Aix*Marseille université
Socialement engagée

Année 2022-23

Domaine Sciences et Technologies
Licence informatique : parcours MI
Approfondissements informatique :
SOLID et patrons de conception
Code UE : SIN6U28T

Principes SOLID et patrons de conception

Principes SOLID et patrons de conception

Arnaud Labourel

1 Principes SOLID

1.1 Introduction

1.1.1 Cinq principes pour un code maintenable

En programmation orientée objet, il existe cinq principes de conception destinés (regroupés sous l'acronyme SOLID) qui visent à produire des architectures logicielles plus compréhensibles, flexibles et maintenables. Ces principes sont un sous-ensemble de nombreux principes promus par l'ingénieur logiciel et instructeur américain Robert Cecil Martin (familièrement connu sous le nom *Uncle Bob*). Bien qu'ils s'appliquent à toute conception orientée objet, les principes SOLID peuvent également former une philosophie de base pour des méthodologies telles que le développement agile. La théorie des principes SOLID a été introduite par Martin dans son article *Design Principles and Design Patterns* de 2000, bien que l'acronyme SOLID ait été introduit plus tard par Michael Feathers.

Les cinq principes SOLID sont les suivants :

- **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

Le but de ces principes est donc de garantir la maintenabilité d'un programme, c'est-à-dire sa capacité à :

- absorber les changements avec un minimum d'effort ;
- implémenter les nouvelles fonctionnalités sans toucher aux anciennes ;
- modifier les fonctionnalités existantes en modifiant localement le code.

L'application des principes SOLID a pour objectifs :

- de limiter les modules impactés ;
- de simplifier les tests ;
- de rester conforme aux spécifications qui n'ont pas changé.

1.1.2 Approche qualité des 5S

Les principes SOLID sont donc ceux qui seront étudiés dans ce cours et constitueront son ossature. Dans la suite de ce cours, nous allons donc détailler ces cinq principes et expliquer pourquoi il est important de les respecter afin d'obtenir du code maintenable. Mais avant cela, il est utile de nous intéresser à une approche qualité pour la gestion de projet, venue du Japon au milieu du siècle dernier et qui peut s'appliquer à l'informatique. Cette approche ou philosophie des 5S peut être résumé par les cinq points suivants :

- *Seiri* ("s'organiser") : les différents éléments d'un code doivent être structurés et aisément identifiables.

Ainsi une action aussi anodine que de nommer les identifiants, méthodes et classes ne l'est pas tant que cela et doit requérir toute votre attention ;
— *Seiton* ("situer") : un morceau de code doit se trouver là où l'on s'attend logiquement à ce qu'il se trouve. Si ce n'est pas le cas, cela veut dire qu'il n'est pas à sa place et que la structure du code n'a pas été pensée correctement ;
— *Seiso* ("scintiller") : l'espace de travail doit être propre ! Pensez à la cuisine d'un grand restaurant : on ne travaille pas sur un plan de travail comportant de la vaisselle sale, des ingrédients d'un autre plat, ... Au niveau du code la présence de commentaires non informatifs ou de code ancien désactivé par une mise en commentaire constitue une pollution de l'espace de travail à laquelle il faut remédier ;
— *Seiketsu* ("standardiser") : dans un travail en équipe il faut que des conventions soient respectées, que chaque développeur suive les mêmes règles pour que le code conserve une homogénéité ;
— *Shutsuke* ("suivi") : suivre le travail des autres permet de s'interroger sur ses propres pratiques et d'évoluer positivement (en tout cas il faut l'espérer...).

On peut se poser la question de pourquoi s'imposer ces règles qui viennent s'ajouter à celles qu'il faut déjà suivre pour qu'un programme soit fonctionnel. En fait, le but de ces règles est de gagner du temps. Le temps de travail des personnes participant au projet est généralement la ressource qui est la plus coûteuse dans un projet. Cela peut paraître contre-intuitif, car appliquer les règles énoncées ci-dessus prend clairement du temps. Cependant, ce temps n'est pas perdu, car l'objectif est d'en gagner dans le futur. En effet, l'objectif est de garder un bon cadre de travail afin d'être efficace. C'est un peu le même principe qu'avoir une pièce bien rangée vous fait globalement gagner du temps, car cela vous permet d'accéder facilement à vos affaires, et ce même si ranger prend du temps.

Comme vous venez de le voir, en termes de bonnes pratiques on parle souvent de règles pour évoquer les principes recommandés par une approche ou l'autre. Il est important de comprendre qu'il s'agit bien de recommandations et non de lois immuables. Tout ce que nous allons voir dans la suite est à adapter au contexte dans lequel vous allez l'appliquer, il n'y a pas de loi universelle permettant d'obtenir à coup sûr un code propre et maintenable, seulement des indications de pratiques qui ont été reconnues profitables. De surcroît, il se peut que certaines considérations, notamment des contraintes de performances, puissent rendre difficile l'application de certaines bonnes pratiques de programmation dans des cas très spécifiques.

1.1.3 La vie d'un programme

On peut se demander pourquoi il est si important d'avoir une certaine méthodologie (et donc de se fixer des règles et d'appliquer des bonnes pratiques) lorsqu'on participe à un projet de développement logiciel d'envergure. Pour justifier cette approche, nous allons considérer l'historique de la vie d'un programme quelconque. On peut faire une analogie avec la vie humaine et découper la vie d'un projet en 5 phases :

— La naissance : tout le monde vient s'extasier sur le beau bébé qui vient de naître. Les parents sont fiers de présenter leur rejeton. Le code est beau, pur, les développeurs ont porté une grande attention à sa création.
— L'enfance : en commençant à marcher, courir, sauter, etc., l'enfant se blesse. Pour réparer un premier bug ou ajouter rapidement une nouvelle fonctionnalité les développeurs travaillent à la va-vite, ils ajoutent une *rustine* (c'est-à-dire une modification du code sommaire et temporaire visant à corriger rapidement

un bug ou dysfonctionnement) au projet. Le code devient donc moins pur et moins beau.
— L'adolescence : le moment de la rébellion. Il faut intervenir de plus en plus souvent, car les bugs se multiplient. Les développeurs multiplient les rustines et le code devient de moins en moins maintenable.
— L'âge adulte : il faut avancer coûte que coûte. Les modifications de code précédentes sont un lourd héritage et toute amélioration ou correction prend énormément de temps. Parfois la correction d'un bug déclenche l'apparition de nombreux autres bugs. Mais il faut continuer à avancer : le programme est en production, il n'y a pas d'autre choix que de perdre un temps précieux dès qu'il faut modifier le code.
— La vieillesse : certaines fonctionnalités sont défaillantes, mais on ne peut plus les réparer. À force d'ajouter des rustines le code n'est plus maintenable, plus aucun développeur ne peut effectuer la moindre modification sans tout casser. Il n'y a plus rien à faire qu'attendre une mort inexorable.

Respecter les bonnes pratiques de conception et de développement permet à un code de vieillir sereinement, de conserver ces fonctionnalités le plus longtemps possible. C'est comme pour un être humain : mener une vie d'excès ne permet pas d'envisager une longue vie en ayant la jouissance complète de ces capacités intellectuelles et physiques. Avec le code informatique, c'est encore plus important, car on développe rarement seul. Une mauvaise hygiène de vie aura des répercussions sur les développeurs faisant également partis du projet et fera nécessairement naître des tensions.

1.1.4 Le zen du développement

En Python, l'un des documents qui définit le langage est une ôde aux bonnes pratiques. Il s'agit du PEP 20 (*Python Enhancement Proposal*), intitulé le *Zen of Python* :

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

En français cela donne :

```
Le beau est préférable au laid.
L'explicite est préférable à l'implicite.
Le simple est préférable au complexe.
Le complexe est préférable au compliqué.
L'horizontal est préférable à l'imbriqué.
L'aéré est préférable au dense.
La lisibilité compte.
Les cas spéciaux ne le sont pas assez pour transgresser les règles.
Sauf si le cas pratique bat le cas théorique.
Les erreurs ne devraient jamais arriver silencieusement.
Sauf si on les a explicitement rendues silencieuses.
En cas de doute, ne tentez pas de deviner.
Il devrait y avoir une, et de préférence une seule, manière évidente de le faire.
Même si cette manière peut ne pas sembler évidente au premier abord sauf si vous êtes néerlandais.
Même si jamais est souvent mieux que tout de suite.
Si l'implémentation est difficile à expliquer, c'est que c'est une mauvaise idée.
Si l'implémentation est facile à expliquer, c'est que c'est peut-être une bonne idée.
Les espaces de noms sont une brillante idée, créons-en plus !
```

Dans cette introduction, nous avons pu observer qu'il n'y avait pas que les principes SOLID qui existent en termes de bonnes pratiques de programmation. D'ailleurs nous avions même évoqué d'autres bonnes pratiques dans le premier cours portant sur la gestion de version et les tests. Toutes ces recommandations ont vu le jour à peu près à la même période. Comme on vient de l'expliquer elles ont toutes le même objectif : coder proprement de manière à conserver un code maintenable le plus longtemps possible. Finalement, à y regarder un peu plus en détail, que ce soit la philosophie des 5S, le Zen de Python ou les principes SOLID, tous reprennent plus ou moins les mêmes idées... il y a donc sans doute un enseignement intéressant à en tirer !

1.2 Principe de responsabilité unique

Le "S" de SOLID signifie *Single Responsability Principle*, également généralement noté SRP. Robert Cecil Martin dans son livre "Agile Software Development, Principles, Patterns, and Practices" définit ce principe de la manière suivante :

> Single Responsability Principle : A class should have only one reason to change.

En français, cela donne :

> Principe de responsabilité unique : une classe ne doit avoir qu'une seule raison de changer.

Les classes (et les méthodes) ne devraient avoir qu'une seule fonctionnalité.

Considérons une implémentation d'un jeu dans lequel il a des tours de jeu et un comptage de points comme le bowling. On pourrait imaginer qu'un tel jeu aura une classe `Game` qui aura la responsabilité de se souvenir du numéro du tour en cours (à quel carreau on est dans le cas du bowling) ainsi que du calcul du score (quel est le nombre de points obtenu par chacun des joueurs). Pour respecter le principe SRP, il faudrait séparer ces deux fonctionnalités en deux classes de sorte que chaque classe n'ait qu'une seule responsabilité. La classe `Game` garderait la responsabilité du suivi des tours alors qu'une nouvelle classe `Scorer` aurait la responsabilité de calculer le score.

On peut se poser la question de savoir pourquoi il est important de séparer ces deux responsabilités dans des classes distinctes. La raison en est que chaque responsabilité correspond à une direction dans lequel on peut faire un changement dans la classe. Si une classe possède plus d'une responsabilité, elle aura donc plus d'une raison de changer. Si une classe a plusieurs responsabilités, elles sont couplées. Dans ce cas, la modification d'une des responsabilités nécessite de :

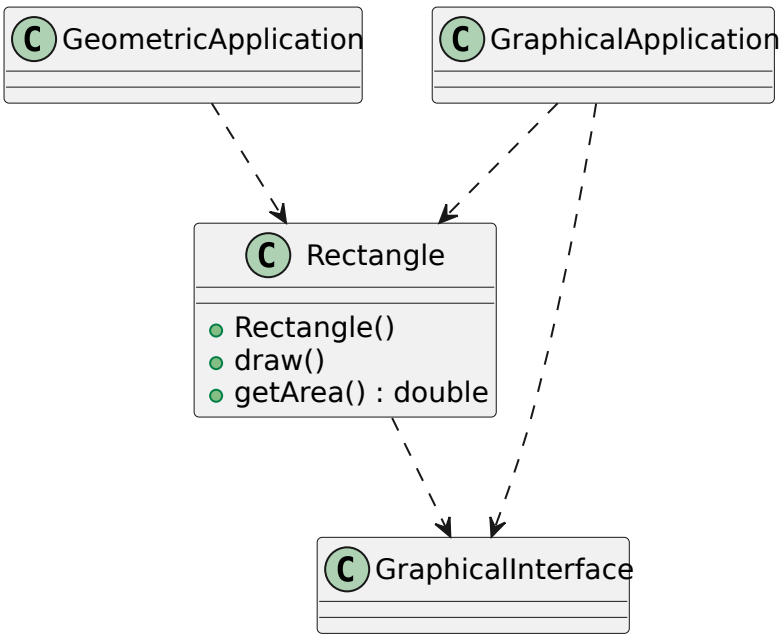
— tester à nouveau l'implémentation des autres responsabilités ;
— modifier potentiellement les autres responsabilités (les modifications apportées à une responsabilité pouvant compromettre la capacité de la classe à assurer ses autres responsabilités) ;
— déployer à nouveau les autres responsabilités.

