



UNIVERSITÉ  
DE MONTPELLIER



# Design Patterns

Qualité de Développement

R3.04

(cours 3)

Nadjib Lazaar ([nadjib.lazaar@umontpellier.fr](mailto:nadjib.lazaar@umontpellier.fr))

# **Structural Patterns**

**Les Patrons de structuration**

# Structural Patterns

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

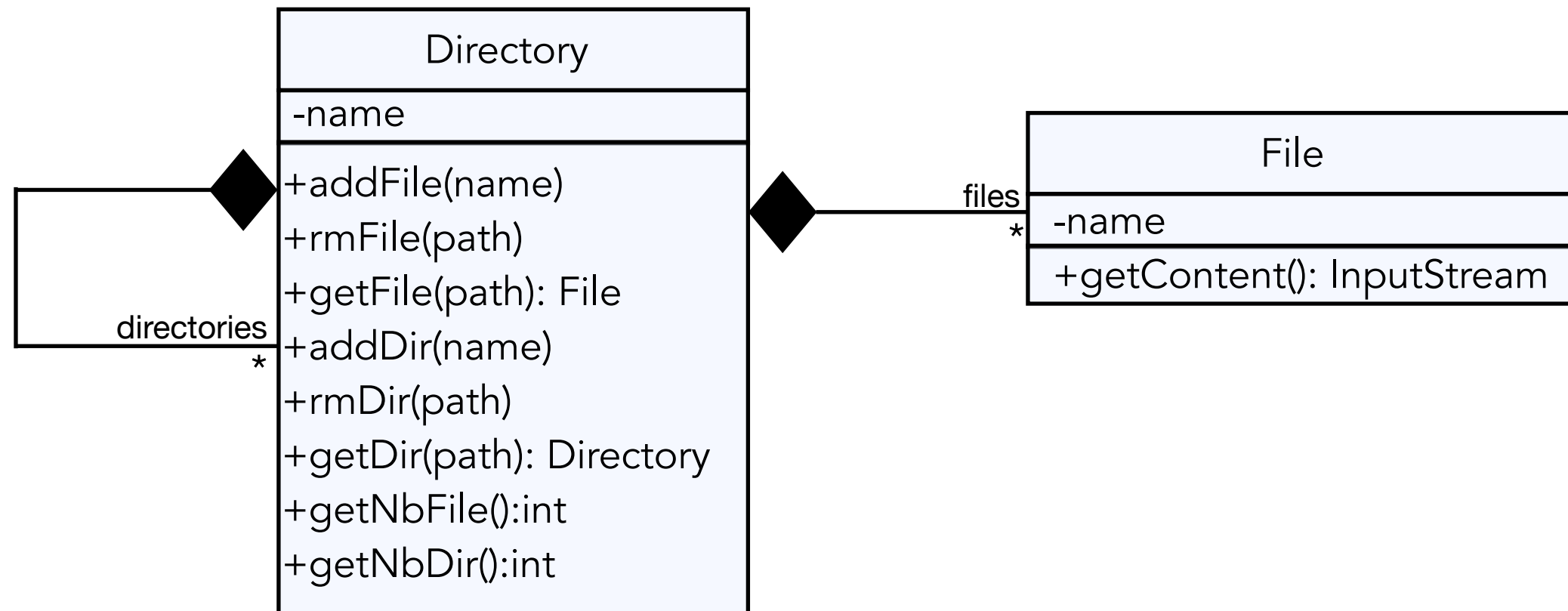
# Structural Patterns

## 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

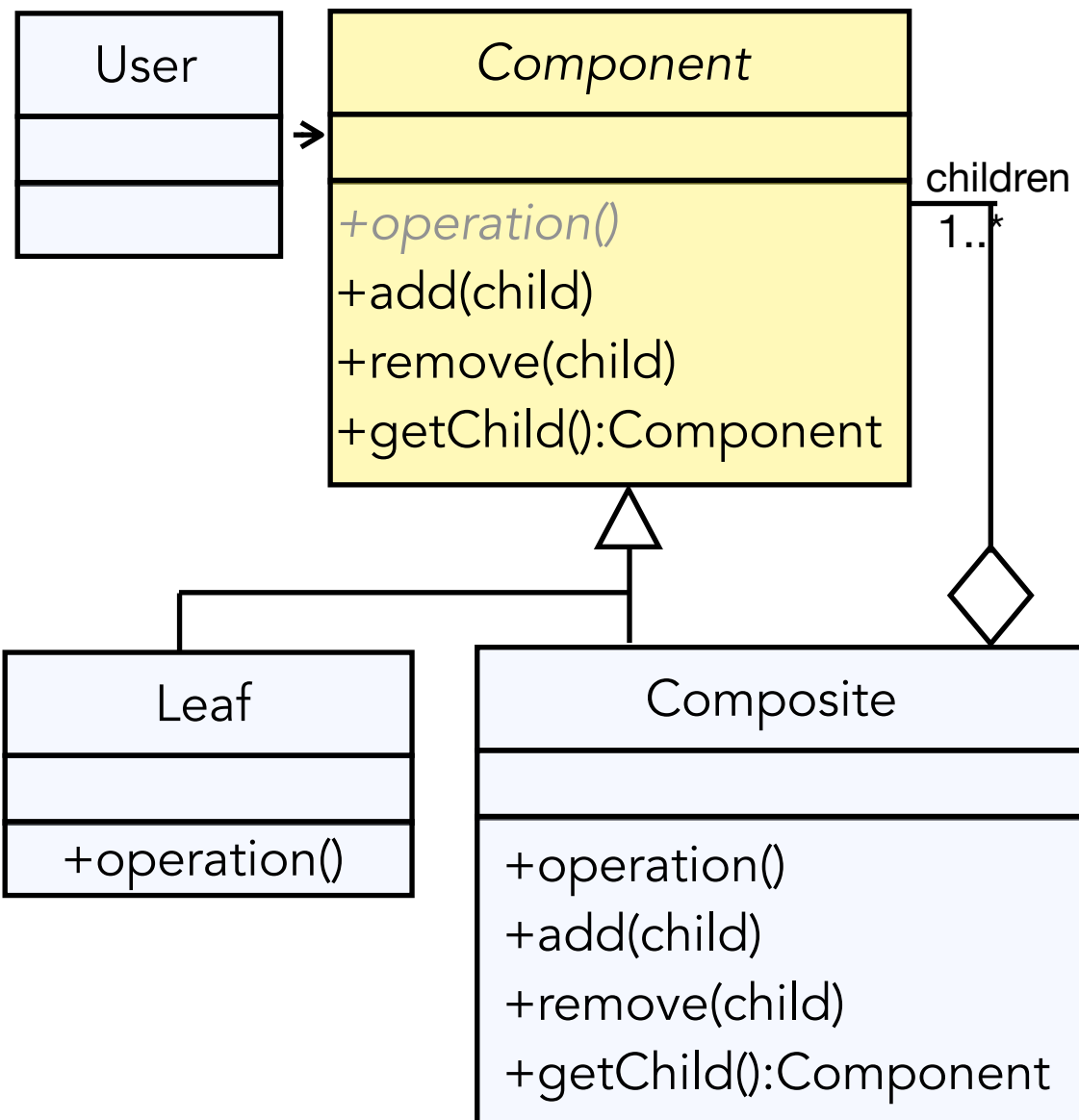
## Hard Disk Example



- 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

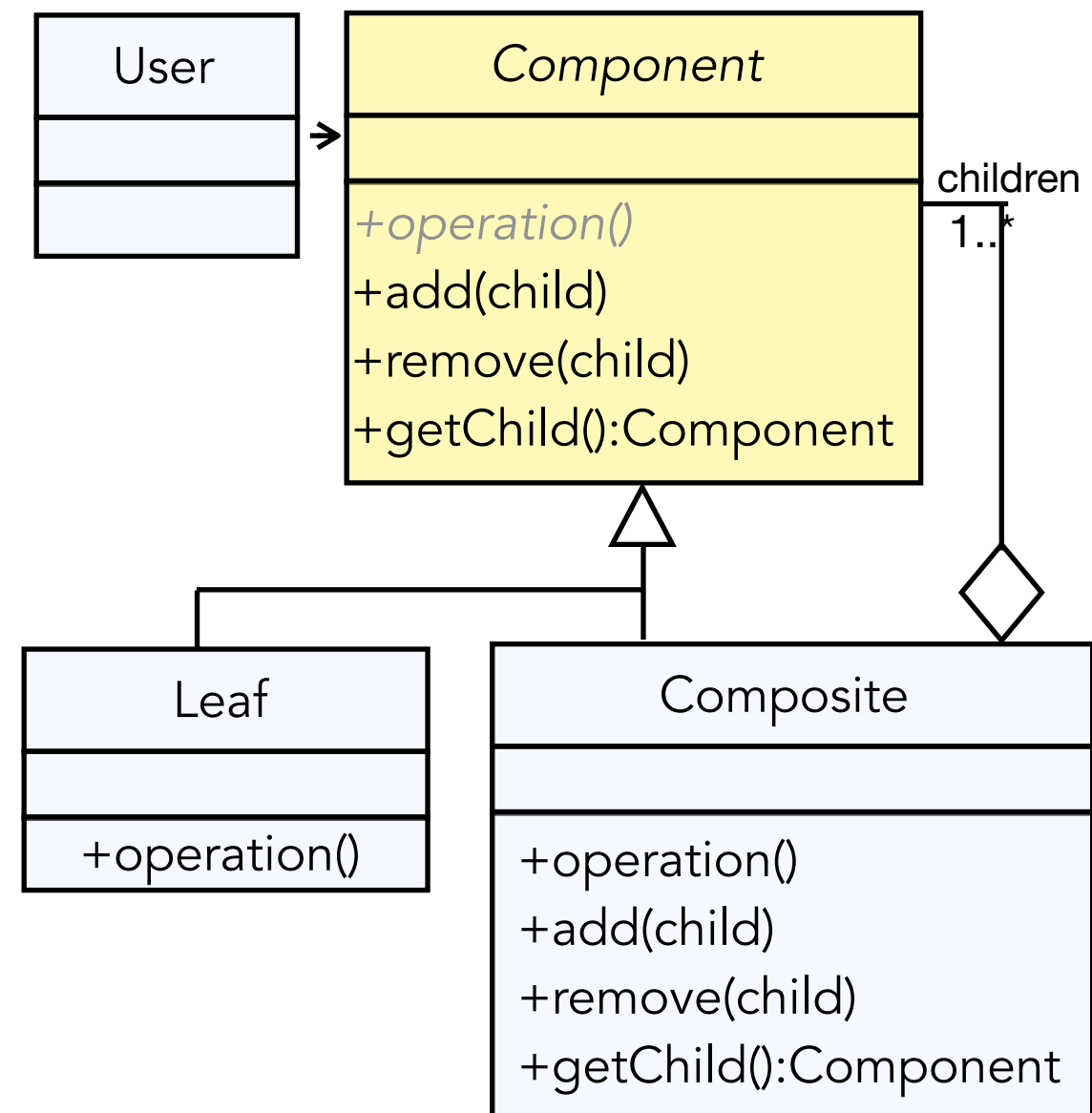
Uniform / TypeSafe



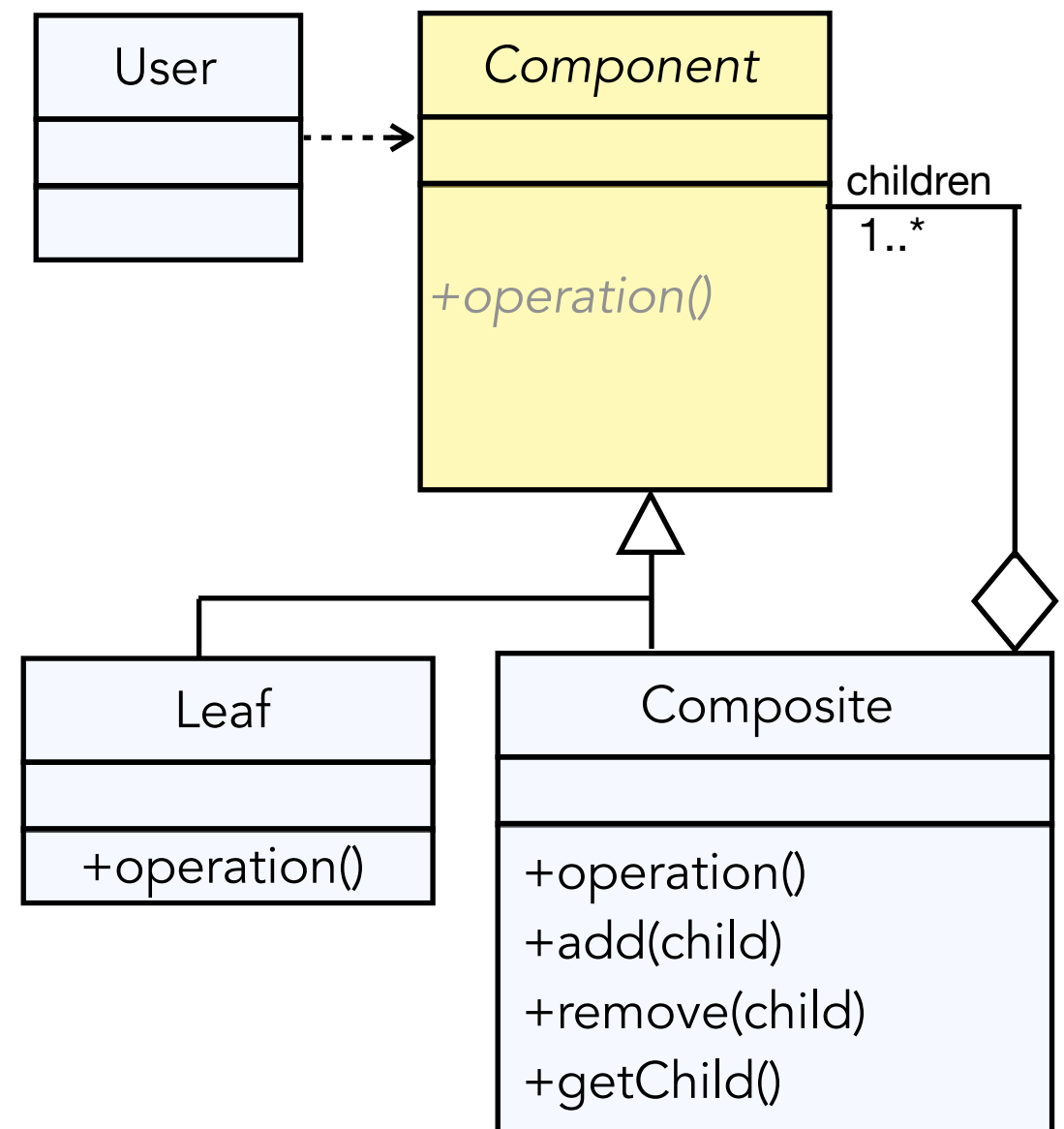
Design for Uniformity

# Composite Pattern

Uniform / TypeSafe



Design for Uniformity



Design for Type Safety

