

# Initiation génie logiciel : Tests unitaires et patrons de conception

Arnaud Labourel (arnaud.labourel@univ-amu.fr)

18 septembre 2024



# Section 1

## Tests

# Différents types de tests

## Règle

Un code non testé n'a aucune valeur.

## Corollaire

Tout code doit être testé

## Différents types de tests

- **Tests unitaires** : Tester les différentes parties (méthodes, classes) d'un programme indépendamment les unes des autres.
- **Tests d'intégration** : Tester une portion du programme (plusieurs classes).
- **Tests de non-régression** : Vérifier que le nouveau code ajouté ne corrompt pas les codes précédents : les tests précédemment réussis doivent encore l'être.

# Tests unitaires

- Tester une unité de code : classe, méthodes, ...
- Vérifier un comportement :
  - ▶ cas normaux
  - ▶ cas limites
  - ▶ cas anormaux

## Tests unitaires en java : JUnit avec assertJ

- JUnit : un framework de test unitaire pour Java
- AssertJ : surcouche de JUnit pour réaliser des tests à base d'**assertions**

# Utilisation de JUnit (1/3)

## Règles

- 1 classe de test = un ensemble de méthodes de test
- 1 classe de test par classe à tester
- 1 méthode de test = 1 cas de test
- 1 cas de test = (description, données d'entrée, résultat attendu)

## Structure d'une méthode de test de base

- méthode d'instance publique
- annotée avec `@Test` (à mettre avant la déclaration de la méthode)
- ne prend aucun paramètre
- ne renvoie rien (`void`)
- lève une `AssertionError` en cas de test échoué

## Conventions de nommage

- nom d'une classe de test : *NameTestedClassTest*
- nom d'une méthode de test : *testNameTestedMethod*

## Structure d'un projet avec tests

Les tests sont séparés du code de production : répertoire `main` pour le code de production et répertoire `test` pour le code de tests.

⇒ nécessaire de séparer les tests du code de production car :

- on ne donne pas l'accès au code de test au client par exemple
- les tests ont un rôle spécifique différent du code de production

# Utilisation de JUnit (3/3)

```
import org.junit.jupiter.api.Test;
import static org.assertj.core.api.Assertions.*;

public class NameTestedClassTest {
    @Test
    void testNameTestedMethod_conditionTest(){
        /* code containing assertions to test
        nameTestedMethod */
    }
}
```

# Assertions assertJ (1/2)

- `assertThat(condition).isTrue()` : vérifie que `condition` est vraie.
- `assertThat(condition).isFalse()` : vérifie que `condition` est faux.
- `assertThat(actual).isEqualTo(expected)` : vérifie que `expected` est égal à `actual` égal : `equals` pour les objets et `==` pour les types primitifs.
- `assertThat(actual).isCloseTo(expected, within(delta))` : vérifie que  $|expected - actual| \leq delta$
- `assertThat(object).isNull()` : vérifie que la référence est null

# Assertions assertJ (2/2)

- `assertThat(object).isNotNull()` : vérifie que la référence **n'est pas** null
- `assertThat(actual).isSameAs(expected)` : vérifie que les deux objets sont les mêmes (même référence).
- `assertThat(list).containsExactly(e1, e2, e3)` : vérifie que la liste `list` contient uniquement les éléments `e1`, `e2` et `e3` dans cet ordre.
- `assertThat(list1).containsExactlyElementsOf(list2)` : vérifie que les deux listes `list1` et `list2` contiennent les mêmes éléments dans le même ordre.
- `fail(message)` : échoue en affichant le `String` `message`

## Message

Il est possible de provoquer l'affichage d'un message lors d'un test faux en appelant `as(message)` sur le retour d'un `assertThat`.

## Exemple de classe à tester : RationalNumber

```
public class RationalNumber {
    public final int numerator;
    public final int denominator;
    public RationalNumber(int numerator, int denominator) {
        int gcd = gcd(numerator, denominator);
        this.numerator = numerator / gcd;
        this.denominator = denominator / gcd;
    }
    public RationalNumber add(RationalNumber val) {
        int numerator = (this.numerator * val.denominator)
            + (this.denominator * val.numerator);
        int denominator = this.denominator * val.denominator;
        return new RationalNumber(numerator, denominator);
    }
}
```

# Exemple de classe de test

```
import org.junit.jupiter.api.Test;
import static org.assertj.core.api.Assertions.*;
public class RationalNumberTest {
    @Test
    void testAdd_onePlusOneIsTwo(){
        RationalNumber one = new RationalNumber(1, 1);
        RationalNumber onePlusOne = one.add(one);
        assertThat(onePlusOne.numerator)
            .as("Numerator of one plus one is two.")
            .isEqualTo(2);
        assertThat(onePlusOne.denominator)
            .as("Denominator of one plus one is one.")
            .isEqualTo(1);
    }
}
```

