

Design patterns

Éléments de conception réutilisables

P. Laroque

octobre 2009



Outline

- 1 Introduction
- 2 Quelques Patterns Importants
 - Stratégie (strategy)
 - Observateur (observer)
 - Décorateur (decorator)
 - Commande (command)
 - Adaptateur (adapter)
 - Template de méthode (method template)
 - Itérateur et Composite (iterator, composite)
 - Etat (state)
 - Singleton
 - Fabrique (factory)
 - Méthode de fabrique (factory method)
 - Fabrique abstraite (abstract factory)
- 3 Composition de patterns

Qu'est-ce que c'est?

- “algorithme” au niveau de la conception
- schéma suffisamment général pour être appliqué dans plusieurs situations proches
- morceau récurrent d'architecture logicielle

A quoi ça sert?

- Permet à un concepteur débutant d'éviter des écueils bien connus
- Facilite la maintenance de l'application à concevoir

Sources

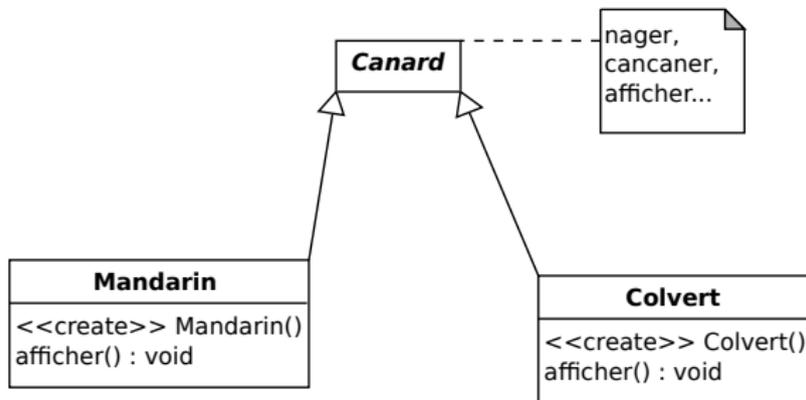
- 1970's: C. Alexander, en architecture
- 1994: E. Gamma [Gamma], catalogue de patterns: la “bible”, complet mais pas de processus
- 2003: C. Larman [Larman], processus (proche analyse) mais peu de patterns
- 2004: E. Freeman [hfdp], approche pédagogique mais incomplet

Principe du cours

- S'inspire de [hfdp]
- Sélection de patterns en situation
- Groupement de plusieurs patterns sur une étude de cas
- Code applicatif en Java

Problème initial

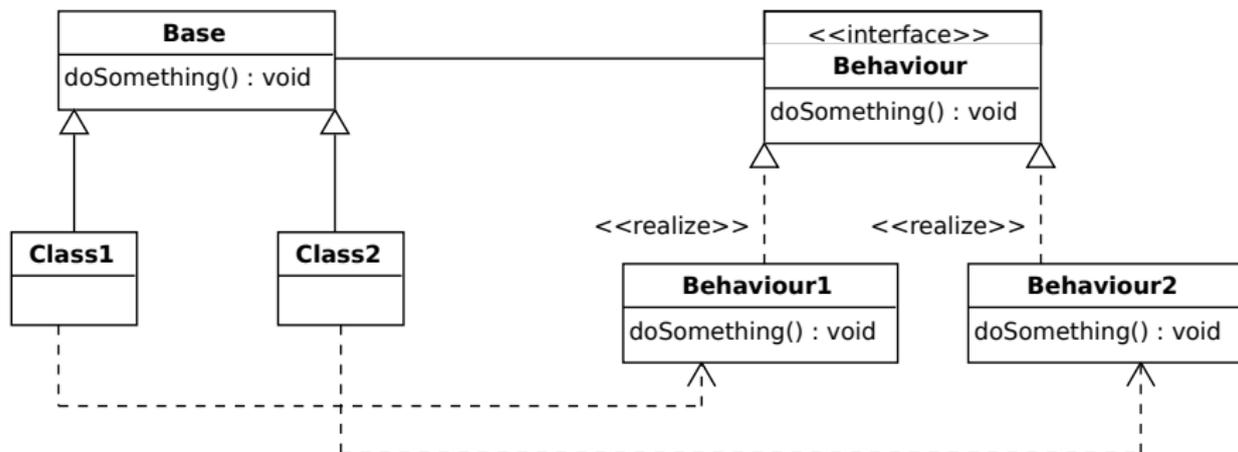
- Une application de jeu de simulation sur les canards
- Les canards nagent et crient:



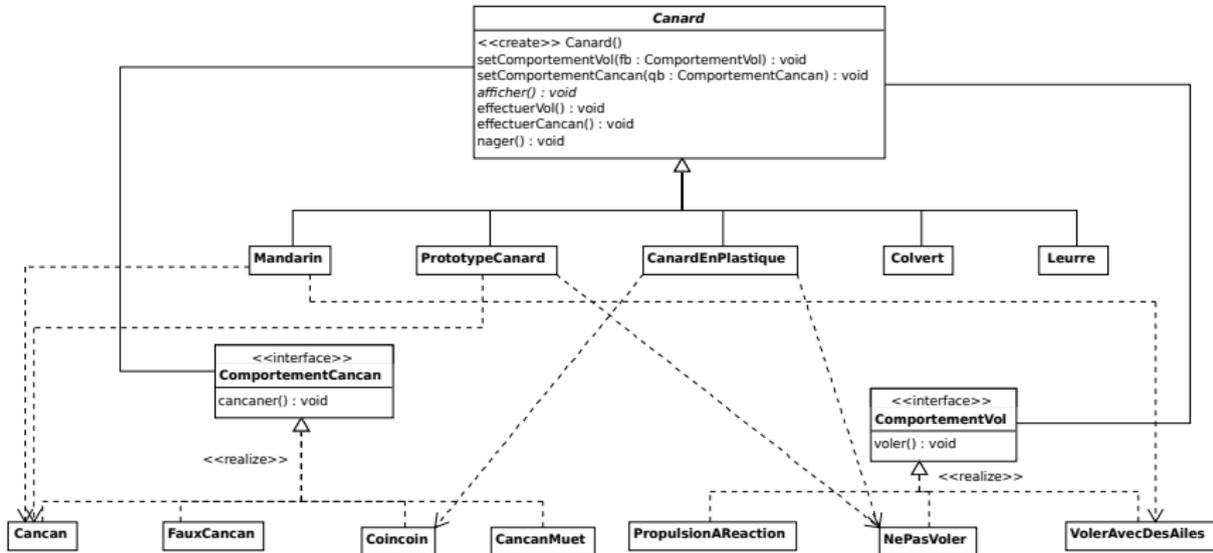
Faire voler les canards?

- Simple: ajout d'une méthode `voler()` dans `Canard` ...
- ... Problème: les canards en plastique ne volent pas!
- Solution: redéfinir `voler()` dans `CanardEnPlastique` ?
- Non: le problème se pose dans d'autres classes (le canard *lambda* ne vole pas non plus)
- Il faut penser à séparer ce qui varie de ce qui demeure constant
- On va essayer d'encapsuler les parties variables hors du code stable

Le pattern "Strategy"



Canards "stratégiques"



La classe Canard I

```
public abstract class Canard {
    ComportementVol comportementVol;
    ComportementCancan comportementCancan;

    public Canard() {
    }

    public void setComportementVol (ComportementVol fb) {
        comportementVol = fb;
    }

    public void setComportementCancan(ComportementCancan qb) {
        comportementCancan = qb;
    }

    abstract void afficher();

    public void effectuerVol() {
        comportementVol.voler();
    }
}
```

La classe Canard II

```
public void effectuerCancan() {  
    comportementCancan.cancaner();  
}  
  
public void nager() {  
    System.out.println("Tous les canards flottent, même les leurres!");  
}  
}
```

La classe Mandarin

```
public class Mandarin extends Canard {  
  
    public Mandarin() {  
        comportementVol = new VolerAvecDesAiles();  
        comportementCancan = new Cancan();  
    }  
  
    public void afficher() {  
        System.out.println("Je suis un vrai mandarin");  
    }  
}
```

Résumé

- "Program to an interface": les behaviours
- Chaque variation de comportement est une implémentation de l'interface
- Chaque classe qui a ce comportement référence une instance de (une classe dérivée de) la behaviour: *changement dynamique possible de comportement!*
- Ajout de nouveaux comportements: indolore

Le problème

- Créer une interface pour station météo (température, hygrométrie, pression)
- Affichage en "temps réel" à partir de données provenant de la station des (un des trois)
 - conditions actuelles
 - statistiques
 - prévisions simples

Les données d'entrée

DonneesMeteo
temperature : float humidite : float pression : float
<<create>> DonneesMeteo() setMesures(temperature : float,humidite : float,pression : float) : void getPression() : float getTemperature() : float getHumidite() : float

- La méthode `setMesures()` est appelée périodiquement
- Trois affichages à réaliser initialement: à l'avenir, des ajouts / retraits peuvent survenir et de nouveaux affichages être créés

Première approche

On écrit notre version de `setMesures()`:

```
public void setMesures() {
    float temp = getTemperature();
    float humid = getHumidite();
    float press = getPression();
    affichageConditions.update(temp, humid, press);
    affichageStats.update(temp, humid, press);
    affichagePrev.update(temp, humid, press);
}
```

Problèmes

- codage d'implémentations: statique (autres affichages -> modification de `setMesures`)
- on n'a pas encapsulé ce qui varie (la MàJ des affichages); pourtant, même profils...