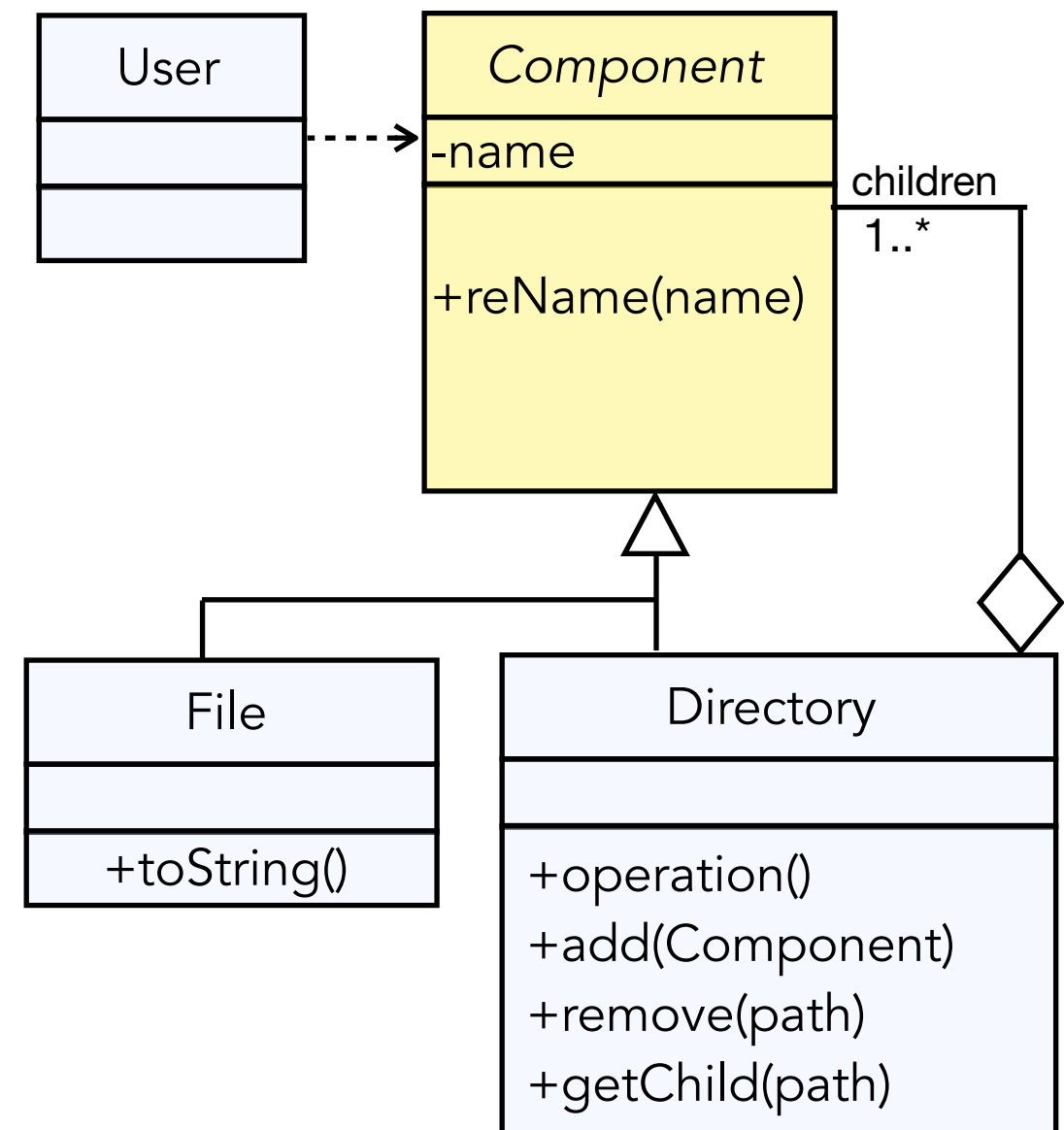
# Composite Pattern

## 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



# Structural Patterns

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

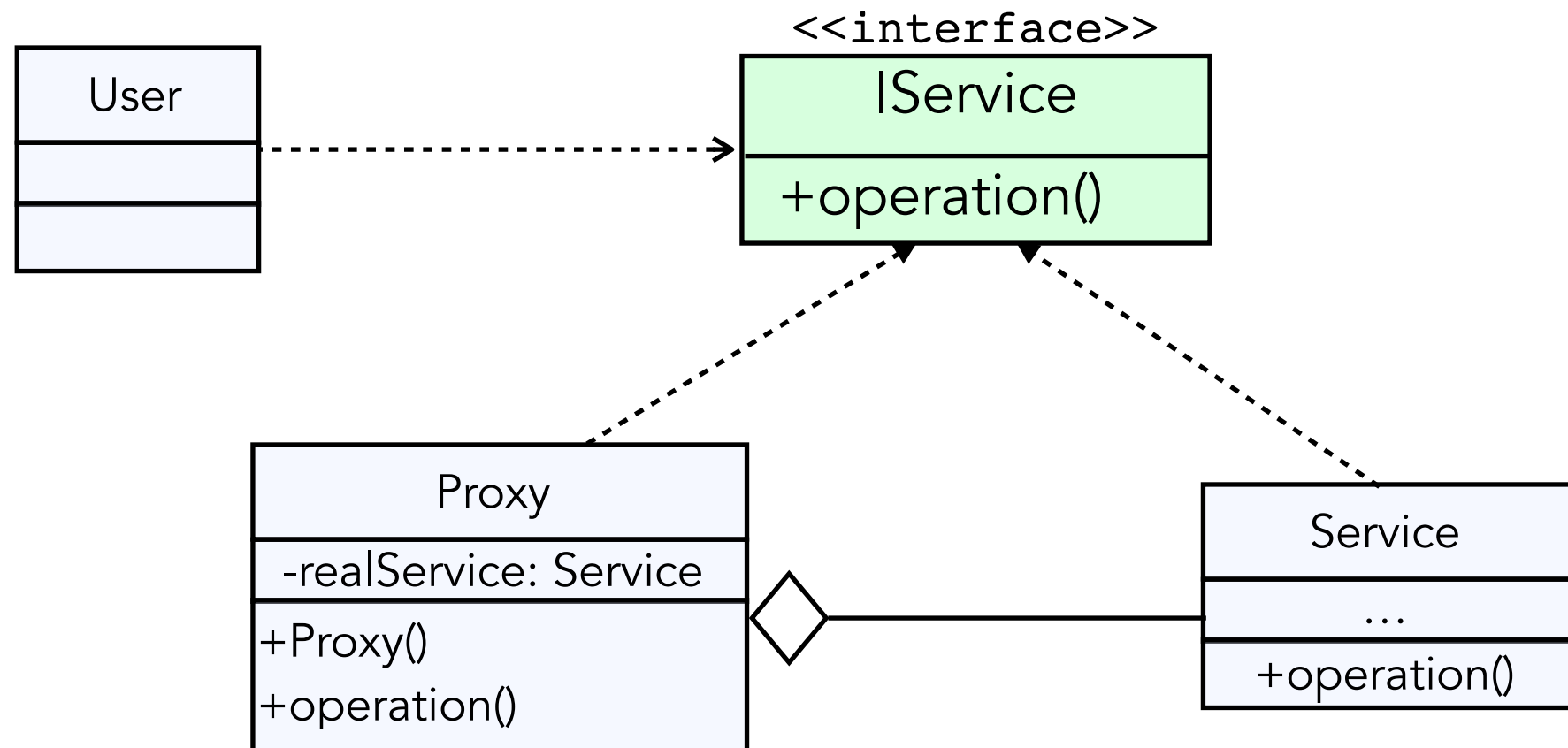
# Structural Patterns

## Proxy Pattern / Le Patron Proxy

- **Le Proxy** est un pattern de conception structurel qui permet de fournir un substitut ou un espace réservé pour un autre objet. Un proxy contrôle l'accès à l'objet d'origine, vous permettant d'effectuer un traitement avant ou après qu'une demande parvienne à l'objet d'origine.
- Principes SOLID :
  - **Single Responsibility** : Le Proxy est chargé d'une mission particulière
  - **Liskov Substitution** : Le Proxy remplace le rôle d'un objet de façon transparente

