

# Les patrons de conception (Design patterns)

Riadh BEN HALIMA  
riadh.benhalima@enis.rnu.tn

Wajdi LOUATI  
wajdi.louati@gmail.com

1

## Plan

- ▶ Historique & Motivation
- ▶ Le patron « Strategy »
- ▶ Le patron « Observer »
- ▶ Le patron « Decorator »
- ▶ Le patron « Abstract Factory »
- ▶ Le patron « Singleton »
- ▶ Le patron « Command »
- ▶ Le patron « Adapter »
- ▶ Le patron « Façade »

▶ 2

## Historique

- ▶ Notion de « patron » d'abord apparue en architecture :
  - ▶ l'architecture des bâtiments
  - ▶ la conception des villes et de leur environnement
- ▶ L'architecte Christopher Alexander le définit comme suit:
  - ▶ «Chaque modèle [patron] décrit un problème qui se manifeste constamment dans notre environnement, et donc décrit le cœur de la solution de ce problème, d'une façon telle que l'on peut réutiliser cette solution des millions de fois.» [Livre: *The Timeless Way of Building*, Oxford University Press 1979]
- ▶ Projeter la notion de patron à du logiciel : "**design pattern**"
  - ▶ premiers patrons à partir de 1987 (partie de la thèse de Erich Gamma)
  - ▶ puis Richard Helm, John Vlissides et Ralph Johnson («Gang of Four, GoF»)
  - ▶ premier catalogue en 1993 : *Elements of Reusable Object-Oriented Software*
- ▶ Vocabulaire:
  - ▶ modèles de conception= patrons de conception= motifs de conception= design patterns

▶ 3

## Motivation

- ▶ Pourquoi définir des patrons de conception
  - ▶ Construire des systèmes plus extensibles, plus robustes au changement
  - ▶ Capitaliser l'*expérience collective des informaticiens*
  - ▶ Réutiliser les solutions qui ont fait leur preuve
  - ▶ Identifier les avantages/inconvénients/limites de ces solutions
  - ▶ Savoir quand les appliquer
- ▶ Complémentaire avec les API
  - ▶ Une API propose des solutions directement utilisables
  - ▶ Un patron explique comment structurer son application avec une API
- ▶ Patron de conception dans le cycle de développement
  - ▶ Intervient en conception détaillée
  - ▶ Reste indépendant du langage d'implantation

▶ 4

## Définition

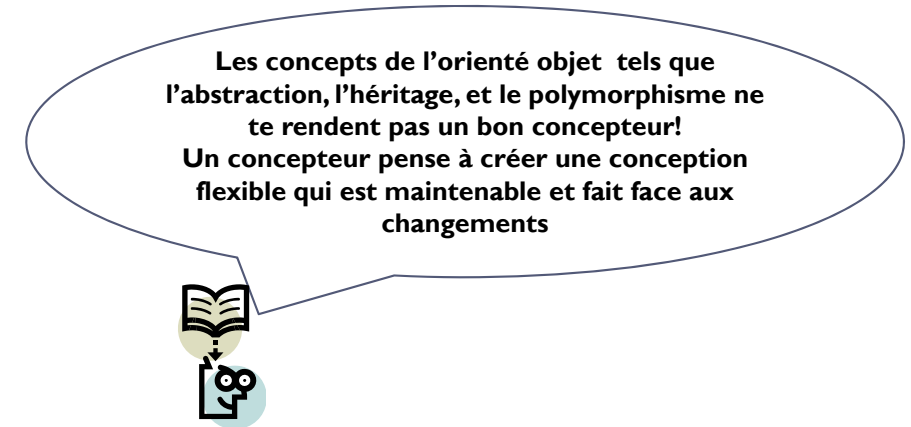
- ▶ **Définition : "Un patron de conception (design pattern) décrit une structure commune et répétitive de composants en interaction (la solution) qui résout un problème de conception dans un contexte particulier"**
- ▶ Au lieu de la réutilisation de code, on parle de la **réutilisation de l'expérience** avec les patrons
- ▶ Un bon patron de conception :
  - ▶ résout un problème
  - ▶ correspond à une solution éprouvée
  - ▶ favorise la réutilisabilité, l'extensibilité, etc.

▶ 5

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Intérêt et utilisation des patrons de conception

- ▶ La meilleure manière d'utilisation des patrons de conception est de les mémoriser en tête, puis reconnaître leurs emplacements et les appliquer dans la conception des applications



▶ 6

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

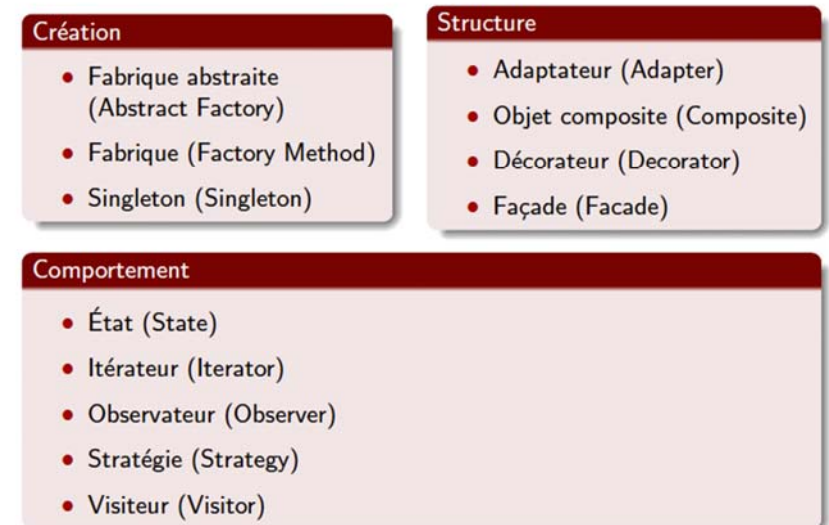
## Ce qu'il ne faut pas attendre des patrons de conception

- ▶ Une solution universelle prête à l'emploi
- ▶ Une bibliothèque de classes réutilisables
- ▶ L'automatisation totale de l'instanciation d'un patron de conception
- ▶ La disparition du facteur humain

▶ 7

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Trois sortes de patrons



▶ 8

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

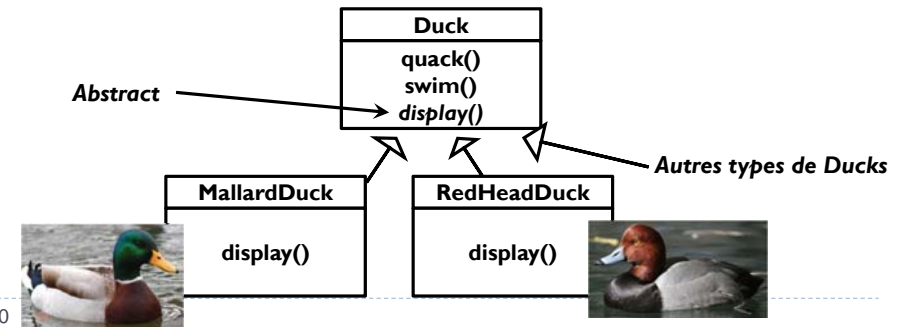
# Le patron "Strategy"

9

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## SimUDuck: Conception (1/22)

- Objectif: développement d'un jeu de simulation d'un bassin pour les canards
- Besoin: nager, cancaner, afficher, etc..
  - Supporter une large variété de canards
- Conception: OO
  - Une supère classe Canard (Duck) dont tous les canards héritent

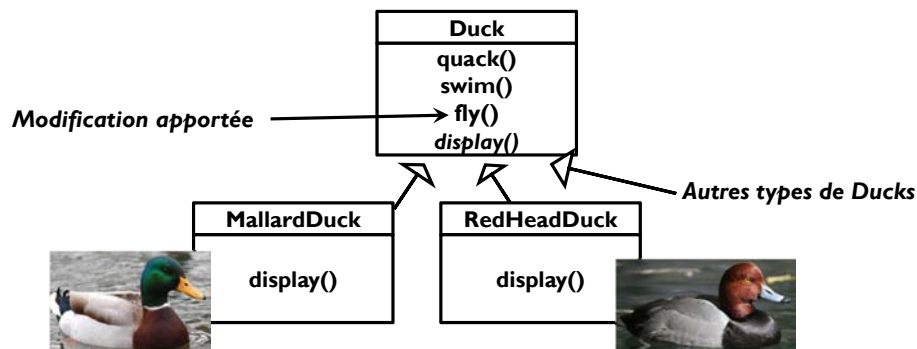


10

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## SimUDuck : Innovation (2/22)

- Objectif: Innovation (pour impressionner et vendre +)
- Besoin: simuler le vol des canards!
- Conception: OO
  - Ajouter la méthode fly() à la supère classe

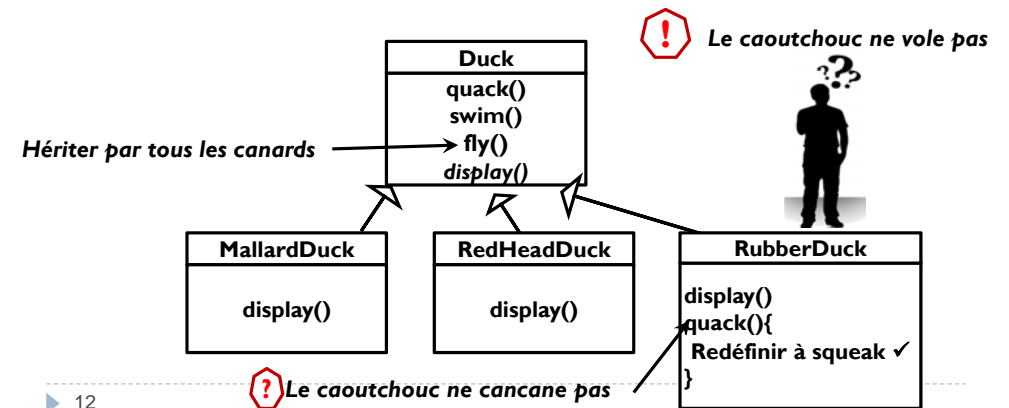


11

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## SimUDuck : Problèmes (3/22)

- Besoin: Au moment de la démonstration du simulateur, on nous demande de simuler des canards en caoutchouc
- Conception: OO
  - Ajouter la classe RubberDuck qui hérite de la supère classe Duck



12

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## SimUDuck : Constat (4/22)

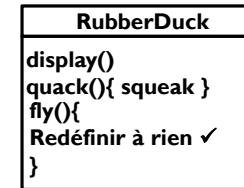
- Problème 1: Le canard en caoutchouc ne cancanne pas!
- Solution : Redéfinir la méthode quack() à squeak() (résolu)
- Problème 2: Le canard en caoutchouc ne vole pas! Toutefois, il hérite la méthode fly() de la supère classe Duck!
- Constat:
  - Ce que nous avons cru une utilisation formidable de l'héritage dans le but de la réutilisation, s'est terminé mal au moment de la mise à jour!
  - Une mise à jour du code a causé un effet global sur l'application!

► 13

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## SimUDuck : Solution?? (5/22)

- Problème 2: Le canard en caoutchouc ne vole pas! Toutefois, il hérite la méthode fly() de la supère classe Duck!
- Solution: Redéfinir la méthode fly() de RubberDuck



- Question: est ce que c'est résolu pour tous types de canards?

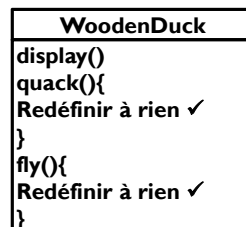


► 14

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## SimUDuck : Un autre canard (6/22)

- Nouveau type de canard: Canard en bois
- Problèmes levés:
  - Ce canard ne cancanne pas
  - Ce canard ne vole pas
- Solution: redéfinir (une autre fois) les méthodes quack() et fly()



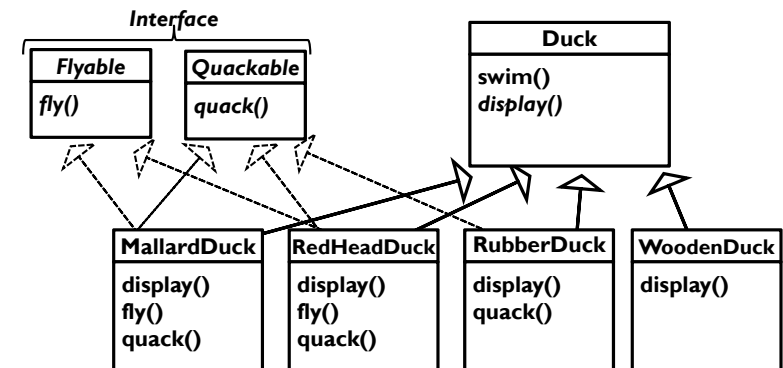
- Inconvénients de l'utilisation de l'héritage
  - Il est difficile de connaître le comportement de tous les canards
  - Un changement non-intentionnelle, affecte les autres canards

► 15

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## SimUDuck : Des interfaces? (7/X)

- Hypothèse: On nous demande de mettre à jour SimUDuck tous les 6 mois: La spécification demeure changeante
  - Vérifier fly() et quack() pour chaque nouveau canard
  - Ré-écrire (si besoin) fly() et quack()
- Solution possible pour contourner le problème: les interfaces

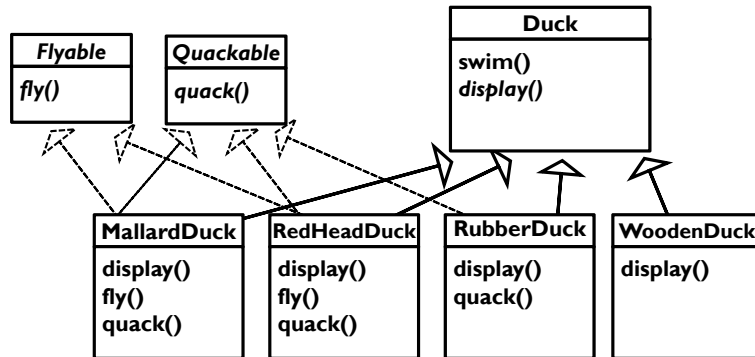


► 16

Que dites vous de cette conception ?

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## SimUDuck : Inconvénients (8/22)



- ▶ **Constat:**
  - ▶ **Duplication de code:** méthodes fly() et quack() dans les sous-classes
  - ▶ Autant d'interfaces tant qu'il y a un ensemble de canards ayant exclusivement un comportement commun (pondre: lay() pour les canards qui peuvent déposer un œuf)
- ▶ **Problème:** si on veut modifier/adapter légèrement la méthode fly(), il faut le faire pour toutes les classes des canards (10 classes, 100, ou +)

▶ 17

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## SimUDuck : Moment de réflexion (9/22)

- ▶ Pas toutes les sous-classes qui ont besoin de voler (fly) ou de cancaner (quack)
  - ▶ L'héritage n'est pas la bonne solution
- ▶ Les interfaces Flyable et Quackable résolvent une partie du problème
  - ▶ Détruit complètement la réutilisation du code pour ces comportements
  - ▶ La maintenance et la mise à jour représentent un vrai calvaire
- ▶ Supposant qu'il existe plus qu'une façon de voler
  - ▶ Maintenance plus difficile...

▶ 18

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## SimUDuck : Solution (10/22)

- ▶ **Solution:**
  - ▶ Design pattern : solution ultime, cheval blanc, sauveur...
- ▶ Trouvons une solution avec l'"ancienne-mode" et ce en appliquant les bonnes principes de la conception OO
  - ▶ Concevoir une application, ou un besoin de modification/changement peut être appliqué avec le moindre possible d'impact sur le code existant

**Donner des raisons de changement de code  
dans votre application**

▶ 19

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## SimUDuck : Principe de conception (11/22)

▶ **Règle I:** Identifier les aspects variables de mon application et les séparer de ce qui reste invariant

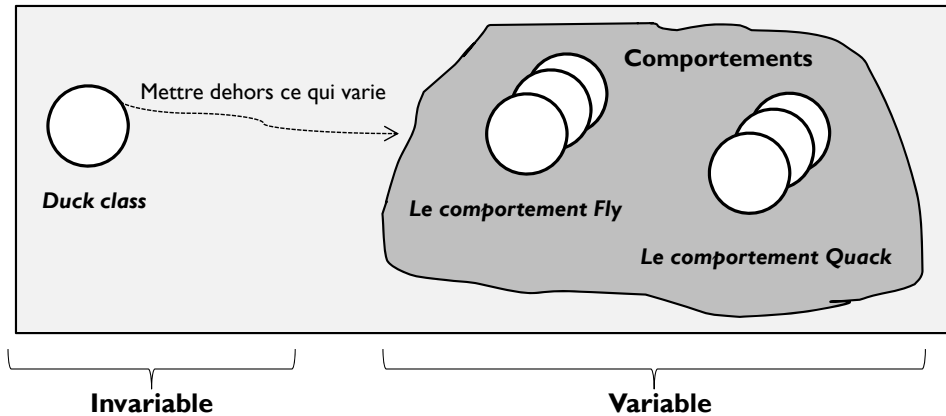
- ▶ C'est la base de tous les patrons de conception
- ▶ Système plus flexible+peu de conséquences inattendues
- ▶ **Mise en œuvre**
  - ▶ Prendre la partie qui varie et l'encapsuler. De cette façon, un changement ultérieur affecte la partie variable, sans toucher à celle invariable

▶ 20

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## SimUDuck : Séparation (12/22)

- La classe Duck est toujours la supère classe
- Les comportements fly() et quack() sont retirés, et mis dans une autre structure



21

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## SimUDuck : Conception des comportements (13/22)

- Conception initiale: l'inflexibilité des comportements a engendré des troubles
- On veut affecter les comportements aux instances des Ducks tout en permettant:
  - La création d'une instance (MallardDuck),
  - L'initialisation avec un type de comportement (type de vol)
  - La possibilité de changer le type de vol dynamiquement (?)

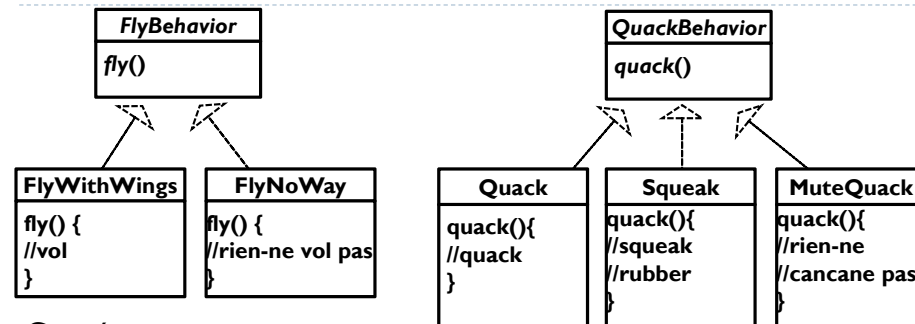
**Règle 2:** Programmer des interfaces, et non des implémentations

- Programmer pour les super-types!
- On utilise des interfaces pour représenter chaque comportement: **FlyBehavior** et **QuackBehavior**

22

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## SimUDuck : Conception des comportements (14/22)



### Conséquences:

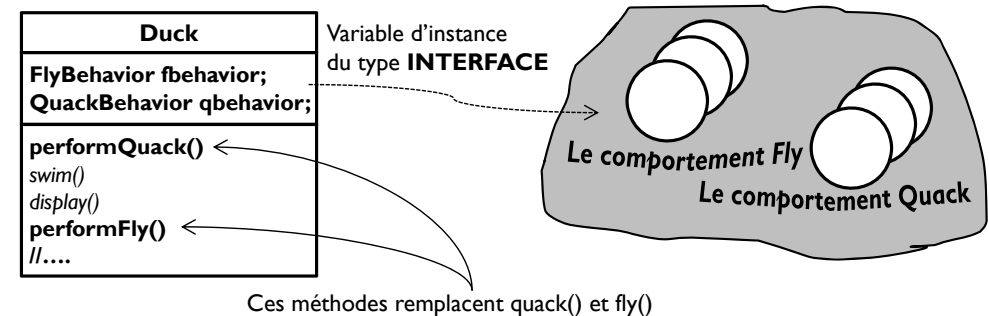
- On peut ajouter un nouveau comportement sans modifier ni le code des comportements existants, ni le code des classes des canards qui utilisent les comportements voler/cancaner
- Avec cette conception, d'autres objets peuvent réutiliser le comportement fly et quack, parce qu'ils ne sont plus cachés dans les classes canards.

23

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## SimUDuck : Intégration des comportements(15/22)

- La supère classe Duck, dont hérite tous les canards



- La clé: le canard délègue les comportements fly et quack, au lieu d'utiliser les méthodes fly() et quack() définies dans la supère classe Duck.

24

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## SimUDuck : Implémentation de la supère classe(16/22)

- ▶ La supère classe Duck, dont hérite tous les canards

```
public class Duck{
    QuackBehavior qbehavior;
    FlyBehavior fbehavior;
    //...

    public void performQuack(){
        qbehavior.quack();
    }
    //...
}
```

→ Chaque type de canard initialise ces attributs selon ses besoins.  
(par FlyWithWings pour le MallardDuck )

→ Grace au polymorphisme, la bonne méthode sera invoquée dans la sous-classe du type de canard.  
(Déléguée à la classe gérant le comportement)

▶ 25

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## SimUDuck : Implémentation d'un canard (17/22)



→ Cette classe inclut les méthodes réalisant le comportement fly et quack, par héritage (performQuack(), etc..)

```
public class MallardDuck extend Duck{
    public MallardDuck (){
        fbehavior = new FlyWithWings();
        qbehavior = new Quack();
    }
    public void display(){
        System.out.println("Je suis un canard Mallard");
    }
}
```

→ Initialisation des attributs déclarés dans la supère classe Duck

▶ 26

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## SimUDuck : Tester le code du canard(18/22)

- ▶ Développer et compiler [Utiliser NetBeans/Eclipse]:
  - ▶ La classe abstraite Duck (Duck.java)
  - ▶ Le comportements: FlyBehavior.java, FlyWithWings.java et FlyNoWay.java,
  - ▶ Le comportement : QuackBehavior.java, Quack.java, Squeak.java et MuteQuack.java
  - ▶ Les classes MallardDuck.java et WoodenDuck.java
- ▶ Tester toutes les méthodes des canards créés dans un main: MallardDuckSim.java

▶ 27

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## SimUDuck : Le comportement dynamique (19/22)

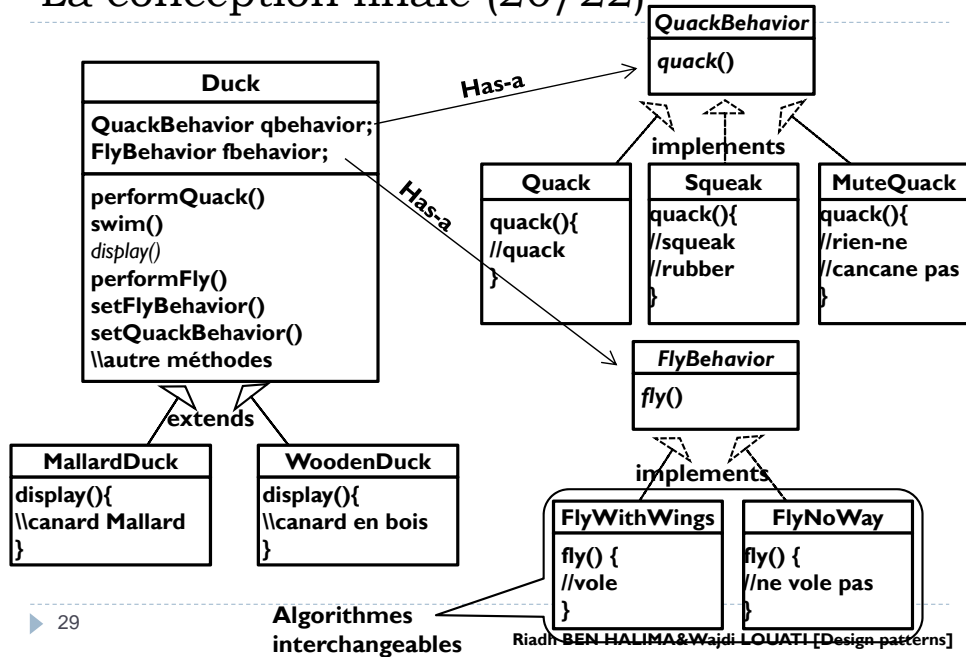
- ▶ Changement dynamique de comportement
  - ▶ Ajouter les méthodes: setFlyBehavior() et setQuackBehavior()
  - ▶ Développer le canard RedHeadDuck (RedHeadDuck.java)
  - ▶ Implanter le nouveau comportement "vole-force-fusée" FlyRocketPowered (FlyRocketPowered.java)
  - ▶ Tester le nouveau canard dans un main RedHeadSim.java
  - ▶ Changer le comportement "voler" de FlyWithWings à FlyRocketPowered. Penser à utiliser le setter afin d'obtenir ces deux affichages: "Je peux voler" & "Je vole comme une fusée"
- ▶ Donner (et ne pas implémenter) les modifications à faire afin d'ajouter le comportement manger: eat()

▶ 28

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]



## SimUDuck : La conception finale (20/22)



29

Riadh BEN HALIMA & Wajdi LOUATI [Design patterns]

## SimUDuck : Composition/Héritage (21/22)

- Has-a: liaison intéressante
  - Chaque canard possède un `FlyBehavior` et `QuackBehavior` qui délègue flying et quacking
  - Composition (mettre les 2 classes ensemble)
  - Encapsuler une famille d'algorithmes dans leur propre ensemble de classes
  - Remplacer l'héritage pour favoriser le changement dynamique du comportement

**Règle 3:** Favoriser la composition sur l'héritage

30

Riadh BEN HALIMA & Wajdi LOUATI [Design patterns]

## SimUDuck : Notre premier patron (22/22)

- Notre premier patron: **STRATEGIE**

**Le patron stratégie** cherche principalement à séparer un objet de ses comportements/algorithmes en encapsulant ces derniers dans des classes à part.