Ce type de couplage conduit à des conceptions fragiles qui se brisent de manière inattendue lorsqu'elles ont besoin d'être modifiées. Une modification dans les spécifications d'une des responsabilités d'une classe peut entraîner l'introduction de bugs et donc une perte de temps.

Séparer les responsabilités et donc respecter SRP a de nombreux avantages :

— Diminution de la complexité du code
— Amélioration de la lisibilité du code
— Meilleure organisation du code
— Modification locale lors des évolutions
— Augmentation de la fiabilité
— Classes davantage réutilisables

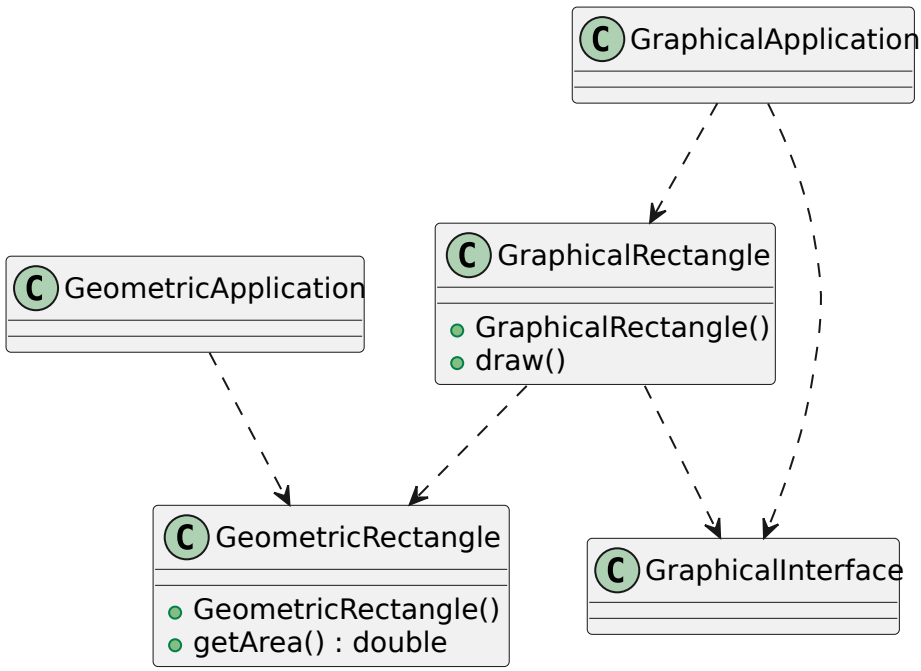
Afin d'illustrer ce principe, nous allons décrire un exemple plus concret. On va considérer une classe `Rectangle` qui a deux méthodes : une méthode `draw()` permettant de dessiner le rectangle et une méthode `area()` calculant l'aire de celui-ci. Deux classes différentes utilisent la classe `Rectangle` : `GeometricApplication` et `GraphicalApplication`. La classe `GeometricApplication` utilise `Rectangle` pour faire des calculs mathématiques sur des formes géométriques (pour faire simple des calculs d'aires), mais ne dessine jamais de rectangle à l'écran. La classe `GraphicalApplication` est de nature graphique et peut également faire de la géométrie informatique, mais elle dessine des rectangles à l'écran en utilisant une classe `GraphicalInterface`. Le diagramme de classe ci-dessous illustre cette architecture.



Cette conception est contraire à SRP. En effet, la classe `Rectangle` a deux responsabilités. La première consiste à fournir un modèle mathématique de la géométrie d'un rectangle. La seconde est de dessiner le rectangle pour une interface graphique en utilisant une interface graphique nommée `GraphicalInterface`.

La violation de SRP entraîne plusieurs problèmes. Tout d'abord, nous devons inclure l'interface graphique `GraphicalInterface` dans l'application de géométrie informatique `GeometricApplication` car `Rectangle` a besoin de cette classe. Cela signifie que `GraphicalInterface` devra être construit et déployé avec l'application de géométrie `GeometricApplication`. Deuxièmement, si une modification de l'application graphique `GraphicalApplication` entraîne une modification de la classe `Rectangle` pour une raison quelconque, cette modification peut nous obliger à reconstruire, retester et redéployer l'application de géométrie informatique `GeometricApplication`. Si nous oublions de le faire, cette application peut dysfonctionner de manière imprévisible.

Une meilleure conception consiste à séparer les deux responsabilités dans deux classes complètement différentes, comme le montre la figure ci-dessous. Cette conception déplace les aspects de calcul géométrique de `Rectangle` dans la classe `GeometricRectangle` et de garder les fonctionnalités de rendu graphique dans `GraphicalRectangle`. Désormais, les modifications apportées à la manière dont les rectangles sont rendus ne peuvent pas affecter `GeometricApplication`.



1.3 Principe d'ouvert/fermé

Le "O" de SOLID signifie Open-Closed Principle, également noté OCP. Robert Cecil Martin dans son livre "Agile Software Development, Principles, Patterns, and Practices" définit ce principe de la manière suivante :

> The Open/Closed Principle (OCP) : Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.

En français, cela donne :

> Les entités logicielles (classes, modules, fonctions, etc.) doivent être ouvertes à l'extension, mais fermées à la modification.

Si on souhaite respecter OCP pour les classes, il doit donc être possible de rajouter une nouvelle fonctionnalité :

- en ajoutant des classes (ouvert pour l'extension)
- sans modifier le code existant d'une classe (fermé à la modification)

Cela signifie que par exemple pour une classe donnée, on doit pouvoir l'étendre, c'est-à-dire l'utiliser pour créer une nouvelle classe, mais pas la modifier pour y intégrer le comportement de la nouvelle classe. Les Avantages sont nombreux :

- Le code existant n'est pas modifié et on a donc pas besoin de le rester ou de le reconstruire ce qui nous permet de gagner en fiabilité.
- Les classes ont plus de chance d'être réutilisables.

— Simplification de l'ajout de nouvelles fonctionnalités.

Afin d'illustrer les problèmes du non-respect d'OCP, on va considérer un exemple pour lequel on a déjà deux classes représentant des formes géométriques : `Rectangle` et `Circle` ayant le code suivant :

```
public class Rectangle {
  public Point point1, point2;

  public Rectangle(Point point1, Point point2) {
    this.point1 = point1;
    this.point2 = point2;
  }
}
```

```
public class Circle {
  public Point center;
  public int radius;

  public Circle(Point center, int radius) {
    this.center = center;
    this.radius = radius;
  }
}
```

Afin de pouvoir dessiner ces deux types de formes géométriques, on a aussi le code d'une classe `GraphicTools` qui permet de dessiner une liste d'objets qui représente des formes géométriques.

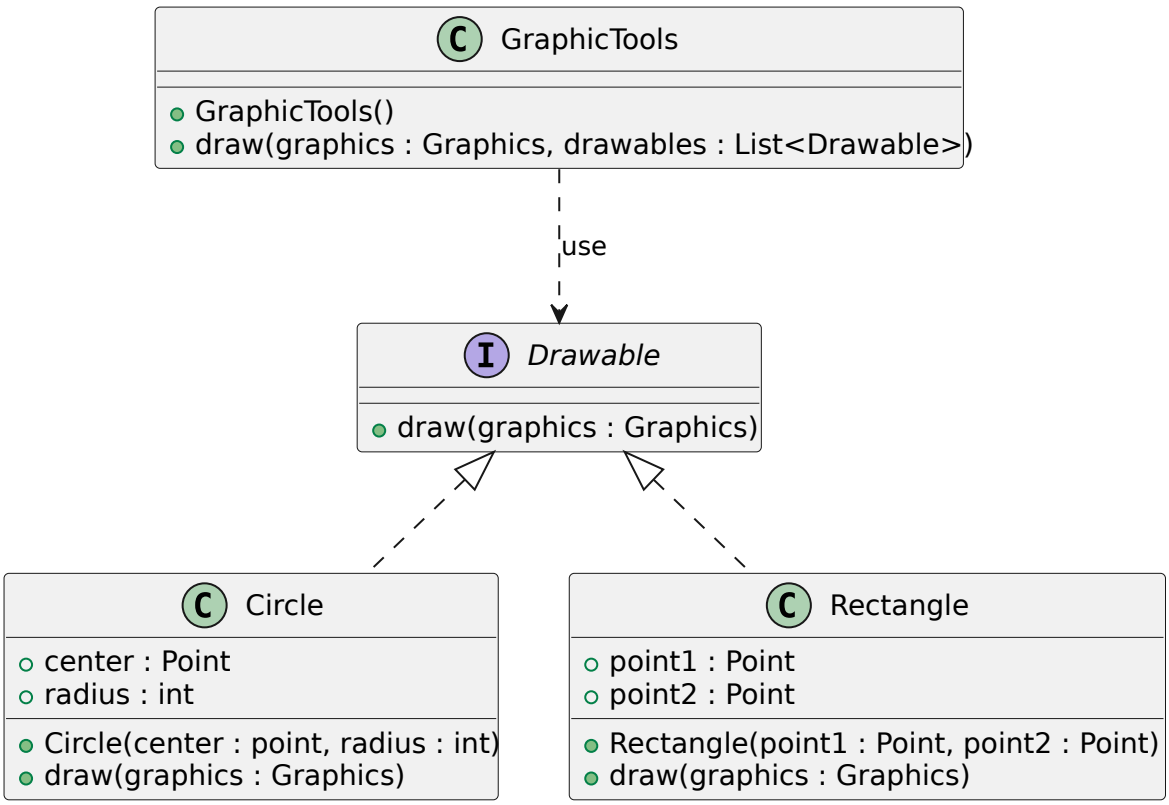
```
public class GraphicTools {
  static void drawShapes(Graphics graphics, List<Object> shapes) {
    for (Object shape : shapes) {
      if (shape instanceof Rectangle) {
        Rectangle rectangle = (Rectangle) shape;
        int x = Math.min(rectangle.point1.x, rectangle.point2.x);
        int y = Math.min(rectangle.point1.y, rectangle.point2.y);
        int width = Math.abs(rectangle.point1.x - rectangle.point2.x);
        int height = Math.abs(rectangle.point1.y - rectangle.point2.y);
        graphics.drawRect(x, y, width, height);
      } else if (shape instanceof Circle) {
        Circle circle = (Circle) shape;
        int x = circle.center.x - circle.radius;
        int y = circle.center.y - circle.radius;
        int width = circle.radius * 2;
        int height = circle.radius * 2;
        graphics.drawOval(x, y, width, height);
      }
    }
  }
}
```

Parce qu'elle n'est fermée à de nouveaux types de formes, la méthode `drawShapes` n'est pas conforme à OCP. Si on voulait étendre cette fonction pour pouvoir dessiner une liste de formes comprenant des triangles, on devrait modifier la fonction. En fait, on devrait modifier la fonction pour tout nouveau type de forme qu'on aurait besoin de dessiner.