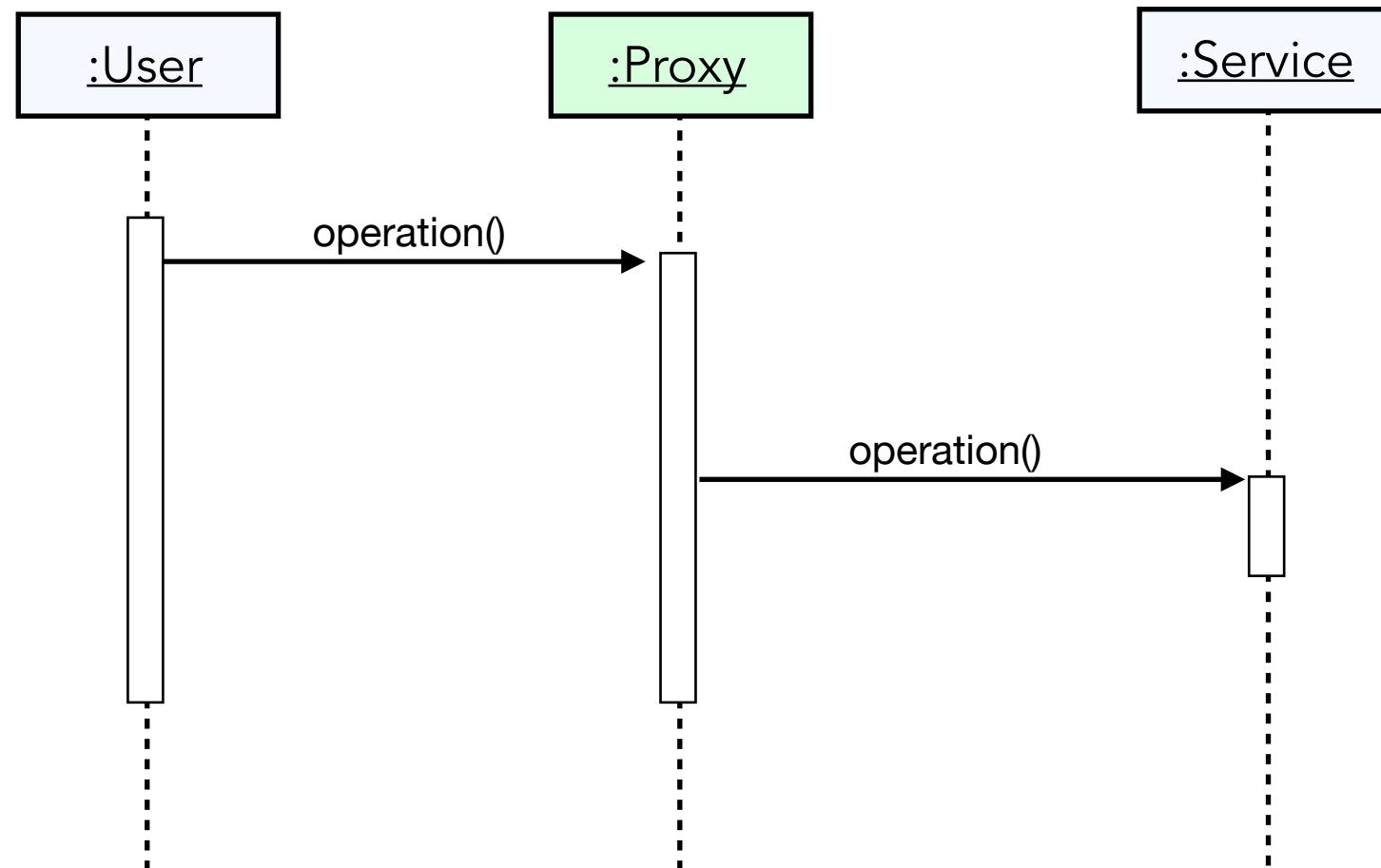
# Proxy Pattern

## UML



# Proxy Pattern

## UML



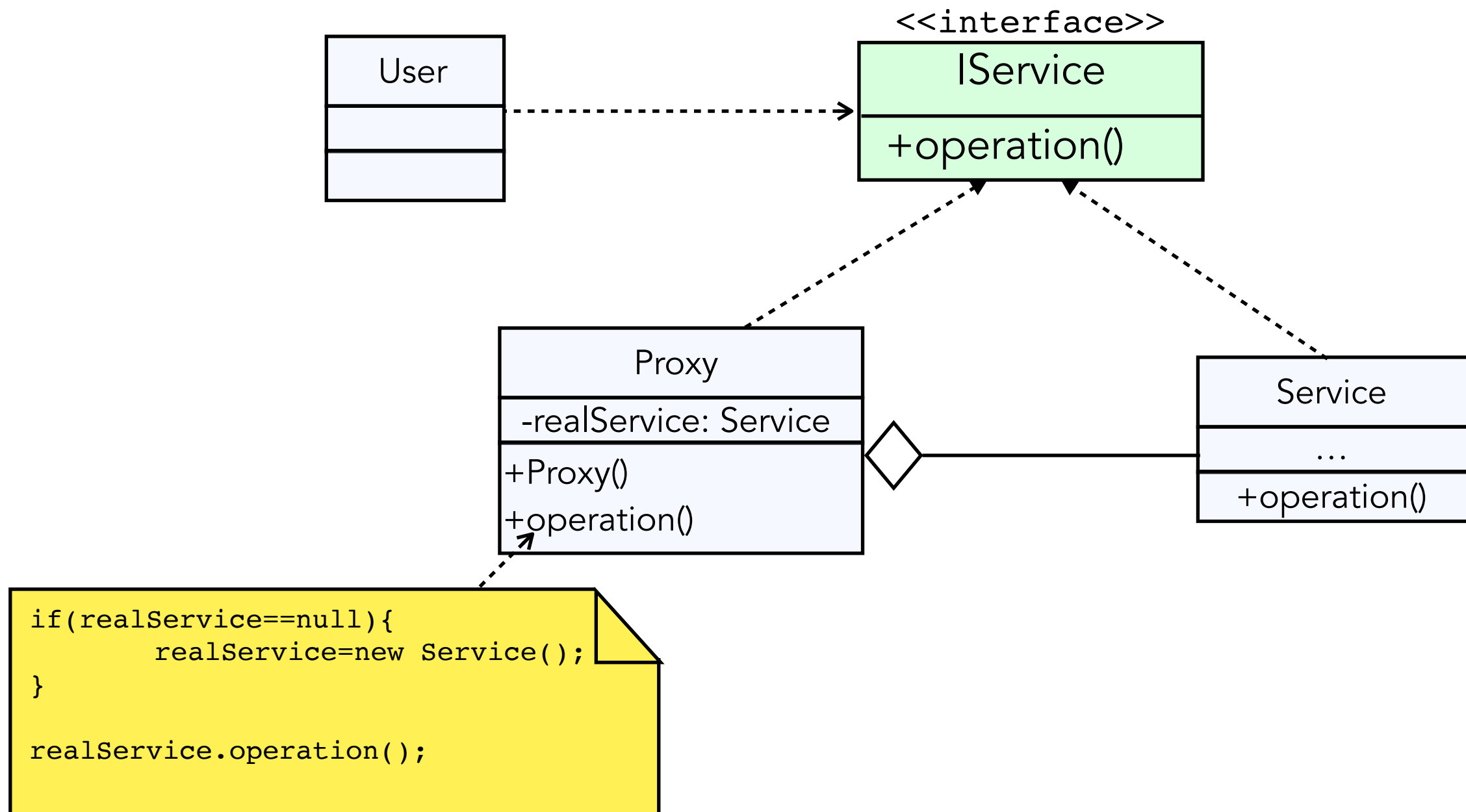
# Proxy Pattern

## Virtual Proxy (Lazy initialization)

- Un virtual Proxy est utile quand on a un objet de service lourd et qui gaspille des ressources système en étant toujours actif, même si vous n'en avez besoin que de temps en temps.

