Initiation génie logiciel : conception logicielle (1/2)

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

20 septembre 2024

amU Faculté des sciences Aix Marseille Université

Section 1

Introduction

Conception Logicielle

Objectif d'une bonne conception logicielle

Minimiser les ressources humaines requises pour construire et maintenir un logiciel.

Méthodologie de développement

- Séparer et découpler les parties des projets en paquets/classes ;
- Limiter et localiser les modifications lors des évolutions ;
- Tester le code ;
- Nettoyer le code (le rendre le plus lisible possible).

Coder proprement

Un code propre :

- respecte les attentes des utilisateurs
- est fiable
- peut évoluer facilement/rapidement
- est compréhensible par tous

Points importants

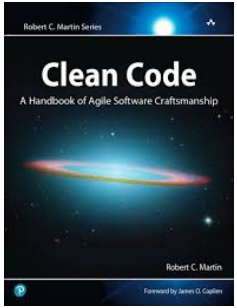
- Nommage des éléments du code : donner l'intention avec le nom
- Fonctions courtes
- Éviter commentaire inutile (réécrire le code plutôt que le commenter)
- Tests unitaires

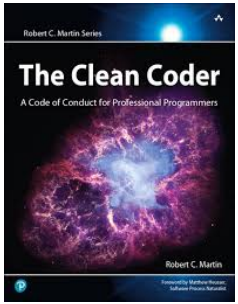
Comment programmer proprement ?

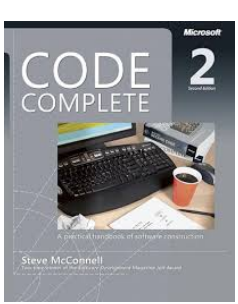
Pour programmer proprement dans un langage objet, il faut :

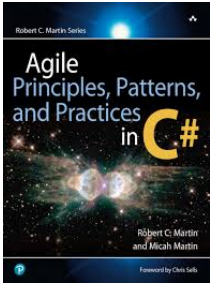
- écrire du code lisible (par un autre humain)
- relire et améliorer le code

Guides de bonnes pratiques :









Exemple de code mal écrit

```java
public class Rec {
  int l, L;

  public Rec(int l, int L) {
    this.l = l;
    this.L = L;
  }

  public static int compte(List<Rec> r){
    for(int i = 0, int s = 0; i < r.size(); i++)
      if(r.get(i).l == r.get(i).L)
        s++;
    return s;
  }
```

Après un petit peu de *refactoring*

```java
public class Rectangle {
  int width, height;
  public Rectangle(int width, int height) {
    this.width = width;
    this.height = height;
  }
  private boolean isSquare(){ return width == height; }
  static int countSquares(List<Rectangle> rectangles){
    int squareCount = 0;
    for(Rectangle rectangle : rectangles)
      if(rectangle.isSquare())
        squareCount++;
    return squareCount;
  }
```

Les cinq principes (pour créer du code) SOLID

- **Single Responsibility Principle (SRP)** : Une classe ne doit avoir qu'une seule responsabilité
- **Open/Closed Principle (OCP)** : Programme ouvert pour l'extension, fermé à la modification
- **Liskov Substitution Principle (LSP)** : Les sous-types doivent être substituables par leurs types de base
- **Interface Segregation Principle (ISP)** : Éviter les interfaces qui contiennent beaucoup de méthodes
- **Dependency Inversion Principle (DIP)** :
  - Les modules d'un programme doivent être indépendants
  - Les modules doivent dépendre d'abstractions

Section 2

Tests

Différents types de tests

Règle

Un code non testé n'a aucune valeur $\Rightarrow$ 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).
- **Test système** : Tester le logiciel complet
- **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 (1/2)

- 1 classe de test = un ensemble de méthodes de test
- 1 classe de test par classe à tester
- nom d'une classe de test : *NameTestedClass*Test
- 1 méthode de test = 1 cas de test
- 1 cas de test = (description, données d'entrée, résultat attendu)
- nom d'une méthode de test : test*NameTestedMethod*

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.

