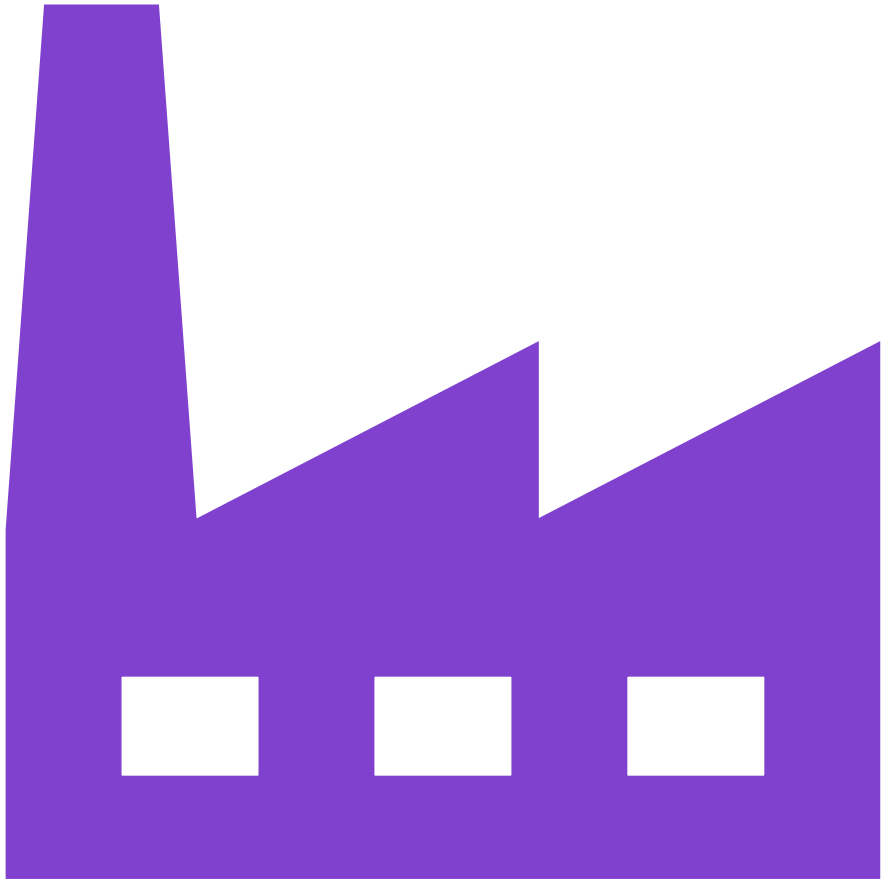


Factory & Abstract Factory Pattern

Fatmi Ylian, Contie Florian,
Castanie Valentin, Bougé Nicolas

Sommaire

Introduction sur les patterns

- Bonnes pratiques résolvant des problèmes de conception récurrents.
- Respecte en partie les règles SOLID.
- « A pattern language » 1977 Christopher Alexander
- « Design Pattern – Element of Reusable Object-Oriented Software » 1994 GoF (Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides)

3 types de Patterns

Créationnels : Instanciation de classe

Structurels : Structurer et s'occuper de la composition des classes

Comportementaux : Gérer la communication d'une classe avec les autres

Le Pattern Factory

Besoin du Pattern Factory

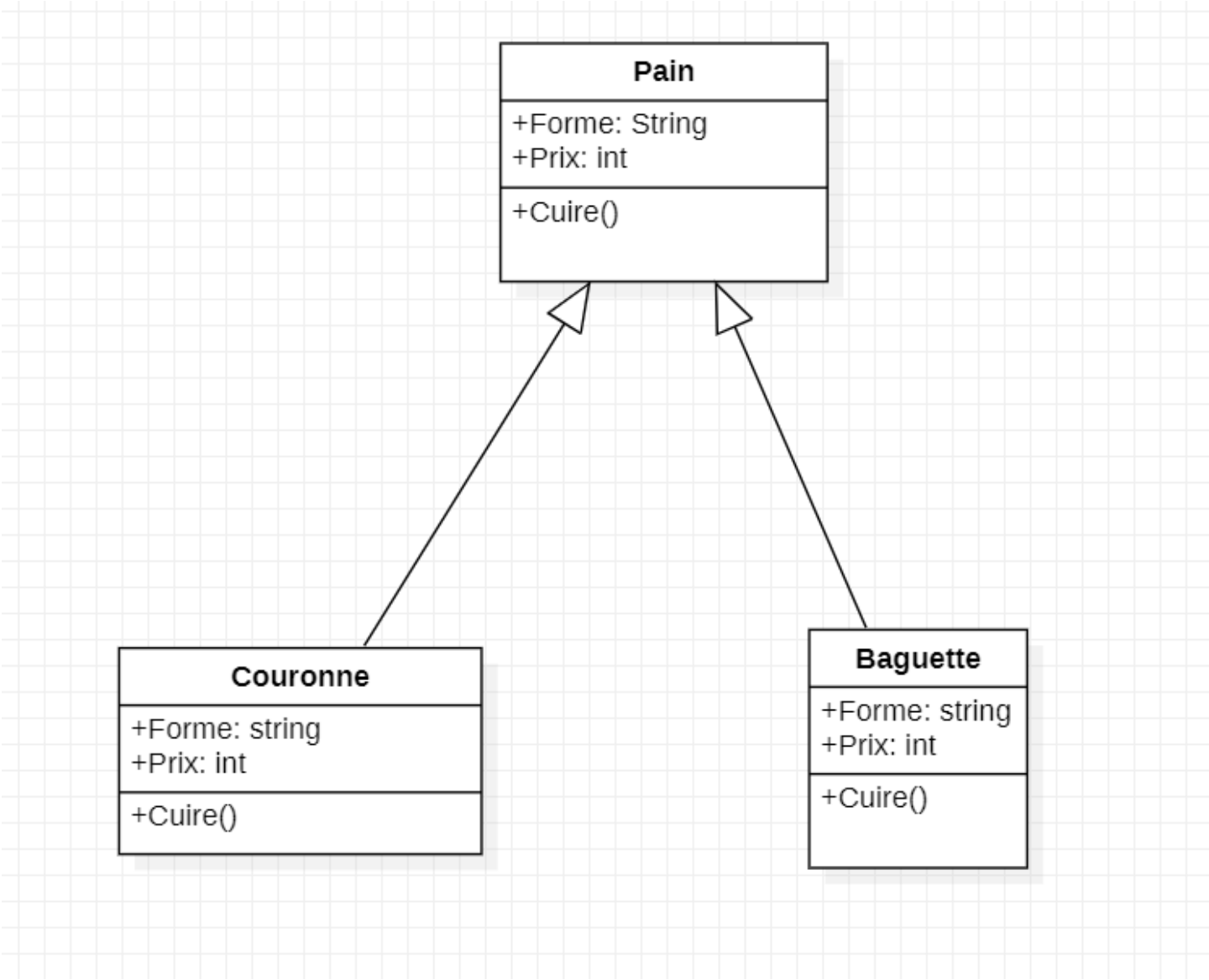
Une boulangerie fabrique et ne vend que des baguettes de pain

