# Simple Factory

# Simple Factory Pattern : le problème par l'exemple

```
public class Jeu {

   ArrayList<Personnage>personnages = new ArrayList<Personnage>();

public Jeu(){
      Personnage principal = new Personnage();
      personnages.add(principal);
   }
}
```

En fonction du type de jeu, on veut initialiser le jeu avec un personnage principal différent.

*Ainsi pour un jeu de guerre, le personnage principal est un guerrier, pour un autre un monstre etc.*

# Simple Factory Pattern : le problème par l'exemple

```
public class Jeu {

   ArrayList<Personnage>personnages = new ArrayList<Personnage>();
   public Jeu(TypeJeu typeJeu) throws Exception{
      Personnage principal;
      if (typeJeu.equals(TypeJeu.guerre))
         principal = new Guerrier();
      else
         if (typeJeu.equals(TypeJeu.horreur))
            principal = new Monstre();
         else throw new Exception("Jeu inconnu");
      personnages.add(principal);
   }
}
```

Et si on ajoute ou retire des classes de Personnage ? ...

# Simple Factory Pattern : Solution par l'exemple

Fabriquer un jeu c'est un peu plus compliqué... les personnages dans un jeu pour enfants sont différents d'un jeu de guerre...

Il nous faut une fabrique par type de jeu...

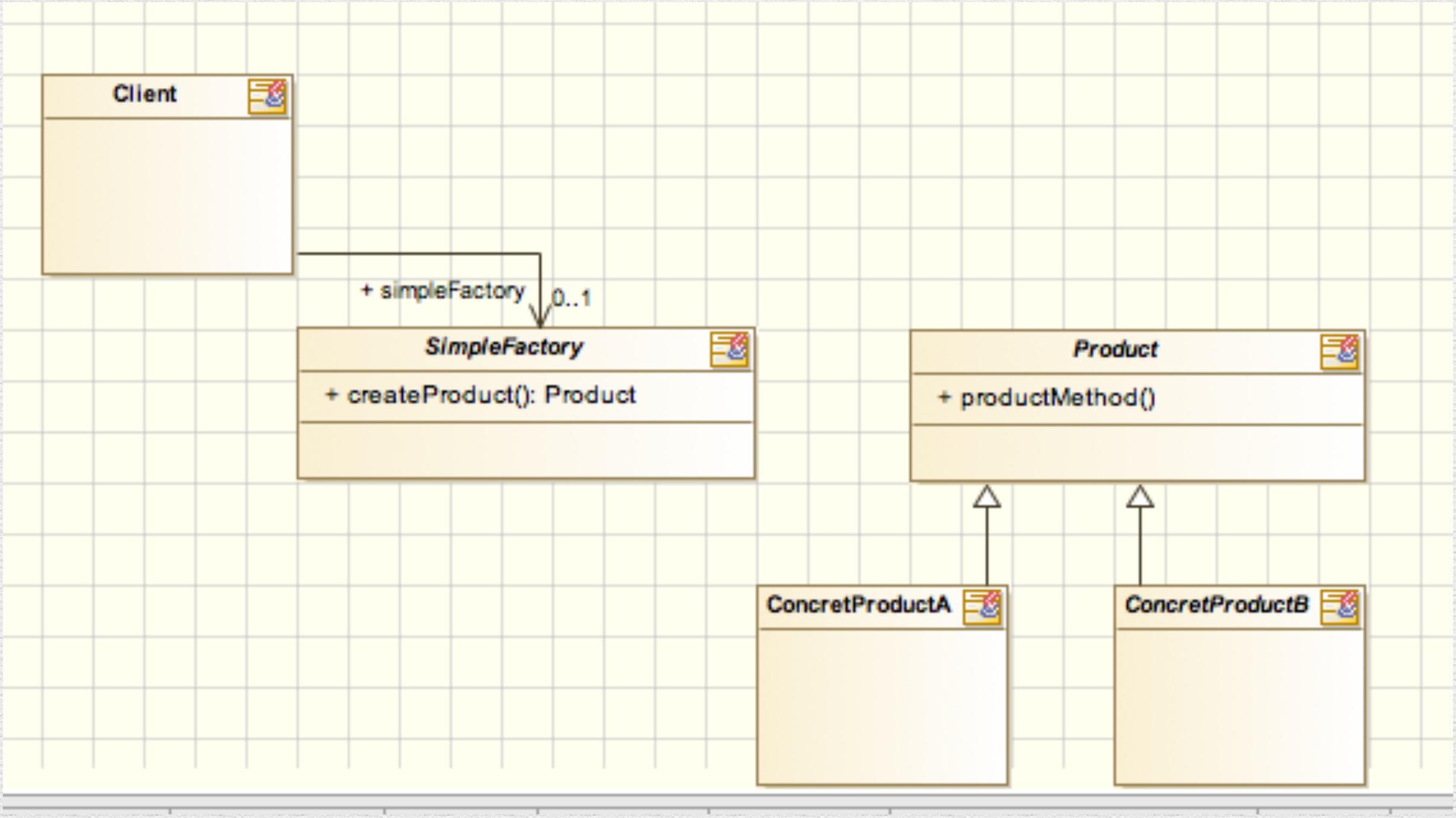
# Simple Factory Pattern : Solution par l'exemple

```
public class FabriquePersonnages {

   Personnage creerPersonnage(String s){
      Personnage principal;
      if (s.equals("Monstre"))
         principal = new Monstre();
      else
         if (s.equals("Guerrier"))
            principal = new Guerrier();
         else
            principal =  new Improbable();
      return principal;
   }

}
```

```
Class<?> classe = Class.forName("dpSimpleFactory."+ s);
Personnage secondaire = (Personnage) classe.newInstance();
```

```
public class Jeu {

   ArrayList<Personnage>personnages = new ArrayList<Personnage>();

   FabriquePersonnages fabrique = new FabriquePersonnages();
   public Jeu(TypeJeu typeJeu) throws Exception{
      Personnage principal;
      if (typeJeu.equals(TypeJeu.guerre))
         principal = fabrique.creerPersonnage("Guerrier");
      else
         if (typeJeu.equals(TypeJeu.horreur))
            principal =  fabrique.creerPersonnage("Monstre");
         else throw new Exception("Jeu inconnu");
      personnages.add(principal);
   }
```

# On résume …


Client
+ simpleFactory
0..1
SimpleFactory
+ createProduct(): Product
Product
+ productMethod()
ConcretProductA
ConcretProductB

# Autre exemple

```
class Pizza {

    String name;

   public String getName() {
      return name; }

   public void prepare() {
   }

   public void box() {
   }

   public void cut() {
   }

   public void bake() {
   }
}

class CheesePizza extends Pizza {}
class GreekPizza extends Pizza {}
class PepperoniPizza extends Pizza
{}
```

