

Méthodologies de développement Design Patterns

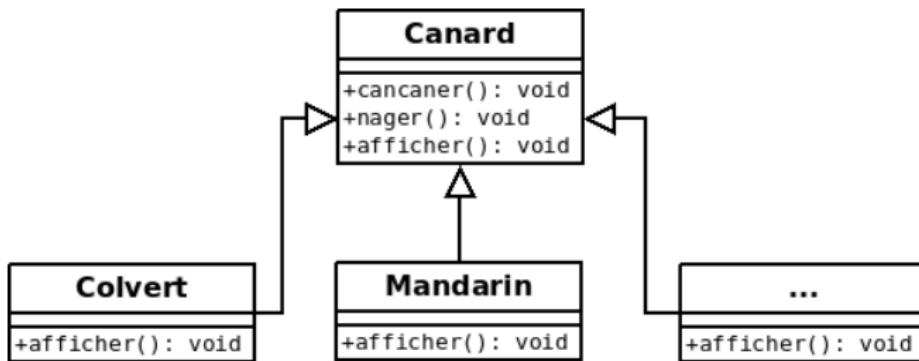
Florian Richoux

2017-2018

Strategy

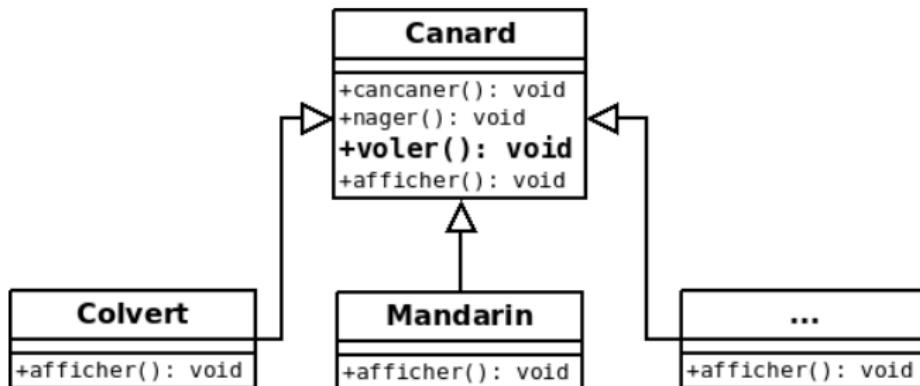
Faire des canards

Imaginez que l'on souhaite programmer le comportement de toutes sortes de canards.



Ajout d'une méthode

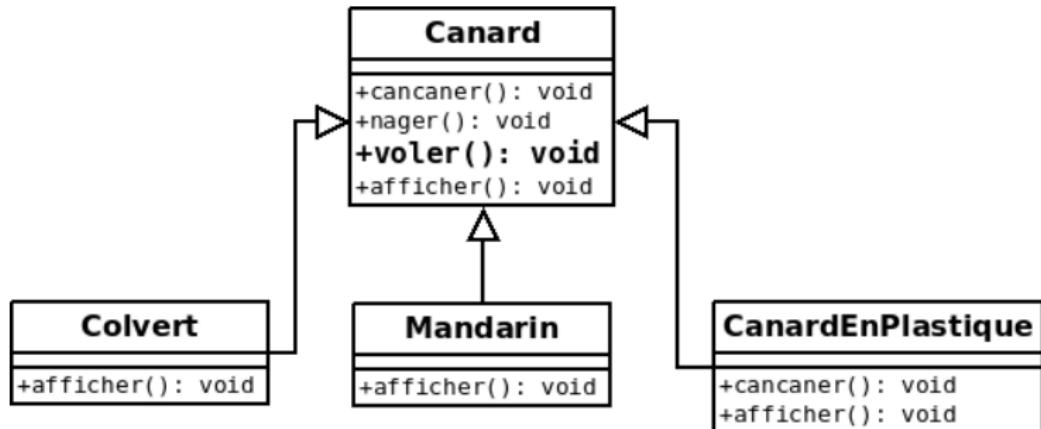
Comme tout va bien jusque là, on étends les fonctionnalités.



Problème après l'ajout d'une méthode

Pas de bol

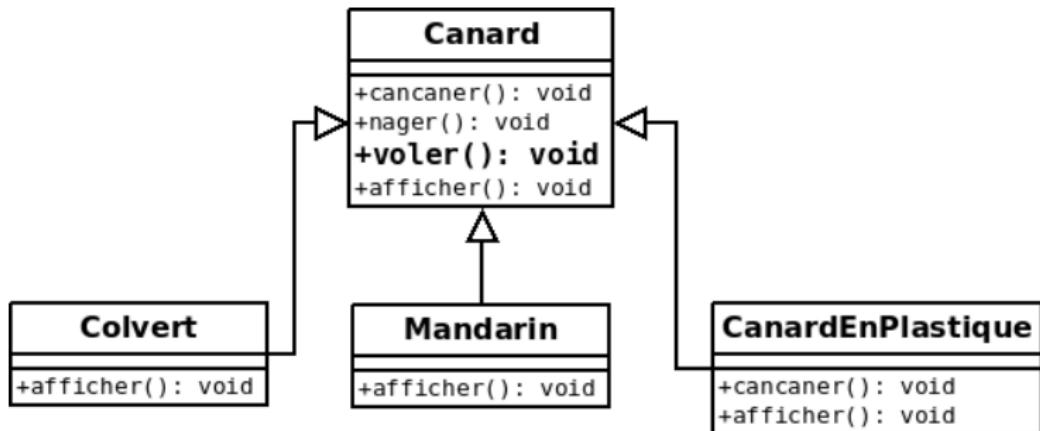
On a un canard en plastique, et ça vole pas...



Problème après l'ajout d'une méthode

Tiens tiens

CanardEnPlastique redéfinit la méthode *cancaner()* (pour couiner, sans doute). On pourrait pas redéfinir *voler()* et la laisser vide ?



Redéfinition de *voler()*

```
class CanardEnPlastique extends Canard
{
    public void cancaner()
    {
        System.out.println("Couinement");
    }

    public void afficher()
    {
        ...
    }

    public void voler() {}
}
```

On a résolu notre problème, mais...

On a résolu notre problème, mais...

Pas assez malin

En plus d'être inélégant, cette solution n'en est pas une :

- ▶ il faut **modifier chaque classe** qui posent problème.
- ▶ si un jour on **rajoute d'autres méthodes** à Canard, il faudra sans doute recommencer !

On a résolu notre problème, mais...

Pas assez malin

En plus d'être inélégant, cette solution n'en est pas une :

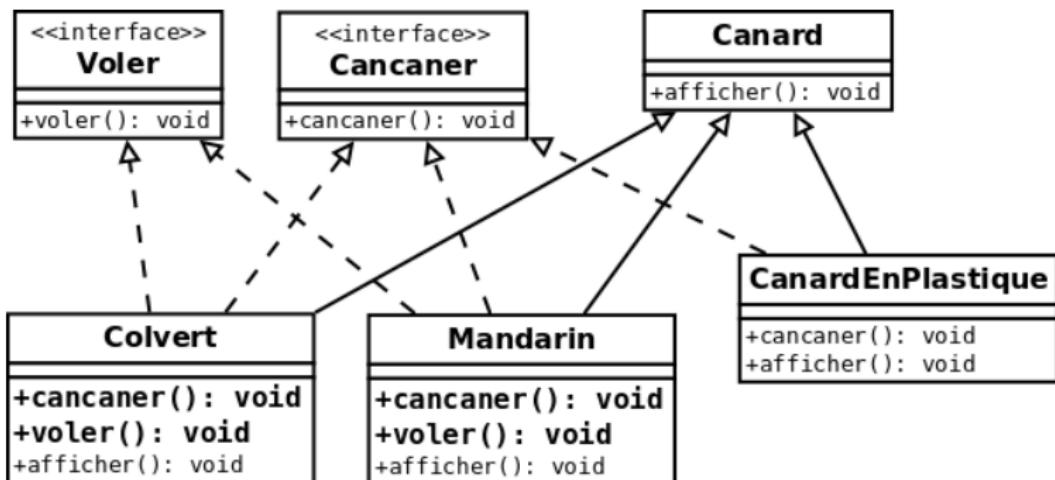
- ▶ il faut **modifier chaque classe** qui posent problème.
- ▶ si un jour on **rajoute d'autres méthodes** à Canard, il faudra sans doute recommencer !

Mauvaise structure

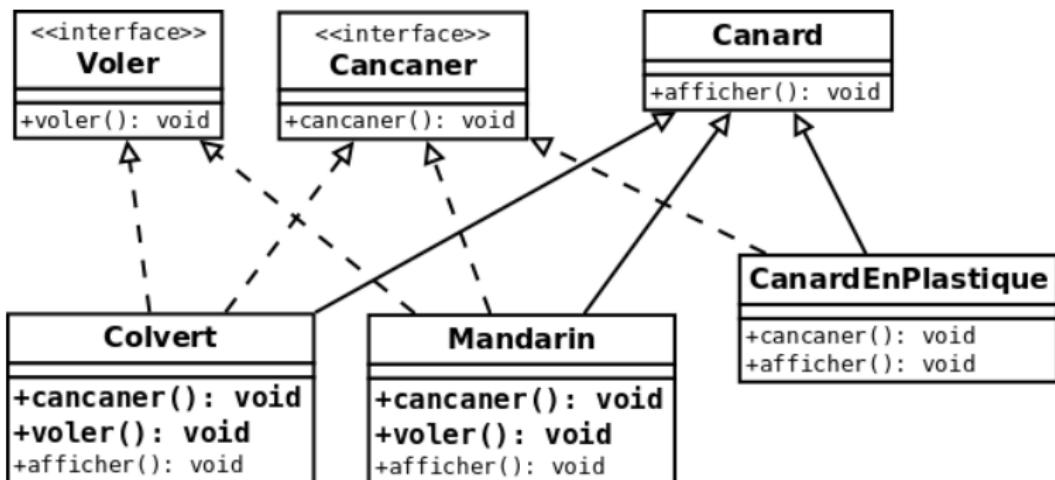
Le problème vient de la structure du programme, il faut la changer :

- ▶ Des modifications peuvent involontairement affecter des canards.
- ▶ Le code est dupliqué dans les sous-classes.
- ▶ Difficile de changer les comportements lors de l'exécution.

Nouvelle structure



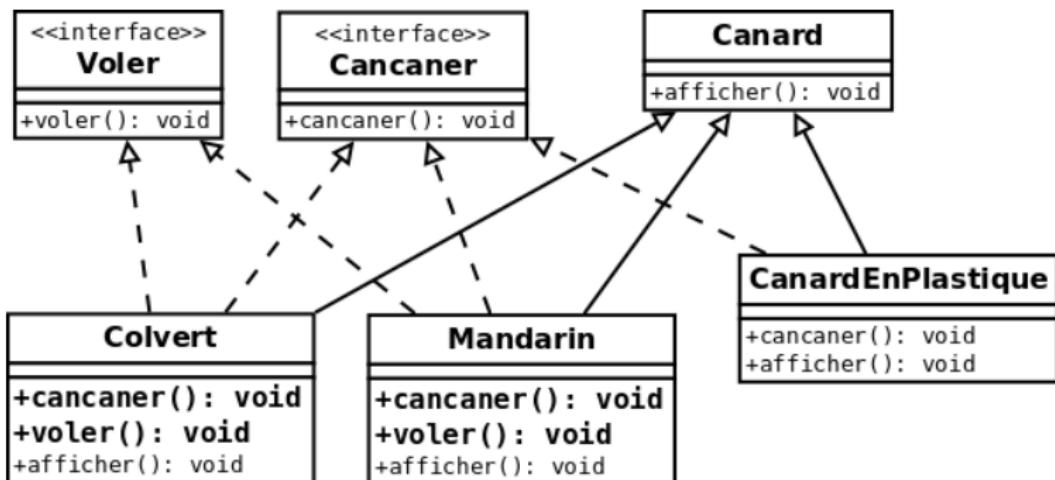
Nouvelle structure



Problème résolu ?

- ▶ Des modifications peuvent involontairement affecter des canards.
- ▶ Le code est dupliqué dans les sous-classes.
- ▶ Difficile de changer les comportements lors de l'exécution.

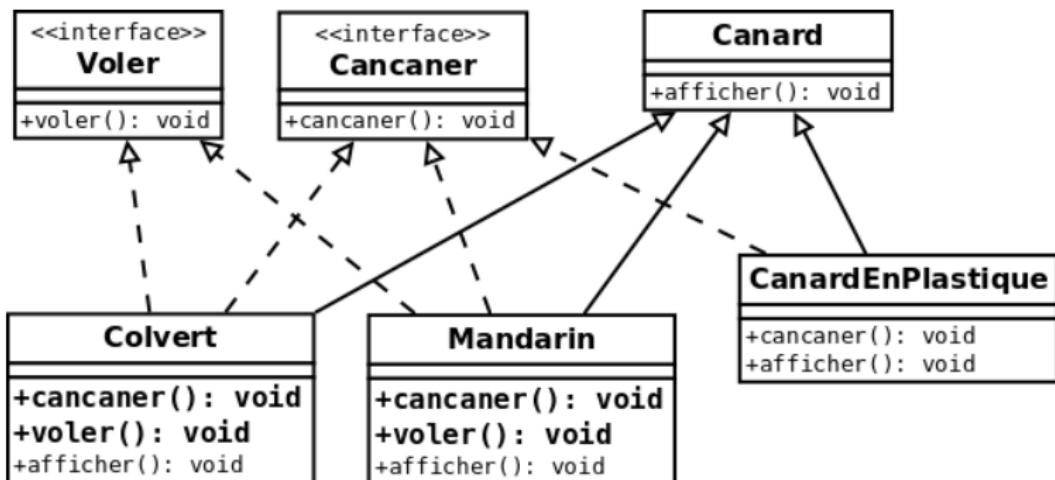
Nouvelle structure



Problème résolu ?

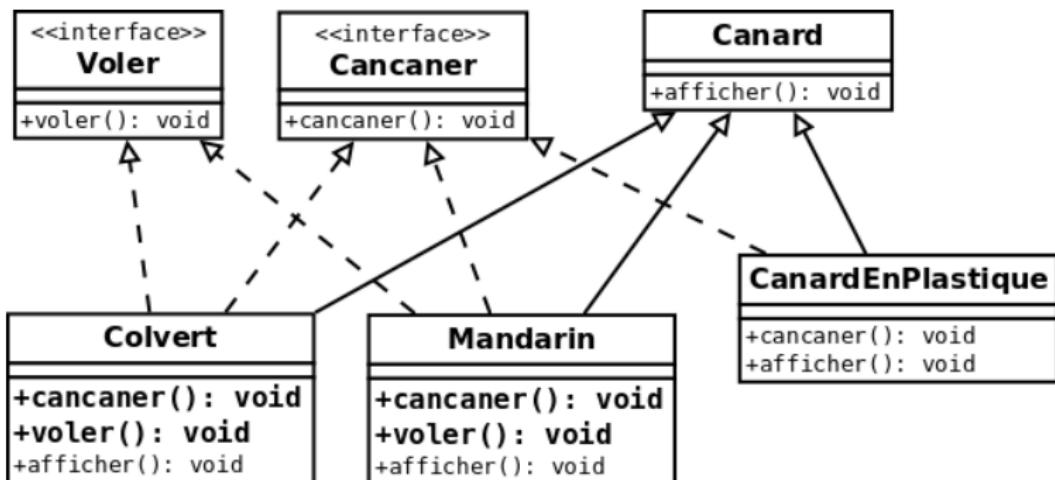
- ▶ Des modifications peuvent involontairement affecter des canards.
- ▶ Le code est dupliqué dans les sous-classes.
- ▶ Difficile de changer les comportements lors de l'exécution.

Nouvelle structure



Types

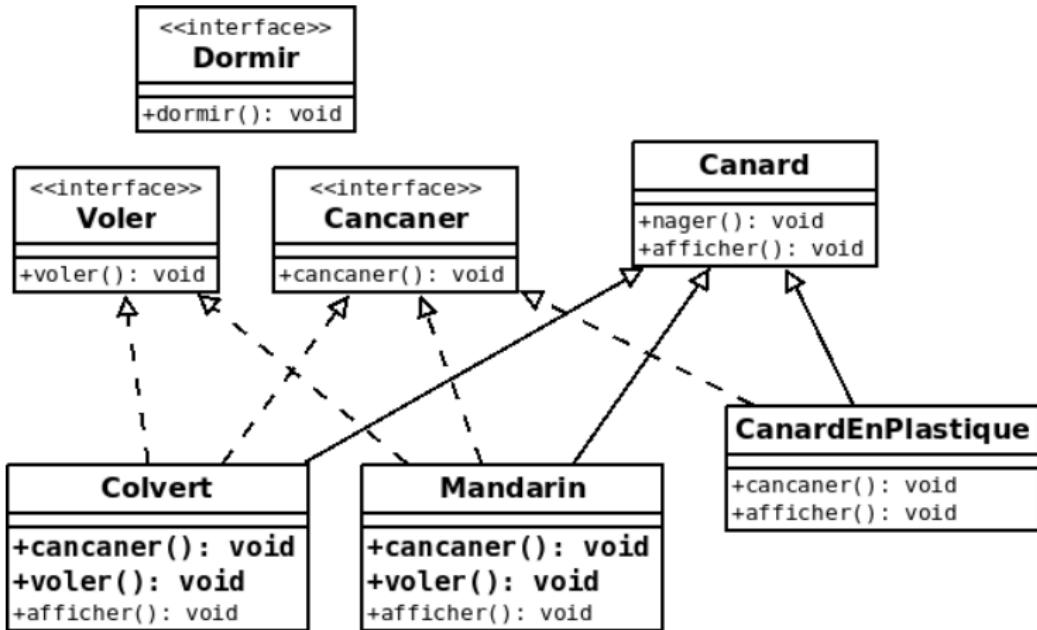
Colvert, Mandarin, etc, de type Canard ET Voler et Cancaner ?



