UMONS
Université de Mons

JDBC / Design patterns

Année académique 2020-2021

Prof Pierre Manneback

Adriano GUTTADAURIA

Ir Olivier DEBAUCHE

POLYTECH MONS

JDBC

Qu'est ce que c'est?

J*ava* **D***ata* **B***ase* **C***onnectivity est un API qui permet de programmer l'accès aux bases de données en Java.*

Elle permet aux applications Java d'accéder par le biais d'une interface commune à des sources de données pour lesquelles il existe des pilotes JDBC. Normalement, il s'agit d'une base de données relationnelle, et des pilotes JDBC sont disponibles pour tous les systèmes connus de bases de données relationnelles.

JDBC

```
import java.sql.*;
public class MySQLAccess {
  public static void main (String[] args) throws ClassNotFoundException, SQLException {
    Class.forName ("com.mysql.cj.jdbc.Driver");
    Connection conn = DriverManager
              .getConnection("jdbc:mysql://s-l-iglab-01/pizzeria?"
                   + "user=sqluser&password=sqluserpw");
    String requete = " SELECT nom, prix FROM ingredient ";
    Statement stmt = conn.createStatement();
    ResultSet res;
    res = stmt.executeQuery(requete);
    String nom;
    float prix;
    while (res.next()) {
      nom = res.getString(1);
      prix = res.getFloat(2);
      System.out.println(nom + "   " + prix);
    }
    stmt.close();   // On libère le Statement
    res.close();    // On libère le ResultSet
    conn.close();  // On libère la Connection
  }
}
```

Nom de la base de données

Voir slide suivant

```
pâte   3
sauce tomate   1.5
crème blanche   1.5
fromage   2
champignon   1
fruits de mer   2.5
jambon   2
olives   1
```

JDBC

Correspondances entre Type SQL et méthode JAVA

Type SQL	Méthode JAVA	
BIT / BOOLEAN	getBoolean	updateBoolean
TINYINT	getByte	updateByte
SMALLINT	getShort	updateShort
INTEGER	getInt	updateInt
BIGINT	getLong	updateLong
REAL	getFloat	updateFloat
DOUBLE	getDouble	updateDouble
CHAR(n) / VARCHAR(n)	getString	updateString
DATE	getDate	updateDate
TIME	getTime	updateTime
TIMESTAMP	getTimestamp	updateTimestamp
DECIMAL	getBigDecimal	updateBigDecimal

JDBC

	Sélection	Mise à jour	Insertion	Suppression
Méthode de de l'interface *Statement*	executeQuery	executeUpdate	execute	
Résultat	Un objet de type ResultSet	Nombre d'enregistrements modifiés ou -1 en cas d'échec		

Conseil: utiliser StringBuilder et sa méthode append pour construire vos requêtes.

Exemple:

```
StringBuilder req = new StringBuilder( "SELECT * FROM" );
        req.append(table_name);
```

Patrons de conception

Il existe 24 Design patterns qui permettent de résoudre des problèmes de programmation.

