Factory
&
Abstract Factory

Présenté par :

Evan TOUZET
Jules DELISLE
Tristan BADANA
Thomas SAZERAT

Sommaire

Introduction générale sur les patterns

Introduction générale sur les patterns

Définition

Méthodes de résolution de problèmes récurrents de conception en développement informatique

- Bonnes pratiques de conception
- Validés par des experts
- Documentés et connus des développeurs
- Accélèrent le processus de développement

Gang of four

Introduction générale sur les patterns

Types de patterns

Il existe 3 types de patterns :

Structurels :

Structure et de la composition d'une classe.

Comportementaux :

Façon dont une classe communique avec les autres.

Créationnels :

Instanciation de classe.

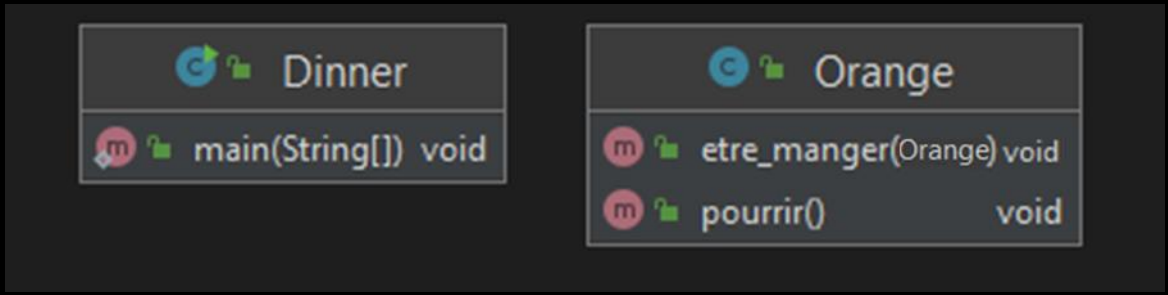
Design Pattern Factory

Factory

Besoin

Programme Dinner où on peut manger des aliments.

Un premier fruit : Orange qui peut être mangé et pourrir.

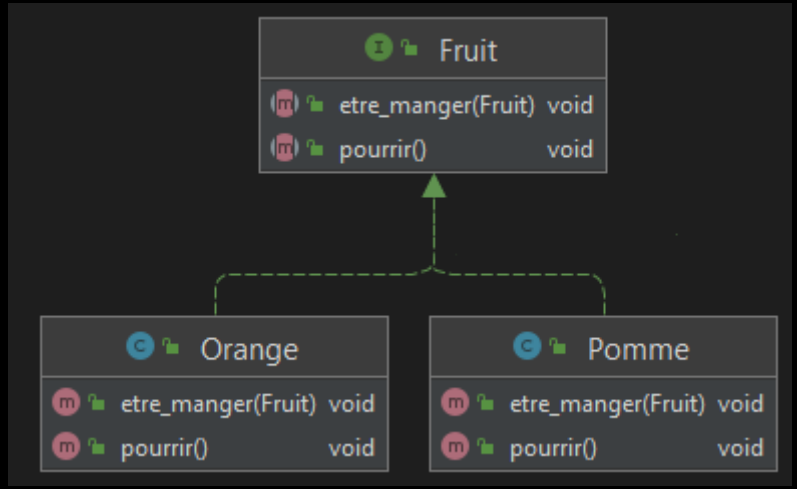


Factory

Besoin

Autre fruit : Pomme, lui aussi peut être mangé et pourrir.

Pomme et Orange implémentent alors l'interface Fruit.



Factory

Solution du factory

Factory : pattern de création

Problématique : comment simplifier la création d'objet et s'assurer de sa bonne implémentation dans le code ?

Solution : Factory sépare le code de construction d'objet et le code qui utilise cet objet.

Factory

Solution du factory

Si plus tard pour créer un fruit, il nous faut créer 3 objets avant, cela risque d'être très long à chaque création de fruit. D'où son intérêt.

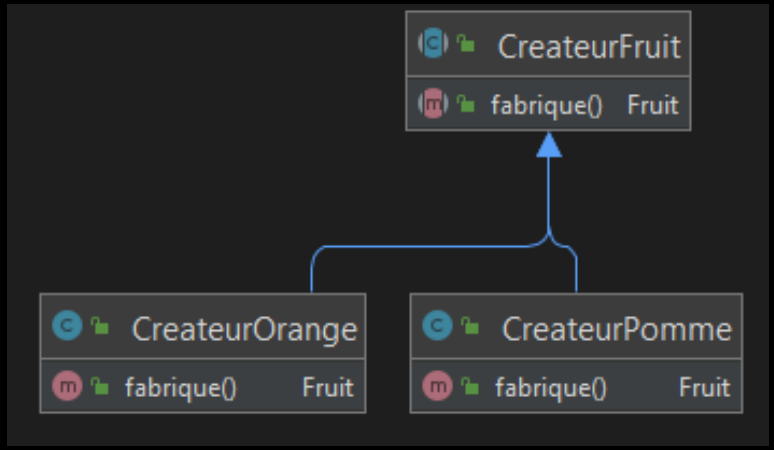
Il est surtout utilisé dans des Frameworks ou des librairies.