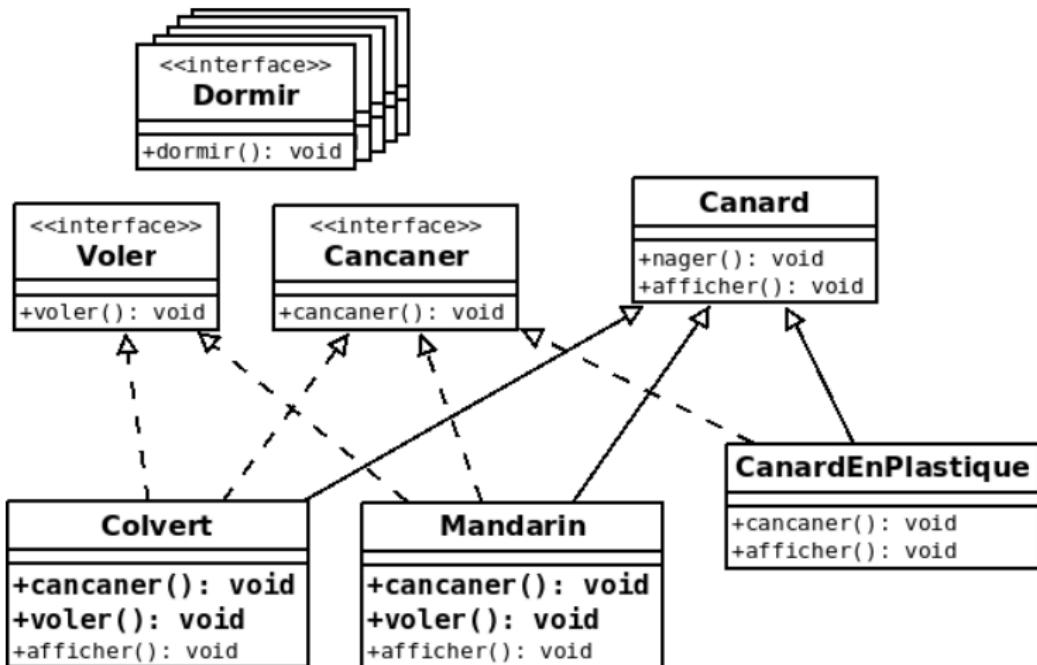
Mais surtout

On doit modifier chaque sous-classe en
spécifiant le comportement d'une
nouvelle fonctionnalité !

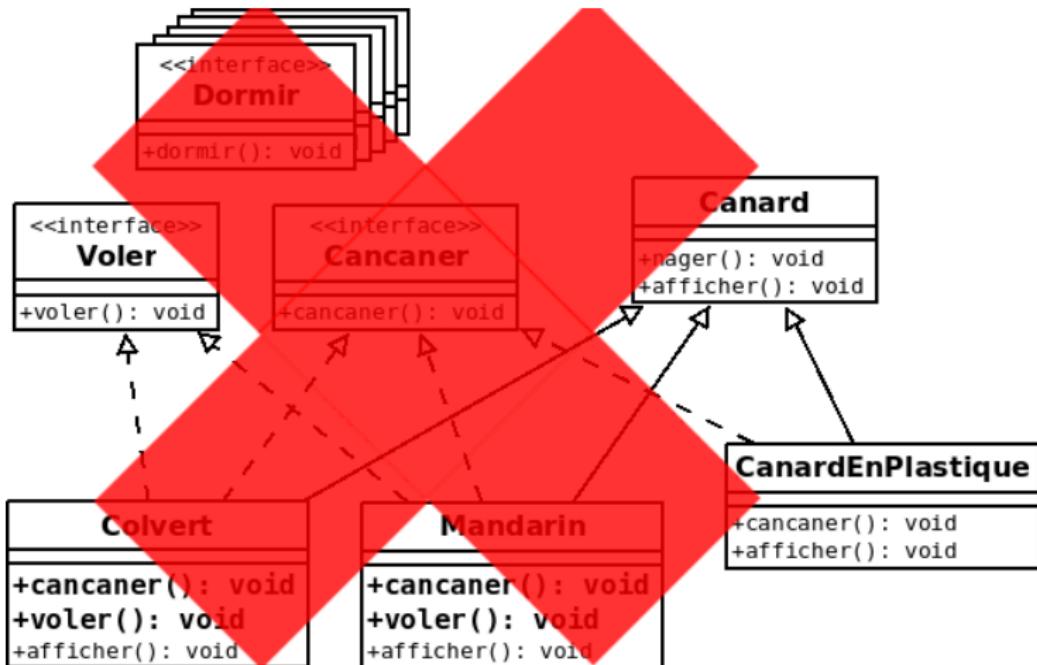
Nouvelle structure



Nouvelle structure



Nouvelle structure



La réalité des choses

Dans la vraie vie

Un programme change **tout le temps** :

- ▶ ajout d'une fonctionnalité,
- ▶ mise à jour,
- ▶ client qui change d'avis,
- ▶ ...

Objectif : minimiser les modifications

- ▶ Gain de temps.
- ▶ Code plus lisible.
- ▶ Moins de nouveaux bugs.

Analyser le code et le problème

Séparer ce qui peut potentiellement changer de ce qui reste constant.

But du jeu

Repérer ce qui varie et l'**encapsuler** (c.-à-d. le mettre dans une classe ou interface) pour l'**isoler** du reste du code.

⇒ Faire de la **composition**.

Ce qui varie d'un canard à l'autre

Tous les comportements: *voler()*, *cancaner()* ...

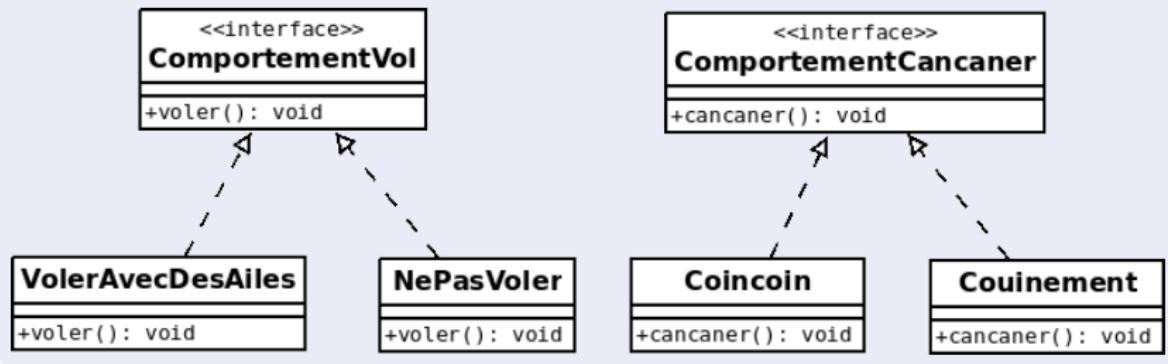
Ce qui ne varie pas d'un canard à l'autre

afficher(), car même si ça n'affiche pas la même chose, tous les canards doivent être capable de s'afficher.

Composition

Comment encapsuler *voler()* et *cancaner()* ?

Avec de bonnes vieilles interfaces



Composition

Avec de bonnes vieilles interfaces

```
interface ComportementCancan
{
    public void cancaner();
}

class Coincoin implements ComportementCancan
{
    public void cancaner()
    {
        System.out.println("Coin Coin");
    }
}

class Couinement implements ComportementCancan
{
    public void cancaner()
    {
        System.out.println("Couine");
    }
}
```

Composition

Avec de bonnes vieilles interfaces

```
interface ComportementVol
{
    public void voler();
}

class VolerAvecDesAiles implements ComportementVol
{
    public void voler()
    {
        System.out.println("Je vole.");
    }
}

class NePasVoler implements ComportementVol
{
    public void voler()
    {
        System.out.println("Je ne peux pas voler.");
    }
}
```

Intégrer ceci à notre classe Canard

Nouveau

Deux nouveaux attributs, deux nouvelles méthodes.

Canard	
#compVol:	ComportementVol
#compCancan:	ComportementCancan
+afficher():	void
+effectuerVol():	void
+effectuerCancan():	void

Intégrer ceci à notre classe Canard

En Java

```
class Canard
{
    protected ComportementCancan compCancan;

    public void effectuerCancan()
    {
        compCancan.cancaner(); //On délègue ce comportement
                               //à l'objet compCancan.
    }
}

class Colvert extends Canard
{
    public Colvert()
    {
        compVol = new VolerAvecDesAiles();
        compCancan = new Coincoin();
    }
}
```

Ce qui donne...

Programme

```
class Programme
{
    public static void main(String[] args)
    {
        Colvert colvert = new Colvert();
        colvert.effectuerCancan();
    }
}
```

À l'écran

```
$> java Programme
Coin Coin
```

Maintenant

- ▶ Facile d'ajouter une fonctionnalité sans (trop) modifier le code.
- ▶ Facile de coder le nouveau comportement d'un fonctionnalité **sans changer** le code existant et **sans déranger** les sous-classes de Canard qui ne sont pas concernées.
- ▶ Le code des comportements se trouve à un et un seul endroit !
- ▶ Facile de **réutiliser** les comportements pour d'autres classes.

On peut même changer des comportements à chaud !

Programme

```
class Programme
{
    public static void main(String[] args)
    {
        Colvert colvertBlesse = new Colvert();
        colvertBlesse.effectuerVol();
        // PAN !
        colvertBlesse.changeCompVol( new NePasVoler() );
        colvertBlesse.effectuerVol();
    }
}
```

À l'écran

```
$> java Programme
Je vole.
Je ne peux pas voler.
```

Avec cette nouvelle structure

- ▶ Des modifications peuvent involontairement affecter des canards.
- ▶ Le code est dupliqué dans les sous-classes.
- ▶ Difficile de changer les comportements lors de l'exécution.

Avec cette nouvelle structure

- ▶ Des modifications peuvent involontairement affecter des canards.
- ▶ Le code est dupliqué dans les sous-classes.
- ▶ Difficile de changer les comportements lors de l'exécution.

Avec cette nouvelle structure

- ▶ Des modifications peuvent involontairement affecter des canards.
- ▶ Le code est dupliqué dans les sous-classes.
- ▶ Difficile de changer les comportements lors de l'exécution.

Avec cette nouvelle structure

- ▶ Des modifications peuvent involontairement affecter des canards.
- ▶ Le code est dupliqué dans les sous-classes.
- ▶ Difficile de changer les comportements lors de l'exécution.

Avec cette nouvelle structure

- ▶ Des modifications peuvent involontairement affecter des canards.
- ▶ Le code est dupliqué dans les sous-classes.
- ▶ Difficile de changer les comportements lors de l'exécution.

On voit que les anciennes structures à base d'héritage n'étaient pas satisfaisantes.

Principe à garder en tête

“Favor composition over inheritance.”

Vous venez d'apprendre votre premier pattern !

Vous venez d'apprendre votre premier pattern !

Pattern Strategy

Définition

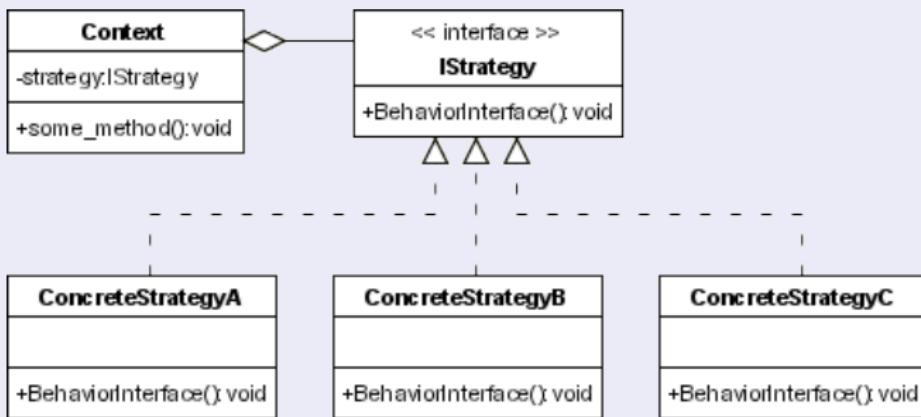
Le pattern Strategy (Stratégie) définit une famille d'algorithmes, encapsule chacun d'eux et les rend interchangeables. Il permet à l'algorithme de varier indépendamment des clients qui l'utilisent.

Modules indépendants

Dans notre exemple, le client est la classe Canard et les familles d'algorithmes sont les différentes manières de cancaner, de voler, etc.

Pattern Strategy

Diagramme UML



3 types de patterns

- ▶ Les patterns de **création** concernent la création de classes ou d'objets.
- ▶ Les patterns de **structure** aident à structurer une composition.
- ▶ Les patterns de **comportement** définissent les interactions entre objets.

Type du pattern Strategy

Le pattern Strategy rentre dans la classe des patterns de **comportement**.

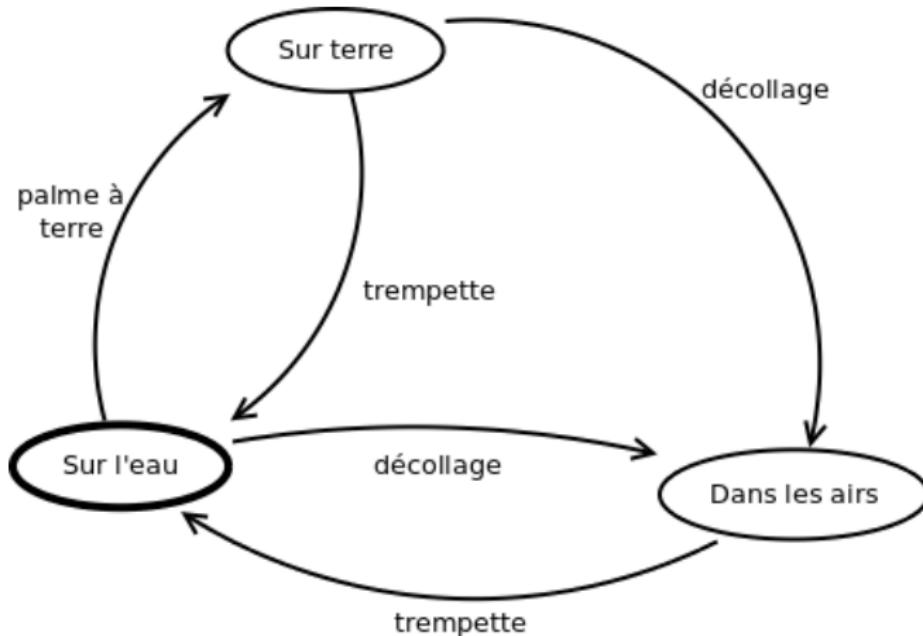
Feuille de TP 01

Strategy

State

On cherche à programmer un simulateur de canard





Une seule action : avancer

Comment coder ça ?

Coder les états : 1^{ere} idée

```
final static int EAU = 0;  
final static int TERRE = 1;  
final static int AIR = 2;  
  
int etat = EAU;
```

Comment coder ça ?

Coder les actions : 1^{ere} idée

```
public void avancer()
{
    if( etat == EAU )
    {
        System.out.println("Canard nage");
    }
    else if( etat == TERRE )
    {
        System.out.println("Canard marche");
    }
    else if( etat == AIR )
    {
        System.out.println("Canard vole");
    }
}
```

Comment coder ça ?

Coder les transitions : 1^{ere} idée

```
public void palmeATerre()
{
    if( etat == EAU )
    {
        etat = TERRE;
    }
    else if( etat == TERRE )
    {
        System.out.println("Canard est déjà à terre");
    }
    else if( etat == AIR )
    {
        System.out.println("Pas d'atterrissage possible");
    }
}
```

Comment coder ça ?