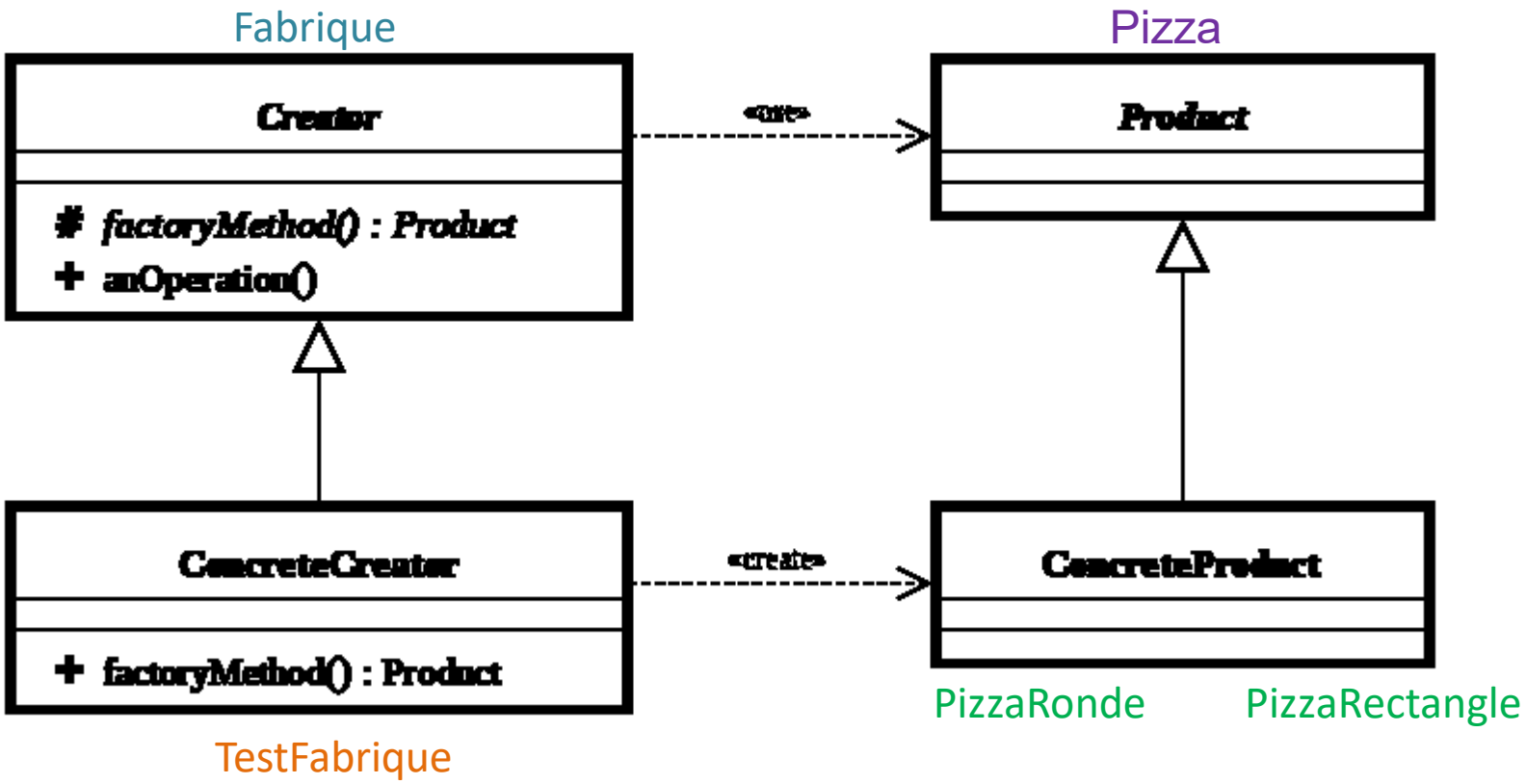
Vous connaissez *template method* et *iterator*.

Nous n'en aborderons que 8.

Construction	Structure	Comportementaux
Factory Method (Fabrique)	Composite (Composite)	Strategy (Stratégie)
Abstract Factory (Fabrique Abstraite)	Decorator (Décorateur)	Observer (Observateur)
Singleton (Singleton)	State (Etat)	

Factory Method

Permet d'encapsuler les méthodes et de se concentrer sur la création d'objet



Factory Method

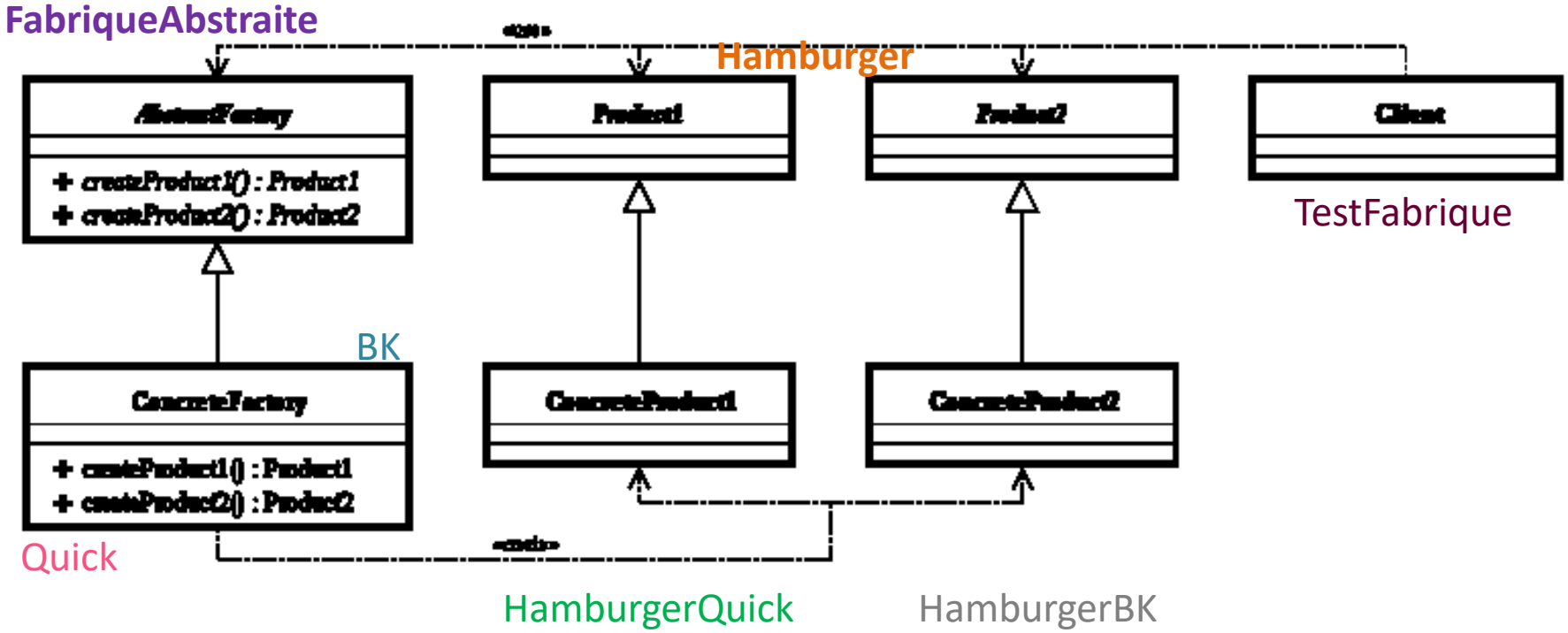
Permet d'encapsuler les méthodes et de se concentrer sur la création d'objet

```
abstract class Pizza {
  abstract void affiche();
}

class PizzaRonde extends Pizza {
  public void affiche() {
    System.out.println("Pizza circulaire");
  }
}

class PizzaRectangle extends Pizza {
  public void affiche() {
    System.out.println("Pizza rectangle");
  }
}
```

```
class Fabrique {
  public Pizza getPizza() {
    double x = Math.random();
    if (x < 0.5) return new PizzaRonde();
      else return new PizzaRectangle();
  }
}

public class TestFabrique {
  public static void main (String args[]) {
    Fabrique fab = new Fabrique();
    for (int i=0; i<4; i++) {
      Pizza p = fab.getPizza();
      p.affiche();
    }
  }
}
```

