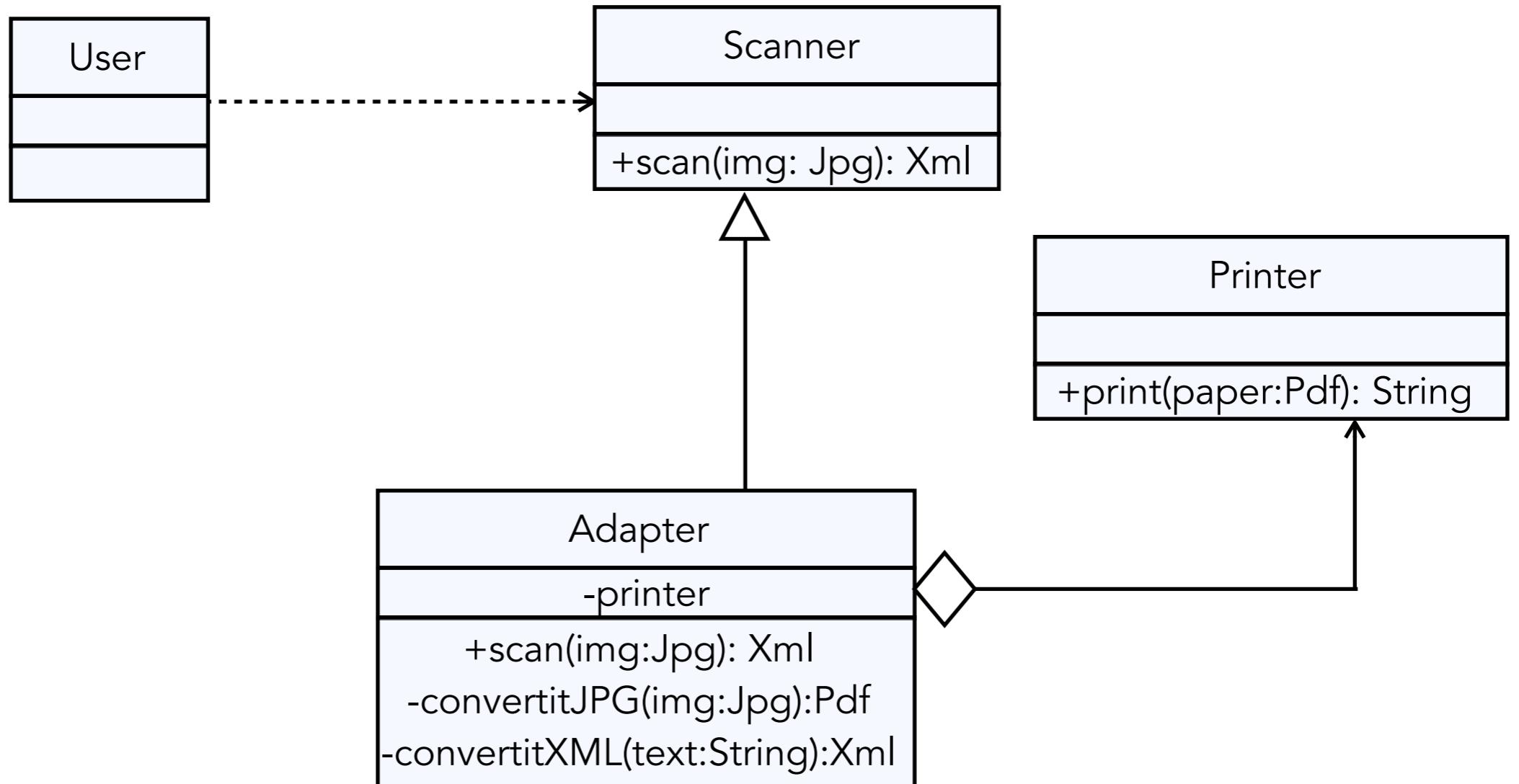
Object Adapter (UML)



- Cette implémentation utilise le principe de composition: l'adaptateur implémente une cible et enveloppe un service. Il peut être implémenté dans tous les langages de programmation courants.