Coder le simulateur : 1^{ere} idée

```
class SimCanard
{
    final static int EAU = 0;
    final static int TERRE = 1;
    final static int AIR = 2;

    private int etat_;

    public SimCanard()
    {
        etat_ = EAU;
    }
}
```

Batterie de tests

```
SimCanard simulateur = new SimCanard();
simulateur.avancer(); // Canard nage

// une simulation normale
simulateur.palmeATerre();
simulateur.decollage();
simulateur.avancer(); // Canard vole

// un canard shooté
simulateur.palmeATerre();
simulateur.palmeATerre();
simulateur.decollage();
simulateur.palmeATerre();
simulateur.trempette();
simulateur.decollage();
simulateur.trempette();
simulateur.trempette();
simulateur.avancer(); // Canard nage
```

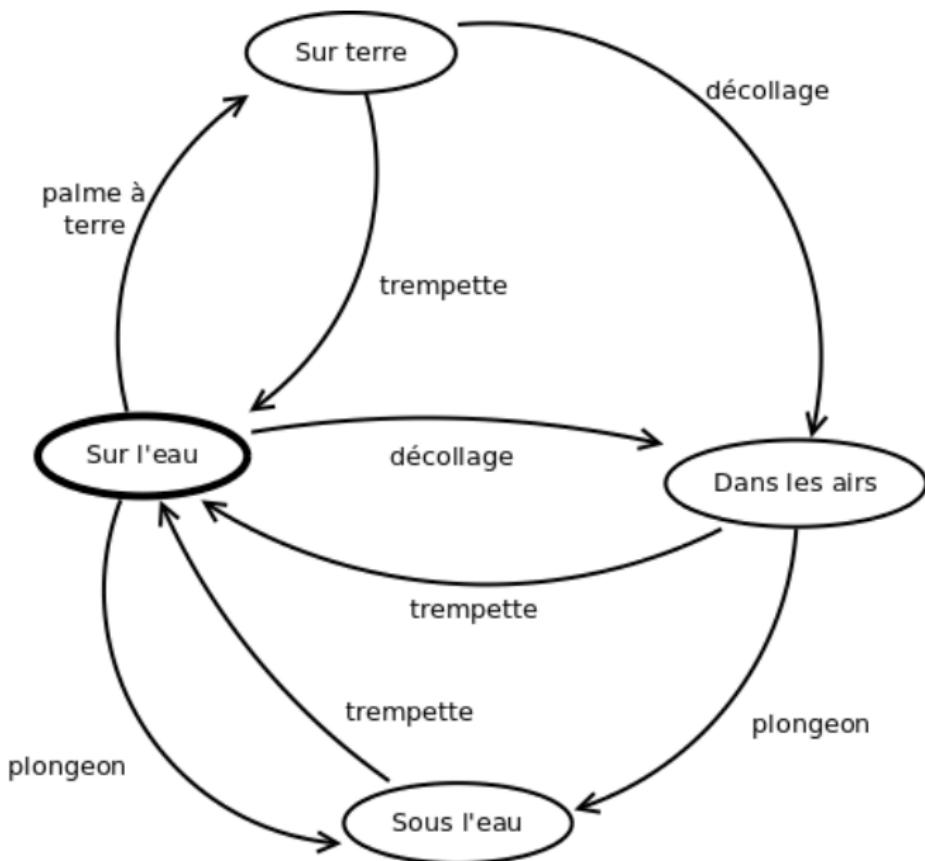
Pour plus de réalisme

Nouvelle fonctionnalité

Quelqu'un a une super idée :
pour pêcher le poisson, le canard peut plonger dans l'eau.



Le nouveau simulateur de canard



FAIL!

- ▶ Introduire un nouvel état. Ça, ça va.
- ▶ Ajouter une nouvelle méthode aussi (plongeon).
- ▶ **Changer TOUTES les méthodes avec un nouveau else if.**

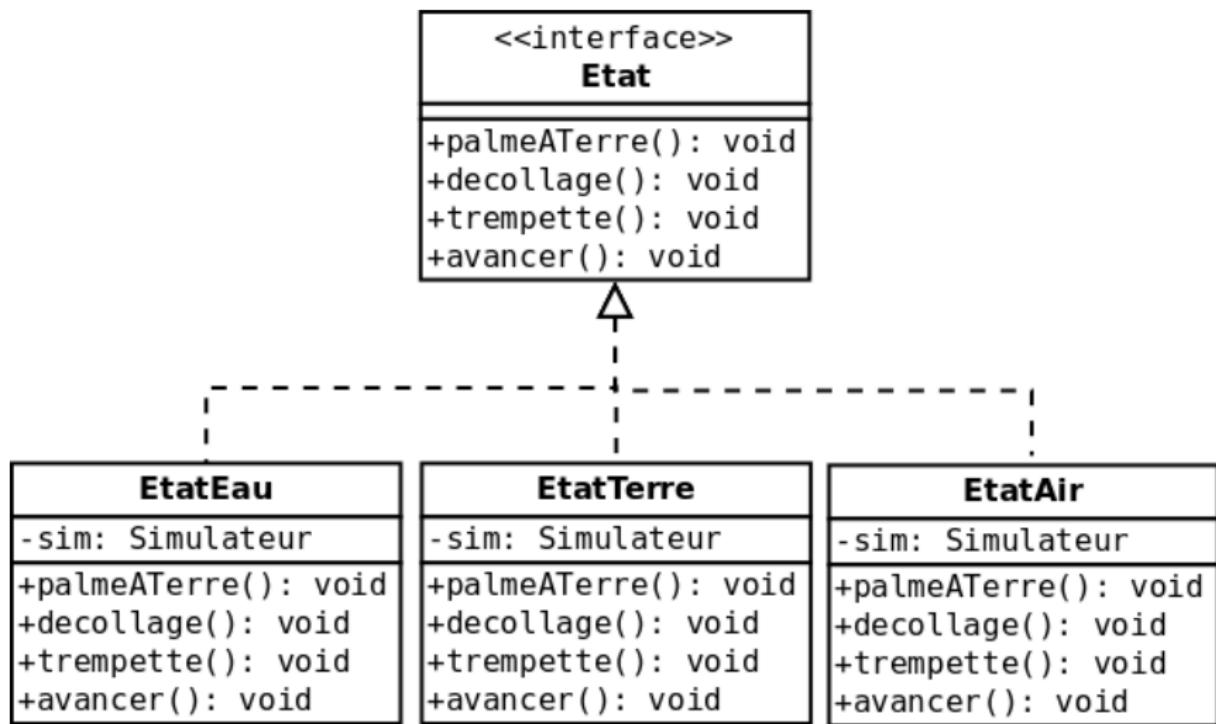
Restructuration de notre programme

- ▶ Définir une interface Etat avec une méthode pour chaque transition et chaque action.
- ▶ Implémenter une classe pour chaque état.
- ▶ Supprimer les conditionnelles et faire de la composition.

Remise à zéro

Restructurons déjà le simulateur classique.

Interface Etat et ses implémentations



Classe SimCanard 1/4

```
class SimCanard
{
    // Attributs
    private Etab etatEau_;
    private Etab etatTerre_;
    private Etab etatAir_;

    private Etab etat_;
```

...

Classe SimCanard 2/4

```
class SimCanard
{
    // Constructeur
    public SimCanard()
    {
        etatEau_ = new EtatEau(this);
        etatTerre_ = new EtatTerre(this);
        etatAir_ = new EtatAir(this);

        etat_ = etatEau_;
    }

    // action
    public void avancer()
    {
        etat_.avancer();
    }

    ...
}
```

Classe SimCanard 3/4

```
class SimCanard
{
    // Transitions
    public void palmeATerre()
    {
        etat_.palmeATerre();
    }

    public void decollage()
    {
        etat_.decollage();
    }

    public void trempette()
    {
        etat_.trempette();
    }

    ...
}
```

Classe SimCanard 4/4

```
class SimCanard
{
    // Pour changer l'état courant
    public void changeEtatEau()
    {
        etat_ = etatEau_;
    }

    public void changeEtatTerre()
    {
        etat_ = etatTerre_;
    }

    public void changeEtatAir()
    {
        etat_ = etatAir_;
    }
}
```

Les implémentations

Classe EtatEau

```
class EtatEau implements Etat {  
    private SimCanard sim_;  
  
    public EtatEau(SimCanard sim) { sim_ = sim; }  
  
    public void palmeATerre()  
{  
    sim_.changeEtatTerre();  
}  
  
    public void trempette() { print("Déjà à l'eau"); }  
  
    public void decollage()  
{  
    sim_.changeEtatAir();  
}  
  
    public void avancer() { print("Canard nage"); }  
}
```

Nouvelle structure

Avec cette nouvelle structure, nous avons :

- ▶ encapsulé le comportement de chaque état, et ainsi isolé ces comportements de la classe cliente (ici, le simulateur).
- ▶ supprimé cette liste de if else imbriqués qui n'en finissait plus.
- ▶ laissé les états indépendants entre eux : ils ne se connaissent pas les uns les autres, notre programme reste faiblement couplé.
- ▶ fixé le comportement des états (normalement, on ne modifie pas ce qui a été écrit) tout en laissant flexible le simulateur pour de futures extensions.

Quelques bons principes de POO

Moins d'héritage, plus de composition

Principe de composition : préférer la composition à l'héritage.

Moins de connaissance, plus d'indépendance

Principe de découplage : avoir des classes qui en sâche le moins possible sur les autres.

Moins de classes concrètes, plus d'abstraction

Principe d'abstraction : structurer autant que possible avec des interfaces et des classes abstraites.

Moins de modifications, plus de souplesse

Principe ouvert-fermé : les classes doivent être ouvertes à l'extension mais fermées à la modification.

Nous venons de voir le pattern **State** (**État**), un autre pattern de comportement.

Définition

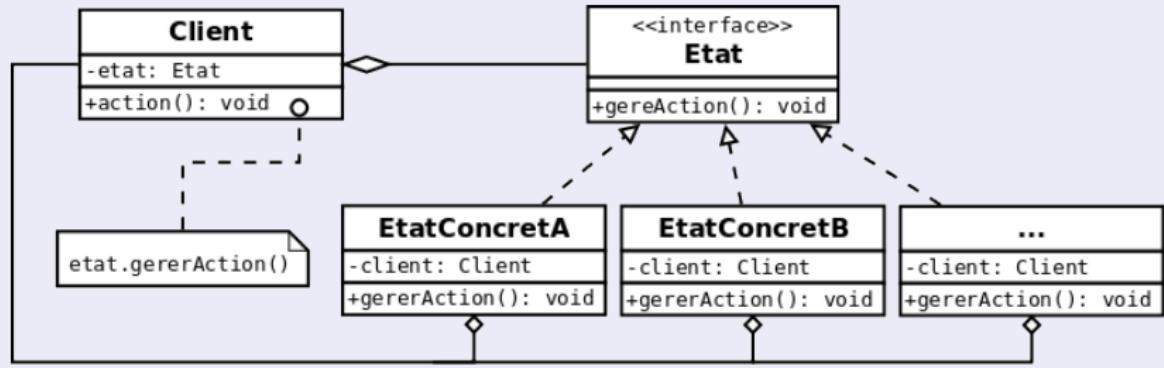
Le pattern State permet à un objet de modifier son comportement quand son état interne change. Tout se passera comme si l'objet changeait de classe.

Traduction de la 2ème phrase

Un objet qui change de comportement donne l'illusion d'être un objet d'une autre classe.

Pattern State

Diagramme UML



Quand utiliser le pattern State ?

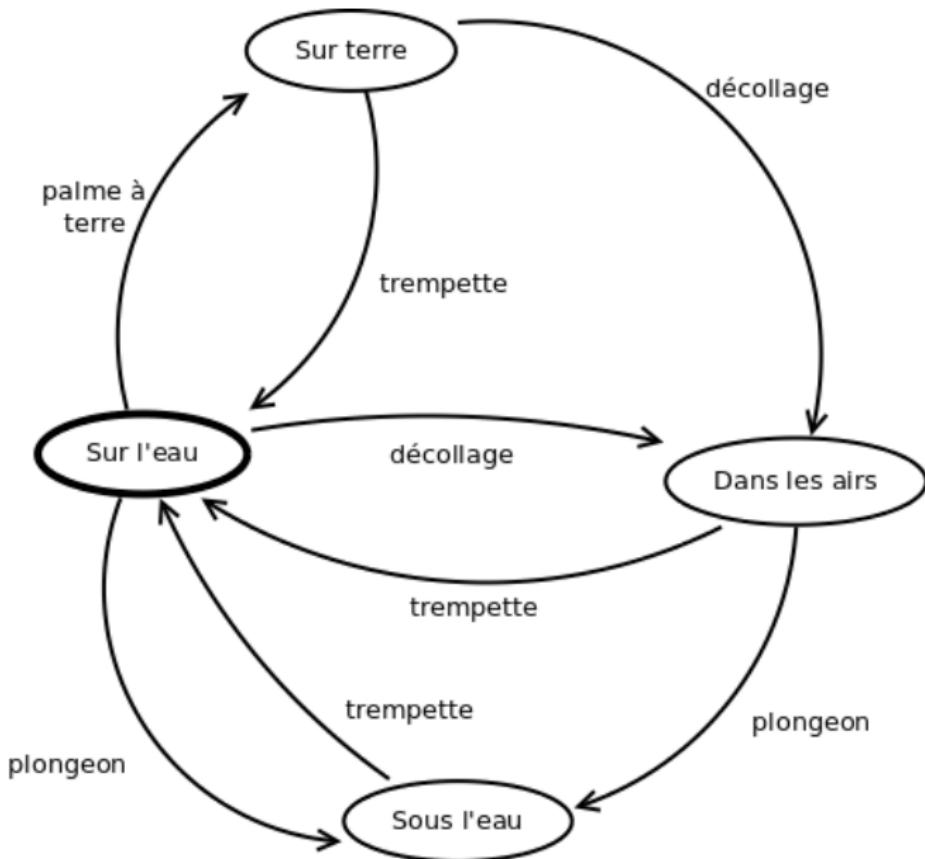
- ▶ On ne peut pas utiliser ce pattern seulement pour remplacer des if-else ou un switch.
- ▶ Il faut un changement d'état dans votre programme, et des **transitions** entre ces états.
- ▶ Il faut aussi des **actions** aux comportements différents selon l'état courant.

Comment repérer un changement d'état ?

Il y a changement d'état si :

- ▶ un ensemble de méthodes dans votre programme doit **changer de comportement** en fonction d'un contexte.
- ▶ ces comportements sont **mutuellement exclusifs**.

Retour sur le nouveau simulateur



Modifications 1/3

```
class SimCanard
{
    // Attributs
    private Etat etatSousEau_;

    // Dans le constructeur
    etatSousEau_ = new EtatSousEau(this);

    // Deux méthodes supplémentaires
    public void plongeon()
    {
        etat_.plongeon();
    }

    public void changeEtatSousEau()
    {
        etat_ = etatSousEau_;
    }
}
```

Modifications 2/3

```
interface Etat
{
    public void plongeon();
}

class EtatSousEau implements Etat
{
    ...
    public void palmeATerre() { }

    public void trempette()
    {
        sim_.changeEtatEau();
    }
    ...
}
```

Modifications 3/3

```
class EtatAir implements Etat
{
    public void plongeon()
    {
        sim_.changetEtatSousEau();
    }
}

class EtatEau implements Etat
{
    public void plongeon()
    {
        sim_.changetEtatSousEau();
    }
}

class EtatTerre implements Etat
{
    public void plongeon() { }
}
```

On peut faire encore mieux !

Java nous offre une possibilité pour encore améliorer notre code.

Laquelle ?

Feuille de TP 02

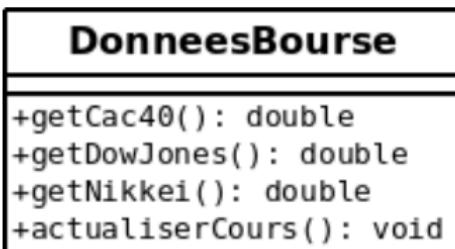
State

Observer

Affichage de la bourse

On vous donne la description d'une classe `DonneesBourse` qui contient des données sur le cours de la bourse en temps réel et on vous demande d'écrire un programme avec trois modes d'affichages :

- ▶ Cours actuel.
- ▶ Statistiques.
- ▶ Prédiction de demain.



Les méthodes de DonneesBourse

- ▶ Accesseurs sur les valeurs de DonneesBourse
- ▶ actualiserCours() appellée quand ces valeurs changent.

On commence à modifier DonneesBourse

Ajout de la méthode actualiserAffichages()

```
class DonneesBourse
{
    // Méthodes définies à l'origine
    // getCac40(), ...

    public void actualiserCours()
    {
        double cac40 = getCac40();
        double dowJones = getDowJones();
        double nikkei = getNikkei();

        cours.actualiser( cac40, dowJones, nikkei );
        stats.actualiser( cac40, dowJones, nikkei );
        prevision.actualiser( cac40, dowJones, nikkei );
    }
}
```

On peut déjà s'arrêter là

Avec cours, stats et prevision, on implémente des classes concrètes.

On peut déjà s'arrêter là

Avec cours, stats et prevision, on implémente des classes concrètes.

Et ?

On peut déjà s'arrêter là

Avec cours, stats et prevision, on implémente des classes concrètes.

Et ?

Rappelez-vous des canards...

Et alors la suppression ou l'ajout de types d'affichage devra passer par la modification du code.

On peut déjà s'arrêter là

Avec cours, stats et prevision, on implémente des classes concrètes.

Et ?

Rappelez-vous des canards...

Et alors la suppression ou l'ajout de types d'affichage devra passer par la modification du code.

Rappel : objectif "maintenance facile"

On souhaite faciliter la maintenance du code, donc devoir le modifier le moins possible.

Interface commune

```
cours.actualiser( cac40, dowJones, nikkei );
stats.actualiser( cac40, dowJones, nikkei );
prevision.actualiser( cac40, dowJones, nikkei );
```

Fonctionne comme un flux RSS

- ➊ Les news internationales du monde.fr vous intéresse, vous souhaitez rester informé.
- ➋ Vous vous abonnez au flux RSS des news internationales.
- ➌ Quand un nouvel article est publié, vous en êtes averti.
- ➍ Si cela ne vous intéresse plus, vous vous désabonnez.

