Génie logiciel

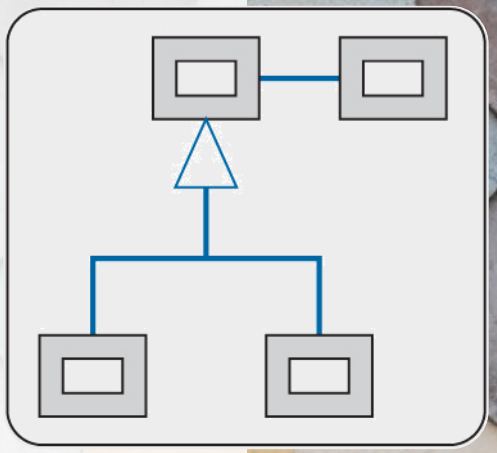
Patrons de conception

Louis-Edouard LAFONTANT

Patrons de conception

Définition: Schéma générique d'une solution à un problème récurrent dans un contexte donné.

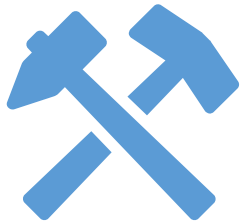
- Découlent d'expériences pratiques



Objectifs

Accroître la **qualité** du code en visant un ou plusieurs des objectifs suivant:

- ✓**Flexibilité** accrue
- ✓Meilleure **compréhension/performance**
- ✓**Fiabilité** accrue