Factory

Solution du factory

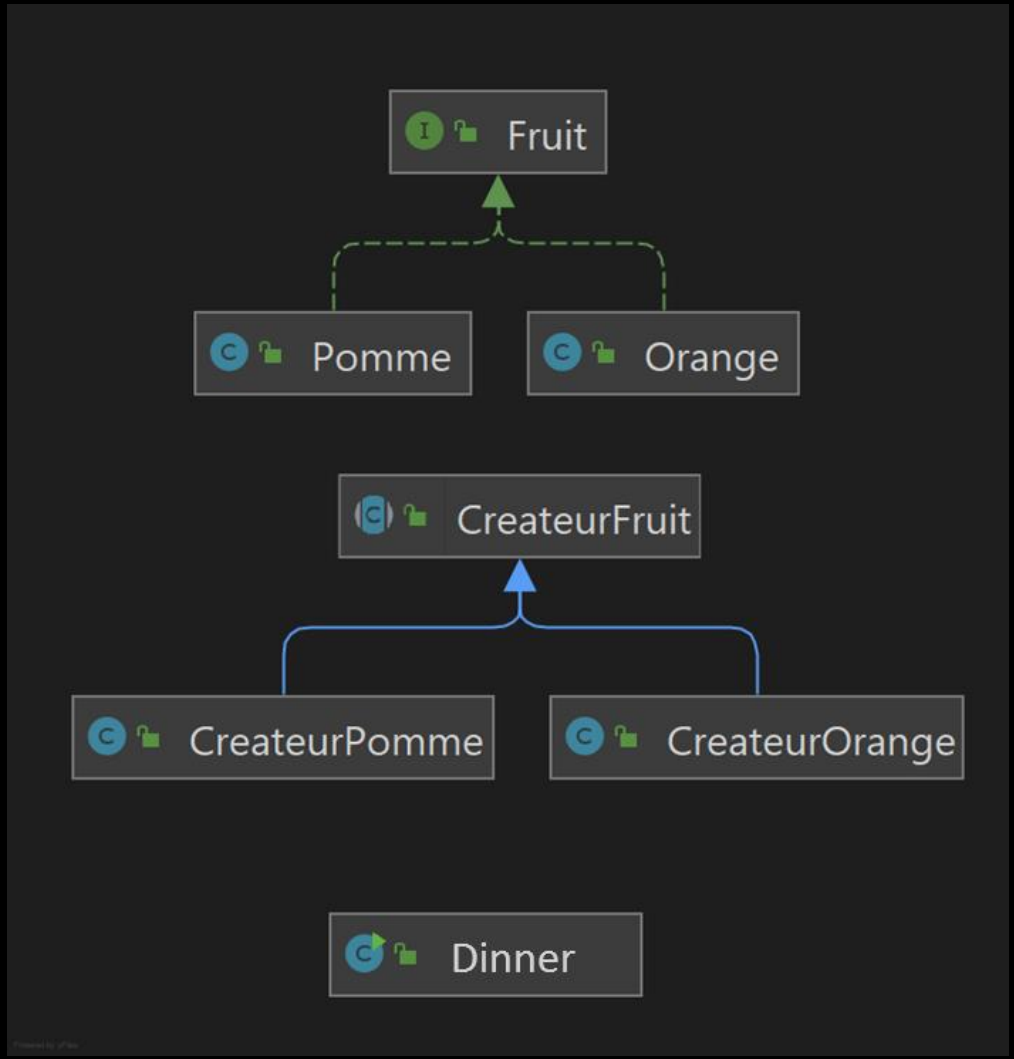
On peut utiliser une classe de factory CreateurFruit permettant de les instancier et qui sera une classe abstraite.

Ainsi que 2 classes CreateurOrange et CreateurPomme.