On va donc modifier le code afin de respecter OCP. La première étape va consister à simplifier le code de la méthode `drawShapes` en extrayant dans des méthodes distinctes les fonctionnalités de dessin des rectangles et des cercles.

```
public class GraphicTools {
  static void draw(Graphics graphics, List<Object> objects) {
    for (Object object : objects) {
      if (object instanceof Rectangle) {
        Rectangle rectangle = (Rectangle)object;
        drawRectangle(graphics, rectangle);
      } else if (object instanceof Circle) {
        Circle circle = (Circle)object;
        drawCircle(graphics, circle);
      }
    }
  }

  static void drawRectangle(Graphics graphics, Rectangle rectangle) {
    int x = Math.min(rectangle.point1.x, rectangle.point2.x);
    int y = Math.min(rectangle.point1.y, rectangle.point2.y);
    int width = Math.abs(rectangle.point1.x - rectangle.point2.x);
    int height = Math.abs(rectangle.point1.y - rectangle.point2.y);
    graphics.drawRect(x, y, width, height);
  }

  static void drawCircle(Graphics graphics, Circle circle) {
    int x = circle.center.x - circle.radius;
    int y = circle.center.y - circle.radius;
    int width = circle.radius * 2;
    int height = circle.radius * 2;
    graphics.drawOval(x, y, width, height);
  }
}
```

La deuxième étape de la modification du code consiste à créer une interface `Drawable` qui va contenir une unique méthode `draw` et qui sera implémentée par `Circle` et `Rectangle`. Cela nous donne le schéma ci-dessous.



Le code du diagramme est le suivant.

```java
public interface Drawable {
  void draw(Graphics graphics);
}
```

```java
public class GraphicTools {
  static void draw(Graphics graphics, List<Drawable> drawables) {
    for (Drawable drawable : drawables)
      drawable.draw(graphics);
  }
}
```

```java
public class Circle implements Drawable {
  public Point center;
  public int radius;

  public Circle(Point center, int radius) {
    this.center = center;
    this.radius = radius;
  }

  public void draw(Graphics graphics) {
    int x = center.x - radius;
    int y = center.y - radius;
    int width = radius * 2;
    int height = radius * 2;
    graphics.drawOval(x, y, width, height);
  }
```

```
}
```

```
public class Rectangle implements Drawable {
  public Point point1, point2;

  public Rectangle(Point point1, Point point2) {
    this.point1 = point1;
    this.point2 = point2;
  }

  public void draw(Graphics graphics) {
    int x = Math.min(point1.x, point2.x);
    int y = Math.min(point1.y, point2.y);
    int width = Math.abs(point1.x - point2.x);
    int height = Math.abs(point1.y - point2.y);
    graphics.drawRect(x, y, width, height);
  }
}
```

Cette solution n'est pas totalement satisfaisante, car les classes `Rectangle` et `Circle` ont deux responsabilités distinctes : le stockage des données géométriques des formes (comment on sauvegarde la position des formes géométriques) et la fonctionnalité de rendu graphique. La classe `Rectangle` a donc deux raisons de changer : un changement si on souhaite sauvegarder le rectangle avec un coin, sa largeur et sa hauteur plutôt que les deux coins opposés et un autre changement dû à un choix différent de bibliothèque graphique qui changera la méthode `draw`. Nous verrons par la suite comment obtenir une meilleure solution avec l'application du patron de conception visiteur.

1.4 Principe de substitution de Liskov

Un des concepts les plus importants afin de pouvoir respecter OCP est l'extension de classe aussi appelé héritage de code. C'est grâce à l'héritage qu'il est possible de créer des classes dérivées qui implémentent de nouvelles méthodes dans les classes de base. C'est un des outils qu'on peut utiliser pour ajouter des services à une classe sans modifier le code de la classe elle-même. On peut donc se poser la question de savoir quelles sont les règles de conception qui régissent cette utilisation de l'héritage. Comment peut-on reconnaître une bonne hiérarchie d'héritage d'une mauvaise ? Quels sont les pièges qui nous amèneront à créer des hiérarchies non conformes à OCP ? Ce sont les questions qui sont répondues par le principe de substitution de Liskov qui correspond au "L" de SOLID pour *Liskov Substitution Principle*, également noté LSP. Robert Cecil Martin dans son livre "Agile Software Development, Principles, Patterns, and Practices" définit ce principe de la manière suivante :

> The Liskov Substitution Principle : Subtypes must be substitutable for their base types.

En français, cela donne :

> Le principe de substitution de Liskov : les sous-types doivent être substituables à leurs types de base.

Barbara Liskov a définit ce principe en 1988 de la manière suivante :

> `S` est un sous-type (extension) correct de `T` Si pour chaque objet `o1` de type `S` il existe un objet `o2` de type `T` tel que pour tous les programmes `P` définis en termes de `T`, le comportement de `P` est inchangé lorsque `o1` est remplacé par `o2`.

Dit autrement, si une classe `D` étend une classe `B` (ou implémente une interface `B`) alors un programme `P` écrit pour manipuler des instances de type `B` doit avoir le même comportement s'il manipule des instances de la classe `D`. L'importance de ce principe devient évidente lorsque l'on considère les conséquences de sa violation. Supposons que nous ayons une fonction `f` qui prend comme argument un objet de type `B`. Supposons également que, lorsqu'on passe à `f` un objet de type `D`, on obtient un comportement incorrect de `f`. Dans ce cas, la classe `D` viole LSP. C'est ce genre de problème que l'on souhaite éviter en respectant LSP.

Afin d'illustrer les problèmes du non-respect de LSP, on va considérer un exemple pour lequel on a déjà deux classes représentant des formes géométriques. On considère tout d'abord une classe `Rectangle` ayant le code suivant :

```
1  public class Rectangle {
2    private double width;
3    private double height;
4
5    public void setWidth(double width) {
6      this.width = width;
7    }
8
9    public void setHeight(double height) {
10     this.height = height;
11   }
12
13   public double getWidth() {
14     return width;
15   }
16
17   public double getHeight() {
18     return height;
19   }
20
21   public double getArea() {
22     return width*height;
23   }
24 }
```

Un carré étant un rectangle, on souhaite définir une classe `Square` qui est une extension de `Rectangle` de la façon suivante :

```
1  public class Square extends Rectangle {
2
3    public void setWidth(double width) {
4      super.setWidth(width);
5      super.setHeight(width);
```

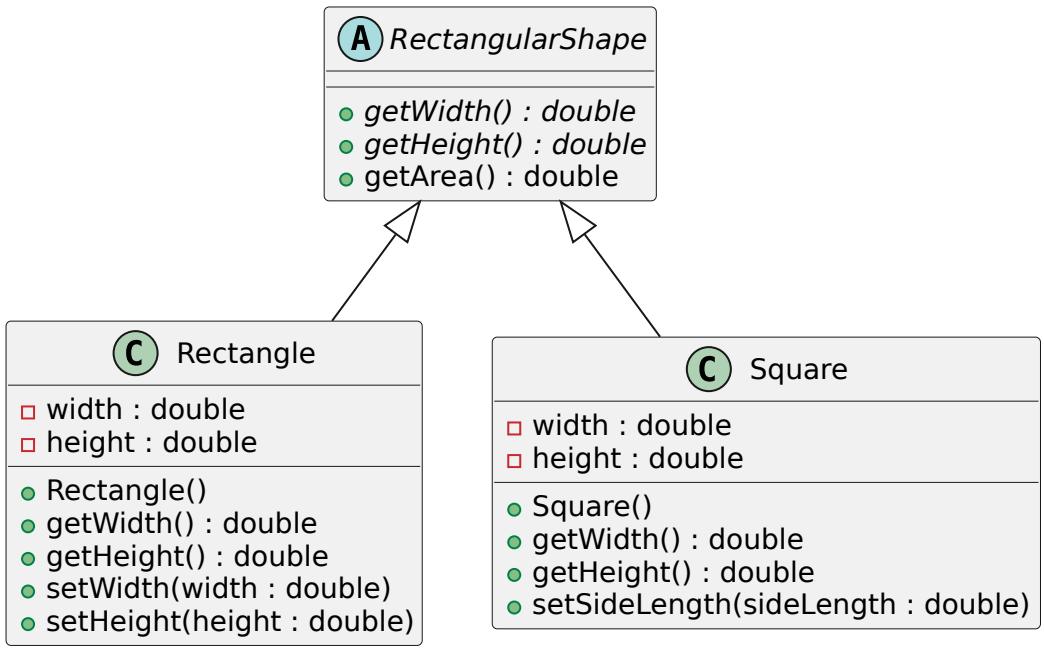
```
6     }
7
8     public void setHeight(double height) {
9       super.setWidth(height);
10      super.setHeight(height);
11    }
12  }
```

Maintenant, supposons que nous testons la méthode `area` de rectangle avec le code suivant :

```
1     void testRectangleArea(Rectangle r){
2       r.setWidth(3);
3       r.setHeight(2);
4       assertThat(r.area()).isEqualTo(3*2);
5     }
```

Si on appelle la méthode avec un objet de type `Rectangle`, le code s'exécute sans problème, car l'assertion est vraie. Si on appelle la méthode avec un objet de type `Square`, l'assertion est fausse, car la largeur est fixée à 2 par l'appel `r.setHeight(2)` et l'aire est donc de 4. En fait la bonne question à se poser n'est pas de savoir si un carré est-il un rectangle, mais plutôt de déterminer si un carré a le même comportement qu'un rectangle. Puisque la réponse à cette deuxième question est négative, ce n'est pas une bonne idée de définir la classe des carrés comme un sous-type de la classe des rectangles.

Pour corriger le problème, une solution consiste à définir une classe abstraite `RectangularShape` qui contiendra les parties communes de `Rectangle` et `Square`, c'est-à-dire le calcul de l'aire et la définition des signatures des accesseurs (*getters*). Cette classe sera étendue par `Rectangle` et `Square`. Cela nous donne le diagramme suivant :



Cela nous donne le code suivant :

```
1   public abstract class RectangularShape {
```

```
  public abstract double getWidth();
  public abstract double getHeight();

  public double getArea() {
    return getWidth() * getHeight();
  }
}

public class Rectangle extends RectangularShape {
  private double width;
  private double height;

  public void setWidth(double width) {
    this.width = width;
  }

  public void setHeight(double height) {
    this.h = height;
  }

  public double getWidth() {
    return width;
  }

  public double getHeight() {
    return height;
  }
}

class Square extends RectangularShape {
  private double sideLength;

  public void setSideLength(double sideLength) {
    this.sideLength = sideLength;
  }

  public double getWidth() {
    return sideLength;
  }

  public double getHeight() {
    return sideLength;
  }
}
```

1.5 Principe de ségrégation des interfaces

Le “I” de SOLID signifie *Interface Segregation Principle*, également noté ISP. Robert Cecil Martin dans son livre “Agile Software Development, Principles, Patterns, and Practices” définit ce principe de la manière suivante :

> The Interface Segregation Principle (ISP) : Clients should not be forced to depend upon interfaces that they do not use.

En français, cela donne :

> Les clients ne doivent pas être obligés de dépendre d'interfaces qu'ils n'utilisent pas.