Abstract Factory

Permet de choisir parmi plusieurs familles d'objets et ensuite d'utiliser les objets d'une même famille



Abstract Factory

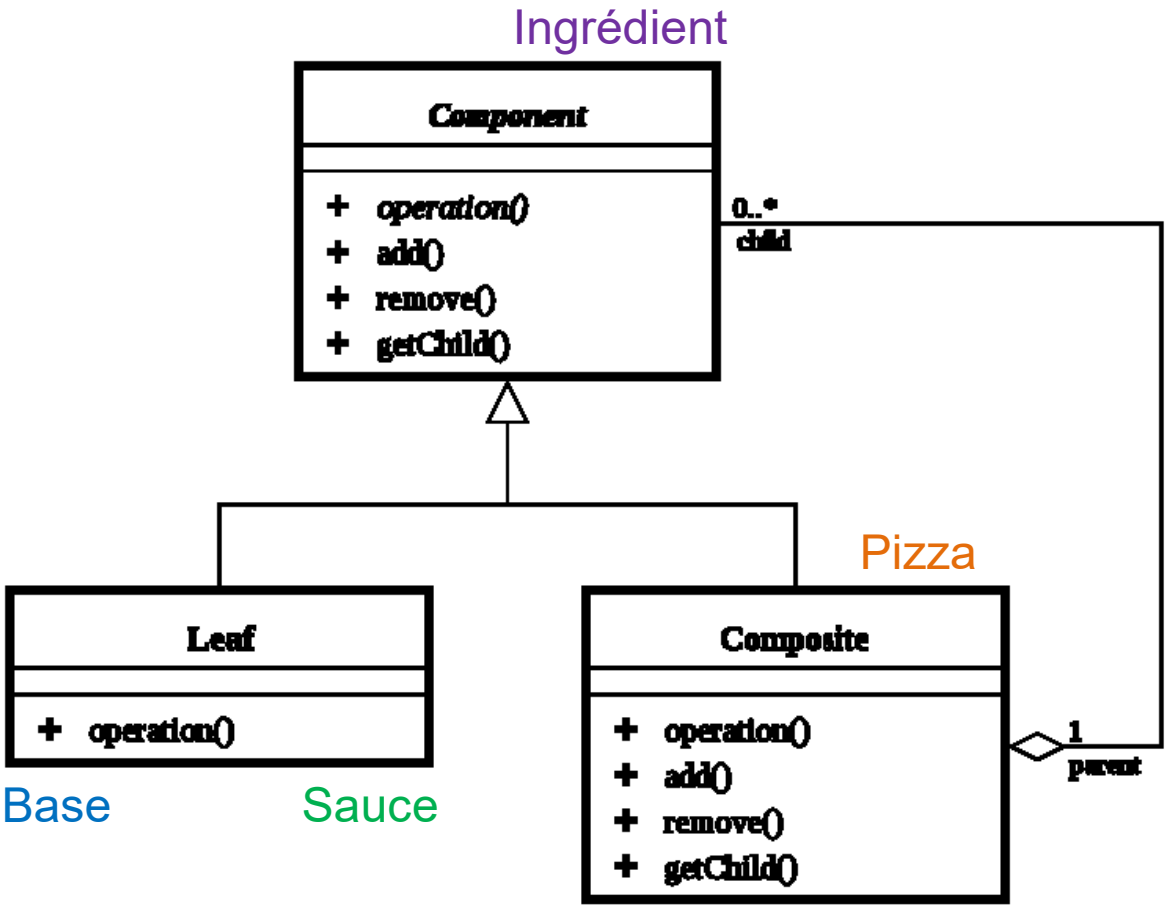
Permet de choisir parmi plusieurs familles d'objets et ensuite d'utiliser les objets d'une même famille

```
abstract class Hamburger {
  public abstract String type();
}
class HamburgerQuick extends Hamburger {
   public String type() { return "Geant"; }
}
class HamburgerBK extends Hamburger {
   public String type() { return "Whopper"; }
}
abstract class FabriqueAbstraite {
   abstract Hamburger creerHamburger();
}
class Quick extends FabriqueAbstraite {
  public Hamburger creerHamburger() {
     return new HamburgerQuick();
  }
}
```

```
class BK extends FabriqueAbstraite {
  public Hamburger creerHamburger() {
     return new HamburgerBK();
  }
}

public class TestFabrique {
   public static void main (String args[]) {
     Hamburger hambu;
     FabriqueAbstraite f = new Quick();
     hambu = f.creerHamburger();
     System.out.println(hambu.type());
   }
}
```