Pattern Observer

Le pattern Observer fonctionne sur le même principe qu'un flux RSS, à une différence de vocabulaire :

- ▶ Les news du monde.fr est ici le **sujet**.
- ▶ Vous, vous êtes les **observateurs**.

Définition

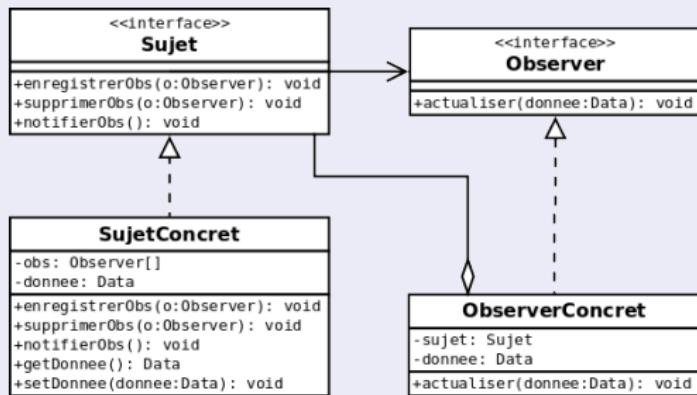
Le pattern Observer (Observateur) définit une relation entre objets de type un-à-plusieurs, de façon que, lorsqu'un objet change d'état, tous ceux qui en dépendent en soient notifiés et soient mis à jour automatiquement.

Relation sujet-observateurs

Les observateurs sont donc “dépendants” du sujet : quand l'état du sujet change, les observateurs en sont informés.

Pattern Observer

Diagramme UML



Communiquer sans se connaître

Les observateurs et les sujets ne savent pratiquement rien des uns les autres.

Principe de découplage

On vérifie le principe de découplage : des classes faiblement liées seront plus faciles à modifier sans mauvaise surprise.

On pourra aussi réutiliser ailleurs des sujets ou observateurs sans problème.

En détails

- ▶ Un sujet sait seulement qu'un observateur implémente une certaine interface. Il ne connaît pas les observateurs **concrets**.
- ▶ L'ajout et la suppression d'observateurs à n'importe quel moment ne change rien pour un sujet : il continuera à mettre à jour les informations comme d'habitude.
- ▶ Pas besoin de modifier le sujet pour ajouter un **nouveau type** d'observateurs.

En java, les interfaces

Interface Sujet

```
interface Sujet
{
    public void enregistrerObs(Observateur o);
    public void supprimerObs(Observateur o);
    public void notifierObs();
}
```

Interface Observateur et Affichage

```
interface Observateur
{
    public void actualiser(double cac40, double dowJones,
                           double nikkei);
}

interface Affichage
{
    public void afficher();
}
```

Class DonneesBourse 1/4

```
class DonneesBourse implements Sujet
{
    private ArrayList<Observateur> observateurs_;
    private double cac40_;
    private double dowJones_;
    private double nikkei_;

    public DonneesBourse()
    {
        observateurs_ = new ArrayList();
    }

    ...
}
```

Class DonneesBourse 2/4

```
class DonneesBourse implements Sujet
{
    ...

    public void enregistrerObs(Observateur o)
    {
        observateurs_.add(o);
    }

    public void supprimerObs(Observateur o)
    {
        observateurs_.remove(o);
    }

    ...
}
```

Class DonneesBourse 3/4

```
class DonneesBourse implements Sujet
{
    ...

    public void notifierObs()
    {
        for( int i = 0; i < observateurs_.size(); ++i)
        {
            observateurs_.get(i).actualiser(cac40_,
                                             dowJones_,
                                             nikkei_);
        }
    }

    ...
}
```

Class DonneesBourse 4/4

```
class DonneesBourse implements Sujet
{
    ...

    public void setCours(double cac40, double dowJones,
                         double nikkei)
    {
        cac40_      = cac40;
        dowJones_   = dowJones;
        nikkei_     = nikkei;

        notifierObs();
    }

    // Accesseurs et autres methodes
}
```

Classe AffichageCours 1/2

```
class AffichageCours implements Observateur, Affichage
{
    private double cac40_;
    private double dowJones_;
    private double nikkei_;
    private Sujet donneesBourse_;

    public AffichageCours(Sujet donneesBourse)
    {
        donneesBourse_ = donneesBourse;
        donneesBourse_.enregistrerObs(this);
    }

    ...
}
```

Classe AffichageCours 2/2

```
public void actualiser(double cac40, double dowJones,
                      double nikkei)
{
    cac40_      = cac40;
    dowJones_   = dowJones;
    nikkei_     = nikkei;
    afficher();
}

public void afficher()
{
    System.out.println(...);
}
```

Programme principal

```
class Bourse
{
    public static void main(String [] args)
    {
        DonneesBourse data = new DonneesBourse();

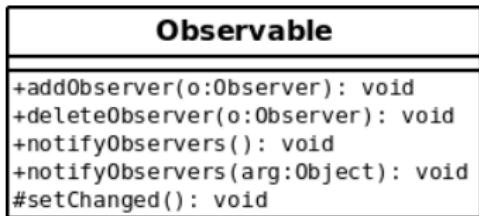
        AffichageCours affCours = new AffichageCours(data);
        AffichageStats affStats = new AffichageStats(data);
        AffichagePrev affPrev = new AffichagePrev(data);

        data.setCours(3376, 13443, 8596);
        data.setCours(3245, 13224, 8612);
        data.setCours(3189, 13378, 8703);
    }
}
```

Le pattern Observer est dans l'API Java !

java.util.Observer et java.util.Observable

- ▶ Un sujet concret **héritera** de la classe Observable de Java.
- ▶ Un observateur concret devra lui **implémenter** l'interface Observer.



Pour devenir Observateur

Comme avant : implémenter `Observer` et appeler la méthode `addObserver` du sujet (et `deleteObserver` pour se désinscrire).

Pour devenir Sujet

- ▶ Devenir `Observable` par héritage,
- ▶ **Nouveau** : d'abord appeler la méthode `setChanged` pour signaler que des valeurs ont changé.
- ▶ Appeler soit `notifyObservers()` soit `notifyObservers(Object arg)`.

Pour recevoir les changements

Un observateur appellera `update(Observerable o, Object arg)`.

- ▶ `arg` sera l'objet transmit par `notifyObservers(Object arg)`.
- ▶ Si un tel objet n'existe pas, `arg` sera null.

Intuitivement

- ▶ Avec `notifyObservers(Object arg)`, seules les **données encapsulées** dans `arg` sont transmises.
- ▶ Avec `notifyObservers()`, c'est l'objet **sujet entier** qui est transmis.

Comment est faite la classe Observable

Dans l'API

```
setChanged()  
{  
    changed = true;  
}  
  
notifyObservers(Object arg)  
{  
    if( changed )  
    {  
        // notifier tout le monde  
    }  
  
    changed = false;  
}  
  
notifyObservers()  
{  
    notifyObservers(null);  
}
```

Class DonneesBourse 1/2

```
import java.util.*;
class DonneesBourse extends Observable {

    // plus d'ArrayList pour les observateurs

    private double cac40_;
    private double dowJones_;
    private double nikkei_;

    public DonneesBourse() { } // RAS

    public void actualiserCours() {
        setChanged();
        notifyObservers();
    }

    ...
}
```

Class DonneesBourse 2/2

```
class DonneesBourse extends Observable {  
    ...  
  
    public void setCours(double cac40, double dowJones,  
                         double nikkei)  
    {  
        cac40_ = cac40;  
        dowJones_ = dowJones;  
        nikkei_ = nikkei;  
  
        actualiserCours(); // 1) setChanged, 2) notify  
    }  
}
```

Class AffichageCours 1/2

```
import java.util.*;
class AffichageCours implements Observer, Affichage
{
    private Observable donneesBourse_;
    private double cac40_;
    private double dowJones_;
    private double nikkei_;

    public AffichageCours(Observable data)
    {
        donneesBourse_ = data;
        data.addObserver(this);
    }

    ...
}
```

Class AffichageCours 2/2

```
class AffichageCours implements Observer, Affichage
{
    ...

    public void update(Observable data, Object arg)
    {
        if( data instanceof DonneesBourse )
        {
            DonneesBourse bourse = (DonneesBourse) data;
            cac40 = bourse.getCac40();
            dowJones = bourse.getDowJones();
            nikkei = bourse.getNikkei();

            afficher();
        }
    }
}
```

Deux problèmes avec ce pattern dans l'API

1) Ordre de notification

Dans la description officielle de Observable : *The order in which notifications will be delivered is unspecified.*

Ainsi, ne **jamais** utiliser le pattern Observer de l'API Java si vous avez un programme qui dépend d'un ordre de notification spécifique.

2) Observable est une classe

Observable n'étant pas une interface, votre sujet doit dériver de Observable et ne peut donc dériver de rien d'autre !

Vous ne pouvez même pas créer une instance de Observable et la composer avec vos classes car setChanged est **protected**, donc hors d'accès à une classe qui n'hérite pas de Observable !

Un monde observable

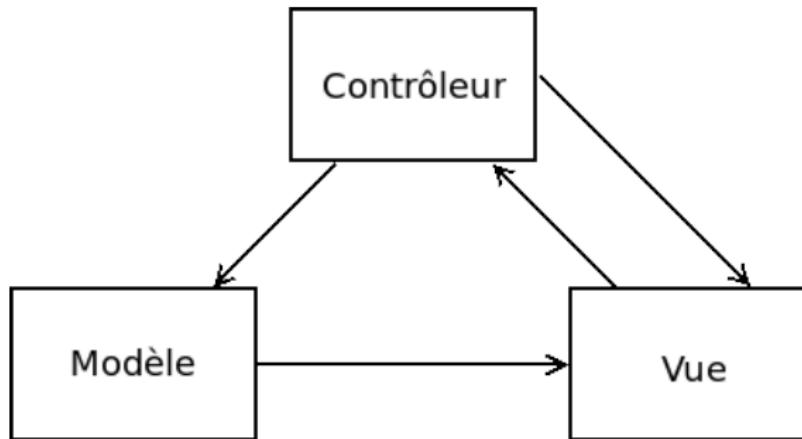
On retrouve le pattern Observer partout dans l'API de Java. Par exemple, il est omniprésent en Swing.

Exemple : JButton

Lors que l'on appelle la méthode `addActionListener` dans un JButton, on ajoute en réalité un observateur de notre bouton.

Pattern d'architecture *Modèle-Vue-Contrôleur*

- ▶ **Le modèle** : contient les données du programme.
- ▶ **La (les) vue(s)** : est l'IHM, affichant les données qu'on lui fournit.
- ▶ **Le contrôleur** : fait la synchronisation entre le modèle et les vues. C'est lui qui reçoit les requêtes de l'utilisateur.



Avantages du MVC

- ▶ Les trois modules sont faiblement couplés : par exemple on peut en réécrire le modèle sans modifier le reste.
- ▶ La synchronisation des vues grâce au pattern Observer.

"Inconvénient" du MVC

- ▶ Parfois complexe à mettre en place, surtout si le code n'est pas bien structuré dès le départ.

Feuille de TP 03

Observer

Decorator

Fil rouge du cours

On souhaite calculer le coût de quelque chose de hautement paramétrable. Par exemple, un café chez Starbucks.

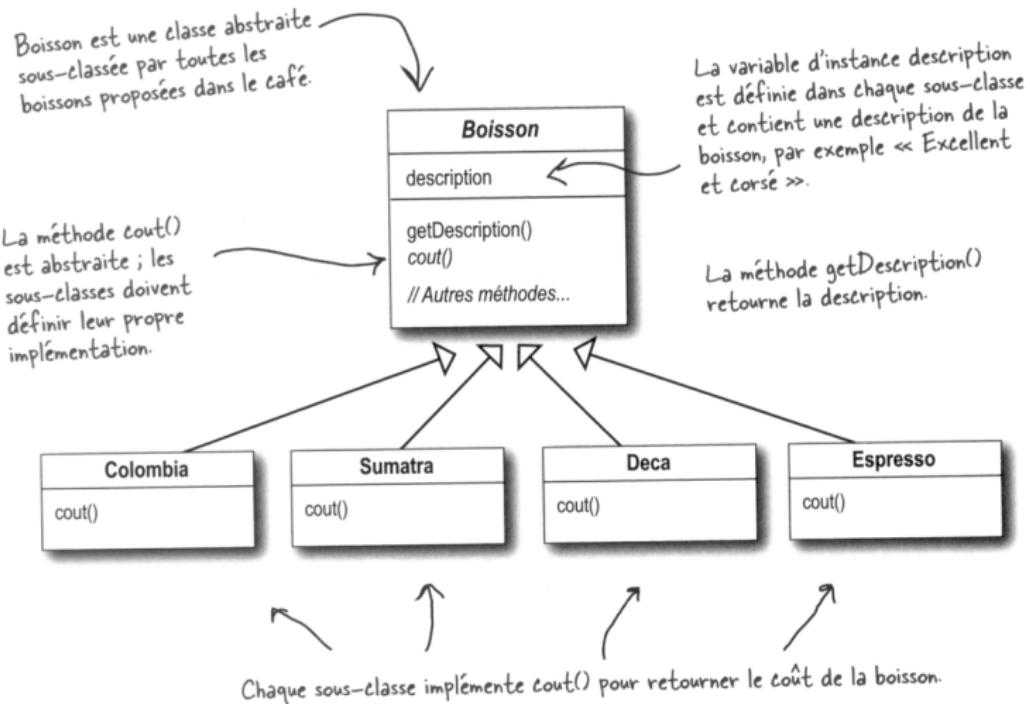
Principe du Starbucks

Vous choisissez votre café, puis :

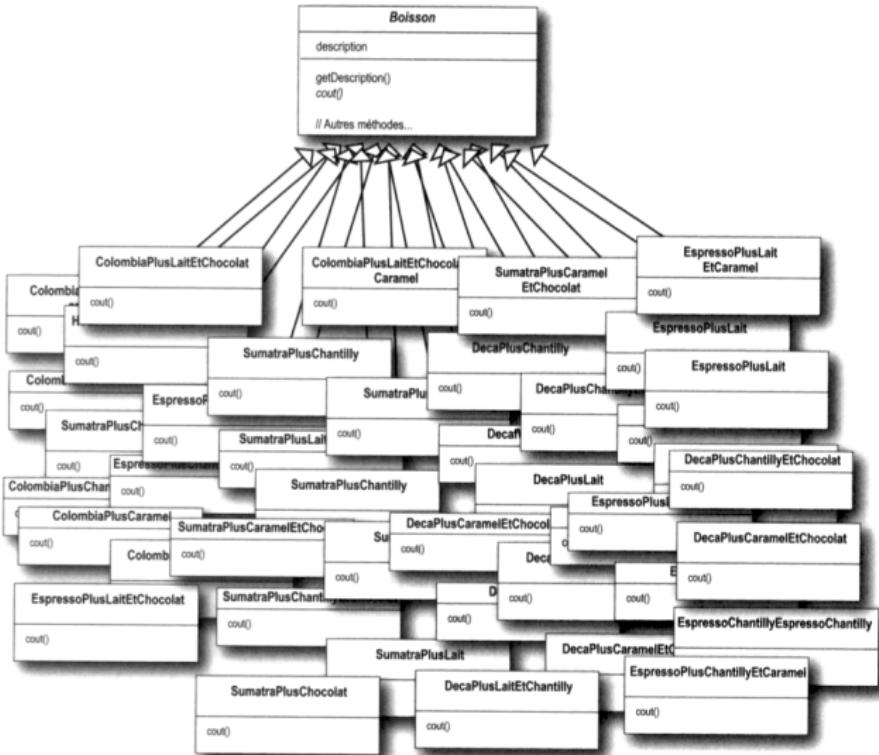
- ▶ la taille,
- ▶ avec ou sans chantilly,
- ▶ avec ou sans chocolat,
- ▶ avec ou sans caramel,
- ▶ ...

Bien sûr, ces choix ont un impact sur le prix final.

Commande de base



Commande complète



Amélioration

- ▶ Utiliser des booléens pour décrire une commande.
- ▶ Hériter ensuite de la classe abstraite Boisson.

```
<<abstract>>
Boisson
#description: String
#lait: bool
#caramel: bool
#chocolat: bool
#chantilly: bool
+getDescription(): String
+cout(): double
+aLait(): bool
+setLait(lait:bool): void
+aCaramel(): bool
+setCaramel(caramel:bool): void
+aChocolat(): bool
+setChocolat(chocolat:bool): void
+aChantilly(): bool
+setChantilly(chantilly:bool): void
```

Amélioration

```
class Colombia extends Boisson {  
    ...  
    public double cout() {  
        double colombia = 1.20;  
  
        return colombia +  
            (lait ? 0.25 : 0) +  
            (caramel ? 0.40 : 0) +  
            (chocolat ? 0.40 : 0) +  
            (chantilly ? 0.30 : 0);  
    }  
}
```

Question

Procéder ainsi n'est pas une bonne idée. Pourquoi ?