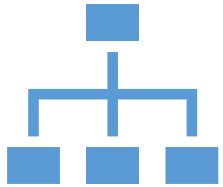


Patron de création

Instanciation des classes

Isoler la création du reste de l'application

Singleton, Fabrique

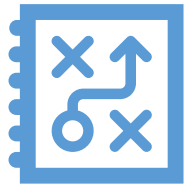


Patron de structure

Organisation des classes

Faciliter l'ajout de fonctionnalités

Adapteur, Composite



Patron de comportement

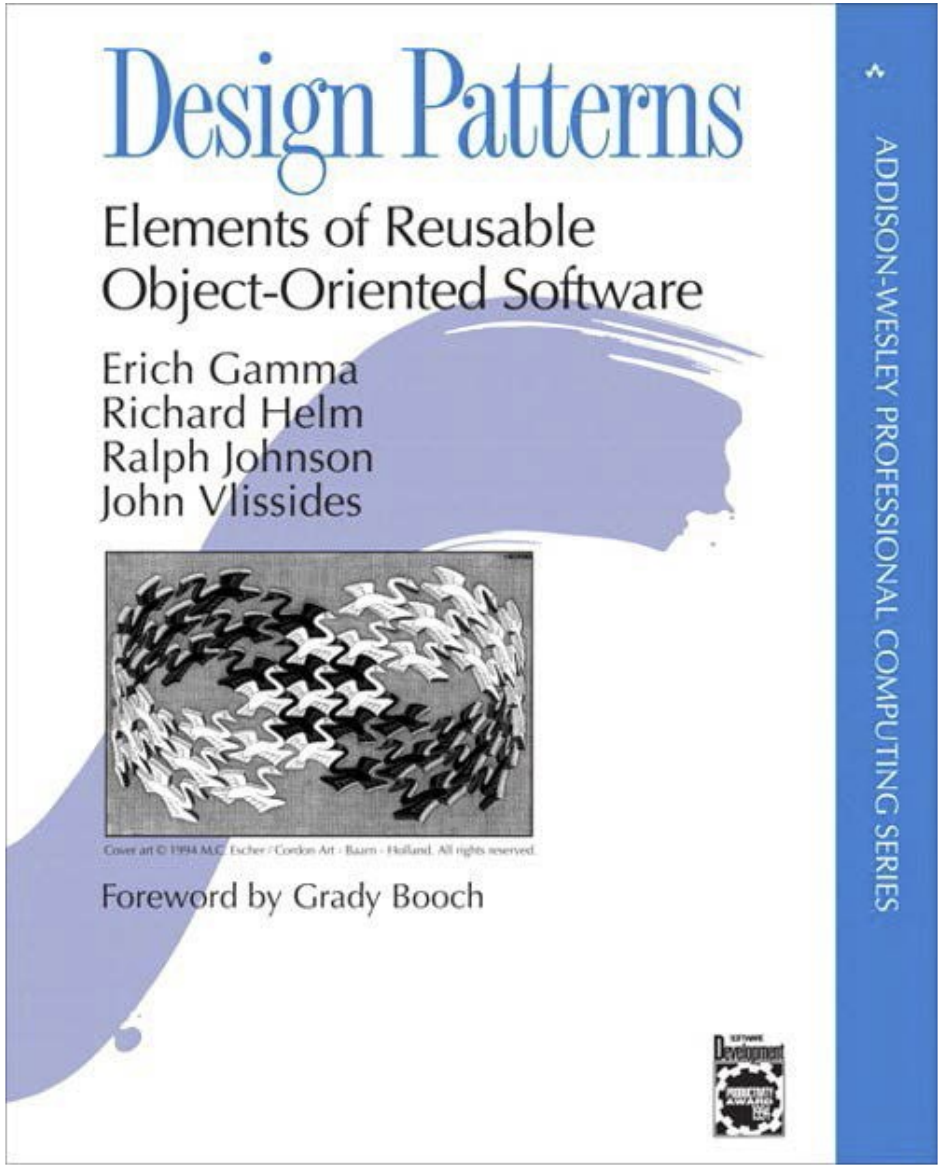
Collaboration entre objets

Partager la responsabilité d'une tache

Patron de méthode, Stratégie

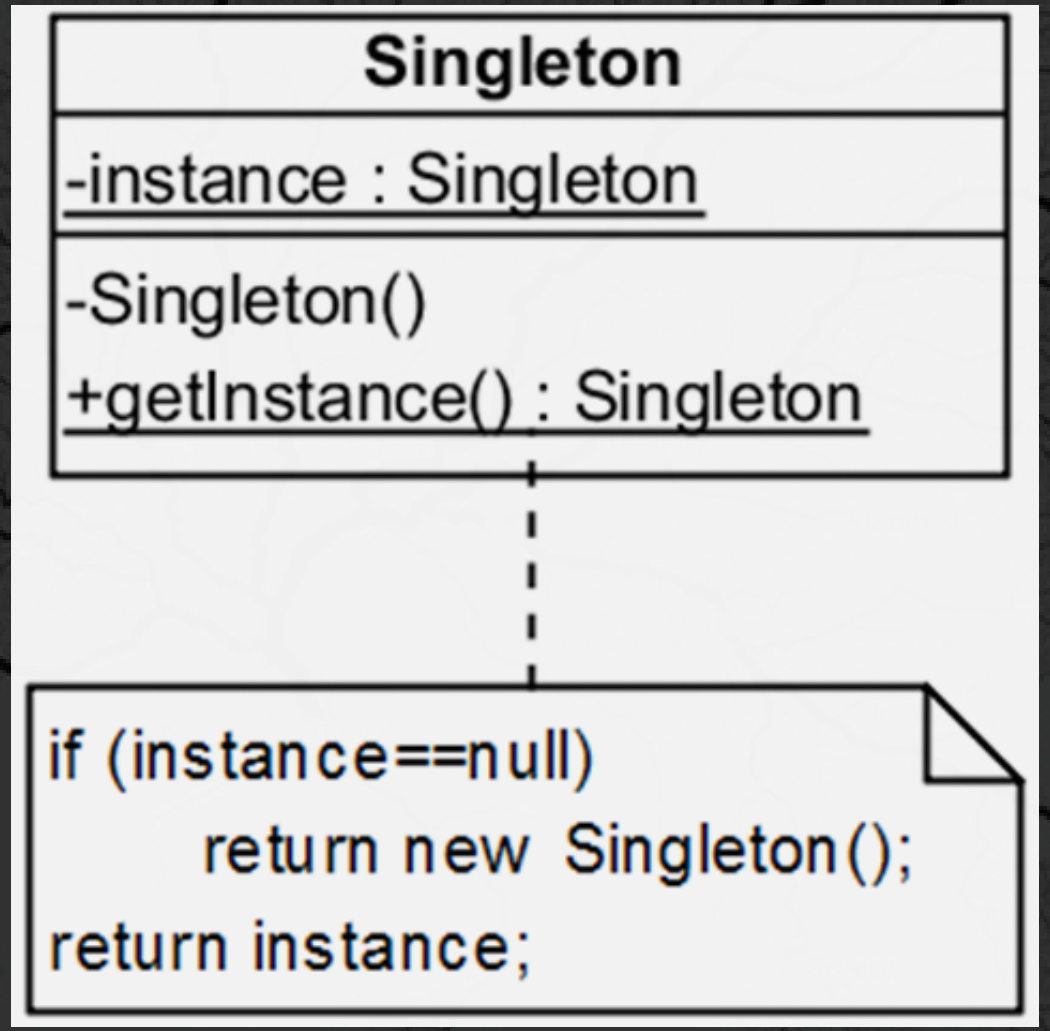
23 patrons de conception

- Livre du « Gang of Four »
- Couvert en détail dans le cours IFT3911



Singleton

Patron de création



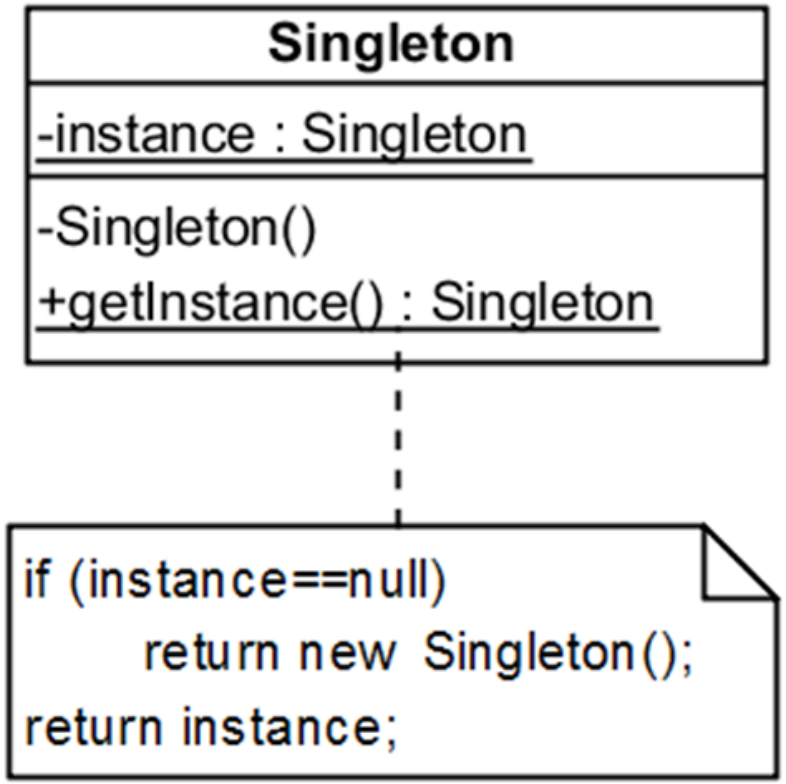
Singleton

- Garantir qu'une classe n'a qu'une seule instance et offre un point d'accès global

- Contrôle l'accès à l'instance
- Accès global sans utiliser de variable globale
- Facilite le raffinement des opérations et représentations
  - Singleton peut-être sous-classé

Structure

```
public class Singleton {
   private static Singleton _instance;
   private Singleton() {}

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



Problème

Gestionnaire d'ID

- Utiliser des identifiants uniques pour tous les objets du jeu
- IDs doivent être décernés par un seul objet
  - Sinon, on peut assigner un ID en double
- L'application doit avoir un accès global à ce distributeur
  - Compliqué et lourd de passer la référence au distributeur partout dans l'application

- S'assurer qu'il n'y a qu'**une seule instance** du distributeur d'ID qui peut être créée et que l'instance est **accessible de partout** dans l'application

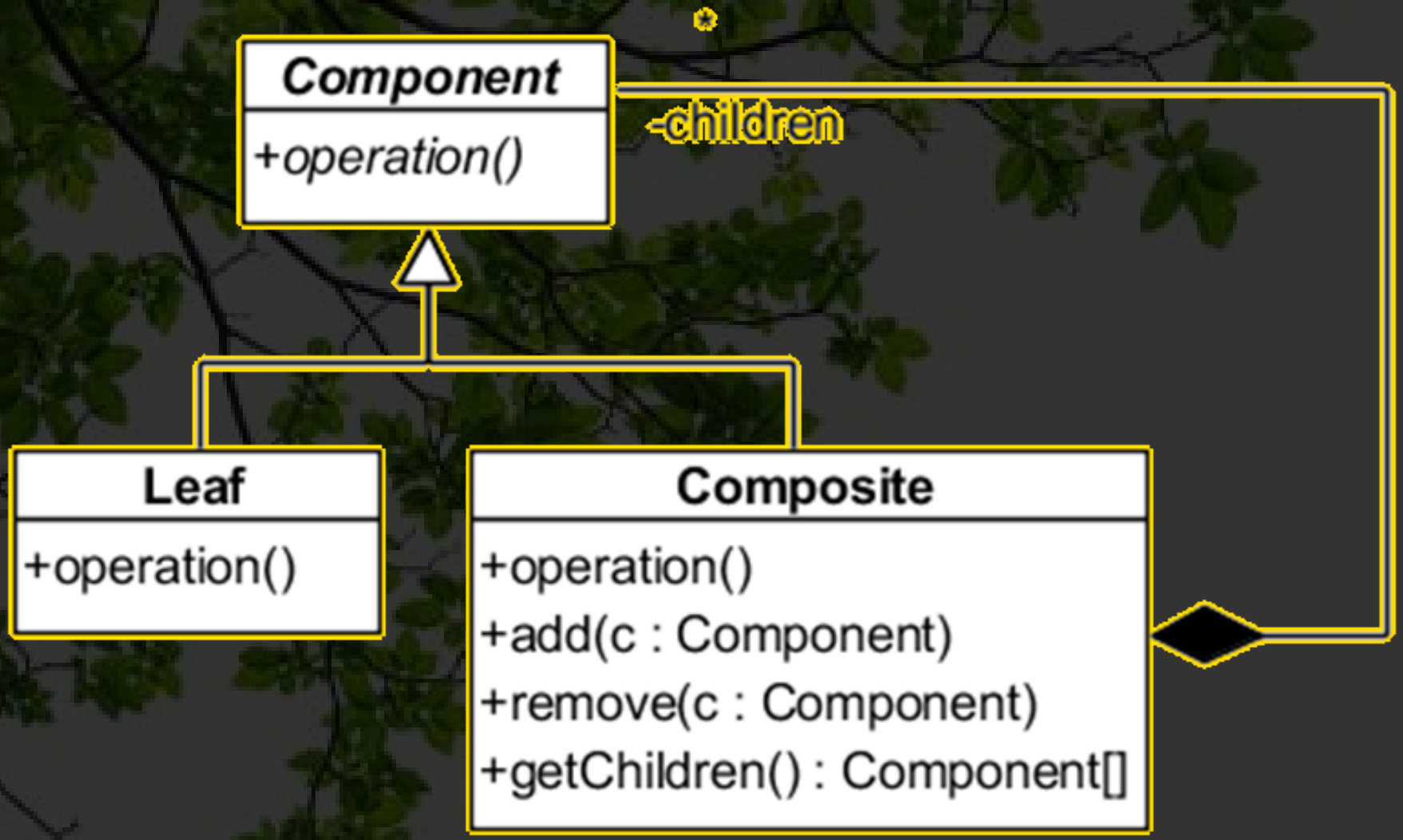
?

Solution