Composite

Permet de combiner des classes ou objets pour former une nouvelle structure



Composite

Permet de combiner des classes ou objets pour former une nouvelle structure

```
Import java.util.*;    // ArrayList
abstract class Ingredient {
  private String nom;
  public void ajoute (Ingredient i) {}
  public abstract void affiche();
  public Ingredient (String nom) {
    this.nom = nom;
  }
  public String getNom() {
    return nom;
  }
}
class Sauce extends Ingredient {
  public Sauce (String nom) { super(nom); }
  public void affiche () {
    System.out.println ("Sauce "+getNom());
  }
}
```

```
class Base extends Ingredient {
  public Base (String nom) { super(nom);}
  public void affiche() {
    System.out.println(getNom());
  }
}
class Pizza extends Ingredient {
  private ArrayList<Ingredient> listeIngredient =
              new ArrayList<Ingredient> ();
  Pizza (String nom) {super(nom);}
  public void ajoute (Ingredient i) {
    listeIngredient.add(i);
  }
  public void affiche() {
    System.out.println("Pizza: " +getNom());
    for (Ingredient i: listeIngredient) { i.affiche(); }
  }
}
```

Composite

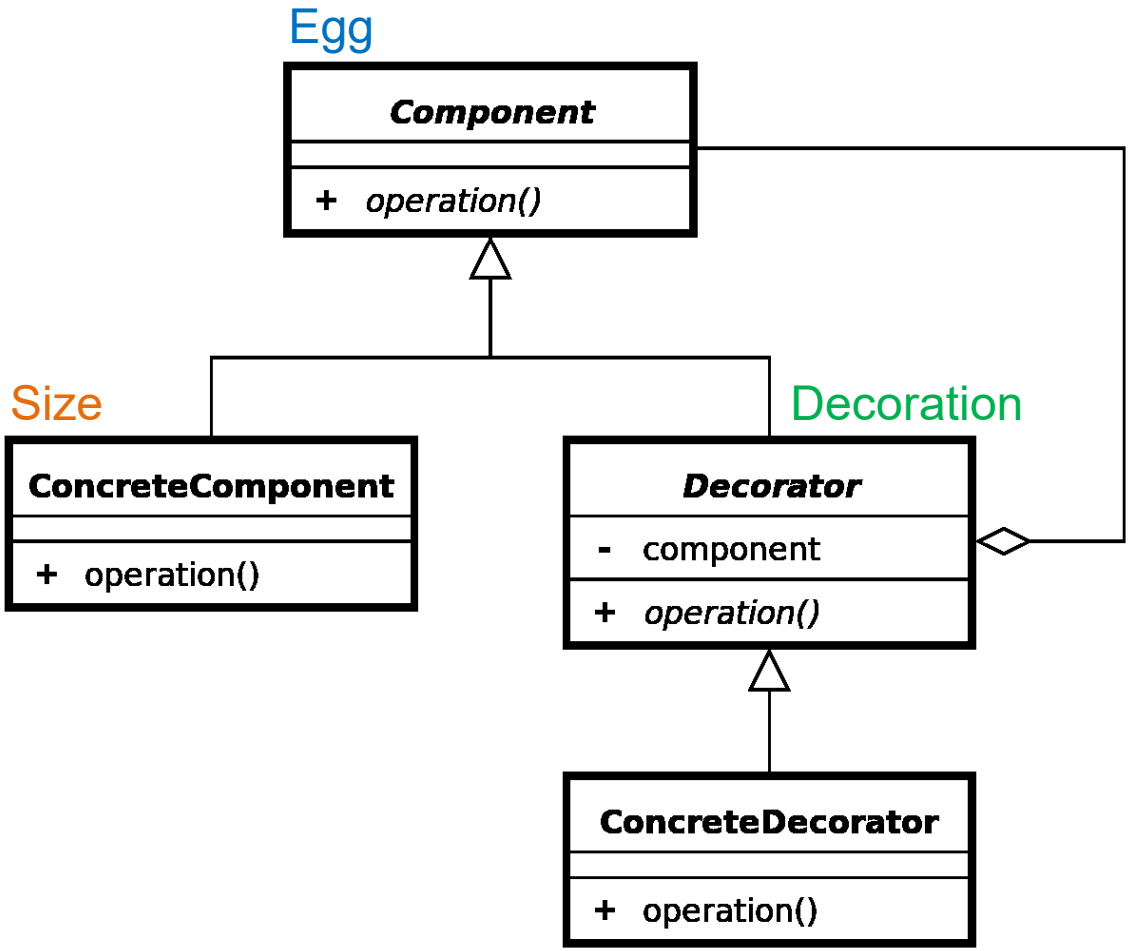
Permet de combiner des classes ou objets pour former une nouvelle structure

```
public class TestComposite {
  public static void main (String args[]) {
    Base tomate = new Base("tomate");
    Base thon = new Base(« Thon");
    Base champignon = new Base("champignon");
    Sauce fromage = new Sauce("fromage");
    Pizza perso = new Pizza("perso");
    perso.ajoute(tomate);
    perso.ajoute(thon);
    perso.ajoute(champignon);
    perso.ajoute(fromage);
    perso.affiche();
  }
}
```

```
Pizza perso
tomate
Thon
champignon
Sauce fromage
```