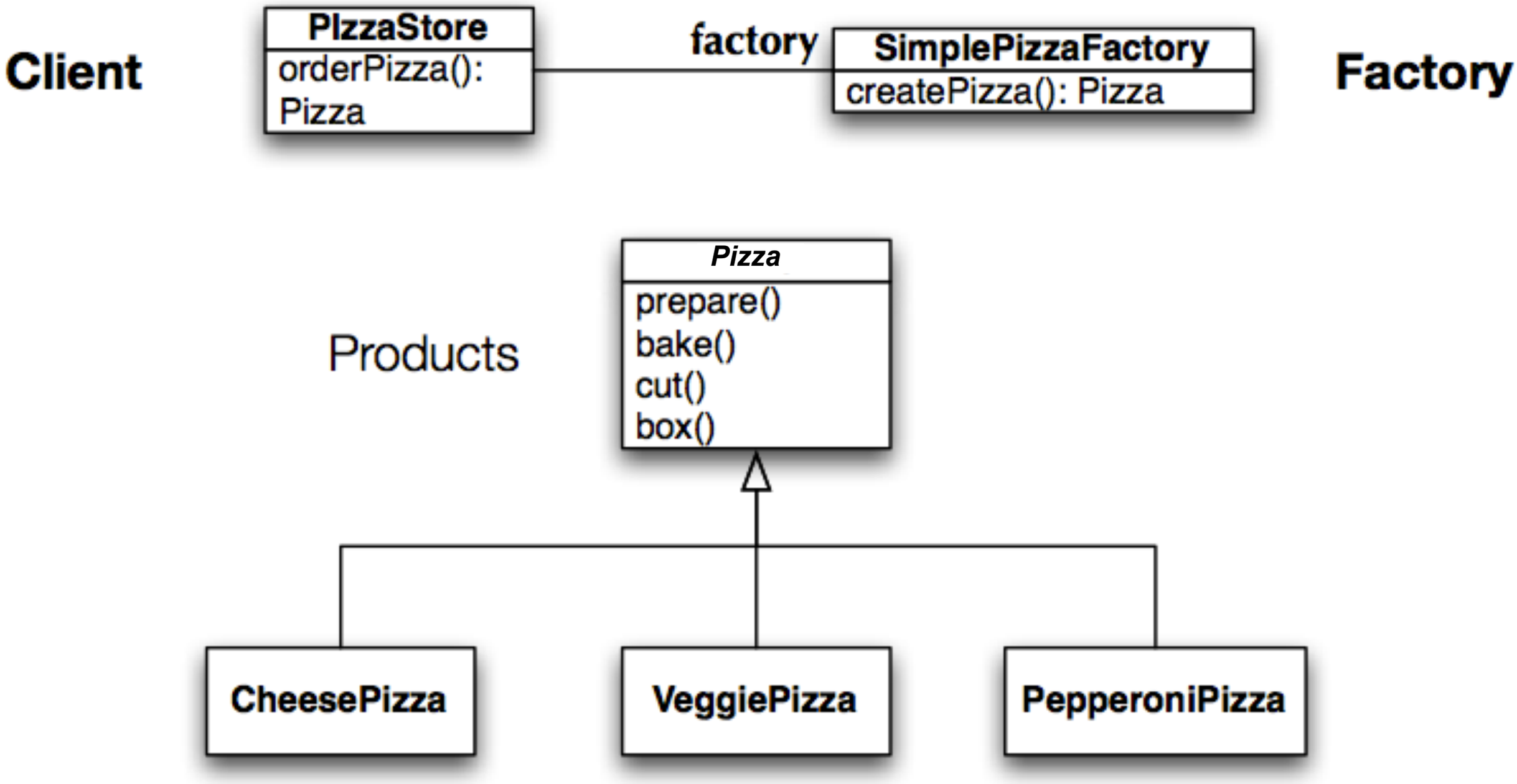
```
package dpFactory;

public class PizzaStore {

   public Pizza orderPizza (String type) {
      Pizza pizza = null;

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

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

      return pizza;
   }
}
```

```
package dpFactory;

public class PizzaStore {
      PizzaFactory factory;

       public PizzaStore(PizzaFactory factory) {
           this.factory =  factory;
       }

       public Pizza orderPizza(String type) {
           Pizza pizza =  factory.createPizza(type);
           pizza.prepare();
           pizza.bake();
           pizza.cut();
           return pizza;
       }
}
```

```
public interface PizzaFactory {

   Pizza createPizza(String type);
}
```

```
public class SimplePizzaFactory
implements PizzaFactory {

    public Pizza createPizza(String type)
{
        Pizza pizza = null;
        if("cheese".equals(type)) {
            pizza = new CheesePizza();
        } else if("onion".equals(type)) {
            pizza = new OnionPizza();
        }
        return pizza;
```

# Class Diagram of New Solution




While this is nice, it is not as flexible as it can be: to increase flexibility we need to look at two design patterns: Factory Method and Abstract Factory

# Design Pattern «Factory Method»

# Des franchises de PizzaStore

# A couple of the concrete product classes

```
public class NYStylePepperoniPizza extends Pizza {

    public NYStylePepperoniPizza() {
        name = "NY Style Pepperoni Pizza";
        dough = "Thin Crust Dough";
        sauce = "Marinara Sauce";

        toppings.add("Grated Reggiano Cheese");
        toppings.add("Sliced Pepperoni");
        toppings.add("Garlic");
        toppings.add("Onion");
        toppings.add("Mushrooms");
        toppings.add("Red Pepper");
    }
}
```


```
public class ChicagoStylePepperoniPizza extends Pizza {

    public ChicagoStylePepperoniPizza() {
        name = "Chicago Style Pepperoni Pizza";
        dough = "Extra Thick Crust Dough";
        sauce = "Plum Tomato Sauce";

        toppings.add("Shredded Mozzarella Cheese");
        toppings.add("Black Olives");
        toppings.add("Spinach");
        toppings.add("Eggplant");
        toppings.add("Sliced Pepperoni");
    }

    void cut() {
        System.out.println("Cutting the pizza into square slices");
    }
}
```

A NY ou à Chicago on ne fait pas les même pizzas au Fromage, au Pepperoni, ..!

```
public class NewYorkPizzaFactory implements PizzaFactory {

    public Pizza createPizza(String type) {
        Pizza pizza = null;
        if (type.equalsIgnoreCase("cheese")) {
            pizza = new NYStyleCheesePizza();
        } else if (type.equalsIgnoreCase("veggie")) {
            pizza = new NYStyleVeggiePizza();
        }
        return pizza;
    }
}
```

Concret Factory Method

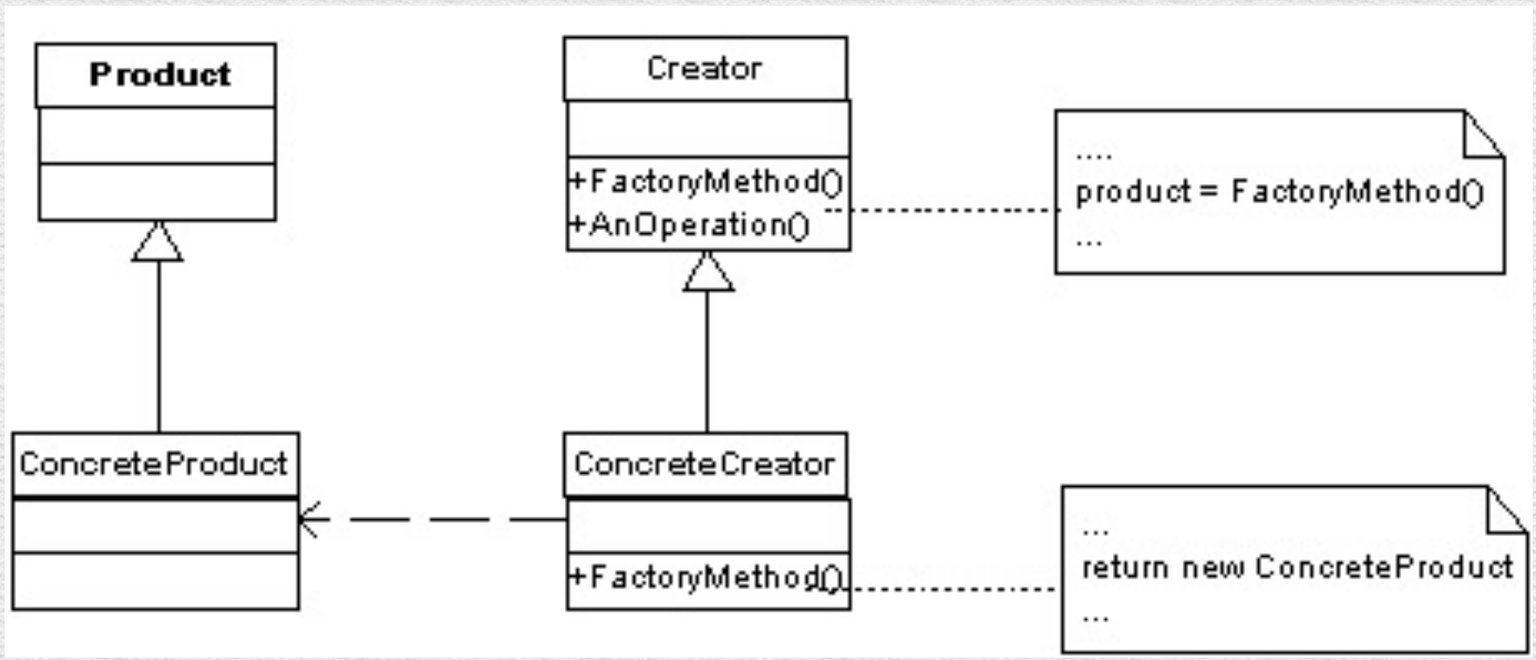
pepperoni .....NYStylePepperoniPizza

# Factory Method Pattern : en résumé le problème

✓Une application doit créer des objets sans connaître les classes de mises en oeuvre de ces objets.