$\Rightarrow$ 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 (2/3)

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

public class NameTestedClassTest {
  @Test
  void testNameTestedMethod(){
    /* 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 : Box

```java
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

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

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

Méthodes de test

- 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

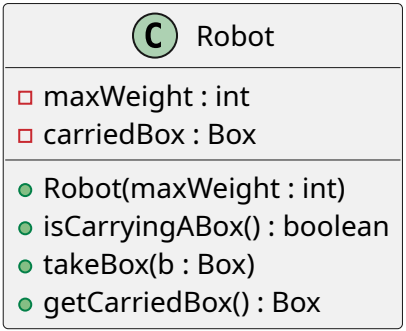
- plusieurs méthodes de tests peuvent être nécessaires pour tester la correction d'une méthode
- principes :

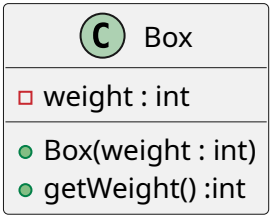
  1. créer la situation initiale et vérifier les « préconditions »
  2. appeler la méthode testée
  3. à l'aide d'assertions, vérifier les « postconditions » = situation attendue après l'exécution de la méthode

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()).isFalse();
  // 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()).isFalse();
  // execution of the tested method
  robbie.takeBox(b);
  // postcondition : the carried box is b
  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 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.

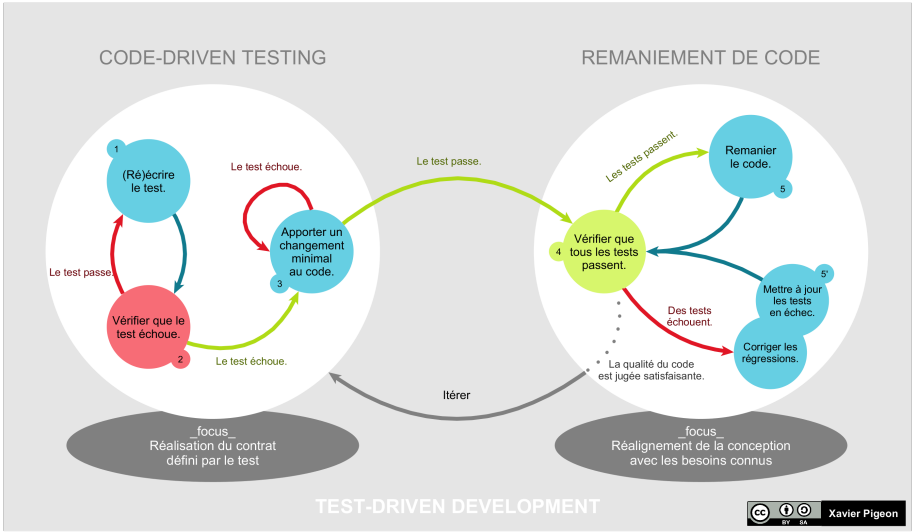
Test unitaires (à retenir)

- 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 3

Principes SOLID et patrons de conception

Les cinq principes (pour créer du code) SOLID

- **Single Responsibility Principle (SRP)** : Une classe ne doit avoir qu'une seule responsabilité
- **Open/Closed Principle (OCP)** : Programme ouvert pour l'extension, fermé à la modification
- **Liskov Substitution Principle (LSP)** : Les sous-types doivent être substituables par leurs types de base
- **Interface Segregation Principle (ISP)** : Éviter les interfaces qui contiennent beaucoup de méthodes
- **Dependency Inversion Principle (DIP)** :
  - Les modules d'un programme doivent être indépendants
  - Les modules doivent dépendre d'abstractions

Patrons de conception (*design patterns*)

Les patrons de conception donnent des solutions (sous forme de schémas à personnaliser) pour résoudre un problème récurrent de conception logicielle.

Trois groupes principaux de patrons :

- de création qui fournissent des mécanismes de création d'objets ;
- structurels qui expliquent comment assembler des objets et des classes en de plus grandes structures ;
- comportementaux qui mettent en place une communication efficace et répartissent les responsabilités entre les objets.

Listes des patrons de conception

- Patrons de création
  - Fabrique (*factory*)
  - Fabrique abstraite (*abstract factory*)
  - Monteur (*builder*)
  - Prototype (*prototype*)
  - Singleton (*singleton*)
- Patrons structurels
  - Adaptateur (*adapter*)
  - Pont (*bridge*)
  - Composite (*composite*)
  - Décorateur (*decorator*)
  - Façade (*facade*)
  - Poids mouche (*flyweight*)
  - Procuration (*proxy*)
- Patrons comportementaux
  - Chaîne de responsabilité (*chain of responsibility*)
  - Commande (*command*)
  - Itérateur (*iterator*)
  - Médiateur (*mediator*)
  - Mémento (*memento*)
  - Observateur (*observer*)
  - État (*state*)
  - Stratégie (*strategy*)
  - Patron de méthode (*template method*)
  - Visiteur (*visitor*)

Classe ListSum

Supposons que nous ayons la classe suivante :