Factory

Solution du factory



Factory

Solution du factory

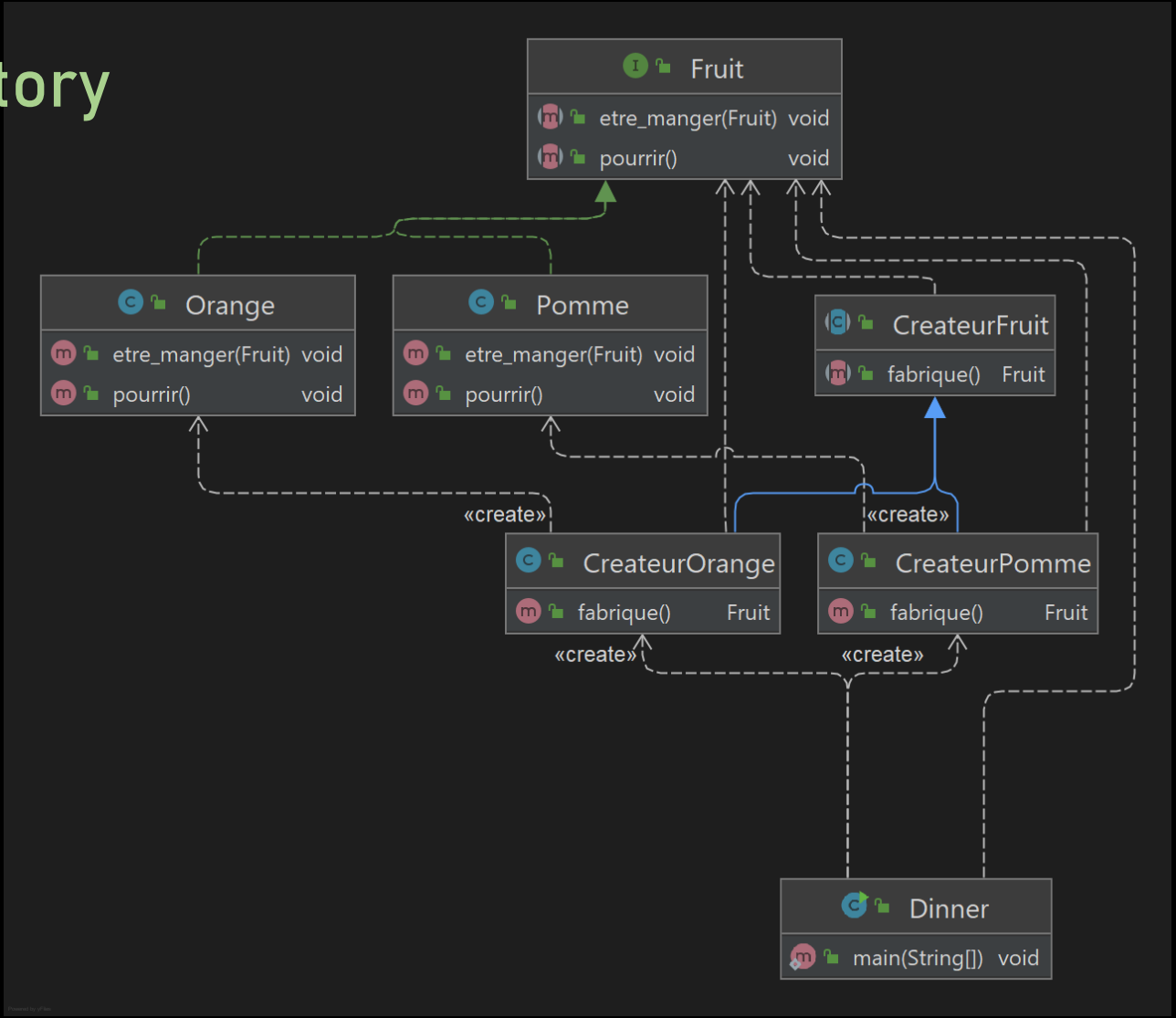
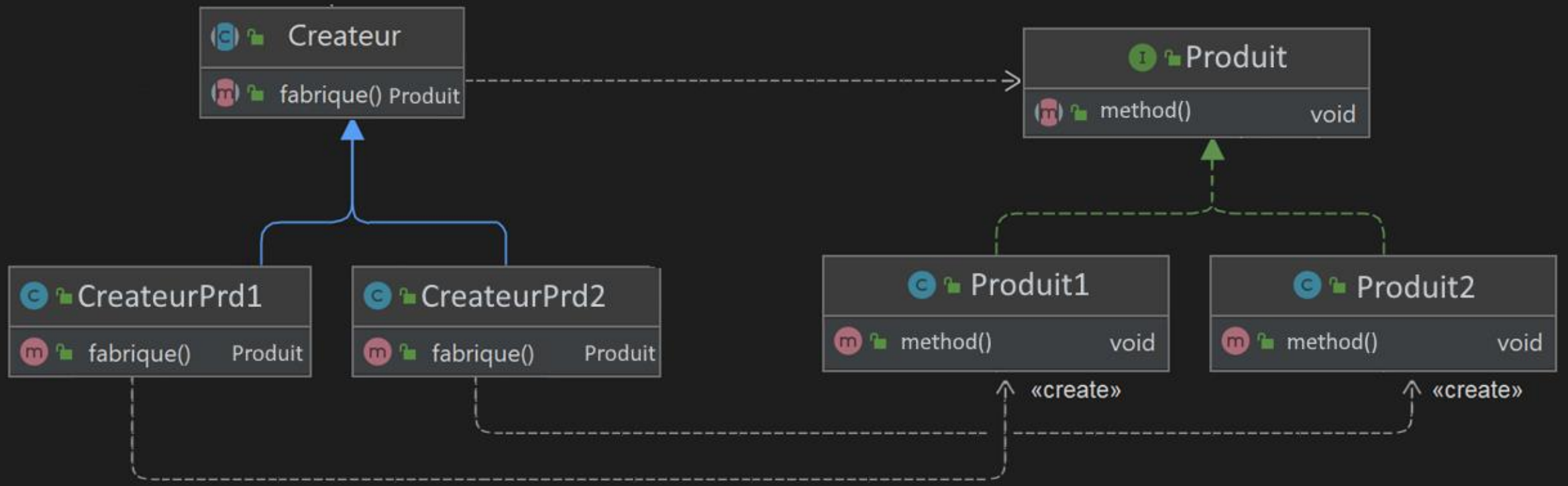


Diagramme de classe

Factory

Diagramme de classe



Code propre : principe SOLID

Principe SOLID

- Single responsibility principle
  - Une classe doit avoir une et une seule responsabilité
- Open/closed principle
  - Une classe doit être ouverte aux extensions, mais fermée aux modifications
- Liskov substitution principle
  - Les sous-types doivent pouvoir être substitués à leurs types de base
- Interface segregation principle
  - Un client ne devrait jamais être forcé de dépendre d'une interface qu'il n'utilise pas
- Dependency inversion principle
  - Les modules de haut niveau ne doivent pas dépendre de modules de plus bas niveau

Principes SOLID respectés

- Single responsibility principle
  - Vous pouvez déplacer tout le code de création des produits au même endroit, pour une meilleure maintenabilité.
- Open/closed principle
  - Vous pouvez ajouter de nouvelles variantes de produits sans endommager l'existant.