Pour ce faire, on doit alors définir une famille de comportements ou d'algorithmes encapsulés et interchangeables.

31

Riadh BEN HALIMA & Wajdi LOUATI [Design patterns]

## Les avantages du patron

- Si les algorithmes/comportements sont dans une classe à part, il est beaucoup plus facile de:
  - se retrouver dans le code principale
  - enlever, ajouter et modifier un algorithme/comportement
  - diminuer l'utilisation de tests conditionnels
  - éliminer la redondance et le couper/coller
  - accroître la réutilisabilité du code ainsi que sa flexibilité

32

Riadh BEN HALIMA & Wajdi LOUATI [Design patterns]



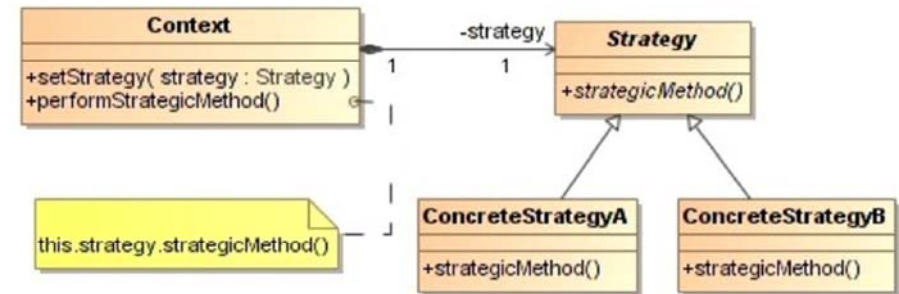
## L'implémentation du patron

- Pour implémenter le patron « Strategy » on doit:
  - 1. Définir une interface commune à tout les algorithmes ou comportements de même famille.
    - Ceci ce fait habituellement en créant une classe abstraite.
  - 2. Créer les classes comportant les algorithmes ou les comportements à partir de l'interface commune.
  - 3. Utiliser la stratégie voulue dans le code de l'objet.

► 33

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Représentation du patron



► 34

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Récapitulatif (1/2)

- Bases de l'OO:
  - Abstraction, Encapsulation, Polymorphisme & Héritage
- Principes de l'OO
  - Encapsuler ce qui varie
  - Favoriser la composition sur l'héritage
  - Programmer avec des interfaces et non des implémentations
- Patron de l'OO (stratégie)
  - Le patron stratégie définit une famille d'algorithmes, les encapsule, et les rend interchangeable. Ce patron laisse l'algorithme varier indépendamment du client qu'il l'utilise.

► 35

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Récapitulatif (2/2)

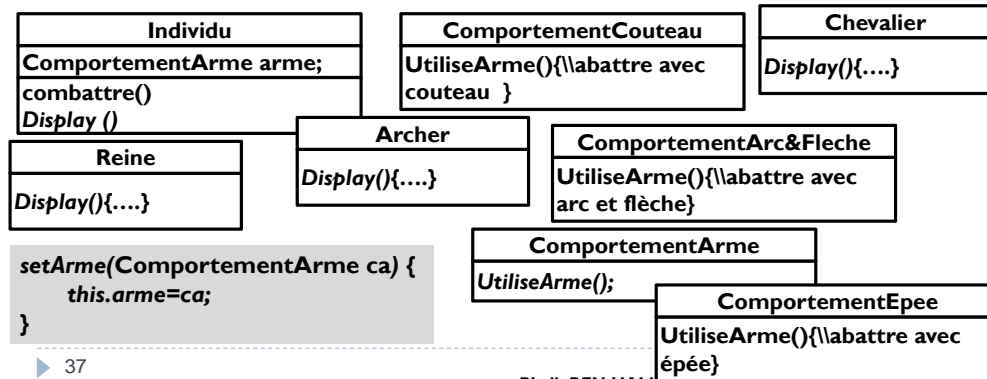
- Connaître les bases de l'OO ne fait pas de toi un bon concepteur
- Les meilleures conceptions OO sont réutilisables, flexibles et maintenables
- Les patrons nous guident à construire des systèmes avec les bonnes qualités de la conception OO
- Les patrons sont des expériences prouvées dans l'OO
- Les patrons ne donnent pas de code, mais plutôt des solutions générales pour des problèmes de conception
- Les patrons ne sont pas inventés, mais découverts
- La majorité des patrons et des principes adressent les problèmes de changement dans le logiciel
- On essaie toujours de prendre ce qui varie des systèmes et on l'encapsule
- Les patrons offrent un langage partagé qui peut maximiser la valeur de la communication avec les autres développeurs

► 36

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Exercice (1/2)

- Ci-dessous, on donne l'ensemble de classes et interfaces d'un jeu d'action et d'aventure. Il y a des classes d'individus avec des classes pour les comportements d'armes que les individus peuvent utiliser. Chaque individu peut utiliser une seule arme à la fois, mais peut la changer à tout moment durant le jeu. La tâche demandée est d'ordonner le tout.



## Exercice (2/2)

1. Arranger les classes
2. Identifier les classes, les classes abstraites des interfaces
3. Relier les entités pas des flèches ou:
  1. représente **extends**
  2. représente **implements**
  3. représente **has-a**
4. Mettre la méthode `setArme()` dans la classe correspondante
5. Implémenter et tester cette conception dans un main. Penser à changer dynamiquement le comportement de l'archer après avoir finir ces arcs.

► 38

Riadh BEN HALIMA & Wajdi LOUATI [Design patterns]

# Le patron "Observer"

► 39

Riadh BEN HALIMA & Wajdi LOUATI [Design patterns]

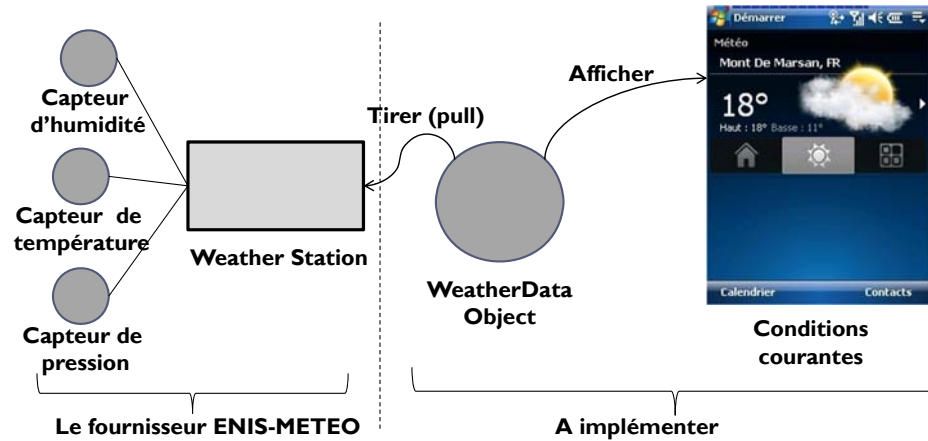
## Station météo: Spécification (1/14)

- Objectif: Construire une nouvelles génération de stations d'observation météo sur Internet
- Besoin: Afficher les conditions courantes, les statistiques météorologique et les prévisions météo.
- Poursuivre les conditions météorologiques (Température, Humidité, Pression, etc.)
  - On veut mettre en œuvre une API (ENIS-METEO) de façon que d'autres développeurs peuvent écrire leur propre afficheur de météo.
- Conception: OO
- Une classe `WeatherData` qui récupère les données de la station météo (ENIS-METEO) et les offre aux afficheurs.

► 40

Riadh BEN HALIMA & Wajdi LOUATI [Design patterns]

## Station météo: Analyse (2/14)

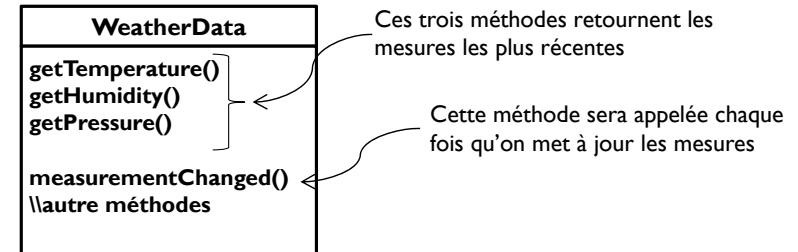


- Il faut créer une application qui utilise l'objet WeatherData afin de mettre à jour trois afficheurs: les conditions courantes, les statistiques météorologiques & les prévisions météo.

41

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Station météo: Conception (3/14)

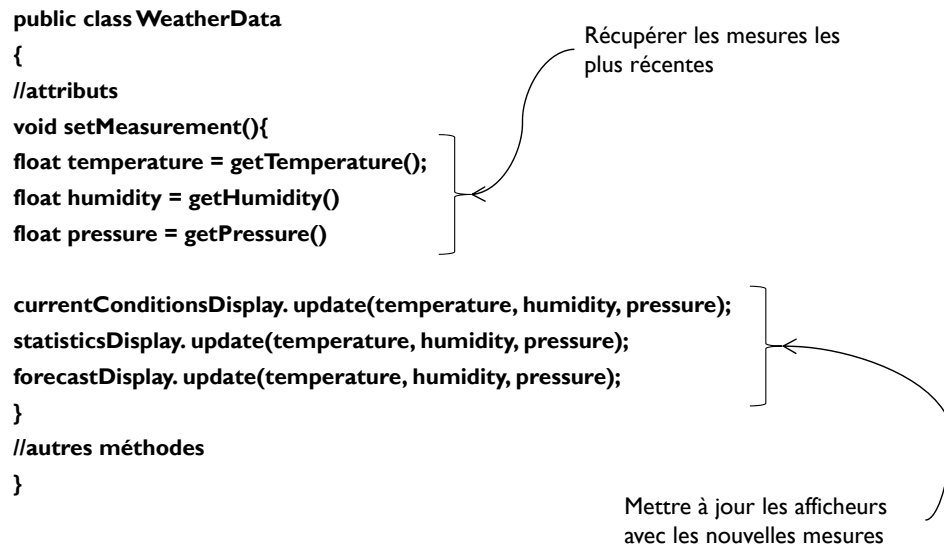


- Nous allons développer trois afficheurs:
  - conditions courantes (CurrentConditionsDisplay.java)
  - statistiques météorologique (StatisticsDisplay.java)
  - prévisions météo (ForecastDisplay.java)
- Rq: On ne s'intéresse pas à la manière dont les variables sont fixées. On suppose que l'objet WeatherData connaît comment les mettre à jour à partir de la station ENIS-METEO

42

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Station météo: Une solution possible (4/14)



43

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

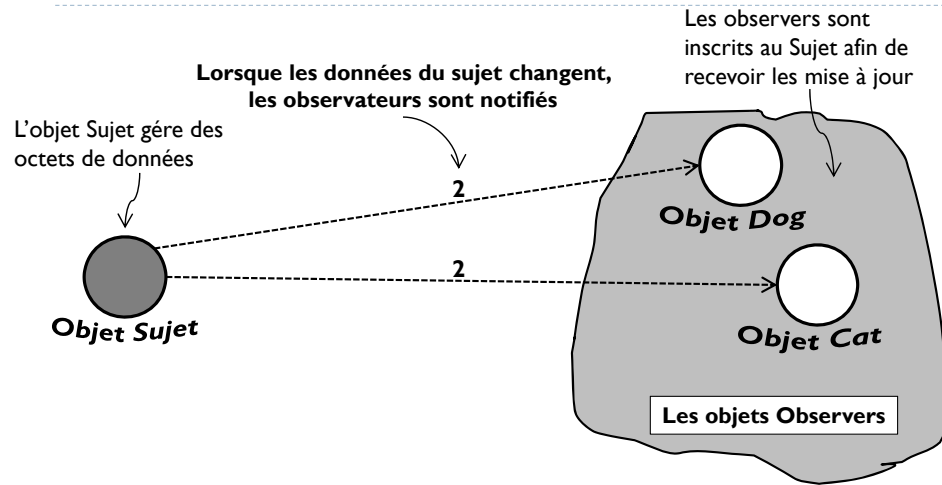
## Station météo: Constat (5/14)

- Problème : En codant des implémentations concrètes, on ne peut pas ajouter/supprimer d'autres afficheurs, sans faire un changement du programme!
  - Rq: on peut au moins utiliser une interface qui contient la méthode update().
- Solution : le **patron observer**
- Exemple:
  - Une maison d'édition commence l'édition d'un journal
  - Vous vous inscrivez à ce journal, et pour chaque nouvelle édition, vous recevez votre copie
  - Vous vous désinscrivez lorsque vous ne voulez plus recevoir des journaux
  - Les gens, les hôtels etc. peuvent constamment s'inscrire et se désinscrire à ce journal.

44

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

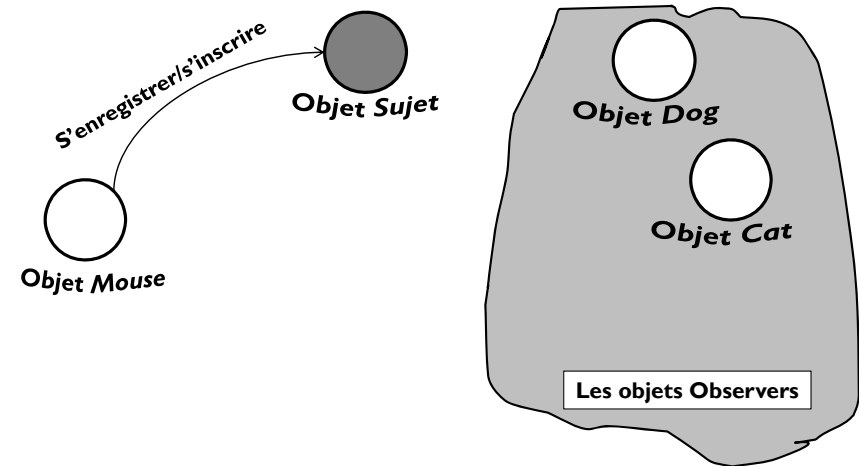
## Station météo: Publier/Souscrire= Patron Observer (6/14)



► 45

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

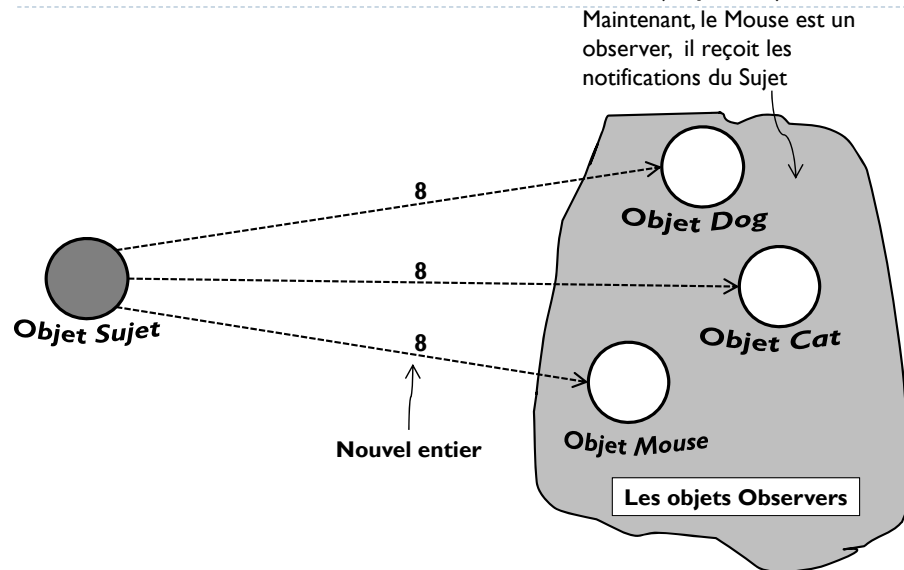
## Station météo: Patron Observer : S'inscrire (7/14)



► 46

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

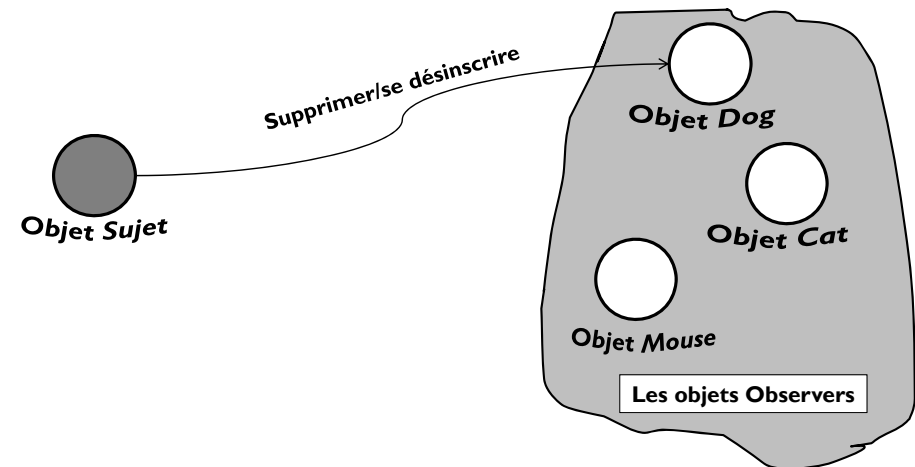
## Station météo: Patron Observer : Notification(8/14)



► 47

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

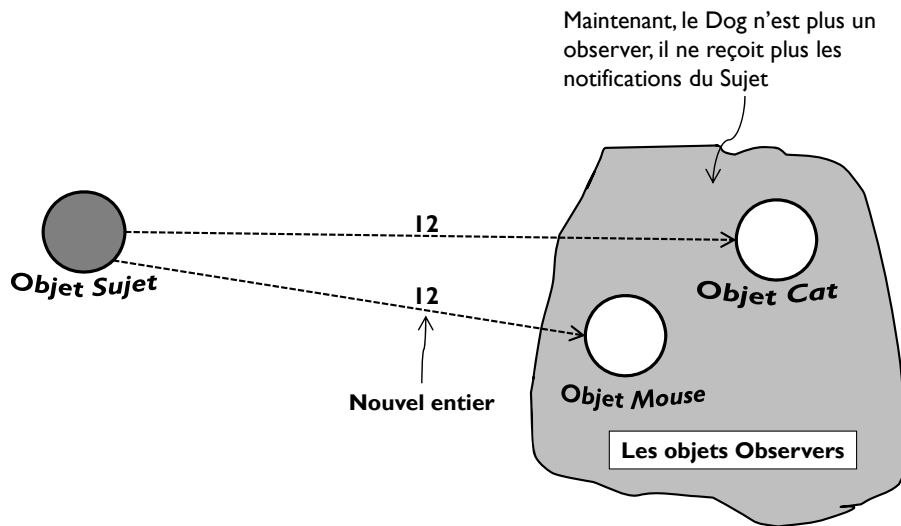
## Station météo: Patron Observer : Désinscrire (9/14)



► 48

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Station météo: Patron Observer : Notification (10/14)



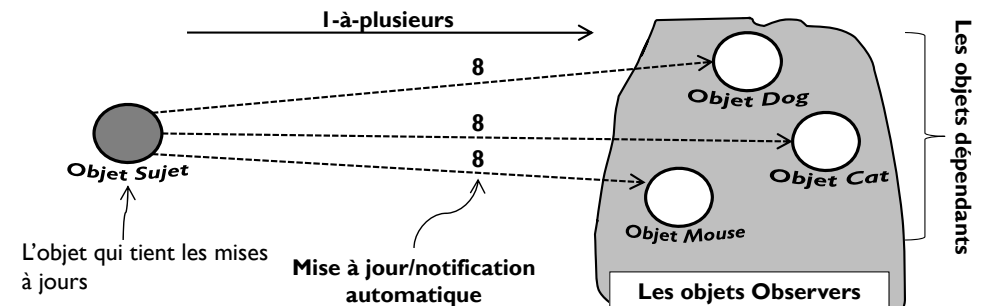
49

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Station météo: Le patron Observer (11/14)

### ► Définition: **Observer**

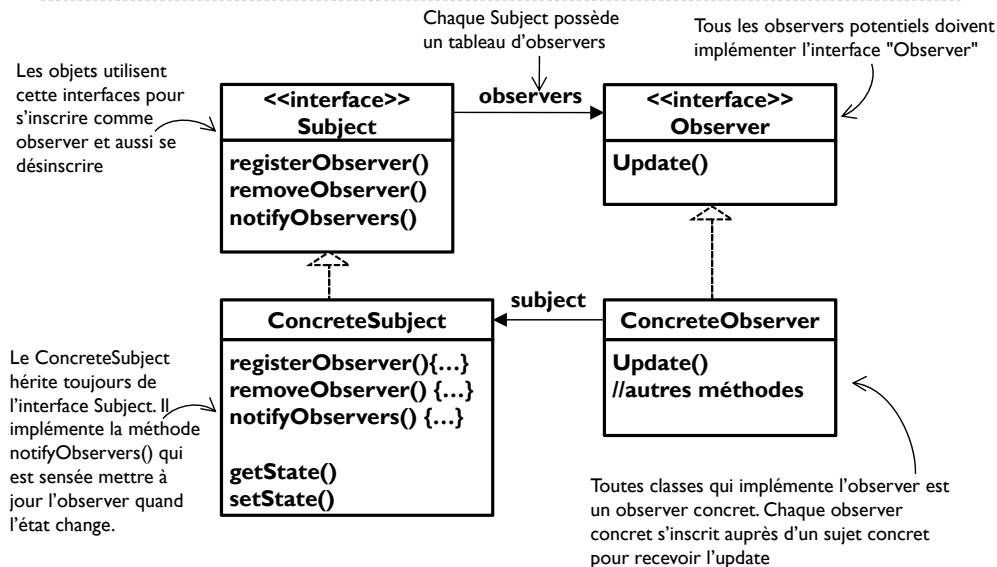
- Le **patron observer** définit une dépendance 1-à-plusieurs entre des objets de façon que à chaque changement de l'état de l'objet, tous ces dépendants sont notifiés et mis à jour automatiquement.