```
public class ListSum {
  private int[] list = new int[10];
  private int size = 0;
  public void add(int value) {
    list[size] = value;
    size++;
  }
  public int eval() {
    int result = 0;
    for (int i = 0; i < size; i++)
      result += list[i];
    return result;
  }
```

Classe ListProduct

Supposons que nous ayons aussi la classe suivante (très similaire) :

```
public class ListProduct {
  private int[] list = new int[10];
  private int size = 0;
  public void add(int value) {
    list[size] = value;
    size++;
  }
  public int eval() {
    int result = 1;
    for (int i = 0; i < size; i++)
      result *= list[i];
    return result;
  }
```

Refactoring pour isoler la répétition

Les deux classes sont très similaires et il y a de la répétition de code.

Il est possible de *refactorer* (réécrire le code) les classes précédentes de façon à isoler les différences dans des méthodes :

```
public class ListSum {
  public int eval() {
    int result = neutral();
    for (int i = 0; i < size; i++)
      result = compute(result, list[i]);
    return result;
  }
  private int neutral() { return 0; }
  private int compute(int a, int b) { return a+b; }
}
```

Refactoring pour isoler la répétition

Il est possible de *refactorer* les classes précédentes de façon à isoler les différences dans des méthodes :

```
public class ListProduct {
  public int eval() {
    int result = neutral();
    for (int i = 0; i < size; i++)
      result = compute(result, list[i]);
    return result;
  }
  private int neutral() { return 1; }
  private int compute(int a, int b) { return a*b; }
}
```

Comment éviter la répétition ?

Après la *refactorisation* du code :

- seules les méthodes `neutral` et `compute` diffèrent
- il serait intéressant de pouvoir supprimer les duplications de code

Deux solutions :

- La délégation en utilisant une interface et l'agrégation $\rightarrow$ patron de conception **Stratégie**.
- L'extension et les classes abstraites $\rightarrow$ patron de conception **Patron de méthode**.

Solution Statégie : interface Operator

Nous allons externaliser les méthodes `neutral` et `compute` dans deux nouvelles classes. Elles vont implémenter l'interface `Operator` :

```
public interface Operator {
  public int neutral();
  public int compute(int a, int b);
}
public class Sum implements Operator {
  public int neutral() { return 0; }
  public int compute(int a, int b) { return a+b; }
}
public class Product implements Operator {
  public int neutral() { return 1; }
  public int compute(int a, int b) { return a*b; }
}
```

Délégation

Les classes `ListSum` et `ListProduct` sont fusionnées dans une unique classe `List` qui délègue les calculs à un objet qui implémente l'interface `Operator` :