```
public class IdManager {
   private static IdManager _instance = new IdManager();
   private string lastID;
   private IdManager {
      this.lastID = -1;
   }
   public static IdManager getInstance() {
      return _instance
   }
   public string generateID() {
      return ++this.lastID;
   }
}
```

Composite

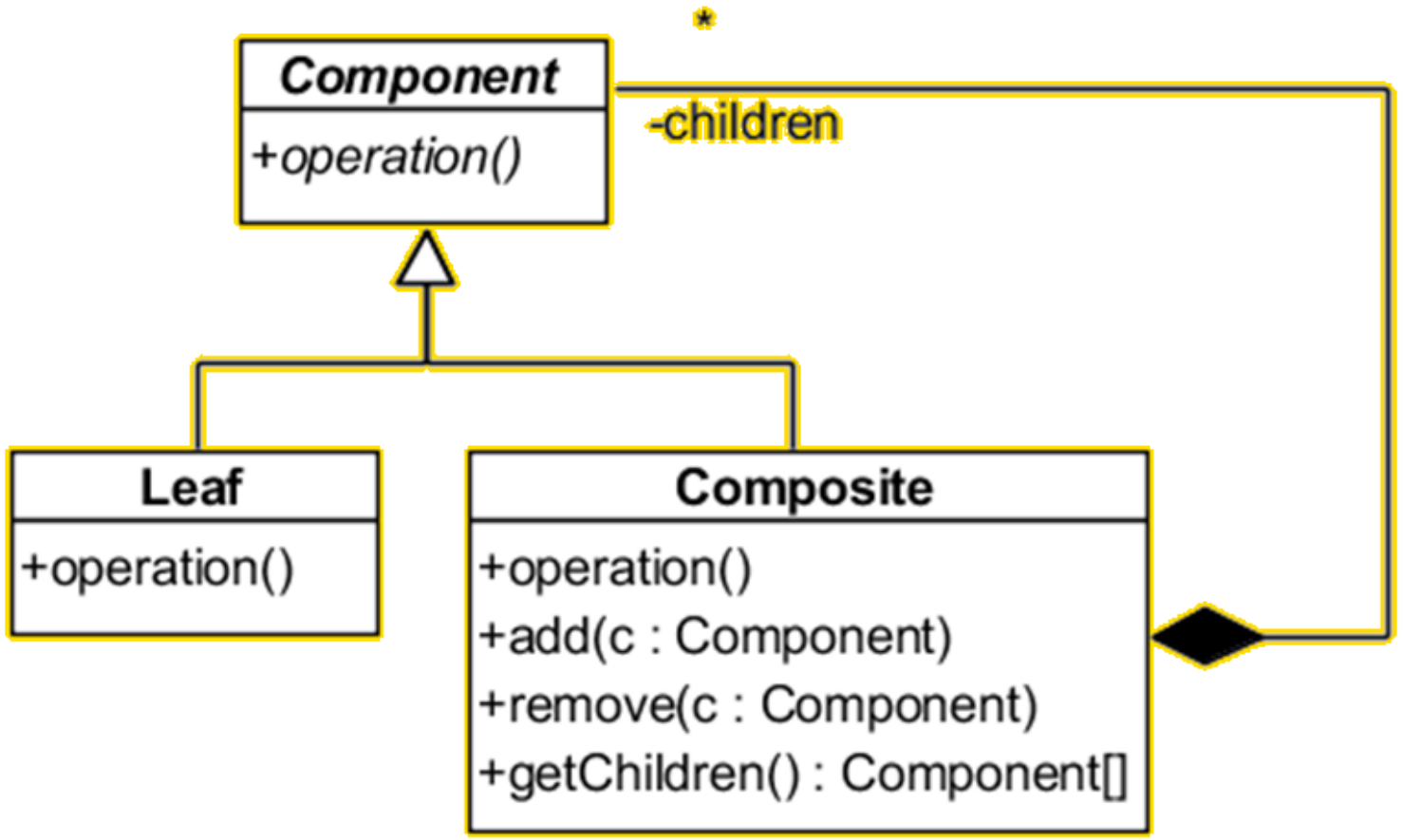
Patron de structure



Composite

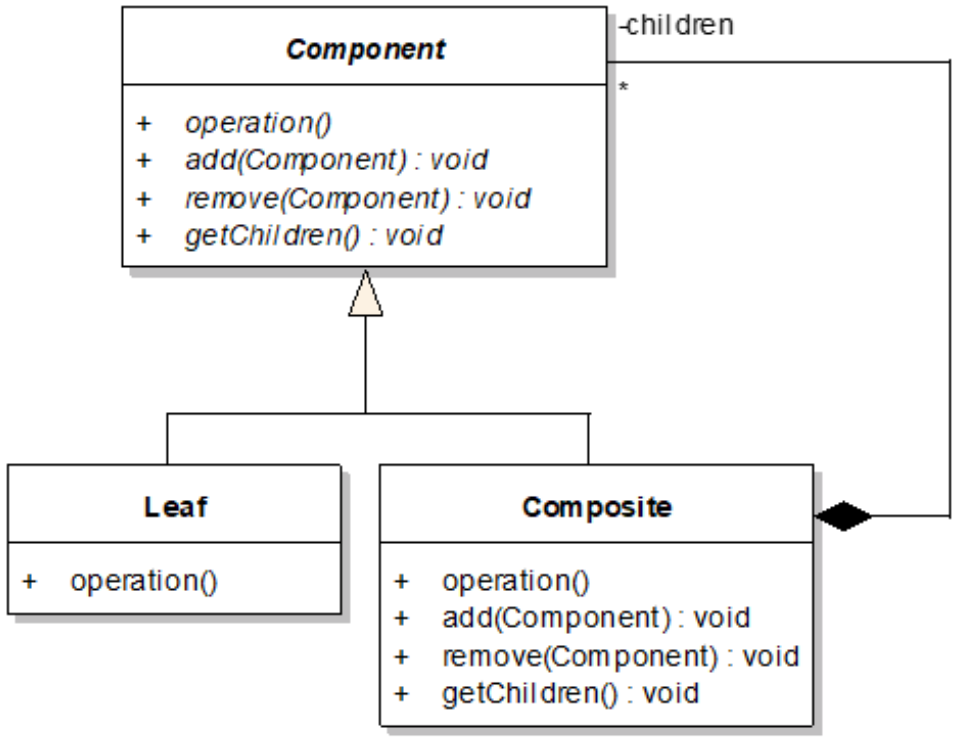
- Compose les objets en structures d'arborescence pour représenter des hiérarchies d'ensemble/partie
- Permet au client de traiter les objets et les compositions de façon uniforme
  - Objets primitifs, atomiques
  - Objets composites
- Définit une **hiérarchie claire** des objets
  - **Composition récursive** d'objets simples et complexes
  - Pas besoin de vérifier le type d'objet (dissimulation d'info)
  - Pas besoin de répétition de switch/case pour chaque opération

Structure

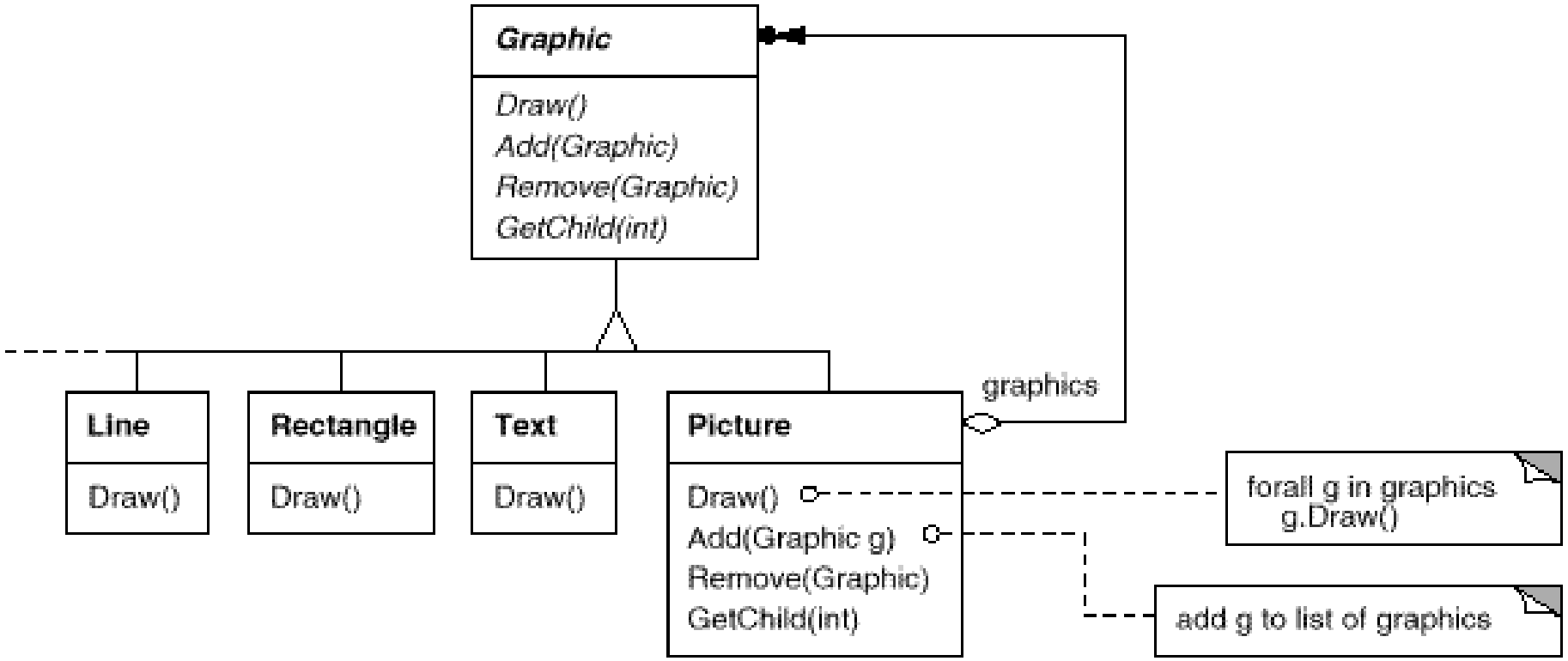

*
Component
+operation()
-children
Leaf
+operation()
Composite
+operation()
+add(c : Component)
+remove(c : Component)
+getChildren() : Component[]

Participants

- `Composant`
  - Déclare l'interface pour les objets
  - Implémente le comportement par défaut
  - Déclare les interfaces de gestion des composants enfant
- `Feuille`
  - Représente les primitives: pas d'enfant
- `Composite`
  - Définit le comportement des composants qui ont des enfants
  - Implémente les opération de gestion des enfants
- `Client`
  - Manipule les objets via l'interface `Composant`



Exemple


Graphic
Draw()
Add(Graphic)
Remove(Graphic)
GetChild(int)
Line
Draw()
Rectangle
Draw()
Text
Draw()
Picture
Draw()
Add(Graphic g)
Remove(Graphic)
GetChild(int)
graphics
forall g in graphics
g.Draw()
add g to list of graphics

Exemple code

```java
public abstract class Employee {
    protected String name;
    protected double salary;

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public double getSalary() {
        return salary;
    }
    public void setSalary(double salary) {
        this.salary = salary;
    }

    public abstract String parenthesize();
}
```

```java
public class Developer extends Employee {
    public Developer(String name)
    {
        this.name = name;
    }

    @Override
    public String parenthesize() {
        return this.name;