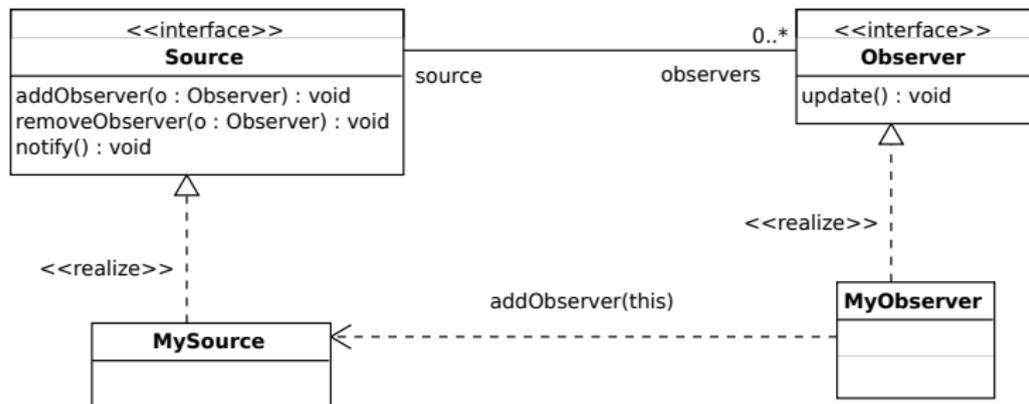
Le pattern Observateur

- 1 Vous faites votre choix parmi les revues de l'OFUP
- 2 Vous vous abonnez à votre revue préférée et recevez les exemplaires lors de leur parution
- 3 Vous arrêtez votre abonnement: vous ne recevez plus de revue

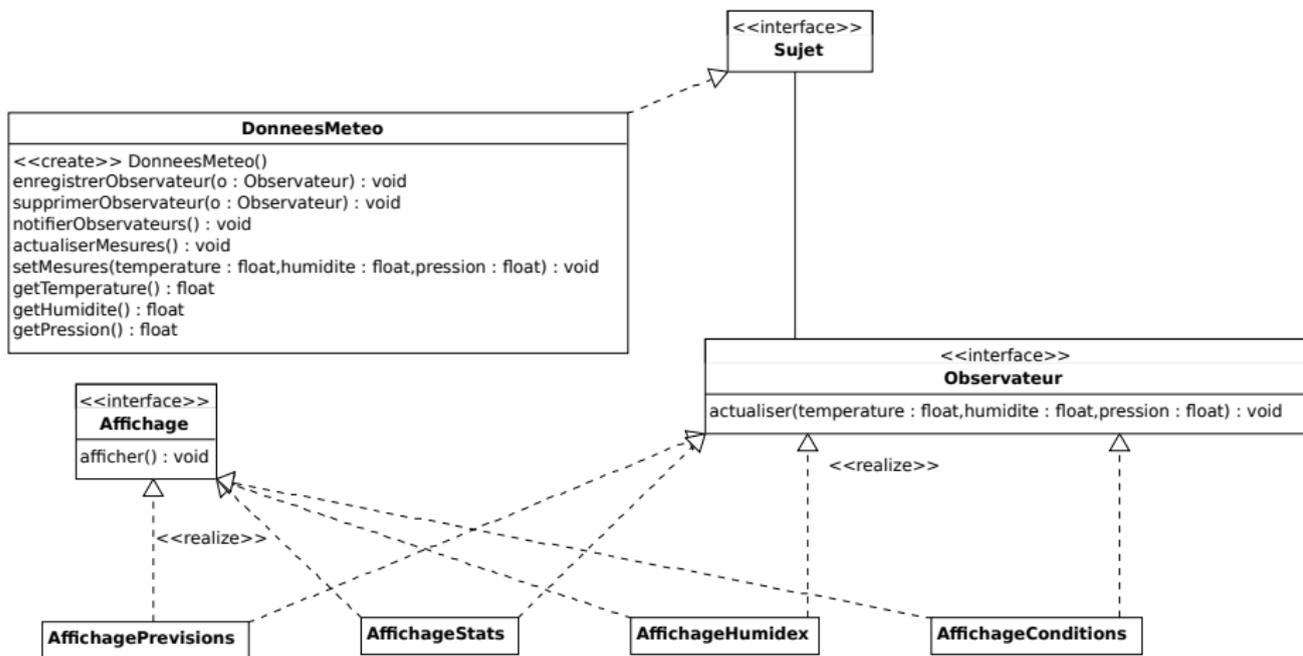
Principe:

- un objet "source" (ou "sujet") est observé par plusieurs objets "abonnés" (intéressés par ses variations)
- quand la source change d'état, tous les abonnés sont prévenus
- chaque abonné réagit alors de la façon qui lui est propre

Les acteurs



Application à la station météo



La classe DonneesMeteo I

```
public class DonneesMeteo implements Sujet {
    private ArrayList observateurs;
    private float temperature;
    private float humidite;
    private float pression;

    public DonneesMeteo() {
        observateurs = new ArrayList();
    }

    public void enregistrerObservateur(Observateur o) {
        observateurs.add(o);
    }

    public void supprimerObservateur(Observateur o) {
        int i = observateurs.indexOf(o);
        if (i >= 0) {
```

La classe DonneesMeteo II

```
        observateurs.remove(i);
    }
}

public void notifierObservateurs() {
    for (int i = 0; i < observateurs.size(); i++) {
        Observateur observer = (Observateur)observateurs.get(i);
        observer.actualiser(temperature, humidite, pression);
    }
}

public void actualiserMesures() {
    notifierObservateurs();
}

public void setMesures(float temperature, float humidite, float pression) {
    this.temperature = temperature;
    this.humidite = humidite;
    this.pression = pression;
}
```

La classe DonneesMeteo III

```
    actualiserMesures();  
}  
  
// autres méthodes de DonneesMeteo  
  
public float getTemperature() {  
    return temperature;  
}  
  
public float getHumidite() {  
    return humidite;  
}  
  
public float getPression() {  
    return pression;  
}  
}
```

Un exemple d'affichage I

```
public class AffichageConditions implements Observateur, Affichage {
    private float temperature;
    private float humidite;
    private Sujet donneesMeteo;

    public AffichageConditions(Sujet donneesMeteo) {
        this.donneesMeteo = donneesMeteo;
        donneesMeteo.enregistrerObservateur(this);
    }

    public void actualiser(float temperature, float humidite, float pression)
        this.temperature = temperature;
        this.humidite = humidite;
        afficher();
    }

    public void afficher() {
```

Un exemple d'affichage II

```
System.out.println("Conditions actuelles: " + temperature
                   + " degrés C et " + humidite + "% d'humidité");
}
}
```

Un petit test

```
public class StationMeteo {  
  
    public static void main(String[] args) {  
        DonneesMeteo donneesMeteo = new DonneesMeteo();  
  
        AffichageConditions affichageCond =  
            new AffichageConditions(donneesMeteo);  
        AffichageStats affichageStat = new AffichageStats(donneesMeteo);  
        AffichagePrevisions affichagePrev = new AffichagePrevisions(donneesMeteo);  
  
        donneesMeteo.setMesures(26, 65, 1020);  
        donneesMeteo.setMesures(28, 70, 1012);  
        donneesMeteo.setMesures(22, 90, 1012);  
    }  
}
```

Résumé

- On a découplé la station des types d'affichages
- On peut donc ajouter de nouveaux types sans modifier le code existant
- Remarques
 - le pattern existe déjà en Java (voir `java.util.Observable`)
 - il est abondamment utilisé dans le JDK, notamment dans l'AWT et dans SWING

Le problème

- Starbuzz coffee veut un système unifié de facturation pour ses points de vente
- Les cafés proposés sont de plusieurs types, et admettent de nombreuses options (lait, chantilly, chocolat, caramel,...) qui ont une influence sur le prix

→ risque d'explosion combinatoire des classes!

Première approche

- Des variables d'instance pour les ingrédients dans la classe de base
- La méthode `prix()` tient alors compte des ingrédients présents
- Les sous-classes invoquent la méthode de base puis font les ajustements nécessaires

Critique

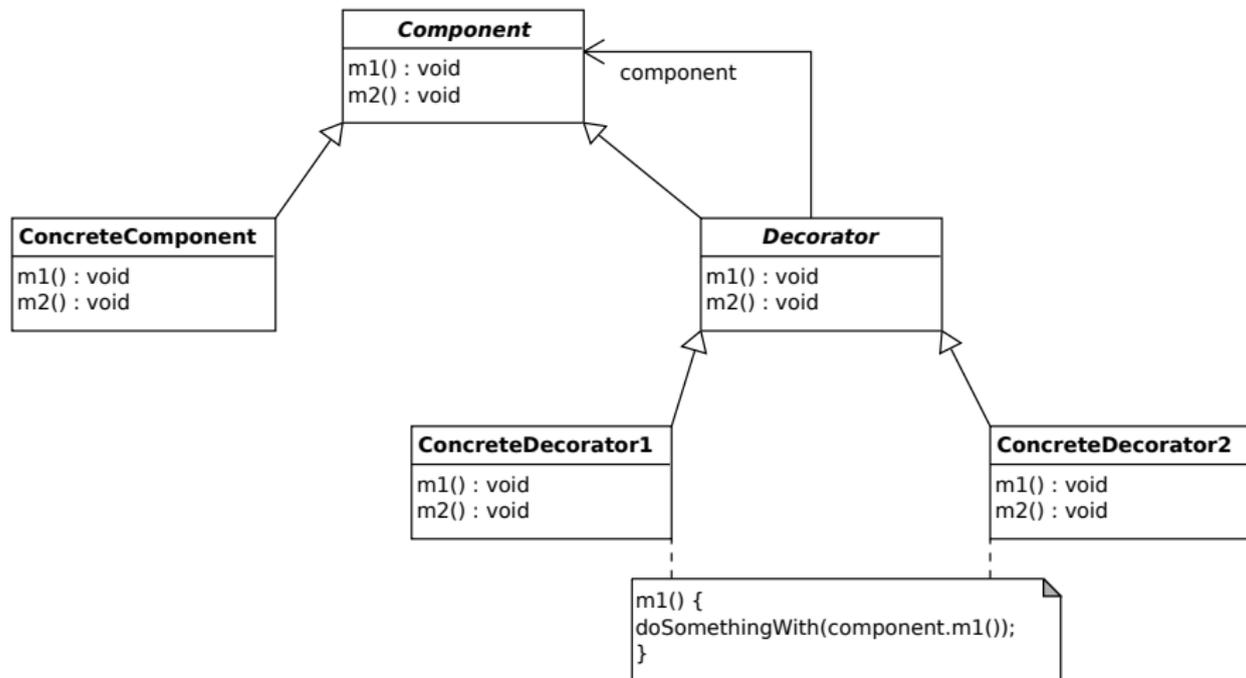
- L'ajout de nouveaux ingrédients modifie la classe de base (et les méthodes `prix()` des classes dérivées)
- Tous les ingrédients ne sont pas nécessairement adaptés à toutes les boissons (thé chantilly?)
- ...
- Un principe important: *open-closed principle* (principe "ouvert-fermé")

