Design Patterns - TD

TD3 initial code

This is a template for the students' assignments.

Course material: http://bit.ly/innopolis-patterns

Assignment info

LAST NAME
BRUEL

First Name
Jean-Michel

Group #

- [x] Teachers
- [ ] 1
- [ ] 2
- [ ] 3
- [ ] 4
- [ ] Innopolis

Requirements

You'll need:

- [x] A GitHub account
- [ ] A Git Bash terminal (if you use Window$)

Try the following command in your terminal to check your `git` environment:

```
git config --global -l
```

Initial tasks

- [x] Click on the Github Classroom link proided by your teacher (in fact, this should be done if you read this).
- [ ] Clone on your machine the Github project generated by Github Classroom.

- ▢ Modify the README file to add your last name, first name and group number.
- ▢ Commit and push using the following message:

commit/push

```
fix #0 Initial task done
```

In the following, every time you'll see à `fix #…` text, make sure all your files are committed, and then push your modifications in the distant repo, making sure you used the corresponding message (`fix #…`) in one of the `commit` messages.

- If you want to check that you're really ready for `fix #0`, you can run the command in your shell: `make check`.
- If you want to list the ToDos of the day, run `make todos`.

This TD exercise is inspired from the excellent book: "Head First: Design Pattern. Bert Bates, Eric Freeman, Elisabeth Freeman, Kathy Sierra. Editions O'Reilly. 2005."

1. The O'Reilly pizzeria

1.1. Existing application

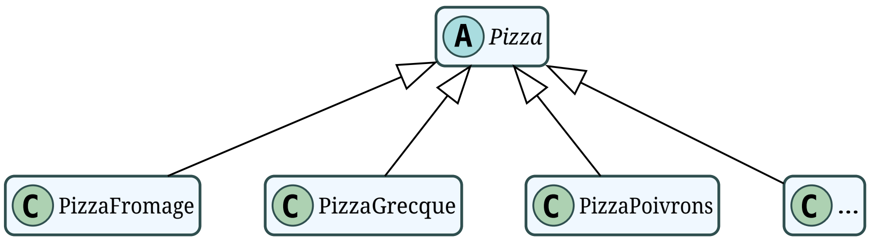
You are hired in a pizzeria to do … computer science! Last year's intern who worked on the code left. You only have at your disposal:

- The following starting code:

`Pizzeria.java` *initial code (source available in your* `src`*)*

```java
/**
 * @author bruel (from O'Reilly Head-First series)
 * @depend - * - Pizza
 */
public class Pizzeria {

        /**
         * @param type
         * @return a Pizza object according to the type
         */
        public Pizza commanderPizza(String type) {

        Pizza pizza;

        if (type.equals("fromage")) {
            pizza = new PizzaFromage();
        } else if (type.equals("grecque")) {
            pizza = new PizzaGrecque();
        } else {
            pizza = new PizzaPoivrons();
        }

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

        return pizza;
        }
}
```

- The following pizza class diagram draft:



- The following piece of test code:

```
Pizzeria boutiqueBrest = new Pizzeria ();
boutiqueBrest.commanderPizza ("cheese");
...
Pizzeria boutiqueStrasbourg = new Pizzeria ();
boutiqueStrasbourg.commanderPizza ("Greek");
```

TODO*:*

- ▢ Identify what varies in this code (if market pressure makes add pizzas to the menu or if a pizza is no longer successful and must disappear, etc.).
- ▢ Isolate this code in a `SimpleFabriqueDePizzas` class.
  - The idea is to replace the `if` in that code by something like `pizza = fabrique.creerPizza(type);`
  - You have inherited from this horrible "if then else". In your implementation, you might want to use a `switch case` and may be an `enum`.
- ▢ Provide the resulting class diagram.
- ▢ What is the advantage of doing this? Don't we transfer just the problem to another object?

commit/push

```
fix #1.1 Simple Pizza Factory
```

2. Almost there…

2.1. Success of the O'Reilly pizzerias: Franchising the pizza stores

Several cities want to open pizzerias like yours. Your boss, very happy with your programs wants to impose on all future pizzerias to use your codes.

The problem: Strasbourg cheese pizzas are different pizzas with Corsican cheeses!

QUESTION

Propose a solution where `SimpleFabriqueDePizzas` would be an abstract class.

commit/push

```
fix #2.1 Use of abstract factories
```

We have end-up with a clean code that is almost a pattern. Often called *Simple Factory*, it is nevertheless not considered as a Design Pattern:

Design pattern : **Simple Factory**

Factory defines an interface for creating an objet, but leaving to its subclasses the choice of the class to be instanciated (see also Abstract Factory).