Limites

- Le code peut devenir plus complexe puisqu'il faut introduire de nombreuses sous-classes pour la mise en place du patron.
- Il est nécessaire que les objets crées appartiennent à une interface commune.

Code : Factory

Factory : code

Classes de base

```java
package factory.presentation;

public interface Fruit {
    void etre_manger(Fruit fruit);
    void pourrir();
}
```

```java
package factory.presentation;

public class Orange implements Fruit {
    private boolean est_pourri;
    //On est obligé d'utilisé fruit et pas Orange
    @Override
    public void etre_manger(Fruit orange) {
        if (this.est_pourri){
            System.out.println("Vous ne pouvez pas manger des fruit pourri");
        }
        else {
            System.out.println("L'orange a été mangé");
            orange = null;
        }
    }

    @Override
    public void pourrir() {
        System.out.println("Je suis une orange pourri");
        this.est_pourri = true;
    }
}
```

```java
package factory.presentation;

public class Pomme implements Fruit {
    private boolean est_pourri;

    @Override
    public void etre_manger(Fruit pomme) {
        if (this.est_pourri){
            System.out.println("Vous ne pouvez pas manger des fruit pourri");
        }
        else {
            System.out.println("La pomme a été mangé");
            pomme = null;
        }
    }

    @Override
    public void pourrir() {
        System.out.println("je suis une pomme pourri !");
        this.est_pourri = true;
    }
}
```

Factory : code

Créateurs

```java
package factory.presentation;

public abstract class CreateurFruit {
    public abstract Fruit fabrique();
}
```

```java
package factory.presentation;

public class CreateurOrange extends CreateurFruit {
    @Override
    public Fruit fabrique(){
        return new Orange();
    }
}
```

```java
package factory.presentation;

public class CreateurPomme extends CreateurFruit {
    @Override
    public Fruit fabrique(){
        return new Pomme();
    }
}
```

Factory : code

Classe cliente & test

```java
package factory.presentation;

public class Utilisateur {
    public static void main(String args[]){
        CreateurOrange createurOrange = new CreateurOrange();
        CreateurPomme createurPomme = new CreateurPomme();

        Fruit pomme = createurPomme.fabrique();
        Fruit orange = createurOrange.fabrique();

        pomme.pourrir();
        orange.etre_manger(orange);
        pomme.etre_manger(pomme);
    }
}
```

```
"C:\Program Files\Java\jdk1.8.0_291\bin\java.exe" ...
je suis une pomme pourri !
L'orange a été mangé
Vous ne pouvez pas manger des fruits pourris

Process finished with exit code 0
```

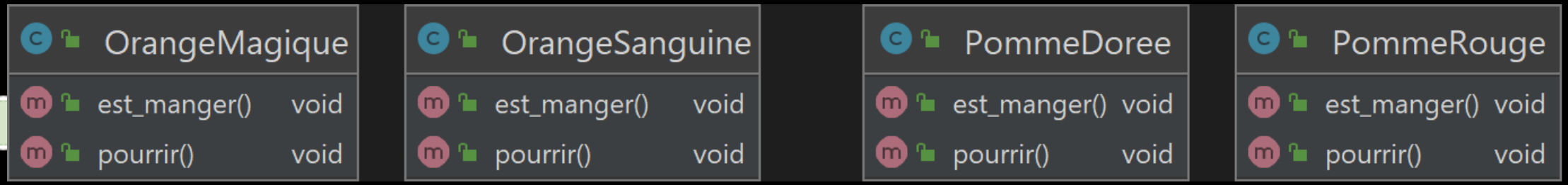
Design Pattern
Abstract Factory

Abstract Factory

Besoin

Créer des objets individuels tout en les faisant correspondre à d'autres objets de la même famille.

Cette fois nous avons des Oranges et des Pommes magiques.