    }
}
```

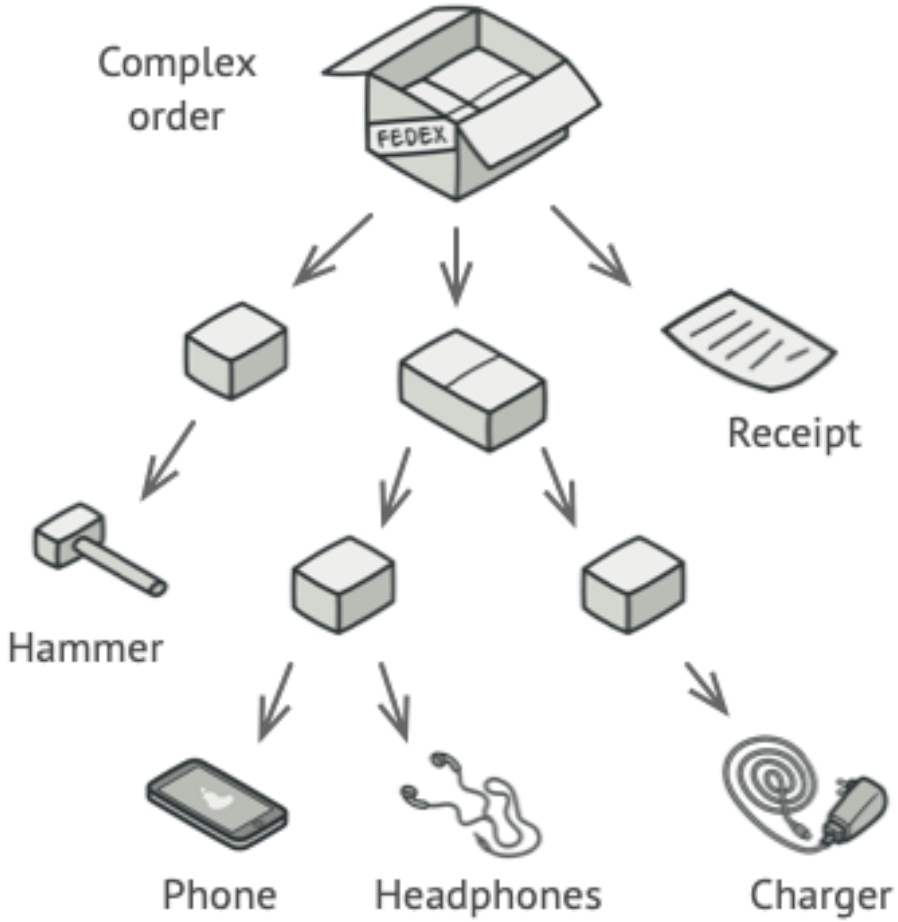
```java
import java.util.ArrayList;
import java.util.List;

public class Manager extends Employee {
    private List<Employee> children;

    public List<Employee> getChildren() {
        return children;
    }

    public void setChildren(List<Employee> children) {
        this.children = children;
    }

    public Manager(String name)
    {
        this.name = name;
        this.children = new ArrayList<Employee>();
    }

    public void add(Employee e) { this.children.add(e); }
    public void remove(Employee e) { this.children.remove(e); }

    @Override
    public String parenthesize() {
        String s = this.name + " (";
        for (Employee e : this.children)
            s += e.parenthesize() + " ";
        return s + ")";
    }

}
```

```java
public class Client {
    public static void main(String[] args)
    {
        Manager john = new Manager("John"),
                mike = new Manager("Mike"),
                ursula = new Manager("Ursula");
        Developer ed = new Developer("Ed"),
                jessie = new Developer("Jessie"),
                jane = new Developer("Jane"),
                brian = new Developer("Brian");

        john.add(mike);
        john.add(ursula);
        john.add(ed);
        mike.add(jessie);
        mike.add(jane);
        ursula.add(brian);

        System.out.println(john.parenthesize());
    }
}
```

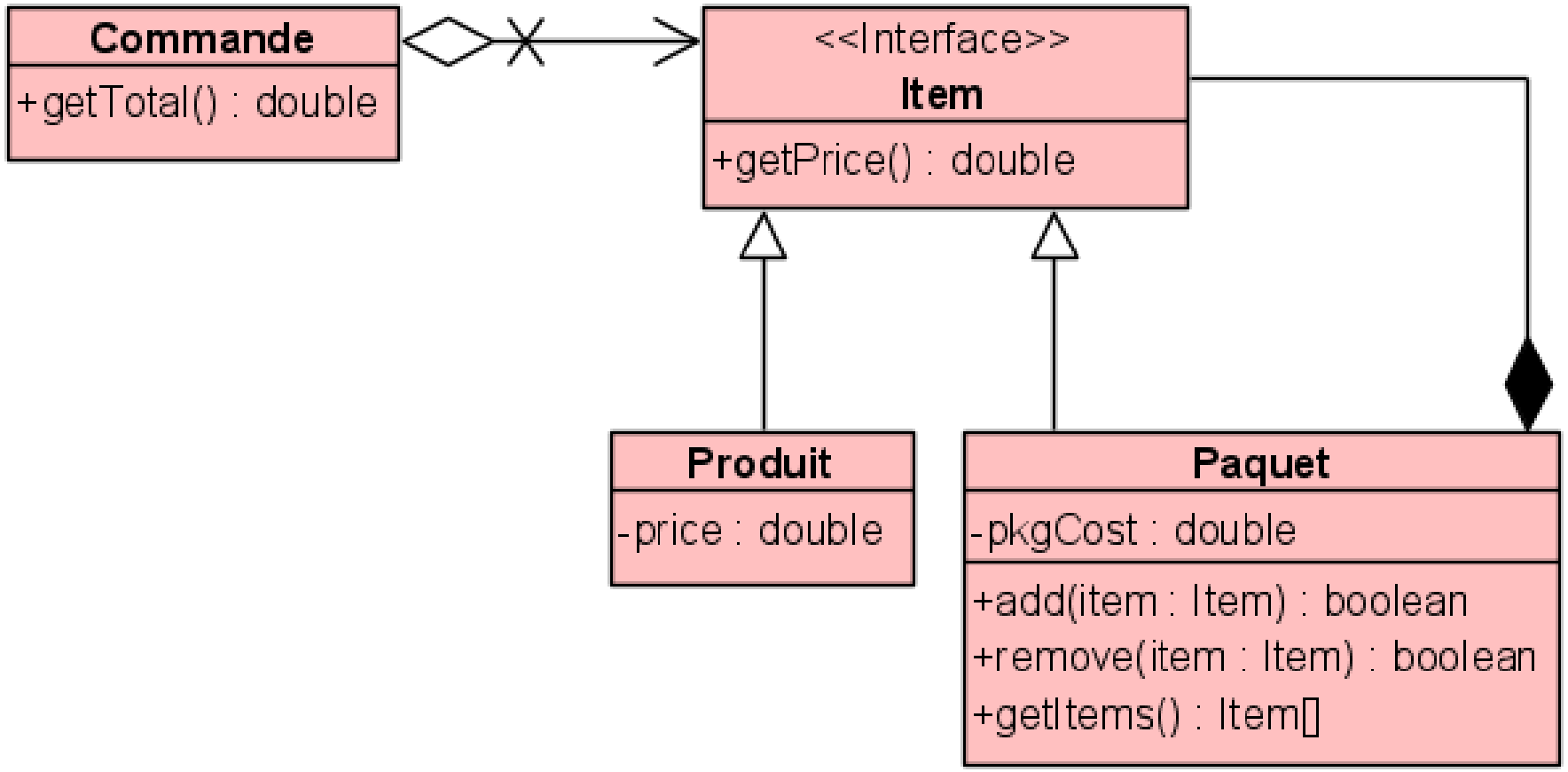
Problème

- Système de commande
- Une commande peut être composé de produits ou paquets
- Nous cherchons à calculer le prix de la commande

?

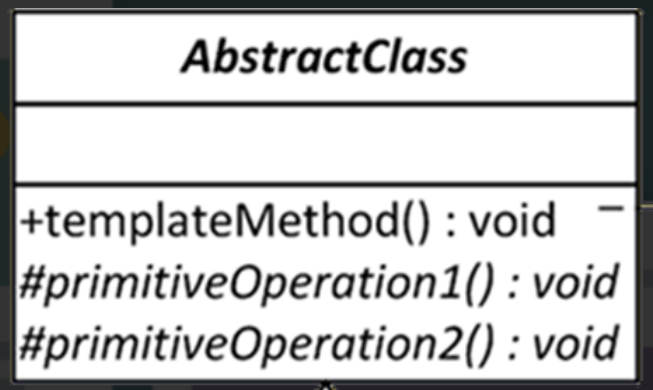


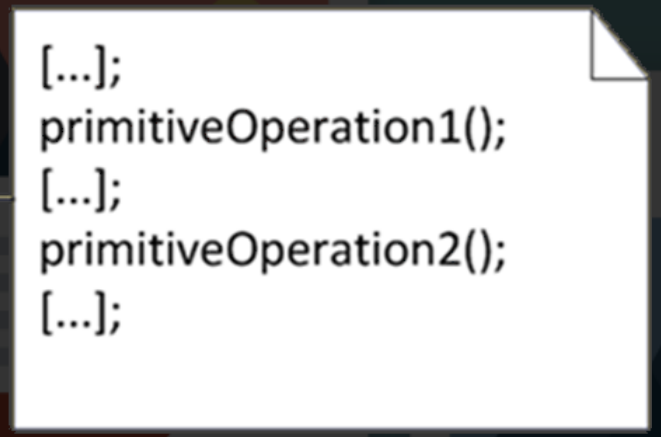
Solution

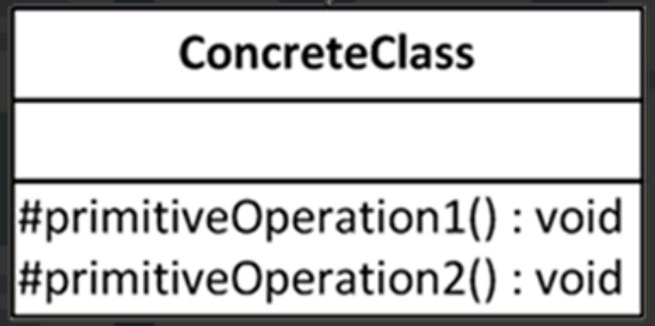

Commande
+getTotal() : double
<<Interface>>
Item
+getPrice() : double
Produit
-price : double
Paquet
-pkgCost : double
+add(item : Item) : boolean
+remove(item : Item) : boolean
+getItems() : Item[]

Patron de méthode

Patron de comportement







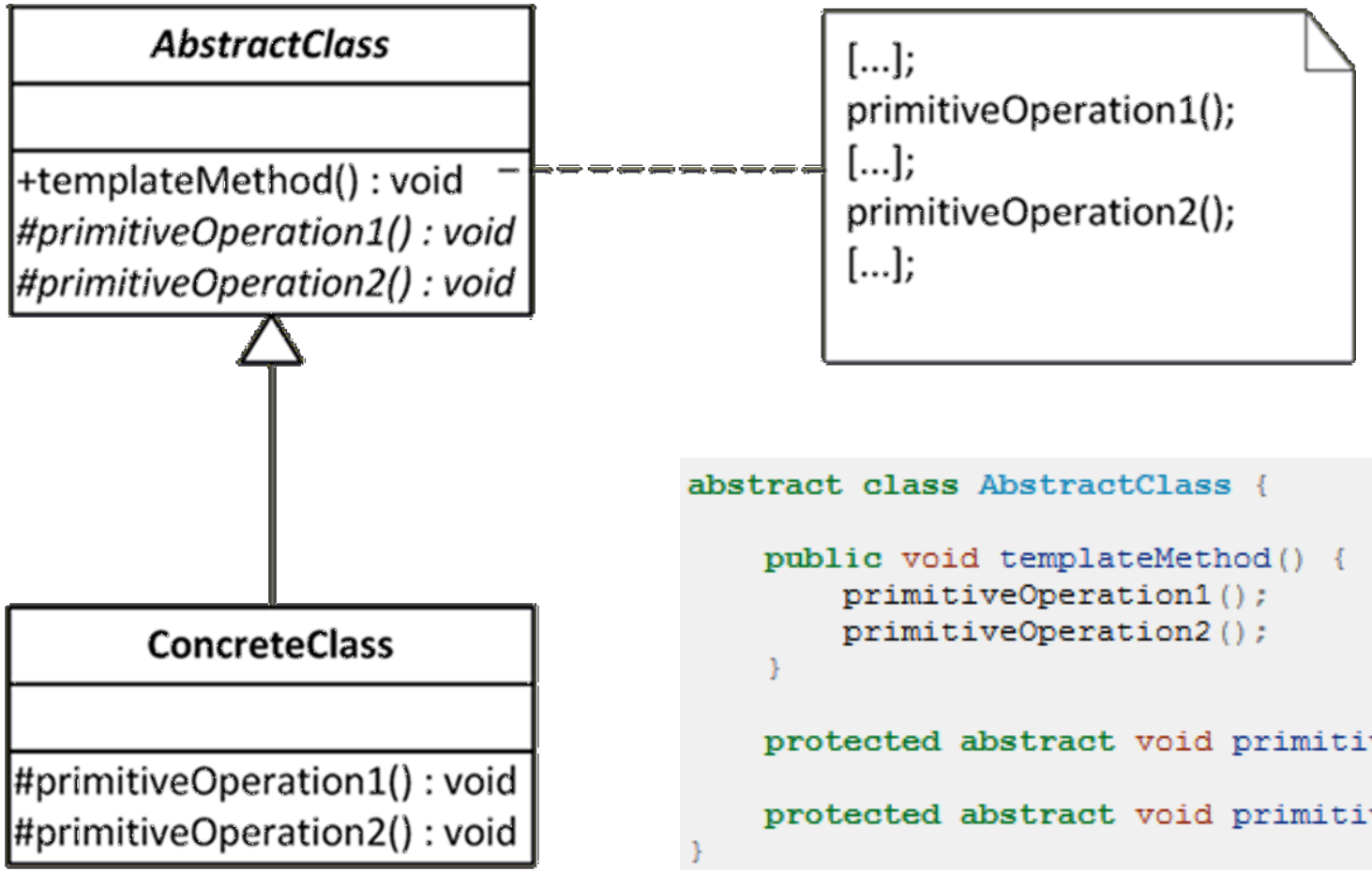