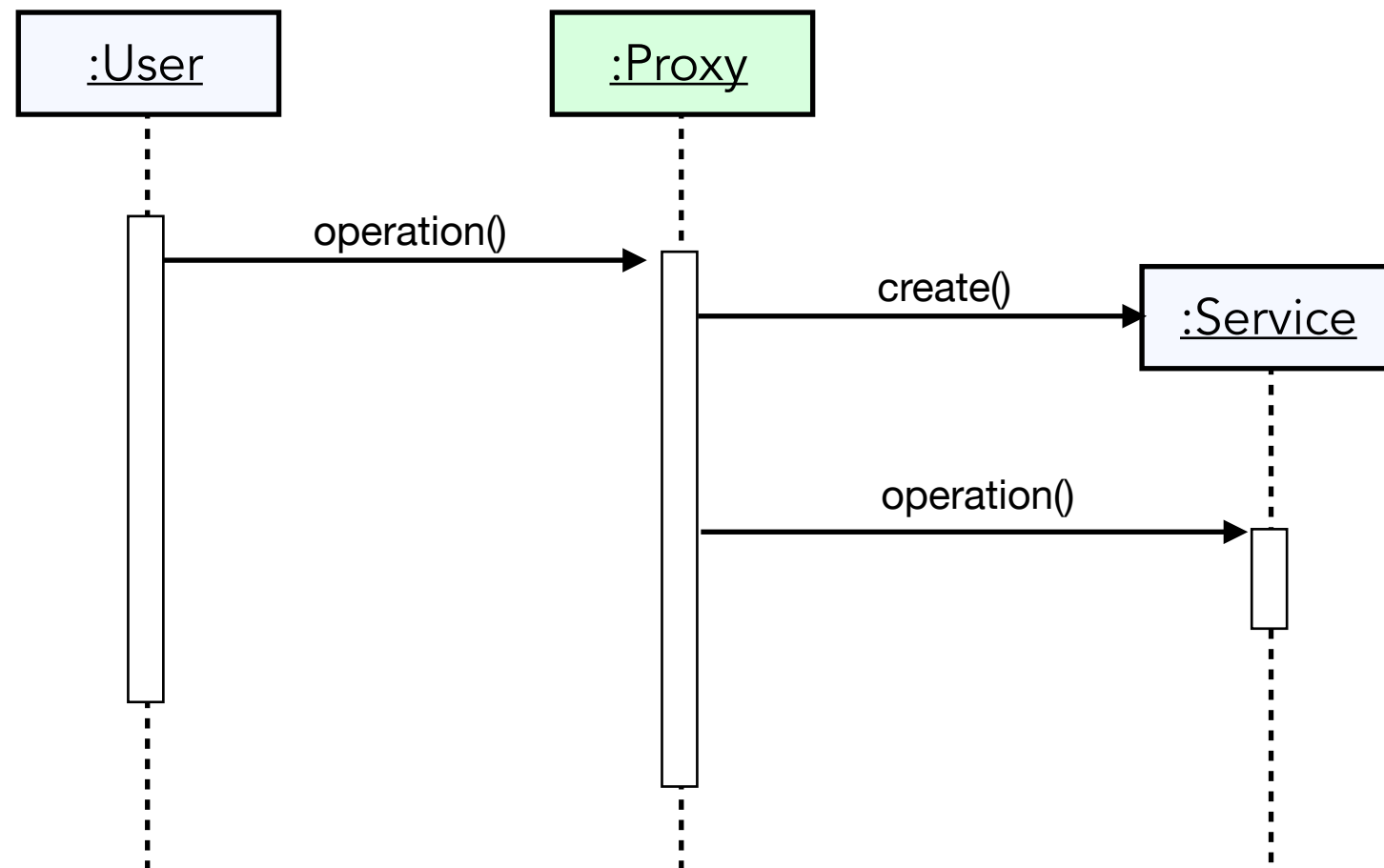
# Proxy Pattern

## Virtual Proxy (Lazy initialization)



# Proxy Pattern

## Virtual Proxy (Lazy initialization)



# Proxy Pattern

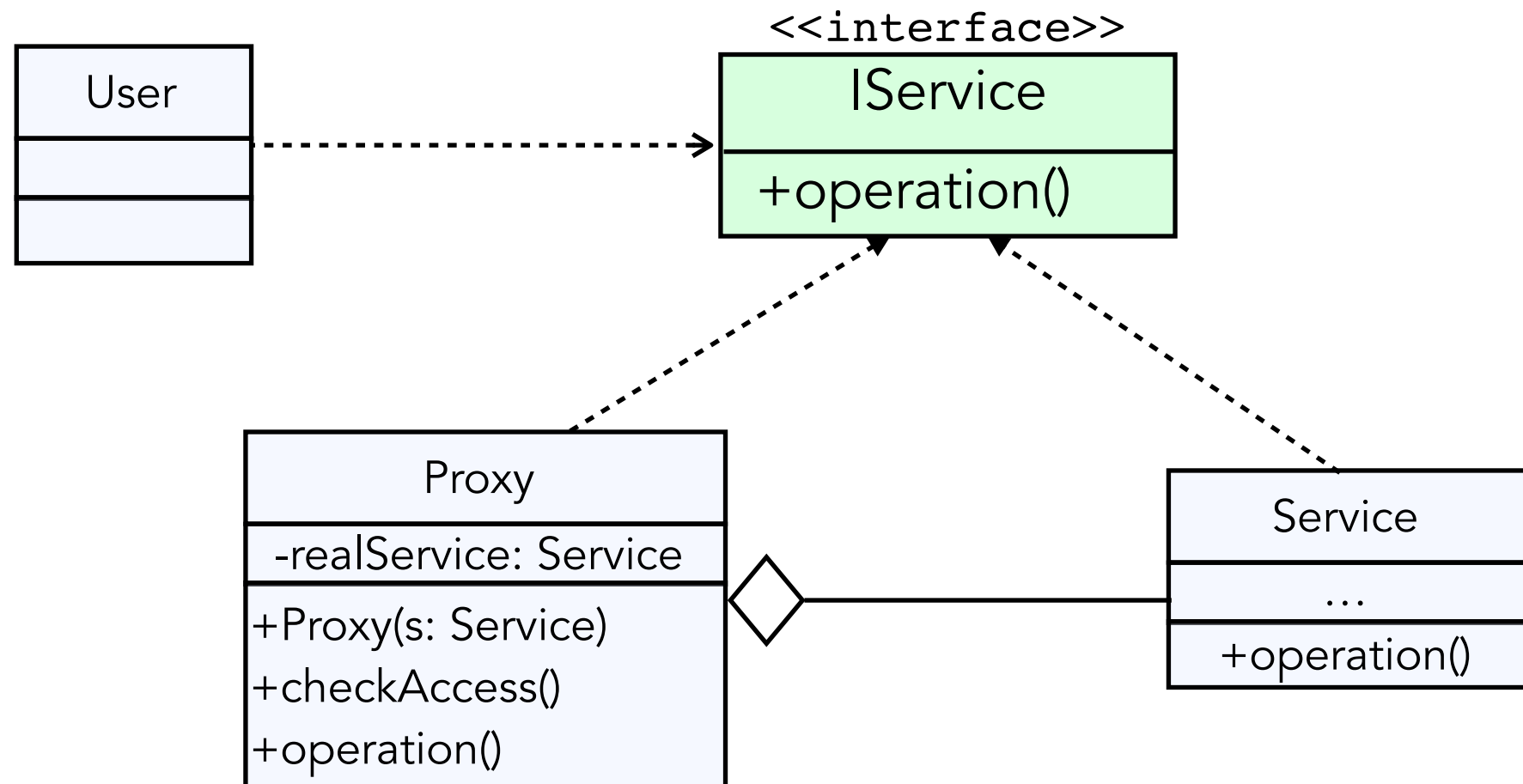
## Protection Proxy

- **Un Proxy de protection** permet de contrôler l'accès à l'objet d'origine. Utile quand on a des droits d'accès différents. Par exemple, les KernelProxies d'un système d'exploitation.



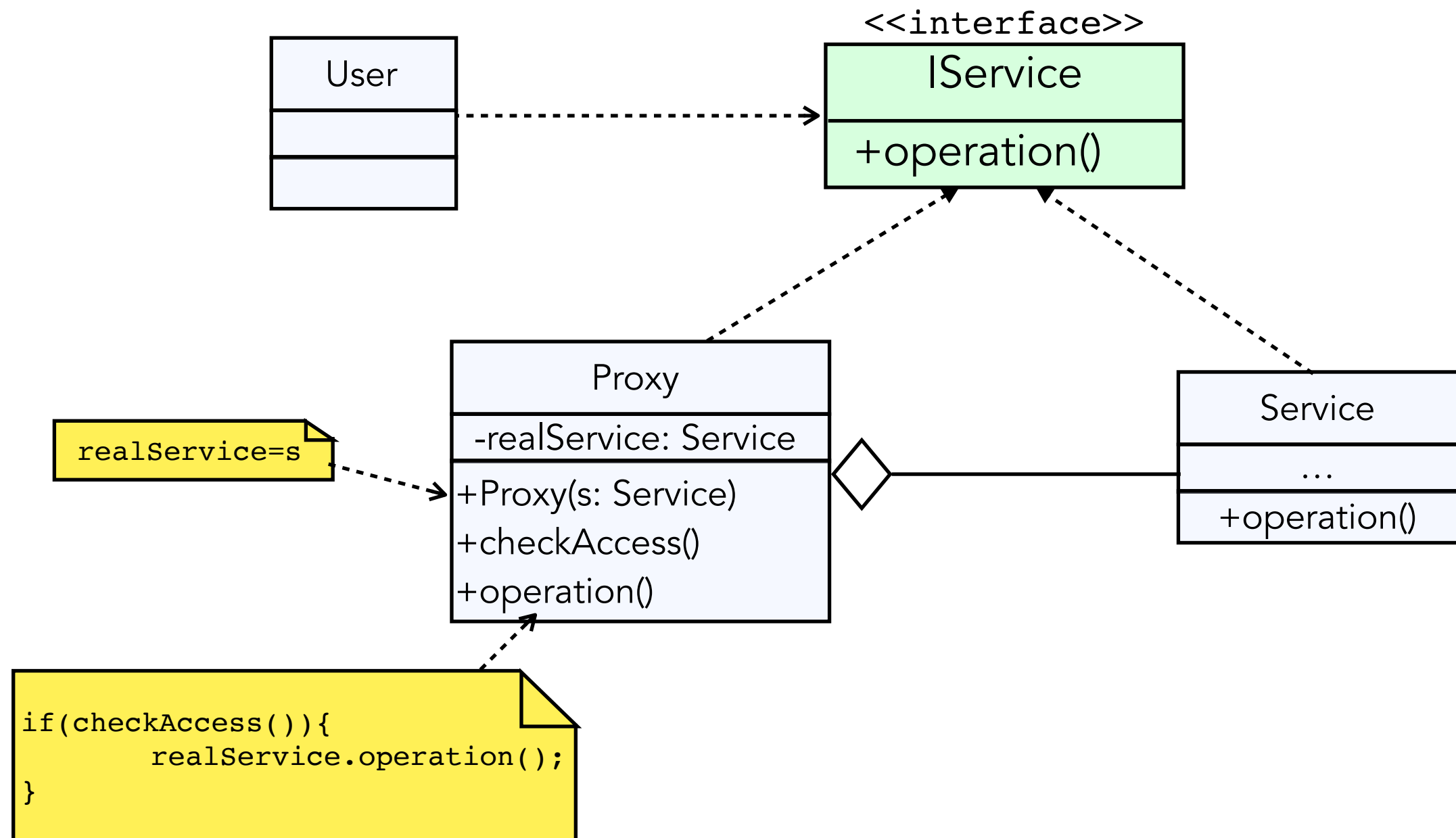
# Proxy Pattern

## Protection Proxy (UML)



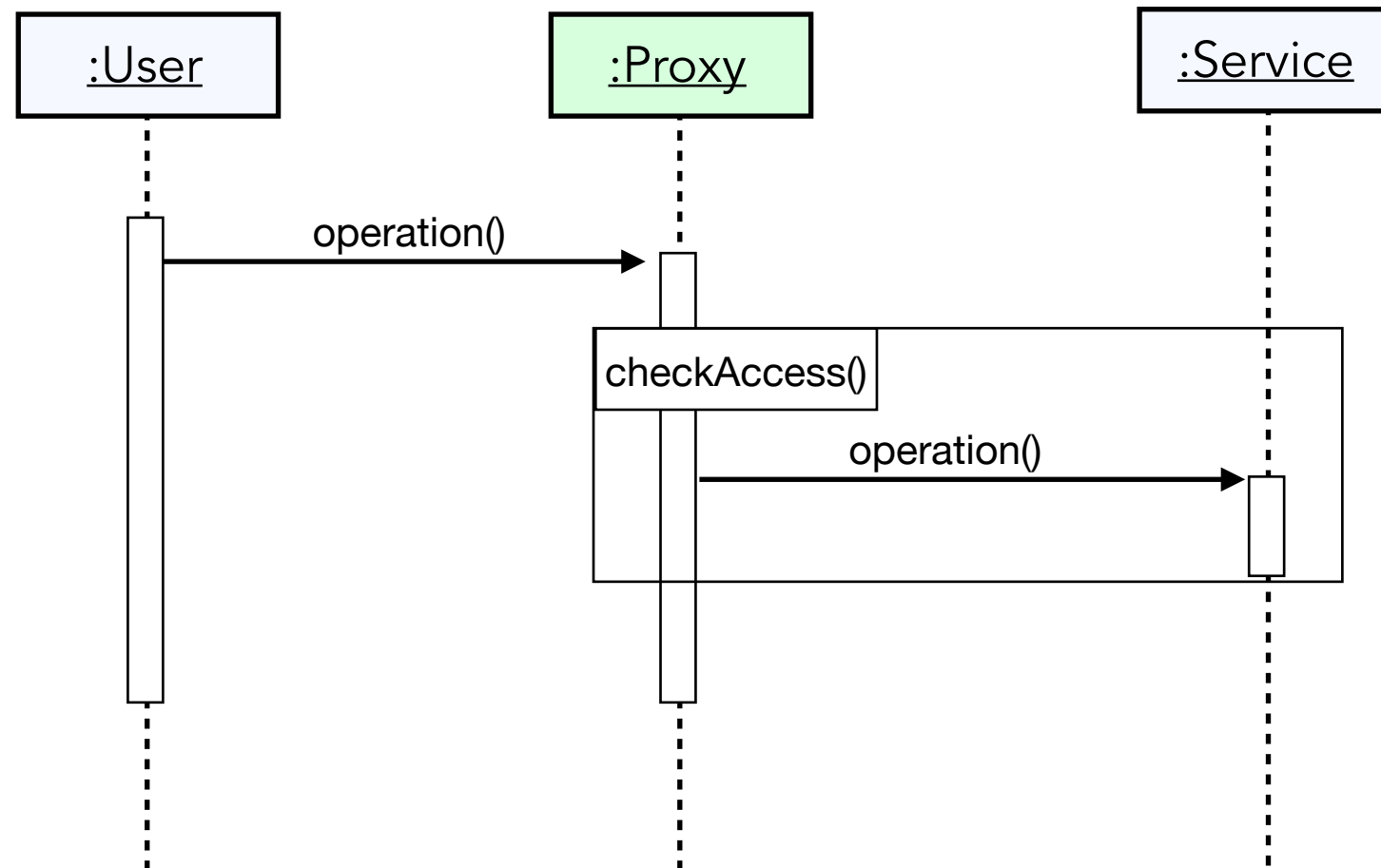
# Proxy Pattern

## Protection Proxy (UML)



# Proxy Pattern

## Protection Proxy (UML)



# Proxy Pattern

## Also

- **Remote Proxy:** Un proxy distant permet d'avoir un représentant local d'un objet qui est dans un serveur distant. Dans ce cas, le proxy transmet la demande du client sur le réseau, gérant tous les détails désagréables liés au travail avec le réseau.
- **Logging Proxy:** Utile quand on souhaite conserver un historique des demandes adressées à l'objet.
- **Caching Proxy:** Réutilisation de résultats déjà calculés au préalable. Souvent utilisé pour des opérations coûteuses ou lentes.