# Exemple de classe à tester : Box

```
public class Box {  
    /**  
     * Create a box with the specified weight  
     * @param weight the weight of the created box  
     */  
    public Box(int weight) {  
        this.weight = weight;  
    }  
    /** weight of the box */  
    private int weight;  
    /** @return this box's weight */  
    public int getWeight() {  
        return this.weight;  
    }  
}
```

## Exemple de classe de test : TestBox

```
import static org.assertj.core.api.Assertions.*;
import org.junit.jupiter.api.Test;

public class BoxTest {
    @Test
    public void testGetWeight_afterCreation() {
        Box someBox = new Box(10);
        assertThat(someBox.getWeight()).isEqualTo(10);
    }
    // ...
}
```

## Exemple de classe à tester (1/2) : Emails

```
public class Emails {
    private String text;
    public Emails(String text) { this.text = text; }
    public List<String> userNames() {
        int pos = 0;
        List<String> users = new ArrayList<String>();
        for(;;) {
            int atIndex = text.indexOf('@', pos);
            if (atIndex == -1) break;
            String userName = userName(atIndex);
            if (userName.length() > 0) users.add(userName);
            pos = atIndex + 1;
        }
        return users;
    }
}
```

## Exemple de classe à tester (2/2) : Emails

```
private String userName(int atIndex) {
    int back = atIndex - 1;
    while (back >= 0 &&
           (Character.isLetterOrDigit(text.charAt(back))
            || text.charAt(back) == '.')) {
        back--;
    }
    return text.substring(back + 1, atIndex);
}
}
```

## Exemple de classe de test (1/3)

```
import org.junit.jupiter.api.Test;
import static org.assertj.core.api.Assertions.*;

public class EmailsTest{
    @Test
    public void testUsers_simpleNames() {
        Emails emails =
            new Emails("foo bart@cs.edu xyz marge@ms.com baz");
        assertThat(emails.getUserNames())
            .containsExactly("bart", "marge");
    }
}
```

## Exemple de classe de test (2/3)

```
@Test
public void testUsers_withSpecialCharacters() {
    Emails emails =
        new Emails("fo f.ast@cs.edu bar&a.2.c@ms.com ");
    assertThat(emails.getUserNames())
        .containsExactly("f.ast", "a.2.c");
}
```

## Exemple de classe de test (3/3)

```
@Test
```

```
public void testUsers_extremalCases() {  
    Emails emails = new Emails("x y@cs 3@ @z@");  
    assertThat(emails.getUserNames())  
        .isNotEmpty();  
        .containsExactly("y", "3", "z");  
    emails = new Emails("no emails here!");  
    assertThat(emails.getUserNames()).isEmpty();  
    emails = new Emails("@@@");  
    assertThat(emails.getUserNames()).isEmpty();  
    emails = new Emails("");  
    assertThat(emails.getUserNames()).isEmpty();  
}  
}
```

- préfixée de l'annotation `@Test`
- signature de la forme `public void testMethod()`
- le corps de la méthode contient des assertions : `assertThat`

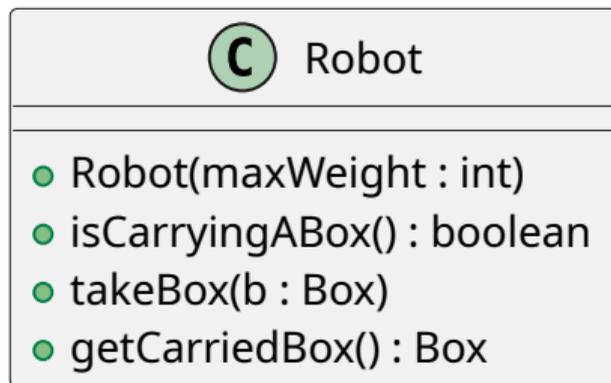
**le test est réussi si toutes les assertions sont vérifiées**

- plusieurs méthodes de tests peuvent être nécessaires pour tester la correction d'une méthode
- principes :
  - 1 créer la situation initiale et vérifier les « préconditions »
  - 2 appeler la méthode testée
  - 3 à l'aide d'assertions, vérifier les « postconditions » = situation attendue après l'exécution de la méthode

# Méthodologie pour créer des tests

Un robot peut porter une caisse d'un poids maximal défini à la construction du robot. Initialement un robot ne porte pas de caisse. S'il porte déjà une caisse il ne peut en prendre une autre.