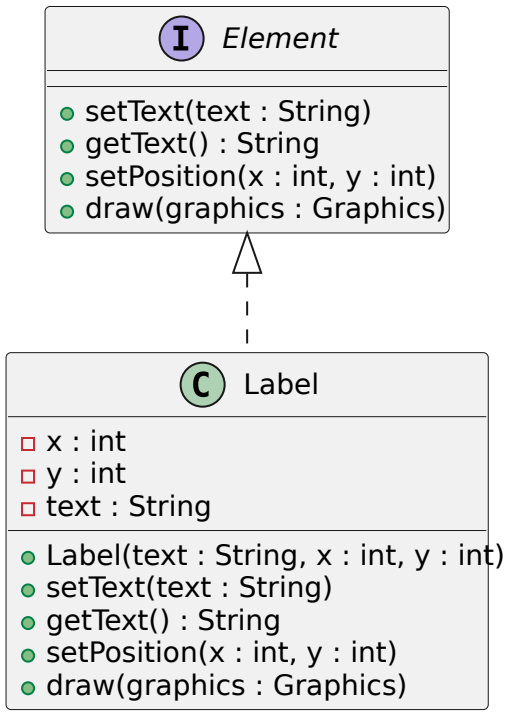
Il faut donc éviter qu'un client ne voit une interface qu'il n'utilise pas. Dans la plupart des cas, appliquer ce principe revient à éviter les interfaces qui contiennent beaucoup de méthodes :

- Découper les interfaces en responsabilités distinctes (SRP)
- Quand une interface grossit, se poser la question du rôle de l'interface
- Éviter de devoir implémenter des services qui n'ont pas à être proposés par la classe qui implémente l'interface
- Limiter les modifications lors de la modification de l'interface

Les avantages sont nombreux :

- Le code existant est moins modifié et on a donc une augmentation de la fiabilité
- Les classes ont plus de chance d'être réutilisables
- Simplification de l'ajout de nouvelles fonctionnalités

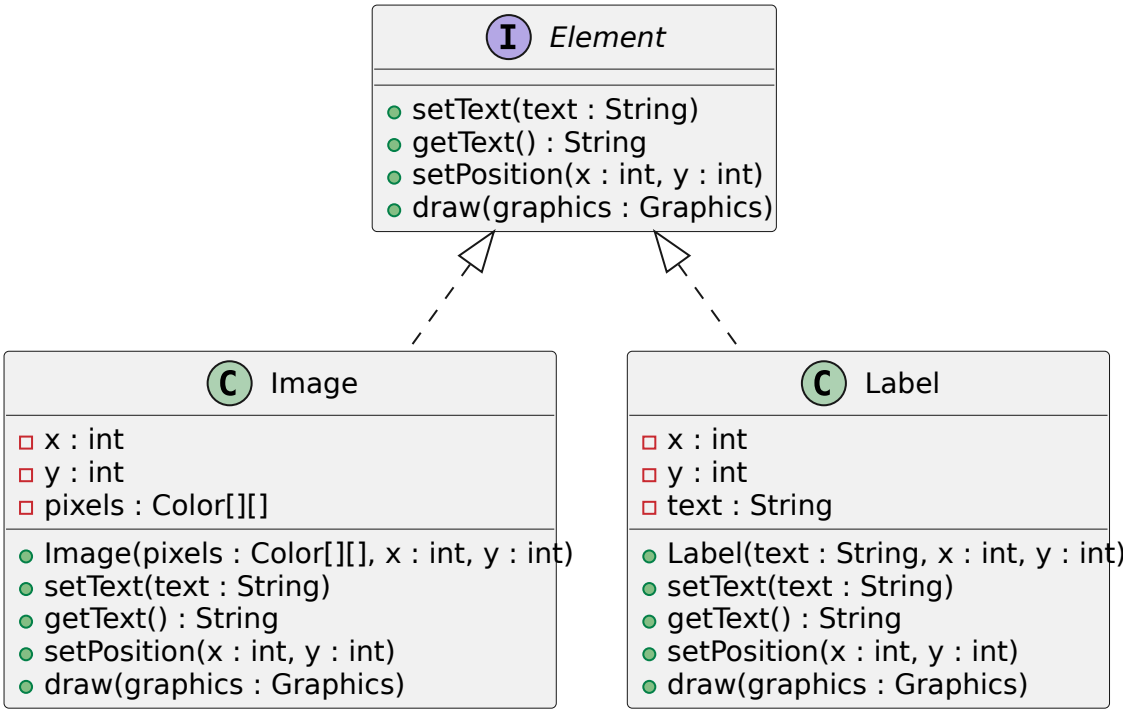
Afin d'illustrer les raisons de ce principe, on va considérer un exemple dans lequel on considère une interface `Element` pour les éléments d'une interface graphique. Cette interface sera implémentée par une classe `Label` qui permet de représenter des étiquettes de texte. On considère donc l'architecture décrite par le diagramme suivant :



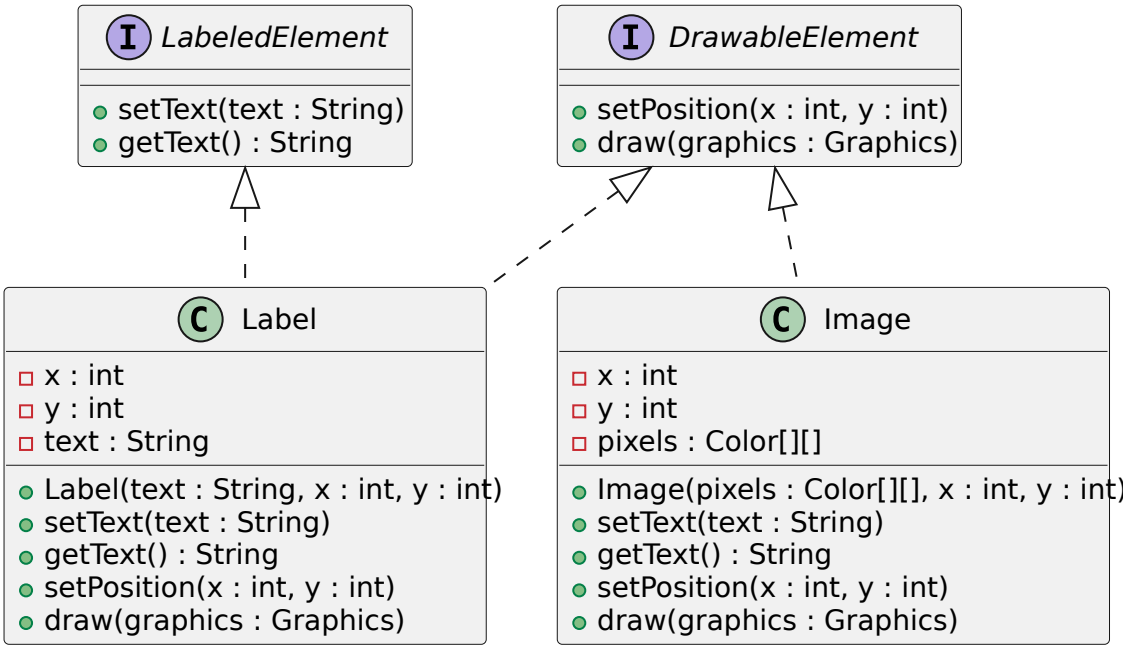
Cela nous donne le code suivant :

```java
public interface Element {
  public void setText(String text);
  public String getText(String text);
  public void setPosition(int x, int y);
  public void draw(Graphics graphics);
}

public class Label implements Element {
  private int x,y;
  private String text;

  public Label(String text, int x, int y) {
    this.text = text; this.x = x; this.y = y;
  }

  public void setText(String text)   {
    this.text = text;
  }

  public String getText(String text) {
    return text;
  }

  public void setPosition(int x, int y) {
    this.x = x; this.y = y;
  }

  public void draw(Graphics graphics) {
    graphics.drawString(text, x, y);
  }
}
```

Supposons maintenant que nous souhaitions ajouter une classe image qui est aussi un élément graphique et qui implémenterait donc `Element`. Cela nous donne le diagramme suivant :



Nous avons un problème, car une image n'a pas de texte. On ne sait donc pas que faire dans les méthodes `setText` et `getText`. La solution est de découper l'interface `Element` en deux en séparant les fonctionnalités liées au texte des fonctionnalités liées au rendu graphique de l'élément. On obtient donc le diagramme suivant :



1.6 Principe d'inversion des dépendances

Le "D" de SOLID signifie *Dependency Inversion Principle*, également noté DIP. Robert Cecil Martin dans son livre "Agile Software Development, Principles, Patterns, and Practices" définit ce principe de la manière suivante :

> High-level modules should not depend on low-level modules. Both should depend on abstractions (e.g. interfaces). Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.

En français, cela donne :

> Les modules de haut niveau ne doivent pas dépendre de ceux de bas niveau. Les deux doivent dépendre d'abstractions (par exemple les interfaces). Les abstractions, elles, ne doivent pas dépendre des détails. Les détails (implémentations concrètes) doivent dépendre des abstractions.

Ainsi, si une classe `A` utilise une classe `B`, il doit être possible de remplacer `B` par une autre classe `C`. `B` et `C` sont alors des implémentations concrètes d'une classe abstraite (ou d'une interface) qui sera utilisée par `A`.

Les modules d'un programme doivent donc être indépendants et doivent dépendre d'abstractions. Pour respecter DIP, il faut donc :

— découpler le plus possible les différents modules de votre programme ;
— les lier quand c'est nécessaire en utilisant des interfaces ;
— spécifier correctement le comportement de chaque module.

Les avantages de l'application de DIP :

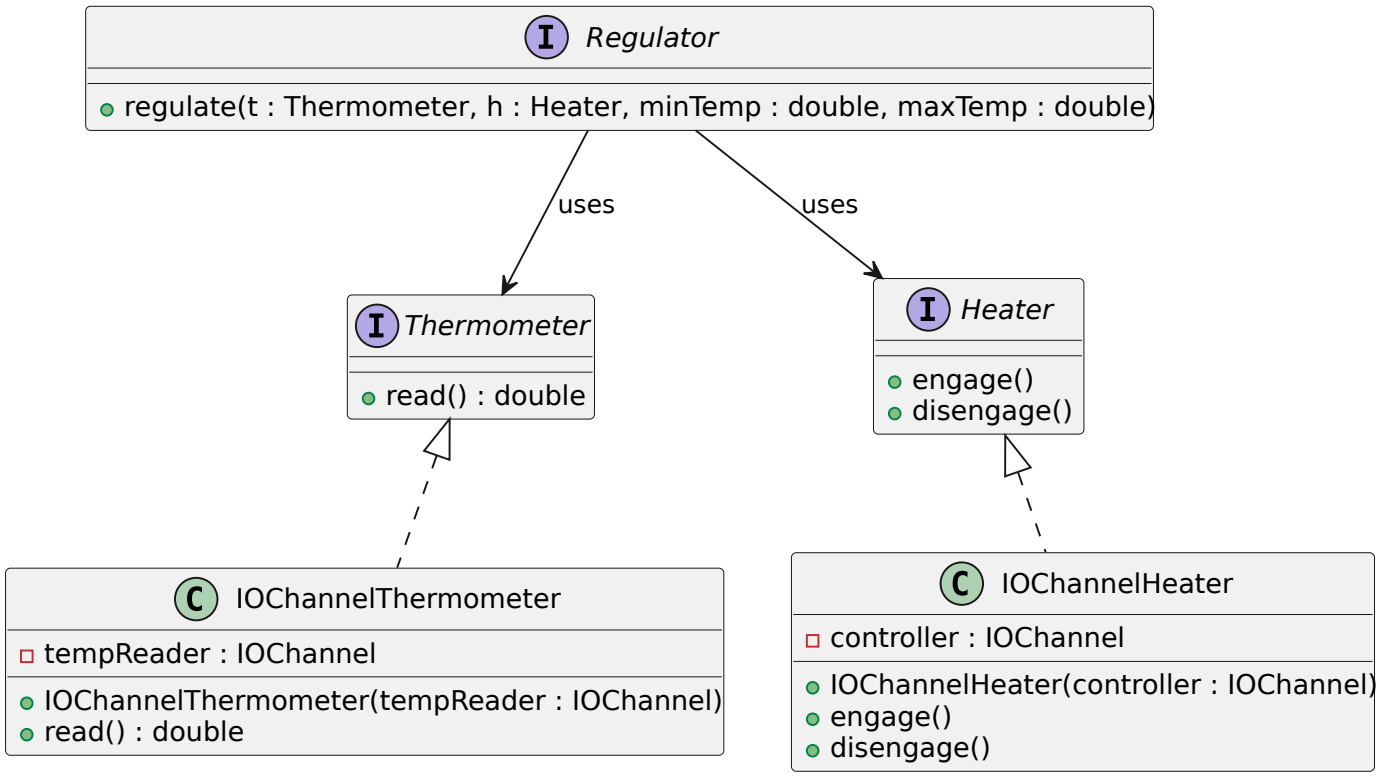
— Permet de remplacer un module par un autre module plus facilement.
— Les modules sont plus facilement réutilisables.
— Simplification de l'ajout de nouvelles fonctionnalités.
— L'intégration est rendue plus facile.