Abstract Factory

Solution de l'abstract factory

Factory : pattern de création

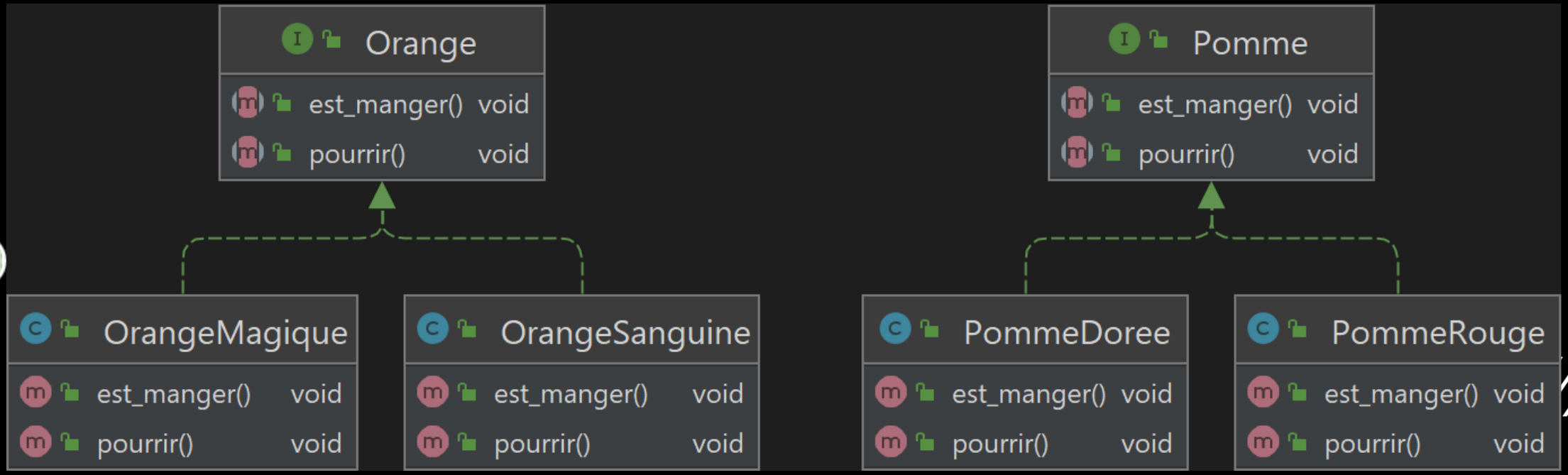
Problématique : comment simplifier la création de variantes de familles d'objet et s'assurer de leur bonne implémentation dans le code ?

Solution : L'Abstract Factory est une interface regroupant toutes les méthodes de création pour les familles d'objets.

Abstract Factory

Solution de l'abstract factory

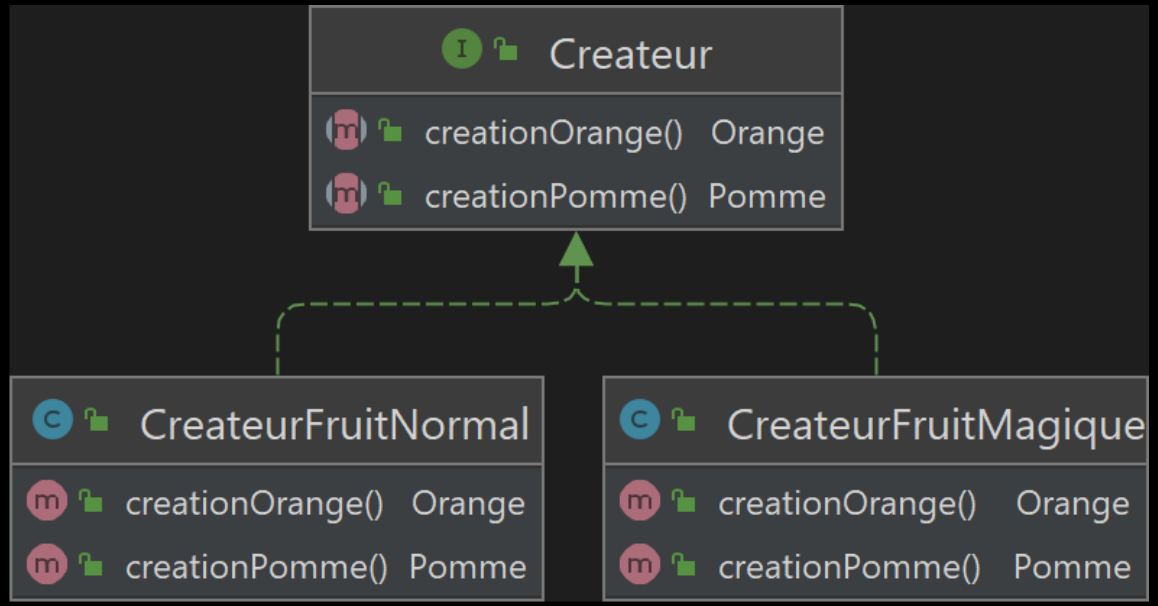
Pas d'interface fruit, mais interfaces spécifiques pour Oranges et Pommes.