Classe Robot



# Méthodologie pour créer des tests

Un robot peut porter une caisse d'un poids maximal défini à la construction du robot. **Initialement un robot ne porte pas de caisse.** S'il porte déjà une caisse il ne peut en prendre une autre.

```
import static org.assertj.core.api.Assertions.*;
import org.junit.jupiter.api.Test;

public class RobotTest {
    @Test
    public void notCarryingABoxWhenCreated() {
        Robot robbie = new Robot(15);
        // no carried box ?
        assertThat(robbie.isCarryingABox()).isFalse();
    }
}
```

# Méthodologie pour créer des tests

Un robot **peut porter une caisse d'un poids maximal défini à la construction du robot**. Initialement un robot ne porte pas de caisse. S'il porte déjà une caisse, il ne peut en prendre une autre.

```
@Test
```

```
public void robotCanTakeLightBox() {  
    // initial configuration : a robot and a box  
    Robot robbie = new Robot(15); Box b = new Box(10);  
    // precondition : robot ne porte rien  
    assertThat(robbie.isCarryingABox()).assertFalse();  
    // execution of the tested method  
    robbie.takeBox(b);  
    // postcondition : the carried box is b  
    assertThat(robbie.getCarriedBox()).isSameAs(b);  
}
```

# Méthodologie pour créer des tests

Un robot **peut porter une caisse d'un poids maximal défini à la construction du robot**. Initialement un robot ne porte pas de caisse. S'il porte déjà une caisse il ne peut en prendre une autre.

```
@Test
```

```
public void robotCannotTakeTooHeavyBox() {  
    Robot robbie = new Robot(15);  
    Box b = new Box(20);  
    // precondition : robot does not carry a box  
    assertThat(robbie.isCarryingABox()).assertFalse();  
    // execution of the tested method  
    robbie.takeBox(b);  
    // postcondition : the carried box is b  
    assertThat(robbie.isCarryingABox()).assertFalse();  
}
```

# Méthodologie pour créer des tests

Un robot peut porter une caisse d'un poids maximal défini à la construction du robot. Initialement un robot ne porte pas de caisse. **S'il porte déjà une caisse il ne peut en prendre une autre.**

```
@Test
```

```
public void robotCanTakeOnlyOneBox() {  
    Robot robbie = new Robot(15); Box b1 = new Box(10);  
    Box b2 = new Box(4); robbie.takeBox(b1);  
    // precondition : the carried box is b  
    assertThat(robbie.getCarriedBox()).isSameAs(b1);  
    // execution of the tested method  
    robbie.takeBox(b2);  
    // postcondition: the carried box is b1 and not b2  
    assertThat(robbie.getCarriedBox()).isNotSameAs(b2).isSameAs(b1);  
}
```

# Méthodologie pour créer des tests

Travailler une méthode à la fois :

- 1 Définir la signature de la méthode,
- 2 Écrire la javadoc de la méthode,
- 3 Écrire les tests qui permettront de contrôler que le code écrit pour la méthode est correct = répond au cahier des charges
- 4 Coder la méthode,
- 5 Exécuter les tests définis à l'étape 3, en vérifiant la non-régression,
- 6 Si les tests sont réussis passer à la méthode suivante (étape 1) sinon recommencer à l'étape 4.

Il ne s'agit pas de travailler plus, mais d'être plus efficace.