La boulangerie veut ajouter un nouveau produit : la tourte

La baguette et la tourte peuvent être cuite

Baguette et Tourte hérite de la classe Pain

Modélisation du besoin



Problème de la modélisation précédente

Comment simplifier la création d'objet ?

Si la boulangerie veut un nouveau type de pain.

Il faut séparer le code de construction et le code qui utilise l'objet.

Modélisation résolvant le problème

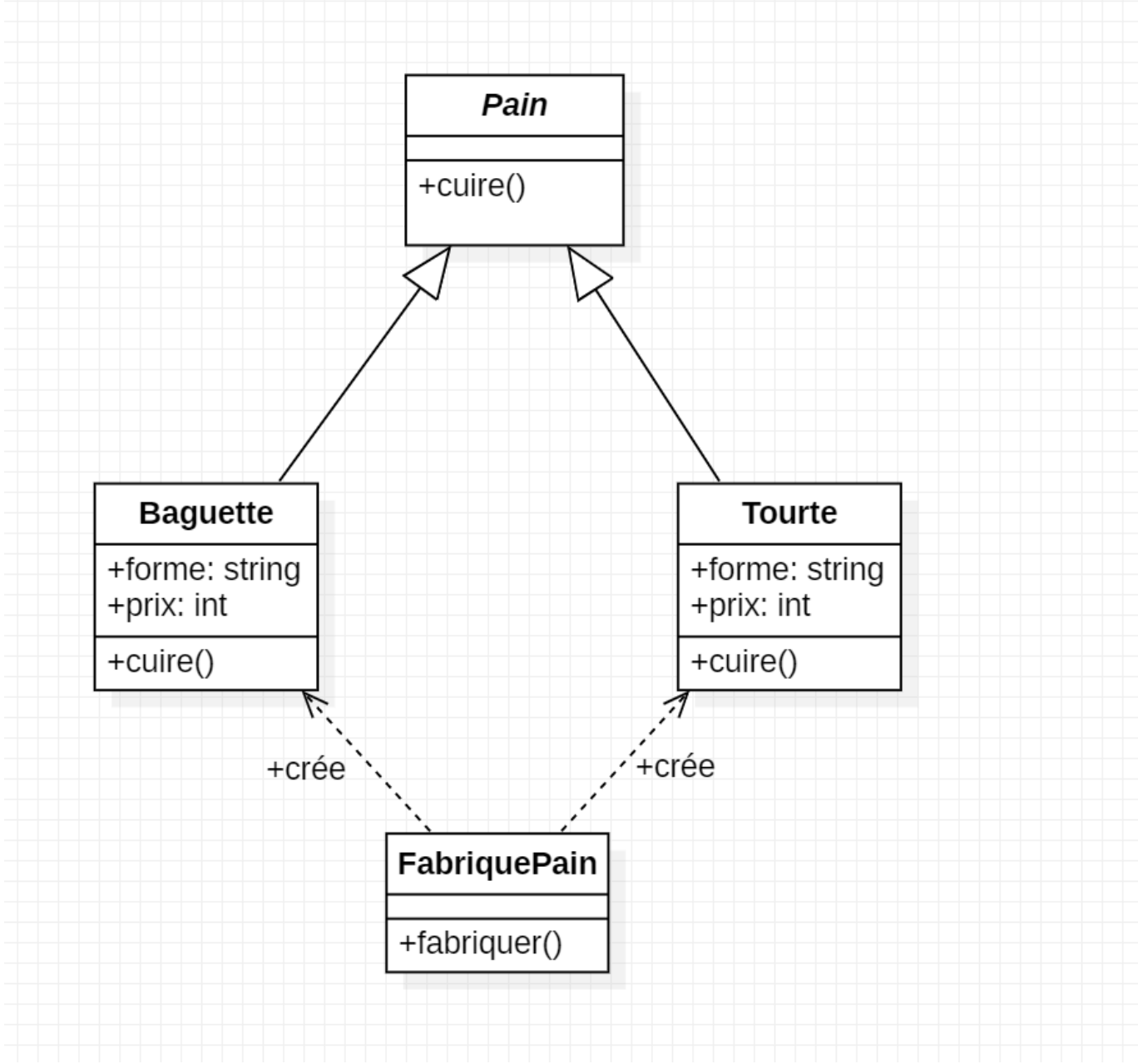
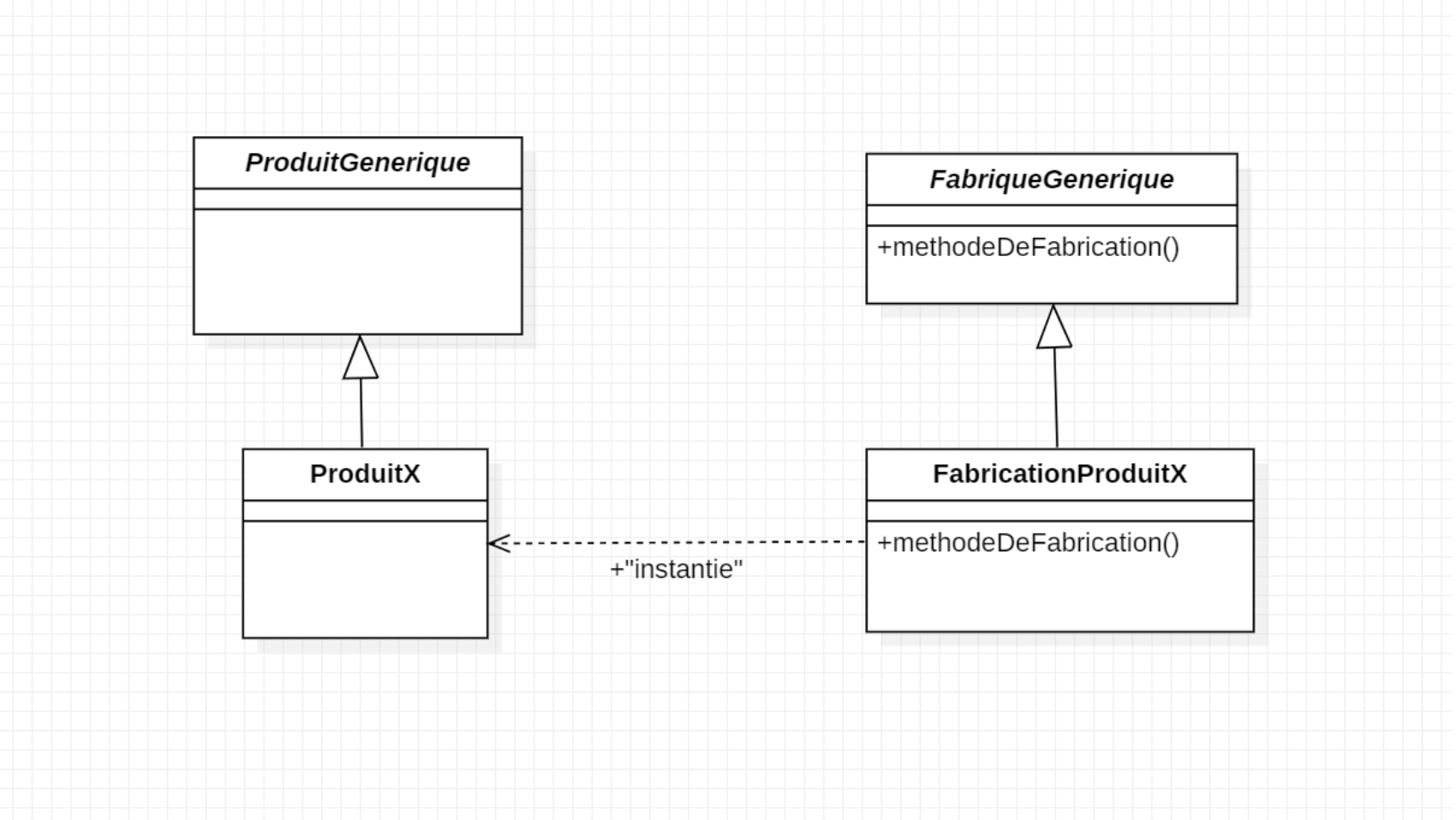
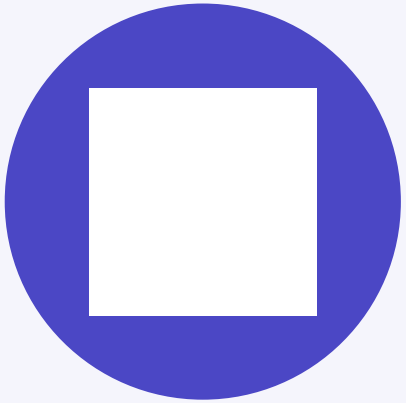