50

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Station météo: Diagramme de classes du patron (12/14)



51

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Station météo: La puissance du couplage faible (13/14)

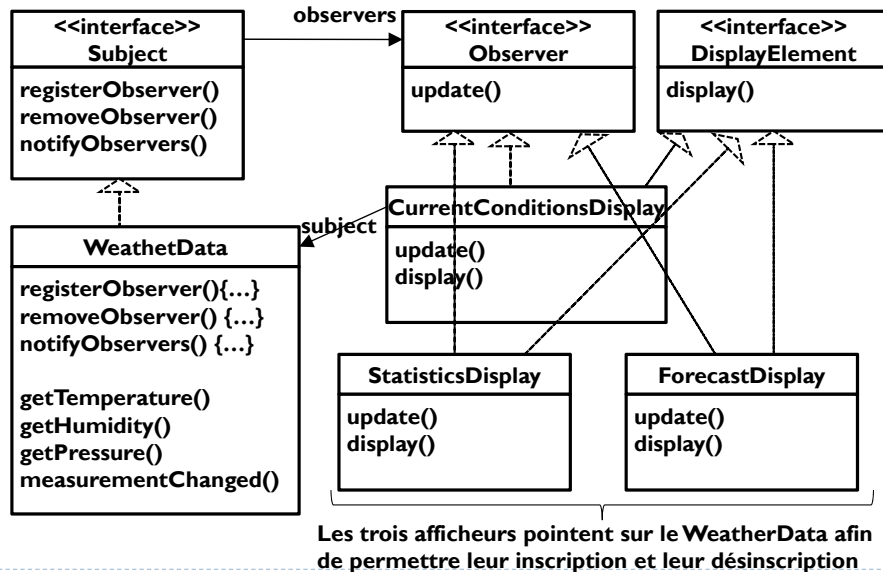
- Deux objets, qui sont faiblement couplés, entrent en interaction avec très peu de connaissance de l'un envers l'autre.
- Le patron Observer permet de réaliser une conception où le Subject et l'Observer sont faiblement couplés
- Couplage faible : plus d'indépendance et de flexibilité
  - La seule chose que le Subject a besoin de connaître sur l'Observer est qu'il implémente une certaine interface.
  - On peut ajouter/remplacer/supprimer un observer à tout moment sans toucher au Subject
  - On peut réutiliser le Subject ou l'Observer facilement parcequ'il ne sont pas fortement couplés.

► **Règle 4:** Opter pour une conception faiblement couplée entre les objets qui interagissent

52

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

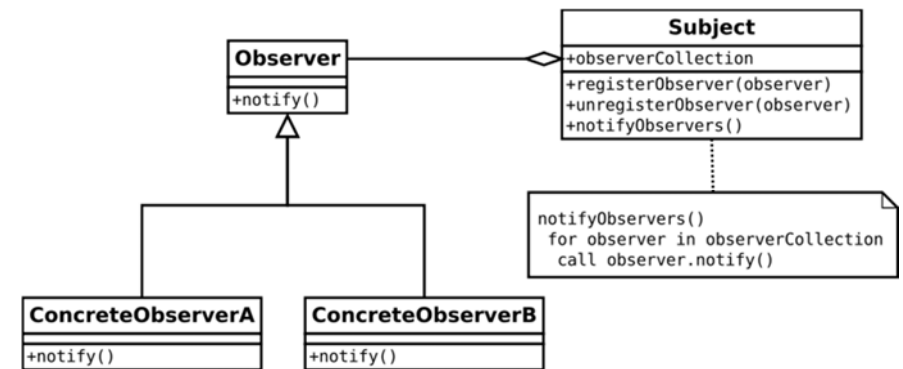
## Station météo: La conception finale (14/14)



53

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Le Patron Observateur



- Le patron de conception observateur/observable est utilisé pour envoyer un signal à des modules qui jouent le rôle d'observateur.
- En cas de notification, les observateurs effectuent alors l'action adéquate en fonction des informations qui parviennent depuis les modules qu'ils observent (les "observables").

54

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Récapitulatif (1/2)

- Bases de l'OO:
  - Abstraction, Encapsulation, Polymorphisme & Héritage
- Principes de l'OO
  - Encapsuler ce qui varie
  - Favoriser la composition sur l'héritage
  - Programmer avec des interfaces et non des implémentations
  - Opter pour une conception faiblement couplée entre les objets qui interagissent
- Patron de l'OO
  - Strategy: définit une famille d'algorithmes interchangeables
  - Observer**: définit une dépendance 1-à-plusieurs entre objets, de façon que pour chaque changement de l'état d'un objet, ses dépendants sont notifiés et mis à jour automatiquement.

55

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Récapitulatif (2/2)

- Le patron observer définit une relation 1-à-plusieurs entre objets.
- Le Subject/Observable met à jour les observateurs à travers une interface commune.
- L'observer est faiblement couplé avec le Subject. Ce dernier ne connaît rien d'eux à part qu'ils implémentent l'interface Observer.
- On peut récupérer les données du Subject en mode pull/push. (Le Subject fait le push, semble plus correct)
- On ne dépend pas d'un ordre spécifique de notification entre les observateurs
- Java possède plusieurs implémentations de ce patron

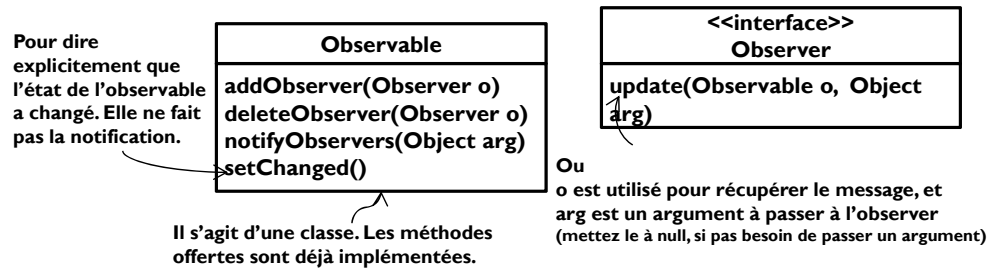
56

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Exercice

- Appliquer le patron Observer sur l'application station météo en se basant sur les implémentations du Subject et de l'Observer offertes par Java :

- java.util.Observable : Subject du patron (Attention! Il s'agit d'une classe)
- java.util.Observer : Observer du patron



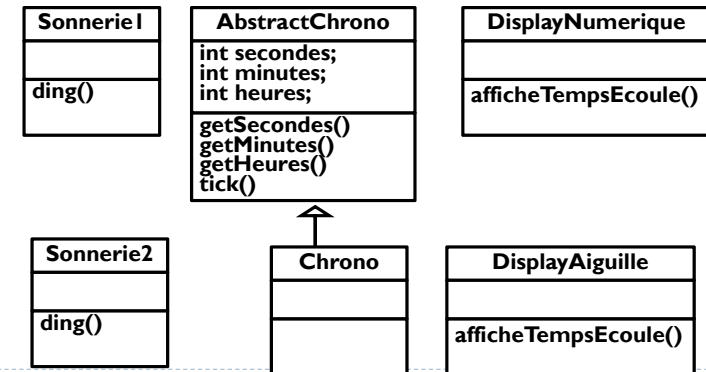
- Expliquer comment Java utilise le patron observer pour la gestion des évènements sur les interfaces graphiques.

► 57

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Exercice (1/2)

- Les méthodes getSecondes, getMinutes et getHeures permettent d'accéder aux valeurs des secondes, minutes et heures respectivement. La classe Chrono est une classe concrète implantant l'interface de la classe AbstractChrono. Lorsqu'une seconde passe (tick()), les Displays doivent afficher les heures, les minutes et les secondes et les Sonneries doit faire entendre un tintement chaque heure.



► 58

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Exercice (2/2)

- Utiliser le pattern Observer pour définir les interactions entre Chrono, Display et Sonnerie
- Donnez le diagramme de classes et l'implémentation.

- ```

public static void main(String[] args) {
    Chrono c=new Chrono();
    Sonnerie_Observer S1=new SonnerieI(c);
    Sonnerie_Observer S2=new Sonnerie2(c);
    Display_Observer D1= new DisplayNumerique(c);
    Display_Observer D2= new DisplayAiguille(c);
    for(int i=1;i<5000;i++)
    { try{Thread.sleep(1000);}
      catch(InterruptedException e) {System.out.print("erreur");}
      c.tick((i%3600)%60, (int)((i%3600)/60), (int)(i/3600)); } }
    
```

► 59

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

# Le patron "Decorator"

► 60

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]



## StarCoffee : Spécification (1/10)

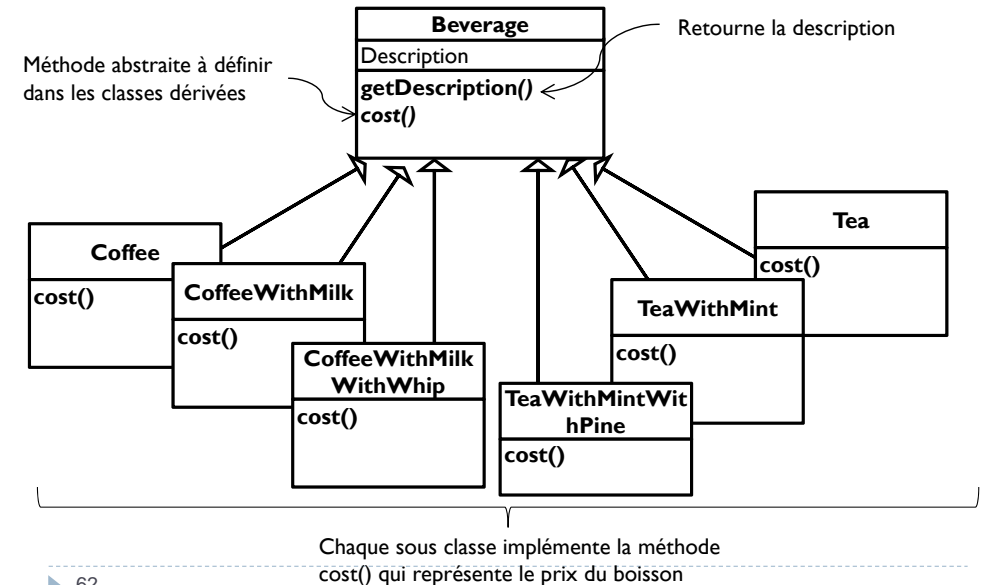


- ▶ **Objectif:** Mettre en œuvre un système de gestion des offres de boisson pour la clientèle de StarCoffee
- ▶ **Besoin:** Décrire les ajouts en extra, et calculer le prix total
  - ▶ Thé, Thé-à-la-menthe, Thé-à-la-mente-aux-pignons, etc..
  - ▶ Café, Café-au-Lait, Café-au-Lait-à-la-mousse, etc..
- ▶ **Conception: OO**
  - ▶ Concevoir une supère classe Boisson que toutes les autres classes héritent.
  - ▶ Définir autant de classes qu'il y en a de types de boissons :
    - ▶ Beverage.java, Coffee.java, CoffeeWithMilk.java, CoffeeWithMilkWithWhip.java, Tea.java, TeaWithMint.java, TeaWithMintWithPine.java, etc.

▶ 61

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## StarCoffee : Conception (2/10)

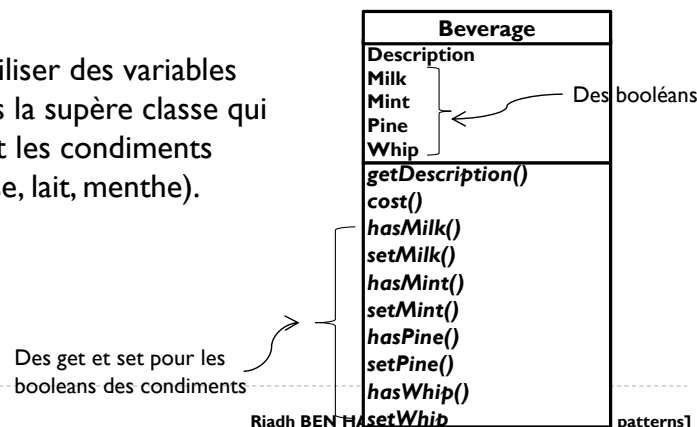


▶ 62

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## StarCoffee : Problème (3/10)

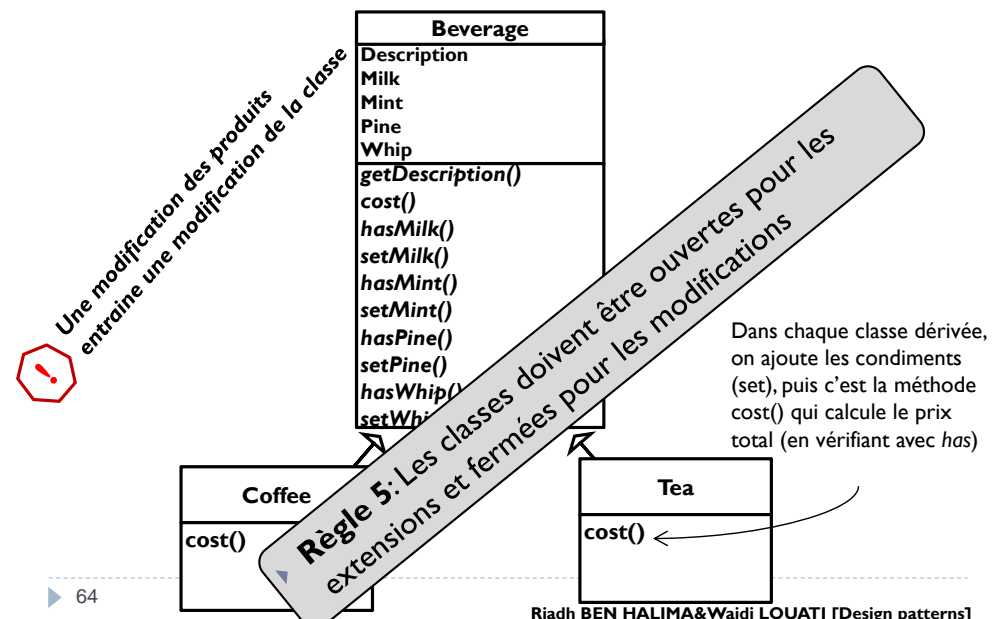
- ▶ **Problème :** En plus des deux produits présentés précédemment, StarCoffee offre une variété d'autres boissons et condiments: (Si on offre 2 types de Thé, on doit ajouter plusieurs autres classes (selon les condiments possibles), etc.
  - ▶ Constat: éclatement du diagramme de classes par un nombre ingérable de classes
- ▶ **Solution :** Utiliser des variables d'instance dans la supère classe qui représenteront les condiments (pignon, mousse, lait, menthe).



▶ 63

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## StarCoffee : Nouvelle conception (4/10)

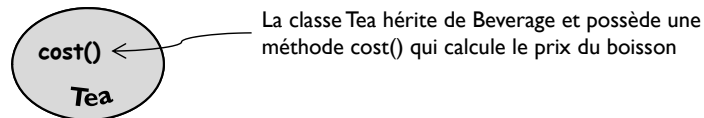


▶ 64

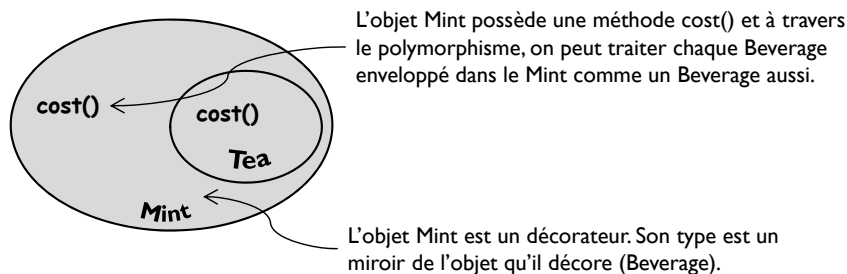
Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## StarCoffee : Un boisson décoré (5/10)

1. On commence par l'objet Tea



2. Le client choisit la menthe, alors on crée un objet Mint qui enveloppe le Tea

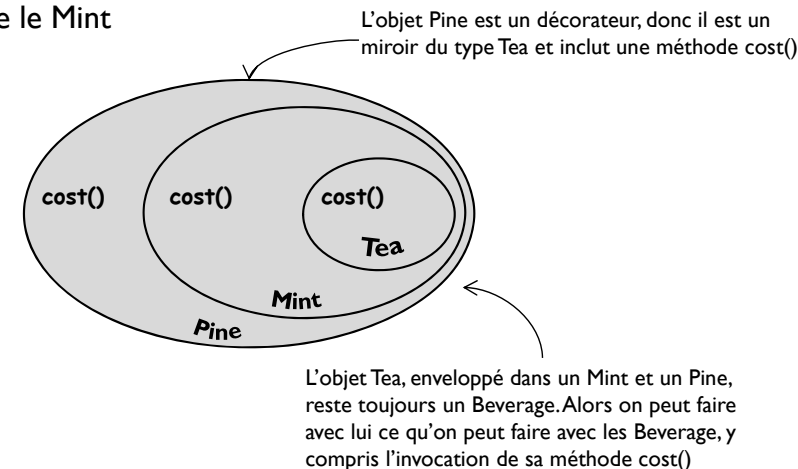


► 65

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## StarCoffee : Un boisson décoré (6/10)

3. Le client veut aussi des pignons, alors on crée un objet Pine qui emballe le Mint

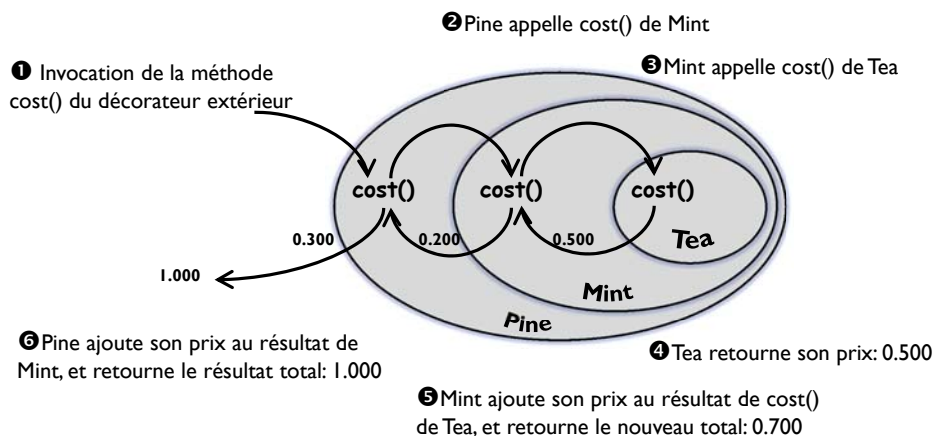


► 66

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## StarCoffee : Le coût du boisson décoré (7/10)

- L'idée est de calculer le coût en partant du décorateur le plus extérieur (Pine) et puis, ce dernier délègue le calcul à l'objet décoré, etc.



► 67

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## StarCoffee : Le patron Decorator (8/10)

- Définition: **Decorator**

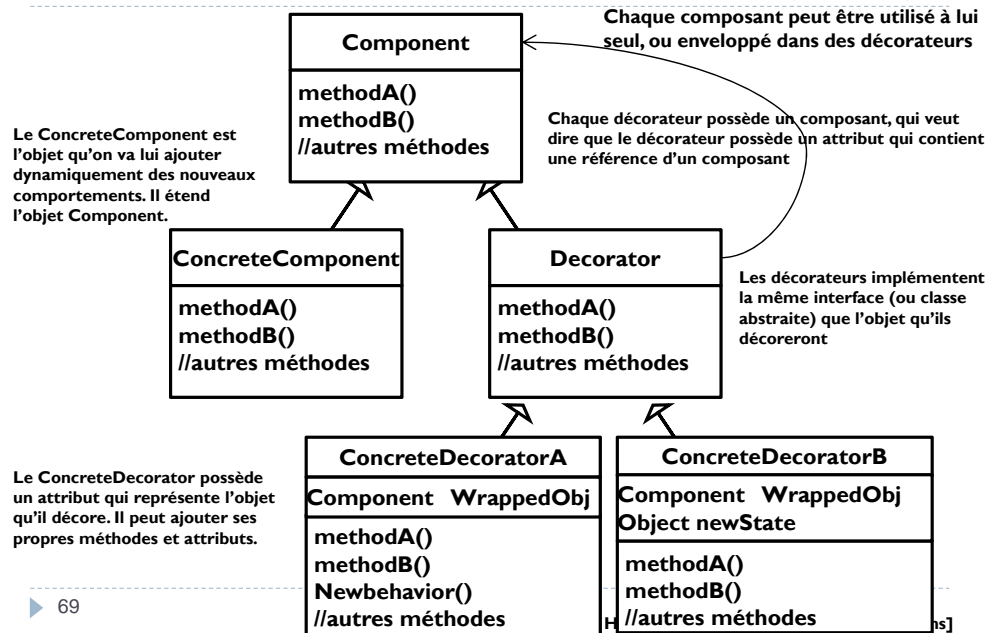
- Le **patron decorator** attache des responsabilités additionnelles à un objet dynamiquement. Les décorateurs offrent une alternative flexible de sous-classement afin d'étendre les fonctionnalités.

► 68

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

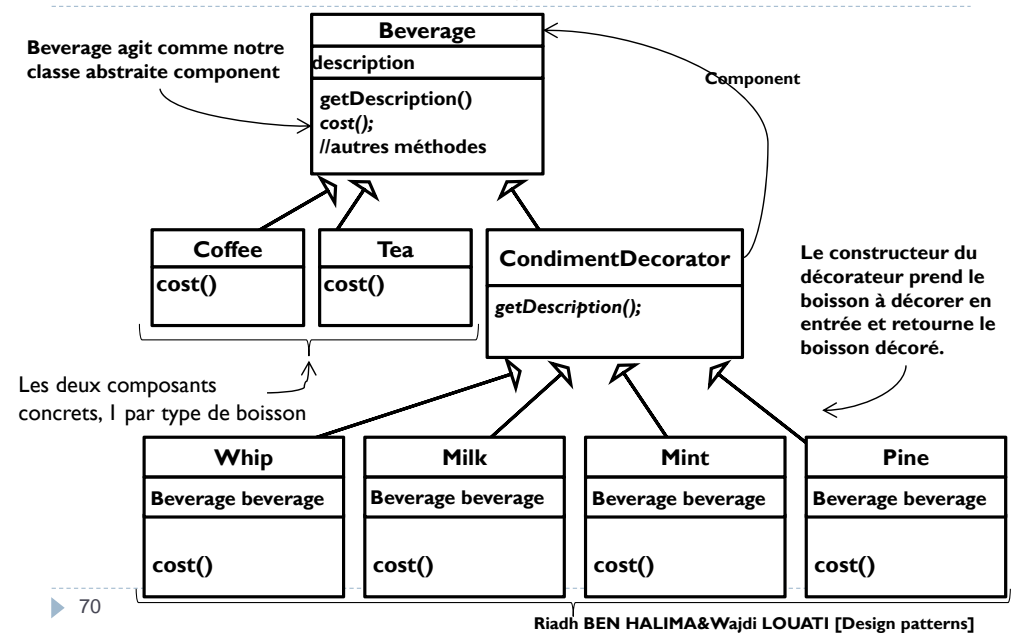
## StarCoffee :

### Le diagramme de classes du patron(9/10)



## StarCoffee :

### La conception finale (10/10)



## Récapitulatif (1/2)

- ▶ Bases de l'OO: Abstraction, Encapsulation, Polymorphisme & Héritage
- ▶ Principes de l'OO
  - ▶ Encapsuler ce qui varie
  - ▶ Favoriser la composition sur l'héritage
  - ▶ Programmer avec des interfaces et non des implémentations
  - ▶ Opter pour une conception faiblement couplée
  - ▶ Les classes doivent être ouvertes pour les extensions et fermées pour les modifications
- ▶ Patron de l'OO
  - ▶ Strategy: définit une famille d'algorithmes interchangeables
  - ▶ Observer: définit une dépendance 1-à-plusieurs entre objets.
  - ▶ **decorator**: attache des responsabilités additionnelles à un objet dynamiquement. Les décorateurs offrent une alternative flexible de sous-classement afin d'étendre les fonctionnalités.