# Structural Patterns

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

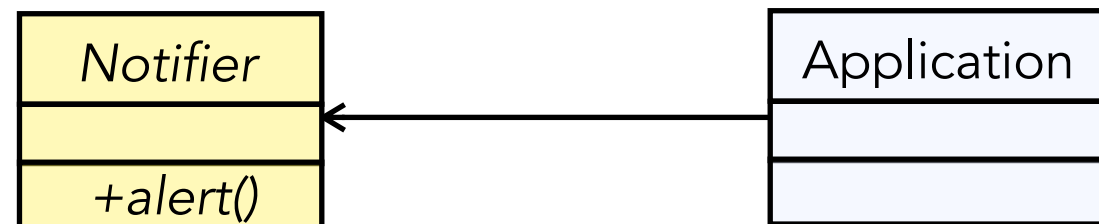
# Structural Patterns

## 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

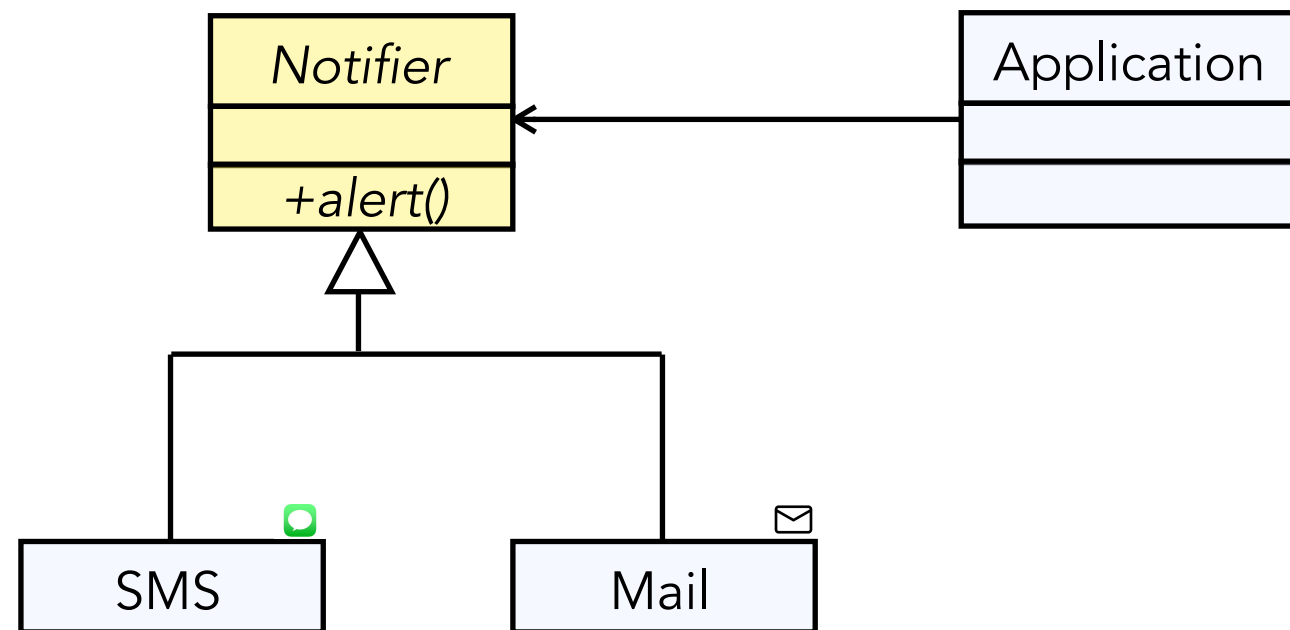
# Decorator

## Notifier example



# Decorator

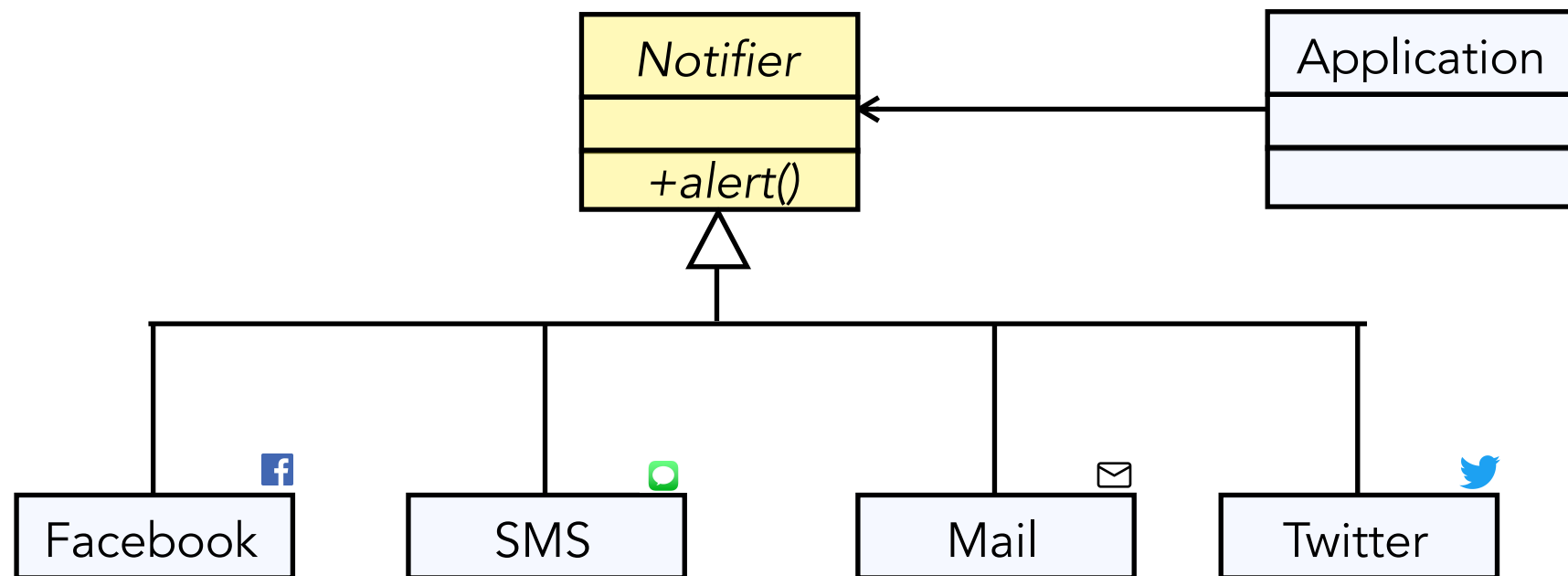
## Notifier example





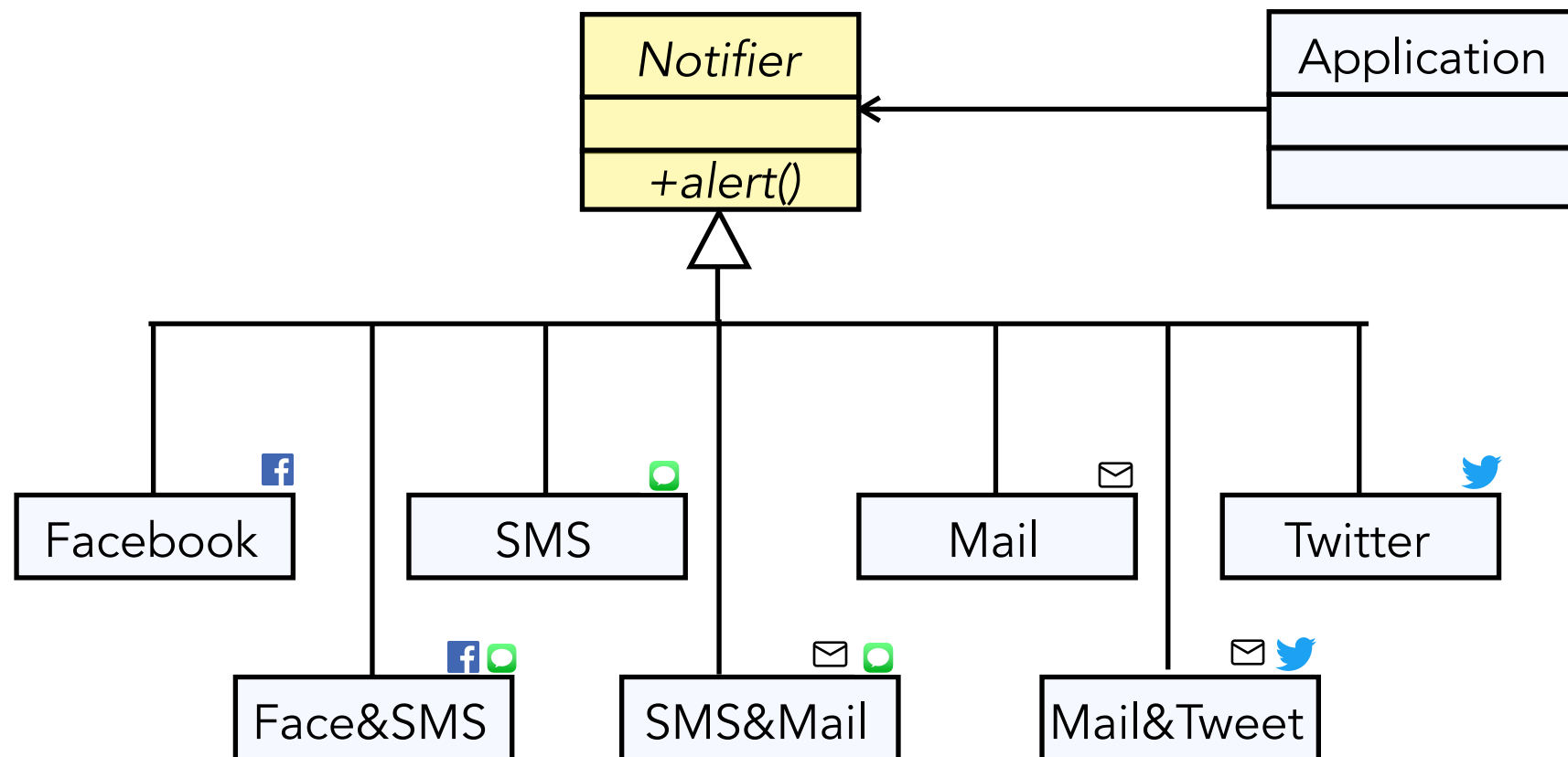