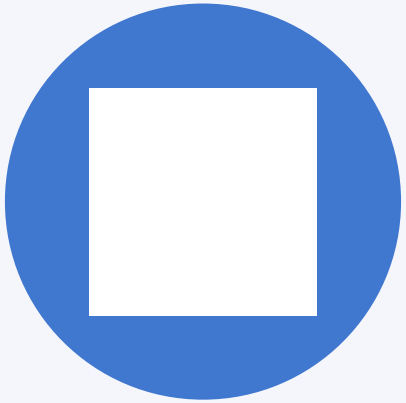
Diagramme générique du Pattern



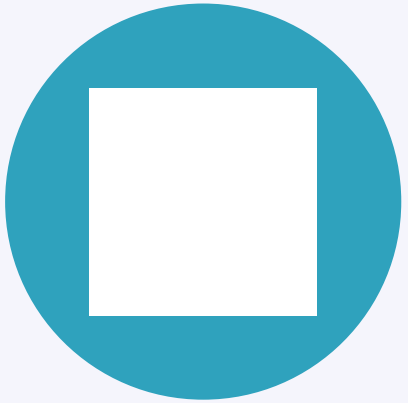
Définition du Pattern Factory



TYPE : CRÉATION

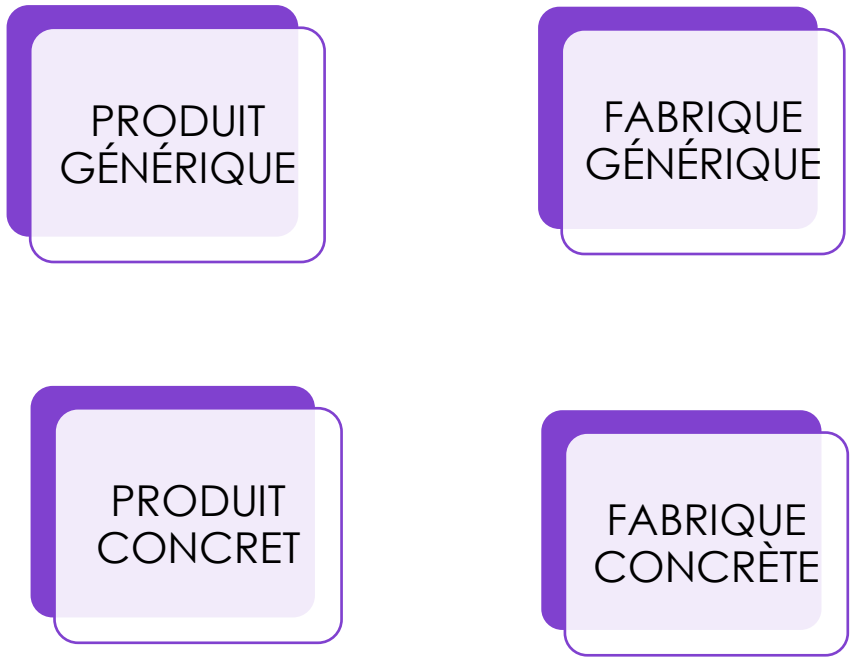


SIMPLIFIER LA CRÉATION D'OBJETS



SÉPARER LE CODE DE CONSTRUCTION

Rôle des classes participants au pattern.



Rappel du principe SOLID

- **SRP : Single Responsability Principle** (Principe de responsabilité unique)
- **OCP : Open Close Principle** (Principe d'ouverture/fermeture)
- **LSP : Liskov Substitution Principle** (Principe de substitution de Liskov)
- **ISP : interface segregation principle** (Principe de ségrégation des interfaces)
- **DIP : Dependency Inversion Principle** (Principe d'inversion des dépendances)

Respect du Principe SOLID du Pattern Factory

SRP : Le code de création est regroupé au même endroit

OCP: possibilité de rajouter des produits sans toucher au coeur du code

Limites du Pattern Factory

- Nombreuses sous classes à implémenter
  => code plus difficile/complexe
- Une interface commune est nécessaire entre les objets crées

Lien avec les autres Patterns

- Abstract factory : les fabriques abstraites sont basées sur un ensemble de fabrique.

- Itérateur :
  utilisation de factory fréquente pour l'itérateur.

Méthode de la java doc mettant en œuvre le pattern factory

Dans la package java.util la classe Calendar voit sa méthode getInstance() utiliser le pattern

Dans la package java.util la classe ResourceBundle voit sa méthode getBundle() utiliser le pattern

Dans la package java.text la classe NumberFormat voit sa méthode getInstance() utiliser le pattern

Code de l'exemple : fabrique de pain

```java
3 usages  2 inheritors
public abstract class Pain {
    2 implementations
    public abstract String getForme();
    2 implementations
    public abstract int getPrix();

    2 implementations
    public abstract void cuire();
}
```

```java
public class Baguette extends Pain{
    2 usages
    private String forme;
    2 usages
    private int prix;

    1 usage
    public Baguette(String forme, int prix) {
        this.forme = forme;
        this.prix = prix;
    }

    @Override
    public String getForme() {
        return this.forme;
    }

    @Override
    public int getPrix() {
        return this.prix;
    }

    @Override
    public void cuire() {
        System.out.println("La baguette a bien cuit !");
    }
}
```

```java
public class FabriquePain {
    public static Pain getPain(String type ,String forme, int prix){
        if("Baguette".equalsIgnoreCase(type)) return new Baguette(forme, prix);
        else if("Tourte".equalsIgnoreCase(type)) return new Tourte(forme, prix);

        return null;
    }
}
```