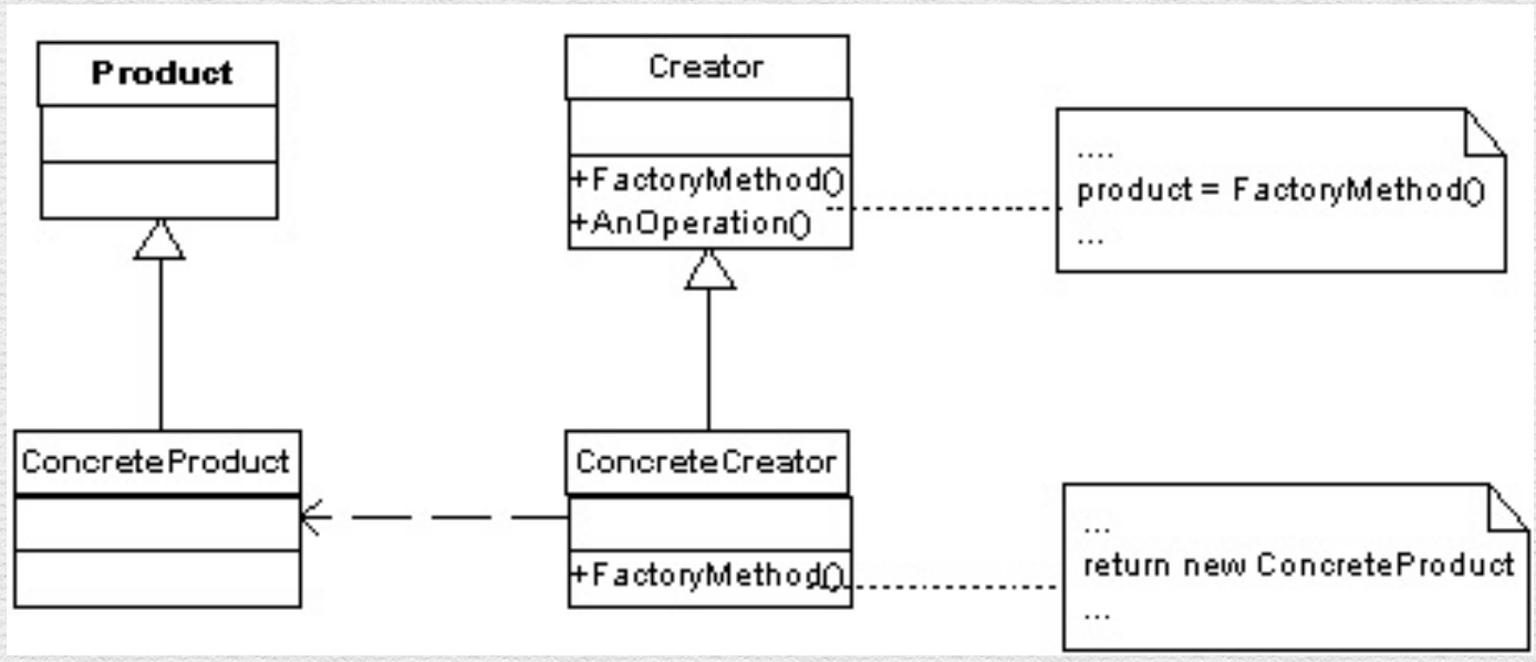
# Factory Pattern : la solution

- ✓Définir une interface pour créer des objets, mais laisser les sous-classes décider quelle classe instancier.
- ✓Tous les objets créés se conforment à la même interface

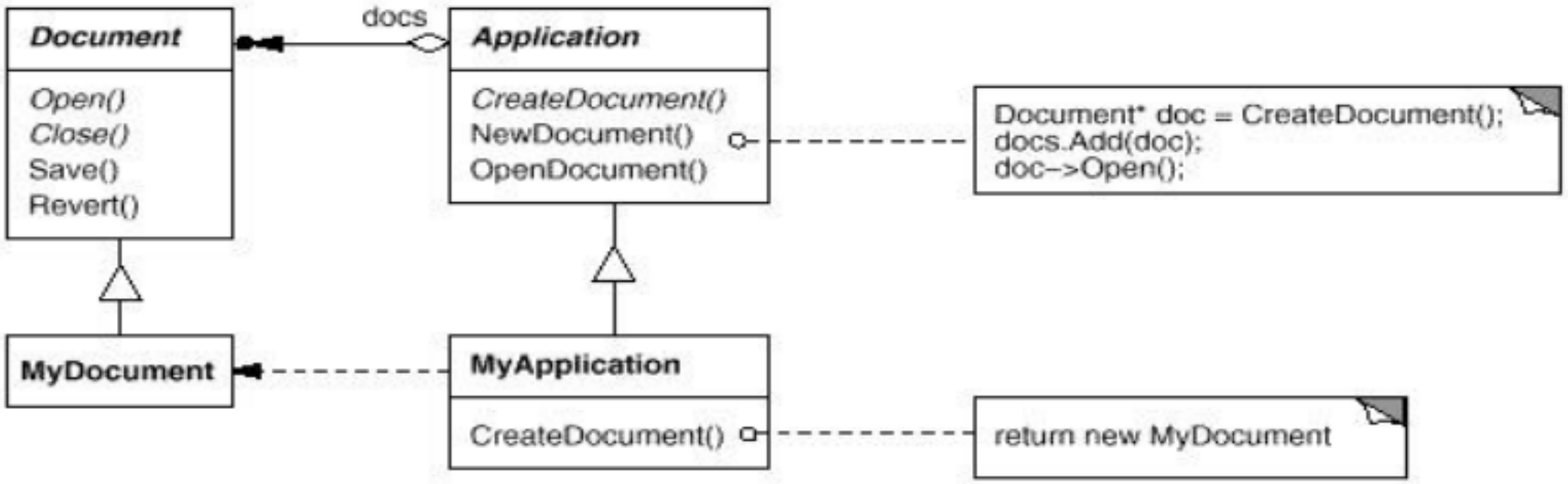


# Factory Pattern : les rôles

- ✓«Produit» définit l'interface des objets créés par la méthode de création.
- ✓Les «Produits concrets» implémentent l'interface «Produit».
- ✓La «fabrique (creator)» déclare la méthode de création qui retourne des objets «Produit».
- ✓Les «fabriques concrètes» surchargent les méthodes pour créer des «Produits concrets».