Les classes doivent être ouvertes à l'extension, mais fermées à la modification

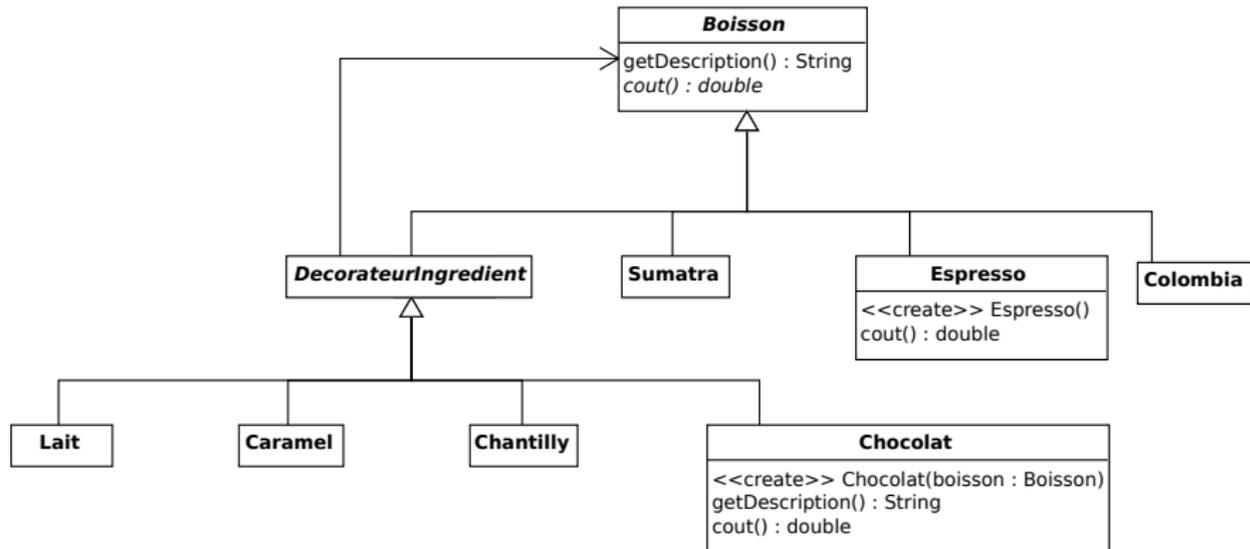
La "décoration"

- Pour ne pas modifier le code, on va le "décorer" avec du code supplémentaire pour prendre en compte les nouvelles informations / besoins
- Par exemple, pour un espresso "java" avec chocolat et chantilly, on
 - prend l'espresso "java"
 - le décore avec un supplément chocolat
 - décore le résultat avec un supplément chantilly
- La méthode `prix()` s'appuie sur la *délégation*

Le pattern Décorateur



Les boissons "Starbuzz"



Comment ça marche?

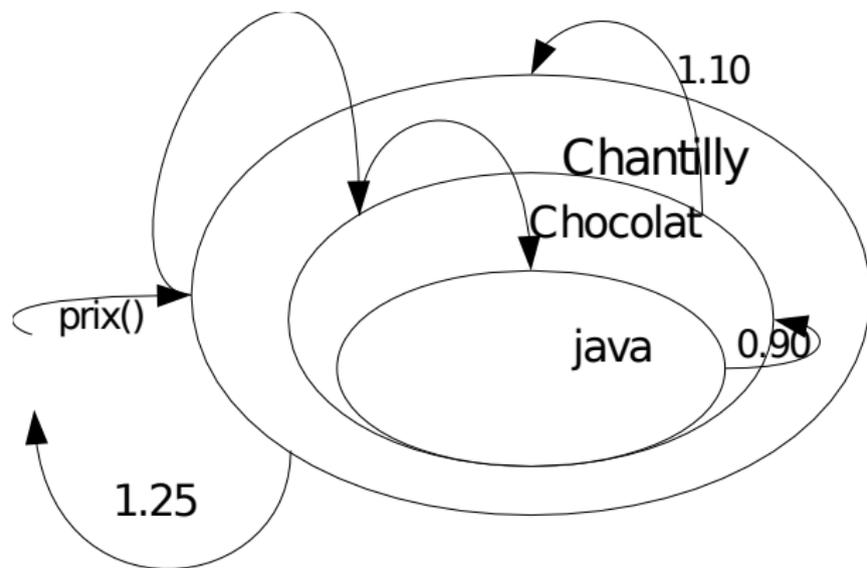
```
public class Chocolat extends DecorateurIngredient {
    Boisson boisson;

    public Chocolat(Boisson boisson) {
        this.boisson = boisson;
    }

    public String getDescription() {
        return boisson.getDescription() + ", Chocolat";
    }

    public double cout() {
        return .20 + boisson.cout();
    }
}
```

Comment ça marche? (2)



Le contexte

- Un boîtier programmable à 7 emplacements * 2 boutons (marche / arrêt) + 1 annulation globale
- Divers appareils ménagers à piloter (TV, plafonnier, porte garage, Alarme, etc.)
- Chaque appareil présente une interface d'utilisation particulière
- Bien entendu, le boîtier de commande doit ignorer le détail du fonctionnement des appareils

Introduction au pattern

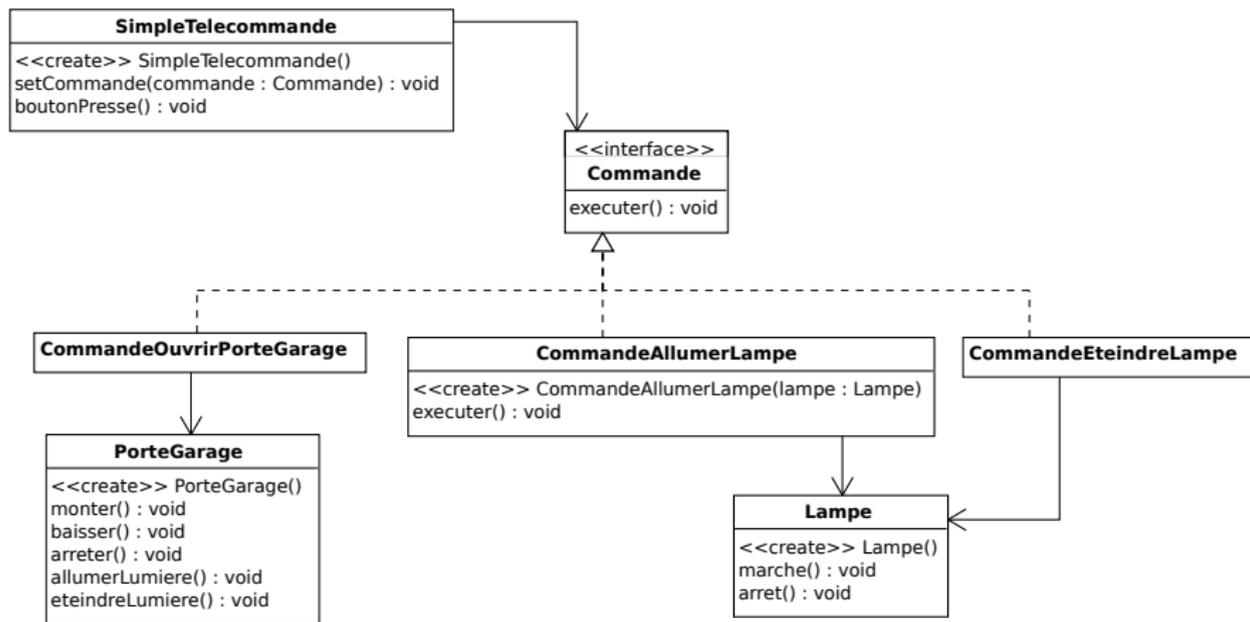
Analogie: le starbuzz coffee. Déroulement des opérations:

- 1 Le CLIENT passe une COMMANDE au SERVEUR
il va créer une commande et la donner au serveur
- 2 le SERVEUR place la COMMANDE et demande au BARMAN de la préparer
il peut ignorer le contenu de la commande!
- 3 le BARMAN prépare la COMMANDE
c'est le seul qui sait comment faire

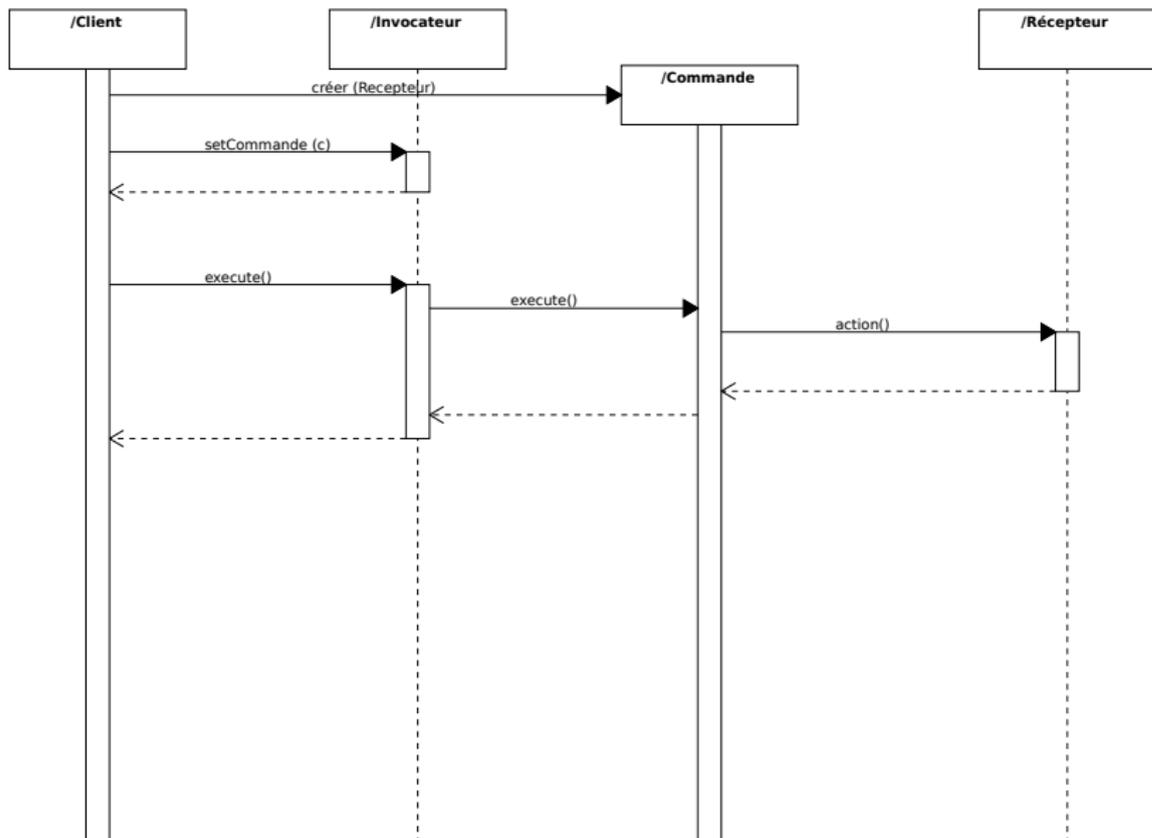
Idée générale:

La commande, située "entre" le serveur et le barman, permet de les rendre indépendants (découplage)

Cas simple



Déroulement des opérations



Exemple de commande

- On suppose que le récepteur est une lampe, avec les méthodes `marche()` et `arret()`

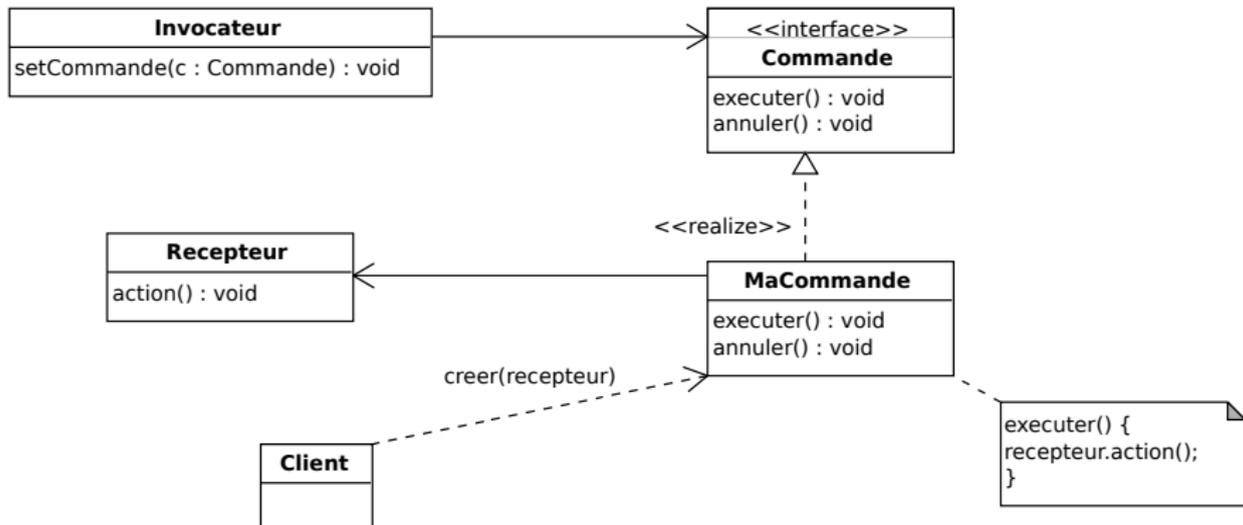
```
public class CommandeAllumerLampe implements Command {
    Lampe lampe;
    public CommandeAllumerLampe(Lampe lampe) {
        this.lampe = lampe;
    }
    public void executer() {
        lampe.marche();
    }
}
```

Exemple d'invocateur

- Une télécommande simple à un bouton

```
public class TelecommandeSimple {
    Commande element1; // un seul, ici
    public TelecommandeSimple() {}
    public void setCommande (Commande c) {
        element1 = c;
    }
    public void boutonPresse() {
        element1.executer();
    }
}
```

Le pattern Commande



Application à la télécommande

- 1 Création de l'invocateur (le boîtier)
- 2 Création des récepteurs
- 3 Création des objets Commande liés aux récepteurs
- 4 Affectation des commandes aux touches du boîtier de télécommande
- 5 Test d'appui sur les touches

Programme de test I

```
public class ChargeurTelecommande {  
  
    public static void main(String[] args) {  
  
        // 1 - Creation de l'invocateur  
        Telecommande teleCommande = new Telecommande();  
  
        // 2 - creation des recepteurs  
        Lampe lampeSejour = new Lampe("Séjour");  
        Lampe lampeCuisine = new Lampe("Cuisine");  
        Ventilateur ventilateur= new Ventilateur("Séjour");  
        PorteGarage porteGarage = new PorteGarage("");  
        Stereo stereo = new Stereo("Séjour");  
    }  
}
```

Programme de test II

```
// 3 - Creation des objets "commande"
CommandeAllumerLampe lampeSejourAllumee =
    new CommandeAllumerLampe(lampeSejour);
CommandeEteindreLampe lampeSejourEteinte =
    new CommandeEteindreLampe(lampeSejour);
CommandeAllumerLampe lampeCuisineAllumee =
    new CommandeAllumerLampe(lampeCuisine);
CommandeEteindreLampe lampeCuisineEteinte =
    new CommandeEteindreLampe(lampeCuisine);
CommandeAllumerVentilateur ventilateurAllume =
    new CommandeAllumerVentilateur(ventilateur);
CommandeEteindreVentilateur ventilateurEteint =
    new CommandeEteindreVentilateur(ventilateur);
CommandeOuvrirPorteGarage porteGarageOuverte =
    new CommandeOuvrirPorteGarage(porteGarage);
```

Programme de test III

```
CommandeFermerPorteGarage porteGarageFermee =
    new CommandeFermerPorteGarage(porteGarage);
CommandeAllumerStereoAvecCD stereoAvecCD =
    new CommandeAllumerStereoAvecCD(stereo);
CommandeEteindreStereo stereoEteinte =
    new CommandeEteindreStereo(stereo);

// 4 - affectation des commandes aux touches du boitier
teleCommande.setCommande(0, lampeSejourAllumee,
                        lampeSejourEteinte);
teleCommande.setCommande(1, lampeCuisineAllumee,
                        lampeCuisineEteinte);
teleCommande.setCommande(2, ventilateurAllume,
                        ventilateurEteint);
teleCommande.setCommande(3, stereoAvecCD,
```

Programme de test IV

```
        stereoEteinte);  
  
System.out.println(teleCommande);  
  
// 5 - test d'appui sur des touches  
teleCommande.boutonMarchePresse(0);  
teleCommande.boutonArretPresse(0);  
teleCommande.boutonMarchePresse(1);  
teleCommande.boutonArretPresse(1);  
teleCommande.boutonMarchePresse(2);  
teleCommande.boutonArretPresse(2);  
teleCommande.boutonMarchePresse(3);  
teleCommande.boutonArretPresse(3);  
    }  
}
```

Annulation I

Comment implémenter l'annulation générale?

- Définir une méthode annuler() dans l'interface Commande et l'implémenter dans toutes les classes (ex. Lampe):

```
public class CommandeAllumerLampe implements Command {
    Lampe lampe;
    public CommandeAllumerLampe(Lampe lampe) {
        this.lampe = lampe;
    }
    public void executer() {
        lampe.marche();
    }
    public void annuler() {
        lampe.arret();
    }
}
```

Annulation II

```
}  
}
```

- Mémoriser la dernière commande exécutée dans le boîtier de télécommande:

```
public class TelecommandeAvecAnnul {  
    Commande[] commandesMarche;  
    Commande[] commandesArret;  
    Commande commandeAnnulation;  
  
    public TelecommandeAvecAnnul() {  
        commandesMarche = new Commande[7];  
        commandesArret = new Commande[7];  
  
        Commande pasDeCommande = new PasDeCommande();  
        for(int i=0;i<7;i++) {  
            commandesMarche[i] = pasDeCommande;
```

Annulation III

```
    commandesArret[i] = pasDeCommande;
}
commandeAnnulation = pasDeCommande;
}

public void setCommande(int empt, Commande comMarche, Commande comArret)
    commandesMarche[empt] = comMarche;
    commandesArret[empt] = comArret;
}

public void boutonMarchePresse(int empt) {
    commandesMarche[empt].executer();
    commandeAnnulation = commandesMarche[empt];
}

public void boutonArretPresse(int empt) {
    commandesArret[empt].executer();
    commandeAnnulation = commandesArret[empt];
}
```

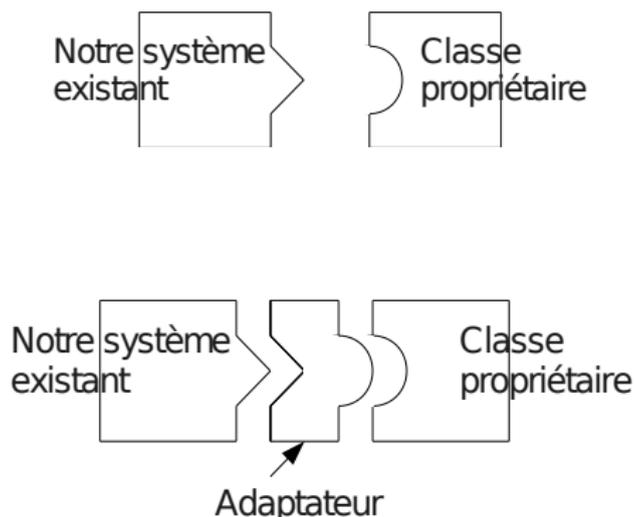
Annulation IV

```
public void boutonAnnulPresse() {
    commandeAnnulation.annuler();
}

public String toString() {
    StringBuffer stringBuffer = new StringBuffer();
    stringBuffer.append("\n----- Télécommande -----\n");
    for (int i = 0; i < commandesMarche.length; i++) {
        stringBuffer.append("[empt " + i + " ] "
            + commandesMarche[i].getClass().getName()
            + "      "
            + commandesArret[i].getClass().getName() + "\n");
    }
    stringBuffer.append("[annulation] "
        + commandeAnnulation.getClass().getName() + "\n");
    return stringBuffer.toString();
}
}
```

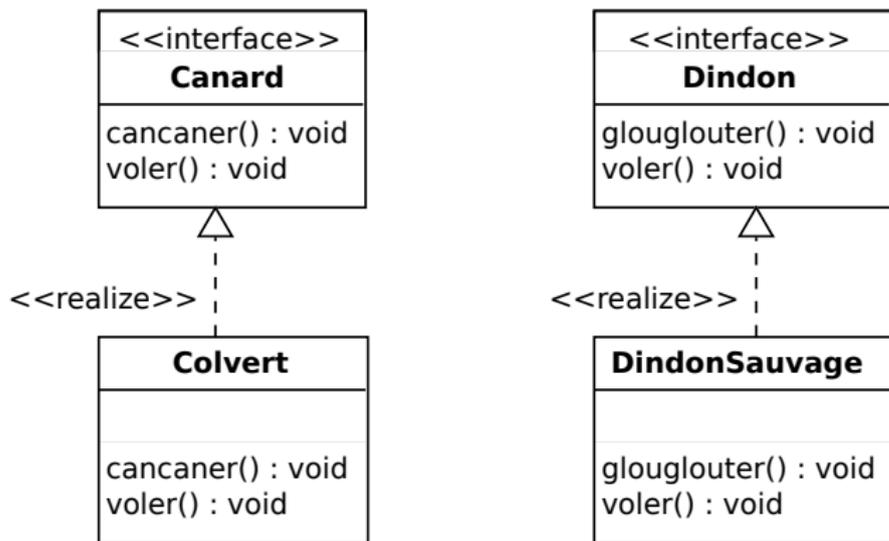
Le problème

- Adaptateur matériel: brancher un appareil électrique français aux US
- Adaptateur OO:



Canards et dindons

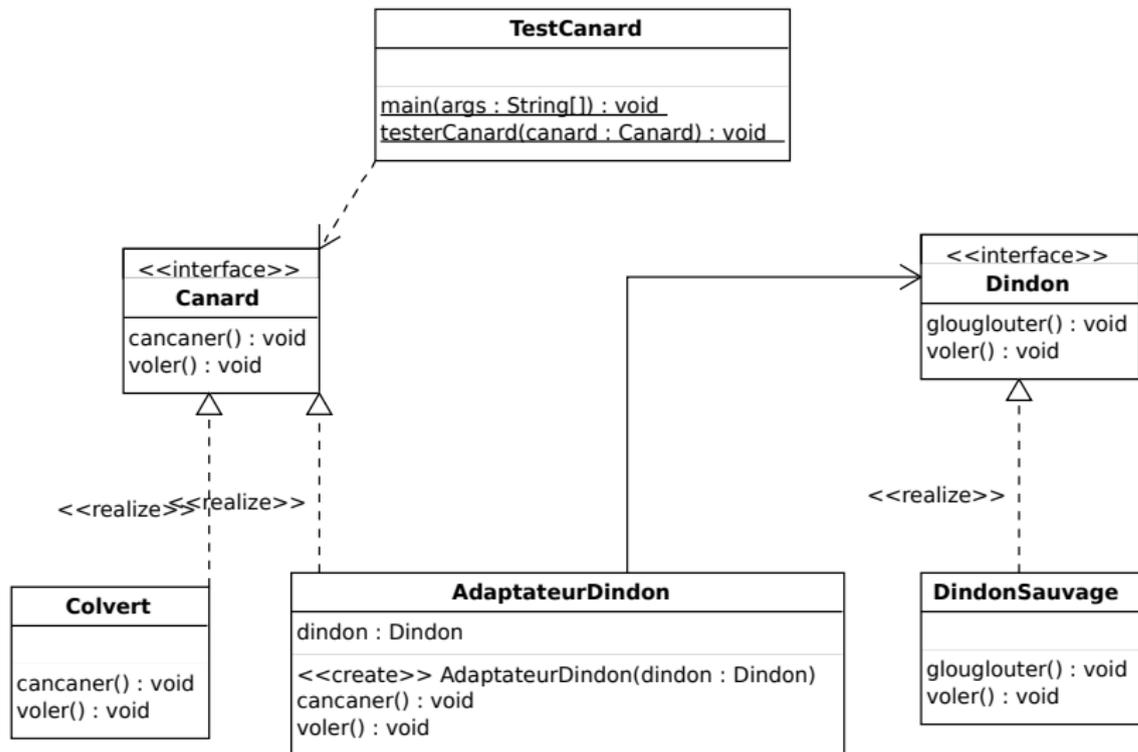
On veut ajouter des dindons à l'application "canards":



Solution:

Créer un adaptateur de dindon vers canard!

L'adaptateur dindon vers canard



Exemple de code

```
public class AdaptateurDindon implements Canard {
    Dindon dindon;

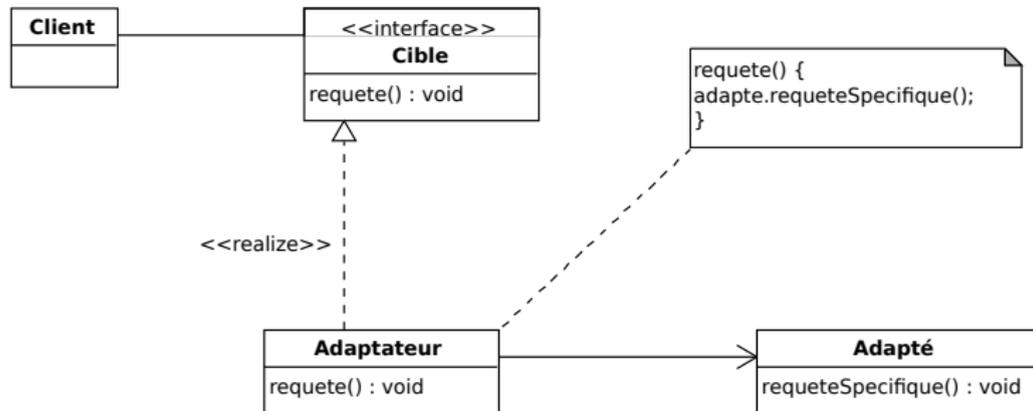
    public AdaptateurDindon(Dindon dindon) {
        this.dindon = dindon;
    }

    public void cancaner() {
        dindon.glouglouter();
    }

    public void voler() {
        for(int i=0; i < 5; i++) {
            dindon.voler();
        }
    }
}
```

Le pattern Adaptateur

But: convertir l'interface d'une classe en une autre compatible avec l'application cliente.



Le contexte

- On repère dans deux algorithmes de parties communes et des parties spécifiques
- On va encapsuler les parties spécifiques pour retenir un algorithme global générique (algorithme "à trou")
- Illustration: le Starbuzz Coffee, préparations du café et du thé

Les algorithmes

Recette du café

- 1 Faire bouillir de l'eau
- 2 *Filtrer le café à l'eau bouillante*
- 3 Verser le café dans une tasse
- 4 *Ajouter du lait et du sucre*

Recette du thé

- 1 Faire bouillir de l'eau
- 2 *Faire infuser le thé dans l'eau bouillante*
- 3 Verser le thé dans une tasse
- 4 *Ajouter du citron*

Première version du code I

```
public class Cafe {  
  
    void suivreRecette() {  
        faireBouillirEau();  
        filtrerCafe();  
        verserDansTasse();  
        ajouterLaitEtSucre();  
    }  
  
    public void faireBouillirEau() {  
        System.out.println("L'eau bout");  
    }  
  
    public void filtrerCafe() {
```

Première version du code II

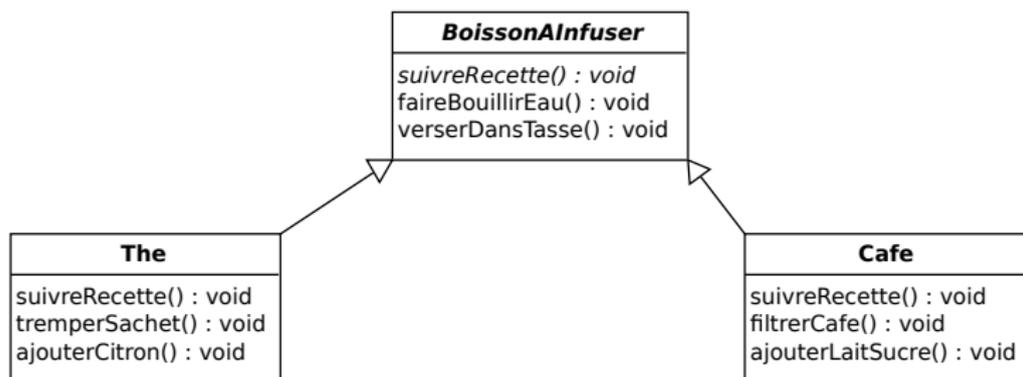
```
    System.out.println("Le café passe");
}

public void verserDansTasse() {
    System.out.println("Je verse dans la tasse");
}

public void ajouterLaitEtSucre() {
    System.out.println("Ajout de lait et de sucre");
}
}
```

Analyse superficielle

Beaucoup de code commun avec The: héritage!



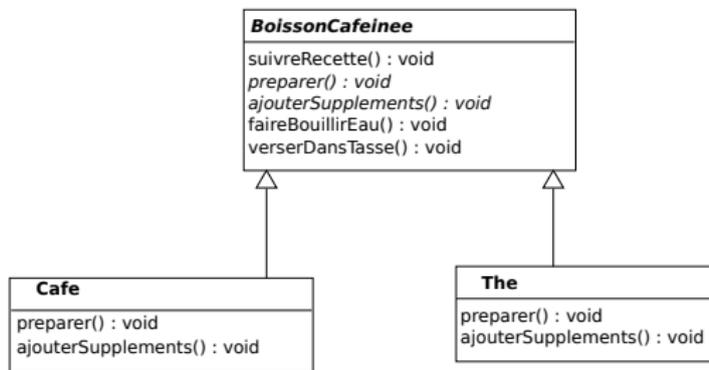
Etape suivante

On peut aller plus loin

les deux recettes utilisent le même *algorithme* → *on va abstraire des portions de suivre* Recette

- 1 faire bouillir de l'eau
- 2 préparer la boisson
- 3 verser la boisson
- 4 ajouter des suppléments éventuels

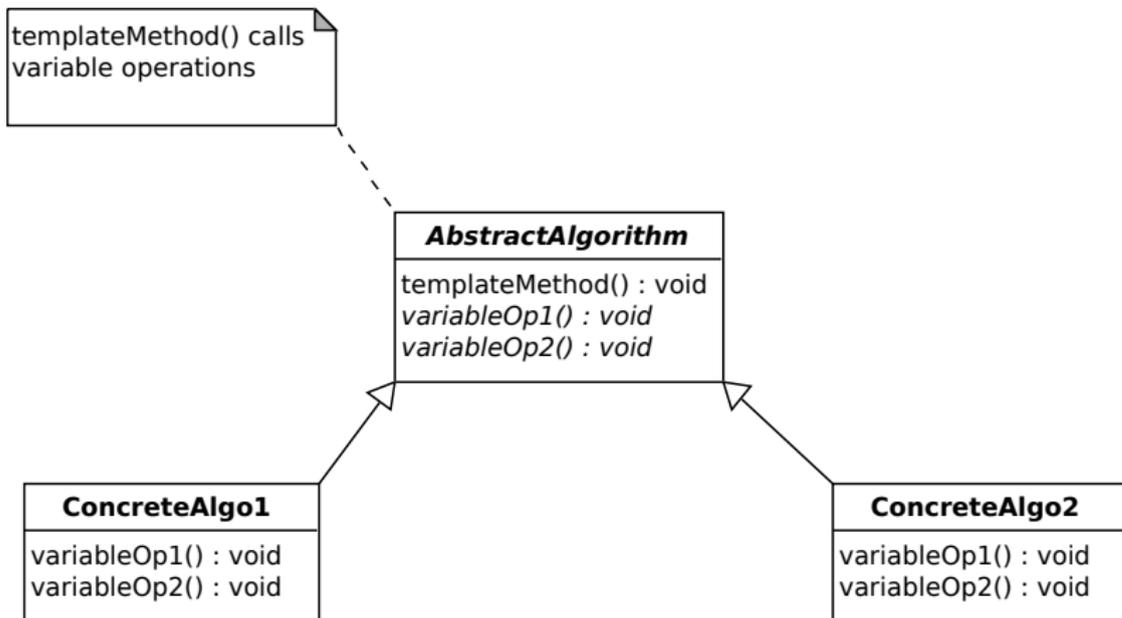
Etape suivante (code)