Le Pattern Abstract Factory

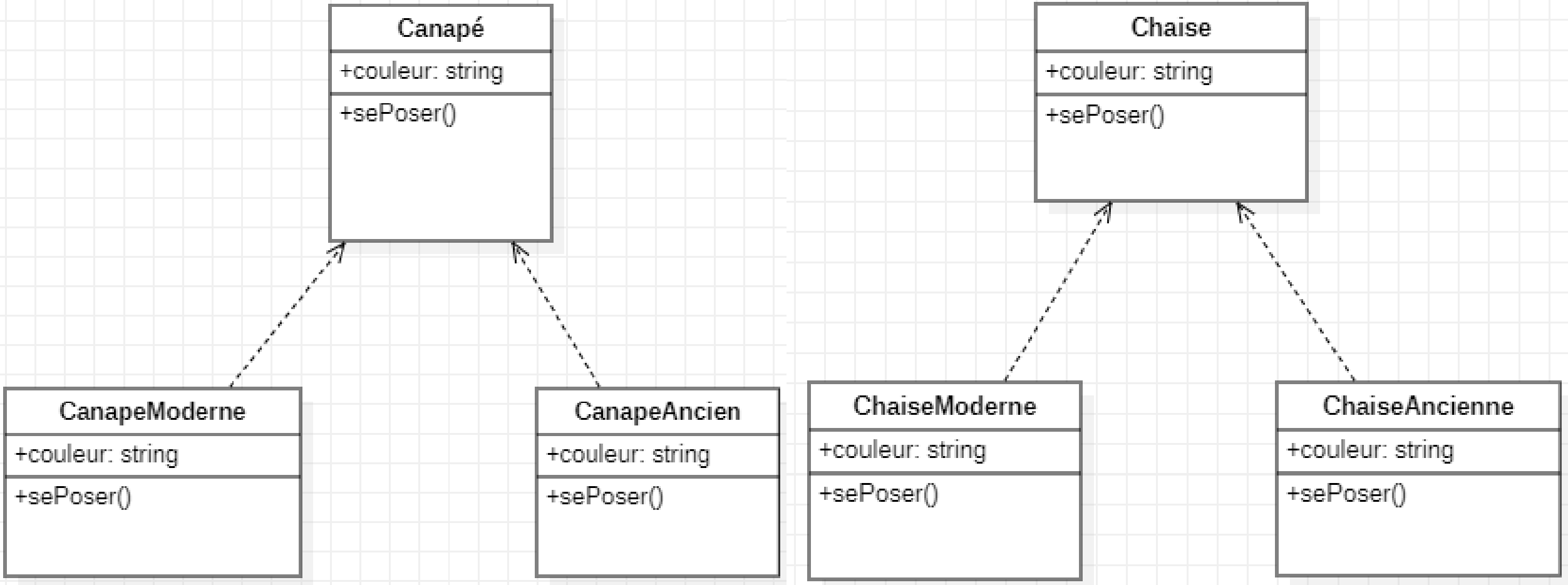
Besoin du pattern Abstract Factory

Un magasin vend différents meuble (table, chaise, …)

Le magasin propose un seul style : moderne, mais veut se diversifié

On veut donc créer différents meubles dans différents styles

Modélisation du besoin



Problème de la modélisation précédente