Adapter Pattern

Photocopier Example (JAVA)



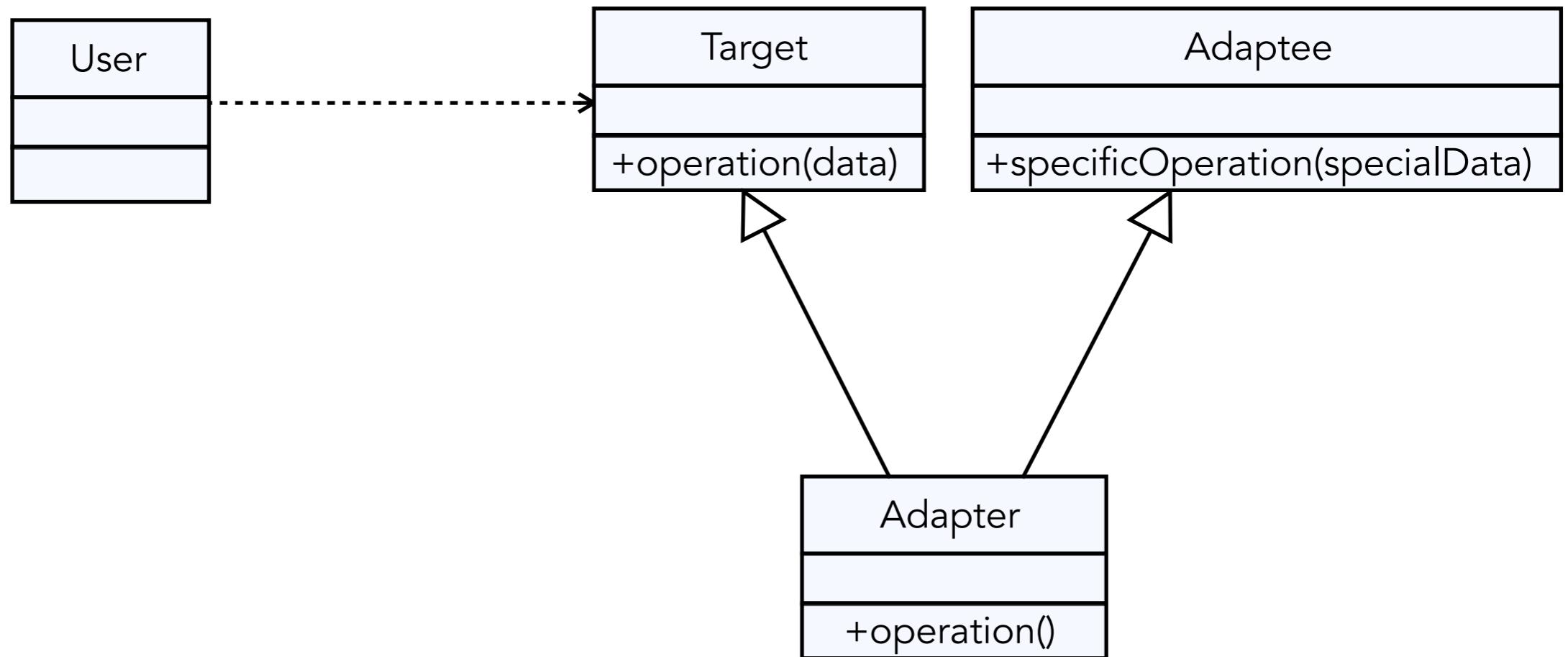
Adapter Pattern

Photocopier Example (JAVA)

```
public class Adapter extends Scanner {  
    private Printer printer = Printer.getInstance();  
  
    public Xml scan(img : Jpg) {  
  
        Pdf tmp = convertitJPG(img);  
        String res = printer.print(tmp);  
        Xml resXML = convertitXML(res);  
  
        return resXML;  
    }  
    private Pdf convertitJPG(img : Jpg) {...}  
  
    private Xml convertitXML(str : String) {...}  
}
```