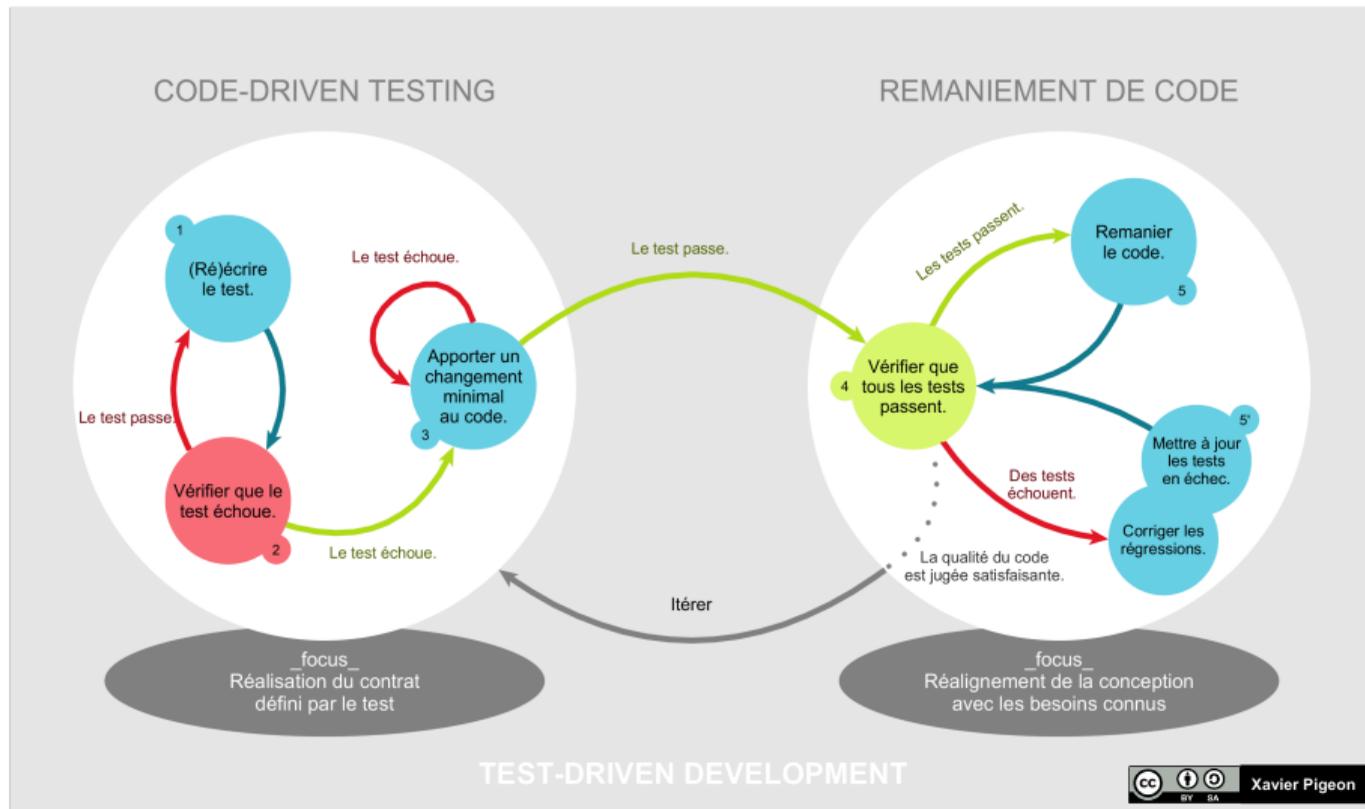
# Test unitaires (à retenir)

- Il est essentiel de tester son code.
- Écrire au moins une méthode de test pour chaque méthode du code de production.
- Il est important de tester tous les types de cas :
  - ▶ cas normaux (utilisation naturelle de la méthode sur une donnée naturelle)
  - ▶ cas limites (utilisation de la méthode sur une donnée “étrange”)
  - ▶ cas anormaux (vérification que les erreurs d'utilisation, c'est-à-dire que les cas d'erreurs sont bien pris en compte et gérés)

## TDD (Test Driven Development)

- Écrire un test qui échoue avant de pouvoir écrire du code
- Écrire une assertion à la fois qui fait échouer un test
- Écrire le minimum de code pour que l'assertion soit satisfaite

# Développer en TDD



## Section 2

# Patrons de création

# Factory

On considère la classe suivante :

```
public interface Button {  
    public void draw();  
}
```

Ainsi qu'une classe l'implémentant :

```
public class SimpleButton implements Button {  
    public void draw() {  
        System.out.println("Simple button.");  
    }  
}
```

La classe SimpleButton est instanciée dans de nombreuses autres classes.

# Ajout d'un autre bouton

```
public class ModernButton implements Button {  
    public void draw() {  
        System.out.println("Modern button.");  
    }  
}
```

Afin d'utiliser ce nouveau bouton, on doit modifier toutes les instanciations présentes dans notre code