# Decorator

## Notifier example



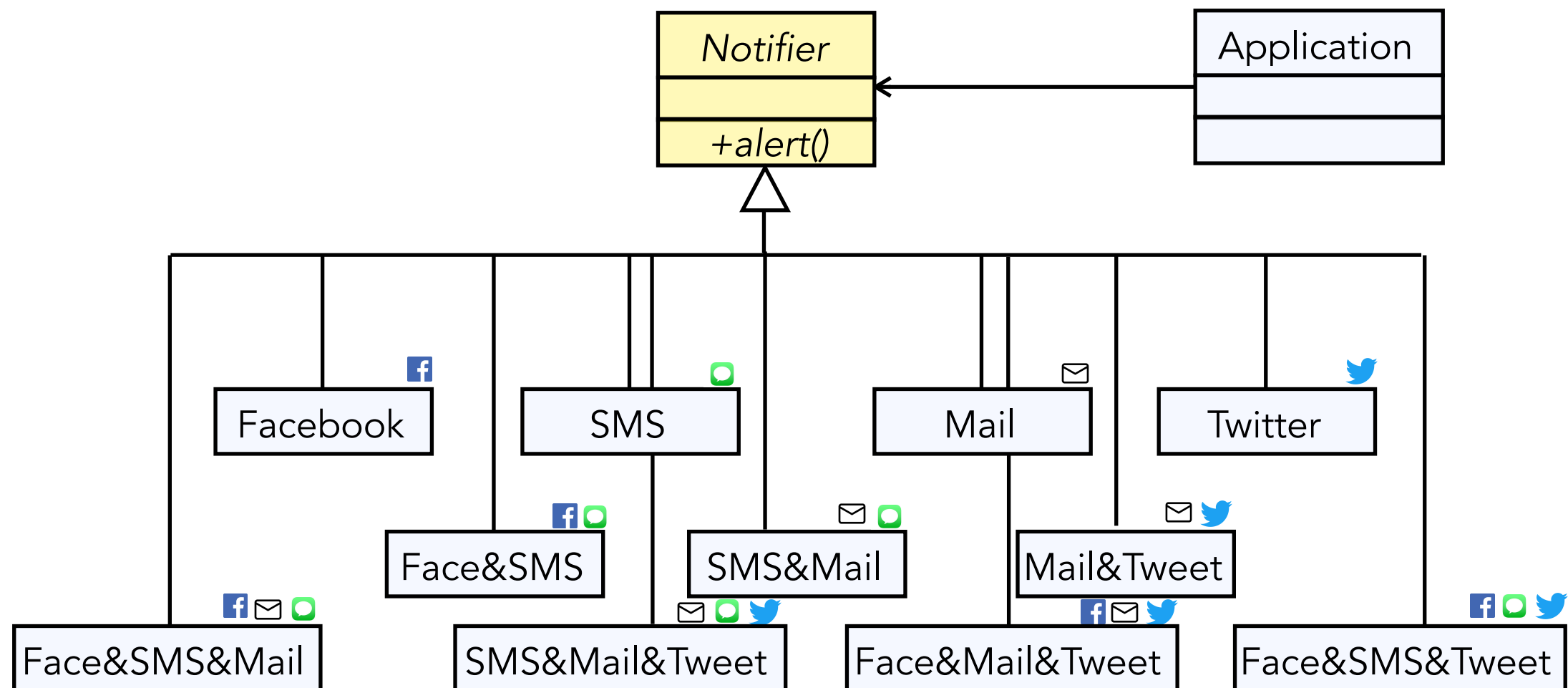
# Decorator

## Notifier example



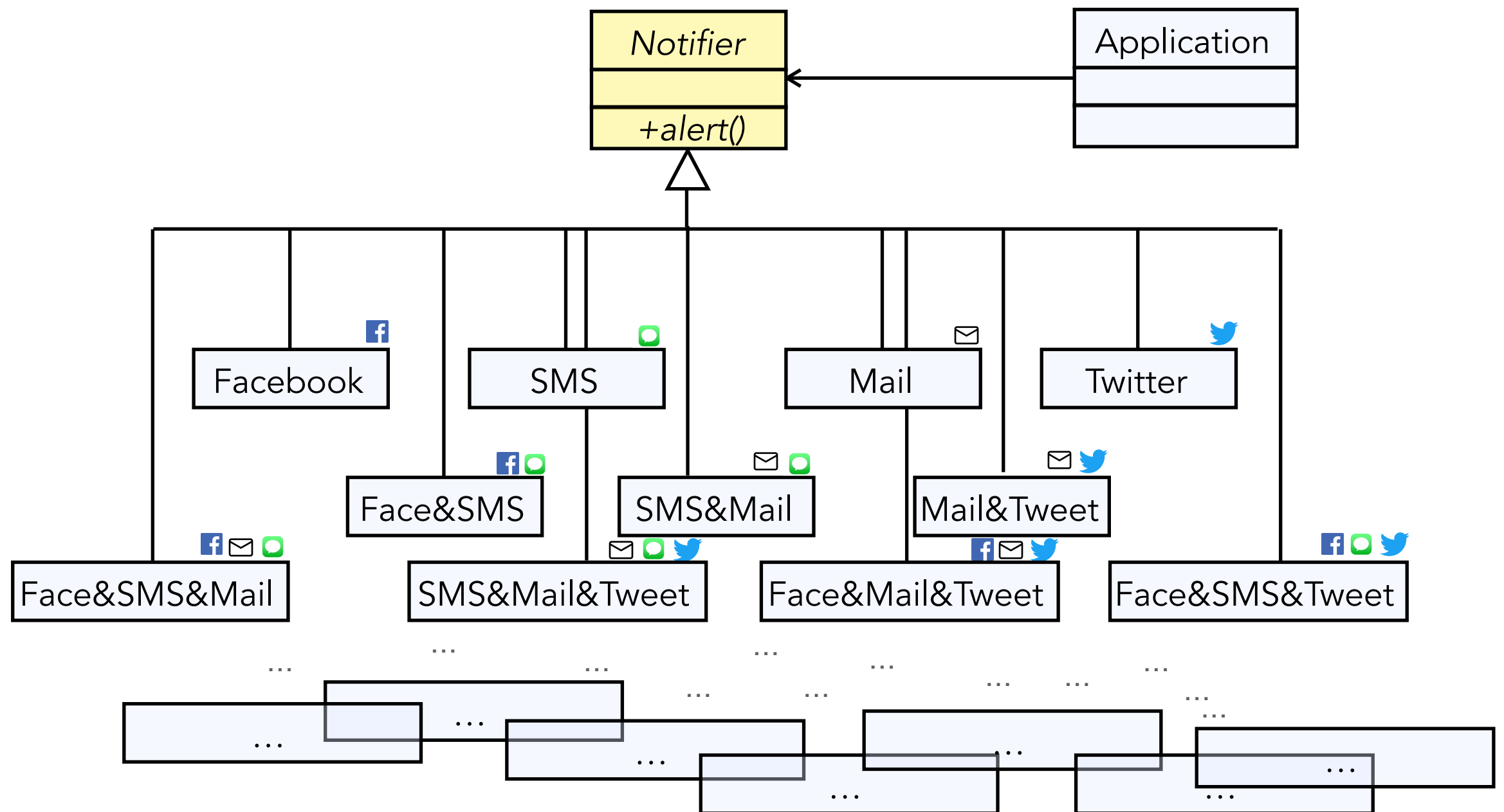
# Decorator

## Notifier example



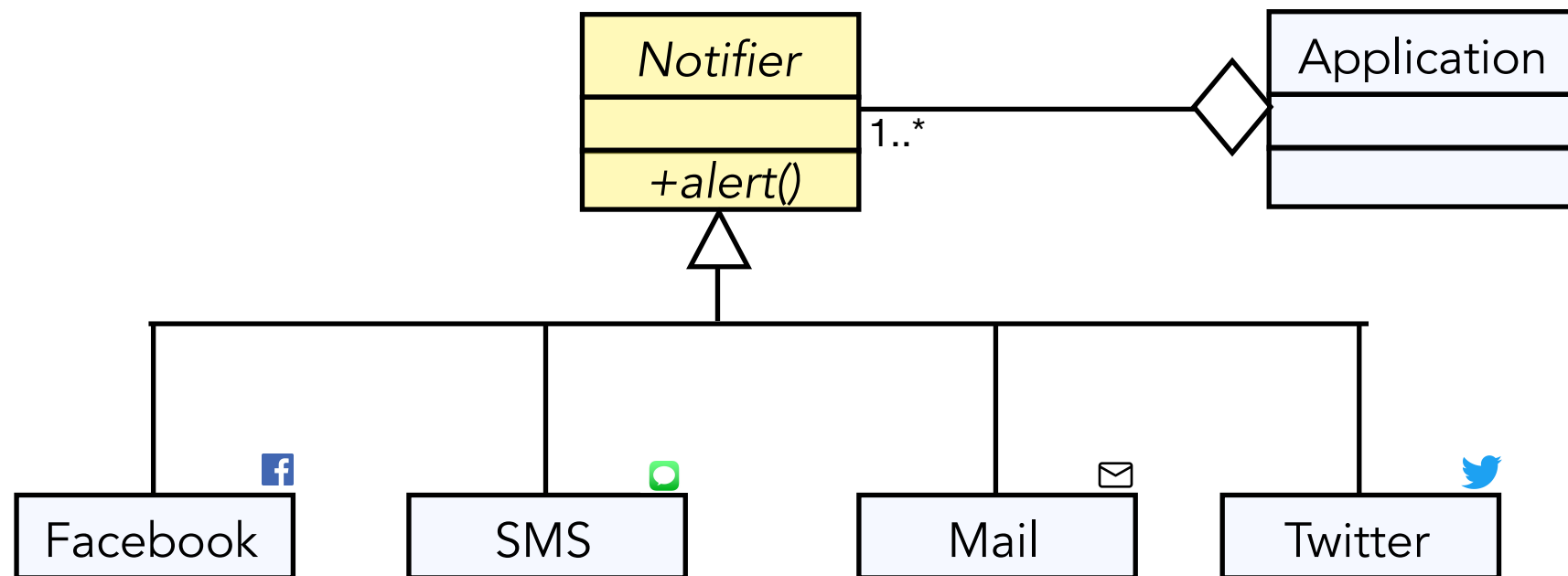
# Decorator

## Notifier example



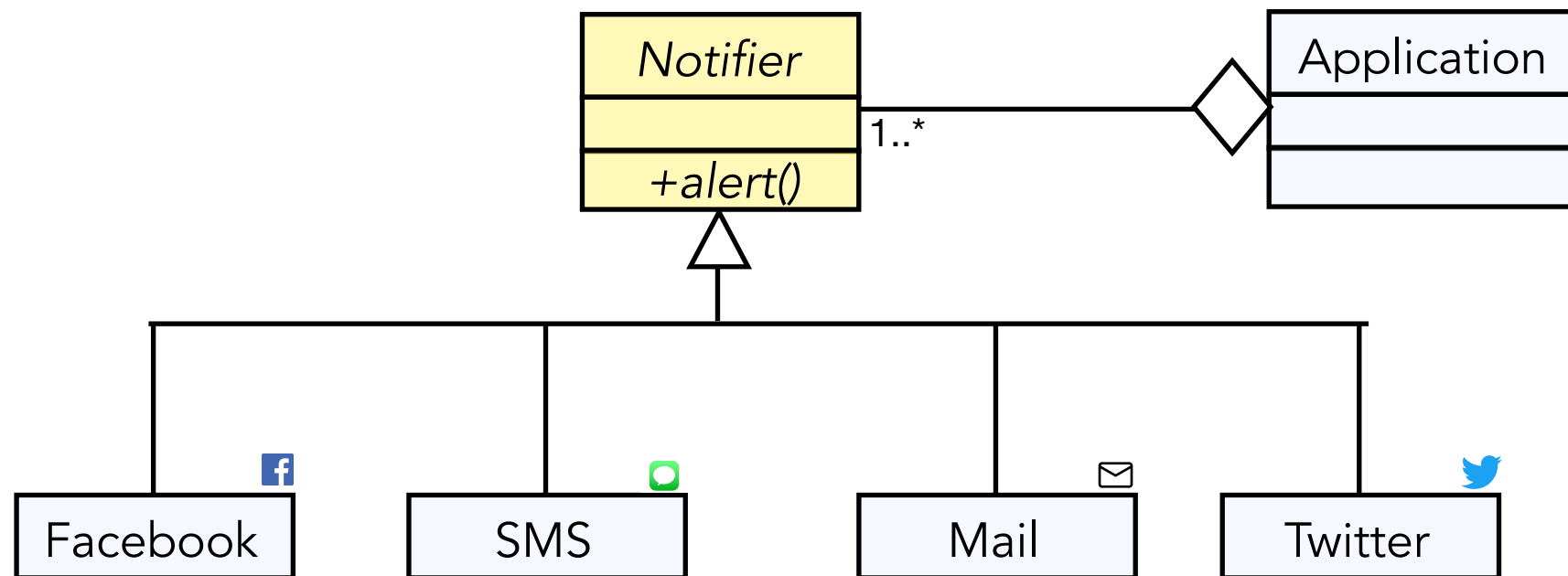
# Decorator



## Notifier example



# Decorator

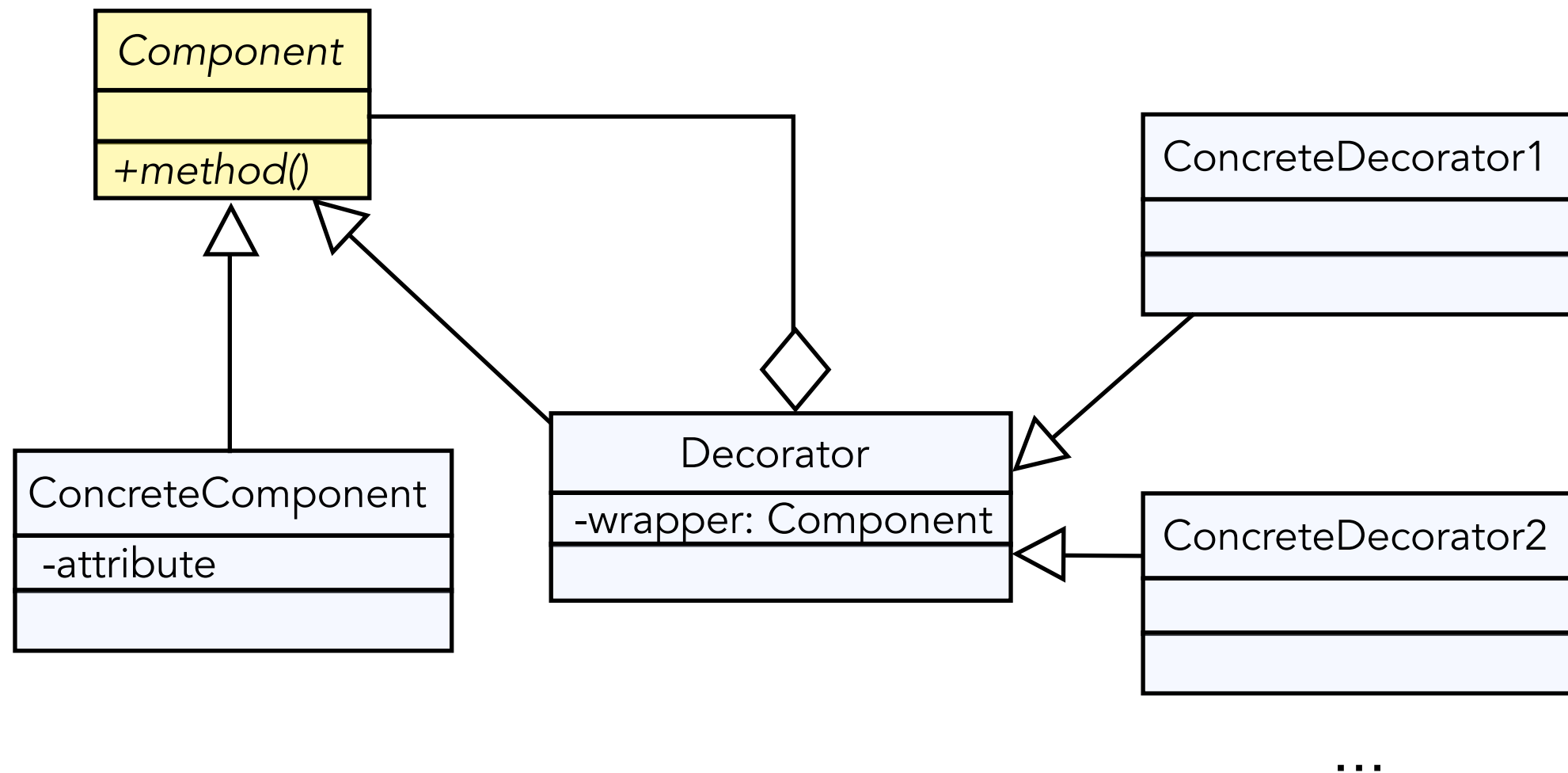
## Notifier example with an acceptable solution