Adapter Pattern

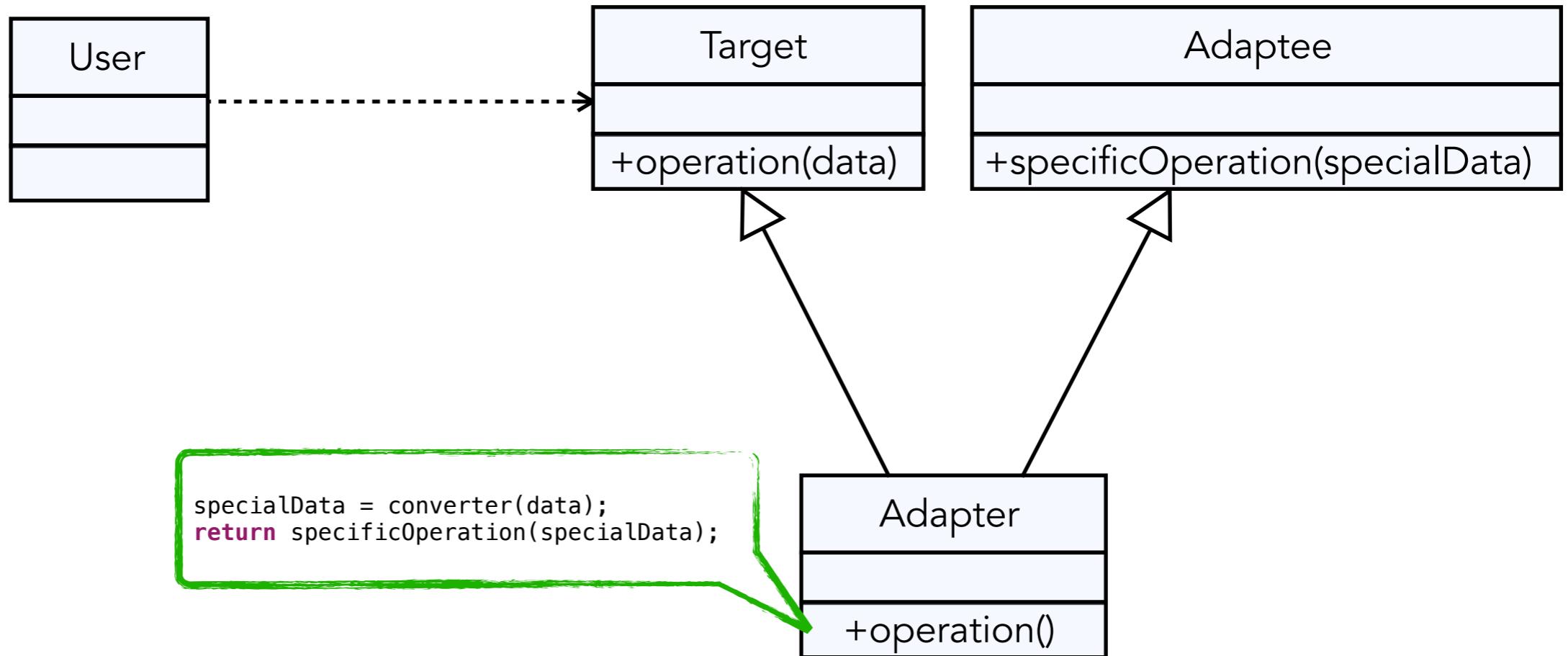
Class Adapter (UML)



- Cette implémentation utilise l'héritage: l'adaptateur hérite de la cible et du service en même temps. Notez que cette approche ne peut être implémentée que dans les langages de programmation qui prennent en charge l'héritage multiple, tels que C++

Adapter Pattern

Class Adapter (UML)



- Cette implémentation utilise l'héritage: l'adaptateur hérite de la cible et du service en même temps. Notez que cette approche ne peut être implémentée que dans les langages de programmation qui prennent en charge l'héritage multiple, tels que C++

Structural Patterns

- Decorator (Wrapper) Pattern / Le Patron Décorateur
- Adapter Pattern / Le Patron Adaptateur
- Composite Pattern / Le Patron Composite
- Proxy Pattern / Le Patron Proxy