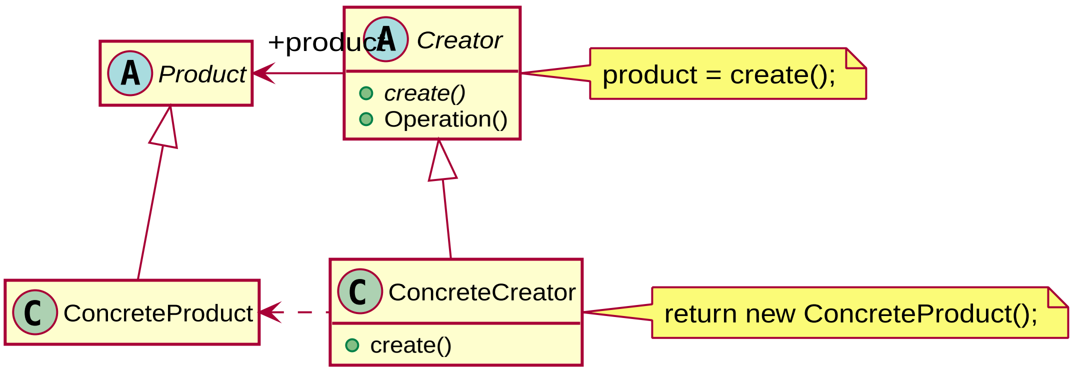


Figure 1. UML model of the Factory pattern

2.2. Problem: everyone works as they see fit!

Pizzerias use your factories well but have changed their procedures: some do not cut the pizzas, change the cooking times, and O'Reilly pizzerias lose their identity. So we need to **restructure** the pizzerias.

A highly paid Italian consultant (fortunately in pizzas!) proposes to return to the following structure:

```
public abstract class Pizzeria {
    public final Pizza commanderPizza(String type) {
        Pizza pizza;

        pizza = creerPizza(type);
        pizza.preparer();
        pizza.cuire();
        pizza.couper();
        pizza.emballer();

        return pizza;
    }

    ....... Pizza creerPizza(String type);
}
```

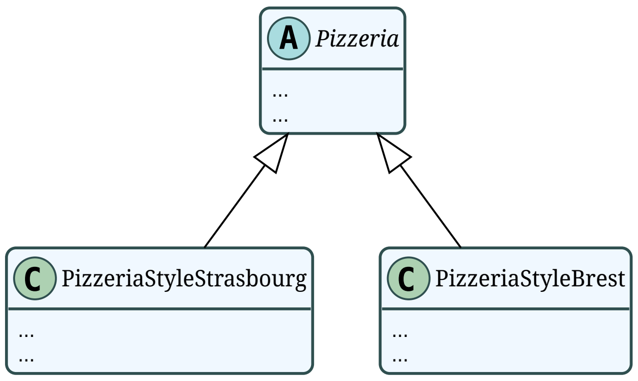
QUESTION

What are the differences with our current design?

2.3. Allowing the subclasses to decide

QUESTION

In the following diagram, place the methods in the right place so that the procedures are followed while having pizza with "regional" variants.



TIP

Each subclass redefines the `creerPizza ()` method, while all the subclasses use the `commanderPizza ()` method defined in `Pizzeria`.

commit/push

```
fix #2.3 Simple Pizza Factory
```

2.4. Declaring a factory method

Just by bringing one or two transformations to `Pizzeria`, we have gone from an object managing the instantiation of our concrete classes to a set of subclasses that now assume this responsibility.

QUESTION

What is the exact declaration of the `creerPizza()` method of the `Pizzeria` class ?

2.5. So, how does it work ?

QUESTION

Give a sequence diagram of a "order a cheese pizza of Strasbourg type".

You will implement the missing class in TP.

commit/push

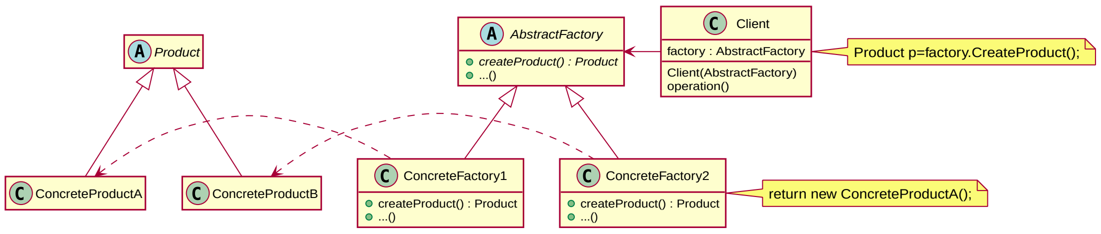
```
fix #2.5 Simple Pizza Factory
```

3. The Abstract Factory pattern

Here we are! We have, in fact, used the Factory pattern:

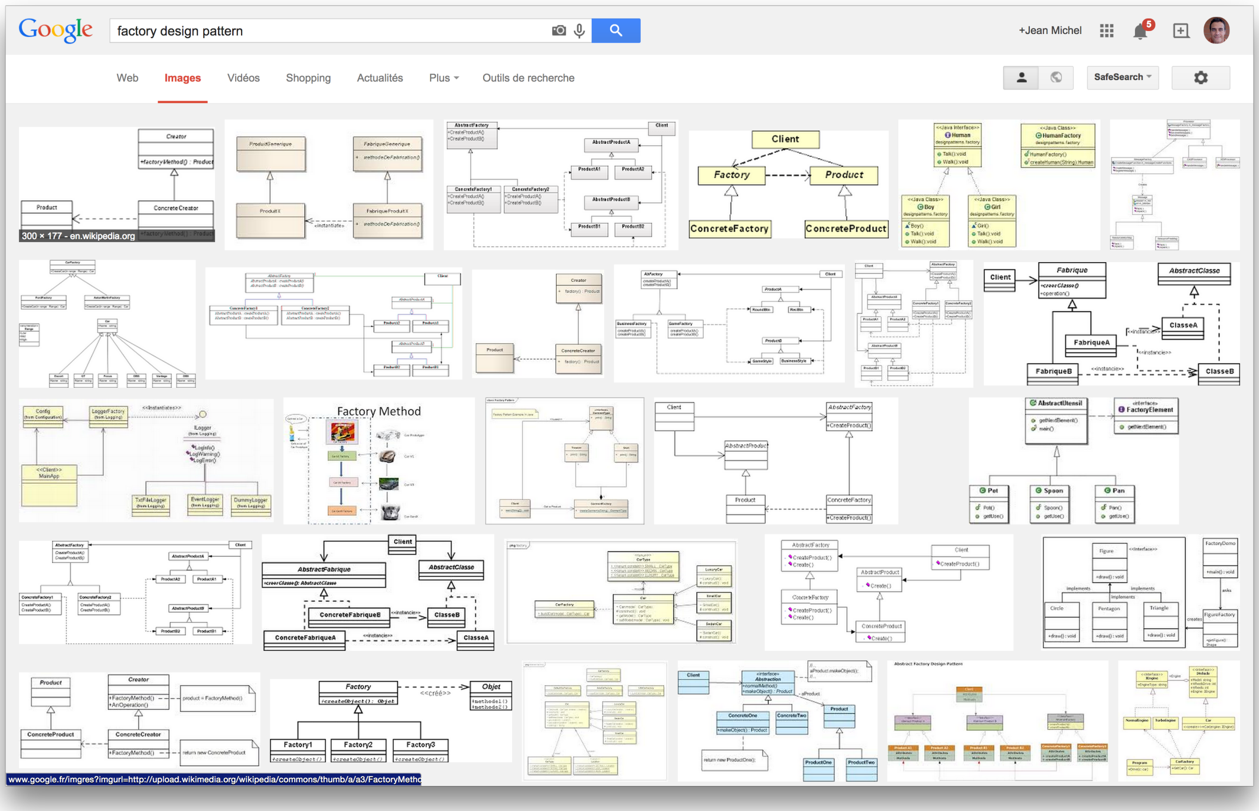
Design pattern : **Factory (simple)**

Abstract Factory defines an interface for creating a familly of objects, without precising their concrete implementation class.



A Product
A AbstractFactory
createProduct() : Product
...()
C Client
factory : AbstractFactory
Client(AbstractFactory)
operation()
Product p=factory.CreateProduct();
C ConcreteProductA
C ConcreteProductB
C ConcreteFactory1
createProduct() : Product
...()
C ConcreteFactory2
createProduct() : Product
...()
return new ConcreteProductA();

Figure 2. UML model of the Abstract Factory pattern



Google
factory design pattern
+Jean Michel
Web
Images
Vidéos
Shopping
Actualités
Plus
Outils de recherche
SafeSearch
300 × 177 - en.wikipedia.org
www.google.fr/imgres?imgurl=http://upload.wikimedia.org/wikipedia/commons/thumb/a/a3/FactoryMethod

Figure 3. Several examples of Factory pattern definitions

4. Translations

Table 1. Translations French/English/Russian of the code vocabulary

FR	US	RU
Grecque	Greek	??
Fromage	Cheese	??

FR	US	RU
Poivron	Pepper	??
preparer	prepare	??
cuire	bake	??
couper	cut	??
emballer	wrap	??
creer	create	??
commander	order	??

5. Contributors

- Jean-Michel Bruel

6. About...