→ violation de OCP

# Solution : patron de conception *factory*

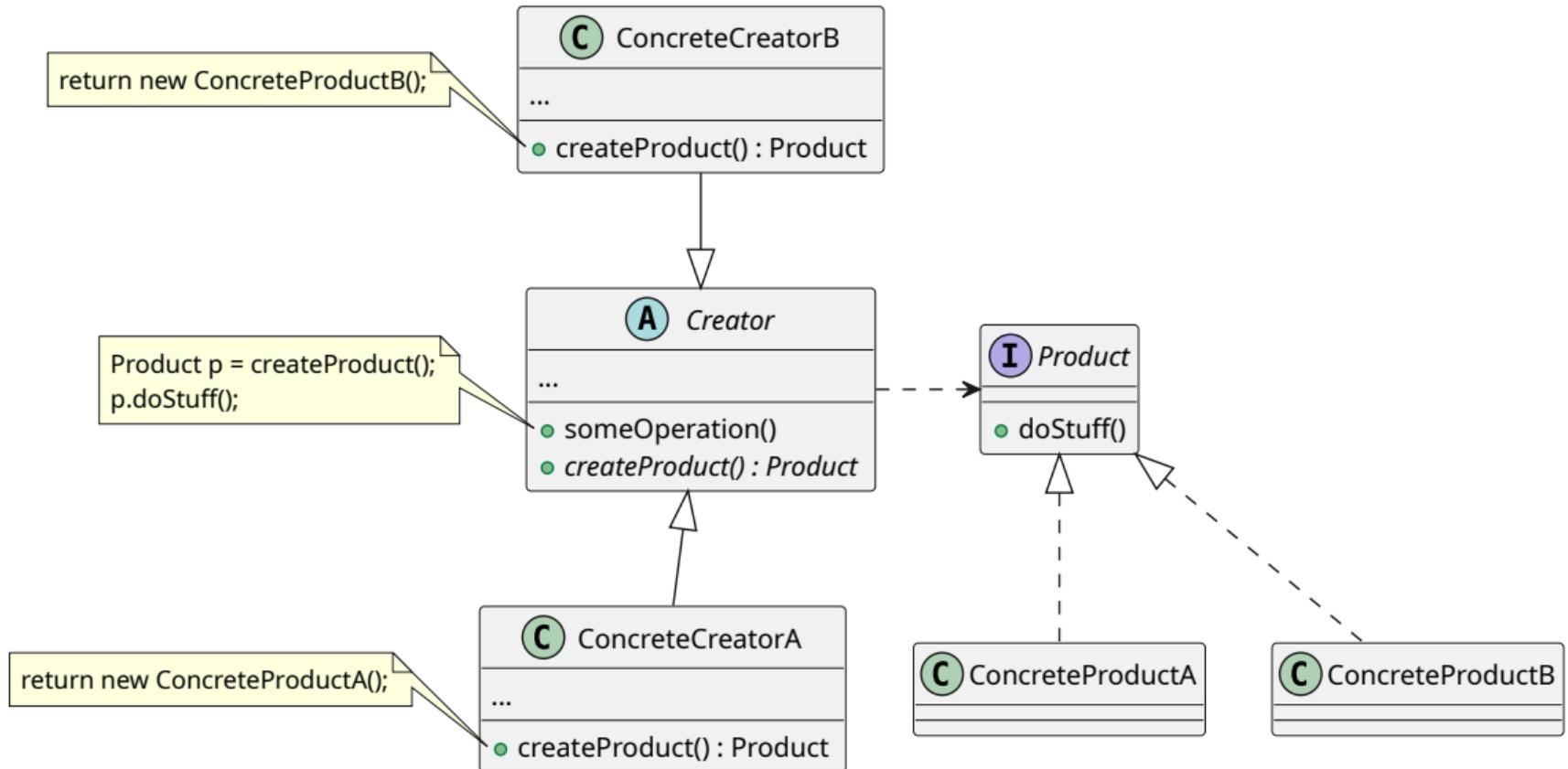
Une fabrique permet d'isoler la création des objets de leurs utilisations :

```
public class ButtonFactory {  
    public Button createButton() {  
        return new SimpleButton();  
    }  
}
```

Toutes les instanciations se font via cette classe. Les modifications nécessaires à l'utilisation de la classe `ModernButton` sont isolées :

```
public class ButtonFactory {  
    public Button createButton() {  
        return new ModernButton();  
    }  
}
```

# Patron de méthode *factory*



# Patron de conception *factory*

## Intention de *factory*

Définit des méthodes pour créer des objets dans une classe mère, mais délègue le choix des types d'objets à créer aux sous-classes.

## Analogie

Vous avez besoin de faire livrer des paquets soit par bateau ou bien par camion. Les deux types de transports sont interchangeables, car ils rendent le même service de base.

On délègue la création des camions ou des navires à des classes filles (chacune ne construisant qu'un type de véhicule).