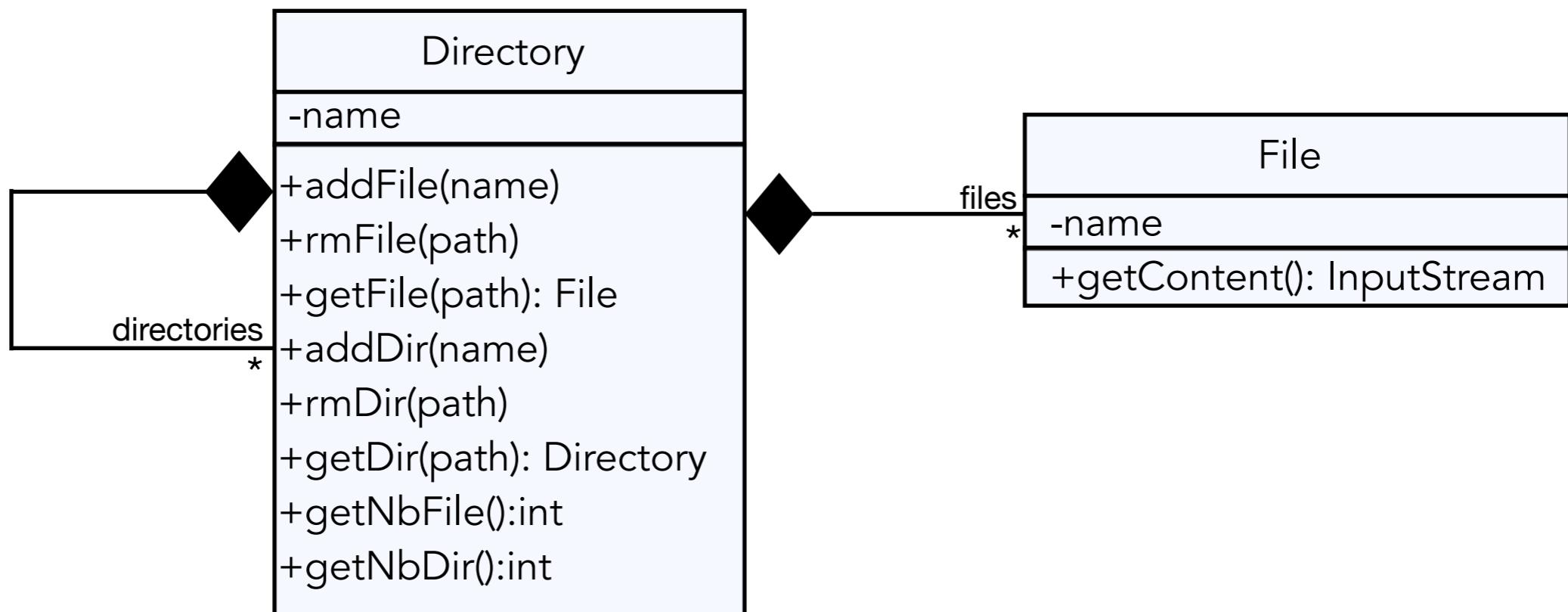
Structural Patterns

Composite Pattern / Le Patron Composite

- **Composite** est un pattern de conception qui permet de composer des objets en arborescence avec une hiérarchie composant/composé, puis de travailler avec ces structures comme s'il s'agissait d'objets individuels.
- Principes SOLID :
 - **Liskov Substitution** : Un composite est équivalent à un composant
 - **Dependency Inversion** : Favorise l'utilisation d'interfaces