# Factory Method Pattern : un exemple



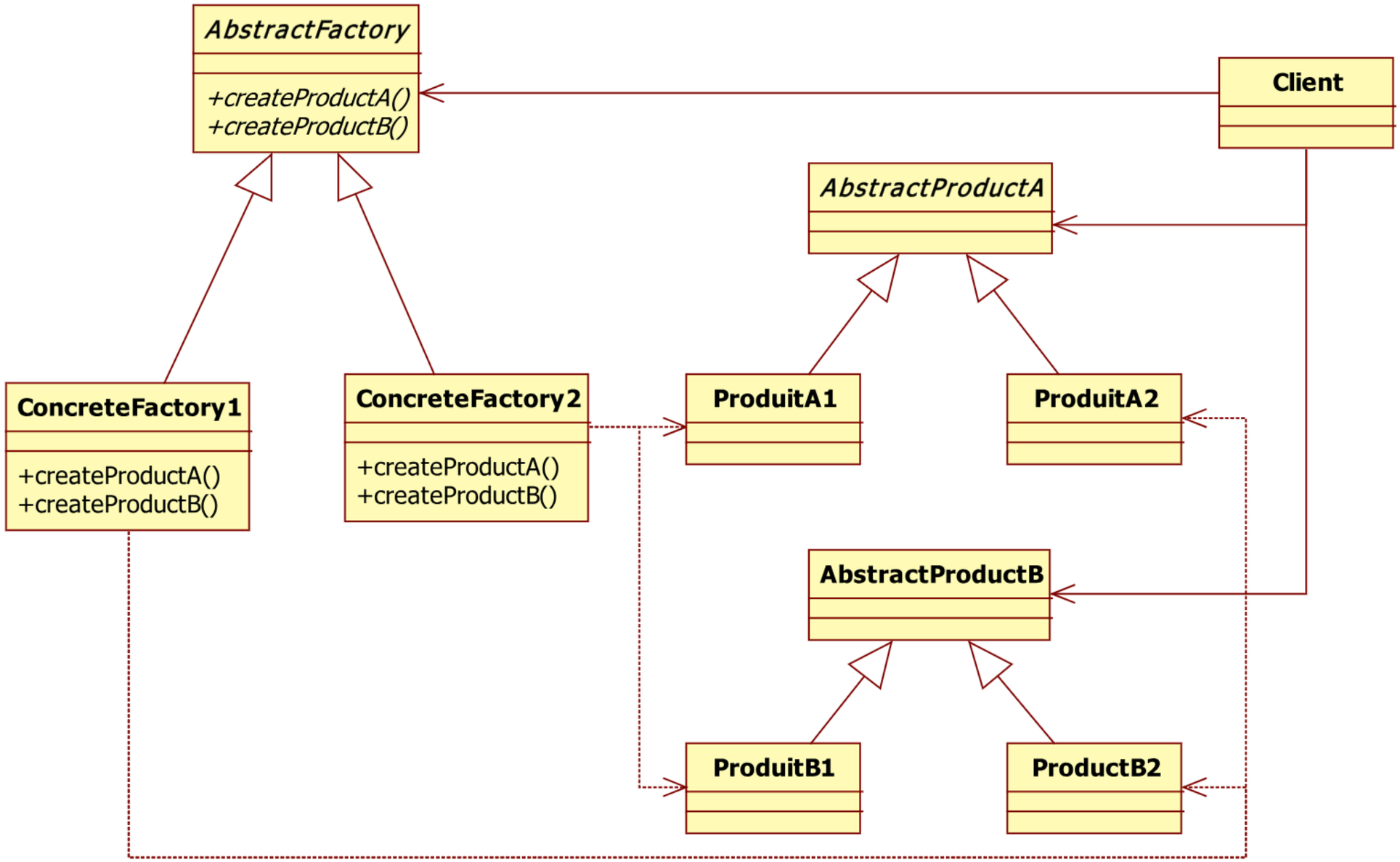

Quelle est la méthode «factory» ?

http://userpages.umbc.edu/~tarr/dp/lectures/Factory.pdf

# Design Pattern «Abstract Factory»

# Abstract factory

## Static diagram :




http://www.goprod.bouhours.net/?page=pattern&pat_id=10

# Moving On

- The factory method approach to the pizza store is a big success, allowing our company to create multiple franchises across the country quickly and easily
  - But (bad news): we have learned that some of the franchises
    - while following our procedures (the abstract code in PizzaStore forces them to)
    - are skimping on ingredients in order to lower costs and increase margins
- Our company's success has always been dependent on the use of fresh, quality ingredients
  - So, something must be done!

    - Mais bien sur on veut les laisser choisir le type de fromage, la pâte, etc..

```
public interface Dough {};

public interface Cheese {};

public interface Pepperoni {};


public class ThinCrustDough implements Dough {

    public String toString() { return "This Crust Dough"; }

}


public class ParmesanCheese implements Cheese {

    public String toString() { return "Shredded Parmesan"; }

}

public class SlicedPepperoni implements Pepperoni {

    public String toString() { return "Sliced Pepperoni"; }

}
```

```
// abstract ingredient factory

public interface PizzaIngredientFactory {

    public Dough createDough();

    public Cheese createCheese();

    public Pepperoni createPepperoni();

}


public class NYPizzaIngredientFactory implements PizzaIngredientFactory {

    public Dough createDough() { return new ThinCrustDough(); }

    public Sauce createCheese() { return new ParmesanCheese(); }

    public Pepperoni createPepperoni() { return new SlicedPepperoni(); }

}
```

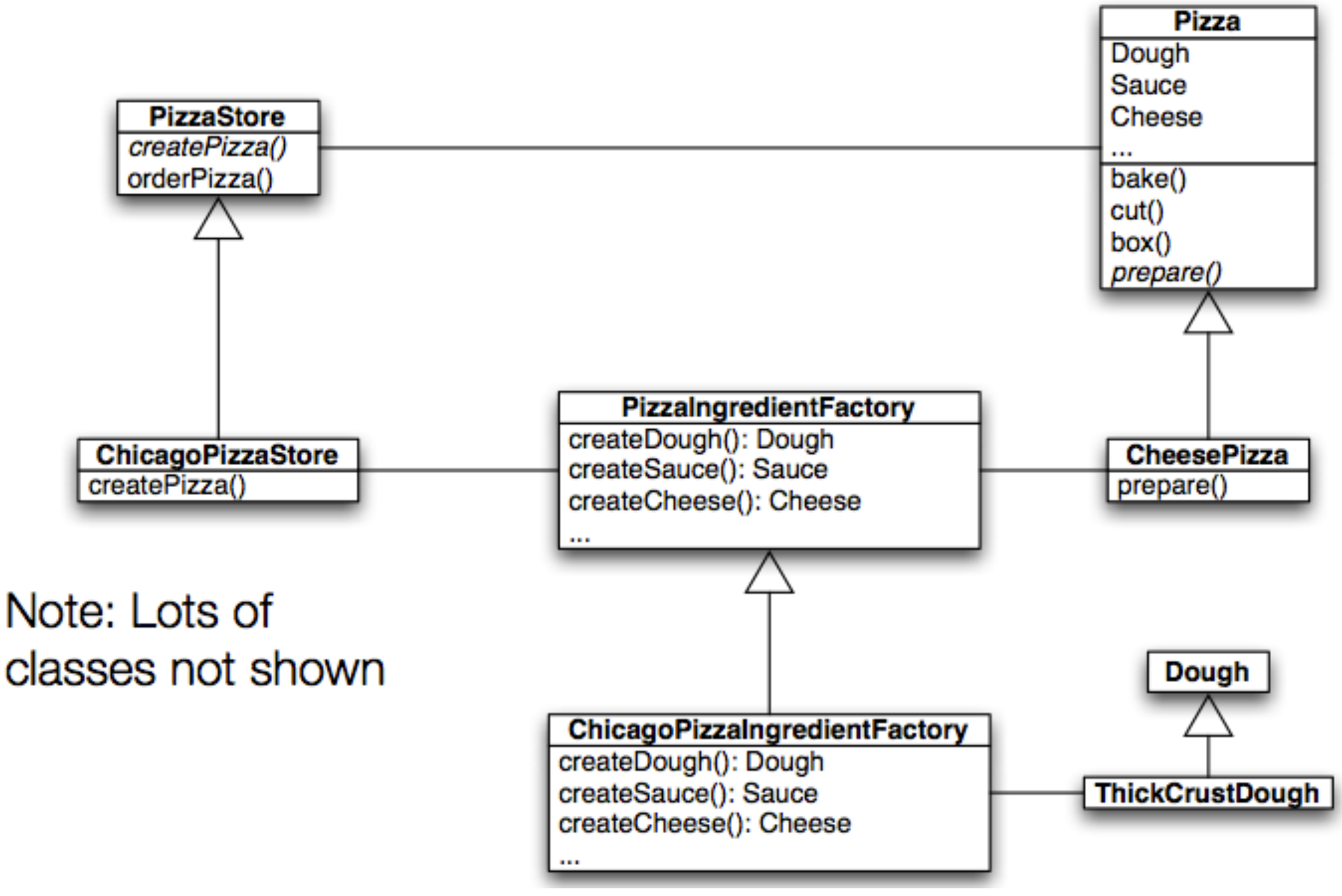
# Within Pizza Subclasses… (I)

```
2  public abstract class Pizza {
3      String name;
4
5      Dough dough;
6      Sauce sauce;
7      Veggie veggies[];
8      Cheese cheese;
9      Pepperoni pepperoni;
.0     Clams clam;
.1
.2     abstract void prepare();
.3
.4     void bake() {
.5         System.out.println("Bake for 25 minutes at 350");
.6     }
.7
.8     void cut() {
```

First, alter the Pizza abstract base class to make the prepare method abstract…

```
public class CheesePizza extends Pizza {

    PizzaIngredientsFactory ingredientFactory;

    public CheesePizza(PizzaIngredientsFactory ingredientFactory){

        this.ingredientFactory = ingredientFactory;

    }

    @Override
    public void prepare() {

        System.out.println("Preparing " + name);

        dough = ingredientFactory.createDough();

        sauce = ingredientFactory.createSauce();

        cheese= ingredientFactory.createCheese();

    }
}
```