→ il devient facile de passer d'un type de transport à l'autre

# Fabrique de plusieurs objets

Imaginons que la fabrique fournisse différents éléments :

```
public class Factory {  
    public Button createButton() { return new SimpleButton(); }  
    public TextBox createTextBox() { return new SimpleTextBox(); }  
    public List createList() { return new SimpleList(); }  
}
```

Pour passer du style simple au moderne, il nous faut changer toutes les méthodes de la fabrique :

```
public class Factory {  
    public Button createButton() { return new ModernButton(); }  
    public TextBox createTextBox() { return new ModernTextBox(); }  
    public List createList() { return new ModernList(); }  
}
```

# Patron de conception *abstract factory*

Le code suivant semble résoudre le problème :

```
public class Factory {
    private String type;
    public Factory(String type) { this.type = type; }
    public Button createButton() {
        if (type.equals("modern")) return new ModernButton();
        else return new SimpleButton();
    }
    public TextBox createTextBox() {
        if (type.equals("modern")) return new ModernTextBox();
        else return new SimpleTextBox();
    }
    /* Idem pour createList. */
}
```

# Problème pour l'introduction de variante d'objet

Que se passe-t-il nous avons besoin d'introduire un nouveau type ?

```
public class Factory {
    private String type;
    public Factory(String type) { this.type = type; }
    public Button createButton() {
        if (type.equals("modern")) return new ModernButton();
        else if (type.equals("simple")) return new SimpleButton();
        else return new OtherButton();
    }
    /* Idem pour createTextBox et createList. */
}
```

Toutes les méthodes de la fabrique doivent être modifiées, ce qui est une violation de OCP.

# Introduction d'une interface fabrique

Les fabriques abstraites permettent de corriger ce défaut :

```
public interface AbstractFactory {  
    public Button createButton();  
    public TextBox createTextBox();  
    public List createList();  
}
```

Les classes qui devront instancier des boutons, des boites de texte ou des listes auront à leur disposition une instance d'une classe qui implémente l'interface ButtonFactory :

```
AbstractFactory factory = new SimpleFactory(); //new ModernFactory();  
Button button = factory.createButton();  
/* ... */
```

# Implémentation de l'interface fabrique

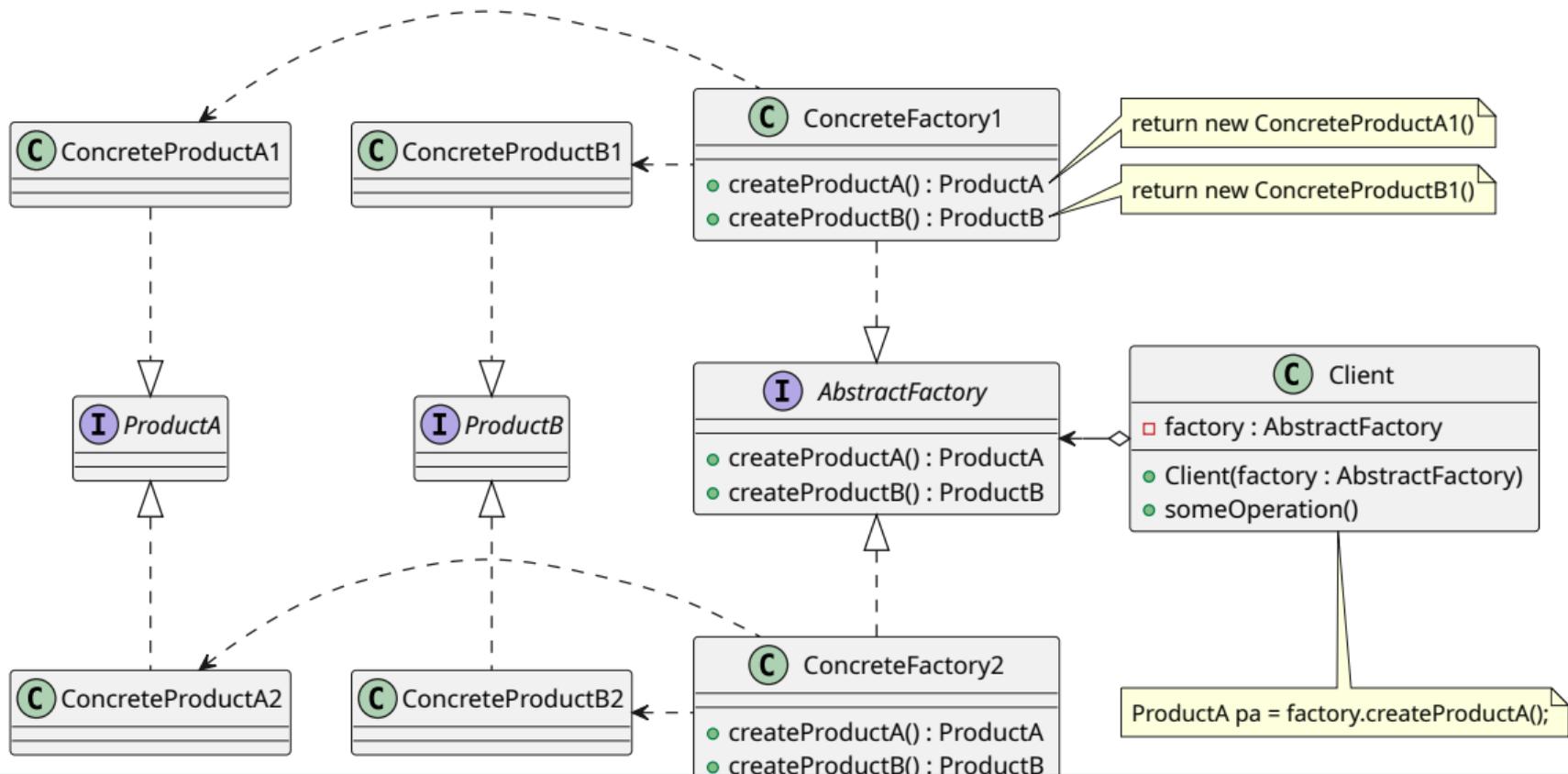
On implémente ensuite une fabrique par type :

```
public class SimpleFactory implements AbstractFactory {
    public Button createButton() { return new SimpleButton(); }
    public TextBox createTextBox() { return new SimpleTextBox(); }
    public List createList() { return new SimpleList(); }
}

public class ModernFactory implements AbstractFactory {
    public Button createButton() { return new ModernButton(); }
    public TextBox createTextBox() { return new ModernTextBox(); }
    public List createList() { return new ModernList(); }
}
```