## Récapitulatif (2/2)

- ▶ L'héritage est une forme d'extension, mais il n'est pas nécessairement la meilleure manière pour obtenir la flexibilité dans notre conception
- ▶ Le patron decorator implique un ensemble de classes de décorations qui sont utilisées pour envelopper les composants concrets.
- ▶ Les classes décorateurs reflètent le type de composant qu'ils décorent.
- ▶ Les décorateurs changent le comportement de leurs composants tout en ajoutant des nouvelles fonctionnalités après/avant (ou à la place de) l'appel des méthodes des composants
- ▶ On peut envelopper un composant dans n'importe quel nombre de décorateurs
- ▶ Les décorateurs sont transparents par rapport au client du composant

## Exercice (1/3)

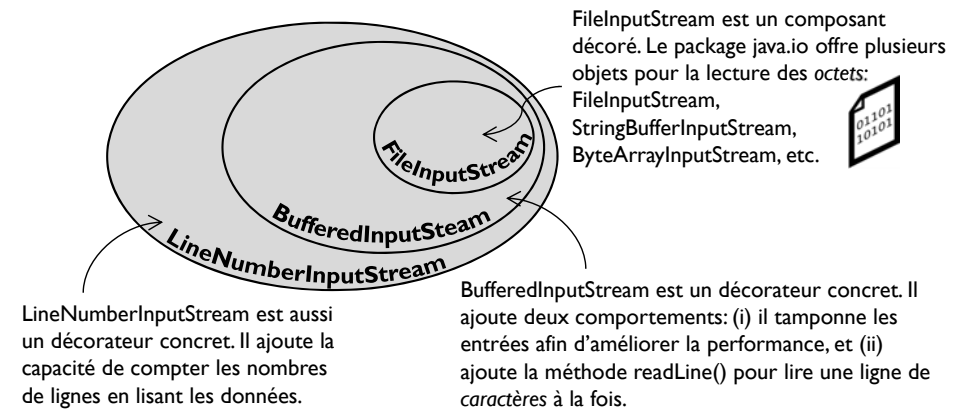
1. Comment faire pour obtenir un café avec "double mousse"?
2. StarCoffee a ajouté un nouveau boisson (Citronnade) au système, comment procéder pour l'inclure dans la conception actuelle?
3. StarCoffee veut introduire des tailles pour ses menus: SMALL, MEDIUM et LARGE. Comment prendre en charge cette nouvelle spécification, si la taille modifie seulement les prix des composants concrets?

73

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Exercice (2/3)

- ▶ Avec le patron decorator, le package java.io doit donner plus de sens, puisqu'il se base largement sur ce patron.

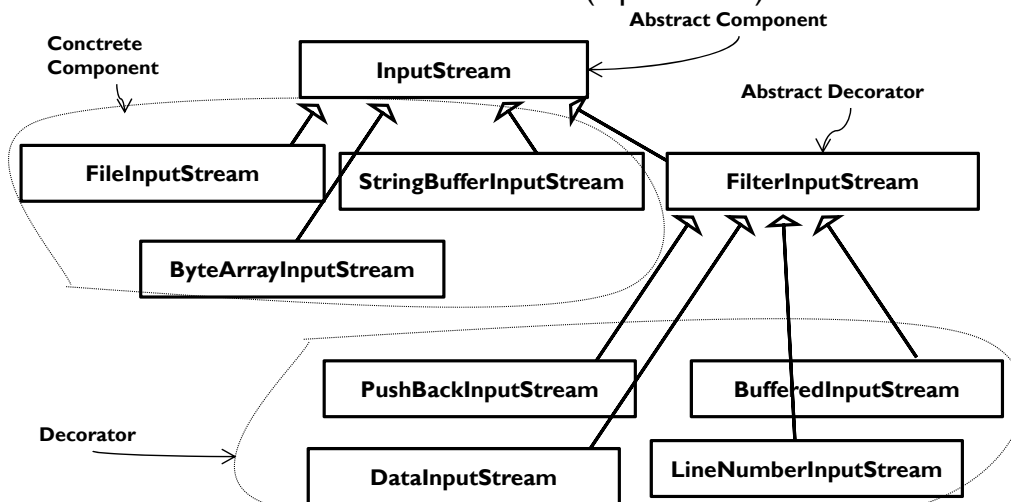


74

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Exercice (3/3): Décoration de java.io

1. Ecrire un décorateur qui convertit tous les caractères majuscules en minuscules dans le flux d'entrée (InputStream).



75

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

- ▶ L'objectif de cet exercice est de mettre en œuvre un système flexible de gestion des offres de voiture pour la clientèle de StarCar. Le besoin de cette société se résume à décrire les options demandées par le client (VitreElectrique, AirBag et ABS) et inclure son cout au prix total de la voiture choisie. Deux types de voiture sont gérés par la société, à savoir, camionnette et berline. Chaque voiture est caractérisée par un cout et une description textuelle. Le prix de chaque type de voiture ainsi que celui de chaque option est à fixer au moment de la création.
- ▶ En utilisant le patron Decorator, donnez le diagramme de classes de l'application CarStar. (Précisez les méthodes et les attributs, correspondant au bout de code ci-dessous)

76

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

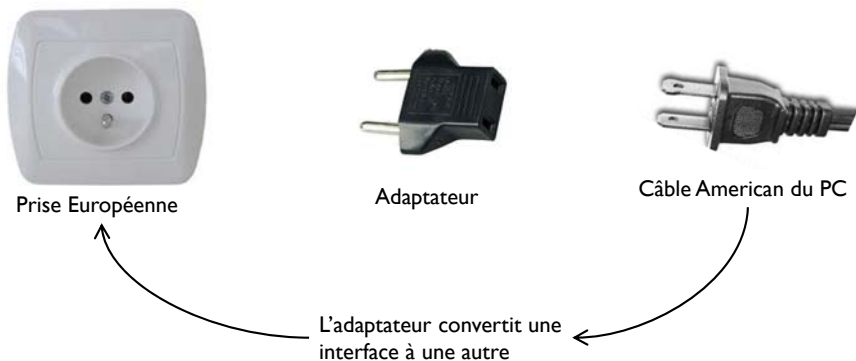
```

public static void main(String[] args) {
    Voiture v1=new Camionnette ("P404",10000);
    Voiture v2=new Berline ("P407",20000);
    v1=new ABS(v1,800);//800 représente le prix de l'option ABS
    v2=new VitreElectrique(v2,1000);// 1000 représente le prix de l'option
    v2=new AirBag(v2,1200);// 1200 représente le prix de l'option
    System.out.println("La voiture est une "+v1.getDescription());
    //affiche: La voiture est une P404 avec ABS
    System.out.println("Son prix est:"+ v1.cost());
    //affiche: Son prix est 10800
    System.out.println("La voiture est une "+v2.getDescription());
    //affiche: La voiture est une P407 avec VitreElectrique,AirBag
    System.out.println("Son prix est:"+ v2.cost());
    //affiche: Son prix est 22200

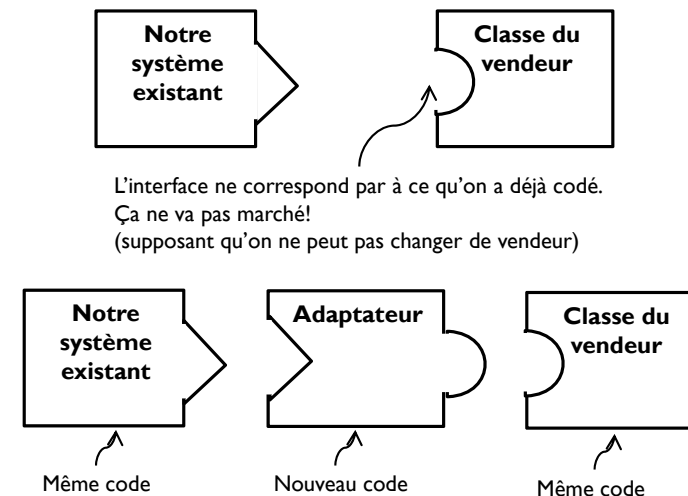
```

# Le patron "Adapter"

## Le patron adapter : Les adaptateurs (1/7)



## Le patron adapter : Les adaptateurs dans l'OO (2/7)



## Le patron adapter : Dinde et Canard (3/7)

- Supposant que le dinde marche et cancale comme le canard

Un canard peut cancaler et voler      Une simple implémentation du comportement du canard

```
interface Duck{
    void quack();
    void fly();
}

class MallardDuck implements Duck{
    public void quack(){
        System.out.println("Quack");
    }
    public void fly(){
        System.out.println("Fly");
    }
}

interface Turkey{
    void gobble();
    void fly();
}

class WildTurkey implements Turkey{
    public void gobble(){
        System.out.println("Gobble");
    }
    public void fly(){
        System.out.println("Fly for a short distance");
    }
}
```

Le dinde ne cancale pas, mais glougloute  
Le dinde peut voler (courte distance)

► 81

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Le patron adapter : L'adaptateur du dinde(4/7)

- Supposons qu'on a un manque de canards et on va utiliser des dindes à leur place → Il faut écrire un "adapter"

Respecter l'interface des canards

```
class TurkeyAdapter implements Duck{
    Turkey turkey;
    TurkeyAdapter(Turkey turkey){
        this.turkey=turkey;
    }

    public void quack(){
        turkey.gobble();
    }
    public void fly(){
        for(int i=0;i<5;i++){
            turkey.fly();
        }
    }
}
```

Une référence vers l'objet à adapter

Translation des méthodes

► 82

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Le patron adapter : Testons l'adaptateur(5/7)

```
public class TestAdapter{
    public static void main (String arg[])
    {
        MallardDuck mallard= new MallardDuck();
        WildTurkey wild = new WildTurkey();
        Duck turkeyAdapter = new TurkeyAdapter(wild);
        test(mallard);
        test(turkeyAdapter);
    }
    static void test(Duck duck)
    {
        duck.quack();
        duck.fly();
    }
}
```

- Donner le résultat d'exécution de cette classe

► 83

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

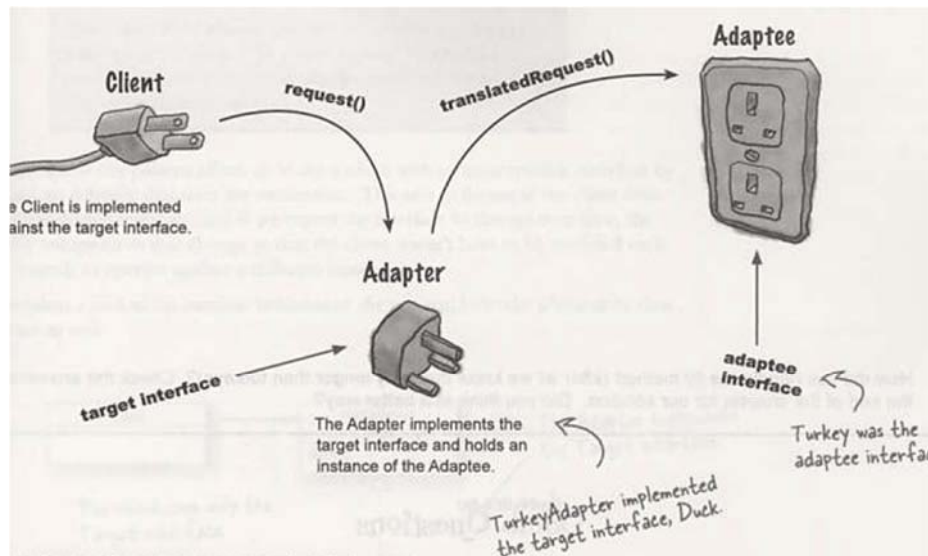
## Le patron adapter (6/7)

- Définition: **Adapter**
- Le **patron Adapter** convertit l'interface d'une classe à une autre interface que le client attend. Les adaptateurs permettent aux classes, aillant des interfaces incompatibles, de travailler ensemble.

► 84

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

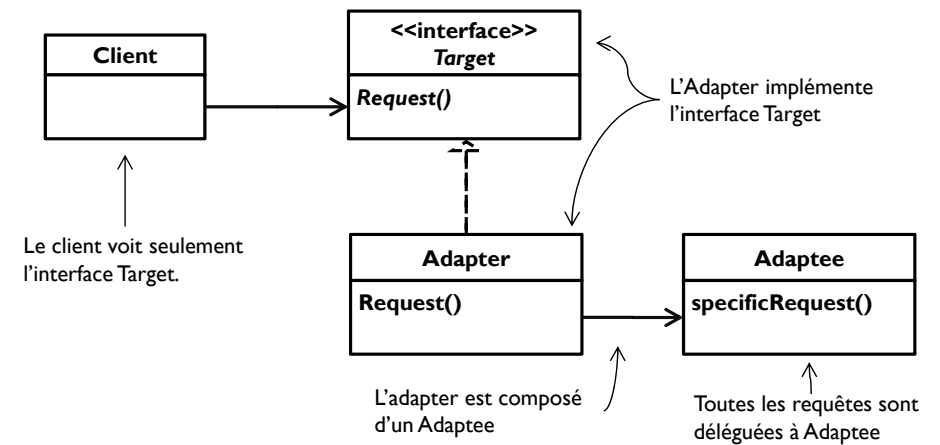
## Le patron adapter (6/7)



85

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Adapter : Le diagramme de classes du patron (7/7)



86

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Le patron "Façade"

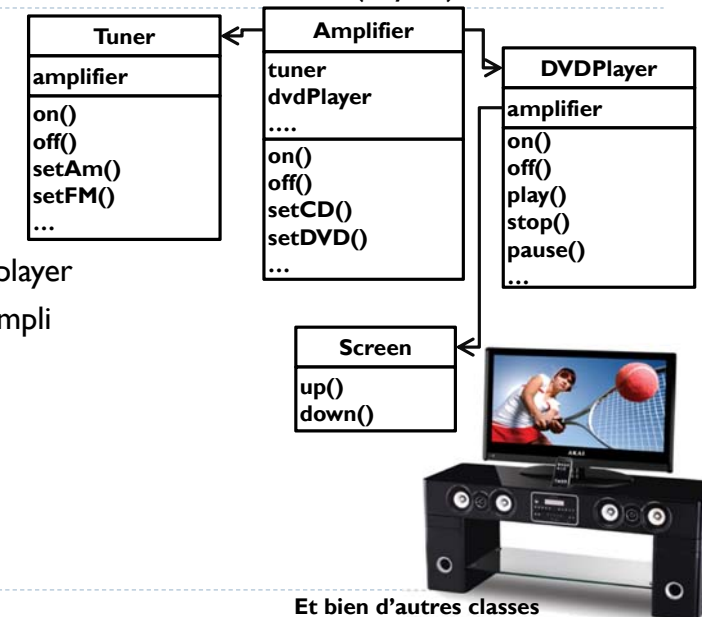
87

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Le patron Facade: Démarrer un home-cinéma (1/4)

► Démarrer le HC:

- ❶ baisser la lumière
- ❷ allumer l'écran
- ❸ démarrer l'ampli
- ❹ démarrer le DvDplayer
- ❺ le brancher avec l'mpli
- ❻ jouer le DvD
- ❼ etc..



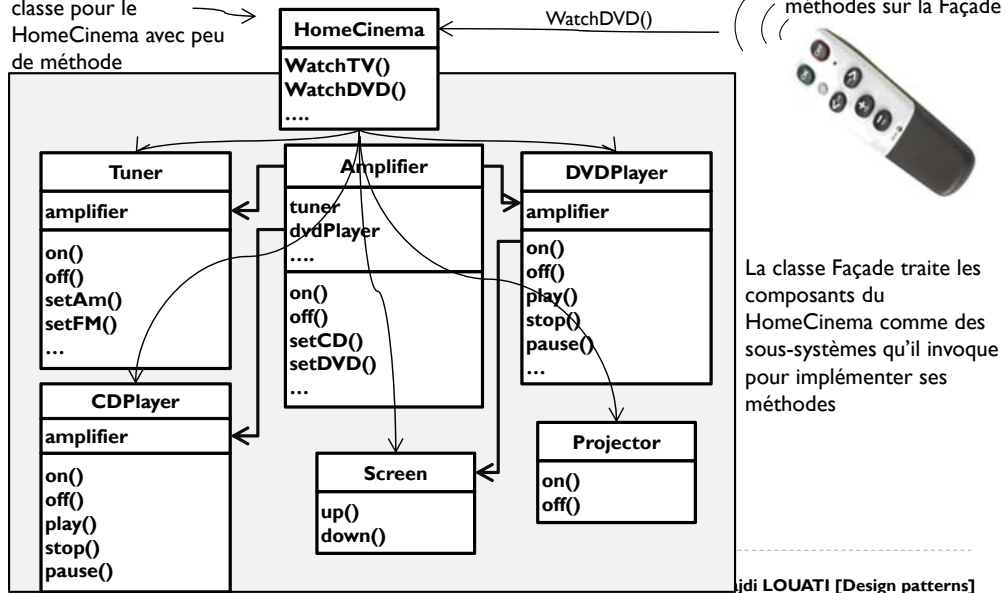
88

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]



## Le patron Facade: Les classes d'un Home cinéma (2/4)

La Façade: une nouvelle classe pour le HomeCinema avec peu de méthode



Wajdi LOUATI [Design patterns]

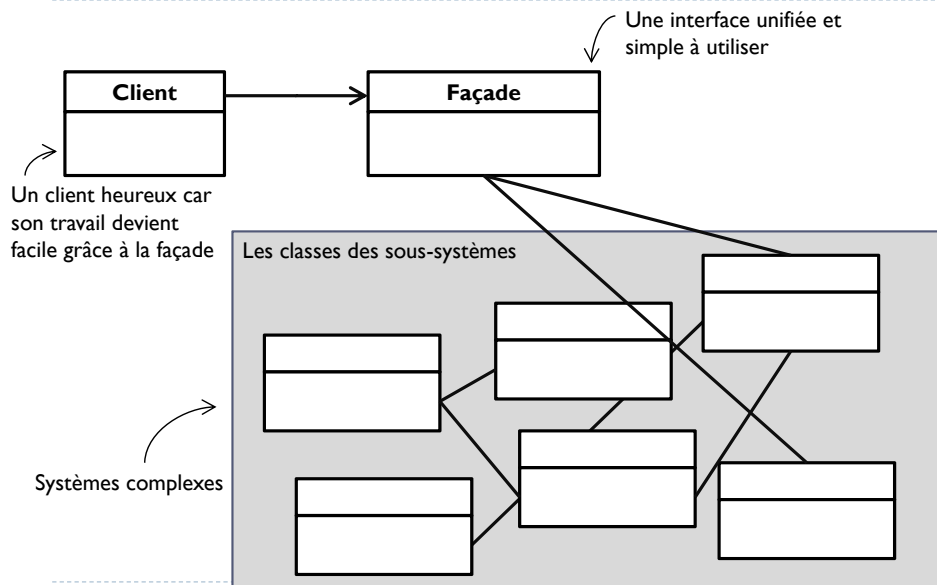
## Le patron façade (3/4)

- Définition: **Façade**
  - Le **patron Façade** présente une interface unifiée pour un ensemble de sous-interfaces dans un système. La façade définit une interface de haut niveau qui rend facile l'utilisation du système.

► 90

Riadh BEN HALIMA & Wajdi LOUATI [Design patterns]

## Façade: Le diagramme de classes du patron (4/4)



► 91

Riadh BEN HALIMA & Wajdi LOUATI [Design patterns]

## Testons nos connaissances!

| Patron    | Rôle                                                        |
|-----------|-------------------------------------------------------------|
| Decorator | Convertit une interface en une autre                        |
| Adapter   | Ne modifie pas l'interface, mais ajoute des responsabilités |
| Facade    | Rend les interfaces simples                                 |

► 92

Riadh BEN HALIMA & Wajdi LOUATI [Design patterns]

# Le patron "Singleton"

► 93

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Singleton : Spécification (1/12)

- Objectif: Créer un type d'objet pour lequel on crée seulement une seule instance
- C'est le patron ayant le diagramme de classes le plus simple
- Il y a plusieurs objets dont on a besoin d'une seule instance: pool d'impression, boîte de dialogue, objet qui manipule les préférences, objet de logging, objet agissant comme pilote de carte graphique/imprimante...
- La création de plus d'une instance de ces objets est une source de problème, telle que la sur-utilisation des ressources, des comportements incorrectes de programme, des résultats inconsistants, etc.

► 94

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Singleton : Créer un singleton (2/12)

- Comment créer un seul objet?
  - New MonObjet()
- Et si un autre objet veut créer un MonObjet? Est-ce qu'il peut appeler new sur MonObjet une autre fois?
  - Oui
- Pour toute classe, est ce qu'on peut l'instancier plus qu'une fois?
  - Oui (il faut que la classe soit publique)
- Que signifie ce code ?
  - C'est une classe qui ne peut pas être instanciée, car elle possède un constructeur privé
- Qui peut utiliser ce constructeur?
  - Le code de MonObjet est le seul code qui peut l'appeler (dans une méthode)

```
public class MonObjet{  
    private MonObjet() {}  
}
```

► 95

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Singleton : Créer un singleton (3/12)

- Comment je peux appeler cette méthode (pour créer une instance) si je n'ai pas d'instance?
  - static
- Que signifie ce code.

```
public class MonObjet{  
    public static MonObjet getInstance() {  
    }  
}
```
- C'est une méthode statique qui peut être appelée à partir du nom de la classe : **MonObjet.getInstance()**
- Si on met les choses ensemble, est ce qu'on peut instancier MonObjet?

```
public class MonObjet{  
    private MonObjet(){ }  
    public static MonObjet getInstance() {  
        return new MonObjet();  
    }  
}
```
- Comment faire pour créer une seule instance?

► 96

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Singleton : Implémentation du patron (4/12)

```
public class Singleton {  
    private static Singleton uniqueInstance;  
    private Singleton() {}  
    public static Singleton getInstance(){  
        if (uniqueInstance == null)  
            uniqueInstance = new Singleton();  
  
        return uniqueInstance;  
    }  
  
    public static void main(String args[]) {  
        Singleton s= Singleton.getInstance();  
    }  
}
```

Nous avons une variable statique pour stocker notre instance

Le constructeur est déclaré privé. Seulement la classe Singleton qui peut instancier cette classe

Cette méthode nous offre une manière pour instancier la classe Singleton

Si uniqueInstance n'est pas à nul, ça veut dire qu'elle a été créée précédemment

97

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Singleton : L'usine de chocolat (5/12)



```
public class ChocolateBoiler{  
    private boolean empty;  
    private boolean boiled;  
    public ChocolateBoiler() {  
        empty=true; boiled=false;  
    }  
    public void fill(){  
        if (empty){  
            //remplir la casserole avec du lait/chocolat  
            empty=false; boiled=false;  
        }  
    }  
    public void drain(){  
        if (!empty && boiled){  
            //vider la casserole  
            empty=true;  
        }  
    }  
    public void boil(){  
        if (!empty && !boiled){  
            //faire bouillir  
            boiled=true;  
        }  
    }  
}
```

Le code démarre lorsque la casserole est vide

Pour remplir la casserole, elle doit être vide. Lorsqu'elle est pleine, on met empty à false.

Pour vider la casserole, elle doit être pleine et déjà mixée. une fois vidée, on met empty à true.

Pour mixer le contenu de la casserole, elle doit être pleine et non déjà mixée. Lorsqu'elle est pleine, on met boiled à true.

98

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Singleton : L'usine de chocolat (6/12)



- Améliorer le code de l'usine de chocolat en le transformant en Singleton

```
public class ChocolateBoiler {  
    private boolean empty;  
    private boolean boiled;  
    private static ChocolateBoiler uniqueInstance;  
  
    private ChocolateBoiler() {  
        empty=true; boiled=false;  
    }  
  
    public static ChocolateBoiler getInstance() {  
        if (uniqueInstance == null)  
            uniqueInstance = new ChocolateBoiler();  
        return uniqueInstance;  
    }  
    //reste du code...  
}
```

99

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Singleton : Le patron Singleton(7/12)

- Définition: **Singleton**

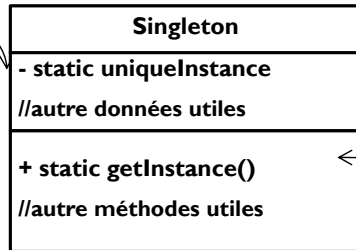
- Le **patron Singleton** assure une seule instance pour une classe, et offre un point d'accès global à cette classe.