Abstract Factory

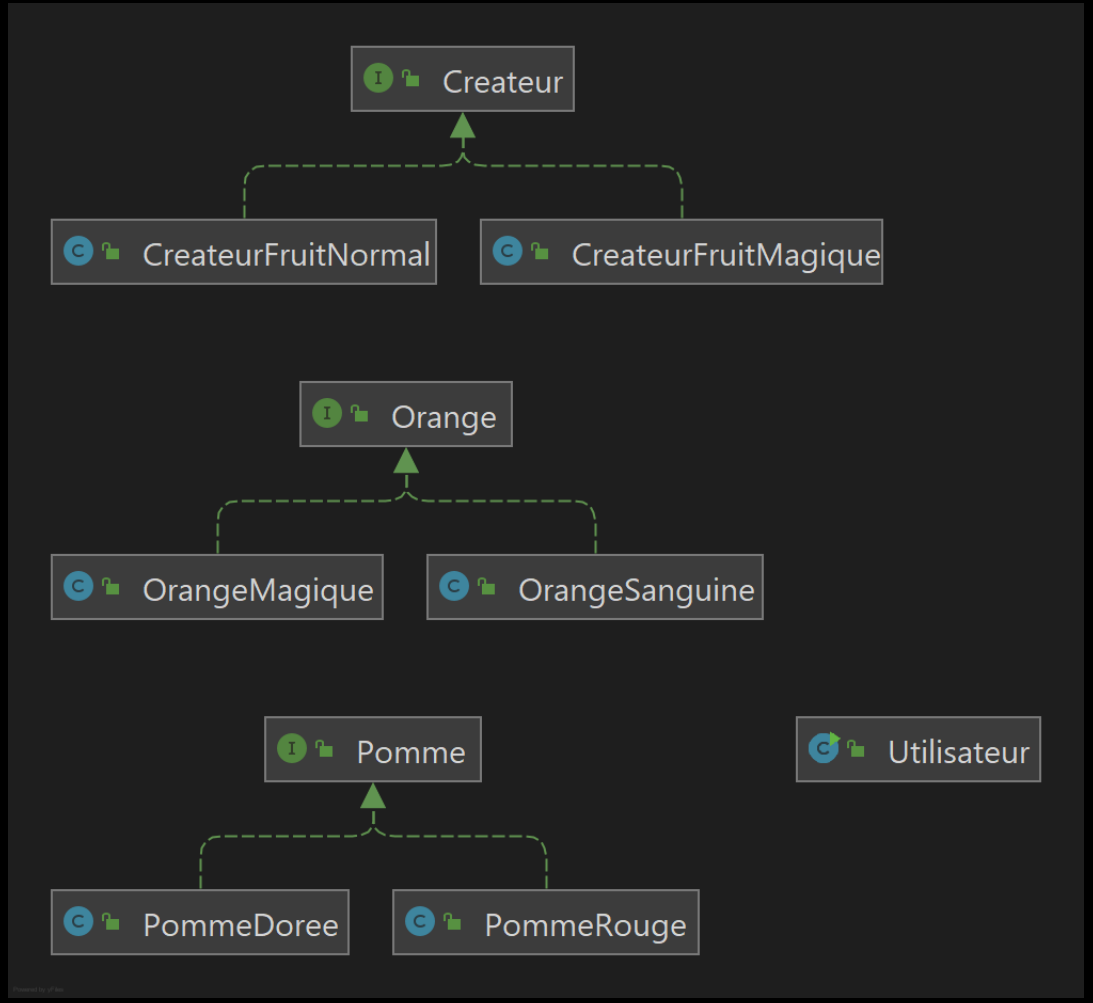
Solution de l'abstract factory

On va alors utiliser une interface Createur implémenté par des CreateurFruitNormal et CreateurFruitMagique