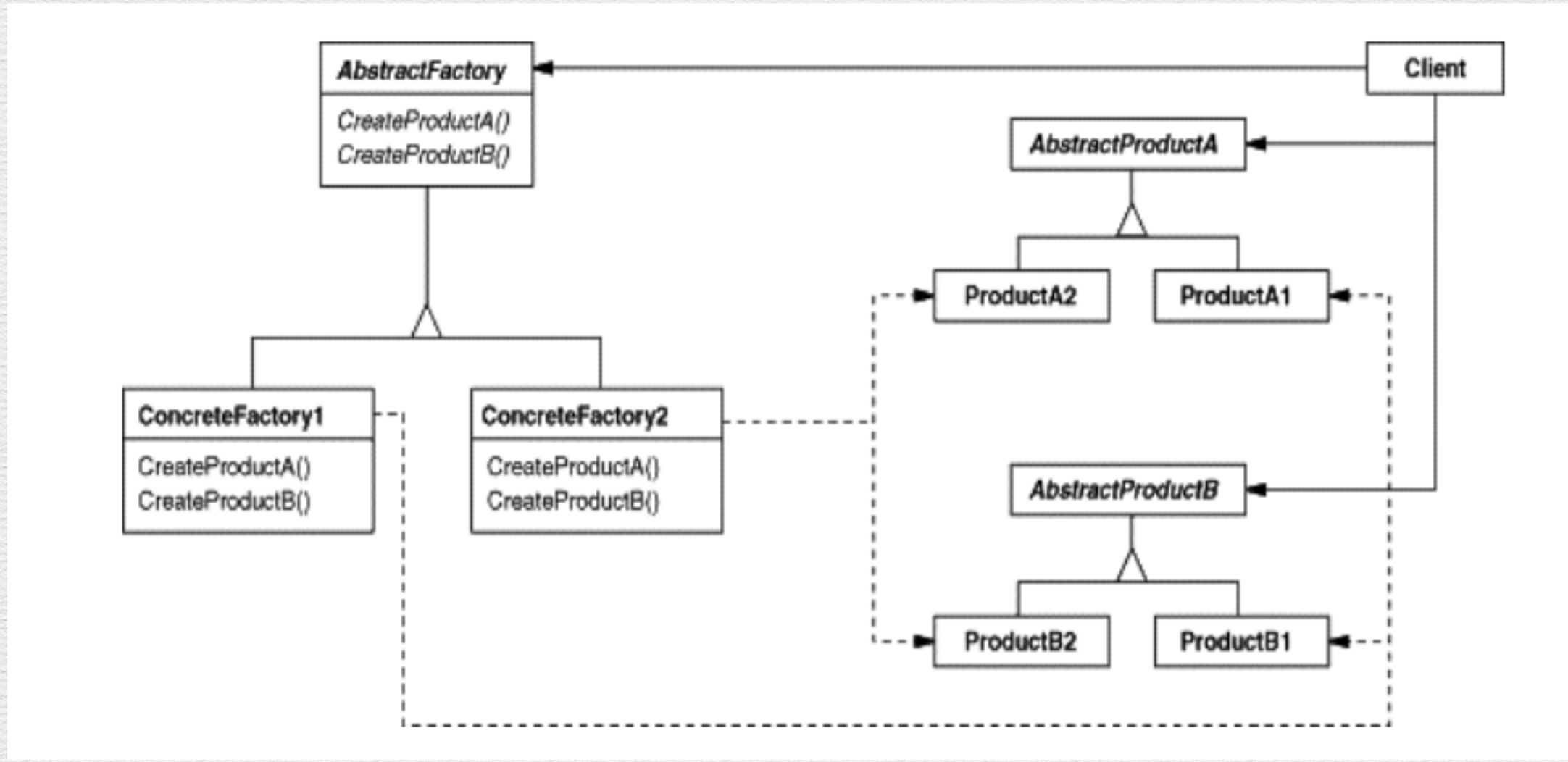
# Class Diagram of Abstract Factory Solution



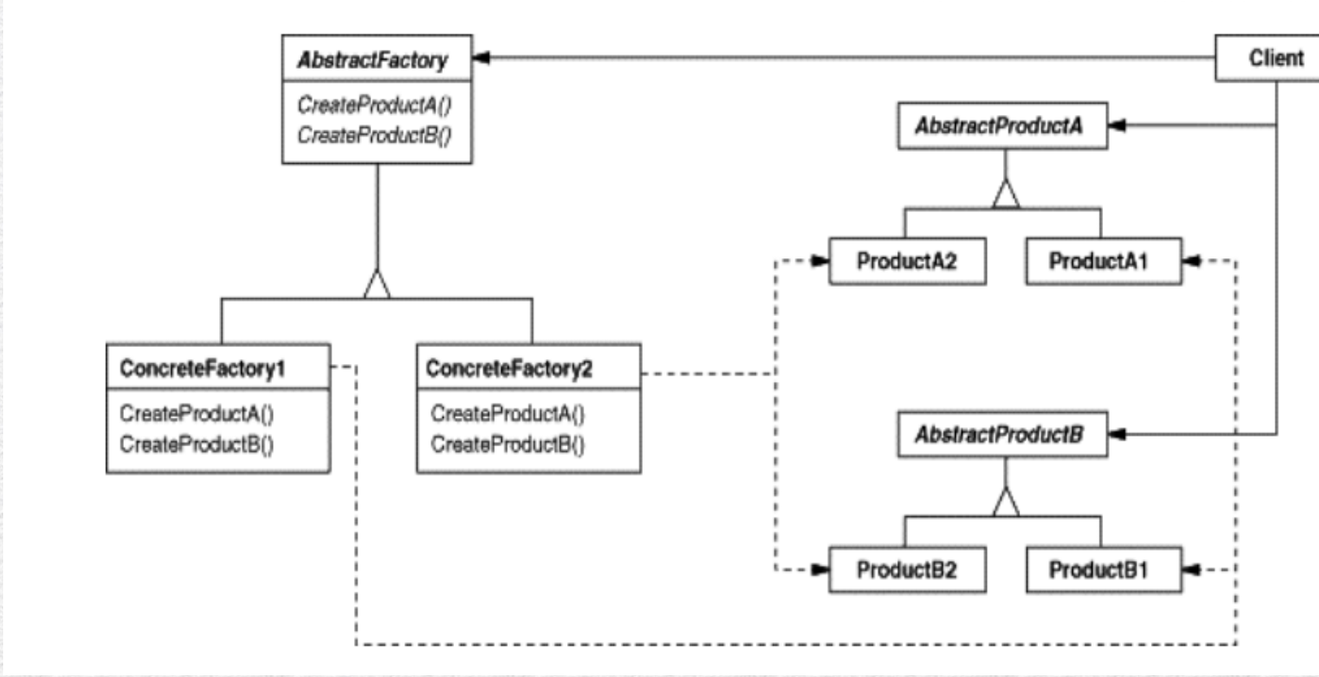
# Abstract Factory Pattern : en résumé le problème

✓Une application doit créer des familles d'objets dépendants sans spécifier leurs classes concrètes.

# Abstract Factory Pattern : la solution



# Abstract Factory Pattern : les rôles




- ✓AbstractFactory
  - ➡Declares an interface for operations that create abstract product objects
- ✓ConcreteFactory
  - ➡Implements the operations to create concrete product objects
- ✓AbstractProduct
  - ➡Declares an interface for a type of product object
- ✓ConcreteProduct
  - ➡› Defines a product object to be created by the corresponding concrete factory
  - ➡› Implements the AbstractProduct interface
- ✓Client
  - ➡Uses only interfaces declared by AbstractFactory and AbstractProduct classes



http://userpages.umbc.edu/~tarr/dp/lectures/Factory.pdf

# Labyrinthe : une factory method (createMaze)

```
/**
 * MazeGame.
 */
public class MazeGame {
// Create the maze.
public Maze createMaze() {
    Maze maze = new Maze();
    Room r1 = new Room(1);
    Room r2 = new Room(2);
    Door door = new Door(r1, r2);
     maze.addRoom(r1);
     maze.addRoom(r2);
   r1.setSide(MazeGame.North, new Wall());
   r1.setSide(MazeGame.East, door);
   r1.setSide(MazeGame.South, new Wall());
    r1.setSide(MazeGame.West, new Wall());
r2.setSide(MazeGame.North, new Wall());
r2.setSide(MazeGame.East, new Wall());
r2.setSide(MazeGame.South, new Wall());
r2.setSide(MazeGame.West, door);
return maze;
```



http://userpages.umbc.edu/~tarr/dp/lectures/Factory.pdf

```
/**
 * MazeGame.
 */
public class MazeGame {
 // Create the maze.
 public Maze createMaze() {
    Maze maze = new Maze();
    Room r1 = new Room(1);
   Room r2 = new Room(2);
    Door door = new Door(r1, r2);
    maze.addRoom(r1);
    maze.addRoom(r2);
   r1.setSide(MazeGame.North, new Wall());
   r1.setSide(MazeGame.East, door);
   r1.setSide(MazeGame.South, new Wall());
   r1.setSide(MazeGame.West, new Wall());
r2.setSide(MazeGame.North, new Wall());
r2.setSide(MazeGame.East, new Wall());
r2.setSide(MazeGame.South, new Wall());
r2.setSide(MazeGame.West, door);
return maze;
```