100

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Singleton: Le diagramme de classes du patron (8/12)

La variable de classe `uniqueInstance` tient la seule instance du Singleton



La méthode `getInstance()` est statique. C'est une méthode de classe qu'on peut y accéder partout dans le code avec `Singleton.getInstance()`. Il s'agit d'une instantiation facile de cette classe

► 101

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Singleton : Problème des threads (9/12)



- Supposant que nous avons deux threads qui vont exécuter la méthode `getInstance()`. Est-ce qu'il y a un cas où on crée 2 instances?

| Thread-1                                                     | Thread-2                                                     | Valeur de <code>UniqueInstance</code> |
|--------------------------------------------------------------|--------------------------------------------------------------|---------------------------------------|
| <code>public static ChocolateBoiler<br/>getInstance()</code> |                                                              | null                                  |
|                                                              | <code>public static ChocolateBoiler<br/>getInstance()</code> | null                                  |
| <code>if (uniqueInstance == null)</code>                     |                                                              | null                                  |
|                                                              | <code>if (uniqueInstance == null)</code>                     | null                                  |
| <code>uniqueInstance = new<br/>ChocolateBoiler();</code>     |                                                              | Object1                               |
| <code>return uniqueInstance;</code>                          |                                                              | Object1                               |
|                                                              | <code>uniqueInstance = new<br/>ChocolateBoiler();</code>     | Object2                               |
|                                                              | <code>return uniqueInstance;</code>                          | Object2                               |

► 102

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Singleton : Gestion du multi-threading (10/12)

- Solution 1: synchroniser l'accès à la méthode `getInstance()`

```

public class Singleton {
    private static Singleton uniqueInstance;

    private Singleton() {}

    public static synchronized Singleton getInstance(){
        if (uniqueInstance == null)
            uniqueInstance = new Singleton();

        return uniqueInstance;
    }
    //autres méthodes utiles
}
  
```

Un seule thread peut accéder, à la fois, à cette méthode

- Inconvénient: **synchronized** réduit la performance d'un facteur de 100
  - Si la méthode `getInstance()` n'est pas critique pour notre application, on peut se contenter de cette solution

► 103

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Singleton : Gestion du multi-threading (11/12)

- Solution 2: création au moment de la définition de la variable de classe

```

public class Singleton {
    private static Singleton uniqueInstance = new Singleton();

    private Singleton() {}

    public static Singleton getInstance(){
        return uniqueInstance;
    }
    //autres méthodes utiles
}
  
```

Initialisation par le JVM avant accès des threads

Il y a déjà une instance, il faut juste la retourner

- La JVM crée une instance de Singleton lors du chargement de la classe. La JVM garantit que l'instance va être créée avant que les threads accèdent la variable statique **uniqueInstance**.

► 104

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Singleton : Gestion du multi-threading (12/12)

- ▶ Solution 3: réduire l'utilisation de la synchronisation dans getInstance()

```
public class Singleton {  
    private volatile static Singleton uniqueInstance;  
  
    private Singleton() {}  
  
    public static Singleton getInstance(){  
        if (uniqueInstance == null) {  
            synchronized(Singleton.class) {  
                if (uniqueInstance == null)  
                    uniqueInstance = new Singleton();  
            }  
        }  
        return uniqueInstance;  
    }  
    //autres méthodes utiles  
}
```

Le mot clé volatile assure que les threads gèrent la variable uniqueInstance correctement au moment de son initialisation

On synchronise seulement la première fois

\*volatile: inclus à java depuis jdk5

▶ 105

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Récapitulatif (1/2)

- ▶ Bases de l'OO: Abstraction, Encapsulation, Polymorphisme & Héritage
- ▶ Principes de l'OO
  - ▶ Encapsuler ce qui varie
  - ▶ Favoriser la composition sur l'héritage
  - ▶ Programmer avec des interfaces et non des implémentations
  - ▶ Opter pour une conception faiblement couplée
  - ▶ Les classes doivent être ouvertes pour les extensions et fermées pour les modifications
  - ▶ Dépendre des abstractions. Ne jamais dépendre de classes concrètes
- ▶ Patron de l'OO
  - ▶ Strategy: définit une famille d'algorithmes interchangeables
  - ▶ Observer: définit une dépendance 1-à-plusieurs entre objets.
  - ▶ decorator: attache des responsabilités additionnelles à un objet dynamiquement.
  - ▶ Abstract Factory: offre une interface de création de familles d'objets
  - ▶ Factory Method: définit une interface de création des objets
  - ▶ **Singleton**: assure à une classe une seule instance et lui offre un point d'accès global

▶ 106

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Récapitulatif (2/2)

- ▶ Le patron singleton assure la création d'au plus une instance d'une classe de notre application
- ▶ Le patron offre aussi un seul point d'accès global à cette instance
- ▶ L'implémentation Java du patron utilise un constructeur privé une méthode statique combinée avec une variable statique
- ▶ Le développeur examine la performance et les contraintes des ressources et choisit soigneusement une implémentation pour une application multi-thread

▶ 107

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

# Le patron "Command"

▶ 108

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

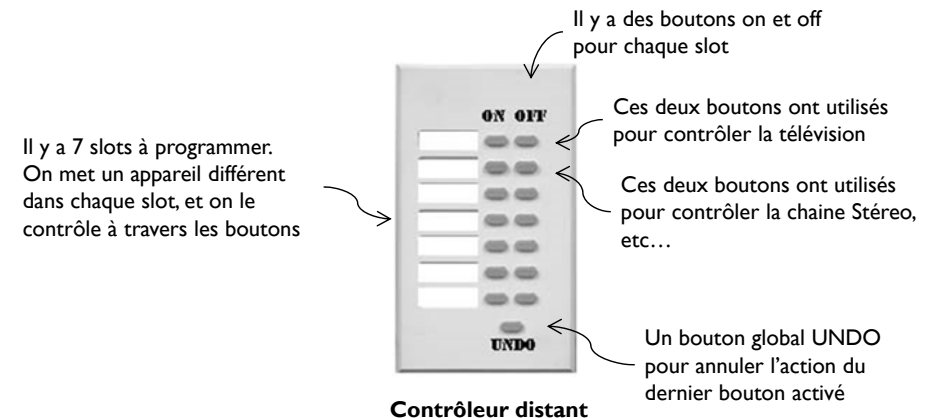
## Home-Automation : Spécification (1/28)

- ▶ Objectif: Mettre en œuvre un système de contrôle distant d'un ensemble d'appareils dans une maison
- ▶ Besoin: programmer les fonctionnalités d'un contrôleur distant (avec 7 slots) selon des classes (prédéfinies par le vendeur) de gestion des appareils installés dans la maison.
- ▶ Conception: OO
  - ▶ Prévoir les relations entre les boutons du contrôleur distant (ON-OFF) avec les fonctionnalités des appareils installés: `setTemperature()`, `setVolume()`, `setDirection()`, etc..

▶ 109

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Home-Automation : Analyse du contrôleur distant (2/28)

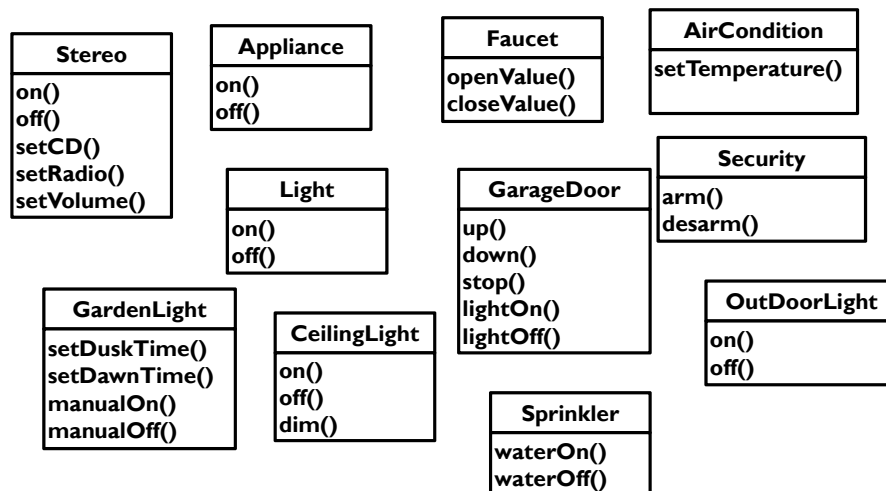


▶ 110

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Home-Automation : Les classes du vendeur (3/28)

- ▶ Les classes du vendeur nous donnent une idée sur les fonctionnalités des appareils installés dans la maison :



▶ 111

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Home-Automation : Discussion de la conception (4/28)

- ▶ On s'attendait à des classes avec des méthodes `on()`-`off()` pour bien correspondre avec le contrôleur distant
- ▶ C'est important de voir ça comme séparation des préoccupation : le contrôleur doit savoir comment interpréter l'appui sur le bouton et créer des requêtes, mais il ne doit pas connaître beaucoup sur les appareils et leurs manières de fonctionnement (comment allumer une lampe)
  - ▶ En d'autre terme, le contrôleur émet des requêtes génériques
  - ▶ Une entité prendra en charge la transformation de cette requête en action

▶ 112

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

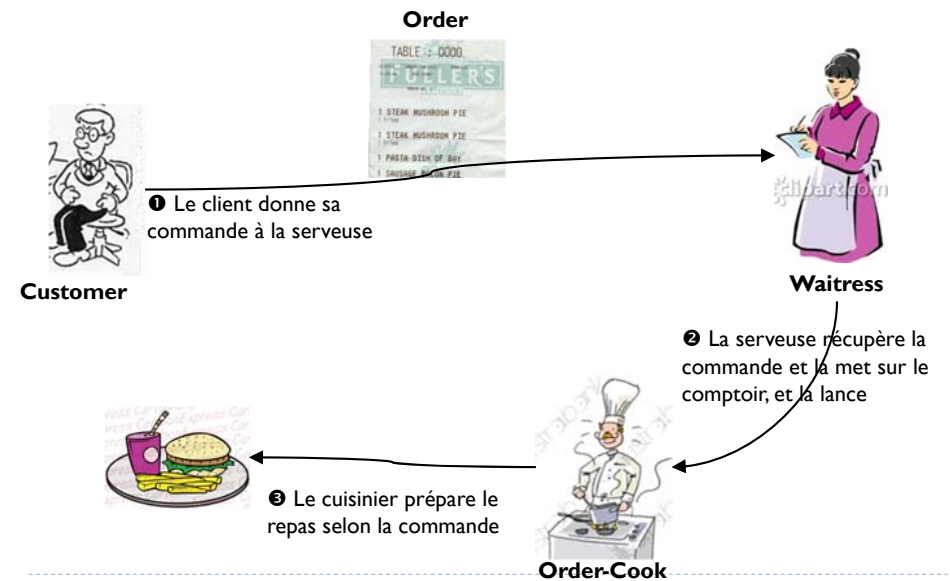
## Home-Automation : Discussion de la conception (4/28)

- ▶ Comment émettre des requêtes à des objets
  - ▶ sans rien connaître des opérations demandées ?
  - ▶ ou sans rien connaître de celui à qui la requête est destinée ?
- ▶ Dissocier (découpler) l'objet qui invoque une opération de l'objet qui possède les connaissances nécessaires pour réaliser cette opération.
  - ▶ En d'autre terme, le contrôleur émet des requêtes génériques
  - ▶ Une entité prendra en charge la transformation de cette requête en action
- ▶ Utiliser le patron **Command**
  - ▶ **Objet Command: Encapsuler une requête sous forme d'objet**

▶ 113

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

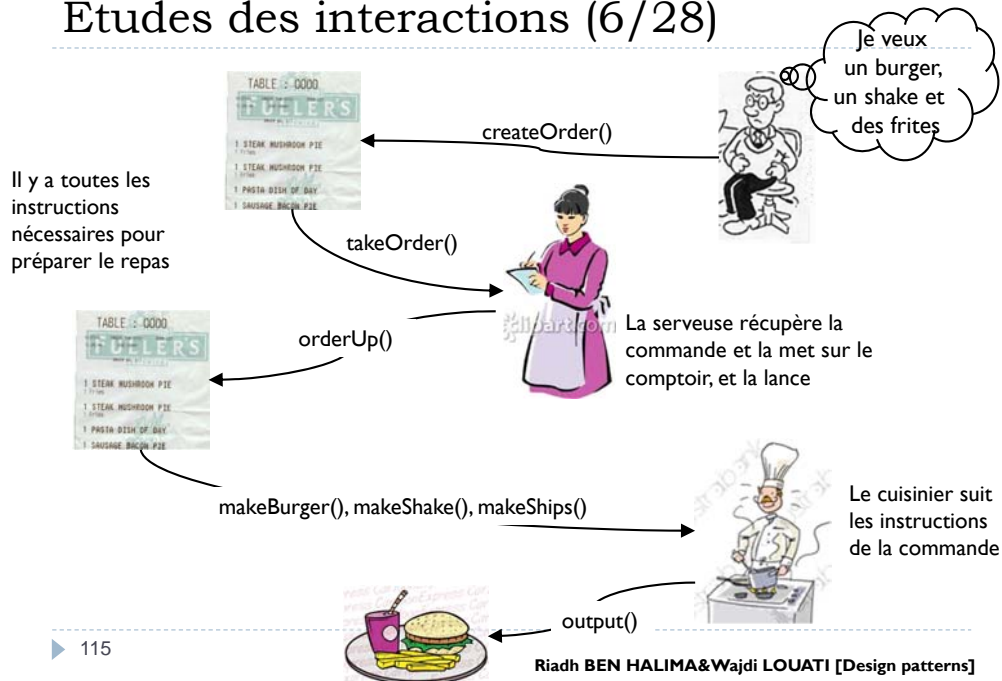
## Command : Commander un dîner (5/28)



▶ 114

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Command : Etudes des interactions (6/28)



▶ 115

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Command : Les rôles et les responsabilités (7/28)

- ▶ **La commande (papier) est une requête pour préparer le repas**
  - ▶ S'il s'agit d'un objet, il peut être passé de la serveuse au comptoir
  - ▶ Son interface consiste à une seule méthode `orderUp()`, qui encapsule les actions nécessaires pour préparer le repas
  - ▶ La serveuse n'a besoin de savoir comment préparer le repas!
- ▶ **La tâche de la serveuse est de prendre la commande et d'invoquer la méthode `orderUp()` dessus**
  - ▶ La méthode `takeOrder()` de la serveuse peut être paramétrée avec différentes commandes de plusieurs clients. Ceci ne la dérange pas car elle sait que `orderUp()` supporte sa commande
- ▶ **Le cuisinier possède les connaissances nécessaires pour préparer le repas**
  - ▶ Suite à l'invocation de `orderUp()`, le cuisinier implémente toutes les méthodes nécessaires pour créer le repas
  - ▶ Noter qu'il est complètement découplé de la serveuse
    - ▶ La serveuse encapsule les détails du repas dans la commande
    - ▶ Le cuisinier prend ses instructions de la commande, et il n'a pas besoin de la contacter

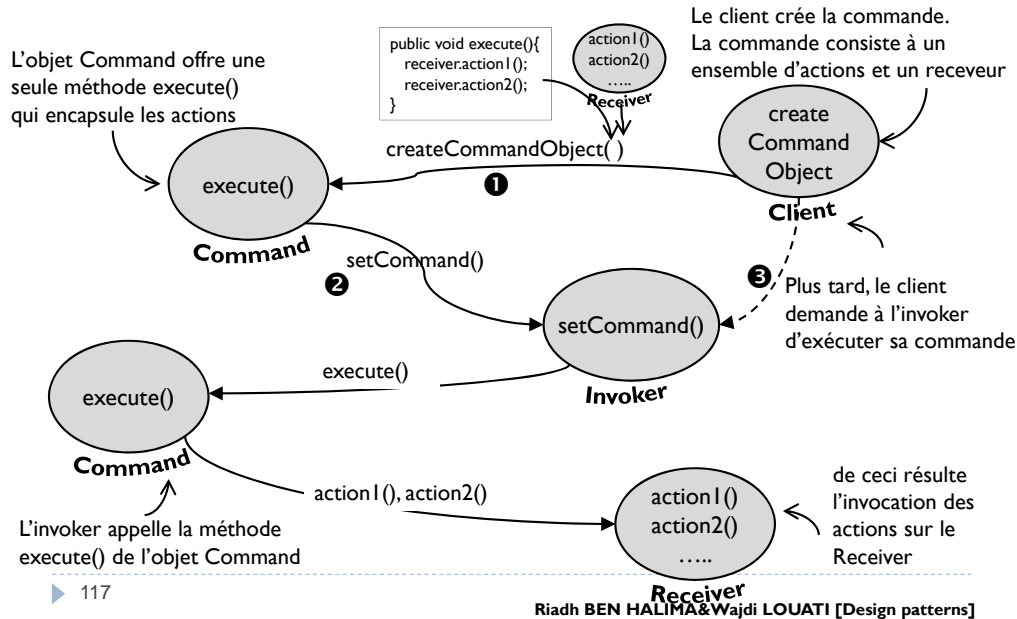
▶ 116

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]



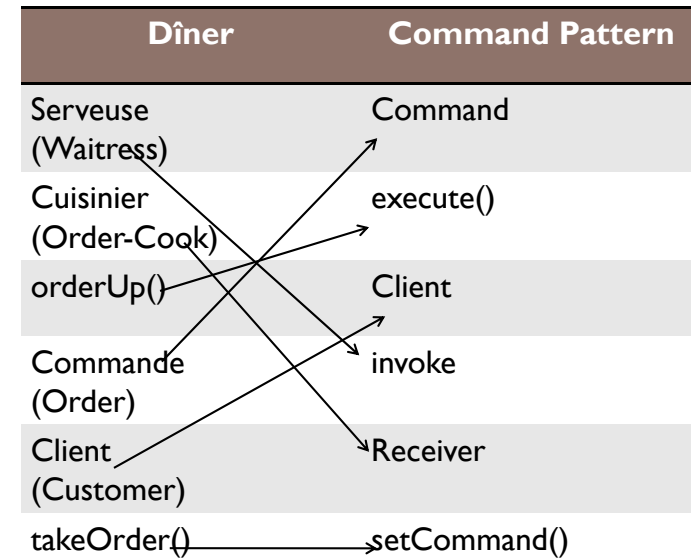
## Command :

### Du dîner vers le patron Commande (8/28)



## Command :

### Correspondance (9/28)



118

Riadh BEN HALIMA & Wajdi LOUATI [Design patterns]

## Command :

### Le patron Command (10/28)

- Définition: **Command**
- Le **patron Command** encapsule une requête comme un objet, ainsi il nous permet de paramétrer d'autres objets avec différentes requêtes, files d'attente ou longues requêtes, et supporte l'annulation d'une opération

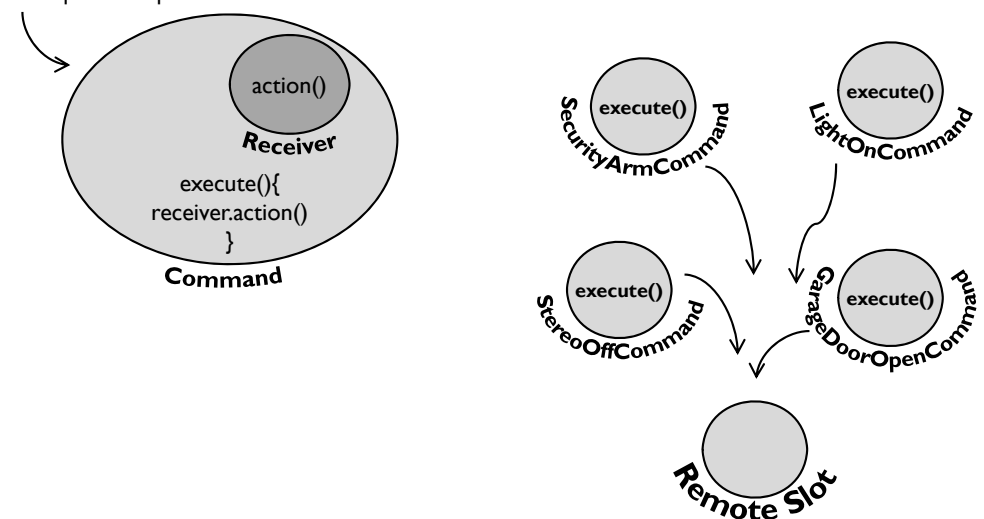
119

Riadh BEN HALIMA & Wajdi LOUATI [Design patterns]

## Home-Automation :

### Des Commandes (11/28)

Une requête encapsulée

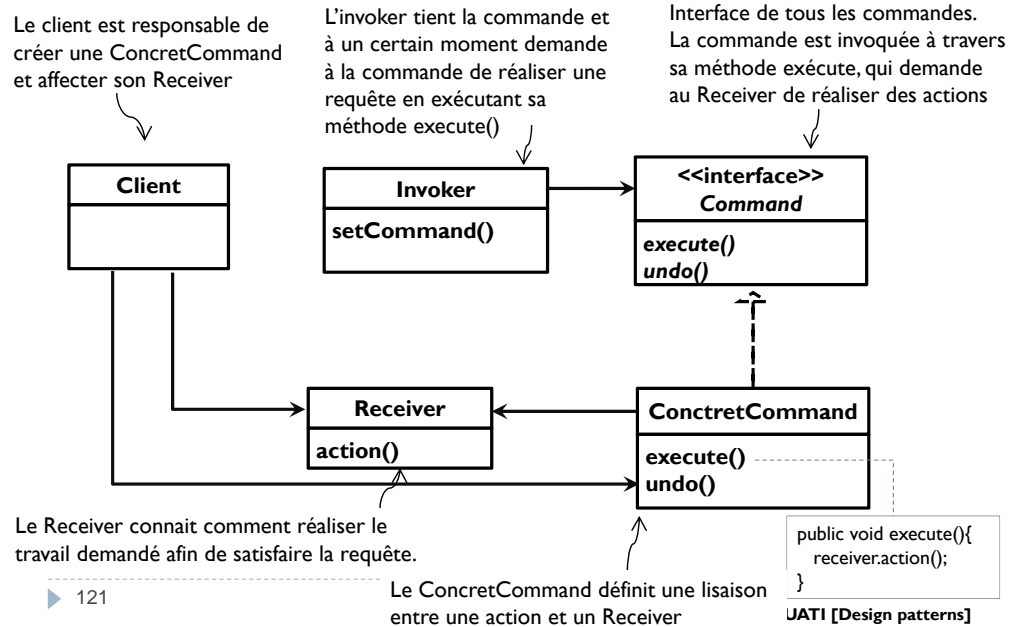


120

Riadh BEN HALIMA & Wajdi LOUATI [Design patterns]

## Command :

### Le diagramme de classes du patron (12/28)



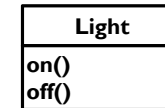
## Home-Automation :

### Notre premier objet Commande (13/28)

#### ► Implémentons l'interface Command

```
public interface Command{
    public void execute();
}
```

#### ► Implémentons une Commande pour allumer la lumière



```
public class LightOnCommand implements Command {
    Light light;
    public LightOnCommand(Light light){
        this.light = light;
    }
    public void execute(){
        light.on();
    }
}
```

► 122

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Home-Automation :

### Notre premier objet Commande (14/28)

#### ► Utilisons l'objet Commande

```
public class SimpleRemoteControl{
    Command slot;

    public SimpleRemoteControl() {}

    public void setCommand(Command command ){
        slot=command;
    }

    public void buttonWasPressed(){
        slot.execute();
    }
}
```

► 123

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Home-Automation :

### Notre premier objet Commande (15/28)

#### ► Testons la fonctionnalité du contrôleur distant

```
public class RemoteControlTest{
    public static void main(String argv[]) {
        SimpleRemoteControl remote = new SimpleRemoteControl();
        Light light=new Light();
        LightOnCommand lightOn=new LightOnCommand (light);

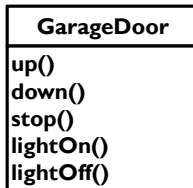
        remote.setCommand(lightOn); //passer la commande à l'invoker
        remote.buttonWasPressed(); //simuler l'appui sur le bouton
    }
}
```

► 124

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Home-Automation : 2<sup>ème</sup> Commande (16/28)

- ▶ Développer la classe GarageDoorOpenCommand



```
public class GarageDoorOpenCommand implements Command {
    GarageDoor garageDoor;
    public GarageDoorOpenCommand (GarageDoor garageDoor){
        this.garageDoor = garageDoor;
    }
    public void execute(){
        garageDoor.up();
        garageDoor.lightOn();
    }
}
```

▶ 125

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Home-Automation : Une autre Commande (17/28)