Abstract Factory

Solution de l'abstract factory



Abstract Factory

Solution de l'abstract factory

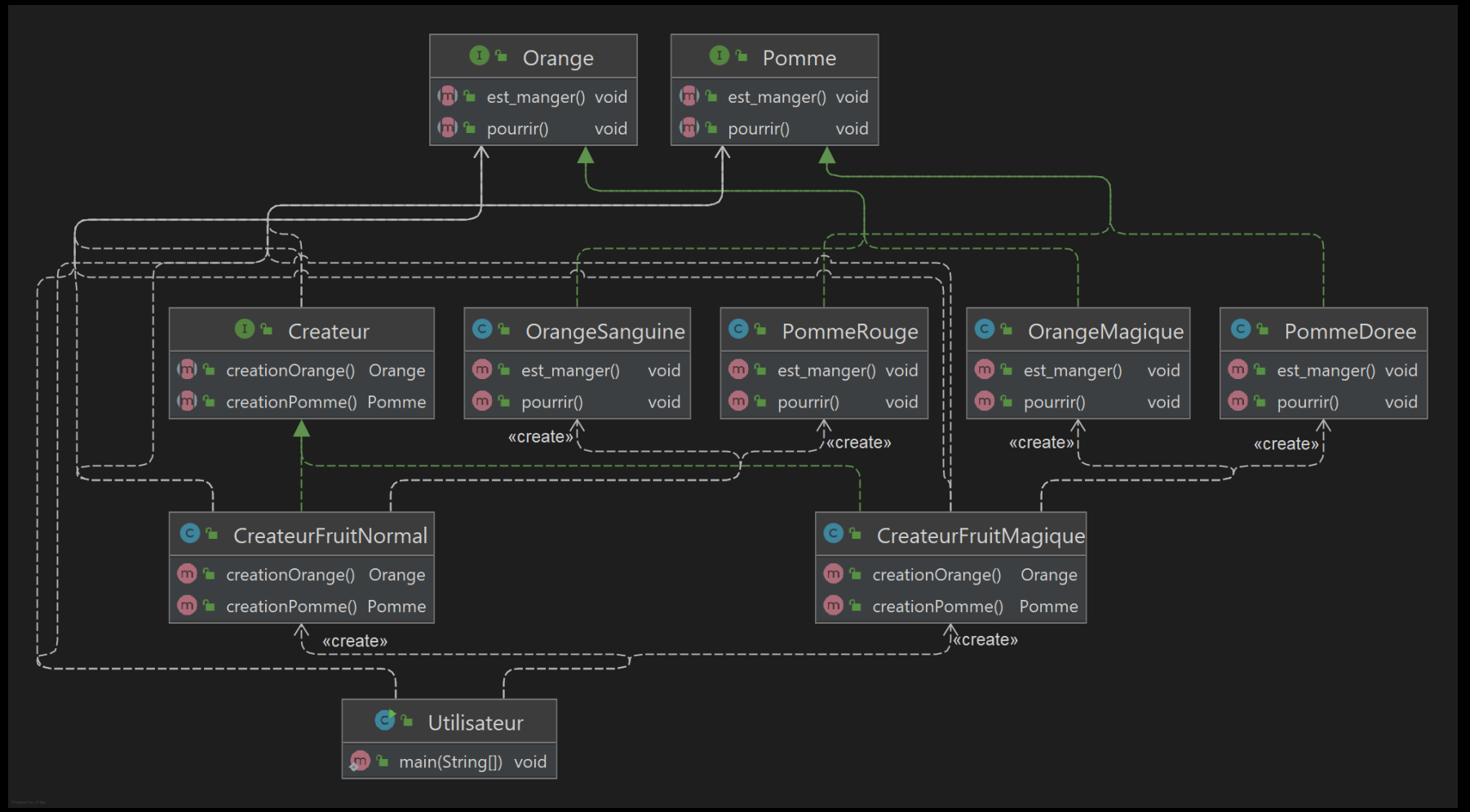
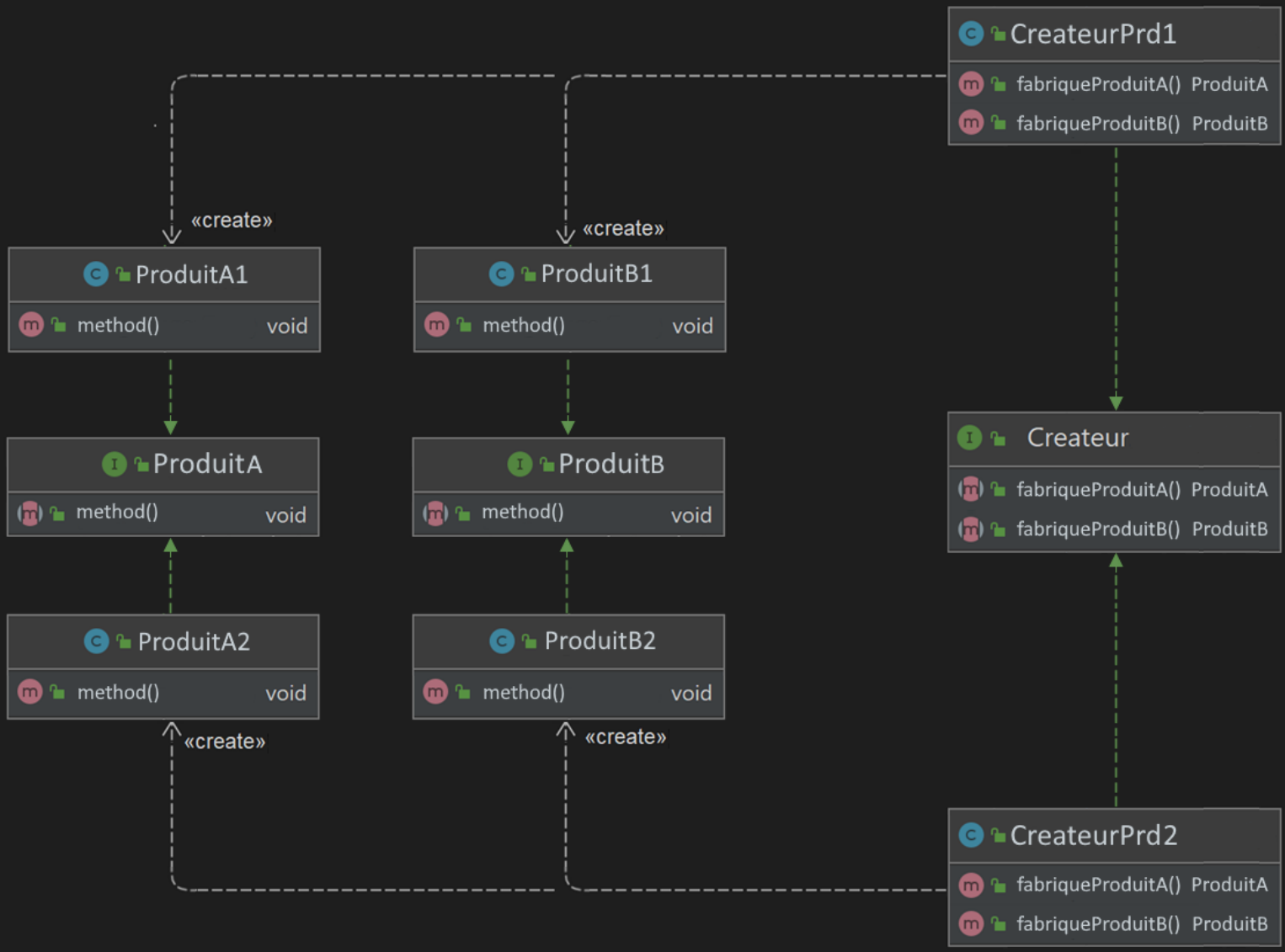