Afin d'illustrer l'application de DIP, nous allons considérer un exemple assez simple. Considérons le logiciel qui pourrait contrôler le régulateur d'un four. Le logiciel peut lire la température actuelle à partir d'un canal d'entrée/sortie et demander au four de s'allumer ou de s'éteindre en écrivant des commandes à un autre canal d'entrée/sortie. La structure du programme pourrait ressembler au code suivant :

```
1  public class Thermostat {
2    enum IOChannel{
3      THERMOMETER, FURNACE
4    }
5    enum Action{
6      ENGAGE, DISENGAGE
7    }
8
9    int read(IOChannel channel){
10     // TODO : add code for reading on a channel
11     return 0;
12   }
13
```

```
14    void write(IOChannel channel, Action action){
15      // TODO : add code for writing on a channel
16    }
17
18    void Regulate(double minTemp, double maxTemp)
19      throws InterruptedException{
20      for(;;) {
21        while (read(IOChannel.THERMOMETER) > minTemp)
22          wait(1);
23        write(IOChannel.FURNACE, Action.ENGAGE);
24        while (read(IOChannel.THERMOMETER) < maxTemp)
25          wait(1);
26        write(IOChannel.FURNACE, Action.DISENGAGE);
27      }
28    }
29  }
```

On comprend assez facilement l'intuition derrière l'algorithme. Néanmoins, un certain nombre de détails techniques liés à des aspects bas niveau (lecture et écriture dans des canaux d'entrées/ sorties) rend le code un peu moins lisible. Ce code ne pourra jamais être réutilisé avec un matériel de contrôle différent qui n'utiliserait pas le même protocole de communication (utilisant des canaux). Ce n'est peut-être pas une grande perte, car le code est très petit. Mais même dans ce cas, il est dommage que l'algorithme ne soit pas réutilisable. Il est préférable inverser les dépendances et de concevoir une architecture comme celle décrite par le diagramme ci-dessous.



2 Patrons de conception

2.1 Introduction

Les *design patterns* (patrons de conception) sont des recettes de conception orientées objet. Ce sont des solutions classiques à des problèmes récurrents de la conception de logiciels. Ils consistent en des plans ou des schémas que l'on peut personnaliser afin de résoudre un problème récurrent dans notre code. Ils furent introduits par quatre développeurs (connus sous le nom de *gang of four*) : Gamma, Helm, Johnson et Vlissides en 1995. Ces développeurs sont partis de l'idée qu'il existe des recettes de conception permettant de répondre de manière similaire à des problèmes semblables. En 1995 donc, ces quatre auteurs publient leur livre "Design Patterns : Elements of Reusable Object-Oriented Software" qui détaille vingt-trois solutions répondant à des problèmes récurrents en développement logiciel. Ces recettes, patrons ou *design patterns* peuvent être classés en trois catégories :

— création : instanciation et configuration des classes et des objets ; — structure : organisation des classes d'un programme dans une structure plus importante ; — comportement : organisation des objets en vue de leur collaboration.

Les patrons de conception sont décrits de manière générique afin de pouvoir être adaptés aux différents problèmes à résoudre : ils ne peuvent donc pas s'appliquer directement à toutes les situations sans adaptation. Vous ne pouvez donc pas vous contenter de trouver un patron et de le recopier dans votre programme. Un patron, ce n'est pas un bout de code spécifique, mais plutôt un concept général pour résoudre un problème précis. Il faut donc toujours réfléchir à leur utilisation dans un contexte donné. Néanmoins, ils permettent de ne pas réinventer la roue, car ils couvrent la plupart des situations aux problématiques auxquelles un développeur est confrontées.

2.2 Patrons de création

Les patrons de création fournissent des mécanismes de création d'objets qui permettent d'augmenter la flexibilité et la réutilisation du code. Les principaux patrons de conceptions sont les suivants :

- *Factory Method* (fabrique) : définit une interface pour la création d'objets dans une classe mère, mais délègue aux sous-classes le choix des types d'objets à créer.
- *Abstract Factory* (Fabrique abstraite) : permet de créer des familles d'objets apparentés sans préciser leur classe concrète.
- *Builder* (monteur) : permet de construire des objets complexes étape par étape. Ce patron permet de construire différentes variations ou représentations d'un objet en utilisant le même code de construction.
- *Prototype* (prototype) : permet de créer de nouveaux objets à partir d'objets existants sans rendre le code dépendant de leur classe.
- *Singleton* (Singleton) : permet de garantir que l'instance d'une classe n'existe qu'en un seul exemplaire, tout en fournissant un point d'accès global à cette instance.

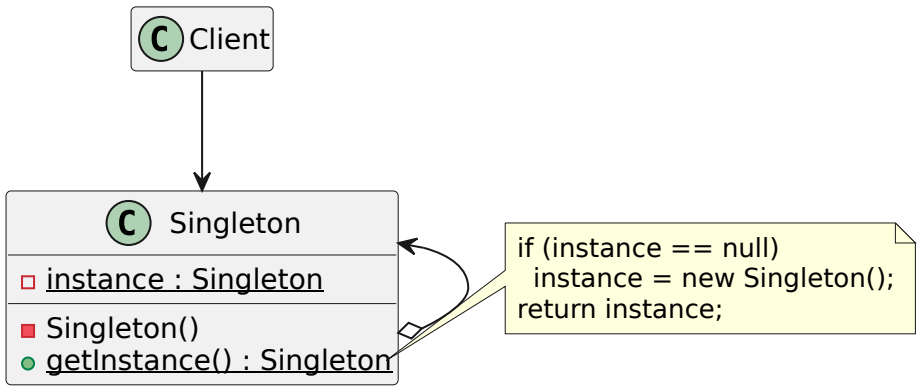
Par la suite, nous allons détailler quelques patrons de création.

2.2.1 *Singleton* (singleton)

Singleton est un patron de conception de création qui garantit que l'instance d'une classe n'existe qu'en un seul exemplaire, tout en fournissant un point d'accès global à cette instance. Le singleton règle deux problèmes à la fois :

— Il garantit l'unicité d'une instance pour une classe. Pour quelle raison voudrait-on maîtriser le nombre d'instances d'une classe ? En général, cette situation se présente lorsque l'on veut contrôler l'accès à une ressource partagée — une base de données ou un fichier par exemple.
— Il fournit un point d'accès global à cette instance, protège cette instance en l'empêchant d'être modifiée.

La solution consiste à rendre le constructeur privé (pour empêcher la création d'une instance sans contrôle), et à créer une méthode statique pour contrôler l'instanciation.



Le code associé est le suivant :

```
1  public class Singleton
2  {
3    private static Singleton instance = null;
4    /**
5     * Private constructor to prevent construction outside the class.
6     */
7    private Singleton()
8    {
9      // Code of the constructor
10   }
11   /**
12    * Keyword synchronized is used to allow only one thread to execute
13    * the method at any given time (preventing concurrent access).
14    *
15    * @return the instance of the unique instance of the class.
16    */
17   public synchronized static Singleton getInstance()
18   {
19     if (instance == null)
20     {
21       instance = new Singleton();
22     }
23     return instance;
24   }
25 }
```

2.2.2 *Factory Method* (fabrique)

Fabrique est un patron de conception de création qui définit une interface pour créer des objets dans une classe mère ou interface, mais délègue le choix des types d'objets à créer aux sous-classes. La fabrique permet donc à une classe de déléguer l'instanciation à des sous-classes.

Supposons que nous disposions de l'interface suivante :

```
1 public interface Button {
2   public void draw();
3 }
```

Il existe une première implémentation de cette interface :

```
1 public class SimpleButton implements Button  {
2   public void draw() {
3     System.out.println("Simple button.");
4   }
5 }
```

La classe `SimpleButton` est instanciée dans de nombreuses autres classes et donc son constructeur apparaît à de nombreux endroits. Supposons que nous fassions une autre implémentation de `Button` :

```
1 public class ModernButton implements Button  {
2   public void draw() {
3     System.out.println("Modern button.");
4   }
5 }
```

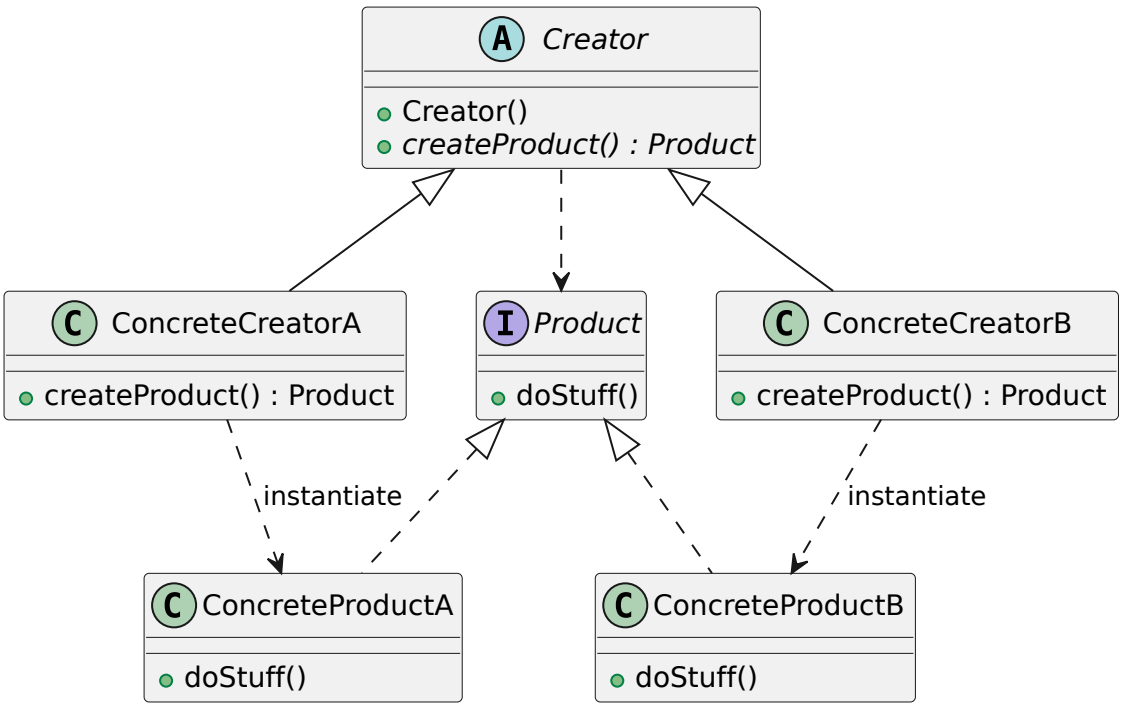
Afin d'utiliser ce nouveau bouton, nous devons modifier toutes les instanciations (appels au constructeur) présentes dans notre code. Il y a donc une violation d'OCP. Afin d'éviter cela, on va donc appliquer le patron de conception fabrique. Une fabrique consiste à isoler la création des objets de leurs utilisations :

```
1 public class ButtonFactory {
2   public Button createButton() {
3     return new SimpleButton();
4   }
5 }
```

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

```
1 public class ButtonFactory {
2   public Button createButton() {
3     return new ModernButton();
4   }
5 }
```