Comment simplifier la création de variante de famille d'objet ?

Si le magasin veut vendre un nouveau style de meuble.

Nous devons utiliser une interface regroupant les méthodes de création de famille d'objet.

Modélisation résolvant le problème

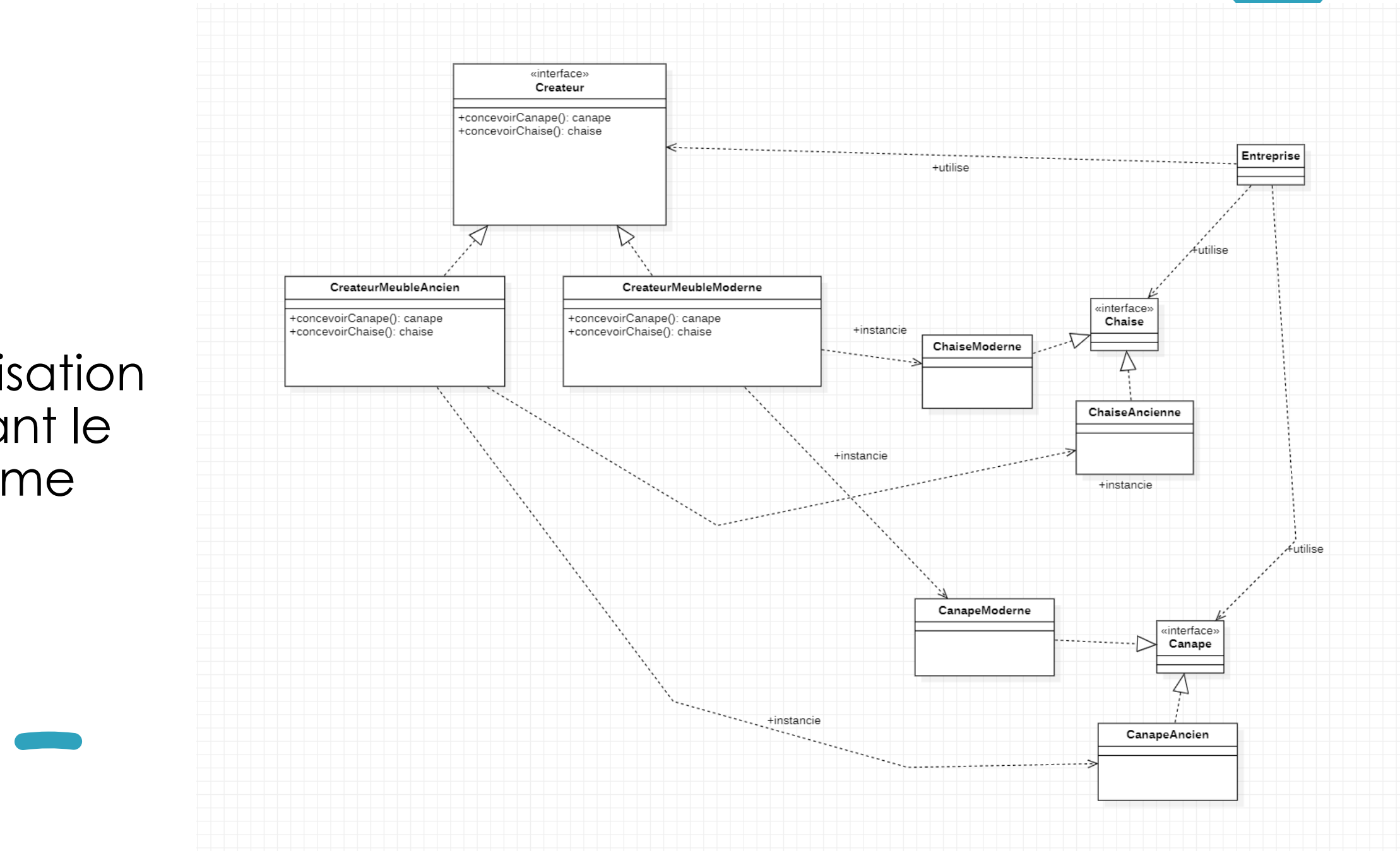
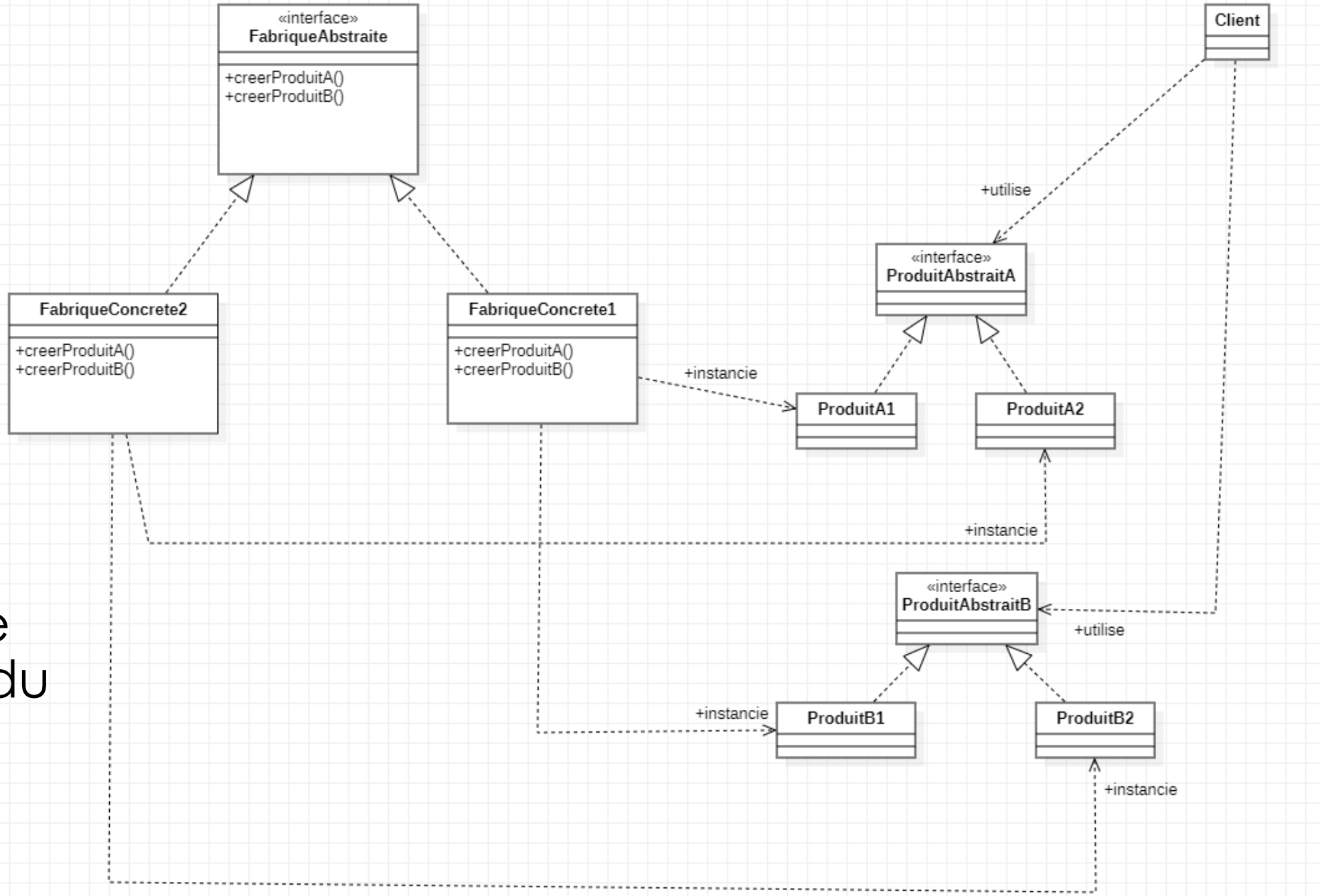