- ▶ Ajoutant cette commande au slot du contrôleur distant

```
public class RemoteControlTest{
    public static void main(String argv[]) {
        SimpleRemoteControl remote = new SimpleRemoteControl();
        Light light = new Light();
        LightOnCommand lightOn=new LightOnCommand(light);

        GarageDoor garageDoor = new GarageDoor();
        GarageDoorOpenCommand garageOpen = new
            GarageDoorOpenCommand(garageDoor);

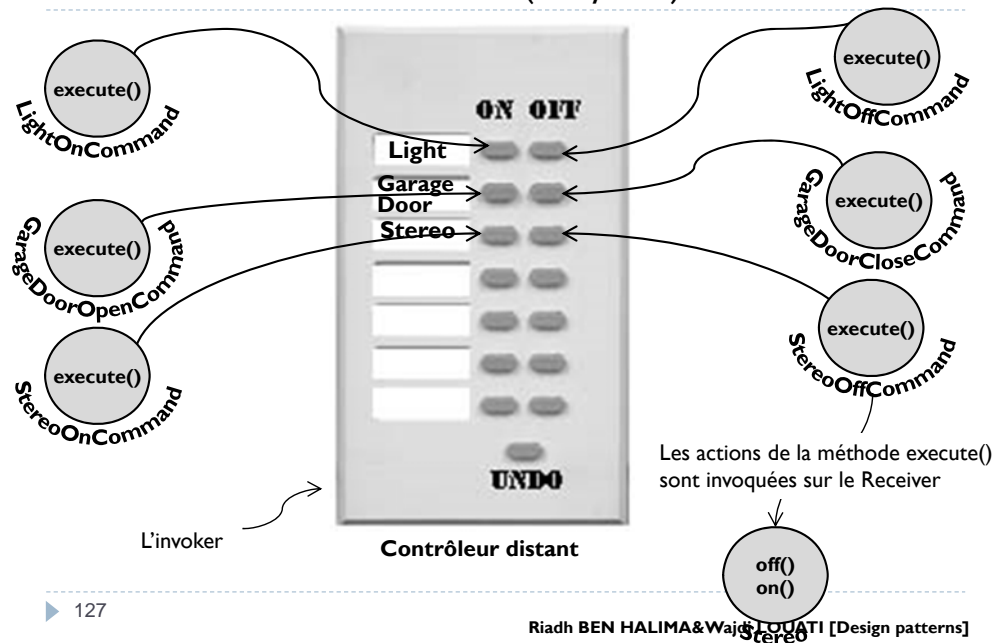
        remote.setCommand(lightOn); //encapsuler la commande
        remote.buttonWasPressed(); //allumer la lumière

        remote.setCommand(garageOpen); //encapsuler la commande
        remote.buttonWasPressed(); //ouvrir la porte du garage
    }
}
```

▶ 126

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

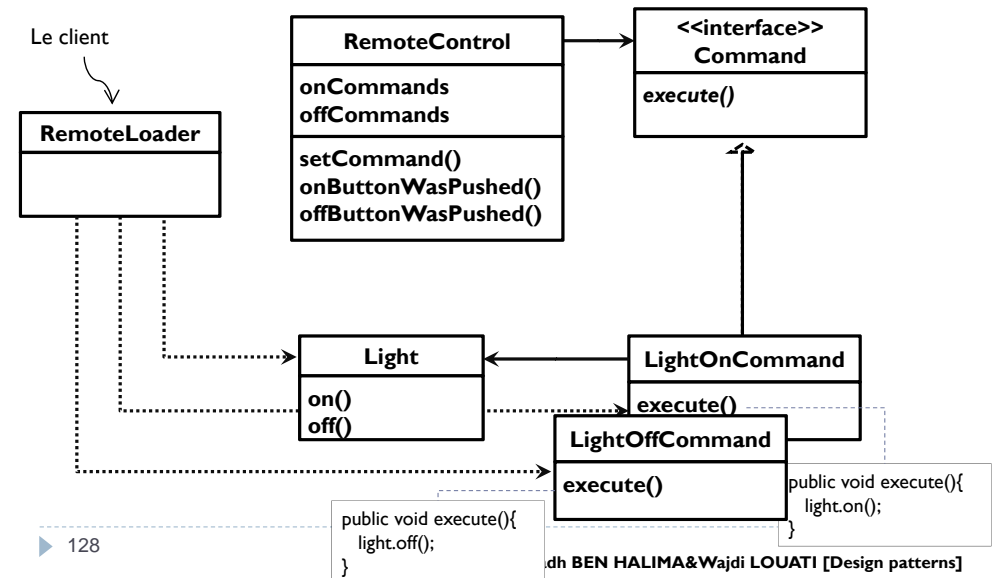
## Home-Automation : Le contrôleur distant (18/28)



▶ 127

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Home-Automation : Le diagramme de classes (19/28)



▶ 128

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Home-Automation : Programmer le contrôleur distant (20/28)

```
class RemoteControl{
    Command onCommands[];
    Command offCommands[];
    public RemoteControl() {
        onCommands = new Command[7];
        offCommands = new Command[7];
        Command noCommand = new NoCommand(); ← Eviter la gestion de null
        for(int i=0;i<7;i++){
            onCommands[i] = noCommand;
            offCommands[i] = noCommand;
        }
    }
    public void setCommand(int slot, Command onCommand,Command offCommand ){
        onCommands[slot] = onCommand;
        offCommands[slot] = offCommand;
    }
    public void onButtonWasPressed(int slot){
        onCommands[slot].execute();
    }
    public void offButtonWasPressed(int slot){
        offCommands[slot].execute();
    }
    public String toString(){
        String s="";
        for(int i=0;i<7;i++){
            s+="Slot["+i+"] "+onCommands[i].getClass().getName() +"
"+offCommands[i].getClass().getName()+"\n";
        }
        return s;
    }
}
```

129

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Home-Automation : Programmer les commandes (21/28)

- ▶ Programmer les classes suivantes:
  - ▶ Light.java, Stereo.java
  - ▶ LightOnCommand.java, LightOffCommand.java, StereoOnWithCDCommand.java, StereoOffCommand.java

```
class LightOnCommand
implements Command {
    Light light;
    public LightOnCommand(Light
light){
        this.light = light;
    }
    public void execute(){
        light.on();
    }
}
```

```
class StereoOnWithCDCommand
implements Command {
    Stereo stereo;
    public StereoOnWithCDCommand(Stereo
stereo){
        this.stereo = stereo;
    }
    public void execute(){
        stereo.on();
        stereo.setCD();
        stereo.setVolume(11);
    }
}
```

▶ 130

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Home-Automation : Tester le contrôleur (22/28)

```
class RemoteLoader{
    public static void main(String arg[]){
        RemoteControl remoteControl = new RemoteControl();
        Light livingRoomLight=new Light();
        LightOnCommand livingRoomLightOnCommand = new
            LightOnCommand(livingRoomLight);
        LightOffCommand livingRoomLightOffCommand = new
            LightOffCommand(livingRoomLight);
        remoteControl.setCommand(0, livingRoomLightOnCommand,livingRoomLightOffCommand);
        remoteControl.onButtonWasPressed(0);
        remoteControl.offButtonWasPressed(0);
        Stereo stereo = new Stereo();
        StereoOnWithCDCommand stereoOnWithCDCommand = new
            StereoOnWithCDCommand(stereo);
        StereoOffCommand stereoOffCommand = new StereoOffCommand(stereo);
        remoteControl.setCommand(1, stereoOnWithCDCommand, stereoOffCommand);
        remoteControl.onButtonWasPressed(1);
        remoteControl.offButtonWasPressed(1);
        System.out.println(remoteControl.toString());
    }
}
```

▶ 131

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

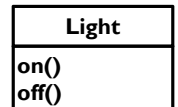
## Home-Automation : Undo: Annuler la dernière opération (23/28)

- ▶ Implémentons l'interface Command
- ▶ Implémentons une Commande pour allumer la lumière

```
public interface Command{
    public void execute();
    public void undo();
}
```

```
public class LightOnCommand implements Command {
    Light light;
    public LightOnCommand(Light light){
        this.light = light;
    }
    public void execute(){
        light.on();
    }
    public void undo(){
        light.off();
    }
}
```

Opération à exécuter en cas  
d'annulation de cette commande



▶ 132

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Home-Automation : Undo: Annuler la dernière opération (24/28)

```
class RemoteControl{
    Command onCommands[];
    Command offCommands[];
    Command undoCommand;
    public RemoteControl() {
        onCommands = new Command[7];
        offCommands = new Command[7];
        Command noCommand = new NoCommand();
        for(int i=0;i<7;i++){
            onCommands[i] = noCommand;
            offCommands[i] = noCommand;
        }
        undoCommand = noCommand; }
    public void onButtonWasPressed(int slot){
        onCommands[slot].execute();
        undoCommand = onCommands[slot];
    }
    public void offButtonWasPressed(int slot){
        offCommands[slot].execute();
        undoCommand = offCommands[slot];
    }
    public void undoButtonWasPressed(){
        undoCommand.undo();
    }
}
```

C'est ou on stockera la dernière commande exécutée pour le bouton undo

Initialisation à NoCommand afin d'éviter le traitement de null

Enregistrer la dernière commande

Lorsqu'on appuie sur le bouton undo, on invoque la méthode undo() pour annuler la dernière commande exécutée

133

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Home-Automation : Tester le contrôleur avec UNDO (25/28)

```
class RemoteLoader{
    public static void main(String arg[]){
        //...
        remoteControl.onButtonWasPressed(0);
        remoteControl.undoButtonWasPressed();
        //...

        remoteControl.onButtonWasPressed(1);
        remoteControl.offButtonWasPressed(1);
        System.out.println(remoteControl.toString());
    }
}
```

Annuler l'allumage de la lumière

134

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Home-Automation : La Macro-Commande (26/28)

► C'est le regroupement de plusieurs commandes en une seule

```
public class MacroCommand implements Command {
    Command [] commands;
    public MacroCommand (Command [] commands){
        this.commands = commands;
    }
    public void execute(){
        for (int i=0;i<commands.length;i++)
            commands[i].execute();
    }
}
```

Stocker les commandes de la macro-commande

Un boucle pour exécuter toutes les commandes de la macro-commande

❶ Créer les commandes à mettre dans la macro-commande

```
LightOnCommand lightOnCommand = new LightOnCommand(light);
StereoOnWithCDCommand stereoOnWithCDCommand = new
    StereoOnWithCDCommand(stereo);
TVOnCommand tvOnCommand = new TVOnCommand(tv);

//créer aussi les Off-Commandes
```

Créer les commandes

135

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Home-Automation : MacroCommand (27/28)

❷ Créer les macro-commandes

```
Command [] on = {lightOnCommand, stereoOnWithCDCommand, tvOnCommand};
Command [] off = {lightOffCommand, stereoOffWithCDCommand, tvOffCommand};
```

```
MacroCommand onMacro= new MacroCommand(on);
MacroCommand offMacro= new MacroCommand(off);
```

Les commandes sous formes de tableau

❸ Affecter les macro-commandes à un bouton

```
remoteControl.setCommand(2, onMacro, offMacro);
```

❹ Exécuter la macro-commande

```
System.out.println("Macro On");
remoteControl.onButtonWasPushed(2);
System.out.println("Macro Off");
remoteControl.offButtonWasPushed(2);
```

Affecter les macro-commandes au bouton du slot n°2

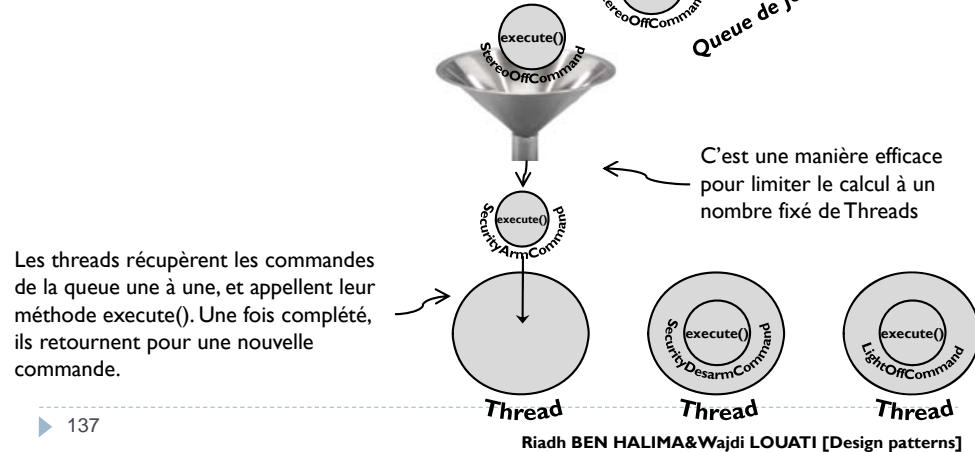
Tester ces macro-commandes et donner le résultat de l'exécution

136

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Autre utilisation du patron Command : Organiser les queues de requêtes (28/28)

- Les commandes nous offrent une manière de paquetage les morceaux de calcul (computation). Les calculs (commandes créées par des applications) seront placés dans des queues (queue de jobs) pour être exécutés.



## Récapitulatif (1/2)

- Bases de l'OO: Abstraction, Encapsulation, Polymorphisme & Héritage
- Principes de l'OO
  - Encapsuler ce qui varie
  - Favoriser la composition sur l'héritage
  - Programmer pour des interfaces
  - Opter pour une conception faiblement couplée
  - Les classes doivent être ouvertes pour les extensions et fermées pour les modifications
  - Dépendre des abstractions. Ne jamais dépendre de classes concrètes
- Patron de l'OO
  - Strategy: définit une famille d'algorithmes interchangeables
  - Observer: définit une dépendance 1-à-plusieurs entre objets.
  - Decorator: attache des responsabilités additionnelles à un objet dynamiquement.
  - Abstract Factory: offre une interface de création de familles d'objets
  - Factory Method: définit une interface de création des objets
  - Singleton: assure à une classe une seule instance
  - Command**: encapsule une requête comme un objet

138

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Récapitulatif (2/2)

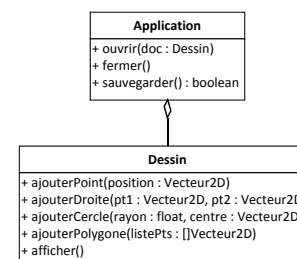
- Le patron Command découple un objet, faisant des requêtes, de l'objet qui sait comment la réaliser
- Un objet Command est au centre de ce découplage et encapsule le Receiver avec une action (ou des actions)
- L'Invoker exécute la requête d'un objet Command en appelant sa méthode execute(), qui invoque les actions sur le Receiver
- Le patron Command supporte l'annulation (undo) par l'implémentation d'une méthode undo() -dans la commande- qui restore l'ancien état du système (avant l'exécution de la méthode execute())
- Les Macro-Commandes sont des simples extensions de Command qui permettent l'invocation de multiple commandes. Pareil, les macro-commandes peuvent supporter les méthodes d'annulation (undo).
- Le patron Command est utilisé aussi pour implémenter l'organisation des requêtes (Jobs) et la gestion de la journalisation (logging)

139

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Exercice 1

- On vous demande de participer à la création d'un nouvel outil graphique. Cet outil permettra de créer de manière intuitive des dessins avec un niveau de complexité plus ou moins élevé. Les dessins pourront être composés d'un ensemble de points, droites, arc de cercles ou autres formes simples telles que des cercles ou des polygones. Cet outil sera similaire au programme appelé «Paint» sous l'environnement Windows. La figure suivante présente un diagramme de classes simplifié pour cette application :



- Vous êtes chargé de concevoir le mécanisme qui permettra de garder une trace des actions de l'utilisateur. Ce dernier pourra ainsi annuler les dernières actions faites.
- Question: Faites les modifications nécessaires au diagramme de classes pour implanter le patron Commande.

140

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]



## Exercice 2

- ▶ Le but de cet exercice est de tester la puissance du pattern Command. Pour ce faire, nous disposons d'une calculatrice offrant les opérations arithmétiques de base (+, - et \*) sur 2 réels et nous voulons transformer les actions (des utilisateurs) de calculs sur cette calculatrice en des commandes.
- ▶ 1. Donne le diagramme de classes décrivant cette transformation avec le pattern Command.
- ▶ 2. Implanter ce diagramme tout en respectant le client suivant :

```
public class Client {  
    public static void main(String[] args) {  
        Calculatrice c=new Calculatrice();  
        PlusCommand plus =new PlusCommand(c);  
        MultipCommand mult=new MultipCommand (c);  
        SoustCommand sous =new SoustCommand(c);  
        CalculatriceControl control =new CalculatriceControl();  
        control.setCommand(0, plus);  
        control.setCommand(1, sous);  
        control.setCommand(2, mult);  
        control.MultiButtonPressed(2, 5, 15);  
        control.SoustButtonPressed(1, 17, 10);  
        control.PlusButtonPressed(0, 12, 15);  
    }  
}
```

▶ 141

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

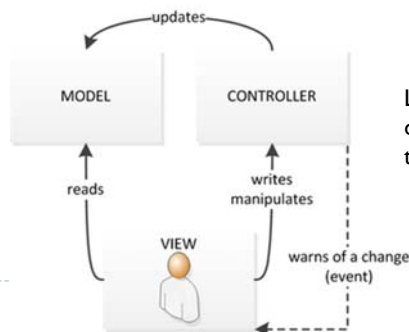
# Le Modèle-Vue-Contrôleur

▶ 142

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Architecture Modèle/Vue/Contrôleur

- ▶ Le modèle MVC: destiné à répondre aux besoins des applications interactives en séparant les problématiques liées aux différents composants au sein de leur architecture respective.
- ▶ Ce paradigme regroupe les fonctions en trois catégories :
  - ▶ un modèle (modèle de données),
  - ▶ une vue (présentation, interface utilisateur)
  - ▶ un contrôleur (logique de contrôle, gestion des événements, synchronisation)



L'idée est de bien séparer les données, la présentation et les traitements.

IA&Wajdi LOUATI [Design patterns]

▶ 143

## Le Modèle

- ▶ Le modèle représente le cœur (algorithmique) de l'application : traitements des données, interactions avec la base de données, etc.
  - ▶ décrit les données manipulées par l'application.
  - ▶ regroupe la gestion de ces données et est responsable de leur intégrité.
  - ▶ La base de données sera l'un de ses composants.
- ▶ Le modèle comporte des méthodes standards pour mettre à jour ces données (insertion, suppression, changement de valeur).
- ▶ Les résultats renvoyés par le modèle ne s'occupent pas de la présentation.
- ▶ Le modèle ne contient aucun lien direct vers le contrôleur ou la vue.
  - ▶ Sa communication avec la vue **s'effectue au travers du patron Observateur.**

▶ 144

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]



## La Vue

- ▶ Ce avec quoi l'utilisateur interagit se nomme précisément la *vue*.
  - ▶ présenter les résultats renvoyés par le modèle.
  - ▶ recevoir toute action de l'utilisateur (*hover*, clic de souris, sélection d'un bouton radio, cochage d'une case, entrée de texte, de mouvements, de voix, etc.).
- ▶ Ces différents événements sont envoyés au contrôleur.
- ▶ La vue n'effectue pas de traitement,
  - ▶ afficher les résultats des traitements effectués par le modèle et interagir avec l'utilisateur.
  - ▶ Plusieurs vues peuvent afficher des informations partielles ou non d'un même modèle.

▶ 145

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Le Contrôleur

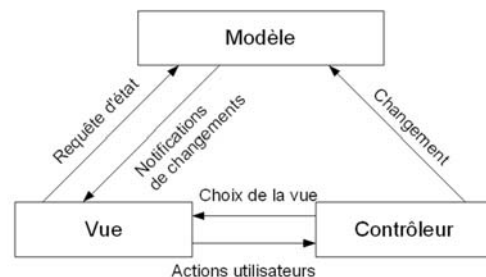
- ▶ Le contrôleur prend en charge la gestion des événements de synchronisation pour mettre à jour la vue ou le modèle
- ▶ reçoit tous les événements de l'utilisateur et enclenche les actions à effectuer
- ▶ Si une action nécessite un changement des données, le contrôleur demande la modification des données au modèle, et ce dernier notifie la vue que les données ont changé pour qu'elle se mette à jour.
- ▶ **D'après le patron de conception observateur/observable, la vue est un « observateur » du modèle qui est lui « observable »**
- ▶ Le contrôleur n'effectue aucun traitement, ne modifie aucune donnée.
  - ▶ Il analyse la requête du client et se contente d'appeler le modèle adéquat et de renvoyer la vue correspondant à la demande.

▶ 146

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Flux de traitement

- ▶ lorsqu'un client envoie une requête à l'application :
  - ▶ la requête envoyée depuis la vue est analysée par le contrôleur (par exemple un clic de souris pour lancer un traitement de données) ;
  - ▶ le contrôleur demande au modèle approprié d'effectuer les traitements et notifie à la vue que la requête est traitée (*via par exemple un callback*) ;
  - ▶ la vue notifiée fait une requête au modèle pour se mettre à jour (par exemple affiche le résultat du traitement *via le modèle*).



▶ 147

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## Architecture MVC ou 3-Tier

- ▶ L'architecture trois tiers est un modèle en couches, c'est-à-dire, que chaque couche communique seulement avec ses couches adjacentes
  - ▶ le flux de contrôle traverse le système de haut en bas
- ▶ Dans le modèle MVC, la vue peut consulter directement le modèle (lecture) sans passer par le contrôleur. Par contre, elle doit nécessairement passer par le contrôleur pour effectuer une modification (écriture).
  - ▶ le contrôleur peut alors envoyer des requêtes à toutes les vues de manière à ce qu'elles se mettent à jour.
- ▶ La différence fondamentale se trouve dans le fait que l'architecture 3-Tier sépare la *couche métier* de la *couche accès aux données*.
- ▶ Pour qu'une application MVC soit une vraie application 3-Tier il faut lui ajouter une couche d'abstraction d'accès aux données de type DAO (Data Access Object).
- ▶ Inversement pour qu'une application 3-Tier respecte MVC il faut lui ajouter une couche de contrôle entre la couche métier et la couche présentation.

▶ 148

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

# Le patron "Factory"

► 149

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## PizzaStore : Créer des pizzas (1/37)



### ► Pizzeria: Créer des pizzas

```
Pizza orderPizza(){  
    Pizza pizza=new Pizza();  
  
    pizza.prepare();  
    pizza.bake();  
    pizza.cut();  
    pizza.box();  
    return pizza;  
}
```

Pour la flexibilité, on veut que celui-ci soit une classe abstraite ou interface, sauf qu'on ne peut pas instancier l'un des deux derniers!

### ► On veut plus qu'un type de pizza....

► 150

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## PizzaStore : Plusieurs types de pizza (2/37)

### ► Pizzeria: Créer plusieurs types de pizza

```
Pizza orderPizza( String type ) {  
    Pizza pizza;
```

On passe le type du pizza à travers la méthode orderPizza()

```
    if (type.equals("cheese")){  
        pizza=new CheesePizza();  
    } else if (type.equals("greek")){  
        pizza=new GreekPizza();  
    } else if (type.equals("pepperoni")){  
        pizza=new PepperoniPizza();  
    }
```

Selon le type de pizza, on crée la pizza concrète, et on la place dans la variable "pizza" (interface et classe mère des pizzas).

```
    pizza.prepare();  
    pizza.bake();  
    pizza.cut();  
    pizza.box();  
    return pizza;  
}
```

Une fois on a la pizza, on prépare la sauce, le nappage (tomate/crème fraîche) et le fromage, puis on la fait cuire, la coupe, et on la met dans une boîte.

► 151

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## PizzaStore : Autres types de pizza (3/37)



- Les concurrents ont ajouté un nouveau type de pizza: (avec Calamars)
  - Ajouter ClamPizza au menu
- On n'a pas vendu beaucoup de GreekPizza dernièrement
  - Suspendre GreekPizza du menu

Ce code est NON fermé pour la modification!

```
Pizza orderPizza(String type){  
    Pizza pizza;  
    if (type.equals("cheese")){  
        pizza=new CheesePizza();  
    } else if (type.equals("greek")){  
        pizza=new GreekPizza();  
    } else if (type.equals("pepperoni")){  
        pizza=new PepperoniPizza();  
    } else if (type.equals("clam")){  
        pizza=new ClamPizza();  
    }  
    pizza.prepare();  
    pizza.bake();  
    pizza.cut();  
    pizza.box();  
    return pizza;  
}
```

Partie variable: On modifie le code autant que la sélection de pizza change.