Composite Pattern

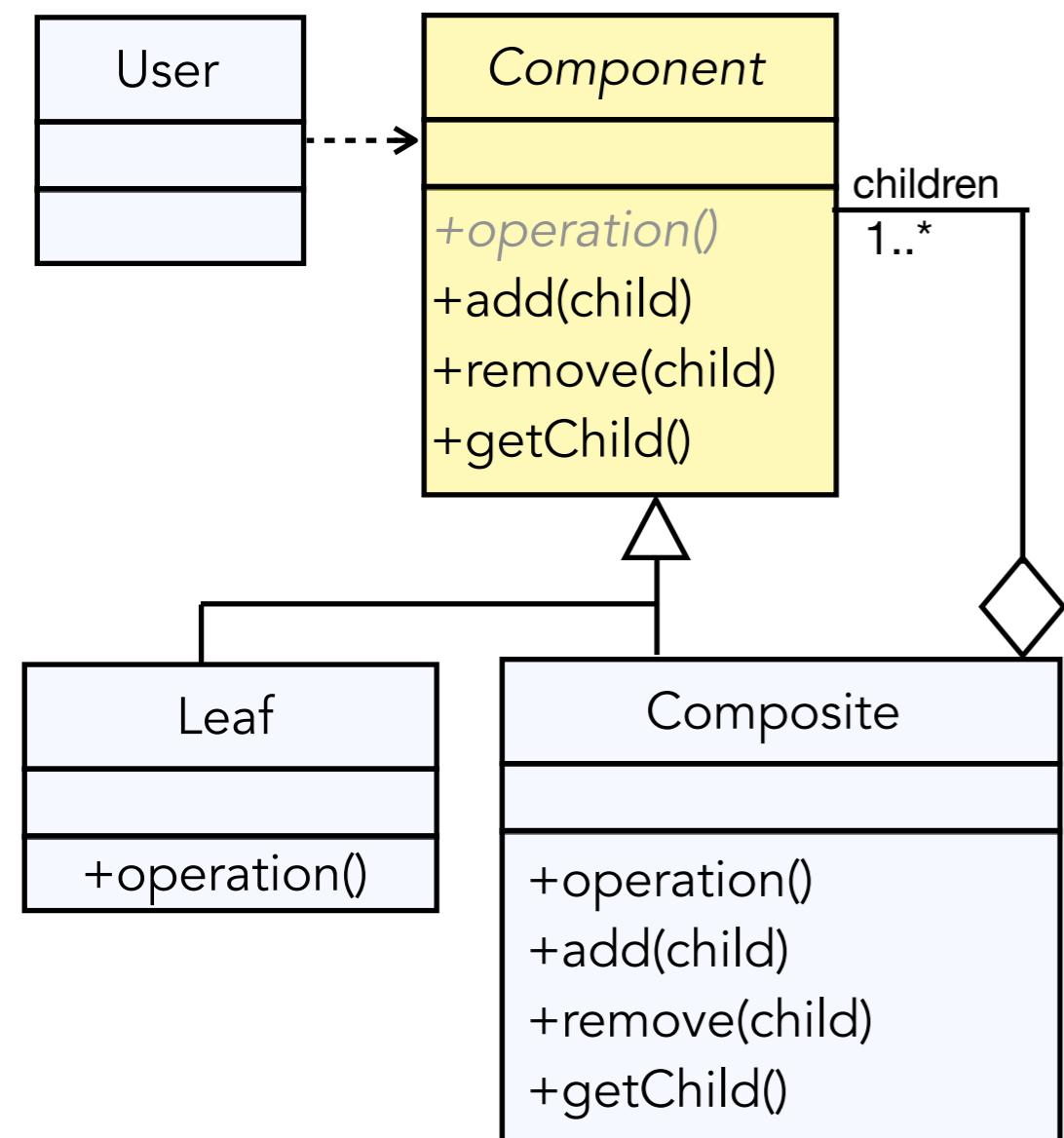
Hard Disk Example



- séparation nette entre les fichiers et les répertoires (définition et traitement)
- Travail en double
- Revoir la classe **Directory** dans le cas d'ajout d'un nouveau type d'élément