Question

Procéder ainsi n'est pas une bonne idée. Pourquoi ?

Fortes dépendances

Même problème qu'avec les canards :

- ▶ Changer les prix entraîne des modifications dans les classes.
- ▶ De même pour l'ajout/suppression d'ingrédients.
- ▶ Et si quelqu'un veut un double chocolat, comment on fait ?

Rappelez-vous : principe ouvert-fermé

Principe ouvert-fermé

Il faut garder en tête le principe ouvert-fermé : être ouvert à l'extension et fermé à la modification.

Encore un pattern pour nous sauver

On va utiliser le pattern Decorator.

Idée intuitive

Pour faire notre café colombien lait-caramel :

- ➊ Instancier un objet Colombia,
- ➋ Le décorer avec un objet Lait,
- ➌ Le décorer avec un objet Caramel,
- ➍ Appeler cout par délégations.

Décorer ?

Décorer = envelopper. Un décorateur **aura le même type que l'objet qu'il décore.**

Question

Pourquoi le décorateur doit avoir le même type que l'objet qu'il décore ?

Question

Pourquoi le décorateur doit avoir le même type que l'objet qu'il décore ?

Réponse

On veut pouvoir faire des choses comme cela :

```
Boisson cafe = new Colombia();
cafe = new Lait( cafe );
cafe = new Caramel( cafe );
```

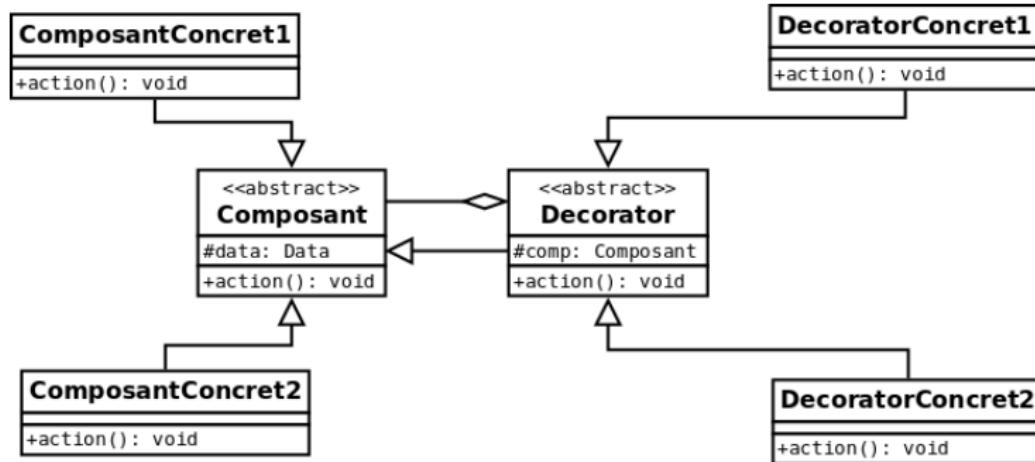
Nous savons que :

- ▶ Les décorateurs ont le même type (donc la même super-classe) que les objets qu'ils décorent.
- ▶ On peut donc manipuler et transmettre un objet décoré à la place de l'objet original.
- ▶ On peut décorer plusieurs fois un objet, éventuellement avec le même décorateur.

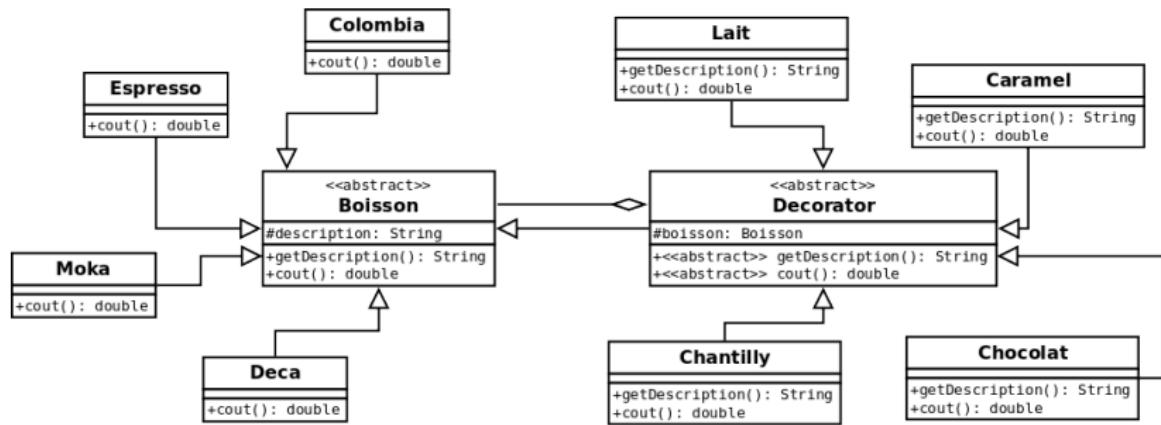
Pattern Decorator

Définition

Le pattern **Decorator** (Décorateur) attache **dynamiquement** des responsabilités supplémentaires à un objet. Il fournit une alternative souple à l'héritage, pour écrire les fonctionnalités.



Pour notre exemple



Euh...

On devait pas éviter l'héritage et préférer la composition ?

Si !

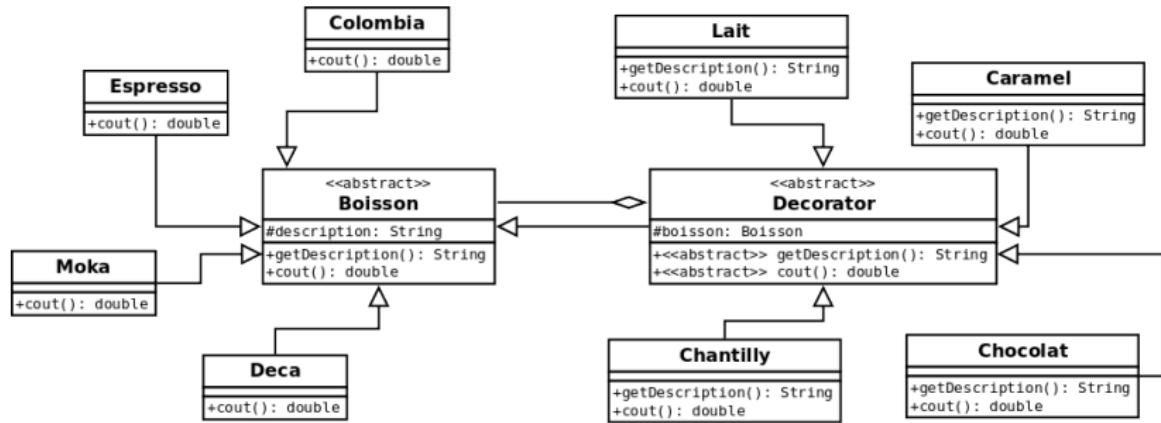
Et c'est ce que l'on fait. L'héritage Composant → Decorator (lire "Decorator hérite de Composant") ne sert ici qu'à **forcer le typage** du décorateur.

On a bien ici une composition : Decorator a une variable Composant en attribut.

Héritage vs. composition

- ▶ Decorator hérite de Composant pour qu'il y ait une correspondance de type, pas pour hériter du comportement de Composant.
- ▶ L'ajout d'un comportement se fait par composition, quand on ajoute un décorateur à un composant. (cafe = **new** Lait(cafe))
- ▶ Héritage = comportements déterminés statiquement, **à la compilation**.
- ▶ Composition = comportements ajoutés dynamiquement, **lors de l'exécution**.

Détails du code



Decorator vs. Strategy

Les patterns Strategy et Decorator semblent faire la même chose : ajouter des comportements. Qu'est ce qui les différencie ?

Decorator vs. Strategy

Les patterns Strategy et Decorator semblent faire la même chose : ajouter des comportements. Qu'est ce qui les différencie ?

Réponse

- ▶ Strategy permet de disposer d'une famille de comportements, et de passer facilement d'un comportement à l'autre.

Decorator vs. Strategy

Les patterns Strategy et Decorator semblent faire la même chose : ajouter des comportements. Qu'est ce qui les différencie ?

Réponse

- ▶ Strategy permet de disposer d'une famille de comportements, et de passer facilement d'un comportement à l'autre.
- ▶ Decorator permet d'activer (voire aussi de désactiver) dynamiquement un comportement.

Pour ceux qui connaissent Builder/Factory

Pattern Factory

Sert à fabriquer des objets en suivant des étapes de fabrications prédéfinis. Ex: la recette d'un gâteau.

Pattern Builder

Sert à fabriquer des objets en choisissant d'y inclure telle ou telle étape. Ex: le montage d'un PC.

Decorator vs. Factory/Builder

Factory et Builder servent **à créer** des objets ayant les comportements que l'on souhaite.

Decorator gère **dynamiquement** des objets déjà créés, et leur ajoute des comportements.

docs.oracle.com/javase/7/docs/api/index.html

Decorator est un pattern de **structure**

Patterns de structure : sert à structurer la composition des classes.

- ▶ Point de vue des classes : on utilise l'**héritage** pour la **correspondance de type** afin d'avoir une interface commune.
- ▶ Point de vue des objets : on utilise la **composition** pour avoir de **nouvelles fonctionnalités**.

Attention !

Decorator est un pattern **très** piège !

Feuille de TP 04

Decorator

Adapter et Façade

Prises électriques



Prises électriques



Prises électriques

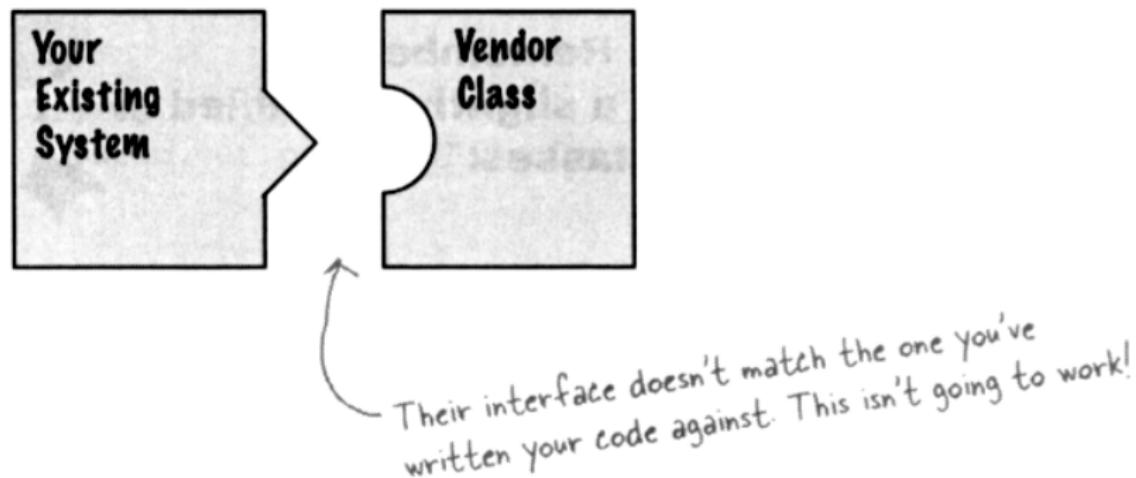


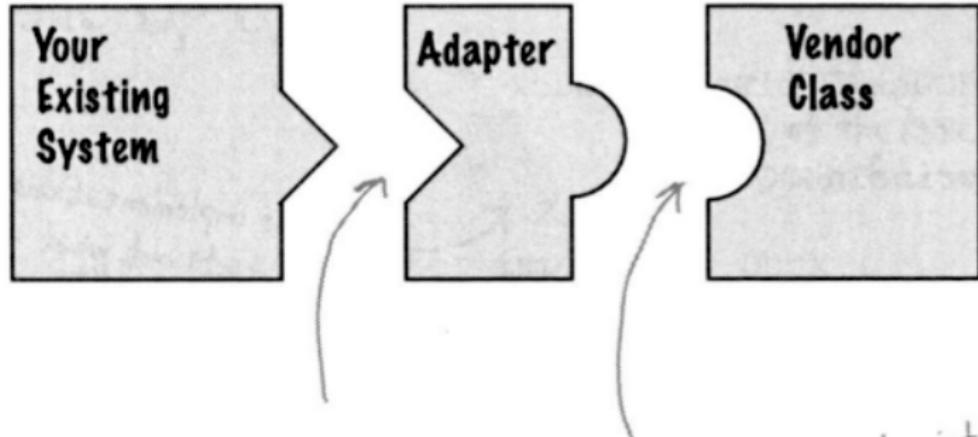
Prises électriques



Prises électriques

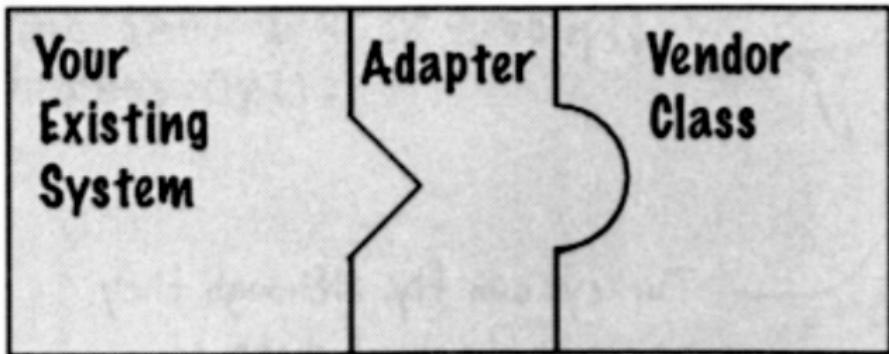






The adapter implements the interface your classes expect.

And talks to the vendor interface to service your requests.



↑
No code changes.

↑
New code.

↑
No code changes.

Exemple à base de canard

Canard

```
interface Canard
{
    public void voler();
    public void cancaner();
}

class Colvert implements Canard
{
    public void voler()
    {
        print("Je vole.");
    }

    public void cancaner()
    {
        print("Coin coin.");
    }
}
```

Exemple à base de canard

Dinde

```
interface Dinde
{
    public void voler();
    public void glousser();
}

class DindeSauvage implements Dinde
{
    public void voler()
    {
        print("Je peux voler mais j'ai la flemme.");
    }

    public void glousser()
    {
        print("Glou glou.");
    }
}
```

Exemple à base de canard

Interface vendeur

```
class Affichage
{
    public static void affiche(Canard canard)
    {
        canard.volter();
        canard.cancaner();
    }
}

class MaClasse
{
    Canard colvert = new Colvert();
    Affichage.affiche( colvert ); // OK

    Dinde dinde = new DindeSauvage();
    Affichage.affiche( dinde ); // KO, ne compile pas
}
```

Adapter l'interface vendeur

```
class AdapterDeDinde implements Canard
{
    private Dinde dinde_;

    public AdapterDeDinde( Dinde dinde )
    {
        dinde_ = dinde;
    }

    public void voler()
    {
        dinde_.voler();
    }

    public void cancaner()
    {
        dinde_.glousser();
    }
}
```

Dans ma classe

```
class MaClasse
{
    public static void main(String [] args)
    {
        Canard colvert = new Colvert();
        Affichage.affiche( colvert ); // OK

        Dinde dinde = new DindeSauvage();
        AdapterDeDinde adapter = new AdapterDeDinde( dinde );
        Affichage.affiche( adapter ); // OK
    }
}
```

À l'écran

Je vole.

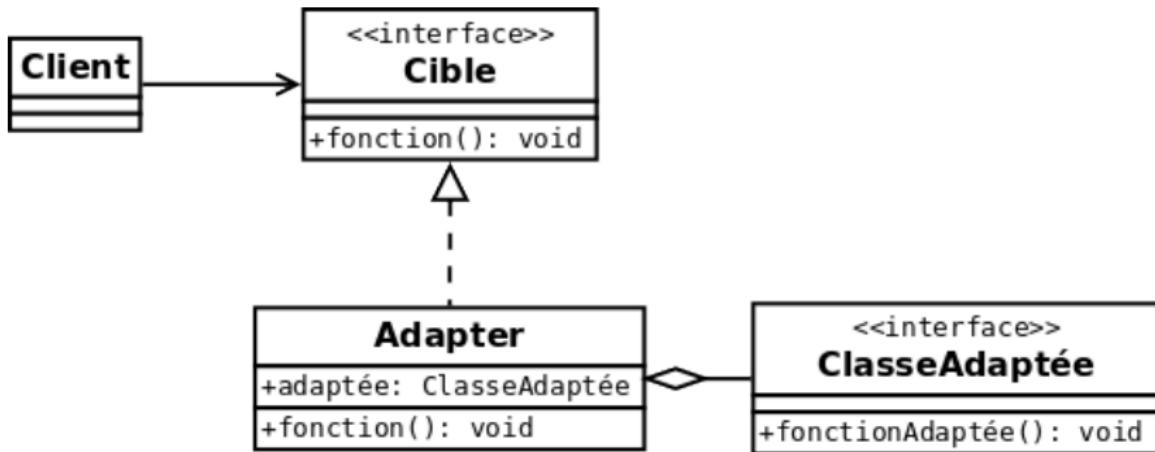
Coin coin.

Je peux voler mais j'ai la flemme.