# Labyrinthe ... magique avec des pièces enchantées??



http://userpages.umbc.edu/~tarr/dp/lectures/Factory.pdf

# Factory Methods pour un Labyrinthe

```
//**
 * MazeGame with a factory methods.
 */
public class MazeGame {
    public Maze makeMaze() {
     return new Maze();}
public Room makeRoom(int n) {
    return new Room(n);}
public Wall makeWall() {
    return new Wall();}
public Door makeDoor(Room r1, Room r2){
       return new Door(r1, r2);}
```



http://userpages.umbc.edu/~tarr/dp/lectures/Factory.pdf

# Factory Methods pour un Labyrinthe & usage

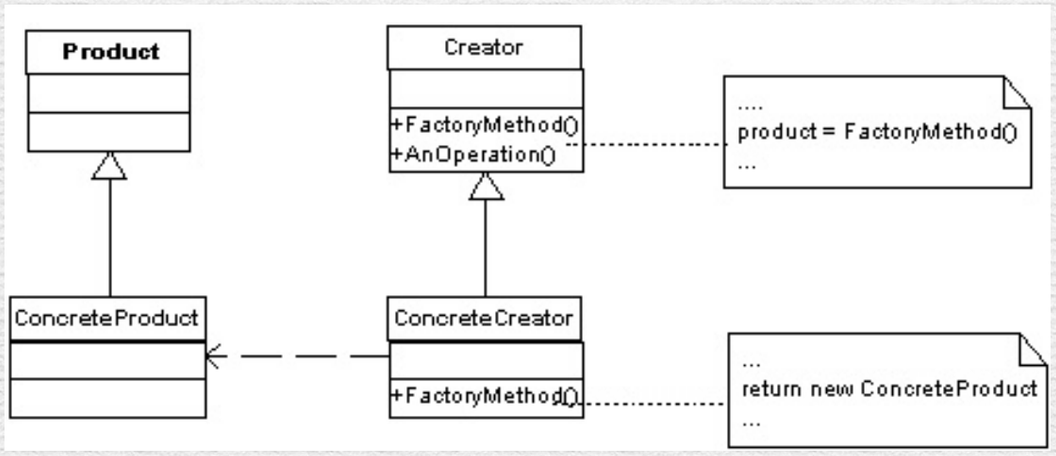
```
public Maze createMaze() {
  Maze maze = makeMaze();
  Room r1 = makeRoom(1);
  Room r2 = makeRoom(2);
  Door door = makeDoor(r1, r2);
  maze.addRoom(r1);
 maze.addRoom(r2);
 r1.setSide(MazeGame.North, makeWall());
r1.setSide(MazeGame.East, door);
r1.setSide(MazeGame.South, makeWall());
r1.setSide(MazeGame.West, makeWall());
r2.setSide(MazeGame.North, makeWall());
r2.setSide(MazeGame.East, makeWall());
r2.setSide(MazeGame.South, makeWall());
r2.setSide(MazeGame.West, door);
return maze;
}
}
```



http://userpages.umbc.edu/~tarr/dp/lectures/Factory.pdf

```
public class EnchantedMazeGame extends MazeGame {
    public Room makeRoom(int n) {
      return new EnchantedRoom(n);}
    public Wall makeWall() {
      return new EnchantedWall();}
    public Door makeDoor(Room r1, Room r2){
      return new EnchantedDoor(r1, r2);}
}
```

# Retour sur le labyrinthe enchanté




- ✓ Creator => MazeGame
- ✓ ConcreteCreator => EnchantedMazeGame (MazeGame is also a ConcreteCreator)
- ✓ Product => MapSite
- ✓ ConcreteProduct => Wall, Room, Door, EnchantedWall, EnchantedRoom, EnchantedDoor



http://userpages.umbc.edu/~tarr/dp/lectures/Factory.pdf

```
// MazeFactory.
public class MazeFactory {
  public Maze makeMaze() {
        return new Maze();}
  public Room makeRoom(int n) {
     return new Room(n);}
public Wall makeWall() {
    return new Wall();}
public Door makeDoor(Room r1, Room r2) {
    return new Door(r1, r2);}
}
```

# Pour créer des familles de labyrinthes

MazeFactory class est une collection de factory methods.



http://userpages.umbc.edu/~tarr/dp/lectures/Factory.pdf

```
public class MazeGame {
  public Maze createMaze(MazeFactory factory) {
    Maze maze = factory.makeMaze();
    Room r1 = factory.makeRoom(1);
    Room r2 = factory.makeRoom(2);
    Door door = factory.makeDoor(r1, r2);
    maze.addRoom(r1);
    maze.addRoom(r2);
    r1.setSide(MazeGame.North, factory.makeWall());
    r1.setSide(MazeGame.East, door);
....
```

# Pour créer des familles de labyrinthes

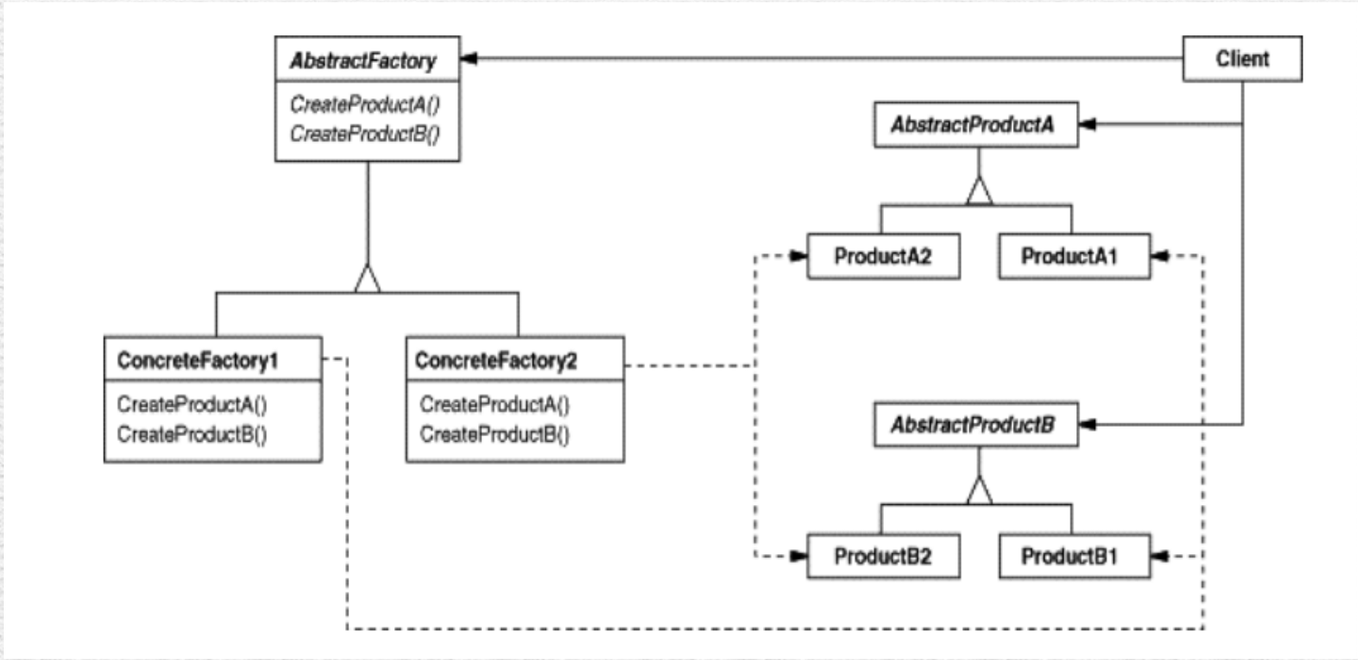
**createMaze** délégue la responsabilité de créer des labyrinthes à la factory.
**MazeGame** se focalise alors sur le jeu plus sur la construction des objets.



http://userpages.umbc.edu/~tarr/dp/lectures/Factory.pdf

# Pour créer des familles de labyrinthes

```
public class EnchantedMazeFactory extends MazeFactory {
  public Room makeRoom(int n) {
        return new EnchantedRoom(n);}
  public Wall makeWall() {
        return new EnchantedWall();}
  public Door makeDoor(Room r1, Room r2){
       return new EnchantedDoor(r1, r2);}
  }
....
```