```
public class List {
  /* attributs et méthode add */
  private Operator operator;
  public List(Operator operator){
    this.operator = operator;
  }
  public int eval() {
    int result = operator.neutral();
    for (int i = 0; i < size; i++)
      result = operator.compute(result, list[i]);
    return result;
  }
```

Délégation

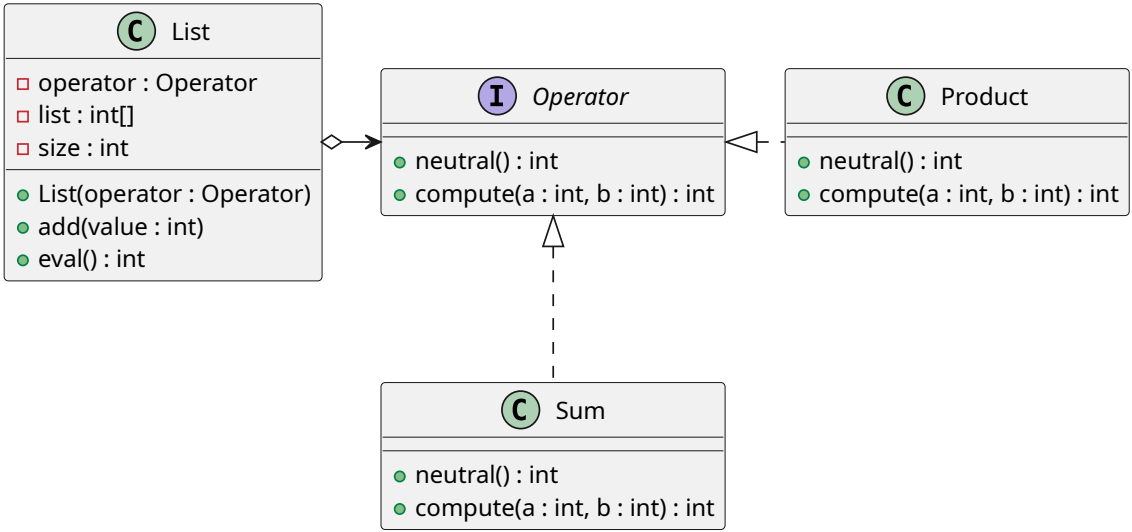
Utilisation des classes `ListSum` et `ListProduct` :

```
ListSum listSum = new ListSum(); listSum.add(2);
listSum.add(3); System.out.println(listSum.eval());
ListProduct listProduct = new ListProduct();
listProduct.add(2); listProduct.add(3);
System.out.println(listProduct.eval());
```

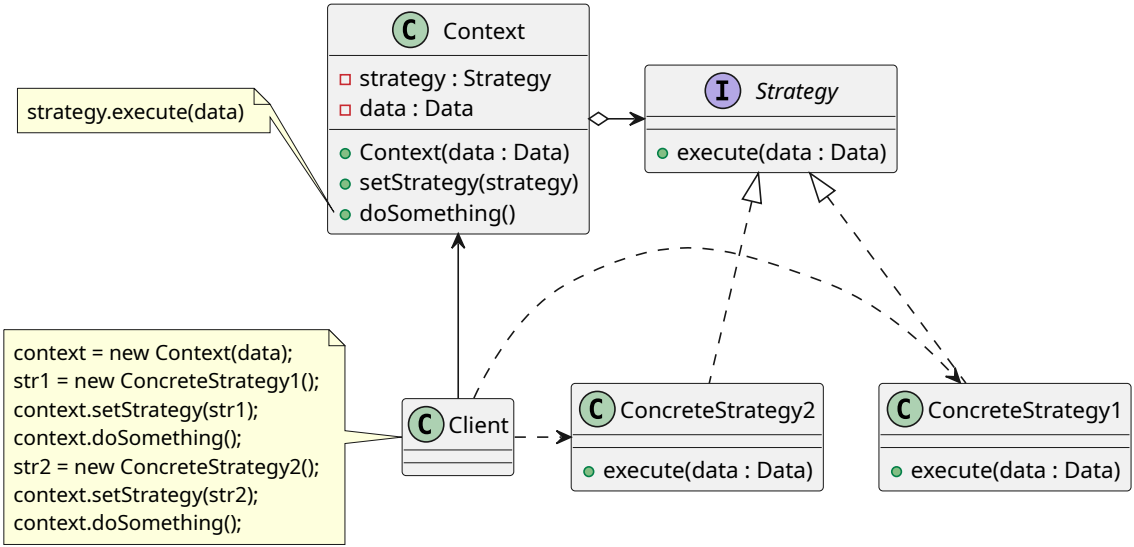
Utilisation après la *refactorisation* du code :