```
abstract class AbstractClass {

    public void templateMethod() {
        primitiveOperation1();
        primitiveOperation2();
    }

    protected abstract void primitiveOperation1();

    protected abstract void primitiveOperation2();
}
```

Patron de méthode

- Définit le **squelette** d'un algorithme dans une opération en **reportant** certaines étapes à des **sous-classes**
- Permet d'**affiner** certaines étapes d'un algorithme sans changer sa structure

- Implémente les **parties invariantes** d'un algorithme et laisse les sous-classes implémenter le comportement qui va **varier**
  - Polymorphisme
- Évite la **duplication de code** en factorisant le comportement commun des sous-classes

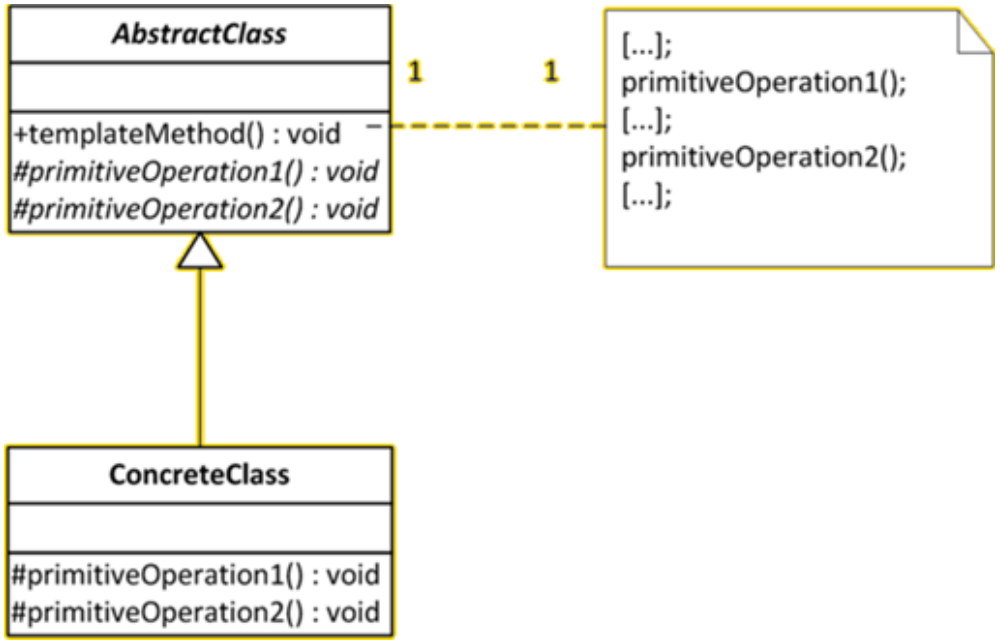
Structure



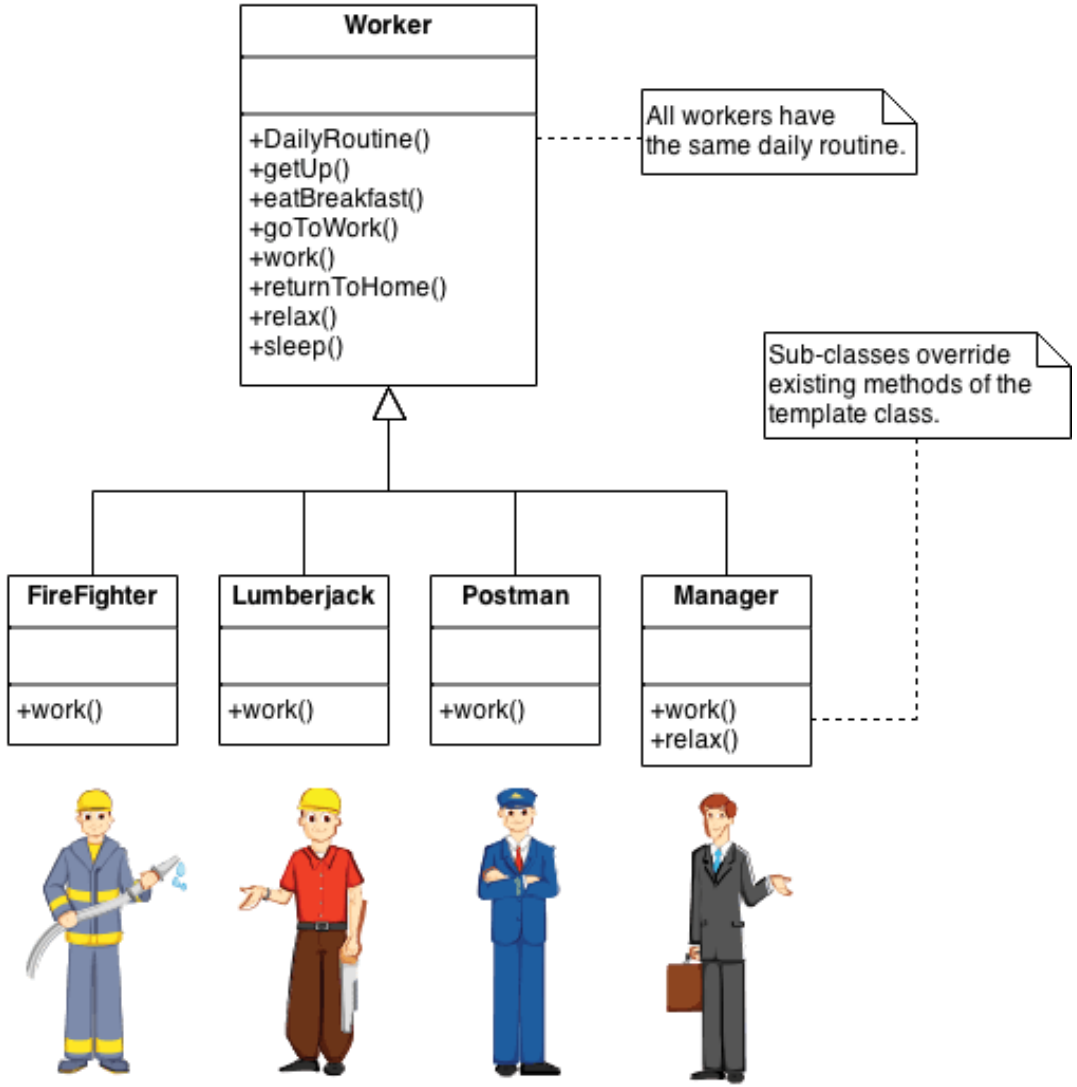