Ex:    Face&SMS&Mail

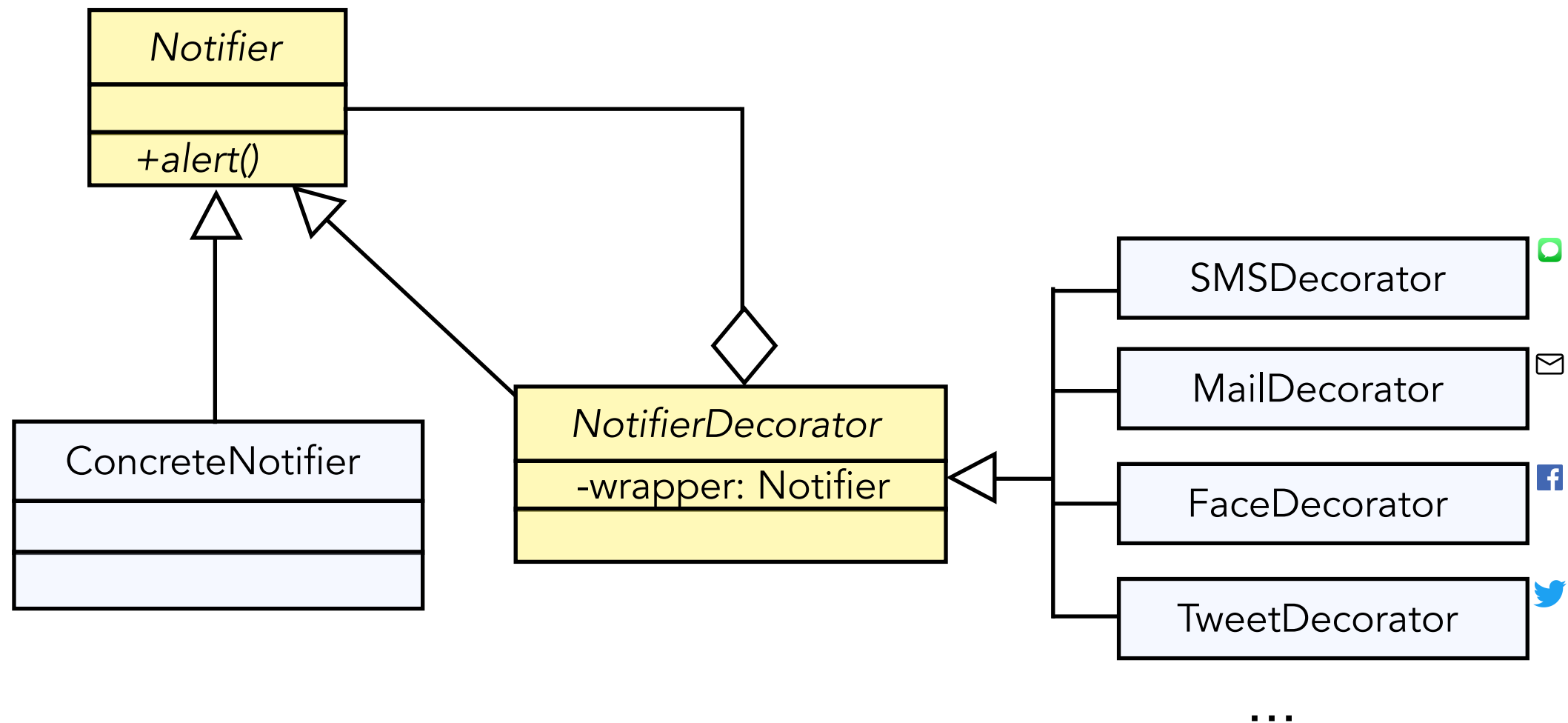
# Decorator

## UML



# Decorator

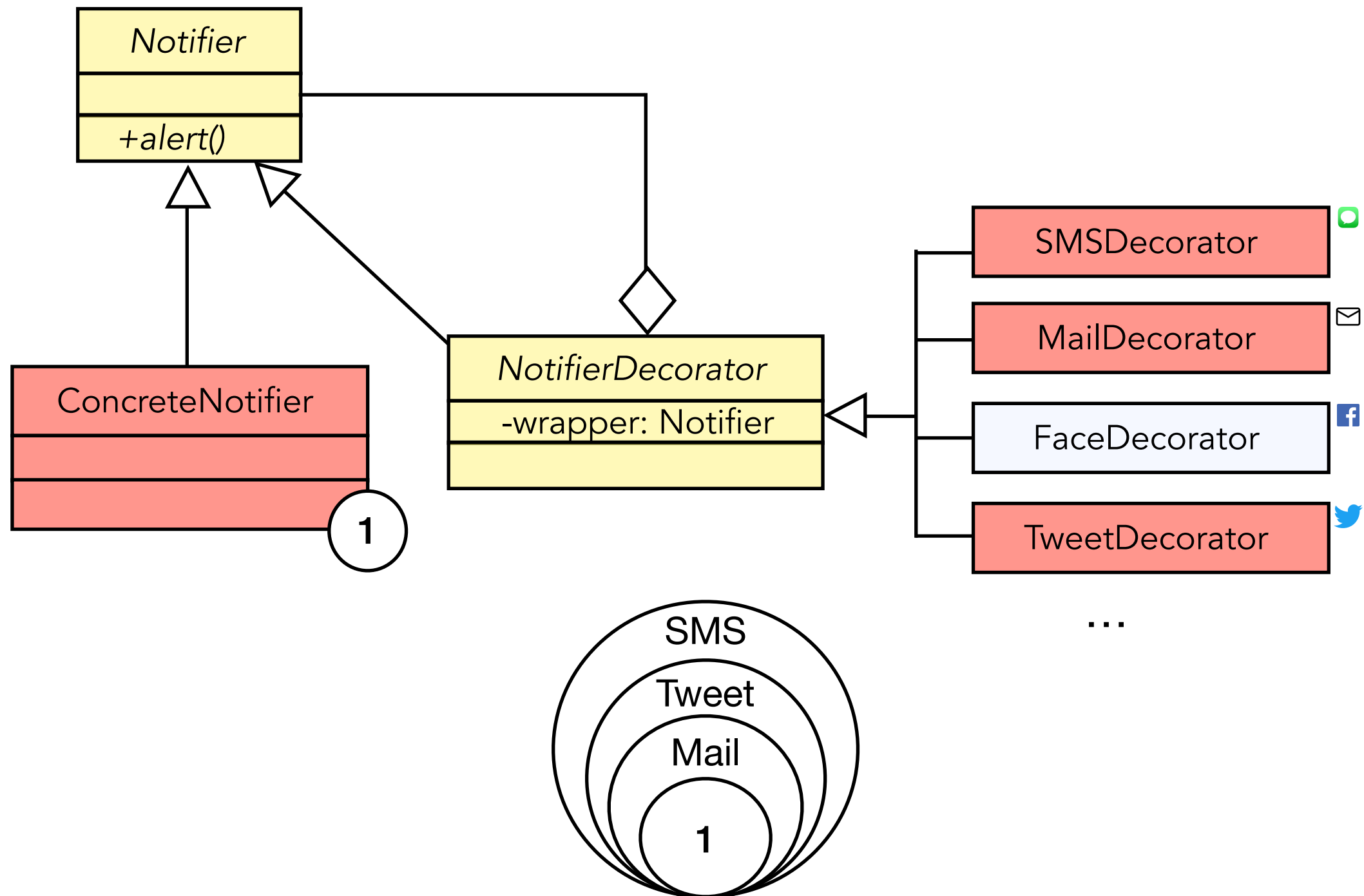
## Notifier example with Decorator Pattern (UML)





# Decorator

## Notifier example with Decorator Pattern (UML)



# Decorator

## Notifier example with Decorator Pattern (JAVA)

```
public abstract class Notifier {  
    abstract String alert();  
}
```

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

# Decorator

## Notifier example with Decorator Pattern (JAVA)

```
public abstract class Notifier {  
    abstract String alert();  
}
```

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

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

# Decorator

## Notifier example with Decorator Pattern (JAVA)

```
public abstract class Notifier {  
    abstract String alert();  
}
```

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

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

```
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);  
    }  
}
```

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

# Decorator

## Notifier example with Decorator Pattern (JAVA)

```
public abstract class Notifier {  
    abstract String alert();  
}
```

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

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

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

```
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);  
    }  
}
```