```
List listSum = new List(new Sum()); listSum.add(2);
listSum.add(3); System.out.println(listSum.eval());
List listProduct = new List(new Product());
listProduct.add(2); listProduct.add(3);
System.out.println(listProduct.eval());
```

Diagramme de la solution avec Stratégie



Patron de conception Stratégie



Patron de conception Stratégie

Intention

Définir une famille d'algorithmes, encapsuler chacun d'entre eux et les rendre interchangeables.

Analogie

Pour vous rendre à l'aéroport, vous pouvez prendre

- le bus,
- appeler un taxi ou
- enfourcher votre vélo.

Ce sont vos stratégies de transport et vous en choisissez une en fonction de vos besoins.

Quand utiliser le patron Stratégie ?

Cas d'utilisation

Une classe définit un comportement spécifique avec plusieurs manières de le réaliser.

Solution

- Séparer les différentes manières de réaliser le comportement de la classe en classes séparées appelées stratégies (partageant la même interface).
- La classe originale (le contexte) garde un attribut qui garde une référence vers une des stratégies.
- Plutôt que de s'occuper de la tâche, le contexte la délègue à l'objet stratégie associé.
- Le contexte n'a pas la responsabilité de la sélection de l'algorithme adapté, c'est le client qui lui envoie la stratégie.

Solution utilisant une classe abstraite

```
public abstract class List {
  private int[] list = new int[10];
  private int size = 0;
  public void add(int value) { list[size] = value; size++; }
  public int eval() {
    int result = neutral(); // utilisation d'une méthode abstraite
    for (int i = 0; i < size; i++)
      result = compute(result, list[i]); // idem
    return result;
  }
  public abstract int neutral(); // méthode abstraite
  public abstract int compute(int a, int b); // idem
}
```

Rappel : mot-clé `abstract`

Classe abstraite

- On peut mettre `abstract` devant le nom de la classe à sa définition pour signifier qu'une classe est abstraite.
- Une classe est abstraite si des méthodes ne sont pas implémentées.
  $\Rightarrow$ Classe abstraite = classe avec des méthodes abstraites
- Tout comme pour une interface, une classe abstraite n'est pas instanciable.

Méthode abstraite

- `abstract` devant le nom du type de retour de la méthode à sa définition pour signifier qu'une méthode est abstraite.
- Méthode abstraite = méthode sans code, juste la signature (type du retour et des paramètres) est définie

Classes abstraites et extension

Tout comme pour les interfaces, il n'est pas possible d'instancier une classe abstraite :

```
List list = new List(); // erreur
System.out.println(list.eval()) // car que faire ici ?
```

Nous allons étendre la classe `List` afin de récupérer les attributs et les méthodes de `List` et définir le code des méthodes abstraites :

```
public class ListSum extends List {
  public int neutral() { return 0; }
  public int compute(int a, int b) { return a+b; }
}
```

Classes abstraites et extension

La classe `ListSum` n'est plus abstraite, toutes ses méthodes sont définies :

- les méthodes `add` et `eval` sont définies dans `List` et `ListSum` hérite du code de ses méthodes.
- les méthodes `neutral` et `compute` qui étaient abstraites dans `List` sont définies dans `ListSum`.

On peut donc instancier la classe `ListSum` :

```
ListSum listSum = new ListSum();
listSum.add(3);
listSum.add(7);
System.out.println(listSum.eval());
```

Classes abstraites et extension

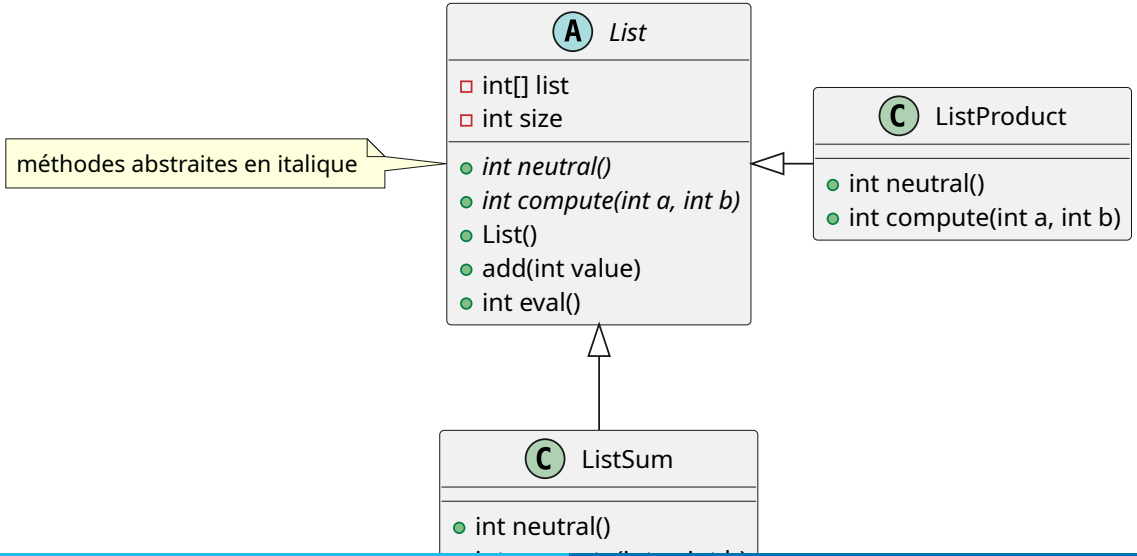
On peut procéder de manière similaire pour créer une classe `ListProduct`