```
abstract class AbstractClass {

    public void templateMethod() {
        primitiveOperation1();
        primitiveOperation2();
    }

    protected abstract void primitiveOperation1();

    protected abstract void primitiveOperation2();
}
```

Participants

- `Classe Abstraite`
  - Définit l'ordre des tâches qui vont être utilisées dans toutes les classes concrètes
  - Choisit quelle tâche primitive doit être définie dans chaque classe concrète
- `Classe Concrète`
  - Implémente les tâches primitives



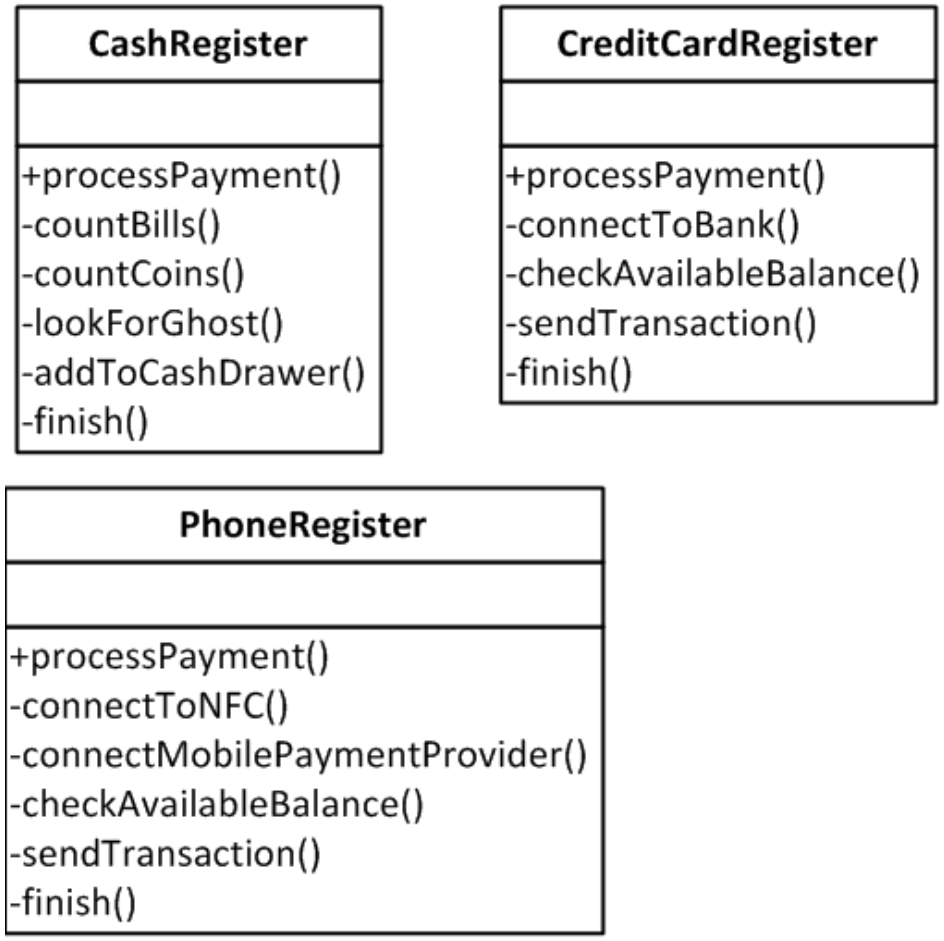
Exemple


Worker
+DailyRoutine()
+getUp()
+eatBreakfast()
+goToWork()
+work()
+returnToHome()
+relax()
+sleep()
All workers have the same daily routine.
Sub-classes override existing methods of the template class.
FireFighter
+work()
Lumberjack
+work()
Postman
+work()
Manager
+work()
+relax()

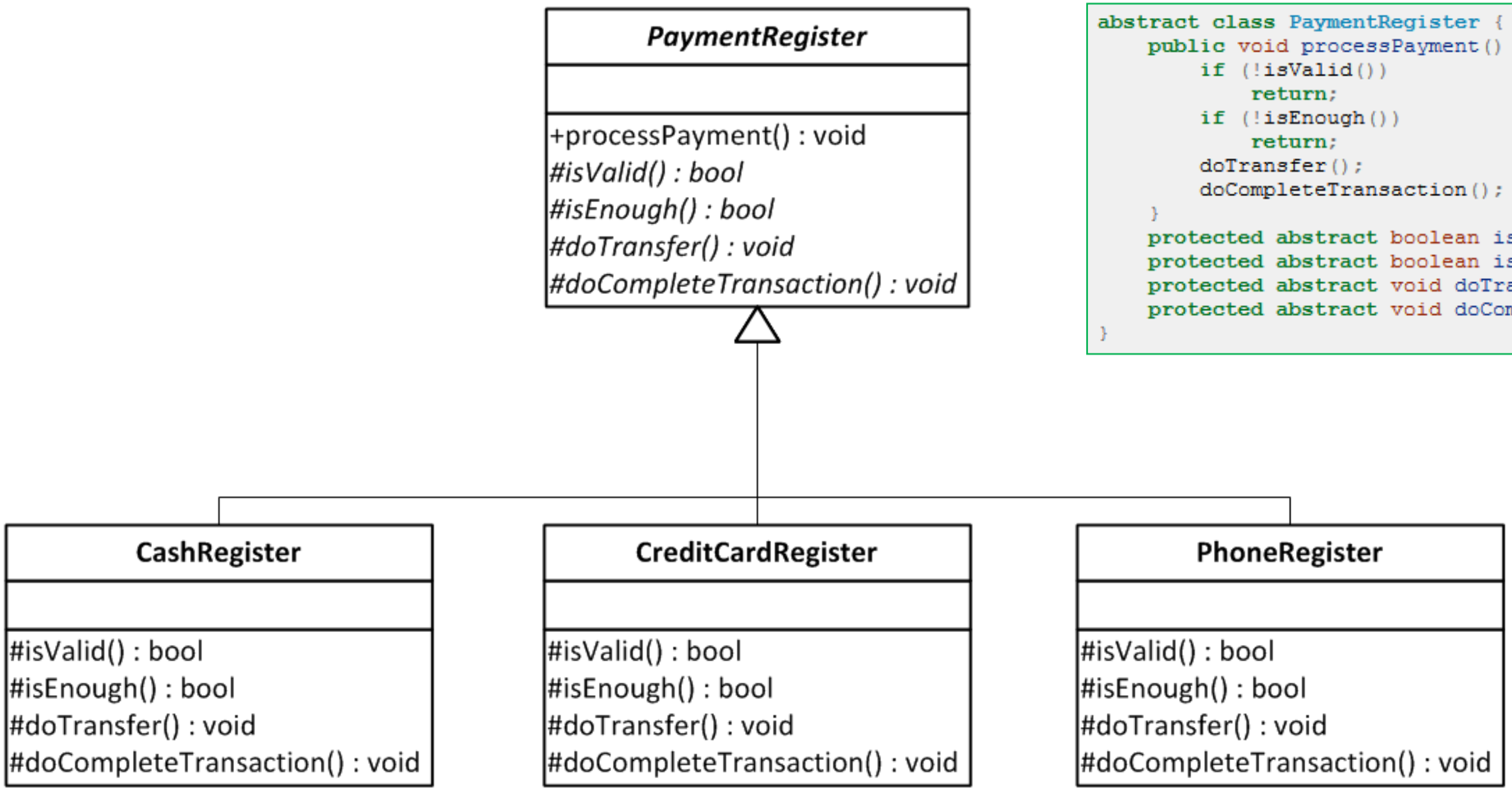
Problème

- Implémentations différentes
  - Comptant, carte, téléphone
- Niveau de détail différent
- Même étapes
- Comment factoriser un même algorithme général en tenant compte des différences dans les détails

?



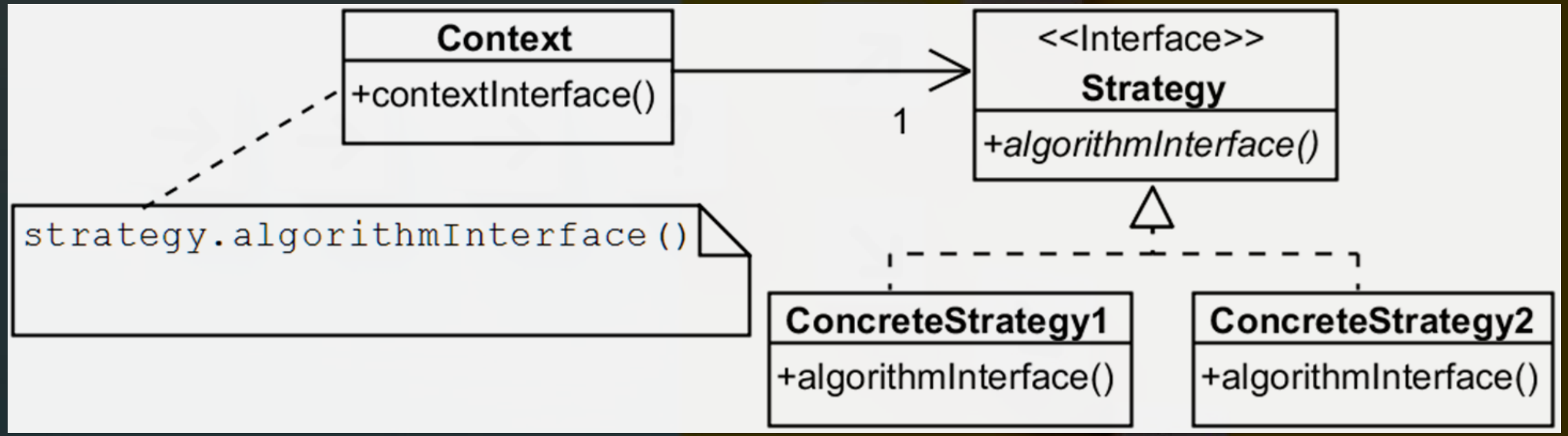
Solution