Diagramme générique du pattern



Définition du Pattern Abstract Factory

Type : création

Simplifier la création de variantes de familles d'objet

Interface regroupant méthodes de création des familles d'objets

Rôle des classes participants au pattern.

FABRIQUE ABSTRAITE

FABRIQUE CONCRÈTE

PRODUIT ABSTRAIT

PRODUIT CONCRET

Respect du Principe SOLID du Pattern Abstract Factory

SRP : Le code de création est regroupé au même endroit

DIP : Les objets de forte valeur métier ne dépendent pas des objets de faible valeur métier

OCP : Le code est ouvert à l'extension mais fermé à la modification

ISP : Un client ne doit pas dépendre d'interfaces qui ne lui correspondent pas

Limites du Pattern Abstract Factory

- Code plus confus pattern factory
- Nécessite ajout de nouvelles interfaces et classes

Lien avec les autres Patterns

- Factory: Les fabriques abstraites se basent en général sur un ensemble de fabrique.
- Monteur : Se concentre sur la création d'objet, une association permettrait une famille d'objet complexes.
- Pont : Le couplage des deux permet d'encapsuler les relations et de cacher la complexité au code client.

Méthode de la java doc mettant en œuvre le pattern abstract factory

Dans le package javax.xml.parsers la classe DocumentBuilderFactory voit sa méthode newInstance() utiliser le pattern

Dans le package javax.xml.transform la classe TransformerFactory voit sa méthode newInstance() utiliser le pattern

Dans le package javax.xml.xpath la classe XPathFactory voit sa méthode newInstance() utiliser le pattern