```
public class ListProduct extends List {
  public int neutral() { return 1; }
  public int compute(int a, int b) { return a*b; }
}
```

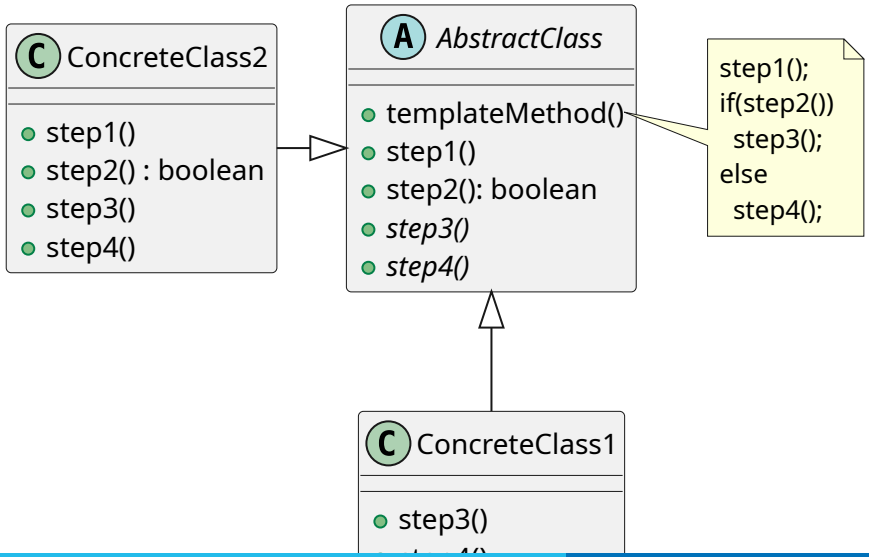
La classe `ListProduct` n'est plus abstraite, toutes ses méthodes sont définies :