L'ajout d'un nouveau type s'effectue par l'ajout d'une nouvelle classe, ce qui respecte OCP.

# Patron de conception *abstract factory*



# Patron de conception *abstract factory*

## Intention

Permet de créer des familles d'objets apparentés sans préciser leur classe concrète.

## Utilisation du patron

Vous avez des variantes de familles de produits (Chaise, Sofa et Table) qui sont disponibles dans des variantes (Moderne ou Victorien).

La fabrique abstraite donne une interface pour créer tous les produits d'une famille. Pour chaque variante de produits, on implémente l'interface avec la création des produits de la variante.

# Quand utiliser *Abstract factory* ?

Quand on doit :

- manipuler des produits d'un même thème (garantie de compatibilité entre les objets) ;
- découpler le code client des produits concrets ;
- déléguer la création des objets à une autre classe (SRP) ;
- ajouter des variantes d'objet (OCP).

## Section 3

# Patrons structurels

# Interface Pencil de dessin

Supposons que nous ayons la classe et l'interface suivantes :

```
public class Drawer {
    public void draw(Pencil pencil) {
        pencil.drawLine(0,0,10,10);
        pencil.drawCircle(5,5,5);
        pencil.drawLine(10,0,0,10);
    }
}

public interface Pencil {
    public void drawLine(int x1, int y1, int x2, int y2);
    public void drawCircle(int x, int y, int radius);
}
```

# Classe de dessin Crayon non-compatible

Nous avons également la classe Crayon suivante :

```
public class Crayon {  
    public void dessinerLigne(Point p1, Point p2) {  
        /* ... */  
    }  
    public void dessinerCercle(Point center, int rayon) {  
        /* ... */  
    }  
}
```

On souhaite utiliser la classe Crayon comme un Pencil pour qu'elle puisse être utilisée par la classe Drawer.

## Solution : classe *adapter*

Pour ce faire, nous définissons l'adaptateur suivant :