Decorator

Permet d'ajouter dynamiquement des fonctionnalités à une classe existante



Decorator

Permet d'ajouter dynamiquement des fonctionnalités à une classe existante

```
abstract class Egg {
  abstract void affiche();
}
public class Size extends Egg {
  private char size;
  public Size (char size) { this.size = size; }
  public void affiche () {
    System.out.print("Œuf taille " + size);
  }
}
public class Decoration extends Egg {
  private Egg e;
  Private String d;
  public Decoration(char e, String d) {
    this.e = new Size(e); this.d = d;
  }
  public void affiche() {
    e.affiche();
    System.out.println(" decoration: " + d);
  }
}
```

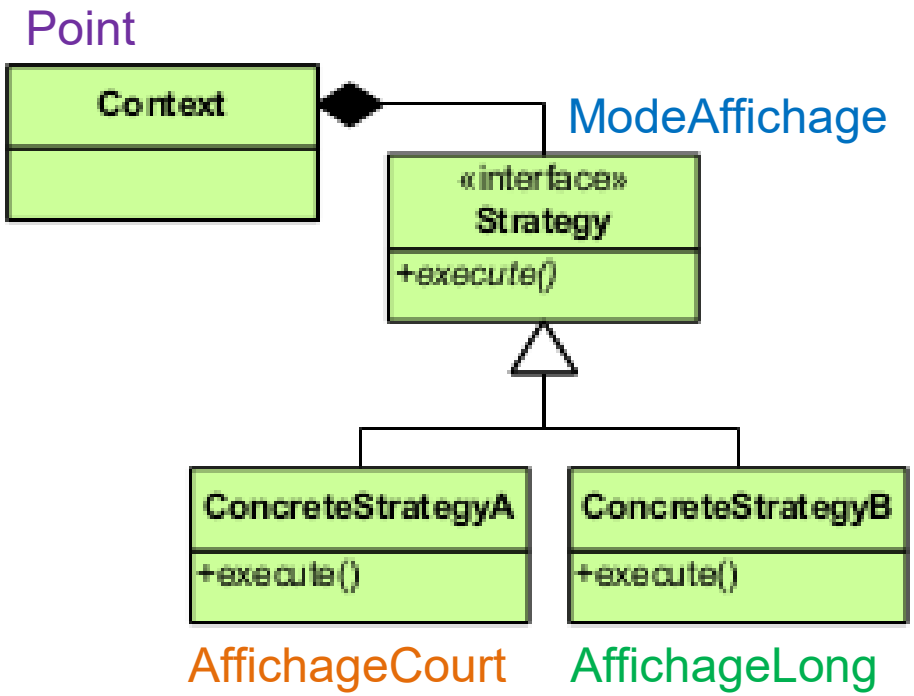
```
public class TestDecorateur {
 public static void main (String args[]) {
   Egg ed  = new Decoration('L', "Paillettes");
   ed.affiche();
   Egg e = new Size('M');
   e.affiche();
 }
}
```