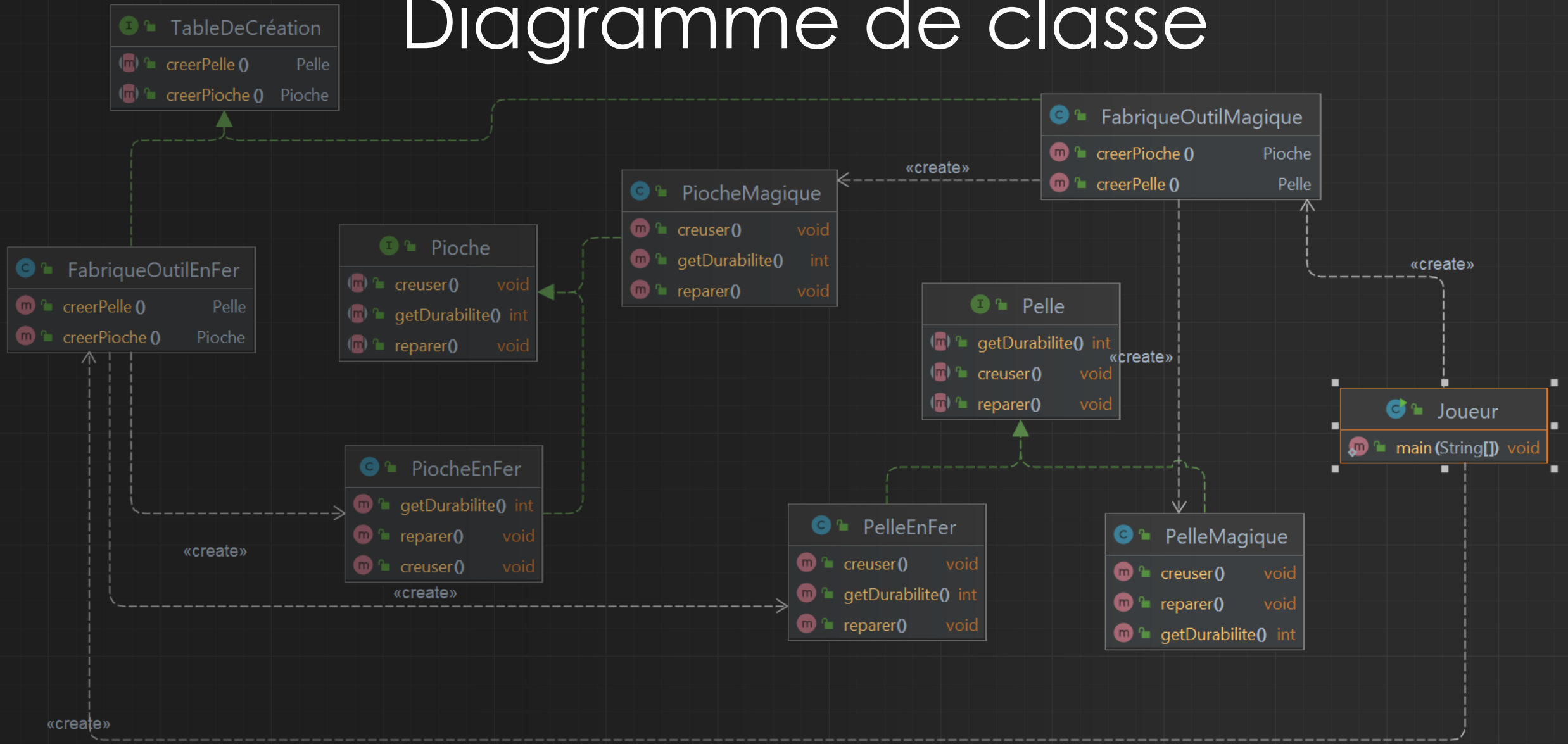
Nouveau contexte pour illustrer le abstract factory pattern

Un joueur d'un jeu quelconque veut créer des outils plus performants

Actuellement il peut créer des outils en fer mais veut des outils incassables

Il va apprendre et créer ces nouveaux outils plus performants

Diagramme de classe



Classe de base :

```java
package model;

public interface Pelle {
    2 usages  2 implementations
    public void creuser();
    2 usages  2 implementations
    public void reparer();

    3 usages  2 implementations
    public int getDurabilite();
}
```

```java
package model;

public class PelleEnFer implements Pelle {

    5 usages
    private int durabilite;

    1 usage
    public PelleEnFer(int durabilite) { this.durabilite = durabilite; }

    2 usages
    @Override
    public void creuser() {
        System.out.println("Vous avez casser un block avec la pelle en fer, elle s'est un peu abimée");
        this.durabilite = this.durabilite - 1;
    }

    2 usages
    @Override
    public void reparer() {
        this.durabilite = 100;
        System.out.println("La pelle à bien été réparé");
    }

    3 usages
    @Override
    public int getDurabilite() { return this.durabilite; }
}
```

```java
package model;

public class PelleMagique implements Pelle {
    2 usages
    @Override
    public void creuser() { System.out.println("Vous avez casser un block avec la pelle magique "); }

    2 usages
    @Override
    public void reparer() { System.out.println("Vous n'avez pas besoin de reparer la pelle, elle est incassable !"); }

    3 usages
    @Override
    public int getDurabilite() { return 100; }
}
```

Fabrique:

```java
package model;

public interface TableDeCréation {
    2 usages  2 implementations
    Pioche creerPioche();
    2 usages  2 implementations
    Pelle creerPelle();
}
```

```java
package model;

public class FabriqueOutilMagique implements TableDeCréation {
    2 usages
    @Override
    public Pioche creerPioche() { return new PiocheMagique(); }

    2 usages
    @Override
    public Pelle creerPelle() { return new PelleMagique(); }
}
```

```java
package model;

public class FabriqueOutilEnFer implements TableDeCréation {
    2 usages
    @Override
    public Pioche creerPioche() { return new PiocheEnFer( durabilite: 100); }

    2 usages
    @Override
    public Pelle creerPelle() { return new PelleEnFer( durabilite: 100); }
}
```