Le pattern template de méthode

- La classe abstraite contient l'algorithme (le template de méthode) et des versions abstraites des éléments variables
- Les éléments variables sont appelés dans l'algorithme
- Chaque classe dérivée implémente sa version des éléments variables

Le pattern (modèle UML)



Principe de Hollywood

- Le composant de haut niveau `AbstractAlgorithm` invoque les méthodes de ses sous-classes
- Les sous-classes n'effectuent pas d'appel vers `AbstractAlgorithm`
- C'est le *principe de Hollywood*: "don't call us, we'll call you"
- Exemple: la fonction de comparaison dans `qsort()` (bibliothèque C)

Le contexte

- Fusion de la cafeteria et de la crêperie
- Deux menus distincts à gérer de manière homogène:
 - la cafeteria a une `ArrayList` de plats
 - la cêperie a un tableau de plats

Plat
nom : String description : String vegetarien : boolean prix : double
<<create>> Plat(nom : String,description : String,vegetarien : boolean,prix : double) getNom() : String getDescription() : String getPrix() : double estVegetarien() : boolean toString() : String

Problème: afficher les plats

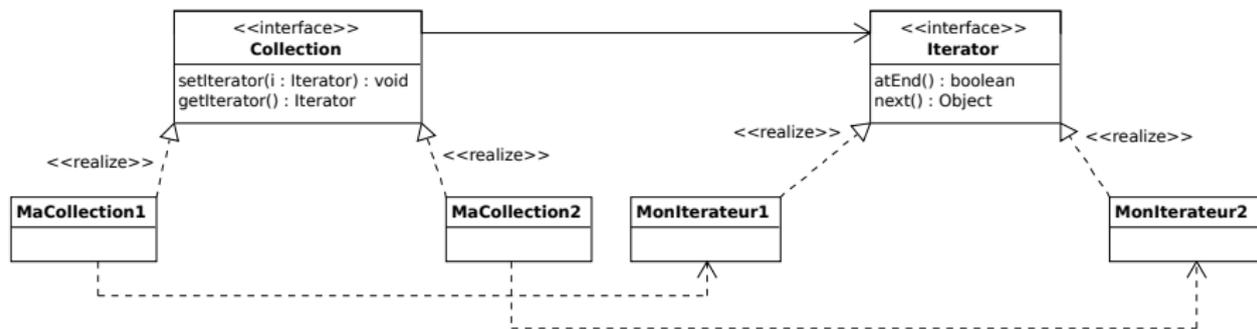
- Hétérogène et redondant
- Expose une connaissance de l'architecture interne

```
for (int i = 0; i < platsCafet.size(); i++)  
    Plat plat = (Plat)platsCafet.get(i);  
    ...  
}
```

```
for (int i = 0; i < platsCreperie.length; i++)  
    Plat plat = platsCreperie[i];  
    ...  
}
```

Le pattern iterator

- chaque collection peut s'attacher un itérateur
- l'itérateur peut tester s'il reste des éléments non vus et passer à l'élément suivant



Test de l'itérateur I

```
public class Serveuse {
    Menu menuCreperie;
    Menu menuCafeteria;

    public Serveuse(Menu menuCreperie, Menu menuCafeteria) {
        this.menuCreperie = menuCreperie;
        this.menuCafeteria = menuCafeteria;
    }

    public void afficherMenu() {
        Iterator itérateurCrepe = menuCreperie.creerItérateur();
        Iterator itérateurCafet = menuCafeteria.creerItérateur();

        System.out.println("MENU\n----\nBRUNCH'");
    }
}
```

Test de l'itérateur II

```
    afficherMenu(iterateurCrepe);
    System.out.println("\nDEJEUNER");
    afficherMenu(iterateurCafet);
}

private void afficherMenu(Iterator itereateur) {
    while (itereateur.hasNext()) {
        Plat plat = (Plat)itereateur.next();
        System.out.print(plat.getNom() + ", ");
        System.out.print(plat.getPrix() + " -- ");
        System.out.println(plat.getDescription());
    }
}

public void afficherMenuVegetarien() {
```

Test de l'itérateur III

```
System.out.println("\nMENU VEGETARIEN\n----\nBRUNCH");
afficherMenuVegetarien(menuCreperie.creerIterateur());
System.out.println("\nDEJEUNER");
afficherMenuVegetarien(menuCafeteria.creerIterateur());
}

public boolean estPlatVegetarien(String nom) {
    Iterator itereurCrepe = menuCreperie.creerIterateur();
    if (estVegetarien(nom, itereurCrepe)) {
        return true;
    }
    Iterator itereurCafet = menuCafeteria.creerIterateur();
    if (estVegetarien(nom, itereurCafet)) {
        return true;
    }
}
```

Test de l'itérateur IV

```
    return false;
}

private void afficherMenuVegetarien(Iterator itérateur) {
    while (itérateur.hasNext()) {
        Plat plat = (Plat)itérateur.next();
        if (plat.estVegetarien()) {
            System.out.print(plat.getNom());
            System.out.println("\t\t" + plat.getPrix());
            System.out.println("\t" + plat.getDescription());
        }
    }
}
```

Test de l'itérateur V

```
private boolean estVegetarien(String nom, Iterator itérateur) {
    while (itérateur.hasNext()) {
        Plat plat = (Plat)itérateur.next();
        if (plat.getNom().equals(nom)) {
            if (plat.estVegetarien()) {
                return true;
            }
        }
    }
    return false;
}
}
```

Critique

- La serveuse traite toujours deux menus explicites et deux itérateurs
 - On va créer une interface commune, Menu (la Collection du pattern)
 - On pourra alors itérer une seule fois sur tous les menus
- En outre, tout ajout d'un nouveau menu impose de retoucher le code de la serveuse
 - regroupement possible des menus de la serveuse dans une collection: on peut itérer sur cette collection!

Le code de la "nouvelle" serveuse I

```
public class Serveuse {
    ArrayList menus;

    public Serveuse(ArrayList menus) {
        this.menus = menus;
    }

    public void afficherMenu() {
        Iterator menuIterator = menus.iterator();
        while(menuIterator.hasNext()) {
            Menu menu = (Menu)menuIterator.next();
            afficherMenu(menu.createIterator());
        }
    }
}
```

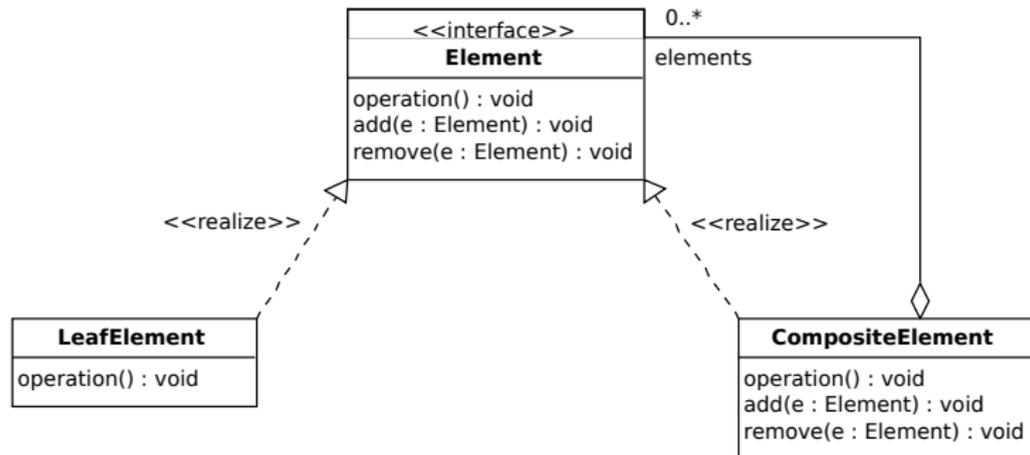
Le code de la "nouvelle" serveuse II

```
}  
  
void afficherMenu(Iterator itérateur) {  
    while (itérateur.hasNext()) {  
        Plat plat = (Plat)itérateur.next();  
        System.out.print(plat.getNom() + ", ");  
        System.out.print(plat.getPrix() + " -- ");  
        System.out.println(plat.getDescription());  
    }  
}  
}
```

Gestion des sous-menus

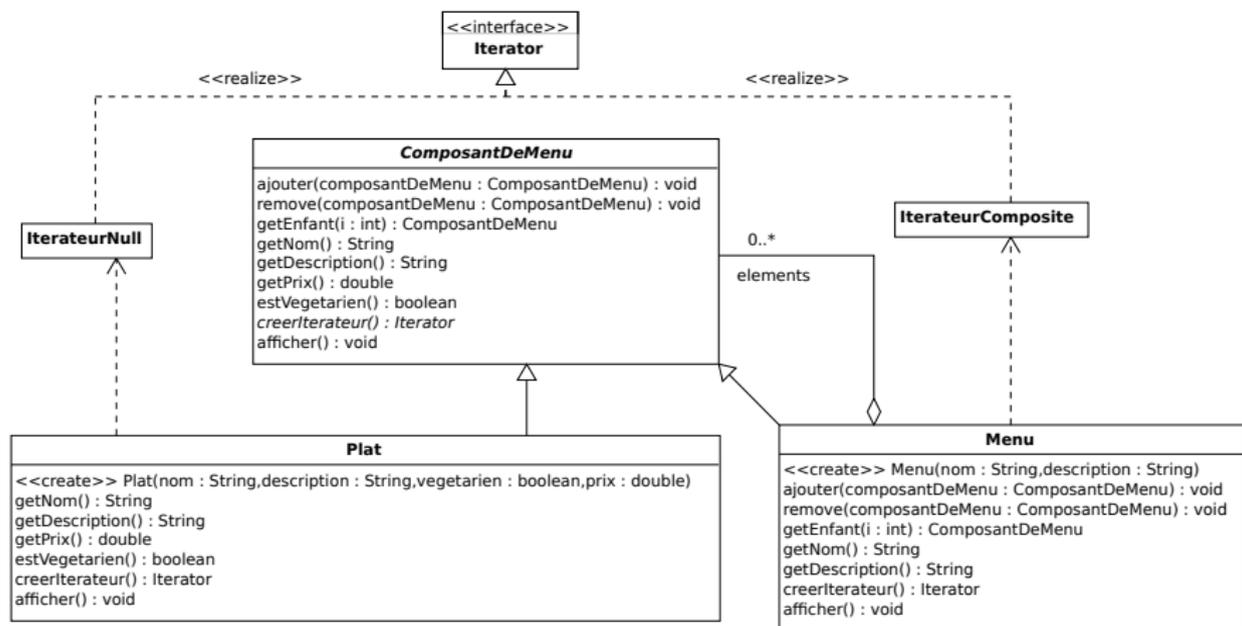
- On veut maintenant ajouter un sous-menu pour les desserts
- Problème: un menu est une collection de plats, pas une collection de menus!
- Solution: décrire et utiliser une structure de menus hiérarchique (arborescente), chaque nœud pouvant être un plat (feuille) ou un menu (nœud interne)
- Il faut donc que Plat et Menu aient un ancêtre commun dans l'arbre d'héritage: pattern "Composite"