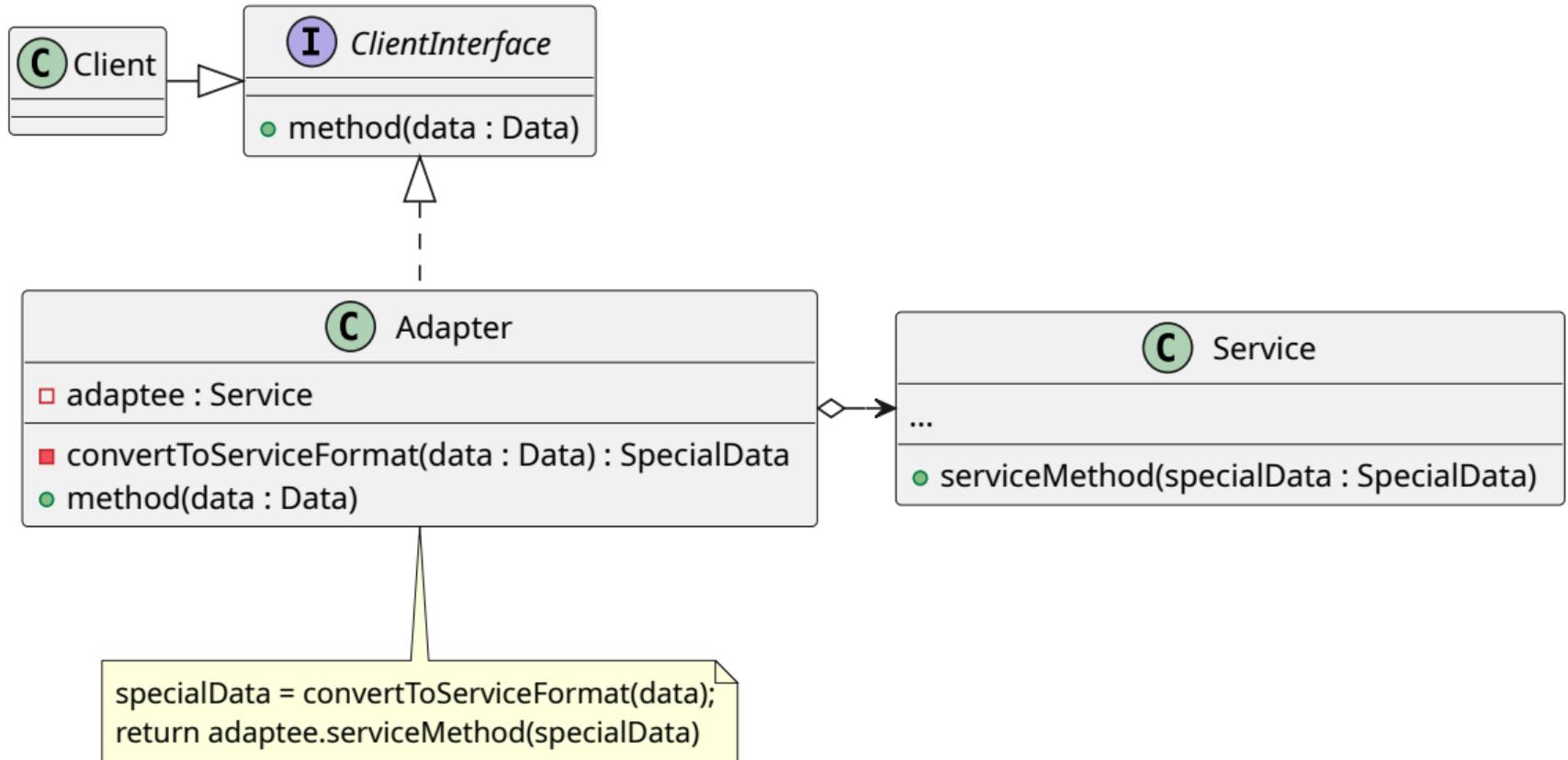
```
public class CrayonAdapter implements Pencil {
    private Crayon crayon;
    public CrayonAdapter(Crayon c) { this.crayon = c; }
    public void drawLine(int x1, int y1, int x2, int y2) {
        crayon.dessinerLigne(new Point(x1, y1), new Point(x2, y2));
    }
    public void drawCircle(int x, int y, int radius) {
        crayon.dessinerCercle(new Point(x, y), radius);
    }
}
```

Cette nouvelle classe est utilisable de la façon suivante :

```
Pencil pencil = new CrayonAdapter(new Crayon());  
Drawer drawer = new Drawer();  
drawer.draw(pencil);
```

On n'a pas modifié les classes existantes (respect OCP).

# Patron de conception *adapter*



# Patron de conception *adapter*

## Intention

Permet de faire collaborer des objets ayant des interfaces normalement incompatibles.

## Utilisation du patron

Classe déjà écrite qui délègue certaines opérations à une autre classe. Changer la classe déléguée en utilisant une autre classe pas tout à fait compatible (opérations similaires, mais pas la même interface).

Solution :

- 1 Créer une interface pour les opérations déléguées
- 2 Créer un adaptateur qui convertit l'interface de votre nouvelle classe

# Avantages et inconvénients

## Avantages :

- Découpler l'interface ou le code de conversion des données, de la logique métier du programme (SRP).
- Permet de réutiliser du code existant sans le modifier (OCP)

## Désavantage :

- Peut rajouter de la complexité inutile dans le code (parfois plus simple de recoder la classe service).