```
abstract class PaymentRegister {
    public void processPayment() {
        if (!isValid())
            return;
        if (!isEnough())
            return;
        doTransfer();
        doCompleteTransaction();
    }
    protected abstract boolean isValid();
    protected abstract boolean isEnough();
    protected abstract void doTransfer();
    protected abstract void doCompleteTransaction();
}
```

Stratégie

Patron de comportement

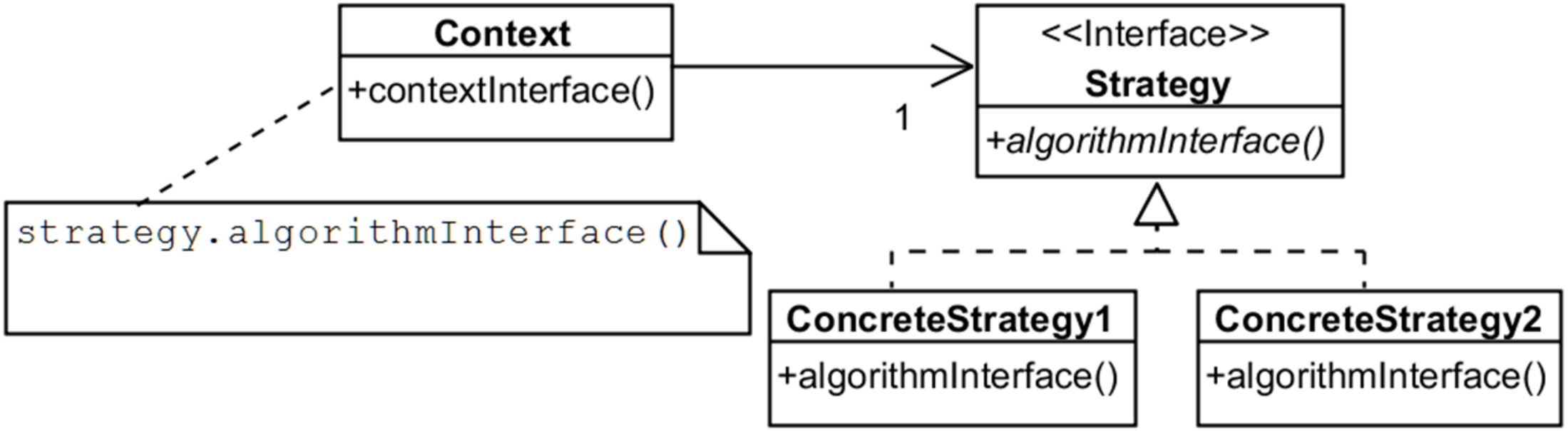


Stratégie

- Définit une **famille d'algorithmes**, encapsule chacun et rend-les interchangeables

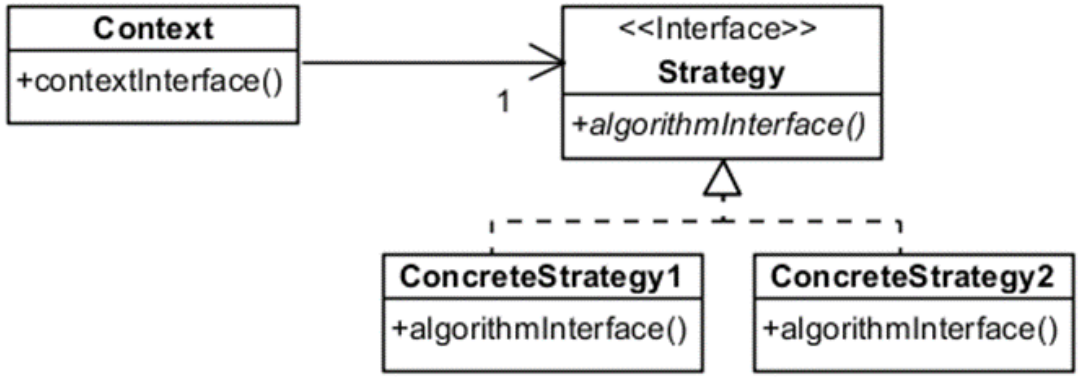
- Algorithme varie indépendamment du client qui l'utilise

Structure

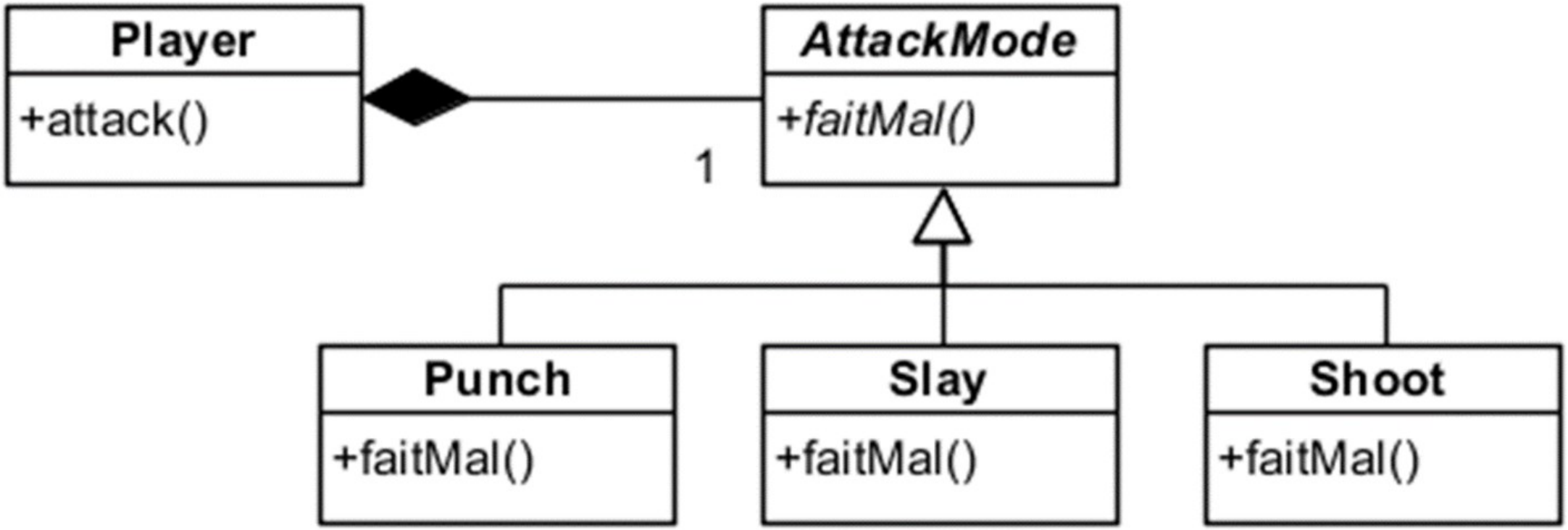

Context
+contextInterface()
1
<<Interface>>
Strategy
+algorithmInterface()
strategy.algorithmInterface()
ConcreteStrategy1
+algorithmInterface()
ConcreteStrategy2
+algorithmInterface()

Participants

- **`Stratégie`**
  - Déclare l'interface commune pour tous les algorithmes
  - Contexte utilise cette interface
- **`StratégieConcrète`**
  - Implémente l'algorithme
- **`Contexte`**
  - Fortement couplé avec la `Startégie`
  - Peut offrir une interface pour laisser la `Stratégie` accéder à ses données



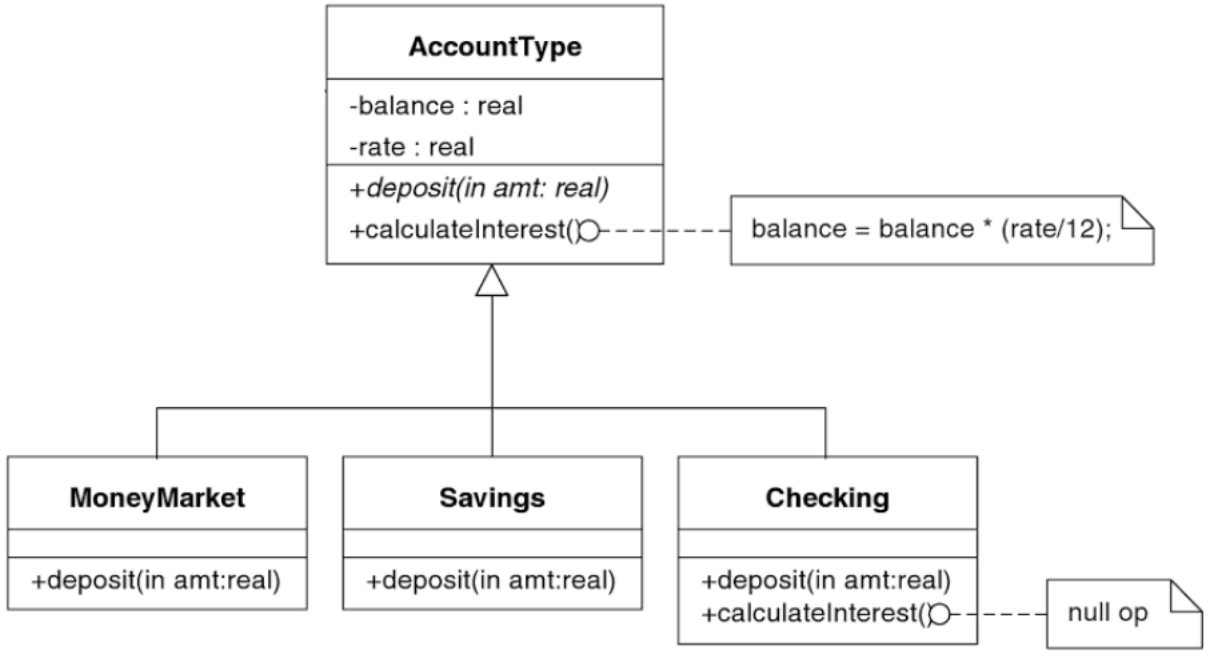
Exemple


Player
+attack()
AttackMode
+faitMal()
1
Punch
+faitMal()
Slay
+faitMal()
Shoot
+faitMal()

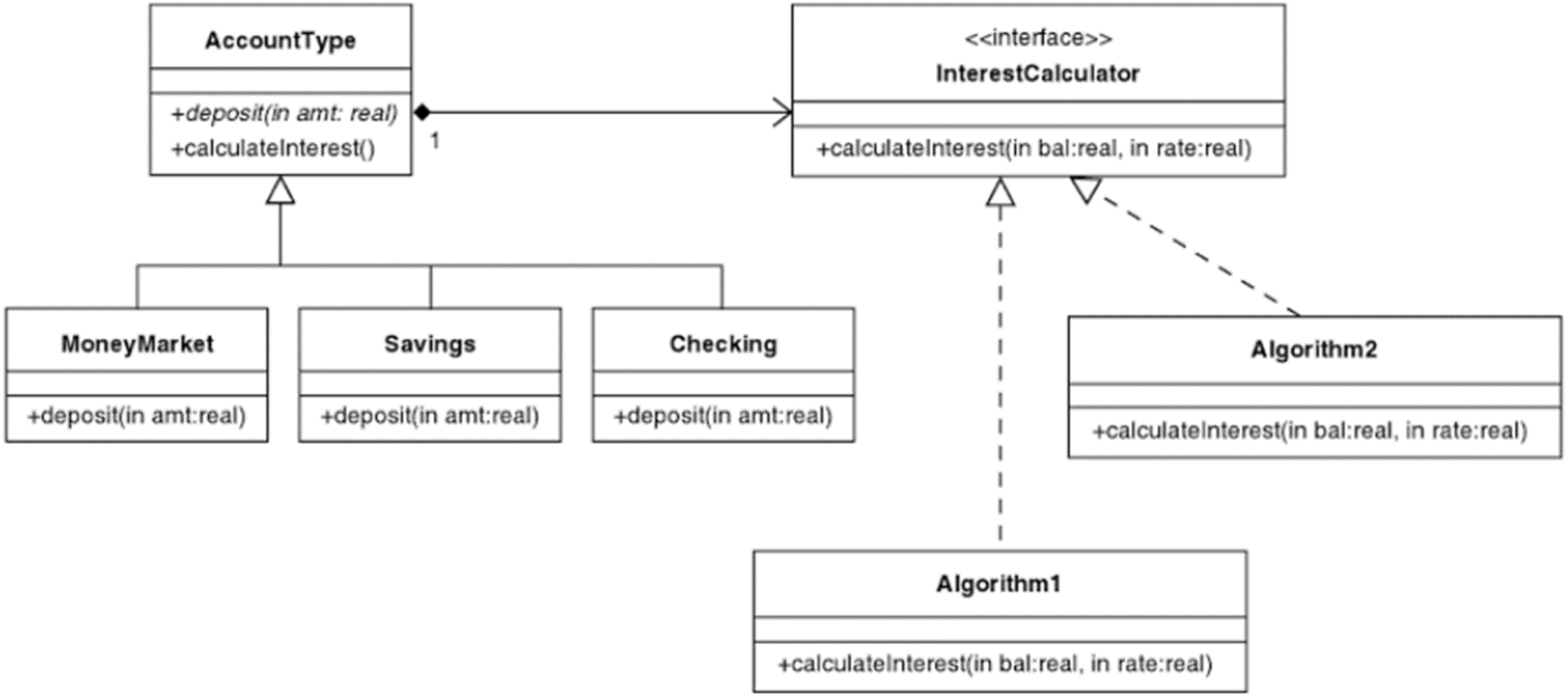
Problème

- Utilisation de polymorphisme et patron de méthode
  - Réutiliser le code du calcul d'intérêt
  - Donner un comportement par défaut dans le super-type
- Mais pas tous les types de comptes ont des intérêts