Le pattern Composite



Application au restaurant

- On va regrouper Plat et Menu sous ElementDeMenu et appliquer le pattern Composite (rq: ItérateurNull.hasNext() renvoie false):



Code final de la serveuse

```
public class Serveuse {
    ComposantDeMenu tousMenus;

    public Serveuse(ComposantDeMenu tousMenus) {
        this.tousMenus = tousMenus;
    }

    public void afficherMenu() {
        tousMenus.afficher();
    }
}
```

Test du couple composite / itérateur I

```
public class TestMenu {
    public static void main(String args[]) {
        ComposantDeMenu menuCreperie =
            new Menu("MENU CREPERIE", "Brunch");
        ComposantDeMenu menuCafeteria =
            new Menu("MENU CAFETERIA", "Dejeuner");
        ComposantDeMenu menuBrasserie =
            new Menu("MENU BRASSERIE", "Diner");
        ComposantDeMenu menuDesserts =
            new Menu("MENU DESSERT", "Rien que des desserts !");

        ComposantDeMenu tousMenus =
            new Menu("TOUS LES MENUS", "Toutes nos offres");

        tousMenus.ajouter(menuCreperie);
        tousMenus.ajouter(menuCafeteria);
        tousMenus.ajouter(menuBrasserie);
    }
}
```

Test du couple composite / itérateur II

```
menuCreperie.ajouter
  (new Plat(
    "Crepe a l'oeuf",
    "Crepe avec oeuf au plat ou brouille",
    true,
    2.99));
menuCreperie.ajouter
  (new Plat(
    "Crepe complete",
    "Crepe avec oeuf au plat et jambon",
    false,
    2.99));
menuCreperie.ajouter
  (new Plat(
    "Crepe forestiere",
    "Myrtilles fraiches et sirop de myrtille",
    true,
    3.49));
```

Test du couple composite / itérateur III

```
menuCreperie.ajouter  
  (new Plat(  
    "Crepe du chef",  
    "Creme fraiche et fruits rouges au choix",  
    true,  
    3.59));
```

```
menuCafeteria.ajouter  
  (new Plat(  
    "Salade printaniere",  
    "Salade verte, tomates, concombre, olives, pommes de terre",  
    true,  
    2.99));
```

```
menuCafeteria.ajouter  
  (new Plat(  
    "Salade Parisienne",  
    "Salade verte, tomates, poulet, emmental",  
    false,  
    2.99));
```

Test du couple composite / itérateur IV

```
menuCafeteria.ajouter
  (new Plat(
    "Soupe du jour",
    "Soupe du jour et croutons grilles",
    true,
    3.29));
menuCafeteria.ajouter
  (new Plat(
    "Quiche aux fruits de mer",
    "Pate brisee, crevettes, moules, champignons",
    false,
    3.05));
menuCafeteria.ajouter
  (new Plat(
    "Quiche aux epinards",
    "Pate feuilletée, pommes de terre, epinards, creme fraiche",
    true,
    3.99));
menuCafeteria.ajouter
```

Test du couple composite / itérateur V

```
(new Plat(  
    "Pasta al pesto",  
    "Spaghetti, ail, basilic, parmesan",  
    true,  
    3.89));
```

```
menuCafeteria.ajouter(menuDesserts);
```

```
menuDesserts.ajouter
```

```
(new Plat(  
    "Tarte du chef",  
    "Tarte aux pommes et boule de glace a la vanille",  
    true,  
    1.59));
```

```
menuDesserts.ajouter
```

```
(new Plat(  
    "Charlotte maison",  
    "Charlotte aux poires et sauce au chocolat",  
    true,
```

Test du couple composite / itérateur VI

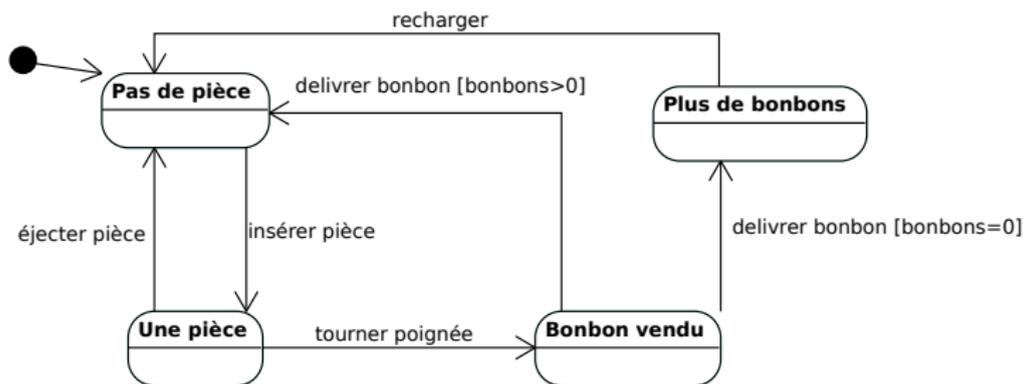
```
        1.99));  
menuDesserts.ajouter  
    (new Plat(  
        "Duos de sorbets",  
        "Une boule fraise et une boule citron vert",  
        true,  
        1.89));  
  
menuBrasserie.ajouter  
    (new Plat(  
        "Omelette sarladaise",  
        "Omelette aux champignons et pommes sautees",  
        true,  
        3.99));  
menuBrasserie.ajouter  
    (new Plat(  
        "Soupe de poissons",  
        "Soupe de poissons, rouille et croutons",  
        false,
```

Test du couple composite / itérateur VII

```
        3.69));  
menuBrasserie.ajouter  
    (new Plat(  
        "Tagliatelles Primavera",  
        "Pates fraiches, brocoli, petits pois, creme fraiche",  
        true,  
        4.29));  
  
Serveuse serveuse = new Serveuse(tousMenus);  
  
serveuse.afficherMenu();  
}  
}
```

Le contexte

Un distributeur de bonbons:



Première analyse

- Représenter les états par une variable interne discrète

```
final static public int PAS_DE_PIECE = 0;  
final static public int PLUS_DE_BONBONS = 1;  
final static public int UNE_PIECE = 2;  
final static public int BONBON_VENDU = 3;
```

Première analyse (suite)

- Tester l'état dans toutes les actions:

```
public void insererPiece() {
    if (etat == UNE_PIECE) {
        System.out.println("Vous ne pouvez plus insérer de pièce s");
    } else if (etat == PAS_DE_PIECE) {
        etat = A_PIECE;
        System.out.println("Vous avez inséré une pièce");
    } else if (etat == PLUS_DE_BONBONS) {
        System.out.println("Vous ne pouvez pas insérer de pièce, nous sommes à la fin");
    } else if (etat == BONBON_VENDU) {
        System.out.println("Veuillez patienter, le bonbon va tomber");
    }
}
```

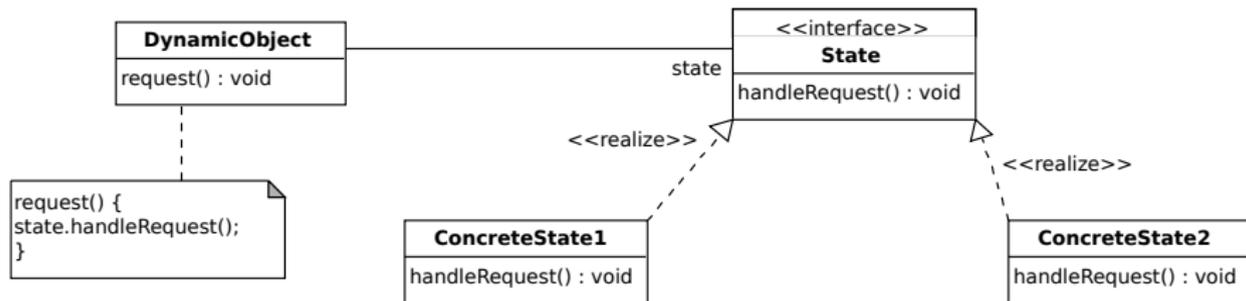
Critique

- Explicitation nécessaire du traitement de *chaque* état dans *chaque* méthode
- Lourdeur du code (structures conditionnelles trop nombreuses)
- Extension difficile du statechart puisque très intégré au code
- Par exemple, "un bonbon gratuit avec une chance sur 10":
 - ajout d'un état GAGNANT
 - modification du code de toutes les méthodes du distributeur pour gérer ce nouvel état