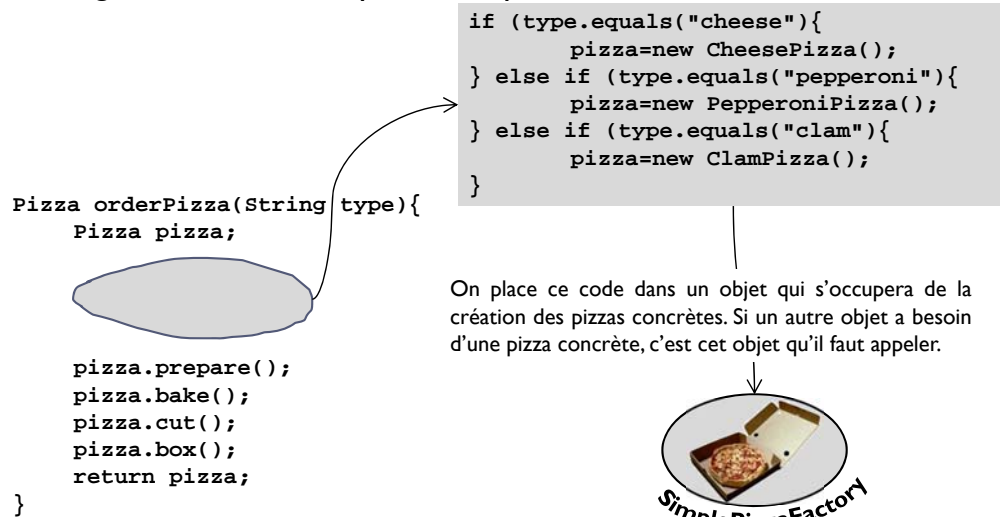
Partie invariable: Généralement, ces opérations sont les mêmes pour des années et des années.

► 152

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## PizzaStore : Encapsuler la création (4/37)

- Règle I de l'OO: Encapsuler ce qui varie

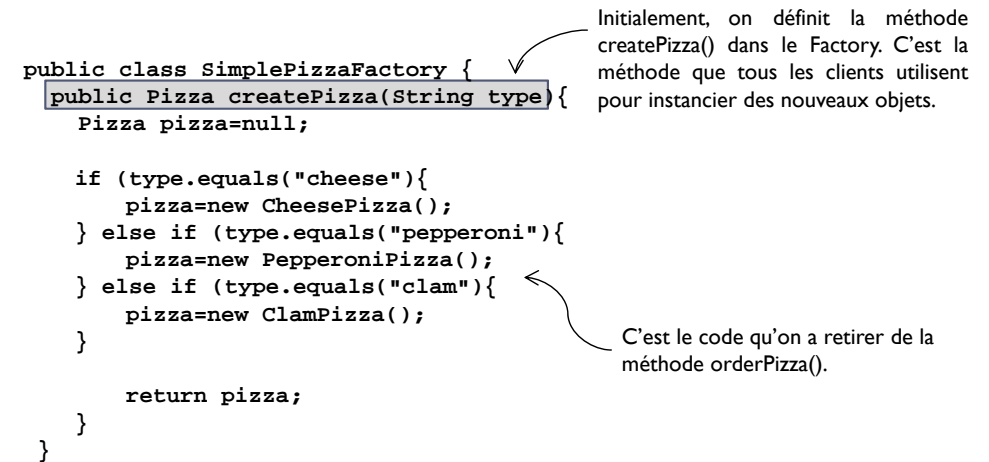


► 153

On attribue le nom Factory à ce nouvel objet  
Riadh BEN HALIMA & Wajdi LOUATI [Design patterns]

## PizzaStore : Un Simple Factory (5/37)

- Le seul rôle de SimplePizzaFactory est de créer des pizzas pour ces clients

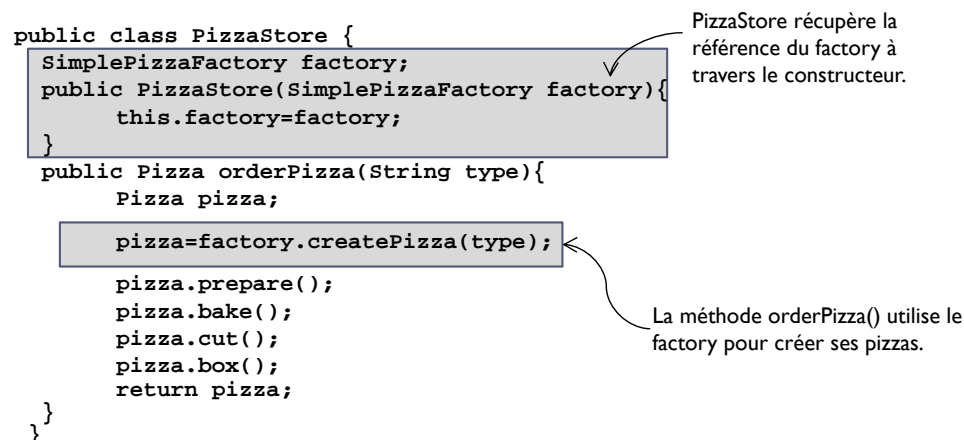


► 154

Riadh BEN HALIMA & Wajdi LOUATI [Design patterns]

## PizzaStore : Retravillons la classe PizzaStore (6/37)

- Maintenant, PizzaStore utilise SimplePizzaFactory pour créer des pizzas

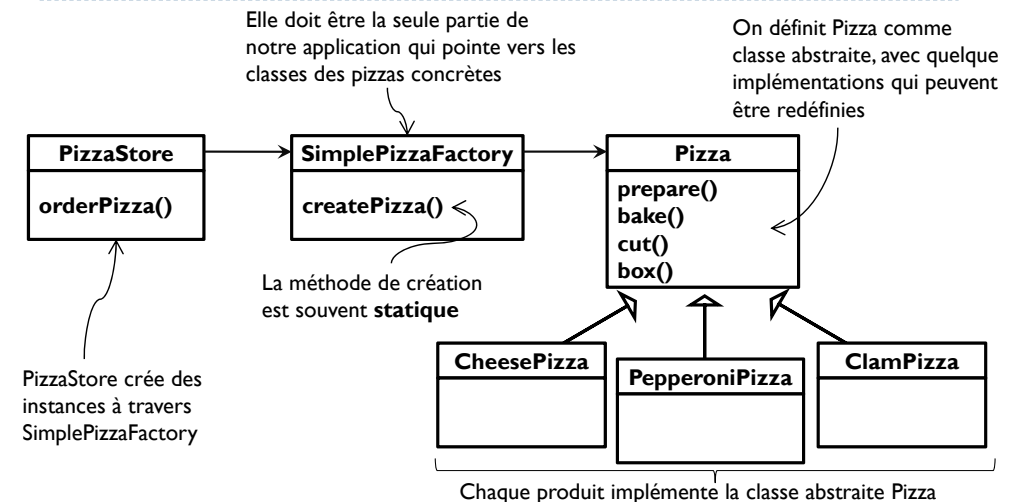


- Noter qu'on a remplacé l'opérateur `new` par une méthode concrète
  - Plus d'instanciation concrète ici

► 155

Riadh BEN HALIMA & Wajdi LOUATI [Design patterns]

## PizzaStore : Le diagramme de classes (7/37)



- Actuellement, Simple Factory n'est pas un patron. C'est plutôt un "style" de programmation

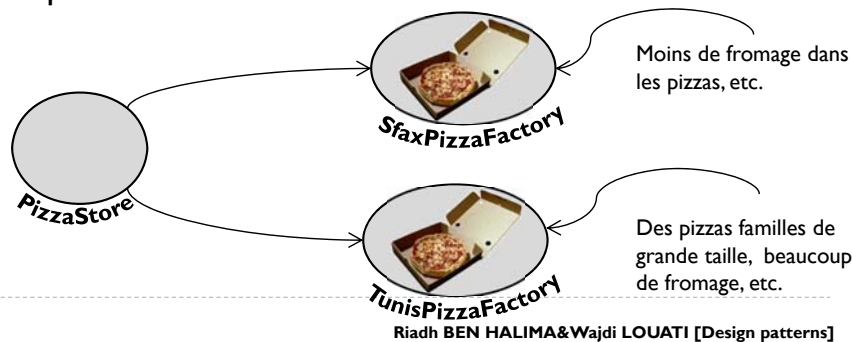
► 156

Riadh BEN HALIMA & Wajdi LOUATI [Design patterns]

## PizzaStore : Franchise de PizzaStore (8/37)



- ▶ Objectif: Franchiser PizzaStore: Créer plusieurs stores dans plusieurs villes (Sfax, Tunis, etc.)
- ▶ Besoin: Développer une application qui gère la création des pizzas pour chaque store
  - ▶ Chaque store offre différents types de pizza
- ▶ Conception: OO



▶ 157

## PizzaStore : Différents styles des PizzaStores (9/37)

- ▶ Si on prépare les mêmes pizzas

```

SfaxPizzaFactory sfactory = new SfaxPizzaFactory();
PizzaStore sfstore = new PizzaStore(sfactory);
sfstore.orderPizza("cheese");
    
```

Créer des pizzas à travers SfaxPizzaFactory

Créer les mêmes pizzas à travers TunisPizzaFactory

```

TunisPizzaFactory tnfactory = new TunisPizzaFactory();
PizzaStore tnstore = new PizzaStore(tnfactory);
tnstore.orderPizza("cheese");
    
```

- ▶ Et si chaque PizzaStore prépare ses propres styles de pizza

Je prépare les pizzas depuis des années et je veux ajouter mes propres touches d'amélioration des procédures de mon PizzaStore



Riadh BEN HALIMA & Wajdi LOUATI [Design patterns]

▶ 158

## PizzaStore : Le framework de PizzaStore (10/37)

- ▶ On donne la liberté aux franchises de créer leurs propres styles
  - ▶ Déplacer la création dans une méthode createPizza() et garder la même méthode orderPizza() pour tous les stores
  - ▶ Chaque région l'étend afin de spécifier son propre style

```

public abstract class PizzaStore {
    public Pizza orderPizza(String type) {
        Pizza pizza;
        pizza = createPizza(type);
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
    abstract Pizza createPizza(String type);
}
    
```

CreatePizza() est une méthode PizzaStore plutôt que dans le Factory

Tout ceci apparaît le même

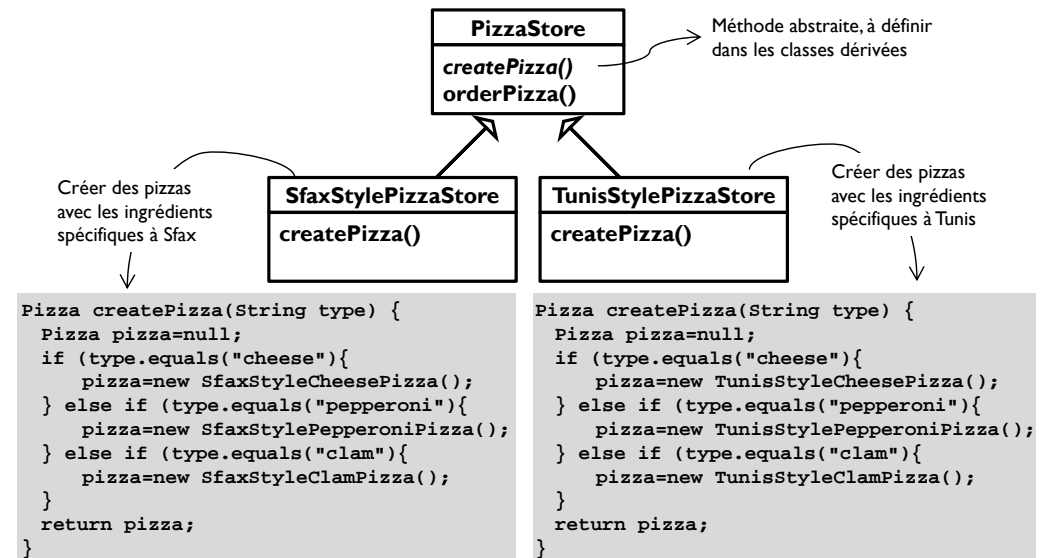
On a transféré notre objet Factory dans cette méthode

Notre "méthode Factory" est maintenant abstraite dans PizzaStore

Riadh BEN HALIMA & Wajdi LOUATI [Design patterns]

▶ 159

## PizzaStore : Les styles de pizzas (11/37)



▶ 160

Riadh BEN HALIMA & Wajdi LOUATI [Design patterns]

## PizzaStore : Un PizzaStore de Style Sfaxien (12/37)

- Les bénéfices d'une franchise
  - On obtient des fonctionnalités des pizzas communes (prepare(), bake(), cut() et box())
  - Chaque région définit sa propre méthode createPizza() qui spécifie son style de pizza

```
public class SfaxStylePizzaStore extends PizzaStore {
    Pizza createPizza(String item) {
        if (item.equals("cheese")) {
            return new SfaxStyleCheesePizza();
        } else if (item.equals("pepperoni")) {
            return new SfaxStylePepperoniPizza();
        } else if (item.equals("clam")) {
            return new SfaxStyleClamPizza();
        }
    }
}
```

On hérite la méthode orderPizza() de PizzaStore

On implémente createPizza() puisqu'elle est abstraite: C'est la "méthode Factory"

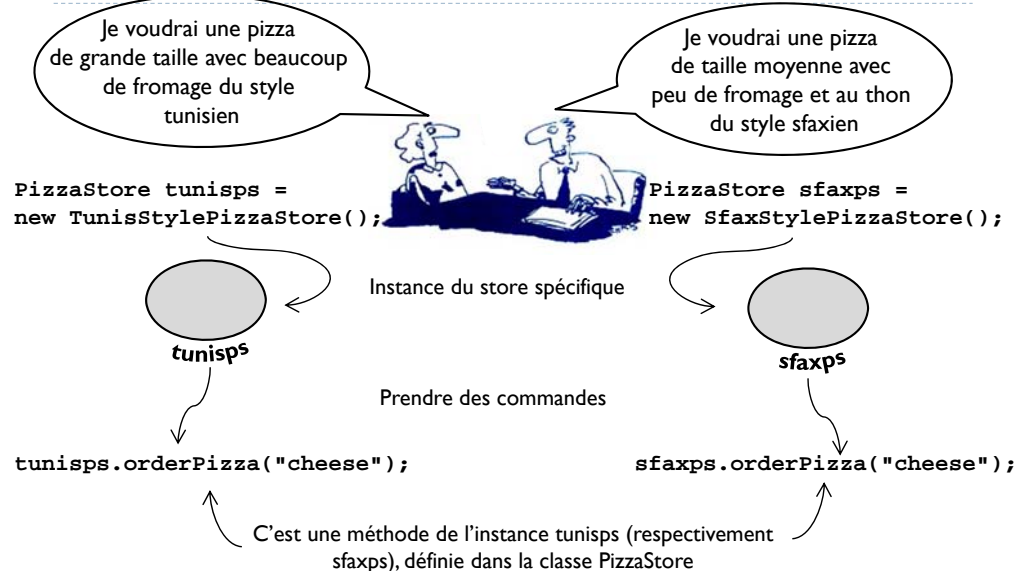
Créer des pizzas du style sfaxien!!

- Rq: Toutes les responsabilités d'instanciation sont déplacées vers la méthode createPizza() qui agit comme un factory
- La méthode factory gère la création des objets et leur encapsulation
  - Découplage du code du client dans la supère classe de la création de l'objet dans les sous-classes

161

Riadh BEN HALIMA & Wajdi LOUATI [Design patterns]

## PizzaStore : Commander des pizzas (13/37)



162

Riadh BEN HALIMA & Wajdi LOUATI [Design patterns]

## PizzaStore : Commander des pizzas (14/37)

```
tunisps.orderPizza("cheese");
```

```
sfaxps.orderPizza("cheese");
```

La méthode orderPizza() appelle createPizza()

```
Pizza pizza= createPizza("cheese");
```

```
Pizza pizza= createPizza("cheese");
```

La méthode createPizza() est implémentée dans la classe dérivée

Elle retourne une pizza au fromage style tunisien

Elle retourne une pizza au fromage style sfaxien

On termine la préparation

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

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



Pizza



Pizza

De style tunisien

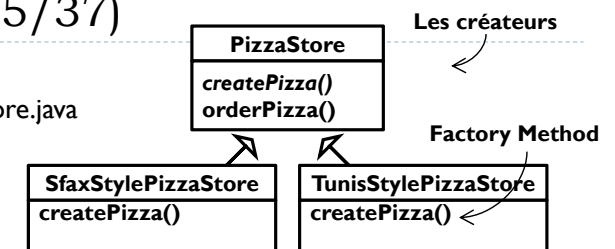
De style sfaxien

163

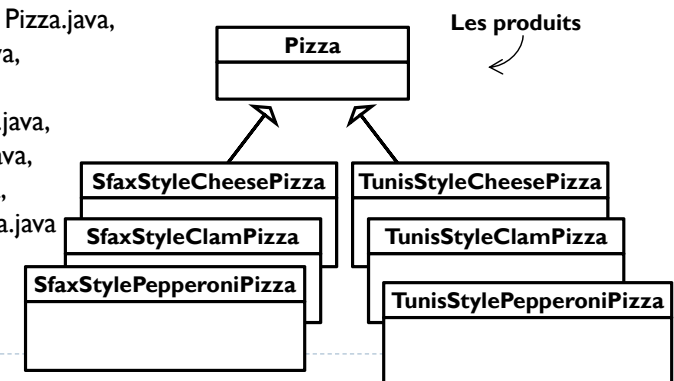
Riadh BEN HALIMA & Wajdi LOUATI [Design patterns]

## PizzaStore : Implémenter (15/37)

- Implémenter les classes: PizzaStore.java, SfaxPizzaStore.java et TunisPizzaStore.java



- Implémenter les classes: Pizza.java, SfaxStyleCheesePizza.java, SfaxStyleClamPizza.java, SfaxStylePepperoniPizza.java, TunisStyleCheesePizza.java, TunisStyleClamPizza.java, TunisStylePepperoniPizza.java



164

Riadh BEN HALIMA & Wajdi LOUATI [Design patterns]



## PizzaStore : Le patron Factory Method (16/37)

### ► Définition: **Factory Method**

- Le **patron factory method** définit une interface de création des objets, et laisse les classes-dérivées décider de la classe de l'instanciation. La méthode factory permet à une classe de déléguer l'instanciation à ces classes dérivées.

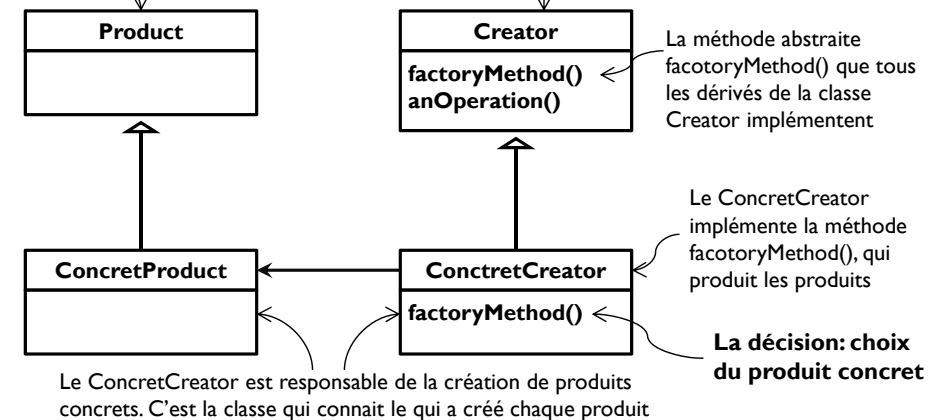
► 165

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## PizzaStore : Le diagramme de classes du patron (17/37)

La classe mère de tous les produits concrets. Elle représente aussi le type générique qui référence vers les instances des classes concrètes

Le Creator est une classe qui contient l'implémentation de toutes les méthodes qui manipulent les produits, à l'exception de factoryMethod()



► 166

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## PizzaStore : Un PizzaStore dépendant (18/37)

- Hypothèse: On n'a jamais entendu parler du factory
  - Compter le nombre d'objets de pizzas concrètes dont cette classe dépend
  - Refaire le compte si on ajoute des pizzas de style bizertin

```
public class DependentPizzaStore {
    Pizza createPizza(String style, String type) {
        Pizza pizza=null;
        if (style.equals("sfax")){
            if (type.equals("cheese")){
                pizza=new SfaxStyleCheesePizza();
            } else if (type.equals("pepperoni")){
                pizza=new SfaxStylePepperoniPizza();
            } else if (type.equals("clam")){
                pizza=new SfaxStyleClamPizza();
            }
        } else if (style.equals("Tunis")){
            if (type.equals("cheese")){
                pizza=new TunisStyleCheesePizza();
            } else if (type.equals("pepperoni")){
                pizza=new TunisStylePepperoniPizza();
            } else if (type.equals("clam")){
                pizza=new TunisStyleClamPizza();
            }
        } else {System.out.println("Erreur: type de pizza invalide");}
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
}
```

Gérer toutes les pizzas de style sfaxien

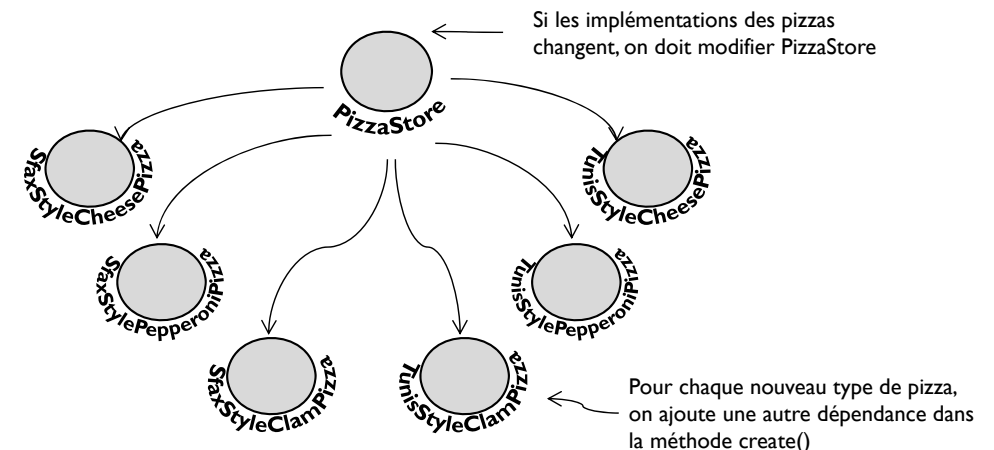
Gérer toutes les pizzas de style tunisois

► 167

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## PizzaStore : La dépendance entre les objets (19/37)

- Cette version de PizzaStore dépend de tous les objets pizzas parce qu'elle les crée directement
- On dit que PizzaStore dépend des implémentations des pizzas parce que chaque changement des implémentations concrètes des pizzas, affecte le PizzaStore



► 168

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## PizzaStore :

### L'inversion de dépendance (20/37)

- ▶ Réduire les dépendances aux classes concrètes dans notre code, est une "bonne chose"
- ▶ Le principe qui formalise cette notion s'appelle "principe d'inversion de dépendance" :

▶ **Règle 5:** Dépendre des abstractions. Ne jamais dépendre de classes concrètes.