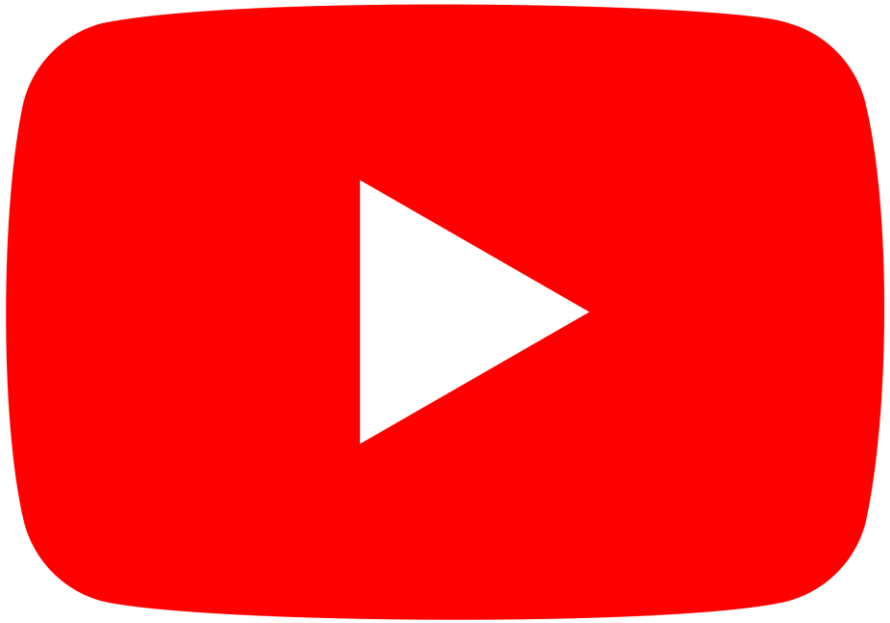
Utilisateur et résultat :

```
//////////////////////////////// Pioche en fer//////////////////////////////////////////
Vous avez casser un block avec la pioche en fer, elle s'est un peu abimée
Il reste 99 durabilité sur la pioche en fer
La pioche à bien été réparé
Il reste 100 durabilité sur la pioche en fer

//////////////////////////////// Pioche magique//////////////////////////////////////////
Vous avez casser un block avec la pioche magique
Il reste 100 durabilité sur la pioche en fer
Vous n'avez pas besoin de reparer la pioche, elle est incassable !

//////////////////////////////// Pelle en fer//////////////////////////////////////////
Vous avez casser un block avec la pelle en fer, elle s'est un peu abimée
Il reste 99 durabilité sur la pioche en fer
La pelle à bien été réparé
Il reste 100 durabilité sur la pioche en fer

//////////////////////////////// Pelle magique//////////////////////////////////////////
Vous avez casser un block avec la pelle magique
Il reste 100 durabilité sur la pioche en fer
Vous n'avez pas besoin de reparer la pelle, elle est incassable !
```

```java
package model;

public class Joueur {

    public static void main(String[] args) {

        Pioche piocheEnFer = new FabriqueOutilEnFer().creerPioche();
        Pioche piocheMagique = new FabriqueOutilMagique().creerPioche();

        //////////////////////////////// Pioche en fer//////////////////////////////////////////
        System.out.println("//////////////////////////////// Pioche en fer//////////////////////////////////////////");
        piocheEnFer.creuser();
        System.out.println("Il reste " + piocheEnFer.getDurabilite() + " durabilité sur la pioche en fer");
        piocheEnFer.reparer();
        System.out.println("Il reste " + piocheEnFer.getDurabilite() + " durabilité sur la pioche en fer");

        //////////////////////////////// Pioche magique//////////////////////////////////////////
        System.out.println("\n//////////////////////////////// Pioche magique//////////////////////////////////////////");
        piocheMagique.creuser();
        System.out.println("Il reste " + piocheMagique.getDurabilite() + " durabilité sur la pioche en fer");
        piocheMagique.reparer();



        Pelle pelleEnFer = new FabriqueOutilEnFer().creerPelle();
        Pelle pelleMagique = new FabriqueOutilMagique().creerPelle();

        //////////////////////////////// Pelle en fer//////////////////////////////////////////
        System.out.println("\n//////////////////////////////// Pelle en fer//////////////////////////////////////////");
        pelleEnFer.creuser();
        System.out.println("Il reste " + pelleEnFer.getDurabilite() + " durabilité sur la pioche en fer");
        pelleEnFer.reparer();
        System.out.println("Il reste " + pelleEnFer.getDurabilite() + " durabilité sur la pioche en fer");


        //////////////////////////////// Pelle magique//////////////////////////////////////////
        System.out.println("\n//////////////////////////////// Pelle magique//////////////////////////////////////////");
        pelleMagique.creuser();
        System.out.println("Il reste " + pelleMagique.getDurabilite() + " durabilité sur la pioche en fer");
        pelleMagique.reparer();

    }
}
```

Live coding du pattern

Retour sur la boulangerie, transformation en fabrique abstraite



https://www.youtube.com/watch?v=rJ6uBMQPjOo&ab_channel=YlianF

DES QUESTIONS?

Si vous avez des questions, posez-les, plus tard, il sera trop tard, on aura oublié...

QCM :

Vous avez bien tout écouté ?

On va voir ça !

https://forms.gle/fwx7rEhGF7dL4HZJ8

Bibliographie / webographie

- https://www.ionos.fr/digitalguide/sites-internet/developpement-web/quest-ce-que-le-factory-pattern/
- https://www.codingame.com/playgrounds/36103/design-pattern-factory-abstract-factory/design-pattern-factory
- https://fr.wikipedia.org/wiki/Fabrique_(patron_de_conception)
- https://refactoring.guru/fr/design-patterns/abstract-factory
- https://refactoring.guru/fr/design-patterns/factory-method
- https://medium.com/elp-2018/un-design-pattern-abstract-factory-8c9d700b7b29
- https://ryax.tech/fr/design-pattern-cest-quoi-et-pourquoi-lutiliser/
- https://www.usabilis.com/design-patterns-pour-la-composition-interfaces/#:~:text=Le%20concept%20de%20Design%20Pattern,un%20langage%20de%20253%20patterns.
- https://fr.wikipedia.org/wiki/Patron_de_conception
- https://tech.io/playgrounds/36103/design-pattern-factory-abstract-factory/design-pattern-abstract-factory
- https://betterprogramming.pub/understanding-the-abstract-method-design-patterns-bc416aaaf076
- https://itexpert.fr/blog/factory-method-pattern/