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

# 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

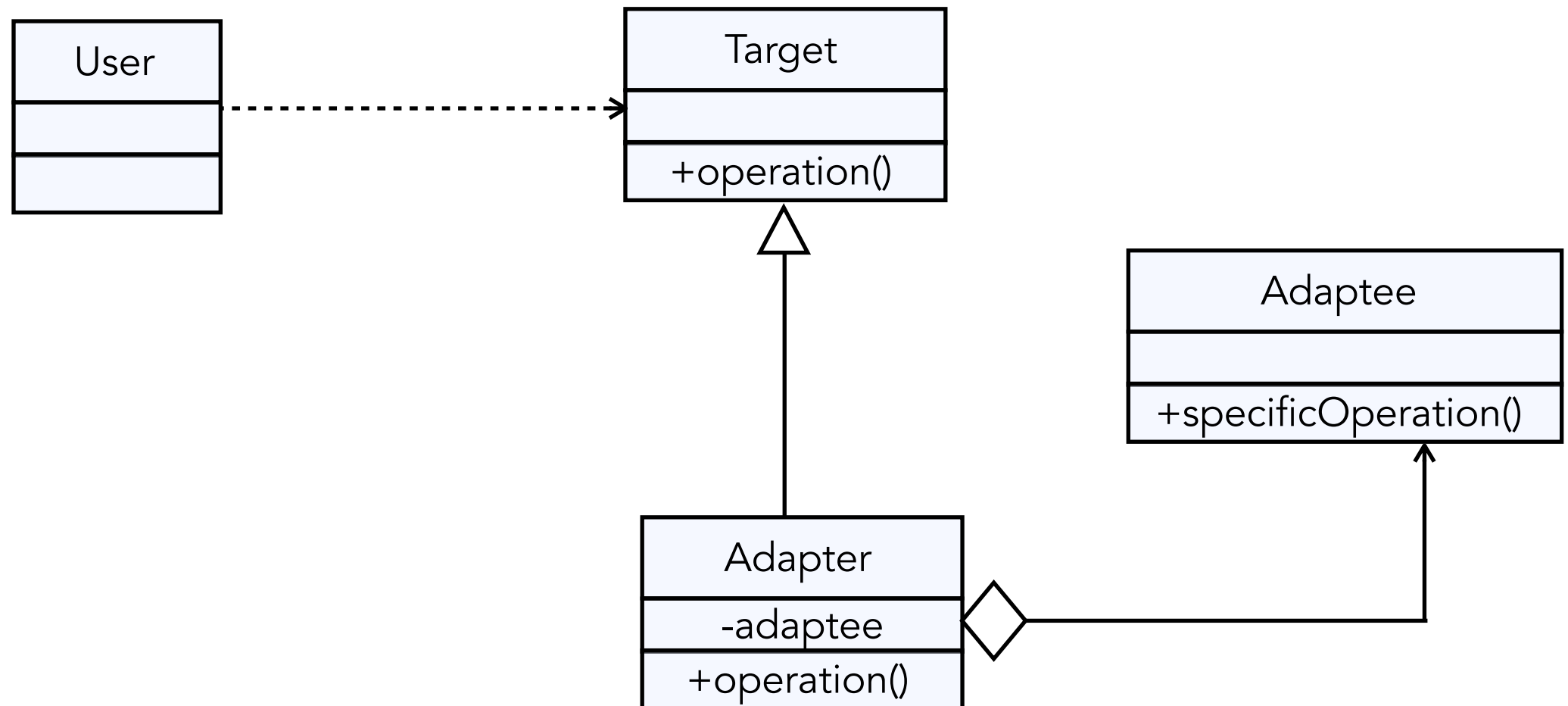
Scanner
+scan(img: Jpg): Xml

Printer
+print(paper: Pdf): Txt



# Adapter Pattern

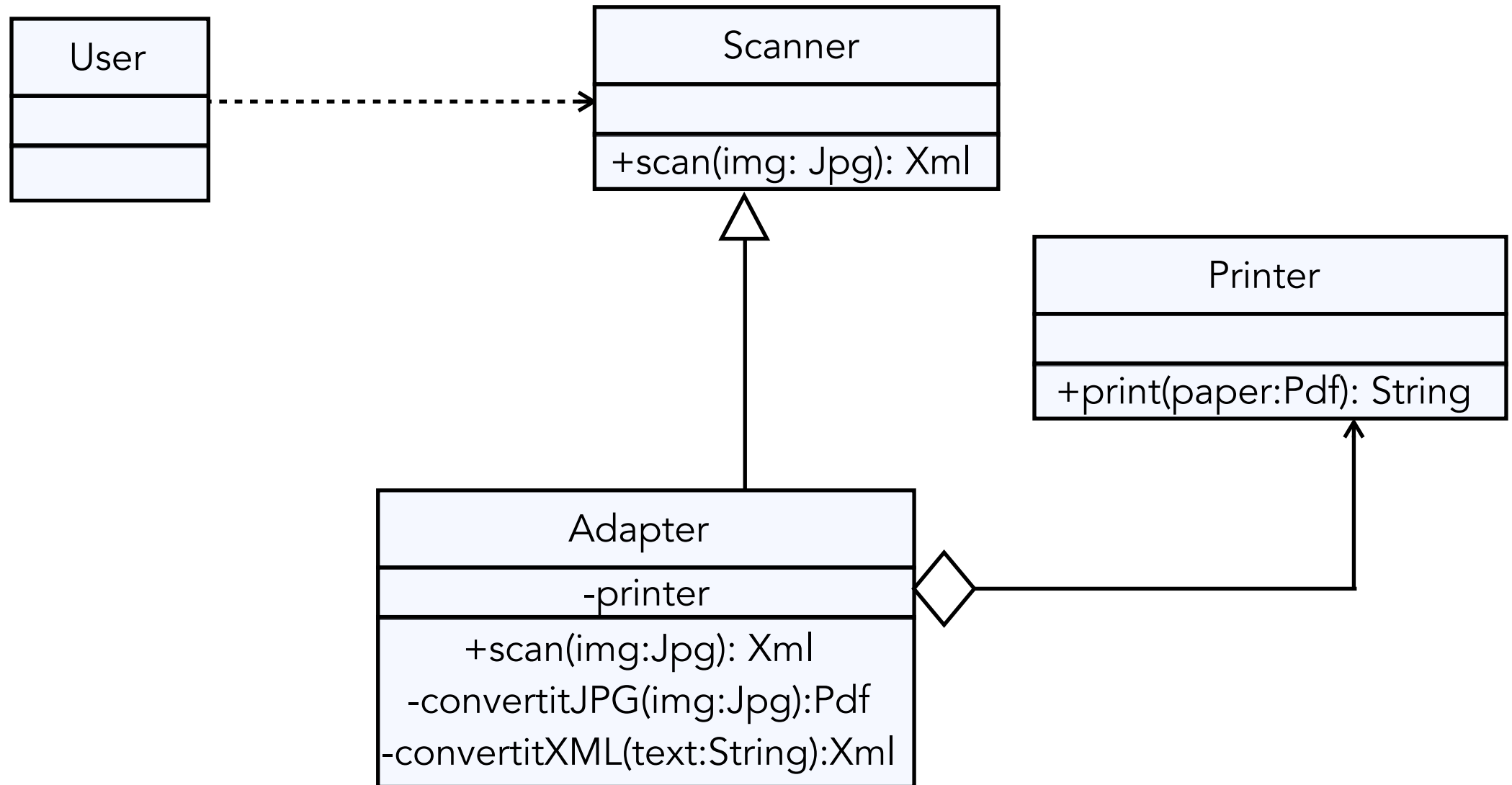
## Object Adapter (UML)



- 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)



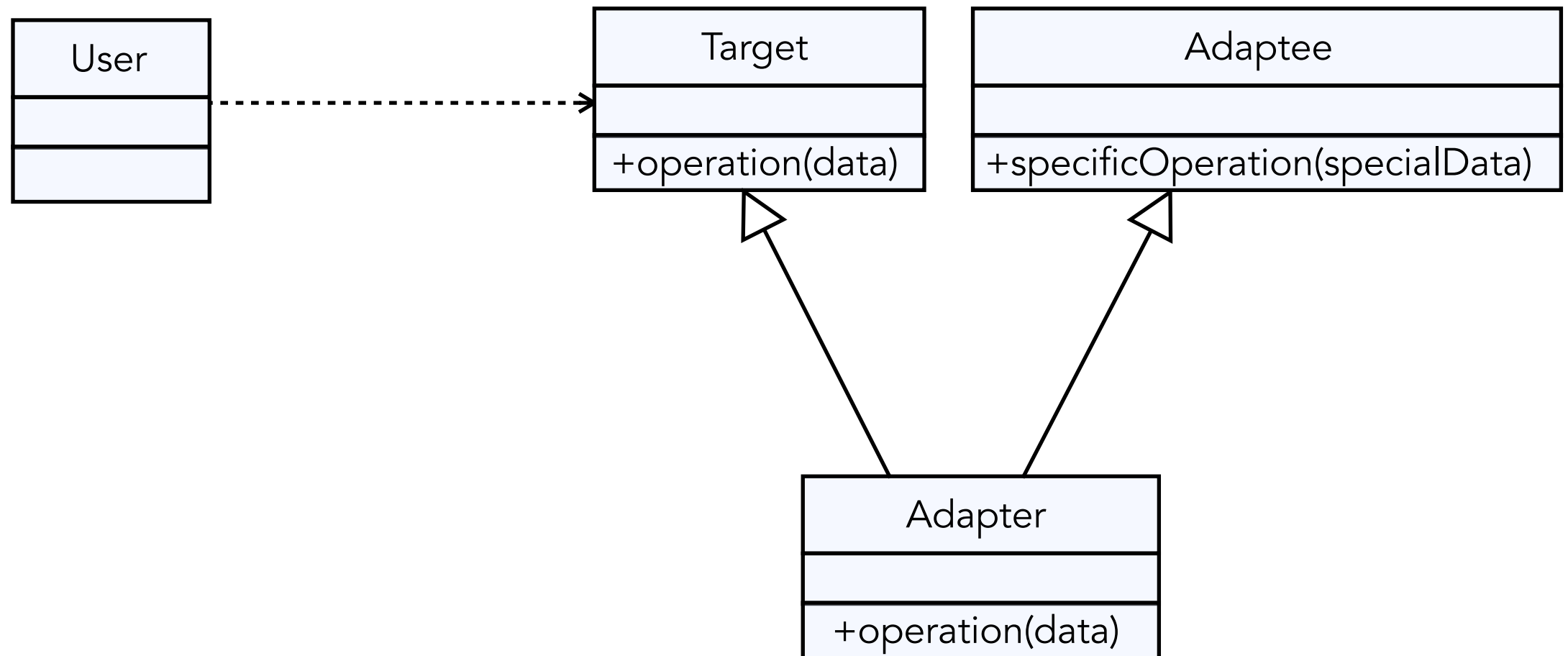
# Adapter Pattern

## 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

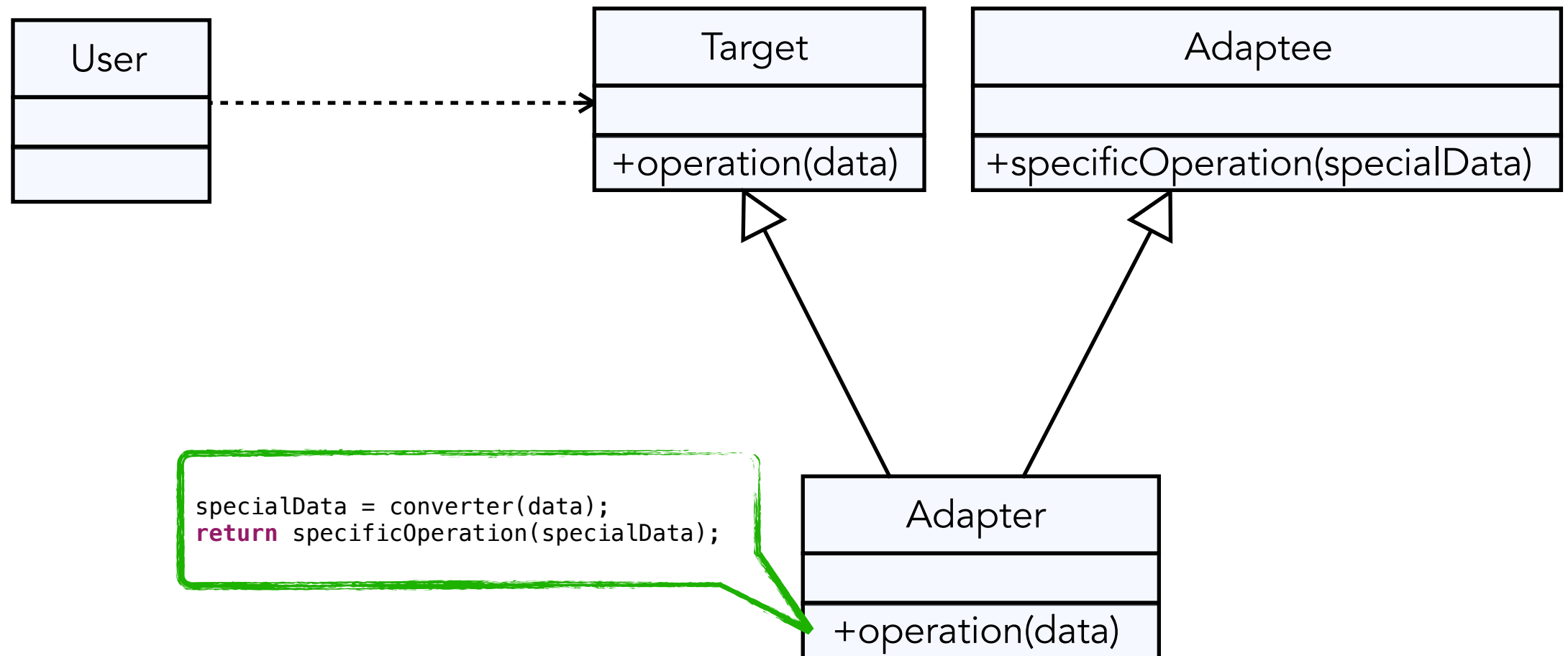
## Class Adapter (UML)



- 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


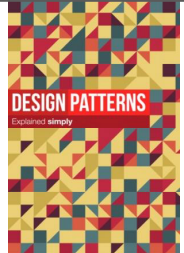

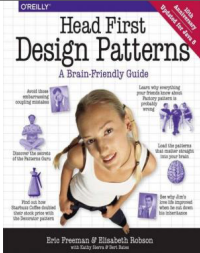
## Class Adapter (UML)



- 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++

# Books

## Design Pattern

<ul style="list-style-type: none"><li>• <b>Design Patterns: Elements of Reusable Object-Oriented Software.</b> Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides (GoF: Gang of Four).</li></ul>	
<ul style="list-style-type: none"><li>• <b>DESIGN PATTERNS Explained simply.</b> Alexander Shvets. 2013</li></ul>	
<ul style="list-style-type: none"><li>• <b>Dive Into Design Patterns.</b> Alexander Shvets. 2019</li></ul>	
<ul style="list-style-type: none"><li>• <b>Head First Design Patterns.</b> Freeman et al. 2014</li></ul>	
<ul style="list-style-type: none"><li>• <b>Java Design Patterns.</b> Vaskaran Sarcar. 2019</li></ul>	