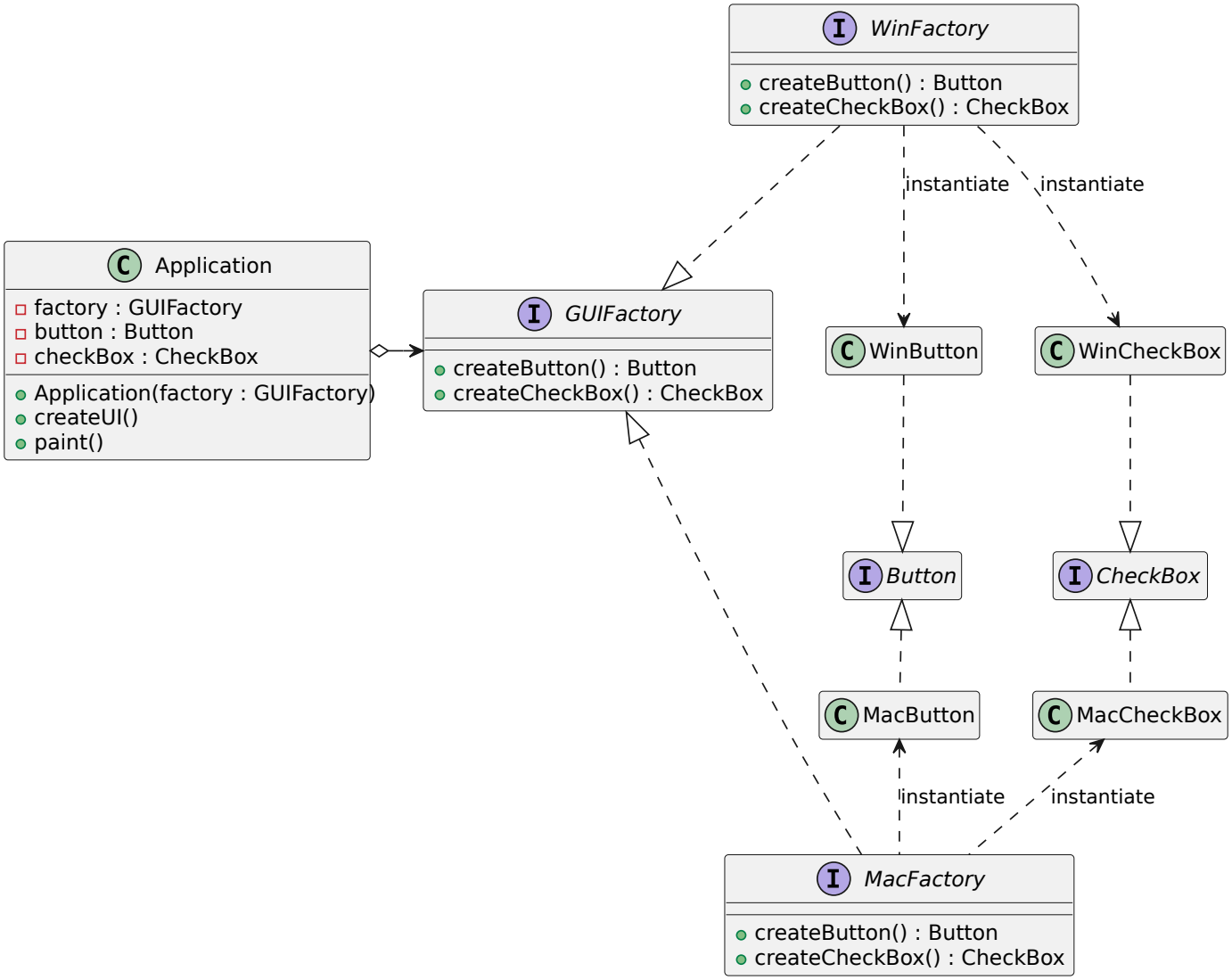
De manière générale le patron de conception a la forme suivante dans laquelle a une classe abstraite de création d'objets qui peut être étendue de plusieurs manières différentes.



2.2.3 *Abstract factory* (fabrique abstraite)

Fabrique abstraite est un patron de conception qui permet de créer des familles d'objets apparentés sans préciser leur classe concrète. On va définir une interface contenant plusieurs méthodes de création d'objets et on implémentera cette interface avec des classes qui garantiront une cohérence entre les différents objets créés.

Pour illustrer ce principe, nous allons considérer la création des éléments d'une interface utilisateur pouvant fonctionner sur plusieurs systèmes : mac ou windows. Pour simplifier notre exemple, nous allons considérer qu'il n'y a que deux types d'éléments dans l'interface utilisateur : des boutons (interface `Button`) et des coches (interface `CheckMark`). Dans notre cas, il faut donc pouvoir créer ces deux types d'objets. On définit donc une interface `GUIFactory` contenant deux méthodes de création d'objet (une pour créer des `Button` et l'autre pour créer des `CheckMark`). Cette interface aura deux implémentations `WinFactory` qui crée des éléments pour windows et `MacFactory` qui crée des éléments pour mac. L'application graphique peut donc utiliser une instance d'une de ces deux classes (suivant le système visé) afin de créer les éléments de l'interface utilisateur. Vous trouverez ci-dessous le diagramme de classes correspondant à cette organisation du code.



L'utilisation du patron fabrique abstraite a ici deux avantages :

— Cela permet de cacher les classes concrètes à l'application et donc d'éviter des dépendances à des classes concrètes (respect de DIP).

— Cela permet de garantir la cohérence entre les objets construits par une même fabrique. Ces objets partagent une propriété commune (dans notre exemple un système d'exploitation). Cela permet d'éviter certaines incohérences comme dans notre exemple, des éléments pour des systèmes différents.

Comme on vient de le voir, la Fabrique Abstraite est un patron de conception de création qui permet de créer des familles d'objets apparentés sans préciser leur classe concrète. Une famille d'objets est un ensemble de classes ayant une certaine cohérence. Par exemple, si on considère des triplets : (Transport, Moteur, Contrôles), plusieurs variantes peuvent exister :

— (Voiture, MoteurCombustion, Volant)

— (Avion, MoteurRéaction, Manche)

Si votre programme fonctionne sans famille de produits, vous n'avez pas besoin d'une fabrique abstraite et une

fabrique simple (patron vu précédemment) peut suffire.

2.3 Patrons structurels

Les patrons structurels servent à guider l'assemblage des objets et des classes afin d'obtenir des structures plus grandes tout en gardant celles-ci flexibles et efficaces. Les principaux patrons de conception structurels sont les suivants :

- *Adapter* (adaptateur) : permet de faire collaborer des objets ayant des interfaces normalement incompatibles.
- *Bridge* (pont) : permet de séparer une grosse classe ou un ensemble de classes connexes en deux hiérarchies — abstraction et implémentation — qui peuvent évoluer indépendamment l'une de l'autre.
- *Composite* (composite) : permet d'agencer les objets dans des arborescences afin de pouvoir traiter celles-ci comme des objets individuels.
- *Decorator* (décorateur) : permet d'affecter dynamiquement de nouveaux comportements à des objets en les plaçant dans des emballeurs qui implémentent ces comportements.
- *Facade* (façade) : procure une interface qui offre un accès simplifié à une librairie, un framework ou à n'importe quel ensemble complexe de classes.
- *Flyweight* (poids mouche) :Permet de stocker plus d'objets dans la RAM en partageant les états similaires entre de multiples objets, plutôt que de stocker les données dans chaque objet.
- *Proxy* (procuration) : permet de fournir un substitut d'un objet. Une procuration donne le contrôle sur l'objet original, vous permettant d'effectuer des manipulations avant ou après que la demande ne lui parvienne.

Nous allons détailler le principe de certains patrons structurels.

2.3.1 Adapter (adaptateur)

L'Adaptateur est un patron de conception structurel qui permet de faire collaborer des objets ayant des interfaces normalement incompatibles. Supposons que la classe suivante existe :

```
1 public class Drawer {
2   public void draw(Pencil pencil) {
3     pencil.drawLine(0,0,10,10);
4     pencil.drawCircle(5,5,5);
5     pencil.drawLine(10,0,0,10);
6   }
7 }
```

Nous avons également l'interface suivante qui permet de dessiner des segments et des cercles à partir des coordonnées des points et du rayon du cercle :

```
1 public interface Pencil {
2   public void drawLine(int x1, int y1, int x2, int y2);
3   public void drawCircle(int x, int y, int radius);
4 }
```

Cette classe `Drawer` et cette interface `Pencil` ne peuvent pas être modifiées. Nous avons également la classe

`PointPencil` suivante qui permet de dessiner des segments et des cercles à partir d'objets `Point2D` :

```
public class PointPencil {
  public void drawLineWithPoints(Point2D p1, Point2D p2) {
    /* ... */
  }

  public void drawCircleWithPoint(Point2D center, int radius) {
    /* ... */
  }
}
```

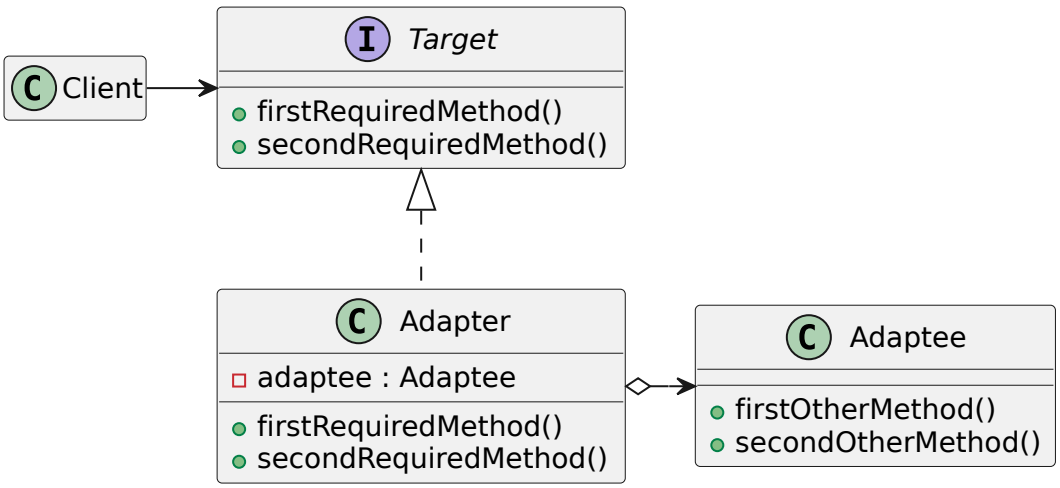
Nous souhaitons utiliser la classe `PointPencil` comme un `Pencil` pour qu'elle puisse être utilisée par la classe `Drawer`. Pour ce faire, nous définissons l'adaptateur suivant :

```
public class PointPencilAdapter implements Pencil {

  private PointPencil pointPencil;

  public CrayonAdapter(PointPencil pointPencil) {
    this.pointPencil = pointPencil;
  }

  public void drawLine(int x1, int y1, int x2, int y2) {
    pointPencil.drawLineWithPoints(new Point2D(x1, y1),
                                   new Point2D(x2, y2));
  }

  public void drawCircle(int x, int y, int radius) {
    pointPencil.drawCircleWithPoint(new Point(x, y), radius);
  }
}
```

Cette classe est utilisable de la façon suivante :

```
Pencil pencil = new PointPencilAdapter(new PointPencil());
Drawer drawer = new Drawer();
drawer.draw(pencil);
```

De manière générale, le patron a le format suivant :



2.3.2 *Decorator* (décorateur)

Décorateur est un patron de conception structurel qui permet d'affecter dynamiquement de nouveaux comportements à des objets en les plaçant dans des emballeurs qui implémentent ces comportements. Supposons que nous avons la classe suivante :

```
1  public class ArrayStack {
2    private int[] stack = new int[10];
3    private int size = 0;
4
5    public void push(int value) {
6      list[size] = value;
7      size++;
8    }
9
10   public int pop() {
11     size--;
12     return list[size];
13   }
14 }
```

Nous souhaitons ajouter des logs pour débugger notre programme :

```
1  public class ArrayStack {
2    private int[] stack = new int[10];
3    private int size = 0;
4
5    public void push(int value) {
6      System.out.println("push("+value+")");
7      list[size] = value;
8      size++;
9    }
10
11   public int pop() {
12      System.out.println("pop()");
13      size--;
14      return list[size];
15   }
```

```
16 }
```

Cette modification a été réalisée en modifiant une classe existante. De plus, une nouvelle modification est nécessaire pour retirer les logs. Nous avons donc un non-respect d'OCP.