```
ListProduct listProduct = new ListProduct();
listProduct.add(3);
listProduct.add(7);
System.out.println(listProduct.eval());
```

Diagramme de la solution avec extension



Patron de conception Patron de méthode



Patron de conception Patron de méthode

Intention

- Définir le squelette d'un algorithme dans une classe mère.
- Laisser les sous-classes redéfinir le code des étapes de l'algorithme sans changer sa structure.

Analogie

Les étapes pour construire une maison sont toujours les mêmes dans le même ordre :

1. poser les fondations,
2. poser la charpente,
3. monter les murs,
4. installer la plomberie pour l'eau et les câbles pour l'électricité.

Chaque étape de construction peut être légèrement modifiée pour différencier un peu la maison des autres.

Quand utiliser le patron de méthode ?

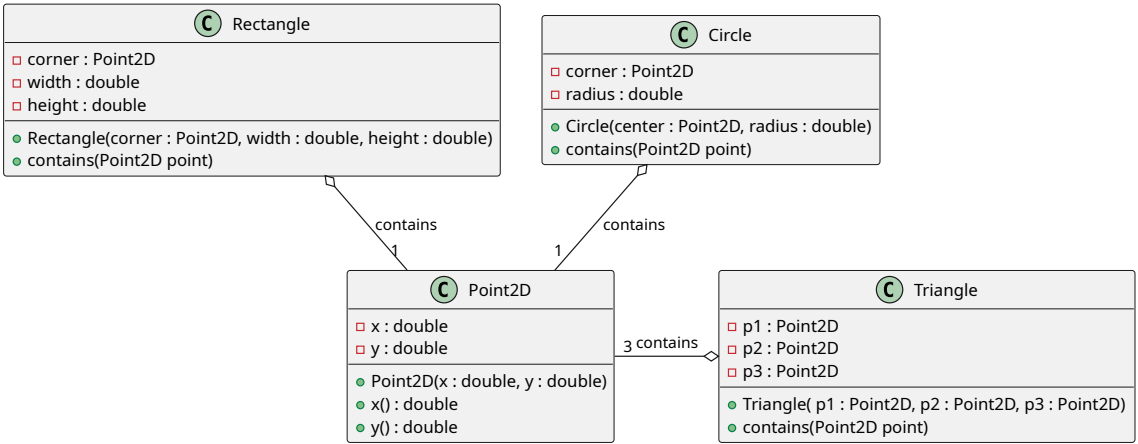
Cas d'utilisation

Plusieurs classes ont la même structure pour un algorithme avec le même enchainement d'étapes, mais une implémentation différente de ces étapes.

Solution

- Découper un algorithme en une série d'étapes et transformer ces étapes en méthodes potentiellement abstraites ;
- Créer une méthode socle exécutant l'algorithme avec des appels à ces méthodes ;
- Fournir une sous-classe concrète en implémentant toutes les étapes abstraites et redéfinissant certaines d'entre elles si besoin

Formes géométriques : coder une union de formes ?



Interface Shape

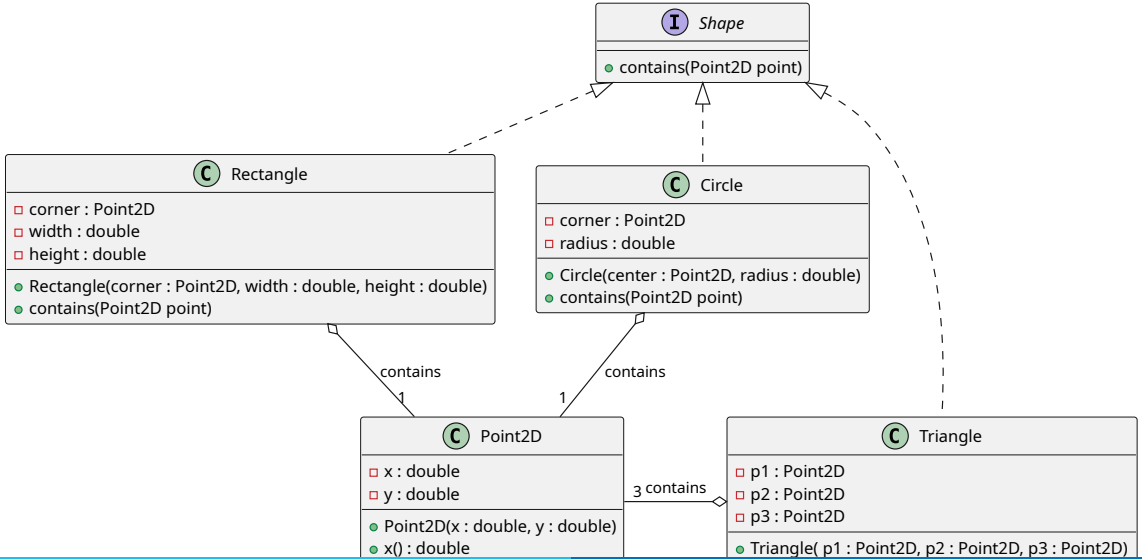
Si on veut une liste qui puisse contenir à la fois des rectangles, des triangles et des cercles, il faut une interface :