Composite Pattern

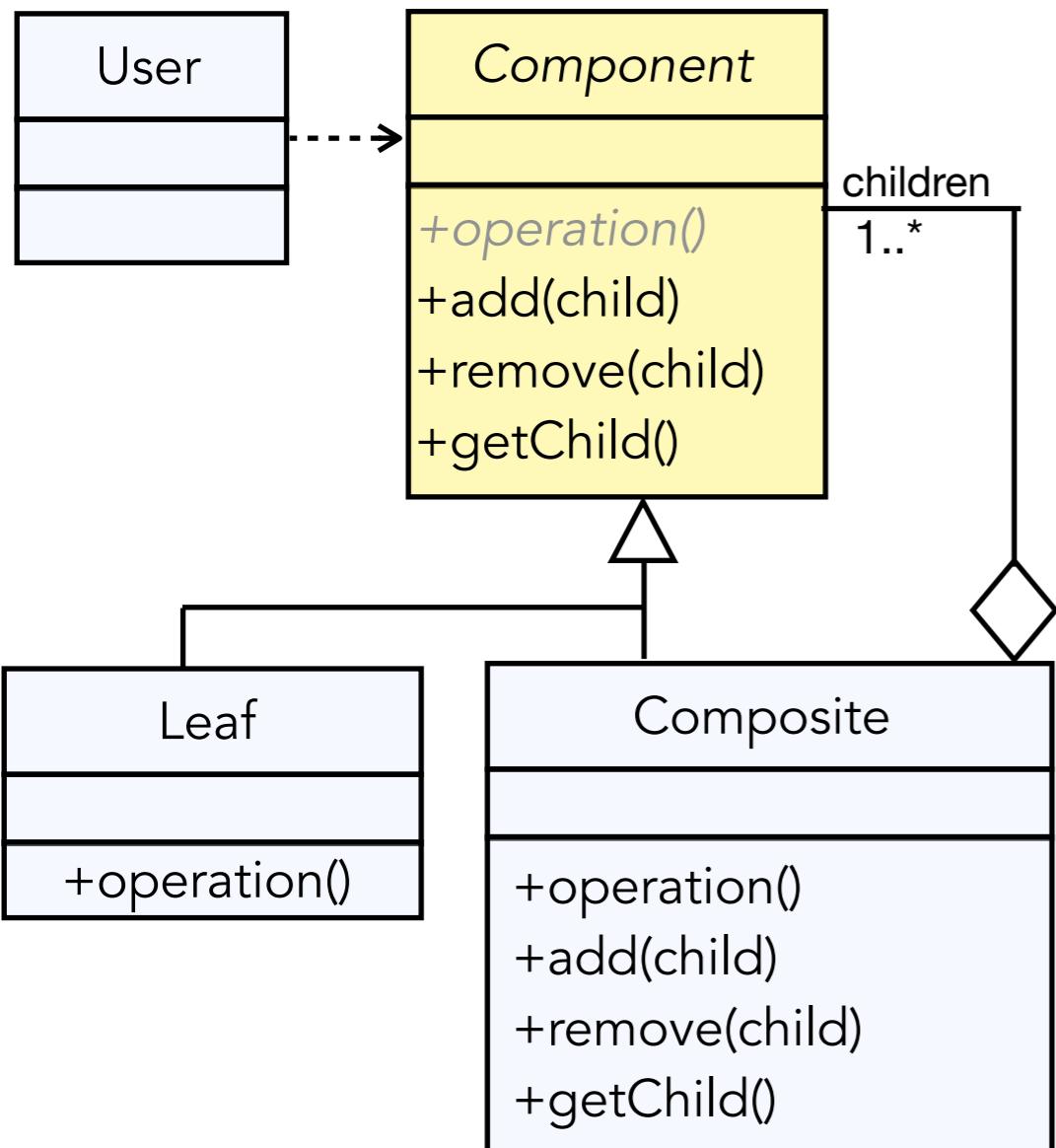
Uniform / TypeSafe



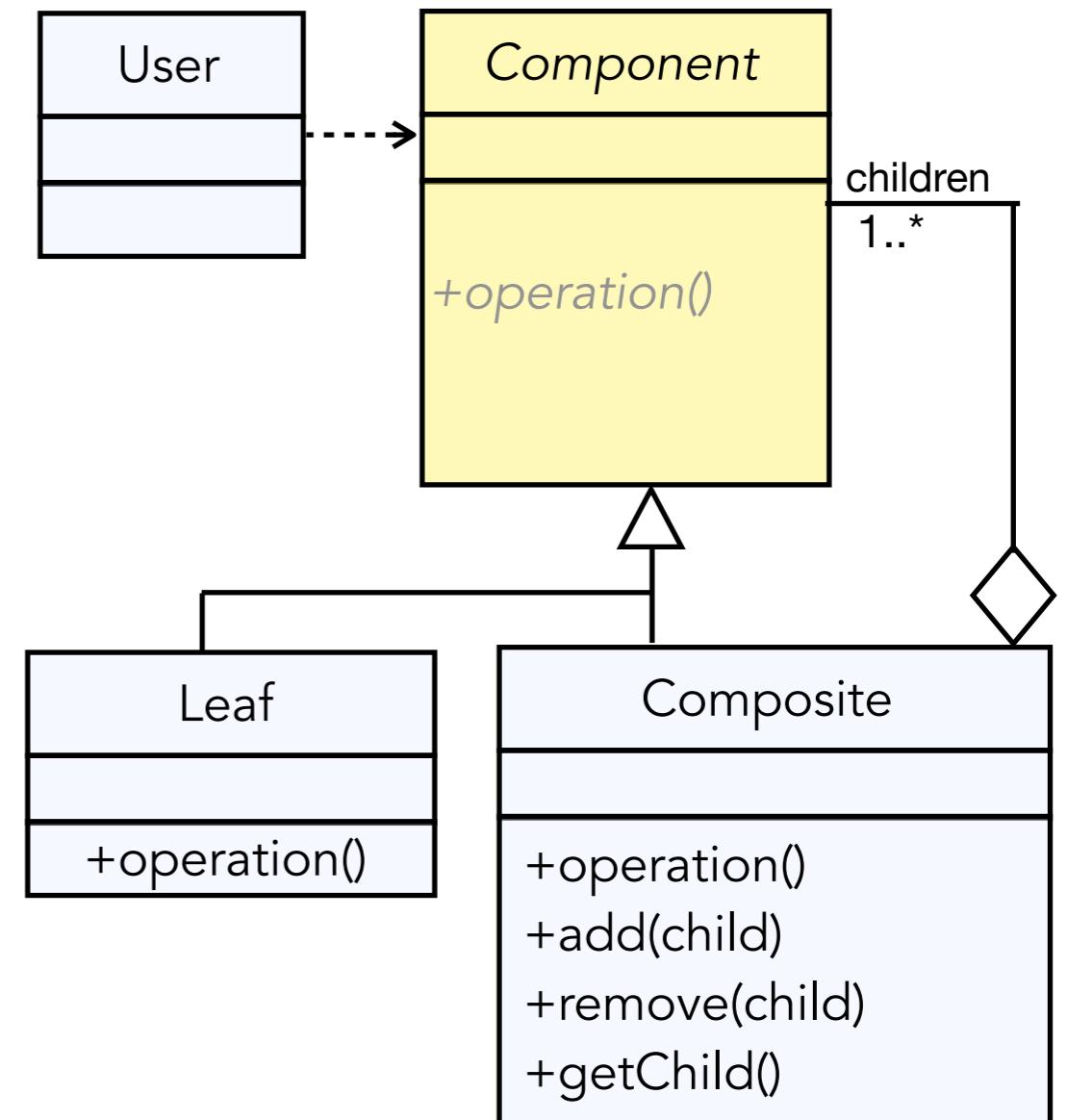
Design for Uniformity

Composite Pattern

Uniform / TypeSafe



Design for Uniformity



Design for Type Safety

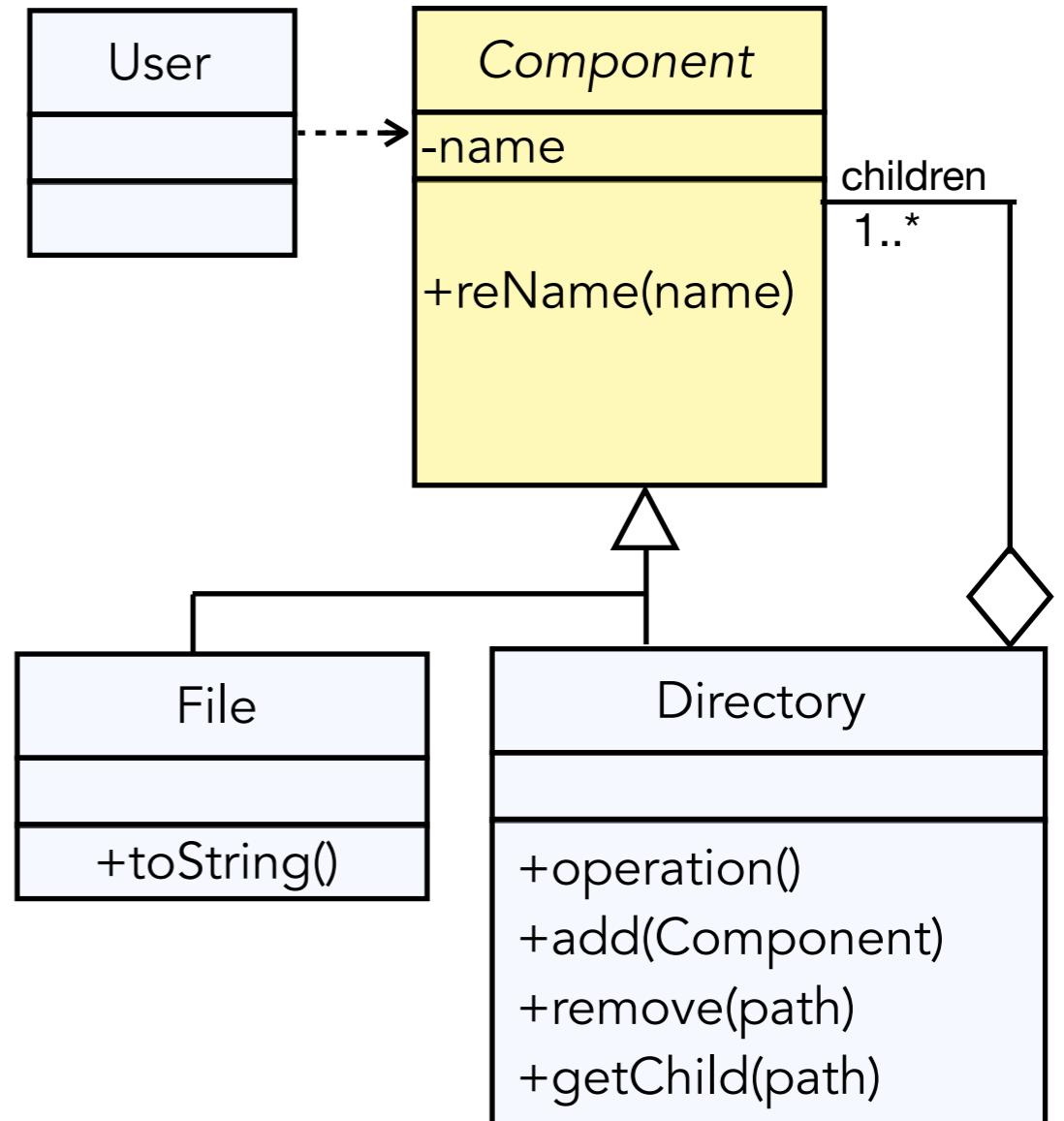
Composite Pattern

Hard Disk Example

```
public abstract class Component {  
    private String name;  
  
    public Component(String name) { this.name = name; }  
  
    public void rename (String name) {  
        this.name = name;  