Œuf taille L décoration: Paillettes

Œuf taille M

Strategy

Permet d'encapsuler différents algorithmes dans des classes séparées



Strategy

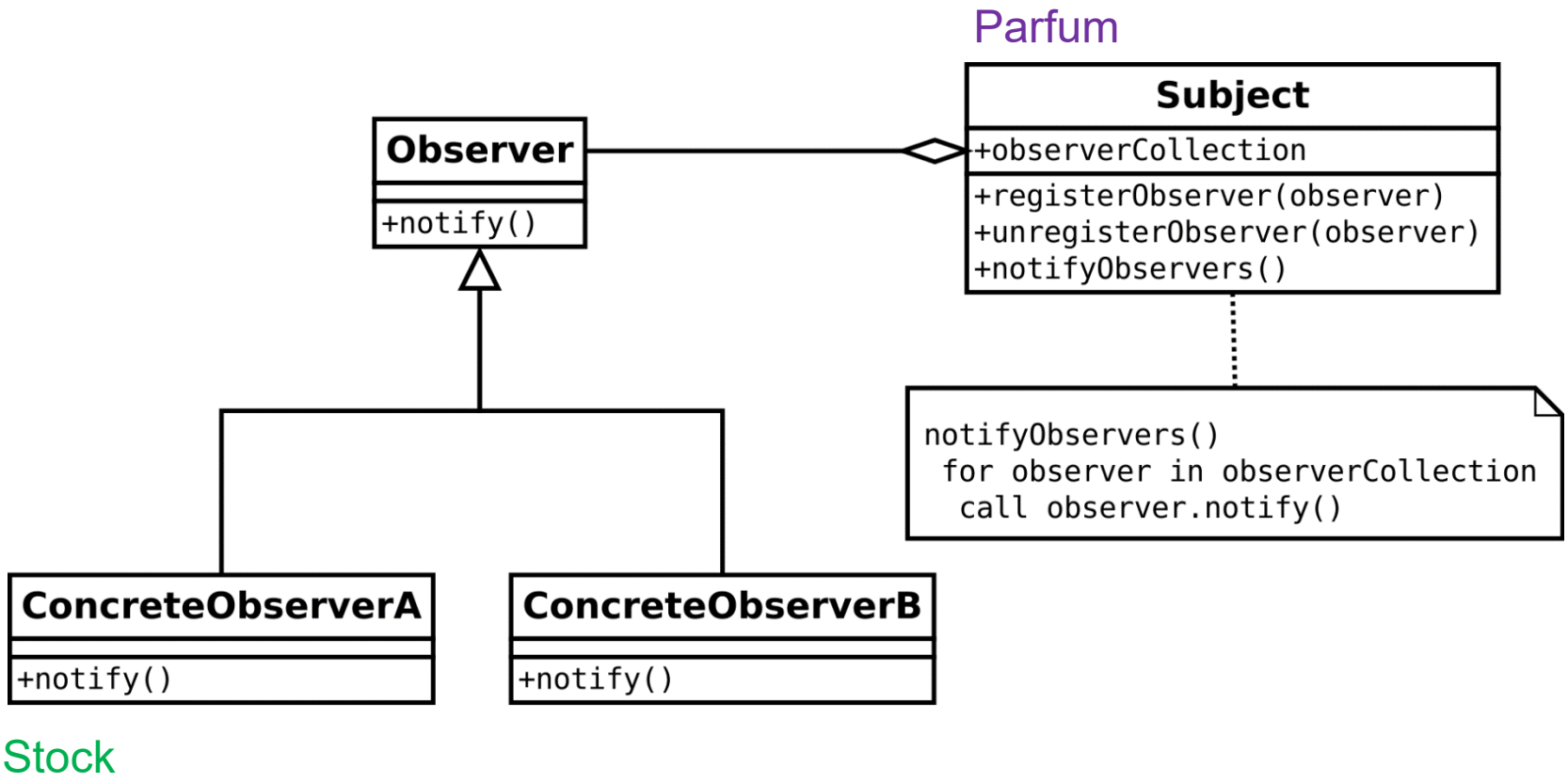
Permet d'encapsuler différents algorithmes dans des classes séparées

```
abstract class ModeAffichage {
  public abstract void presente (int x,int y,int z);
}
class AffichageCourt extends ModeAffichage {
  public void presente(int x, int y, int z) {
    System.out.println("(" +x+ ","+y+ ","+z+")");
  }
}
Class AffichageLong extends ModeAffichage {
  public void presente(int x, int y, int z) {
    System.out.println("abscisse = " + x +
      " ordonnee = "  + y + " altitude = " + z);
  }
}
```

```
class Point {
  private int x, y, z;
  private ModeAffichage mode;
  public Point (int x, int y, int z, ModeAffichage mode)
{
    this.x = x; this.y = y; this.z = z;
    this.mode = mode;
  }
  void affichage() { mode.presente(x,y,z);
}
public class TestStrategie {
 public static void main (String args[]) {
    ModeAffichage court = new AffichageCourt();
    ModeAffichage along = new AffichageLong();
    Point p1 = new Point(12,15,22, court);
    Point p2 = new Point(12,15,22, along);
    p1.affichage();
    p2.affichage();
 }
}
```