```
public interface Shape {
  boolean contains(Point2D point);
}

public class Rectangle implements Shape{...}
public class Triangle implements Shape{...}
public class Circle implements Shape{...}
```

De cette manière on peut ajouter facilement une nouvelle forme : le principe SOLID OCP est respecté.

Interface Shape

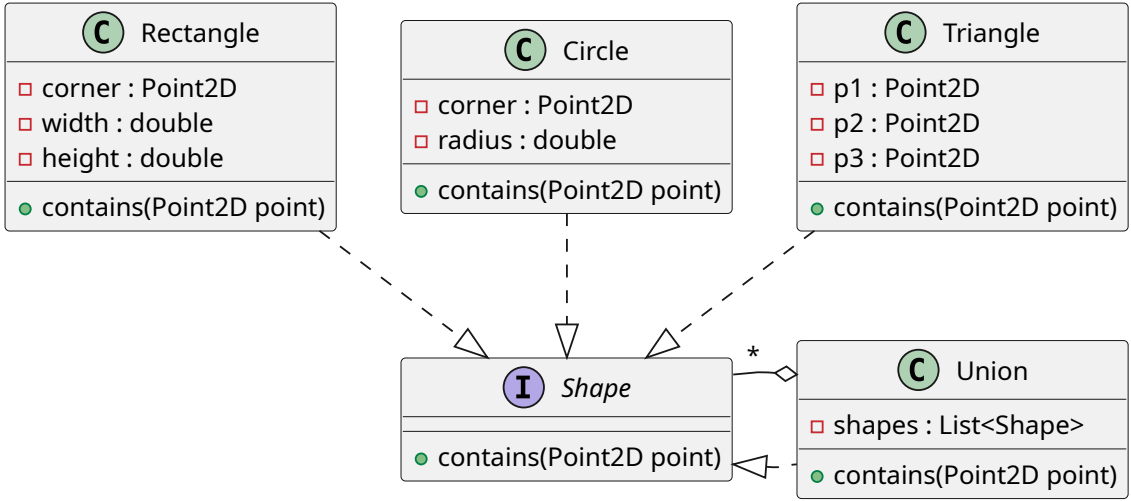


Union de forme

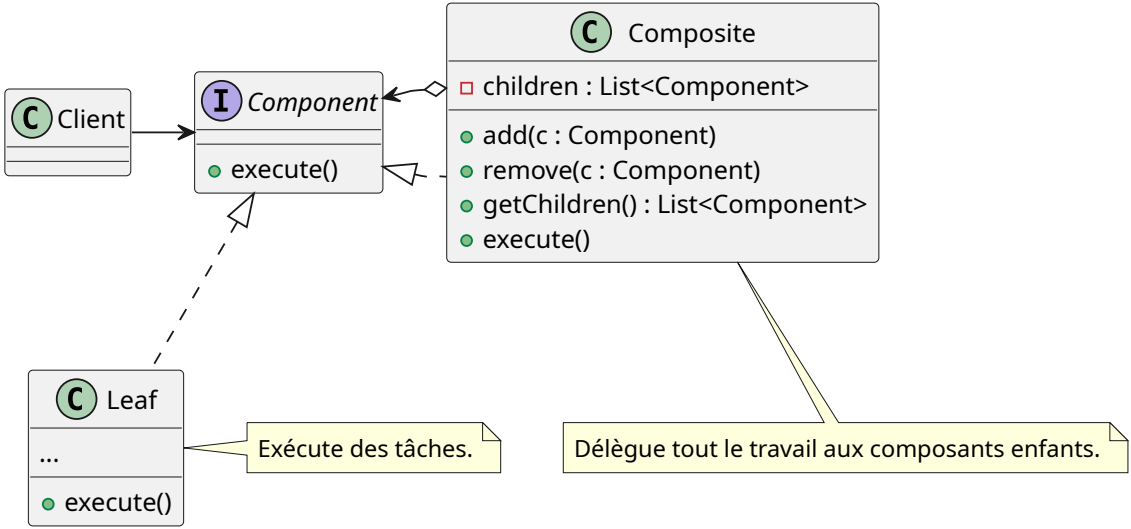
```
public class Union {
  List<Shape> shapes;
  public boolean contains(Point2D point){
    for(Shape shape : shapes)
      if(shape.contains(point))
        return true;
    return false;
  }
}
```

On peut remarquer que `Union` a une méthode `contains` et peut donc implémenter l'interface `Shape`.

Diagramme de classes



Patron de conception composite



Patron de conception composite

Intention

Permet d'organiser des objets en une structure d'arbre pour réaliser une tâche demandant une action de chaque objet.