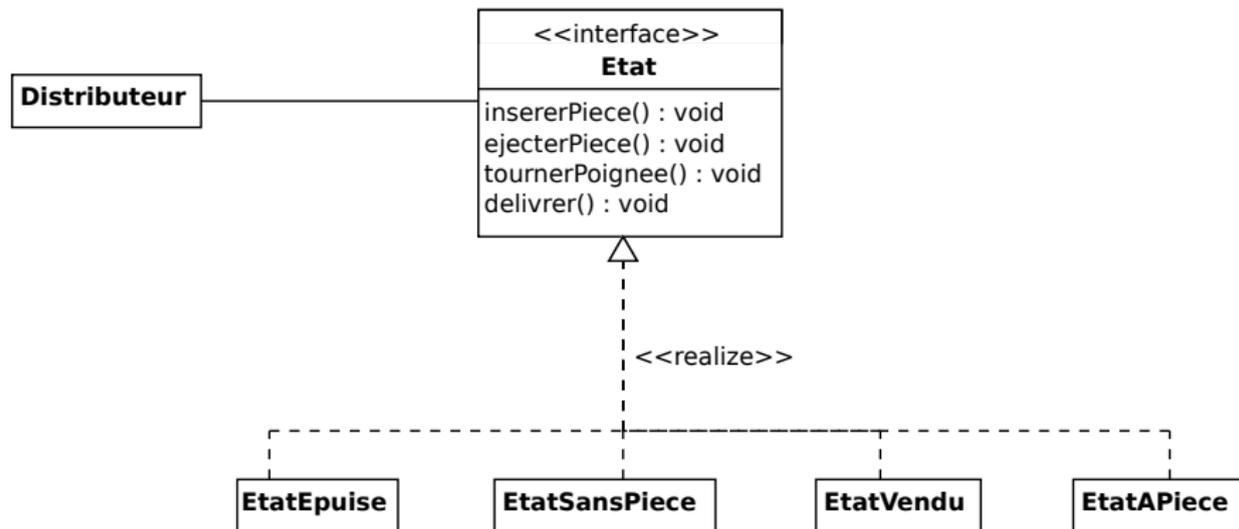
Solution possible

- 1 "Réifier" la notion d'état dans une interface (qui contient une méthode pour chaque action du distributeur)
- 2 Implémenter cette interface dans une classe pour chaque état du distributeur
- 3 Remplacer les conditionnelles par une délégation vers la "bonne" classe état

Le pattern State



Le distributeur revu



Le distributeur possède les méthodes `setEtat()` et `getEtat()` pour la gestion des transitions

Implémentation d'un état I

```
public class EtatSansPiece implements Etat {
    Distributeur distributeur;

    public EtatSansPiece(Distributeur distributeur) {
        this.distributeur = distributeur;
    }

    public void insererPiece() {
        System.out.println("Vous avez inséré une pièce");
        distributeur.setEtat(distributeur.getEtatAPiece());
    }

    public void ejecterPiece() {
        System.out.println("Vous n'avez pas inséré de pièce");
```

Implémentation d'un état II

```
}  
  
public void tournerPoignee() {  
    System.out.println("Vous avez tourné, mais il n'y a pas de  
}  
  
public void delivrer() {  
    System.out.println("Il faut payer d'abord");  
}  
  
public String toString() {  
    return "attend une pièce";  
}  
}
```

Le distributeur I

```
public class Distributeur {  
  
    Etat etatEpuise;  
    Etat etatSansPiece;  
    Etat etatAPiece;  
    Etat etatVendu;  
  
    Etat etat = etatEpuise;  
    int nombre = 0;  
  
    public Distributeur(int nombreBonbons) {  
        etatEpuise = new EtatEpuise(this);  
        etatSansPiece = new EtatSansPiece(this);  
        etatAPiece = new EtatAPiece(this);  
    }  
}
```

Le distributeur II

```
etatVendu = new EtatVendu(this);

this.nombre = nombreBonbons;
if (nombreBonbons > 0) {
    etat = etatSansPiece;
}
}

public void insererPiece() {
    etat.insererPiece();
}

public void ejecterPiece() {
    etat.ejecterPiece();
}
```

Le distributeur III

```
public void tournerPoignee() {  
    etat.tournerPoignee();  
    etat.delivrer();  
}
```

```
void setEtat(Etat etat) {  
    this.etat = etat;  
}
```

```
void liberer() {  
    System.out.println("Un bonbon va sortir...");  
    if (nombre != 0) {  
        nombre = nombre - 1;  
    }  
}
```

Le distributeur IV

```
}
```

```
int getNombre() {  
    return nombre;  
}
```

```
void refill(int nombre) {  
    this.nombre = nombre;  
    etat = etatSansPiece;  
}
```

```
public Etat getEtat() {  
    return etat;  
}
```

Le distributeur V

```
public Etat getEtatEpuise() {  
    return etatEpuise;  
}
```

```
public Etat getEtatSansPiece() {  
    return etatSansPiece;  
}
```

```
public Etat getEtatAPiece() {  
    return etatAPiece;  
}
```

```
public Etat getEtatVendu() {  
    return etatVendu;  
}
```

Le distributeur VI

```
public String toString() {
    StringBuffer result = new StringBuffer();
    result.append("\nDistribon, SARL.");
    result.append("\nDistributeur compatible Java, modèle 2004");
    result.append("\nStock : " + nombre + " bonbon");
    if (nombre != 1) {
        result.append("s");
    }
    result.append("\n");
    result.append("L'appareil " + etat + "\n");
    return result.toString();
}
}
```

Ajouter un état gagnant

- 1 Définir une nouvelle classe implémentant Etat: EtatGagnant
- 2 Ajouter ce nouvel état possible au distributeur
- 3 Gérer le hasard et la transition possible de EtatAPiece vers EtatGagnant

L'état gagnant I

```
public class EtatGagnant implements Etat {
    Distributeur distributeur;

    public EtatGagnant(Distributeur distributeur) {
        this.distributeur = distributeur;
    }

    public void insererPiece() {
        System.out.println
            ("Patientez s'il vous plait, un bonbon est en train d'être délivré");
    }

    public void ejecterPiece() {
        System.out.println
            ("Patientez s'il vous plait, un bonbon est en train d'être délivré");
    }
}
```

L'état gagnant II

```
public void tournerPoignee() {
    System.out.println
        ("Tourner une nouvelle fois la poignée ne vous donnera pas un autre b
}
```

```
public void delivrer() {
    System.out.println
        ("VOUS AVEZ GAGNE ! Deux bonbons pour le prix d'un !");
    distributeur.liberer();
    if (distributeur.getNombre() == 0) {
        distributeur.setEtat(distributeur.getEtatEpuise());
    } else {
        distributeur.liberer();
        if (distributeur.getNombre() > 0) {
            distributeur.setEtat(distributeur.getEtatSansPiece());
        } else {
            System.out.println("Aïe, plus de bonbons !");
            distributeur.setEtat(distributeur.getEtatEpuise());
        }
    }
}
```

L'état gagnant III

```
    }  
}  
  
public String toString() {  
    return "délivre deux bonbons pour le prix d'un, car vous avez gagné !";  
}  
}
```

Le distributeur modifié I

```
public class Distributeur {  
  
    Etat etatEpuise;  
    Etat etatSansPiece;  
    Etat etatAPiece;  
    Etat etatVendu;  
    Etat etatGagnant;  
  
    Etat etat = etatEpuise;  
    int nombre = 0;  
  
    public Distributeur(int nombreBonbons) {  
        etatEpuise = new EtatEpuise(this);  
        etatSansPiece = new EtatSansPiece(this);  
        etatAPiece = new EtatAPiece(this);  
        etatVendu = new EtatVendu(this);  
        etatGagnant = new EtatGagnant(this);  
    }  
}
```

Le distributeur modifié II

```
    this.nombre = nombreBonbons;
    if (nombreBonbons > 0) {
        etat = etatSansPiece;
    }
}

public void insererPiece() {
    etat.insererPiece();
}

public void ejecterPiece() {
    etat.ejecterPiece();
}

public void tournerPoignee() {
    etat.tournerPoignee();
    etat.delivrer();
}
```

Le distributeur modifié III

```
void setEtat(Etat etat) {
    this.etat = etat;
}

void liberer() {
    System.out.println("Un bonbon va sortir...");
    if (nombre != 0) {
        nombre = nombre - 1;
    }
}

int getNombre() {
    return nombre;
}

void remplir(int nombre) {
    this.nombre = nombre;
    etat = etatSansPiece;
}
```

Le distributeur modifié IV

```
}  
  
public Etat getEtat() {  
    return etat;  
}  
  
public Etat getEtatEpuisse() {  
    return etatEpuisse;  
}  
  
public Etat getEtatSansPiece() {  
    return etatSansPiece;  
}  
  
public Etat getEtatAPiece() {  
    return etatAPiece;  
}  
  
public Etat getEtatVendu() {
```

Le distributeur modifié V

```
    return etatVendu;
}

public Etat getEtatGagnant() {
    return etatGagnant;
}

public String toString() {
    StringBuffer result = new StringBuffer();
    result.append("\nDistribon, SARL.");
    result.append("\nDistributeur compatible Java, modèle 2004");
    result.append("\nStock : " + nombre + " bonbon");
    if (nombre != 1) {
        result.append("s");
    }
    result.append("\n");
    result.append("L'appareil " + etat + "\n");
    return result.toString();
}
```

Le distributeur modifié VI

```
}
```

La transition depuis EtatAPiece I

```
public class EtatAPiece implements Etat {
    Random hasard = new Random(System.currentTimeMillis());
    Distributeur distributeur;

    public EtatAPiece(Distributeur distributeur) {
        this.distributeur = distributeur;
    }

    public void insererPiece() {
        System.out.println("Vous ne pouvez pas insérer d'autre pièce");
    }

    public void ejecterPiece() {
        System.out.println("Pièce retournée");
        distributeur.setEtat(distributeur.getEtatSansPiece());
    }
}
```

La transition depuis EtatApiece II

```
public void tournerPoignee() {
    System.out.println("Vous avez tourné....");
    int gagnant = hasard.nextInt(10);
    if ((gagnant == 0) && (distributeur.getNombre() > 1)) {
        distributeur.setEtat(distributeur.getEtatGagnant());
    } else {
        distributeur.setEtat(distributeur.getEtatVendu());
    }
}
```

```
public void delivrer() {
    System.out.println("Pas de bonbon délivré");
}
```

```
public String toString() {
    return "attend que la poignée soit tournée";
}
}
```

Remarques

- Découplage entre les objets et le statechart
- Le pattern est très proche de "Stratégie" (même structure UML)
- Mais les deux différent dans le propos
 - Stratégie affecte un comportement à chaque classe (même s'il *peut* changer au cours du temps)
 - Etat ne donne à la classe que l'état initial, qui change (*obligatoirement*) dans les actions

solution “maison”, indiquer que ça pose des problèmes avec le multi-thread

(synchronisation)

Références

-  "Design Patterns - Catalogue de modèles de conceptions réutilisables", E.Gamma R.Helm R.Johnson J.Vlissides, Vuibert, Juillet 1999
-  "UML et les design patterns", C. Larman, 2^e édition, CampusPress, 2003.
-  "Design patterns tête la première", E. & E. Freeman, K. Sierra, B. Bates, O'Reilly, 2004.