Observer

Permet d'encapsuler différents algorithmes dans des classes séparées



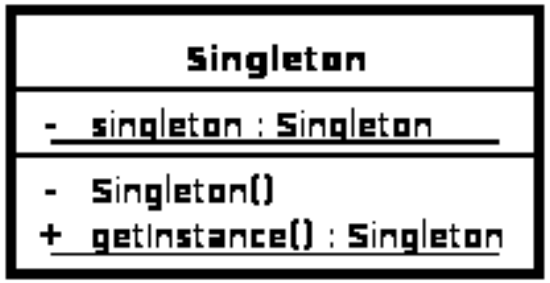
Observer

Permet d'encapsuler différents algorithmes dans des classes séparées

```
class Stock implements Observer {
  public void update (Observable obj, Object o)
  {
    if (obj instanceof Parfum) {
      System.out.println("Nouveau Stock : " +((Parfum)obj).getQt() + " de Parfum: " + ((Parfum)obj).getNom());
    }
  }
}
class Parfum {
  private String nom; private int qt;
  public Parfum(String nom, int qt) {
    this.nom = nom; this.qt = qt;
  }
  public String getNom() { return nom; }
  public int getQt() { return qt; }
  public void ajouterRetirer (int variation) {
    qt += variation;
    if (variation != 0) {
      setChanged();
      notifyObservers();
    }
  }
}
public class TestObserver {
  public static void main (String args[]) {
  Stock obs = new Stock();
  Parfum p
      = new Parfum("Miss Dior Chérie", 3);
  p.addObserver(obs);
  p.ajouterretirer(3);   // Ajoute 3
  p.ajouterretirer(-5);   // Retire 5
}
```

Singleton

Permet d'assurer qu'il n'existe qu'une seule instance de l'objet qui sert de point d'entrée



Singleton

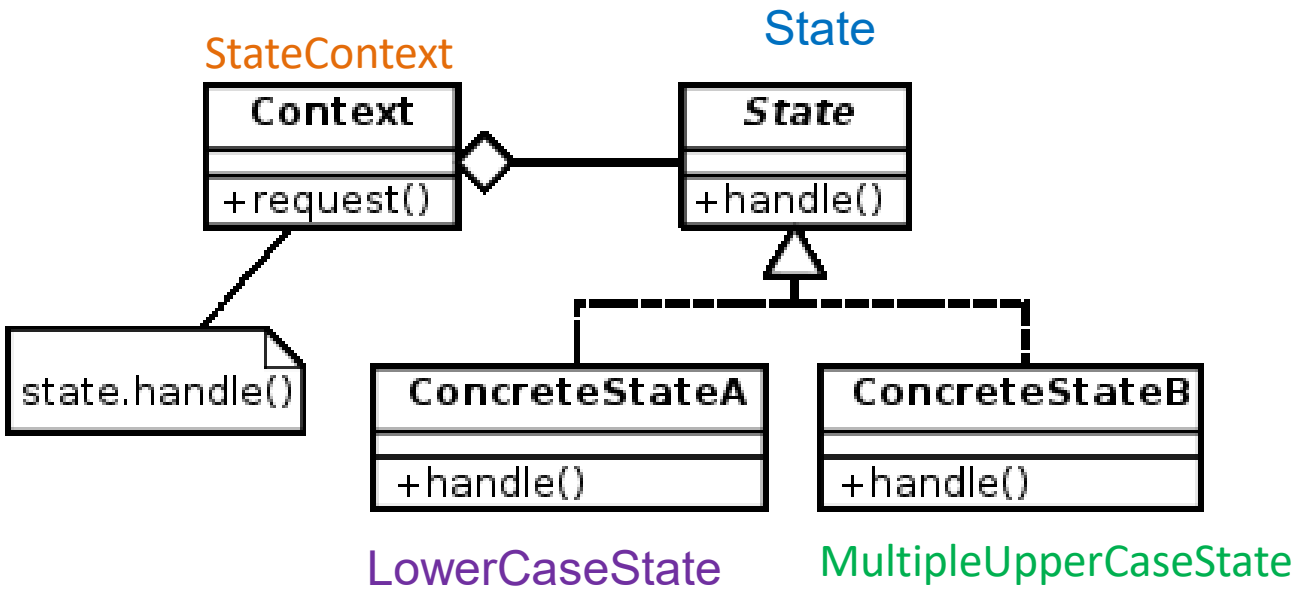
Permet d'assurer qu'il n'existe qu'une seule instance de l'objet qui sert de point d'entrée

```
public final class Singleton {
   private static Singleton instance;
   public String value;

   private Singleton(String value) {
      // The following code emulates slow initialization.
      try {
         Thread.sleep(1000);
      } catch (InterruptedException ex) {
         ex.printStackTrace();
      }
      this.value = value;
   }

   public static Singleton getInstance(String value) {
      if (instance == null) {
         instance = new Singleton(value);
      }
      return instance;
   }
}
```

```
public class DemoSingleThread {
   public static void main(String[] args) {
      System.out.println("If you see the same value, then singleton was reused (yay!)" +
"\n" + "If you see different values, then 2 singletons were created (booo!!)" + "\n\n" +
            "RESULT:" + "\n");
      Singleton singleton = Singleton.getInstance("FOO");
      Singleton anotherSingleton = Singleton.getInstance("BAR");
      System.out.println(singleton.value);

System.out.println(anotherSingleton.value);
   }
}
```