Diagramme de classe


CreateurPrd1
fabriqueProduitA() ProduitA
fabriqueProduitB() ProduitB
«create»
«create»
ProduitA1
method() void
ProduitB1
method() void
ProduitA
method() void
ProduitB
method() void
Createur
fabriqueProduitA() ProduitA
fabriqueProduitB() ProduitB
ProduitA2
method() void
ProduitB2
method() void
«create»
«create»
CreateurPrd2
fabriqueProduitA() ProduitA
fabriqueProduitB() ProduitB

Code propre : principe SOLID

Limites

- Le code peut devenir plus complexe que nécessaire, car ce patron de conception impose l'ajout de nouvelles classes et interfaces.

Code : Abstract factory

Abstract Factory : code

Classes de base

```java
package abstractfactory;

public interface Pomme {
    void est_manger();
    void pourrir();
}
```

```java
package abstractfactory;

public class PommeDoree implements Pomme {

    @Override
    public void est_manger() {
        System.out.println("Vous avez manger une pomme dorée, vous vous santez mieux maintenant");
    }

    @Override
    public void pourrir() {
        System.out.println("La pomme dorée ne pourri pas");
    }
}
```

```java
package abstractfactory;

public class PommeRouge implements Pomme{
    boolean est_pourri;

    public PommeRouge(boolean est_pourri) { this.est_pourri = est_pourri; }

    @Override
    public void est_manger() {
        if (this.est_pourri){
            System.out.println("Vous ne pouvez pas manger un fruit pourri");
        }
        else{
            System.out.println("Vous avez mangé une pomme");
        }
    }

    @Override
    public void pourrir() { this.est_pourri = true; }
}
```

Abstract Factory : code

Créateurs

```java
package abstractfactory;

public interface Createur {
    Pomme creationPomme();
    Orange creationOrange();


}
```

```java
package abstractfactory;

public class CreateurFruitMagique implements Createur {

    @Override
    public Pomme creationPomme() { return new PommeDoree(); }

    @Override
    public Orange creationOrange() { return new OrangeMagique(); }
}
```

```java
package abstractfactory;

public class CreateurFruitNormal implements Createur{

    @Override
    public Pomme creationPomme() { return new PommeRouge( est_pourri: false); }

    @Override
    public Orange creationOrange() { return new OrangeSanguine( est_pourri: false); }
}
```

Abstract Factory : code

Utilisateur

```java
package abstractfactory;

public class Utilisateur {


    public static void main(String args[]){
        CreateurFruitMagique createurFruitMagique = new CreateurFruitMagique();
        CreateurFruitNormal createurFruitNormal = new CreateurFruitNormal();

        Pomme pommeDoree = new CreateurFruitMagique().creationPomme();
        Pomme pommeRouge = new CreateurFruitNormal().creationPomme();

        pommeDoree.est_manger();
        pommeRouge.est_manger();

        Orange orangeSanguine = createurFruitNormal.creationOrange();
        Orange orangeMagique = createurFruitMagique.creationOrange();

        orangeSanguine.pourrir();
        orangeMagique.pourrir();
        orangeSanguine.est_manger();
        orangeMagique.est_manger();

    }
}
```

Abstract Factory : code

Test du code

```
"C:\Program Files\Java\jdk1.8.0_291\bin\java.exe" ...
Vous avez manger une pomme dorée, vous vous santez mieux maintenant
Vous avez mangé une pomme
L'orange est pourri
L'orange magique ne peut pas pourrir
Vous ne pouvez pas manger un fruit pourri
Vous avez manger une orange magique
```

Conclusion

QCM

https://docs.google.com/forms/d/e/1FAIpQLSeHas-tbInrsTuR2MCZXa9oikXpx90d1S1SujRyWvPohCE_mg/viewform?usp=sf_link

Pour quelle situation le Factory pattern est-il le plus utile ?

- Lorsque l'on veut créer plusieurs types d'un même objet
- Lorsque l'on veut créer plusieurs types de différents objets
- Les deux

Quels principes SOLID respectent les designs patterns Factory & Abstract Factory ? (plusieurs réponses)

- Single responsibility principle
- Open/closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle

Pour mettre en place un factory pattern, est-on obligé d'utiliser des interfaces ?

- Oui
- Non

A quel type de pattern appartiennent les patterns Factory et Abstract Factory ?

- Création
- Structurel
- Comportement

En français le pattern Factory pourrait se traduire par :

- Patron de facteur
- Patron d'usine
- Patron de fournisseur

Références

- https://refactoring.guru/fr/design-patterns/factory-method
- https://refactoring.guru/fr/design-patterns/abstract-factory
- https://www.codingame.com/playgrounds/36103/design-pattern-factory-abstract-factory/design-pattern-factory
- https://fr.wikipedia.org/wiki/Fabrique_(patron_de_conception)
- https://www.invivoo.com/design-patterns/