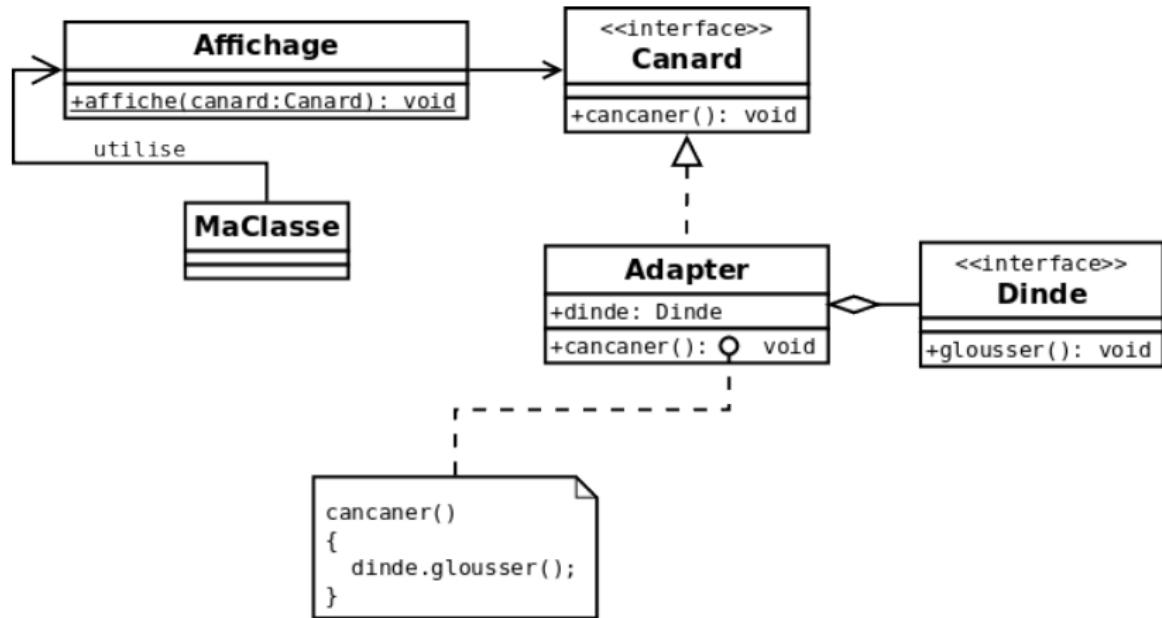
Glou glou.

Définition

Le pattern Adapter (Adaptateur) convertit l'interface d'une classe en une autre conforme à l'attente du client. L'adaptateur permet à des classes de collaborer, qui n'auraient pu le faire du fait d'interfaces incompatibles.



Dans notre exemple



Remarques

- ▶ Dans la vraie vie, ce dont vous n'aurez pas accès aux sources sera souvent **une bibliothèque** et **votre classe** sera souvent **un programme déjà existant** utilisant une autre bibliothèque dans laquelle vous souhaitez utiliser cette nouvelle bibliothèque.
- ▶ Si la bibliothèque change, vous n'avez qu'à modifier l'adaptateur et non votre programme.
- ▶ Vous n'adaptez que le nécessaire, pas tout la bibliothèque.

Adapter le nécessaire



Adapter le nécessaire



Un autre type d'Adapter

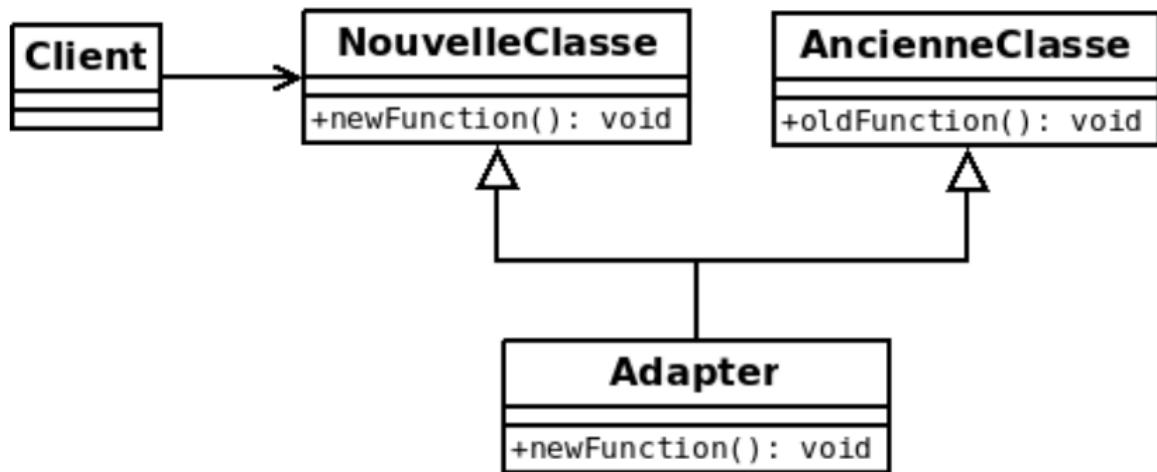
Ce que l'on vient de voir est un adaptateur **d'objets**.

Nous allons voir maintenant qu'il y a aussi des adaptateurs **de classes**.

Un autre type d'Adapter

Ce que l'on vient de voir est un adaptateur **d'objets**.

Nous allons voir maintenant qu'il y a aussi des adaptateurs **de classes**.



Compte bancaire d'origine

```
class CompteOriginal
{
    int num;

public:
    CompteOriginal(int num) : num(num) {}

    void afficheOriginal()
    {
        cout << "Numero d'un compte original "
            << num << endl;
    }
};
```

Interface des nouveaux comptes bancaires

```
class InterfaceNouveauCompte
{
public:
    virtual void affiche() = 0;
};
```

Nouveau compte bancaire !

```
class NouveauCompte : public InterfaceNouveauCompte
{
    string num;

public:
    NouveauCompte(string num) : num(num) {}

void affiche()
{
    cout << "Numero d'un nouveau compte "
        << num << endl;
}
};
```

Nouveau compte bancaire !

```
class Adapter :  
    public InterfaceNouveauCompte, public CompteOriginal  
{  
public:  
    Adapter( string num ) :  
        CompteOriginal( atoi( num.c_str() ) ) {}  
  
    void affiche()  
    {  
        this->afficheOriginal();  
    }  
};
```

Dans la classe cliente

```
int main()
{
    vector<InterfaceNouveauCompte*> listeComptes;
    listeComptes.push_back
        ( new NouveauCompte( "C1234-XYZ" ) );
    listeComptes.push_back
        ( new NouveauCompte( "R2D2C6P0" ) );
    listeComptes.push_back
        ( new Adapter( "123456789" ) );

    for( const auto &compte : listeComptes )
    {
        compte->affiche();
    }
}
```

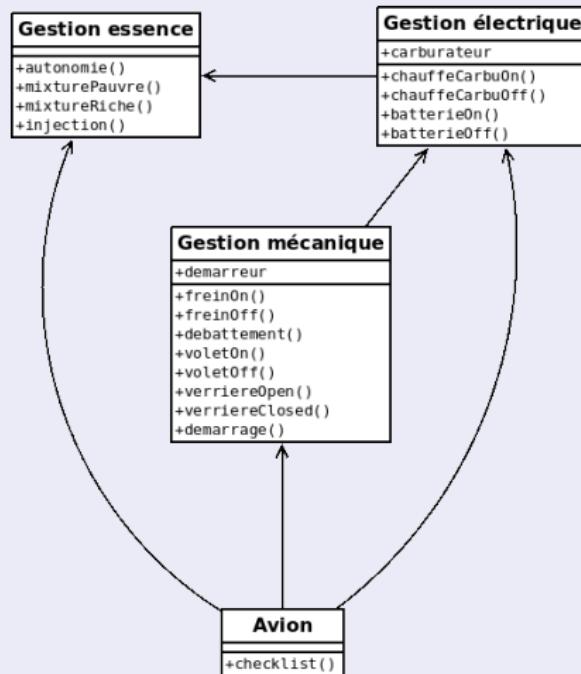
Pattern Façade

Comme Adapter, Façade va s'occuper d'interfaces, mais cette fois-ci pour **simplifier** une interface compliquée.

Check-list (simplifiée) pour un décollage

- ▶ Vérifier l'autonomie essence,
- ▶ Mettre le frein de park,
- ▶ Vérifier le débattement des commandes,
- ▶ Rentrer les volets,
- ▶ Fermer la verrière,
- ▶ Mettre la mixture huile - essence à riche,
- ▶ Chauffe carburateur on/off (suivant température extérieure),
- ▶ Batterie on,
- ▶ Injection essence,
- ▶ Démarrage.

Pseudo-diagramme



Pseudo-code Java

```
class Avion
{
    public void checklist()
    {
        essence.autonomie();
        meca.freinOn();
        meca.debattement();
        meca.voletOff();
        meca.verriereClosed();
        essence.mixtureRiche();
        elec.chauffeCarbuOn();
        elec.batterieOn();
        essence.injection();
        meca.demarrage();
    }
}
```

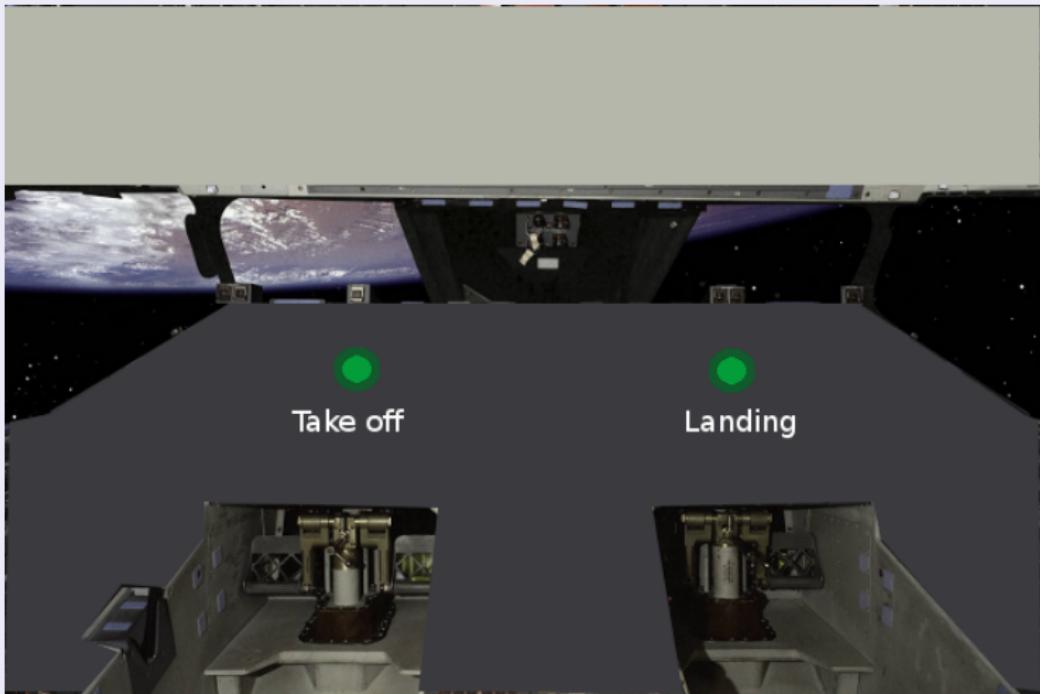
But

À l'origine

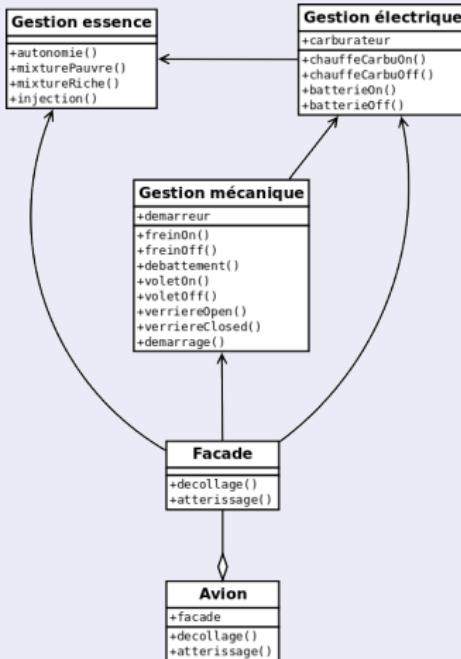


But

Avec le pattern Façade



Pseudo-diagramme



Pseudo-code Java

```
class Avion
{
    Facade facade;

    public void decollage()
    {
        facade.decollage();
    }
}

class Facade
{
    public void decollage()
    {
        ...
    }
}
```

Quels sont les intérêts ?

- ▶ Simplifier l'interface.
- ▶ Si les sous-systèmes changent, le code client lui ne change pas (car l'interface de la façade ne change pas).

Changement de sous-systèmes

Avant



Changement de sous-systèmes

Après



Changement de sous-systèmes

Après

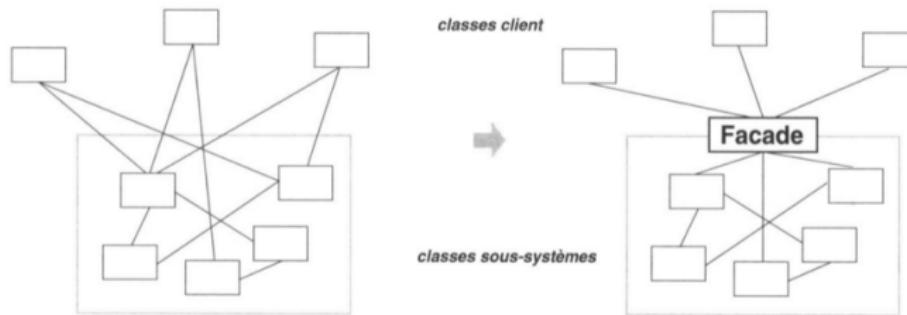


Définition

Le pattern Façade fournit une interface unifiée, à l'ensemble des interfaces d'un sous-système. La façade fournit une interface de plus haut niveau, qui rend le sous-système plus facile à utiliser.

Remarque

Façade **n'encapsule pas** le sous-système, donc si on veut mettre les mains dans le cambouis, on peut toujours !



Bon principe de POO

Loi de Déméter (ou principe de connaissance minimale)

Moins j'en sais, mieux je me porte.

Attention

Cette loi ne fonctionne pas pour vos examens.

En d'autres termes

Ne parlez qu'à vos amis immédiats. Communiquer avec le moins de classes possibles facilite la maintenance. Une classe n'a pas besoin de tout connaître.

Feuille de TP 05

Adapter et Façade

Factory Method et Abstract Factory

Polymorphisme

En POO, le polymorphisme nous permet de **changer le comportement du programme dynamiquement**, pendant son exécution.

Exemple

```
class A {  
    public int f() { return 42; }  
}  
  
class B extends A {  
    public int f() { return 24; }  
}  
  
class Main {  
    public static void main( String[] args ) {  
        A tab[] = new A[4];  
  
        tab[0] = new A();  
        tab[1] = new B();  
        tab[2] = new A();  
        tab[3] = new B();  
  
        for( int i = 0; i < 4; ++i )  
            System.out.println( tab[i].f() );  
    }  
}
```

Polymorphisme

En POO, le polymorphisme nous permet de **changer le comportement du programme dynamiquement**, pendant son exécution.

Et pour la création d'objet ?

A t-on la même flexibilité pour la création d'objet ?

Instancier un objet en Java

```
TypeVariable variable;  
variable = new TypeObjet();
```

Instancier un objet en Java

```
TypeVariable variable;  
variable = new TypeObjet();
```

Passage obligatoire

On doit bien avoir le mot-clé **new** pour instancier un objet, et derrière ce mot-clé doit apparaître un type **qui doit être connu à la compilation.**

Instancier un objet en Java

```
TypeVariable variable;  
variable = new TypeObjet();
```

Passage obligatoire

On doit bien avoir le mot-clé **new** pour instancier un objet, et derrière ce mot-clé doit apparaître un type **qui doit être connu à la compilation**.

Différence fondamentale

On a donc une différence fondamentale (de souplesse) entre la création d'objet et l'invocation de méthodes.

So what?

Ok, mais quand on a besoin d'instancier un objet, c'est qu'on le connaît à l'avance, non ?

Pas forcément

- ▶ Laisser la connaissance de l'objet à créer à une autre entité (par exemple une méthode qui s'occupera de créer l'objet).
- ▶ Vous avez cette connaissance (sous forme d'une string, par exemple), mais pas accès au constructeur.

Patterns Factory

Instancier dynamiquement, c'est possible en POO grâce aux patterns **Factory**. Avec Observer, ils sont parmi les patterns **les plus utilisés**.

Exemple de ce cours : la pizzeria

On va créer dynamiquement des pizzas.

Commander une pizza

Classe Pizzeria

```
class Pizzeria
{
    ...

    public Pizza commander()
    {
        Pizza pizza = new Pizza();

        pizza.preparer();
        pizza.cuire();
        pizza.couper();
        pizza.dansLaBoite();

        return pizza;
    }
}
```

Commander une certaine pizza

Pas n'importe quelle pizza

```
public Pizza commander( String type ) {  
    Pizza pizza;  
  
    if( type.equals( "Margherita" ) )  
        pizza = new Margherita();  
    else if( type.equals( "Royale" ) )  
        pizza = new Royale();  
    else if( type.equals( "Calzone" ) )  
        pizza = new Calzone();  
  
    pizza.preparer();  
    pizza.cuire();  
    pizza.couper();  
    pizza.dansLaBoite();  
  
    return pizza;  
}  
}
```

Changements dans le menu

Vous ne vendez pratiquement aucune Margherita, et en contre partie vous avez une forte demande pour des pizzas végétariennes que vous ne proposez pas encore.