Afin de résoudre cette difficulté, nous allons utiliser le patron décorateur. On commence par définir l'interface suivante :

```
1  public interface Stack {
2    public void push(int value);
3    public int pop();
4  }
```

On peut facilement faire en sorte que `ArrayStack` implémente cette interface (en utilisant le mot-clé `implements`) car la classe `ArrayStack` définit déjà les méthodes de l'interface.

```
1  public class ArrayStack implements Stack {
2    /* ... */
3  }
```

Maintenant, il suffit de définir un décorateur :

```
1  public class VerboseStack implements Stack {
2    private Stack stack;
3
4    public VerboseStack(Stack stack) { this.stack = stack; }
5
6    public void push(int value) {
7      System.out.println("push("+value+")");
8      stack.push(value);
9    }
10
11   public int pop() {
12      System.out.println("pop()");
13      return stack.pop();
14   }
15 }
```

Supposons que nous ayons le code suivant :

```
1  Stack stack = new ArrayStack(10);
2  stack.push(2);
3  stack.pop();
```

Il est très facile d'introduire le décorateur :

```
1  Stack stack = new ArrayStack(10);
2  stack = new VerboseStack(stack);
3  stack.push(2);
4  stack.pop();
```

Ce code produit la sortie suivante :

```
push(2);
pop();
```

Nous pouvons aussi définir un nouveau décorateur :

```
public class CounterStack implements Stack {
  private Stack stack;
  private int size;

  public CounterStack(Stack stack) {
    this.stack = stack;
    size = 0;
  }

  public void push(int value) {
    size++;
    stack.push(value);
  }

  public int pop() {
    size--;
    return stack.pop();
  }

  public int getSize() {
    return size;
  }
}
```

Un exemple d'utilisateur du décorateur précédent :

```
Stack stack = new ArrayStack(10);
CounterStack counterStack = new CounterStack(stack);
stack = counterStack;
stack.push(2);
stack.push(3);
stack.pop();
System.out.println(counterStack.getSize());
```

Ce code produit la sortie suivante :

```
1
```

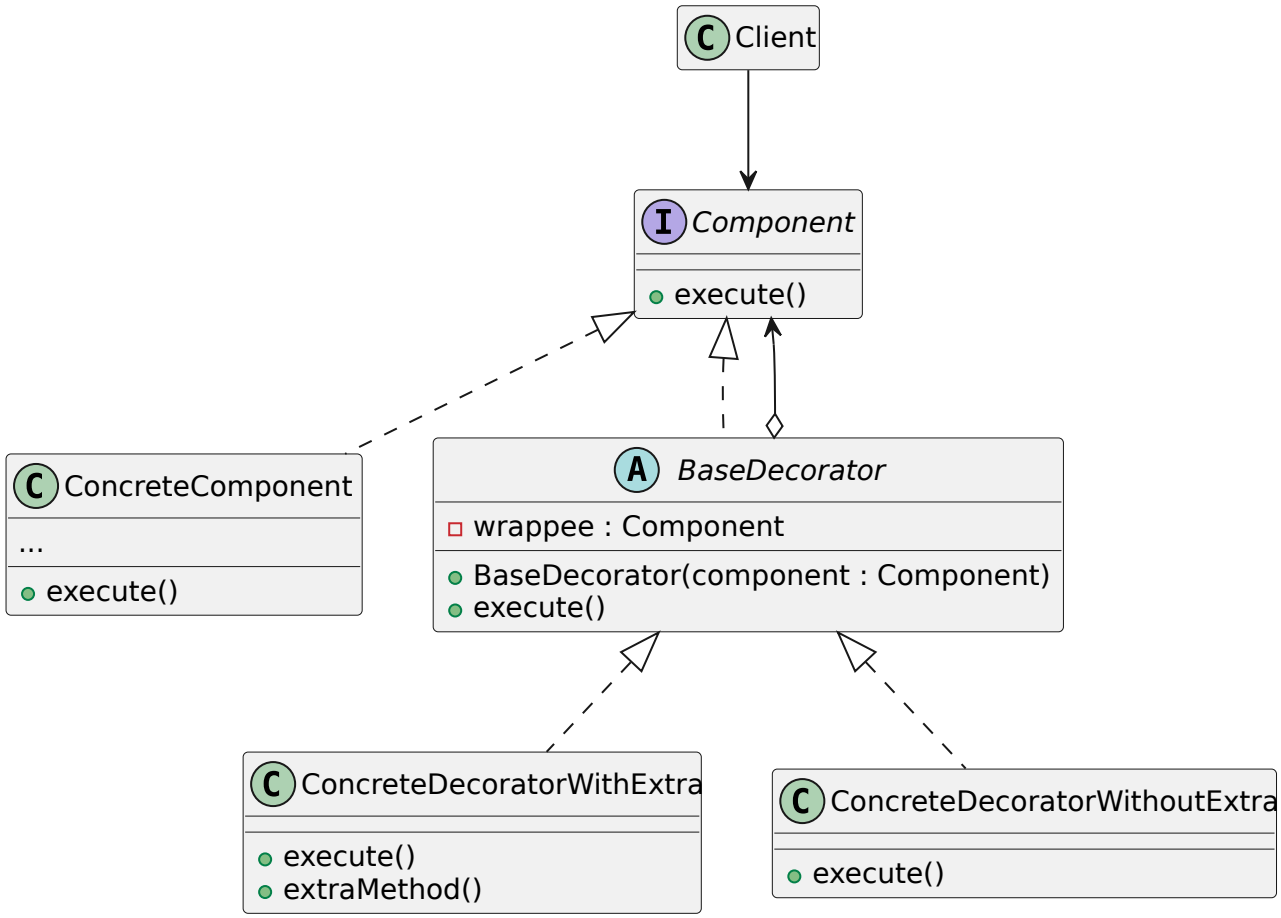
Il est possible d'utiliser plusieurs décorateurs simultanément :

```
Stack stack = new ArrayStack(10);
Stack verboseStack = new VerboseStack(stack);
CounterStack counterStack = new CounterStack(verboseStack);
stack = counterStack;
stack.push(2);
stack.push(3);
stack.pop();
System.out.println(counterStack.getSize());
```

Ce code produit la sortie suivante :

```
push(2);
push(3);
pop();
1
```

La structure générale du patron décorateur est la suivante :



2.4 Patrons comportementaux

Les comportementaux qui permettent de gérer les algorithmes et la répartition des responsabilités entre les objets. Les principaux patrons comportementaux sont les suivants :

— *Chain of Responsibility* (chaîne de responsabilité) : permet de faire circuler une demande dans une chaîne de *handlers*. Lorsqu'un *handler* reçoit une demande, il décide de la traiter ou de l'envoyer au *handler* suivant de la chaîne.

— *Command* (commande) : Prend une action à effectuer et la transforme en un objet autonome qui contient tous les détails de cette action. Cette transformation permet de paramétrer des méthodes avec différentes actions, planifier leur exécution, les mettre dans une file d'attente ou d'annuler des opérations effectuées.

— *Iterator* (itérateur) : permet de parcourir les éléments d'une collection sans révéler sa représentation interne (liste, pile, arbre, etc.).

— *Mediator* (médiateur) : Permet de diminuer les dépendances chaotiques entre les objets. Ce patron

restreint les communications directes entre les objets et les force à collaborer uniquement via un objet médiateur.

— *Memento* (Mémento) : Permet de sauvegarder et de rétablir l'état précédent d'un objet sans révéler les détails de son implémentation.

— *Observer* (observateur) : Permet de mettre en place un mécanisme de souscription pour envoyer des notifications à plusieurs objets, au sujet d'événements concernant les objets qu'ils observent.

— *State* (état) : Modifie le comportement d'un objet lorsque son état interne change. L'objet donne l'impression qu'il change de classe.

— *Strategy* (stratégie) : Permet de définir une famille d'algorithmes, de les mettre dans des classes séparées et de rendre leurs objets interchangeables.

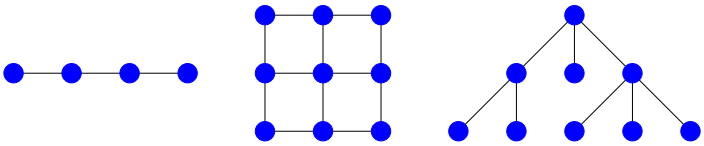
— *Template Method* (patron de méthode) : Permet de mettre le squelette d'un algorithme dans la classe mère, mais laisse les sous-classes redéfinir certaines étapes de l'algorithme sans changer sa structure.

— *Visitor* (Patron de méthode) : Permet de séparer les algorithmes et les objets sur lesquels ils opèrent.

2.4.1 *Iterator* (itérateur)

Itérateur est un patron de conception comportemental qui permet de parcourir les éléments d'une collection sans révéler sa représentation interne (liste, pile, arbre, ...).

Les collections font partie des types de données les plus usitées en programmation. Elles servent de conteneur pour un groupe d'objets généralement en utilisant une structure linéaire : listes, tableaux ... Néanmoins, certaines d'entre elles sont basées sur structures plus complexes comme des arbres, des graphes ou d'autres structures complexes de données.



Quelle que soit sa structure, une collection doit fournir un moyen d'accéder à ses éléments pour permettre au code de les utiliser. Elle doit donner la possibilité de parcourir tous ses éléments sans passer plusieurs fois par les mêmes. Le but du patron de conception itérateur est d'extraire le comportement qui permet de parcourir une collection et de le mettre dans un objet que l'on nomme itérateur.

En java (c'est le cas aussi pour d'autres langages), le patron de conception itérateur est défini dans le langage et la bibliothèque standard. La première chose à définir pour le patron de conception itérateur est l'interface itérable qui indique que l'on peut itérer (parcourir les éléments) sur la collection. En Java, l'interface est la suivante (on a volontairement omis certaines méthodes) :

```
1 public interface Iterable<T> {
2     /**
3      * Returns an iterator over elements of type {@code T}.
4      *
5      * @return an Iterator.
6      */
7     Iterator<T> iterator();
```

```
8 }
```

L'interface `Iterable` est d'ailleurs étendue par l'interface `Collection` de Java. Elle ne contient qu'une méthode qui retourne un objet pour itérer sur la collection.

L'interface `Iterator` donne les méthodes à implémenter pour un objet permettant de parcourir les éléments d'une collection :

```
1  public interface Iterator<E> {
2      /**
3       * Returns {@code true} if the iteration has more elements.
4       * (In other words, returns {@code true} if {@link #next} would
5       * return an element rather than throwing an exception.)
6       *
7       * @return {@code true} if the iteration has more elements
8       */
9      boolean hasNext();
10
11     /**
12      * Returns the next element in the iteration.
13      *
14      * @return the next element in the iteration
15      * @throws NoSuchElementException if the iteration has no more elements
16      */
17     E next();
18 }
```

On a deux méthodes à implémenter `hasNext` qui permet de tester s'il reste des éléments sur lesquels on peut itérer et `next` qui renvoie le prochain élément dans l'ordre d'itération et met à jour l'itérateur pour que le prochain appel de `next` renvoie l'élément après et ainsi de suite. Pour itérer sur un tableau, on peut écrire le code suivant :

```
1  public class IteratorArray {
2    Object[] array;
3    int position = 0;
4    IteratorArray(Object[] array){
5      this.array = array;
6    }
7
8    boolean hasNext(){
9      return (position<array.length);
10   }
11
12   Object next(){
13     position++;
14     return array[position-1];
15   }
16 }
```

Pour illustrer l'utilisation du patron de conception itérateur, on considère la classe suivante qui permet de stocker des éléments dans une grille avec des lignes et des colonnes.

```
1  public class GridContainer<E> implements Iterable<E> {
```

```
  private final int numberOfRows;
  private final int numberOfColumns;
  private final Object[][] elements;
  public GridContainer(int numberOfRows, int numberOfColumns) {
    if(numberOfRows <= 0)
      throw new IllegalArgumentException("The number of rows must be positive
          and not equal to " + numberOfRows);
    if(numberOfColumns <= 0)
      throw new IllegalArgumentException("The number of columns must be
          positive and not equal to " + numberOfColumns);
    this.numberOfRows = numberOfRows;
    this.numberOfColumns = numberOfColumns;
    elements = new Object[numberOfRows][numberOfColumns];
  }
  @SuppressWarnings("unchecked")
  public E getElement(int rowIndex, int columnIndex) {
    return (E) elements[rowIndex][columnIndex];
  }
  public void setElement(int rowIndex, int columnIndex, E value) {
    elements[rowIndex][columnIndex] = value;
  }

  public int getNumberOfRows() {
    return numberOfRows;
  }

  public int getNumberOfColumns() {
    return numberOfColumns;
  }

  @Override
  public Iterator<E> iterator() {
    return new GridIterator<>(this);
  }
}
```