?



Solution


AccountType
+deposit(in amt: real)
+calculateInterest()
1
<<interface>>
InterestCalculator
+calculateInterest(in bal:real, in rate:real)
MoneyMarket
+deposit(in amt:real)
Savings
+deposit(in amt:real)
Checking
+deposit(in amt:real)
Algorithm2
+calculateInterest(in bal:real, in rate:real)
Algorithm1
+calculateInterest(in bal:real, in rate:real)

Résumé+

Forces et faiblesses

Forces

- Promeut la **réutilisation** en résolvant un problème de conception général
- Fournit une documentation de la conception en spécifiant des **abstractions**
- **Implémentations** déjà existantes
  - Pas besoin de programmer et documenter cette partie du code
  - Mais, besoin d'être testées et adaptées à notre contexte
- Il est plus facile de **comprendre** un programme qui utilise des patrons de conception
  - Même sans avoir vu ce programme avant!

Faiblesses

- Pas de manière **systématique** de déterminer où et quand **utiliser** un patron de conception
- Programmes plus complexes emploient **plusieurs patrons** qui interagissent entre eux
  - Problème de gestion des dépendances entre patrons peut être très complexe et perdre leurs avantages
- Le fait qu'on ait besoin de 23+ patrons de conception peut indiquer que notre **langage/paradigme** n'est pas assez puissant, ou trop générique

Principes de conception

- **SOLID**
  - **Responsabilité unique**
  - **Modifier par extension**
  - **Respecter ces parents**
  - **Segmenter les interfaces**
  - **Dépendre des abstractions**
- **GRASP**

+ KISS: Garder ça le plus simple que possible
+ DRY: Éviter de se répéter
+ YAGNI: Ajouter uniquement au besoin