Modifications

Vous devez pas mal modifier votre classe Pizzeria !

- ▶ Supprimer la Margherita des if.
- ▶ Ajouter une nouvelle pizza végétarienne.

Ce qui change et ne change pas

Nouveau menu

```
public Pizza commander( String type ) {  
    Pizza pizza;  
  
    if( type.equals( "Vegetarienne" ) )  
        pizza = new Vegetarienne();  
    else if( type.equals( "Royale" ) )  
        pizza = new Royale();  
    else if( type.equals( "Calzone" ) )  
        pizza = new Calzone();  
  
    pizza.preparer();  
    pizza.cuire();  
    pizza.couper();  
    pizza.dansLaBoite();  
  
    return pizza;  
}  
}
```

Rappel : bon principe de POO

Encapsuler ce qui change, ou est susceptible de changer.

Ce qui change

On extrait ce qui change (le bloc if/else) et on le case dans un méthode quelque part.

Ce qui ne bouge pas

On garde le reste (appels des méthodes preparer(), cuire(), ...) dans la classe Pizzeria.

Classe FactorySimple

```
class FactorySimple
{
    public Pizza fairePizza( String type )
    {
        Pizza pizza;

        if( type.equals( "Vegetarienne" ) )
            pizza = new Vegetarienne();
        else if( type.equals( "Royale" ) )
            pizza = new Royale();
        else if( type.equals( "Calzone" ) )
            pizza = new Calzone();

        return pizza;
    }
}
```

La nouvelle pizzeria

Nouvelle classe Pizzeria

```
class Pizzeria
{
    private FabriqueSimple fs_;
    public Pizzeria( FabriqueSimple fs ) { fs_ = fs; }
    ...

    public Pizza commander( String type )
    {
        Pizza pizza = fs_.fairePizza( type );

        //Remarque : plus de new dans la méthode !

        pizza.preparer();
        pizza.cuire();
        pizza.couper();
        pizza.dansLaBoite();

        return pizza;
    }
}
```

Euh...

On n'a fait que de déplacer le problème dans une autre classe...

C'est vrai...

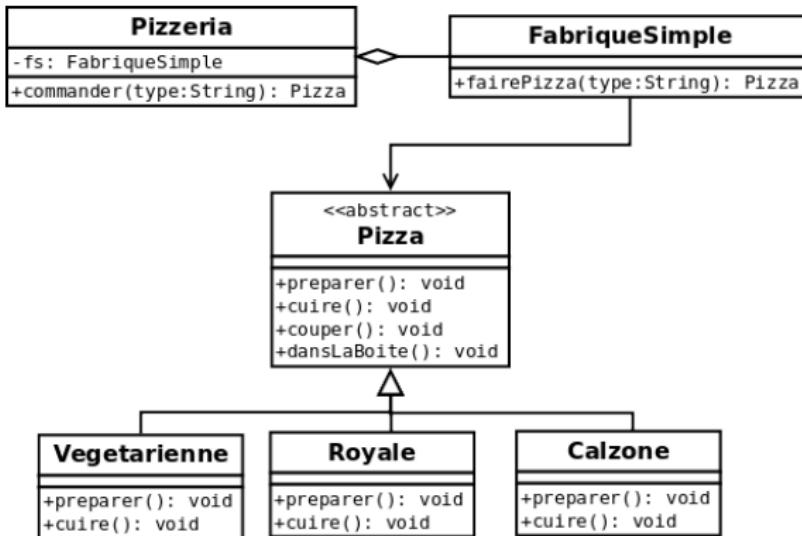
La solution n'est pas idéale, mais à quand même son utilité : si plusieurs classes font appel à la méthode *fairePizza(String)* de *FabriqueSimple*, on y gagne.

Exemple

On pourrait avoir une classe *Livraison* qui a aussi besoin d'instancier un objet Pizza, ou encore une classe *Menu*.

Simple Factory

La Simple Factory (fabrique simple) n'est communément pas admise comme un pattern. Il s'agit plus d'une astuce de programmation.



Revenons à nos pizzas

Votre pizzeria fait un carton !

Du coup, expansion mondiale : franchise à Nantes, New-York et Tokyo.

Vous souhaitez :

- ▶ que les enseignes suivent toutes la même procédure...
- ▶ ...mais adapter les pizzas au goût local.

Différentes pizzas du même type

```
tokyoPizzeria.commander( "Royale" );
```

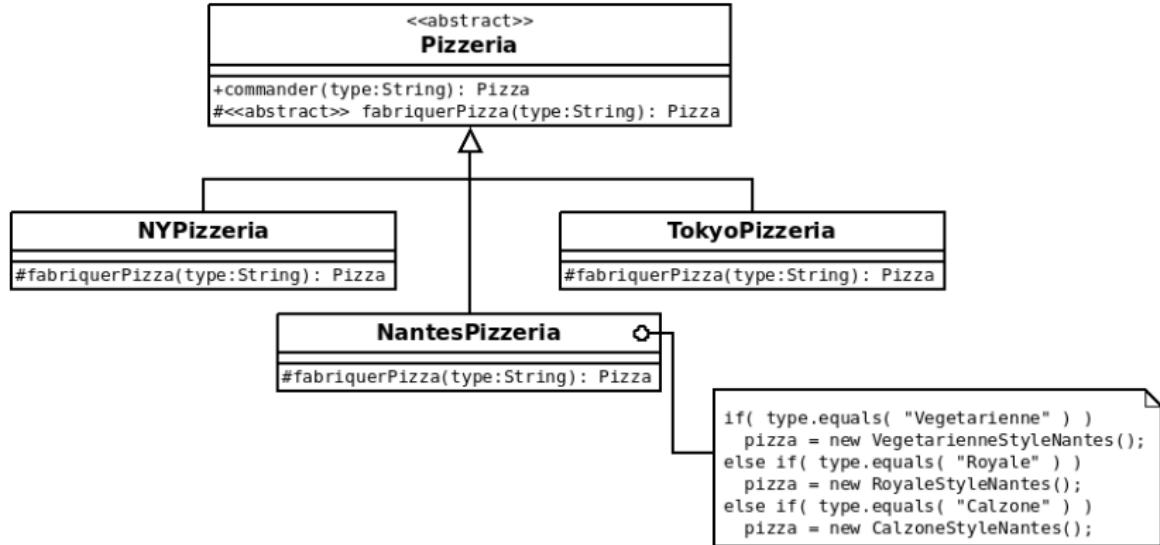
≠

```
nantesPizzeria.commander( "Royale" );
```

Pizzeria abstraite

```
abstract class Pizzeria {  
    public Pizza commander( String type ) {  
        Pizza pizza = fabriquerPizza( type );  
  
        pizza.preparer();  
        pizza.cuire();  
        pizza.couper();  
        pizza.dansLaBoite();  
  
        return pizza;  
    }  
  
    protected abstract Pizza fabriquerPizza(String type);  
}
```

Pizzeria abstraite



- ➊ Choix de sa pizzeria.

```
Pizzeria nantesPizzeria = new NantesPizzeria();
```

- ➋ Commande de son type de pizza.

```
nantesPizzeria.commande( "Royale" );
```

- ➌ Appel de la fabrique simple.

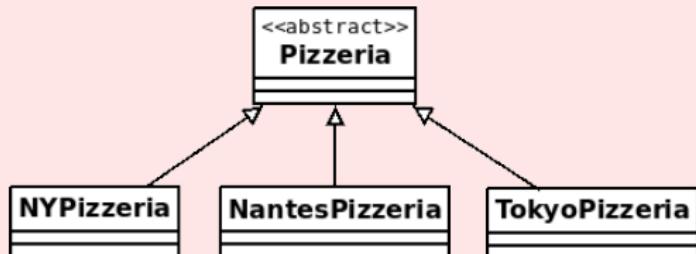
```
Pizza pizza = fabriquePizza( "Royale" );
```

- ➍ Une fois la pizza créée, appels des méthodes communes à toutes les pizzerias et les pizzas.

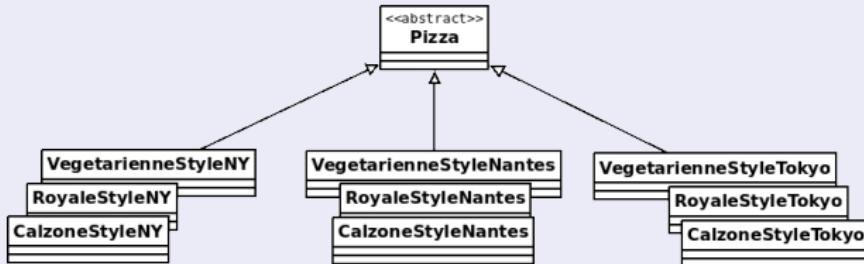
```
pizza.preparer(); // Dans le code, on ne connaît pas  
pizza.cuire(); // la classe concrete de pizza.  
pizza.couper(); // Pizzeria est donc decouplée  
pizza.dansLaBoite(); // de la classe Pizza !
```

Le point sur ce que l'on connaît

Classes créatrices

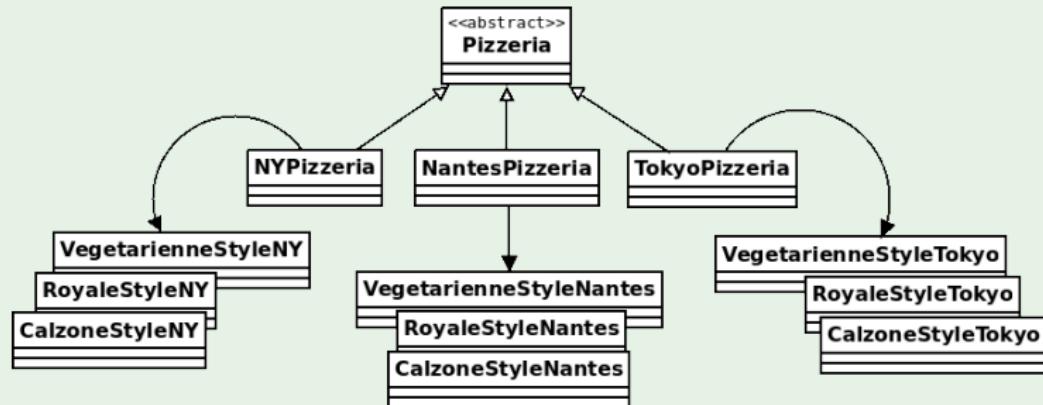


Classes produits



Pattern Factory Method

L'idée

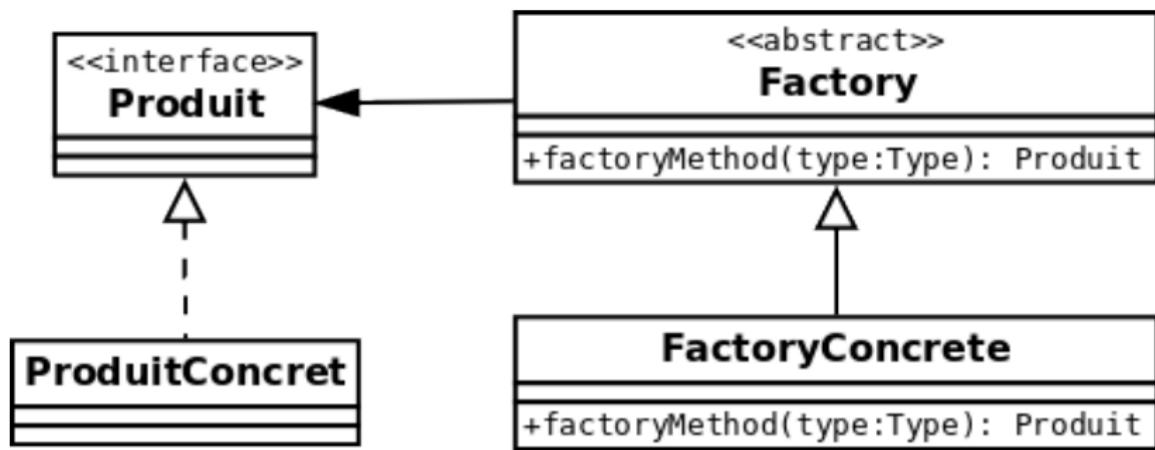


Point-clé

Le point-clé des patterns Factory est l'**encapsulation des connaissances pour la création** des objets souhaités.

Définition

Le pattern **Factory Method**, aussi juste appelé **Factory** (pattern Fabrique ou Fabrication en français), définit une interface pour la création d'un objet, mais en laissant à des sous-classes le *choix* des classes à instancier. Factory Method permet à une classe de déléguer l'instanciation à des sous-classes.



1ère manière de voir les choses

Factory Method = Pattern Strategy où les algorithmes à changer sont des Simple Factories.

2ème manière de voir les choses

Factory Method = constructeur virtuel (autorisant le polymorphisme).

Flashback

... en laissant à des sous-classes le **choix** des classes à instancier.

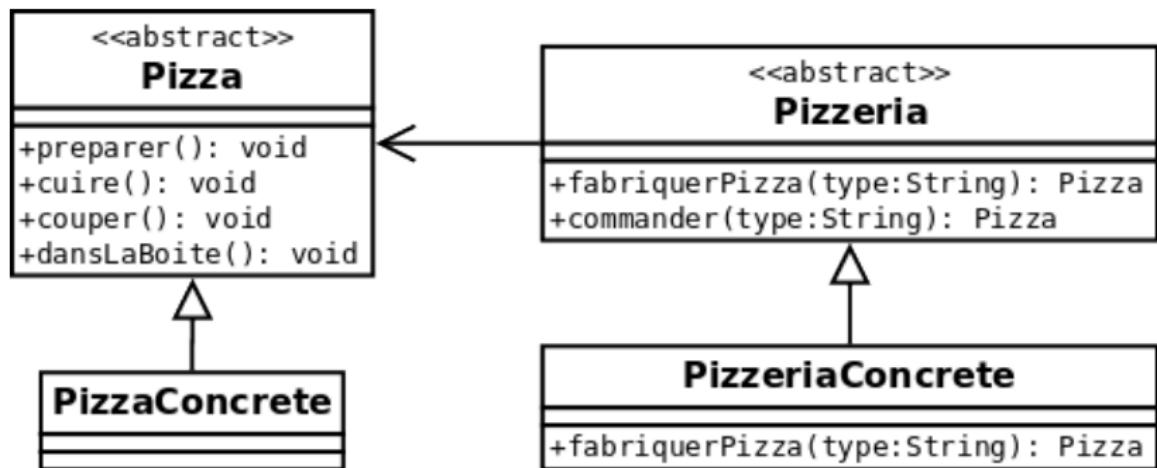
Kezako?

C'est bien sûr le programmeur ou l'utilisateur qui choisit les classes à instancier. Ce qu'il faut lire, c'est que l'interface de création **ne connaît pas la classe concrète à instancier** et laisse le soin aux sous-classes de s'en charger.

Exemple

Pizzeria sait qu'il faut fabriquer une pizza avec *fabriquerPizza("Royale")*, mais à aucune idée de ce qu'est une pizza royale, ni comment l'instancier. Par contre, *NantesPizzeria* sait faire des royales nantaises, elle !

Pattern Factory Method



Encapsuler par Factory Method

L'encapsulation du code de création d'objets par une Factory Method permet :

- ▶ de concentrer le code de création d'objets à un endroit, facilitant la maintenance.
- ▶ de pouvoir choisir dynamiquement le type de l'objet à instancier.
- ▶ **d'avoir des classes clientes qui dépendent d'une interface pour créer des objets, et non de classes concrètes**

Principe d'inversion de dépendance

Dépendez d'abstraction, pas de classes concrètes.