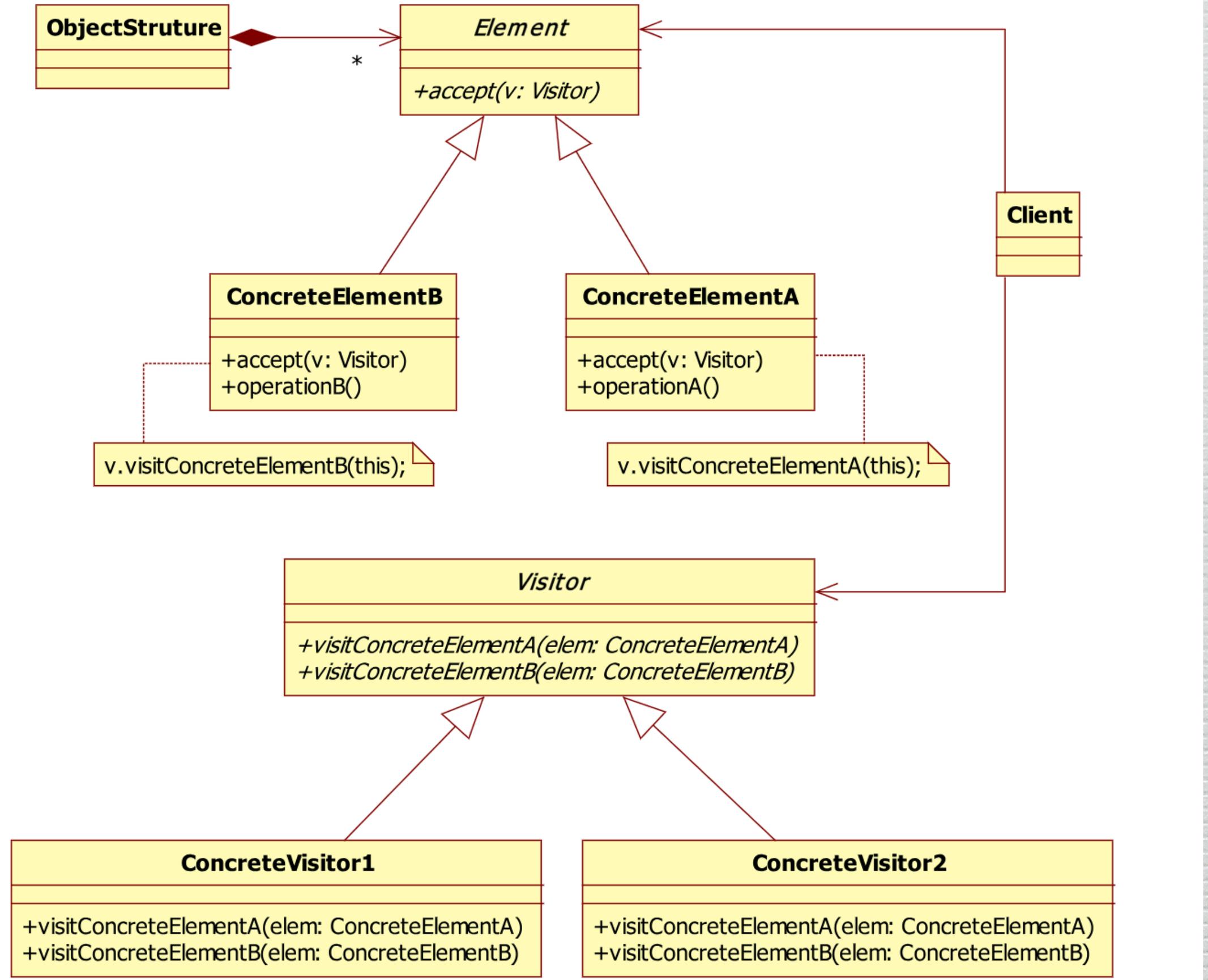

- ✓ AbstractFactory => MazeFactory
- ✓ ConcreteFactory => EnchantedMazeFactory (MazeFactory is also a ConcreteFactory)
- ✓ AbstractProduct => MapSite
- ✓ ConcreteProduct => Wall, Room, Door, EnchantedWall, EnchantedRoom, EnchantedDoor
- ✓ Client => MazeGame

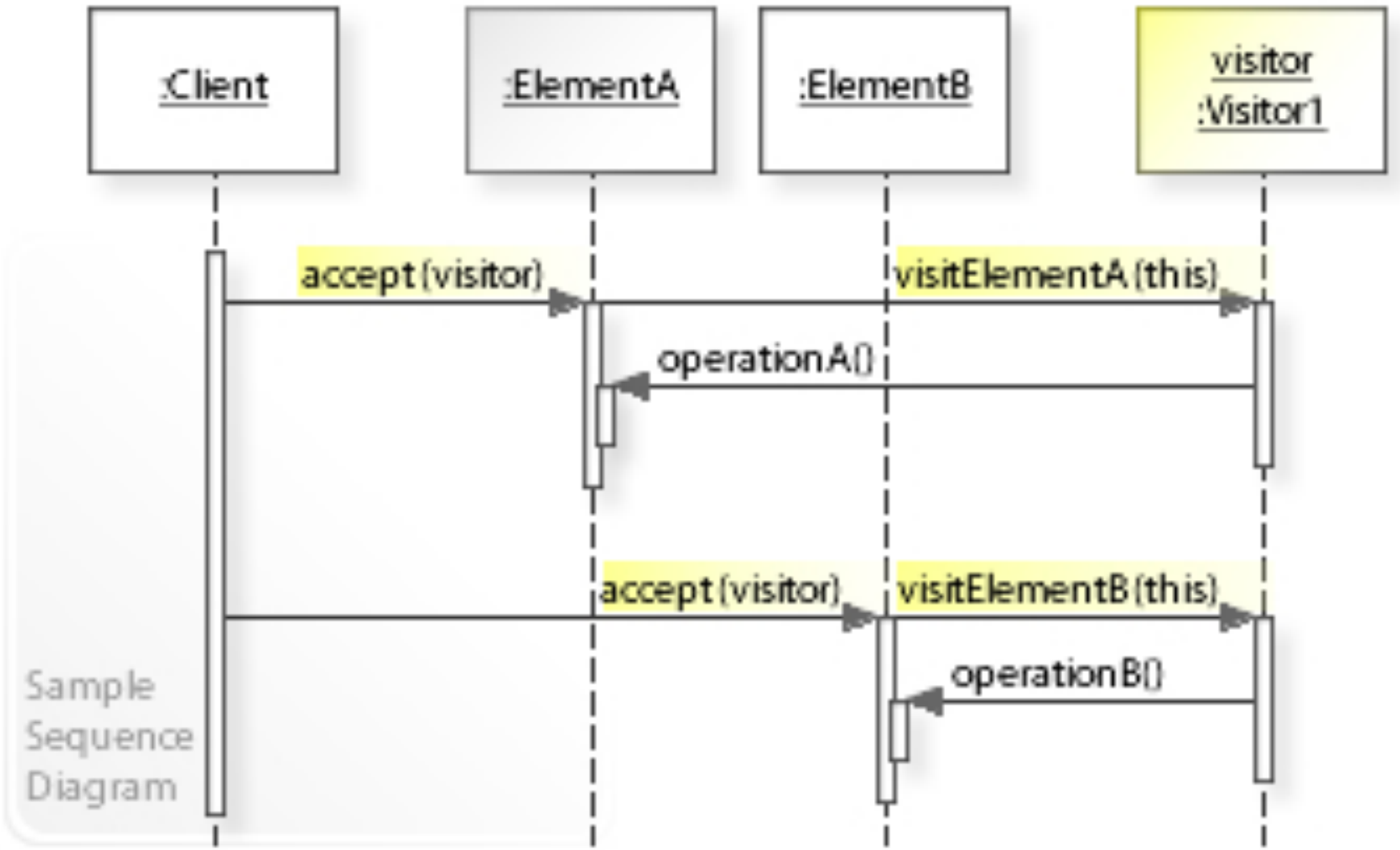


http://userpages.umbc.edu/~tarr/dp/lectures/Factory.pdf

# Design Pattern
# Visitor

https://www.journaldev.com/1769/visitor-design-pattern-java


ObjectStruture
*
Element
+accept(v: Visitor)
Client
ConcreteElementB
+accept(v: Visitor)
+operationB()
ConcreteElementA
+accept(v: Visitor)
+operationA()
v.visitConcreteElementB(this);
v.visitConcreteElementA(this);
Visitor
+visitConcreteElementA(elem: ConcreteElementA)
+visitConcreteElementB(elem: ConcreteElementB)
ConcreteVisitor1
+visitConcreteElementA(elem: ConcreteElementA)
+visitConcreteElementB(elem: ConcreteElementB)
ConcreteVisitor2
+visitConcreteElementA(elem: ConcreteElementA)
+visitConcreteElementB(elem: ConcreteElementB)




https://upload.wikimedia.org/wikipedia/commons/0/00/W3sDesign_Visitor_Design_Pattern_UML.jpg