    }  
}
```

```
public class File extends Component {  
    @Override  
    public String toString() { return "file: "+getName();  
}  
}
```

```
public class Directory extends Component {  
  
    private ArrayList<Component> children  
        = new ArrayList<>();  
  
    public add(Component c) { children.add(c); }  
  
    public remove(String path) { children.remove(path); }  
  
    public Component getChild(String path) { return  
children.get(path); }  
}
```

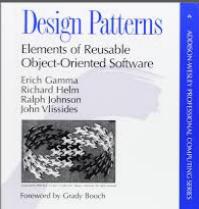


Design for Type Safety

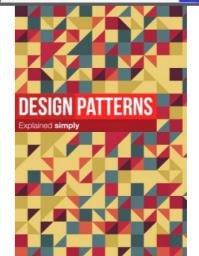
Books

Design Pattern

- **Design Patterns: Elements of Reusable Object-Oriented Software.** Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides (GoF: Gang of Four).



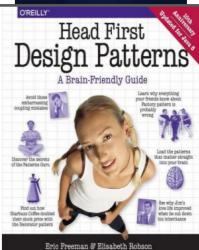
- **DESIGN PATTERNS Explained simply.** Alexander Shvets. 2013



- **Dive Into Design Patterns.** Alexander Shvets. 2019



- **Head First Design Patterns.** Freeman et al. 2014



- **Java Design Patterns.** Vaskaran Sarcar. 2019