L'itérateur permettant de visiter les éléments de la grille ligne par ligne a le code suivant :

```
public class GridContainer<E> implements Iterable<E> {
  private final int numberOfRows;
  private final int numberOfColumns;
  private final Object[][] elements;
  public GridContainer(int numberOfRows, int numberOfColumns) {
    if(numberOfRows <= 0)
      throw new IllegalArgumentException("The number of rows must be positive
          and not equal to " + numberOfRows);
    if(numberOfColumns <= 0)
      throw new IllegalArgumentException("The number of columns must be
          positive and not equal to " + numberOfColumns);
    this.numberOfRows = numberOfRows;
    this.numberOfColumns = numberOfColumns;
    elements = new Object[numberOfRows][numberOfColumns];
  }
  @SuppressWarnings("unchecked")
  public E getElement(int rowIndex, int columnIndex) {
    return (E) elements[rowIndex][columnIndex];
```

```
17    }
18    public void setElement(int rowIndex, int columnIndex, E value) {
19      elements[rowIndex][columnIndex] = value;
20    }
21
22    public int getNumberOfRows() {
23      return numberOfRows;
24    }
25
26    public int getNumberOfColumns() {
27      return numberOfColumns;
28    }
29
30    @Override
31    public Iterator<E> iterator() {
32      return new GridIterator<>(this);
33    }
34  }
```

Puisque `GridContainer` implémente `Iterable`, on peut l'utiliser dans une boucle *for each* (aussi appelé *enhanced for*) qui permet directement de parcourir les éléments de `GridContainer` dans l'ordre défini par l'itérateur. Si on considére le code suivant :

```
1  GridContainer<String> grid = new GridContainer<>(2,3);
2  for(int row = 0; row<grid.getNumberOfRows(); row++)
3    for (int column = 0; column<grid.getNumberOfColumns(); column++)
4      grid.setElement(row,column,"(" + row + ", " + column + ")");
5  for (String element : grid)
6    System.out.println(element);
```

Ce code produit la sortie suivante :

```
1  (0, 0)
2  (0, 1)
3  (0, 2)
4  (1, 0)
5  (1, 1)
6  (1, 2)
```

2.4.2 *Visitor* (visiteur)

Visiteur est un patron de conception comportemental qui vous permet de séparer les implémentations de méthodes et les objets sur lesquels ils opèrent.

Dans une section précédente, nous avons défini l'interface suivante :

```
1  public interface Shape {
2    void draw(GraphicsContext context);
3    float area();
4  }
```

Dans cette interface, sont regroupés deux services assez différents (calcul géométrique et dessin dans un contexte graphique) ce qui constitue une violation de SRP.

Cette interface est implémentée par la classe suivante :

```
1  public class Rectangle implements Shape {
2      public float x, y, w, h;
3
4      public Rectangle(float x, float y, float w, float h) {
5          this.x = x;
6          this.y = y;
7          this.w = w;
8          this.h = h;
9      }
10
11     public void draw(GraphicsContext context) {
12         context.strokeRect(x, y, h, w);
13     }
14
15     public float area() {
16       return w * h;
17     }
18 }
```

Dans cette classe, il y a l'implémentation de deux services assez différents. On peut imaginer des raisons différentes pour lesquelles le code des deux méthodes pourrait changer (changement de bibliothèque graphique pour `draw` et passage à un type de retour `double` pour `area` afin d'éviter des problèmes de représentation d'aires trop grande pour être stocké dans un `float`). La classe ne respecte donc pas SRP dans cette optique.

On peut représenter les correspondances entre classes et méthodes dans un tableau :

		classes			
		Rectangle	Circle	Polygon	···
méthodes	draw()				
	area()				
	···				

Dans le tableau ci-dessus, on a :

- Une classe par colonne
- Une méthode par ligne
- SRP violé, car plusieurs responsabilités dans chaque classe
- OCP violé, car le nombre de lignes peut augmenter

Une solution est de définir une classe par ligne en utilisant le patron de conception visiteur. On commence par créer une interface `Shape` qui va accepter des visiteurs.

```
1  public interface Shape {
2    <R> R accept(ShapeVisitor<R> visitor);
3  }
```

Ensuite, on définit une interface pour les visiteurs avec une méthode de visite par classe à visiter. L'interface est paramétrée par le type de retour de la méthode de visite. Cela permet d'avoir par exemple un visiteur pour

le calcul des aires qui renvoie un `Float` et un visiteur de dessin qui ne renvoie rien avec `Void`.

```
public interface ShapeVisitor<R> {
  R visit(Rectangle rectangle);
  R visit(Circle circle);
}
```

Dans les classes `Circle` et `Rectangle`, qui implémentent chacune l'interface `Shape`, n'ont plus qu'une seule méthode `accept` et un constructeur :

```
public class Circle implements Shape {
  public final float x, y;
  public final float radius;

  public Circle(float x, float y, float radius) {
    this.x = x;
    this.y = y;
    this.radius = radius;
  }

  @Override
  public <R> R accept(ShapeVisitor<R> visitor) {
    return visitor.visit(this);
  }
}
```

```
public class Rectangle implements Shape {
  public float x, y, w, h;

  public Rectangle(float x, float y, float w, float h) {
    this.x = x; this.y = y; this.w = w; this.h = h;
  }

  @Override
  public <R> R accept(ShapeVisitor<R> visitor) {
    return visitor.visit(this);
  }
}
```

La méthode `accept` se contente d'appeler la méthode `visit` du visiteur sur l'objet courant et de renvoyer le résultat. Toute l'implémentation de la méthode sera donc dans le visiteur. On peut remarquer que bien que le nom de fonction appelée soit le même (`visit`), les méthodes appelées sont différentes. En effet, dans `Circle` c'est la méthode `R visit(Circle circle)` qui est appelée, car `this` est de type `Circle` alors que dans `Rectangle`, c'est la méthode `R visit(Rectangle rectangle)` qui est appelée.

Maintenant, on doit fournir des implémentations de l'interface `Visitor` qui permettent le dessin et le calcul de l'aire. On commence donc par l'implémentation du visiteur par une classe `DrawerVisitor` :

```
public class DrawerVisitor implements ShapeVisitor<Void> {

  private GraphicsContext context;

  public DrawerVisitor(GraphicsContext context) {
```

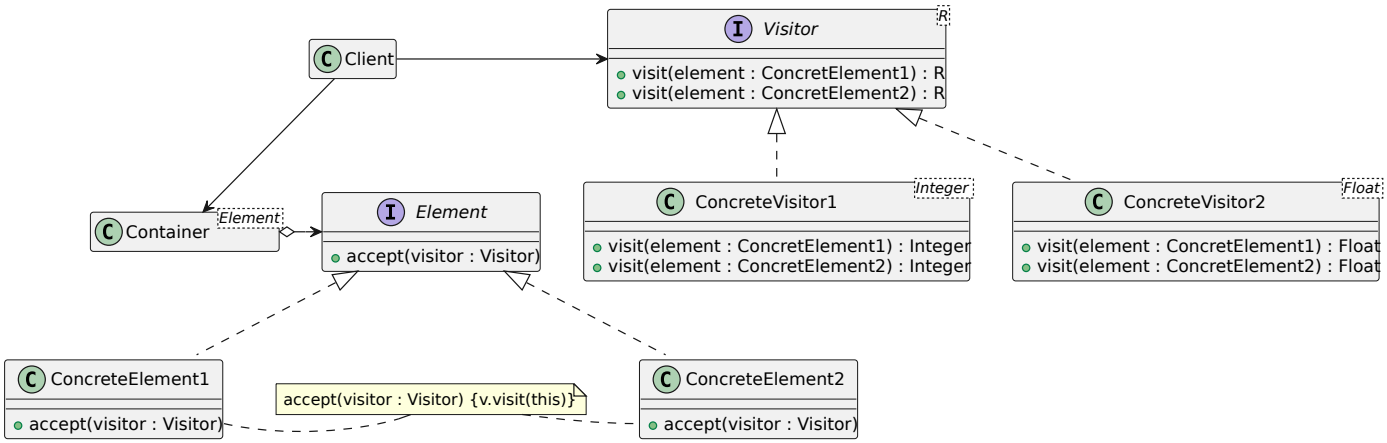
```
6       this.context = context;
7     }
8
9     public void draw(List<Shape> shapes) {
10      for (Shape shape : shapes)
11        shape.accept(this);
12    }
13
14    @Override
15    public Void visit(Rectangle rectangle) {
16      context.strokeRect(rectangle.x, rectangle.y,
17                         rectangle.h, rectangle.w);
18      return null;
19    }
20
21    @Override
22    public Void visit(Circle circle) {
23      context.strokeOval(circle.x - circle.radius,
24                         circle.y - circle.radius,
25                         circle.radius*2, circle.radius*2);
26      return null;
27    }
28  }
```

Ici, on a une petite particularité du fait que le type de retour est `Void` qui représente le type `void` dans le cas d'un type générique. Pour ce type de retour, la seule valeur acceptée est `null` d'où les `return null` dans les méthodes de visite.

On peut continuer avec l'implémentation du visiteur `AreaVisitor` :

```
1   public class AreaVisitor implements ShapeVisitor<Float> {
2
3     public float sumOfArea(List<Shape> shapes) {
4       float sum = 0;
5       for (Shape shape : shapes)
6         sum += shape.accept(this);
7       return sum;
8     }
9
10
11    @Override
12    public Float visit(Rectangle rectangle) {
13      return rectangle.w * rectangle.h;
14    }
15
16    @Override
17    public Float visit(Circle circle) {
18      return Math.pow(circle.radius,2) * Math.PI;
19    }
20  }
```

De manière générale, le patron de conception visiteur a la structure suivante :



3 Conclusion

Nous avons vu que 23 patrons de conceptions existent et nous n'en avons étudié que certains. Si ce domaine vous intéresse je vous recommande l'ouvrage de référence, écrit par les quatre concepteurs initiaux des patrons de conceptions : "Design patterns. Catalogue des modèles de conception réutilisables" par Gamma, Helm, Johnson et Vlissides chez Vuibert Informatique. Il faut connaître les patrons de conceptions pour pouvoir les utiliser dans des cas précis, tout en faisant attention de ne pas en abuser. Ils permettent de résoudre beaucoup de problèmes, mais ne sont pas toujours optimaux et peuvent parfois introduire une complexité excessive. De même, il peut être très difficile de respecter tous les principes SOLID. En fait, ces principes s'appliquent dans le cadre d'un code qui évolue. Par exemple, en ce qui concerne SRP, si votre code n'évolue pas d'une manière telle que deux responsabilités présentes au sein d'une même classe changent à des moments différents, il n'est pas nécessaire de les séparer. En effet, les séparer serait une source de complexité inutile. Il y a un corollaire à cela. Un axe de changement n'est un axe de changement que si les changements se produisent. Il n'est pas judicieux d'appliquer SRP, ou tout autre principe, d'ailleurs, s'il n'y a pas de symptôme.

Ce cours finit la partie sur la Conception Orientée Objet. Vous devez maintenant posséder l'essentiel des connaissances vous permettant de concevoir des logiciels avec cette approche. De même, en fonction de vos affinités ou de vos obligations, vous allez développer cette compétence dans un langage particulier. Dans le cadre de ce cours, on a utilisé exclusivement le langage Java, mais rien ne vous empêche d'utiliser d'autres langages permettant le paradigme de programmation orienté objet comme Python, C++, ...