# Visitor : DP de comportement

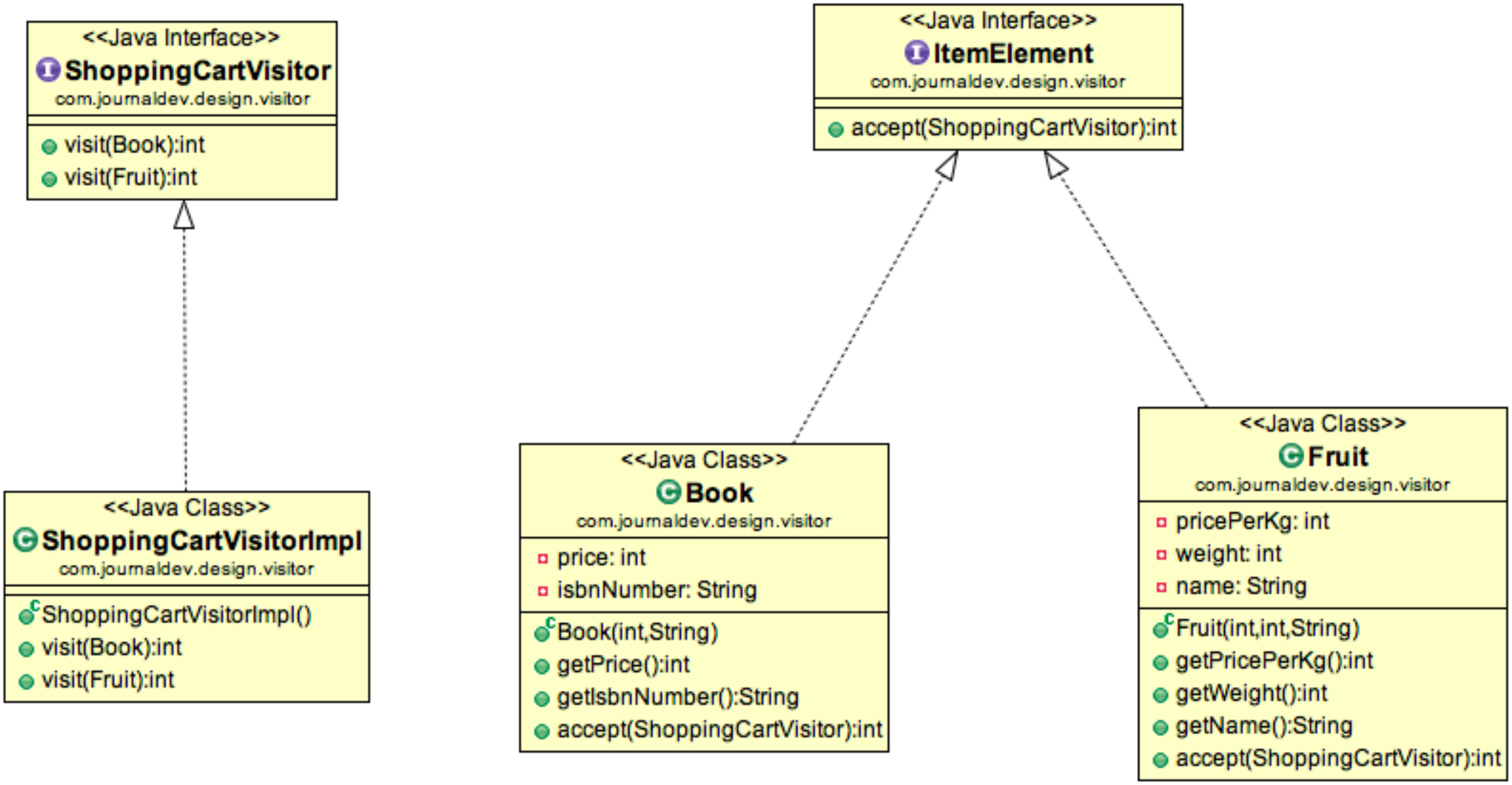
- Représente une opération à exécuter sur les éléments d'une structure d'objet. Le DP Visiteur vous permet de définir une nouvelle opération sans changer les classes des éléments sur lesquels elle opère.

- Utilisez le modèle Visiteur lorsque:

  - une structure d'objet contient de nombreuses classes d'objets avec des interfaces différentes et vous souhaitez effectuer des opérations sur ces objets qui dépendent de leurs classes concrètes.

  - de nombreuses opérations distinctes et non liées doivent être effectuées sur les objets d'une structure d'objet et vous souhaitez éviter de "polluer" leurs classes avec ces opérations.

  - les classes définissant la structure de l'objet changent rarement, mais vous souhaitez souvent définir de nouvelles opérations sur la structure.


<<Java Interface>>
ShoppingCartVisitor
com.journaldev.design.visitor
visit(Book):int
visit(Fruit):int
<<Java Interface>>
ItemElement
com.journaldev.design.visitor
accept(ShoppingCartVisitor):int
<<Java Class>>
ShoppingCartVisitorImpl
com.journaldev.design.visitor
ShoppingCartVisitorImpl()
visit(Book):int
visit(Fruit):int
<<Java Class>>
Book
com.journaldev.design.visitor
price: int
isbnNumber: String
Book(int,String)
getPrice():int
getIsbnNumber():String
accept(ShoppingCartVisitor):int
<<Java Class>>
Fruit
com.journaldev.design.visitor
pricePerKg: int
weight: int
name: String
Fruit(int,int,String)
getPricePerKg():int
getWeight():int
getName():String
accept(ShoppingCartVisitor):int

```
public interface ItemElement {
    public int accept(ShoppingCartVisitor visitor);
}
```

```
public interface ShoppingCartVisitor {
    int visit(Book book);
    int visit(Fruit fruit);
}
```

```
public class ShoppingCartVisitorImpl implements ShoppingCartVisitor {

    @Override
    public int visit(Book book) {
        int cost=0;
        //apply 5$ discount if book price is greater than 50
        if(book.getPrice() > 50){
            cost = book.getPrice()-5;
        }else cost = book.getPrice();
        System.out.println("Book ISBN::"+book.getIsbnNumber() + " cost
="+cost);
        return cost;
    }

    @Override
    public int visit(Fruit fruit) {
        int cost = fruit.getPricePerKg()*fruit.getWeight();
        System.out.println(fruit.getName() + " cost = "+cost);
        return cost;
    }
```

```
public class Book implements ItemElement {

    private int price;
    private String isbnNumber;

    public Book(int cost, String isbn){
        this.price=cost;
        this.isbnNumber=isbn;
    }

    public int getPrice() {
        return price;
    }
    ....
    @Override
    public int accept(ShoppingCartVisitor visitor) {
        return visitor.visit(this);
    }

}
```

```
public class Fruit implements ItemElement {

    private int pricePerKg;
    private int weight;
    private String name;

    public Fruit(int priceKg, int wt, String nm){
        this.pricePerKg=priceKg;
        this.weight=wt;
        this.name = nm;
    }

    public int getPricePerKg() {
        return pricePerKg;
    }
.....
    }

    @Override
    public int accept(ShoppingCartVisitor visitor) {
        return visitor.visit(this);
    }
}
```

```
public class ShoppingCartClient {

   public static void main(String[] args) {
      ItemElement[] items = new ItemElement[]{new Book(20,
"1234"),new Book(100, "5678"),
            new Fruit(10, 2, "Banana"), new Fruit(5, 5, "Apple")};

      int total = calculatePrice(items);
      System.out.println("Total Cost = "+total);
   }

   private static int calculatePrice(ItemElement[] items) {
      ShoppingCartVisitor visitor = new ShoppingCartVisitorImpl();
      int sum=0;
      for(ItemElement item : items){
         sum = sum + item.accept(visitor);
      }
      return sum;
   }
}
```

Et si nous  créons une carte pour le BlackFriday? Ou bien une qui collecte des points en fonction des produits