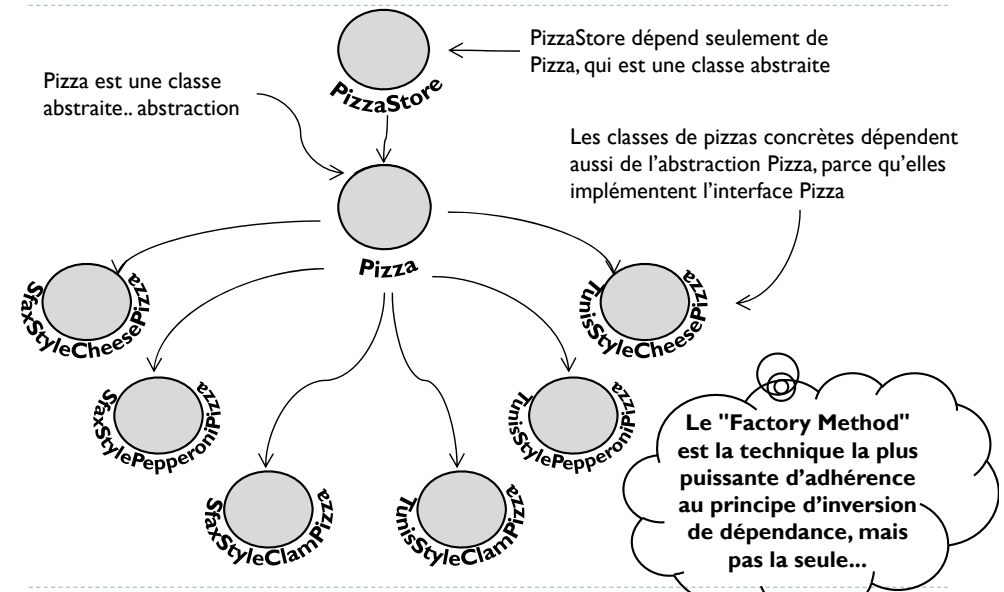
- ▶ Ce principe prétend que nos "haut-niveau" composants ne doivent pas dépendre de nos "bas-niveau" composants; plutôt, les deux doivent dépendre des abstractions.
- ▶ Un composant de haut-niveau (PizzaStore) est une classe dont le comportement dépend des autres composants de bas-niveau (Pizza)

▶ 169

Riadh BEN HALIMA & Wajdi LOUATI [Design patterns]

## PizzaStore :

### Appliquons ce principe (21/37)




▶ 170

Riadh BEN HALIMA & Wajdi LOUATI [Design patterns]

## PizzaStore :

### Les ingrédients des pizzas (22/37)

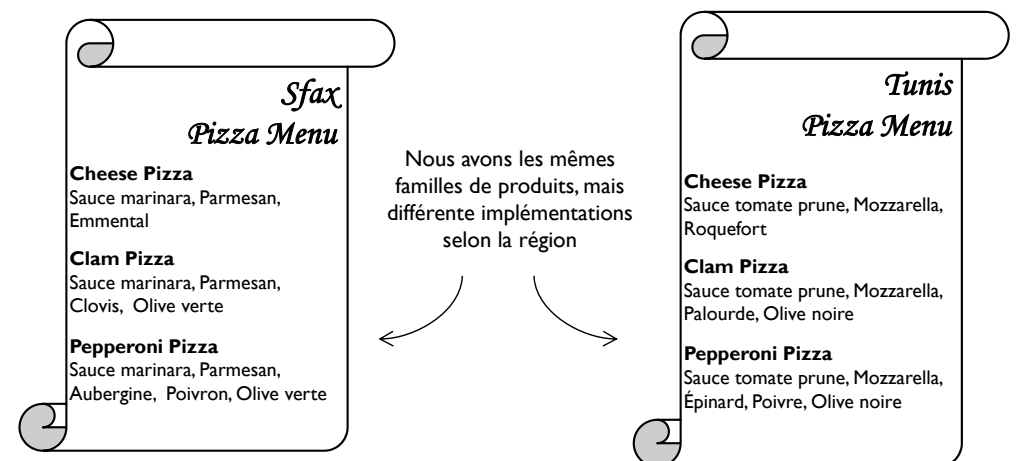
- ▶ Problème : quelques franchises n'ont pas utilisé la même procédure de préparation, et ce en substituant des ingrédients par d'autres de basse qualité, afin d'augmenter leur marge.
- ▶  Il faut assurer la consistance des ingrédients
- ▶ Solution : créer un factory qui produit les ingrédients, et les transporter aux franchises
- ▶ Le seul problème avec ce plan : Ce qui est sauce rouge à Sfax, n'est pas sauce rouge à Tunis
  - ▶ Il y a un ensemble d'ingrédients à transporter à Sfax, et un autre ensemble à transporter à Tunis

▶ 171

Riadh BEN HALIMA & Wajdi LOUATI [Design patterns]

## PizzaStore :

### Les menus des pizzas (23/37)



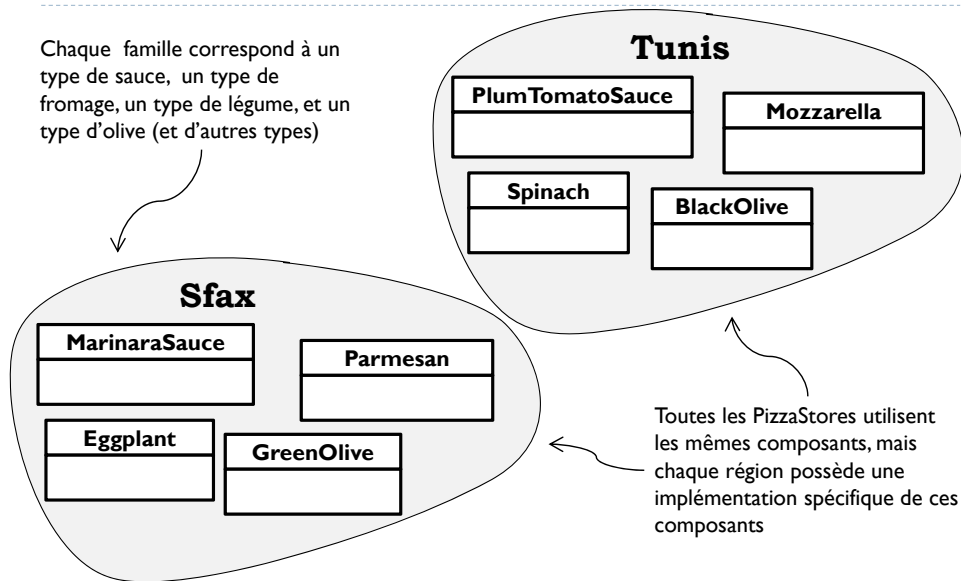
▶ 172

Riadh BEN HALIMA & Wajdi LOUATI [Design patterns]



## PizzaStore : Les familles des ingrédients (24/37)

Chaque famille correspond à un type de sauce, un type de fromage, un type de légume, et un type d'olive (et d'autres types)



► 173

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## PizzaStore : Les factories des ingrédients (25/37)

- Le factory est le responsable de la création de la pâte, la sauce, le fromage, etc.

```
public interface PizzaIngredientFactory {
    public Dough createDough();
    public Sauce createSauce();
    public Cheese createCheese();
    public veggies[] createVeggies();
    public Clam createClam();
}
```

Pour chaque ingrédient, on crée une create() méthode dans notre interface

Beaucoup de nouvelles classes, une par ingrédient

- A faire :
  - Construire un factory pour chaque région: une sous-classe de PizzaIngredientFactory qui implémente chaque create() méthode
  - Implémenter un ensemble de classes d'ingrédients, à utiliser par les factories tels que: OliveVerte, Mozzarella, SauceMarinara, etc.
  - Lier ceci avec notre ancien code de PizzaStore, tout en travaillant nos factories d'ingrédients

► 174

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## PizzaStore : Les factories de Sfax (26/37)

```
public class SfaxPizzaIngredientFactory implements PizzaIngredientFactory
{
    public Dough createDough(){
        return new ThinDough();
    }
    public Sauce createSauce(){
        return new MarinaraSauce();
    }
    public Cheese createCheese(){
        return new Parmesan();
    }
    public Veggies[] createVeggies(){
        Veggies veggies[]={new Garlic(), new Onion(), new Eggplant()};
        return veggies;
    }
    public Clam createClam(){
        return new Clovis();
    }
}
```

Pour chaque famille d'ingrédient, on crée la version sfaxienne

Palourde() pour le cas de tunis

► 175

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## PizzaStore : Retravajillons la classe Pizza (27/37)

```
public abstract class Pizza {
    String name;
    Dough dough;
    Sauce sauce;
    Cheese cheese;
    Veggies veggies[];
    Clam clam;
    abstract void prepare();
    void bake(){
        System.out.println("Cuire durant 25mn à 350°");
    }
    void cut(){
        System.out.println("Couper en morceaux à la diagonale");
    }
    void box(){
        System.out.println("Placer la pizza dans un boitier officiel");
    }
    void setName(String s){
        name=s;
    }
    String getName(){
        return name;
    }
}
```

Les ingrédients d'une paizza (liste non-exhaustive)

La collecte des ingrédients se fait dans cette méthode (à travers un factory d'ingrédients) qui sera définie par les classes dérivées

Les autres méthodes sont les mêmes (à l'exception de prepare())

► 176

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## PizzaStore : Retravillons les classes des Pizzas (28/37)

Pour faire la pizza, on besoin d'un factory. Chaque classe Pizza prend le factory à travers son constructeur

```
public class CheesePizza extends Pizza {
    PizzaIngredientFactory ingredientfactory;
    CheesePizza(PizzaIngredientFactory ingredientfactory){
        this.ingredientfactory = ingredientfactory;
    }
    void prepare(){
        System.out.println("Préparons " +name);
        dough = ingredientfactory.createDough();
        sauce = ingredientfactory.createSauce();
        cheese = ingredientfactory.createCheese();
    }
}
```

Chaque fois que la méthode prepare() a besoin d'ingrédient, elle appelle le factory pour le produire

► 177

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## PizzaStore : Retravillons les classes des Pizzas (29/37)

Un factory pour chaque type de Pizza

```
public class ClamPizza extends Pizza {
    PizzaIngredientFactory ingredientfactory;
    ClamPizza(PizzaIngredientFactory ingredientfactory){
        this.ingredientfactory = ingredientfactory;
    }
    void prepare(){
        System.out.println("Préparons " +name);
        dough = ingredientfactory.createDough();
        sauce = ingredientfactory.createSauce();
        cheese = ingredientfactory.createCheese();
        clam= ingredientfactory.createClam();
    }
}
```

Pour faire une ClamPizza, la méthode prépare les ingrédients correspondants de son factory local.

Si c'est le factory de sfax, on va préparer des clovis

► 178

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

## PizzaStore : Retravillons les PizzaStores (30/37)

Le store de Sfax est composé d'un factory sfaxien d'ingrédients/

```
public class SfaxPizzaStore extends PizzaStore {
    Pizza createPizza(String item) {
        Pizza pizza = null;
        PizzaIngredientFactory ingredientfactory=
            new SfaxPizzaIngredientFactory();

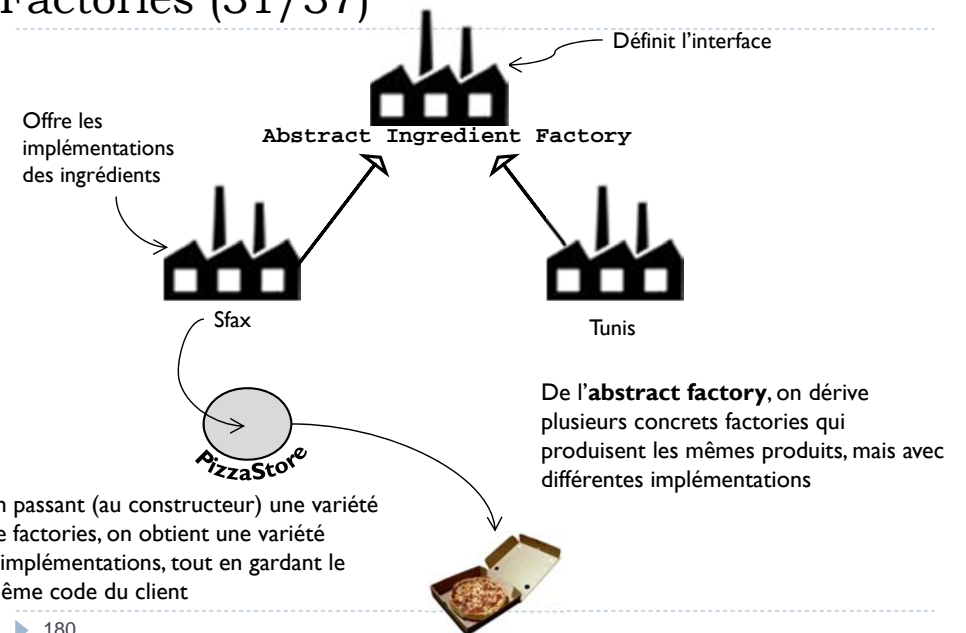
        if (item.equals("cheese"){
            pizza = new CheesePizza(ingredientfactory);
            pizza.setName("Sfax Style Cheese Pizza");
        } else if (item.equals("pepperoni"){
            pizza = new PepperoniPizza(ingredientfactory);
            pizza.setName("Sfax Style Pepperoni Pizza");
        } else if (item.equals("clam"){
            pizza = new ClamPizza(ingredientfactory);
            pizza.setName("Sfax Style Clam Pizza");
        }
        return pizza;
    }
}
```

On passe à chaque pizza le factory censé créer ses ingrédients

► 179

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

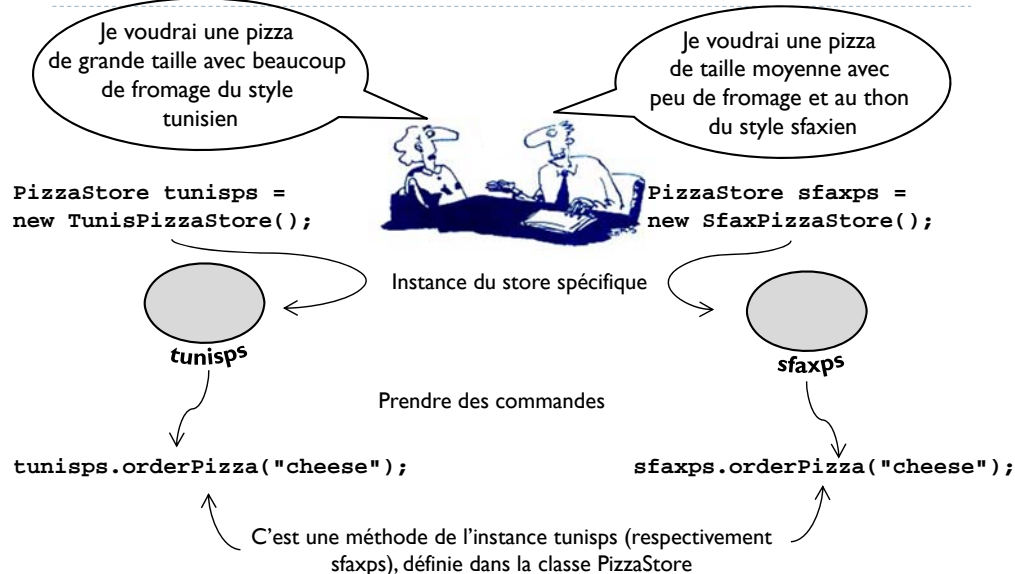
## PizzaStore : Factories (31/37)



► 180

Riadh BEN HALIMA&Wajdi LOUATI [Design patterns]

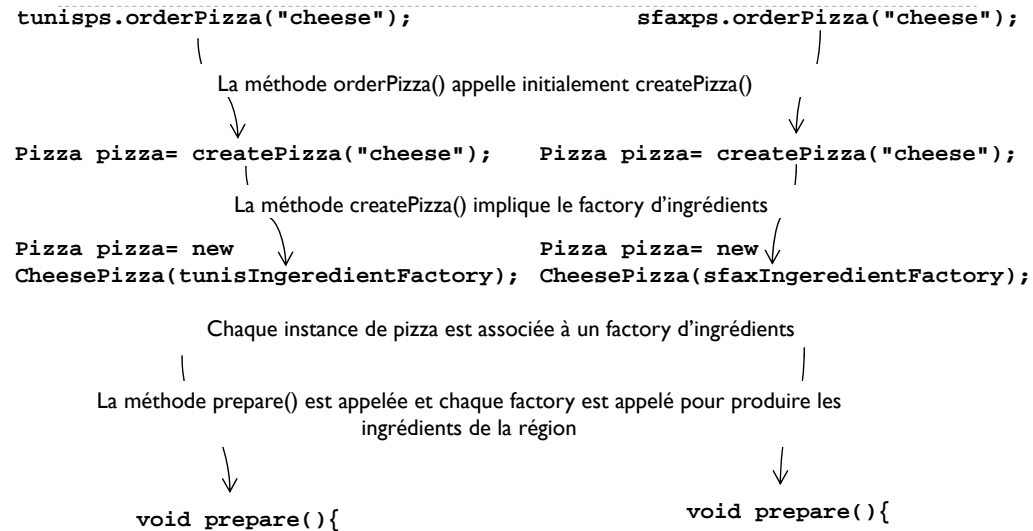
## PizzaStore : Commander des pizzas (32/37)



► 181

Riadh BEN HALIMA & Wajdi LOUATI [Design patterns]

## PizzaStore : Commander des pizzas (33/37)

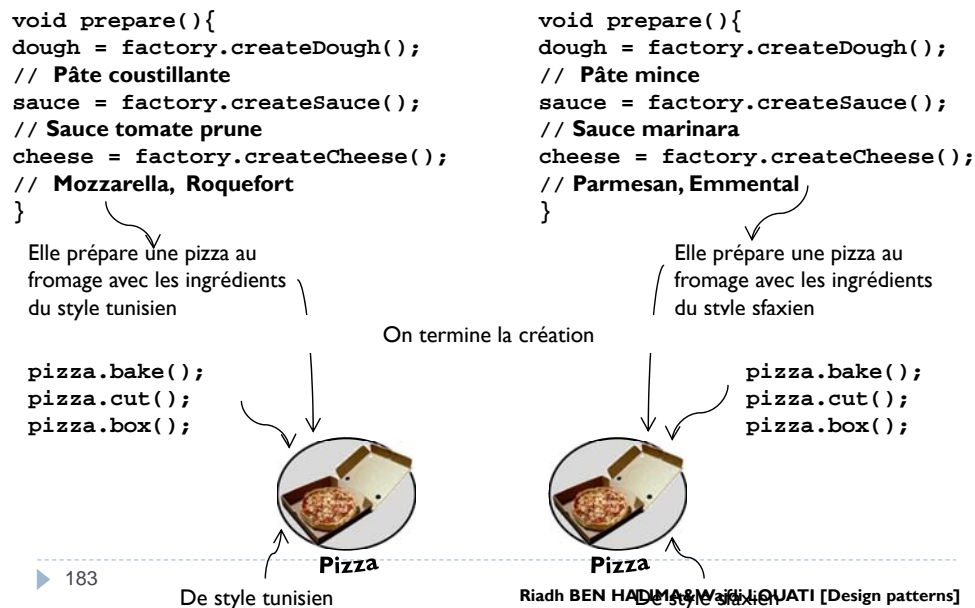


► 182

CheesePizza à la tunisoise

CheesePizza à la sfaxienne  
Riadh BEN HALIMA & Wajdi LOUATI [Design patterns]

## PizzaStore : Commander des pizzas (34/37)



► 183

Riadh BEN HALIMA & Wajdi LOUATI [Design patterns]

## PizzaStore : Le patron Abstract Factory (35/37)

- Définition: **Abstract Factory**
- Le **patron abstract factory** offre une interface de création de familles d'objets dépendants (en relation), sans spécifier leurs classes concrètes.

► 184

Riadh BEN HALIMA & Wajdi LOUATI [Design patterns]

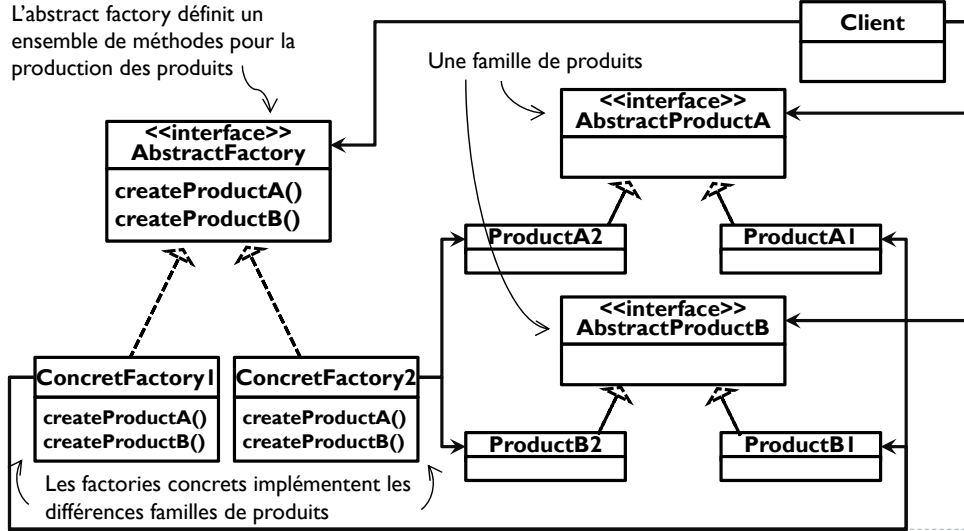
## PizzaStore :

### Le diagramme de classes du patron (36/37)

Le client est composé au moment de l'exécution par un factory concret

L'abstract factory définit un ensemble de méthodes pour la production des produits

Une famille de produits



► 185

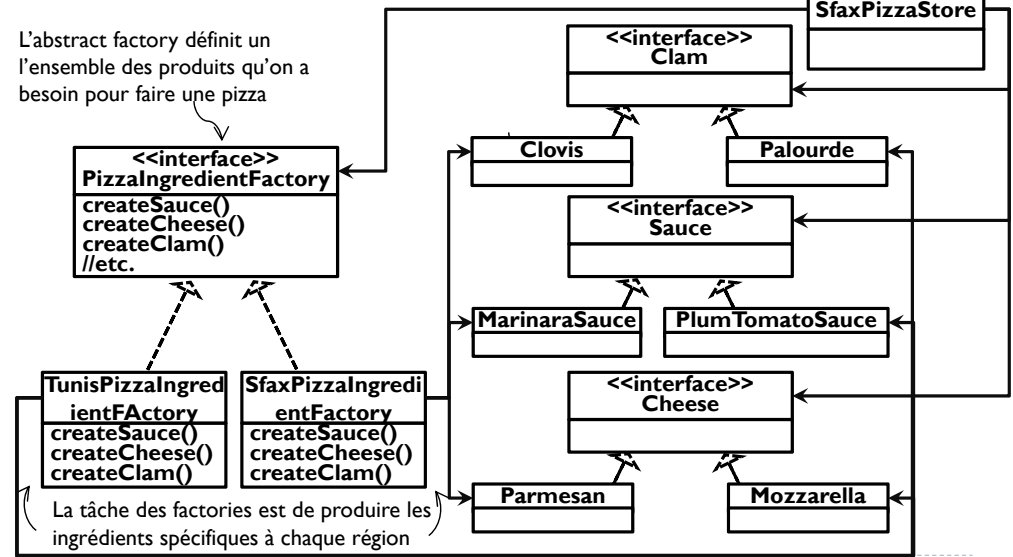
Riadh BEN HALIMA & Wajdi LOUATI [Design patterns]

## PizzaStore :

### La conception finale (37/37)

Les clients de l'abstract factory sont les stores de Sfax et de Tunis

L'abstract factory définit un ensemble des produits qu'on a besoin pour faire une pizza



► 186

Chaque factory produit différent implémentation de chaque famille de produits

Riadh BEN HALIMA & Wajdi LOUATI [Design patterns]

## Récapitulatif (1/2)

- Bases de l'OO: Abstraction, Encapsulation, Polymorphisme & Héritage
- Principes de l'OO
  - Encapsuler ce qui varie
  - Favoriser la composition sur l'héritage
  - Programmer avec des interfaces et non des implémentations
  - Opter pour une conception faiblement couplée
  - Les classes doivent être ouvertes pour les extensions et fermées pour les modifications
  - **Dépendre des abstractions. Ne jamais dépendre de classes concrètes**
- Patron de l'OO
  - Strategy: définit une famille d'algorithmes interchangeables
  - Observer: définit une dépendance 1-à-plusieurs entre objets.
  - Decorator: attache des responsabilités additionnelles à un objet dynamiquement.
  - **Factory Method**: définit une interface de création des objets, et laisse les classes-dérivées décider de la classe de l'instanciation.
  - **Abstract Factory**: offre une interface de création de familles d'objets dépendants, sans spécifier leurs classes concrètes

► 187

Riadh BEN HALIMA & Wajdi LOUATI [Design patterns]

## Récapitulatif (2/2)

- Tous les factories encapsule la création des objets
- Malgré qu'il n'est un vrai patron, le "Simple Factory" est une manière simple de découplage de clients des concrètes classes
- Factory Method repose sur l'héritage: La création d'objet est déléguée aux sous-classes qui implémentent la méthode factory de création d'objets
- Abstract Factory repose sur la composition d'objets: La création d'objet est implémentée dans une méthode exposée dans l'interface du factory
- Tous les patrons factories soutiennent le faible couplage entre notre application et les classes concrètes.
- L'intension de Factory Method est de permettre à une classe de reporter l'instanciation à ses sous-classes.
- L'intension d'Abstract Factory est de créer une famille d'objets en relation sans dépendre de leurs classes concrètes
- L'inversion de dépendance nous guide afin d'éviter les dépendances des classes concrètes, et s'efforcer pour les abstractions
- Factories sont des techniques puissantes de codages des abstractions et non des classes concrètes

► 188

Riadh BEN HALIMA & Wajdi LOUATI [Design patterns]