State

Change le comportement de l'objet en fonction de l'état interne du système



State

Change le comportement de l'objet en fonction de l'état interne du système

```
interface State {
   void writeName(StateContext context, String name);
}

class LowerCaseState implements State {
   @Override
   public void writeName(StateContext context, String name) {
      System.out.println(name.toLowerCase());
      context.setState(new MultipleUpperCaseState());
   }
}

class MultipleUpperCaseState implements State {
private int count = 0;

   @Override
   public void writeName(StateContext context, String name) {
      System.out.println(name.toUpperCase());
      /* Change state after StateMultipleUpperCase's writeName()
gets invoked twice */
      if (++count > 1) {
         context.setState(new LowerCaseState());
      }
   }
}
```

```
class StateContext {
   private State state;

   public StateContext() {
      state = new LowerCaseState();
   }

   void setState(State newState) {
      state = newState;
   }

   public void writeName(String name) {
      state.writeName(this, name);
   }
}
```

State

Change le comportement de l'objet en fonction de l'état interne du système

```
public class StateDemo {
   public static void main(String[] args) {
      StateContext context = new StateContext();

      context.writeName("Monday");
      context.writeName("Tuesday");
      context.writeName("Wednesday");
      context.writeName("Thursday");
      context.writeName("Friday");
      context.writeName("Saturday");
      context.writeName("Sunday");
   }
}
```

```
monday
TUESDAY
WEDNESDAY
thursday
FRIDAY
SATURDAY
sunday
```

Source: https://en.wikipedia.org/wiki/State_pattern

Le distributeur de Pizza

Concevez un programme qui simule le comportement simplifié d'un distributeur de pizza

Les pizzas sont composées des aliments suivants : pâte (3€), sauce tomate (1,5€), crème blanche (1,5€), fromage (2€), champignons (1€), fruits de mer (2,5€), jambon (2€), olives (1€).

Les différentes pizzas disponibles sont :

- **Margherita** : pâte, sauce tomate, fromage
- **Prosciutto** : pâte, sauce tomate, fromage, jambon
- **Frutti di mare**: pâte, sauce tomate, fromage, fruits de mer.
- **Carbonara**: pâte, crème blanche, fromage, jambon.

Ouvrez le code de démarrage fourni sur Moodle.

1. Créez la classe **pizza** et l'interface *PizzaComponent*

2. Les objets ingrédients seront remplis à l'aide de l'accès base de données.

3. Il vous est demandé d'utiliser le patron de conception **composite** pour implémenter les pizzas et leurs aliments.

Concevez un programme qui simule le comportement simplifié d'un distributeur de pizza

4. Les pizzas peuvent venir de deux fast-foods différentes: Pizza Hut ou Domino's Pizza. Les pizzas du Pizza Hut sont similaires à celle du Domino's Pizza, mais elles contiennes en plus des olives.

Il vous est demandé d'utiliser le patron de conception **fabrique abstraite** pour construire les pizzas du Domino's Pizza et du Pizza Hut.

5. Les fabriques peuvent fabriquer au maximum 2 pizzas de chaque sorte. Il vous est demandé d'implémenter les fabriques à l'aide du patron de conception **singleton** afin qu'il n'existe qu'une seul instance de chaque fabrique au sein du système.

6. Indépendamment de leur fast-food d'origine, la pâte des pizzas peut être « décorée ». La pâte « Cheesy Crust » augmente le prix de la pizza de 2€ et change son nom en « Cheesy + 'Nom de la pizza' ». La pâte « Pan » augmente le prix de la pizza de 1,5€ et change son nom en « Pan +'nom de la pizza' ». Il vous est demandé d'employer le patron de conception décorateur pour cette fonctionnalité. Le patron de conception **décorateur** pour réaliser les variantes de pizzas.

Concevez un programme qui simule le comportement simplifié d'un distributeur de pizza

7. En fonction de l'état du distributeur (patron de conception **State**) adapter les actions possibles pour l'utilisateurs quand le distributeur est en cours de fabrication, en attente d'une commande et en panne ou en manque d'ingrédients. Il vous est également demandé d'utiliser les threads.

8. Les distributeurs de pizzas connaissent périodiquement une pénurie dans l'un ou l'autre des ingrédients nécessaires à la fabrication des pizzas. La notification des quantités faibles d'ingrédient disponibles doit être implémentée en suivant l'architecture du patron de conception **Observateur**.

9. Réaliser une interface utilisateur en JavaFX à l'aide de SceneBuilder