Vocabulaire

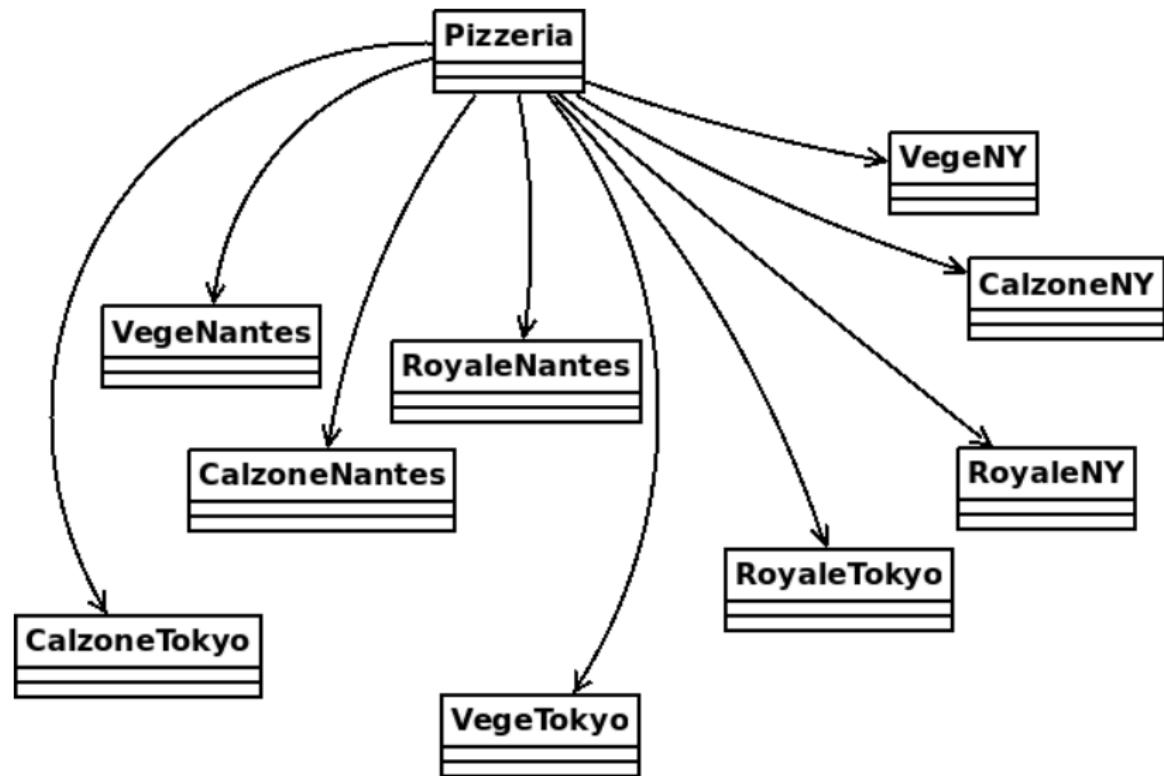
- ▶ Classe de **bas niveau** : classe définissant un comportement qui lui est propre. Ex: Pizza.
- ▶ Classe de **haut niveau** : classe dont le comportement dépend du comportement des classes qu'elle utilise. Ex: Pizzeria (qui dépend de Pizza).

Principe d'inversion de dépendance

Une classe de haut niveau doit dépendre de classes de bas niveaux abstraites et non concrètes.

Forte dépendance de la classe Pizzeria

```
public Pizza commander( String type ) {  
    Pizza pizza;  
  
    if( type.equals( "VegetarienneStyleNantes" ) )  
        pizza = new VegetarienneStyleNantes();  
    else if( type.equals( "RoyaleStyleNantes" ) )  
        pizza = new RoyaleStyleNantes();  
    else if( type.equals( "CalzoneStyleNantes" ) )  
        pizza = new CalzoneStyleNantes();  
    else if( type.equals( "VegetarienneStyleNY" ) )  
        pizza = new VegetarienneStyleNY();  
    else if( type.equals( "RoyaleStyleNY" ) )  
        pizza = new RoyaleStyleNY();  
    else if( type.equals( "CalzoneStyleNY" ) )  
        pizza = new CalzoneStyleNY();  
    ...  
}
```



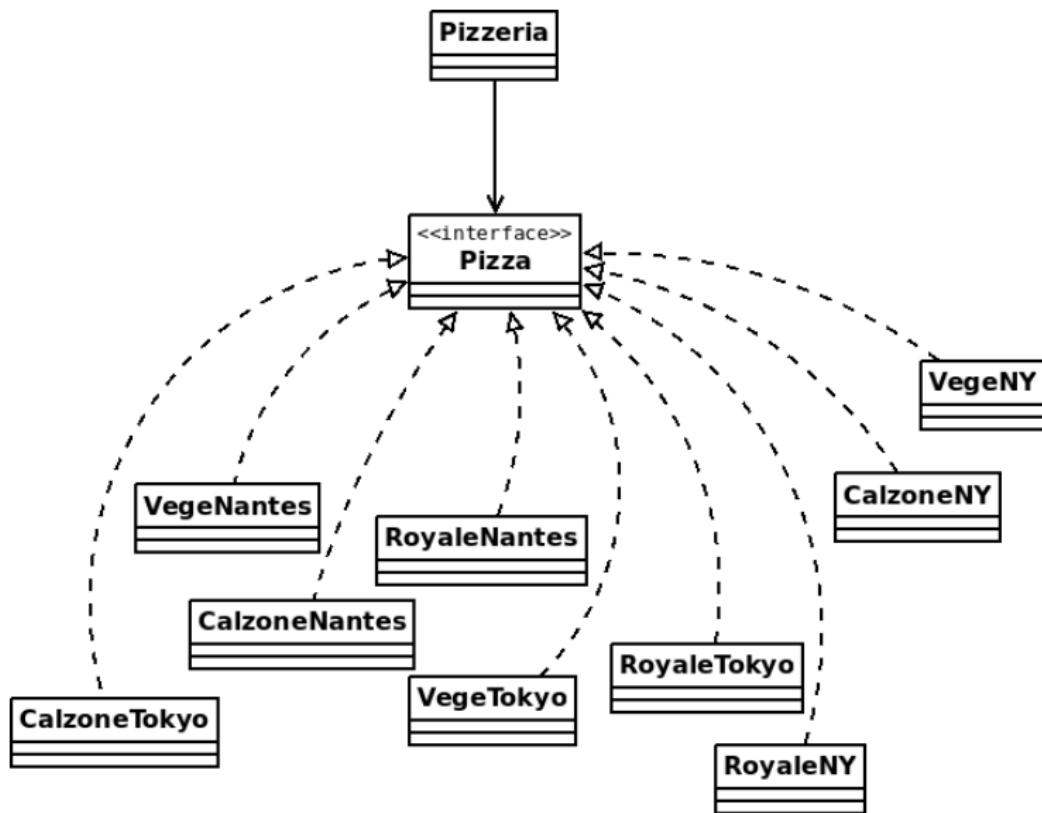
Mauvaise structure

Si une classe concrète change ou est supprimée (ou ajoutée), il faut **modifier** la classe de haut niveau Pizzeria !

Bonne structure

Avec une classe abstraite intermédiaire, on n'a plus ce problème.
Observez dans le prochain schéma l'**inversion des flèches** (donc des dépendances) !

Illustration



Ok, mais...

On a réglé le problème de dépendance de la classe de haut niveau, mais avant la pizzeria avait un accès à chaque pizza concrète, donc on pouvait **instancier** la pizza concrète que l'on souhaitait. Comment on fait maintenant ?

Ok, mais...

On a réglé le problème de dépendance de la classe de haut niveau, mais avant la pizzeria avait un accès à chaque pizza concrète, donc on pouvait **instancier** la pizza concrète que l'on souhaitait. Comment on fait maintenant ?

Ben justement...

Le pattern Factory Method est là pour ça !

Approches **top-down** et **bottom-up**

Factory Method

Encapsule la création d'objets et permet à une classe de créer **un objet en déléguant ce travail à ses sous-classes.**

Abstract Factory

Encapsule la création d'objets et permet à une classe de créer **une famille d'objets.**

Factory Method vs. Abstract Factory

- ▶ Un objet **vs.** une famille d'objets.
- ▶ Délegation aux sous-classes **vs.** création en passant par de la composition.

Problème de notre code avec Factory Method

Problème

On doit écrire une classe concrète pour chaque type de produit de chaque fabrique !

Dans notre exemple

Classes VegetarienneStyleNantes, RoyaleStyleNantes,
VegetarienneStyleTokyo, ...

Dommage

C'est d'autant plus bête que ces produits sont tous plus ou moins
composés des mêmes choses (une pâte, une sauce, ...).

Une famille d'objets

Famille : ensemble d'objets dont la **fabrication est similaire**, partageant les mêmes sous-éléments.

Dans notre exemple

Considérons la pizzeria de Nantes. Il est facile d'imaginer qu'elle va utiliser le même type de sauce tomate, de pâte, etc, pour chacune de ses pizzas.

Donc les pizzas de la pizzeria de Nantes formeront une famille.

Nouvelle classe Pizza

Nouvelle classe Pizza

```
abstract class Pizza
{
    private String nom_;
    private Pate pate_;
    private Sauce sauce_;
    private Fromage fromage_;

    ...

    public abstract void prepare();

    ...
}
```

Flexibilité

On souhaite pouvoir instancier une liste d'ingrédients pour nos pizzas, en laissant la possibilité de **changer facilement un ingrédient** (la pâte par exemple).

Classe CreerIngredients

```
interface CreerIngredients
{
    public Pate creerPate();
    public Sauce creerSauce();
    public Fromage creerFromage();
    ...
}
```

Fabrique concrète d'ingrédients

Classe IngredientsNantes

```
class IngredientsNantes implements CreerIngredients
{
    public Pate creerPate() {
        return new PateFine();
    }

    public Sauce creerSauce() {
        return new SauceTomatePiquante();
    }

    public Fromage creerFromage() {
        return new Mozzarella();
    }

    ...
}
```

Pizza générique aux fruits de mer

Classe PizzaFruitsDeMer

```
class PizzaFruitsDeMer extends Pizza
{
    private CreerIngredients ci_;

    public PizzaFruitsDeMer( CreerIngredients ci )
    {
        ci_ = ci;
    }

    public void prepare()
    {
        pate_          = ci_.creerPate();
        sauce_         = ci_.creerSauce();
        crevette_      = ci_.creerCrevette();
        moule_         = ci_.creerMoule();
        palourde_     = ci_.creerPalourde();
    }
}
```

Passer d'une pizza aux fruits de mer à l'autre

```
Pizza pizza;  
CreerIngredients ci;  
  
// pizza nantaise aux fruits de mer  
ci = new IngredientsNantes();  
pizza = new PizzaFruitsDeMer( ci );  
  
// pizza parisienne aux fruits de mer  
ci = new IngredientsParis();  
pizza = new PizzaFruitsDeMer( ci );
```

Pizzas de la pizzeria de Nantes

Nouvelle classe PizzeriaNantes

```
class PizzeriaNantes extends Pizzeria {
    private CreerIngredients ci_;
    private Pizza pizza_;

    public PizzeriaNantes() {
        ci_ = new IngredientsNantes();
    }

    public Pizza commander( String type ) {
        if( type.equals( "Vegetarienne" ) )
            pizza_ = new PizzaVegetarienne( ci_ );
        else if( type.equals( "Royale" ) )
            pizza_ = new PizzaRoyale( ci_ );
        ...
        pizza_. preparer();
        pizza_. cuire();
        ...
    }
}
```

Remarque importante

Tout comme pour la Factory Method, l'interface de création d'une Abstract Factory **ne dépend pas** des classes de bas niveaux.

Classe CreerIngredients

```
interface CreerIngredients
{
    public Pate creerPate();
    public Sauce creerSauce();
    public Fromage creerFromage();
    ...
}
```

Famille d'objets

Par définition de l'Abstract Factory : toutes les pizzas d'une même pizzeria doivent **partager les mêmes types d'ingrédients** (un même type de sauce, un même type de pâte, ...).

Familles isomorphiques

Par contre, les familles doivent être isomorphiques (de même forme). Une pizza aux fruits de mer **doit** contenir des crevettes, qu'elle soit de la famille de Nantes, New York ou Tokyo !

Bon choix ?

Si cela pose problème pour votre programme, Abstract Factory n'était pas le bon choix.

Classes concrètes des ingrédients

Potentiellement beaucoup de classes concrètes

Une classe concrète par ingrédients ! Même problème qu'avec Factory Method ?

Par exemple pour les sauces

Classes SauceTomate, SauceTomatePiquante, CremeFraiche, SauceMarinara, ...

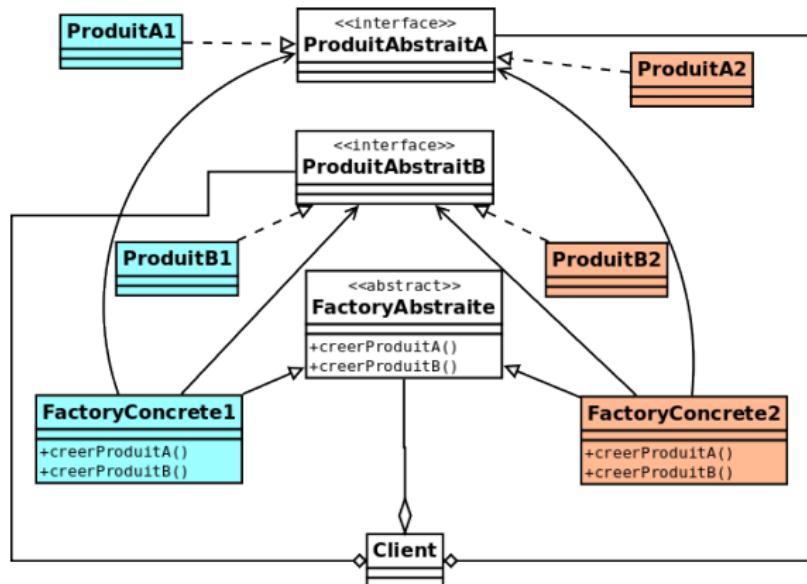
Réutilisation !

Cependant on peut **réutiliser ces objets** si des pizzas partagent certains ingrédients identiques.

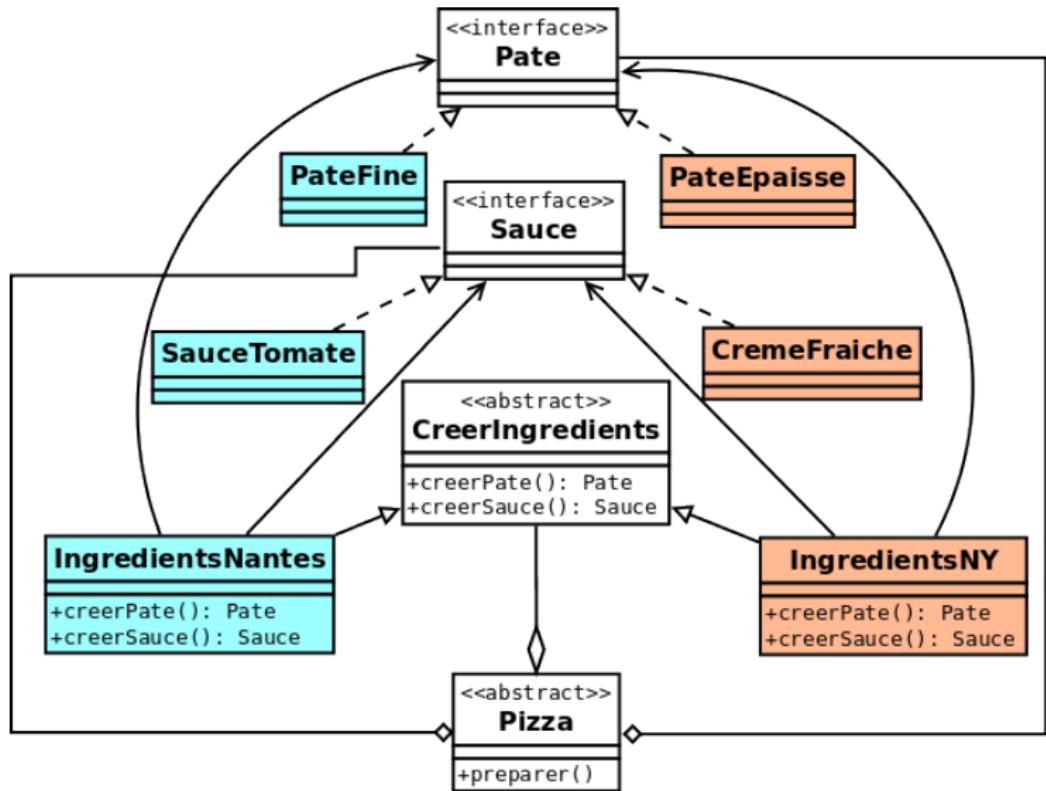
Pattern Abstract Factory

Définition

Le pattern **Abstract Factory** (Fabrique abstraite) fournit une interface pour créer des familles d'objets apparentés ou interdépendants, sans qu'il soit nécessaire de spécifier leurs classes concrètes.



Pattern Abstract Factory



Le nom

Pourquoi Abstract Factory s'appelle **Abstract** Factory ?

Le nom

Pourquoi Abstract Factory s'appelle **Abstract** Factory ?

L'implémentation

L'Abstract Factory est en quelque sorte plusieurs Factory Methods placées dans une classe abstraite (ou une interface). Voir CreerIngredients.

Classe CreerIngredients

```
interface CreerIngredients // classe ‘‘abstraite’’
{
    public Pate creerPate(); // factory method pour
                           // creer l’objet pate

    public Sauce creerSauce(); // factory method pour
                           // creer l’objet sauce

    public Fromage creerFromage(); //factory method
                           // pour le fromage

    ...
}
```

À retenir sur les deux patterns

- ▶ Les deux patterns Factory encapsulent la création d'objets.
- ▶ Les deux patterns Factory découplent les classes de haut niveau des classes de bas niveau.
- ▶ Factory Method est une sorte de constructeur virtuel pour créer un type d'objet.
- ▶ Factory Method se base sur l'héritage pour déléguer la création d'objets aux sous-classes.
- ▶ Abstract Factory permet d'instancier des familles d'objets.
- ▶ Abstract Factory se base sur la composition.

Feuille de TP 06

Factory