amU Faculté des sciences Aix Marseille Université

Année 2024-25

Domaine Sciences et Technologies
Master informatique
Génie logiciel : SOLID
Codes UE : SINA09EL
Principes SOLID

Principes SOLID

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 :

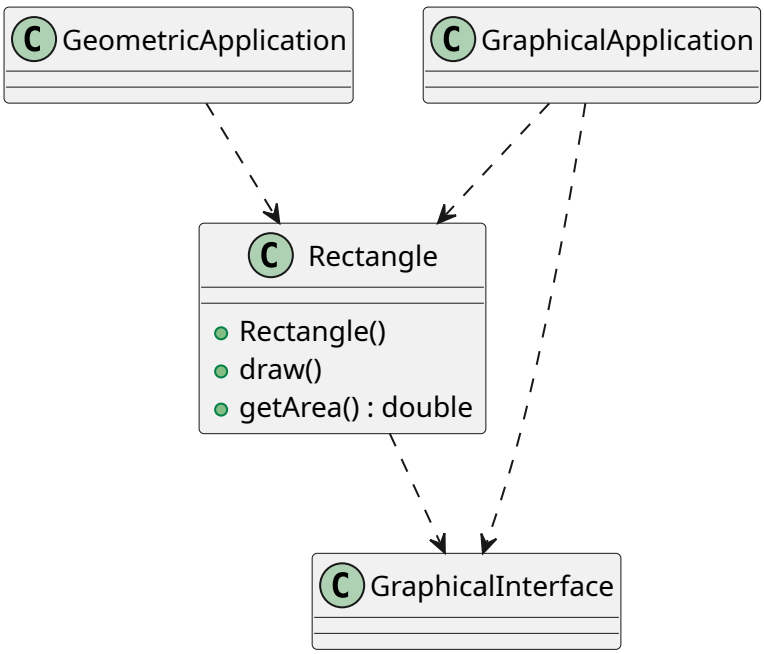
— tester à nouveau l'implémentation des autres responsabilités ;
— modifier potentiellement les autres responsabilités (les modifications apportées à une responsabilité pouvant compromettre la capacité de la classe à assurer ses autres responsabilités) ;
— déployer à nouveau les autres responsabilités.

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

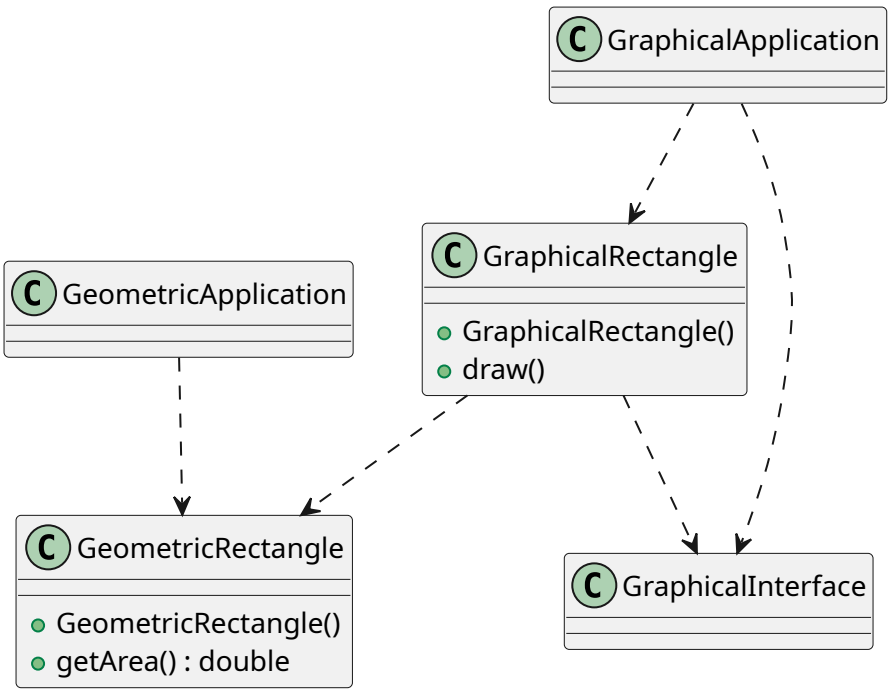
Afin d'illustrer ce principe, nous allons décrire un exemple plus concret. On va considérer une classe `Rectangle` qui a deux méthodes : une méthode `draw()` permettant de dessiner le rectangle et une méthode `area()` calculant l'aire de celui-ci. Deux classes différentes utilisent la classe `Rectangle` : `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);
      }
    }
```

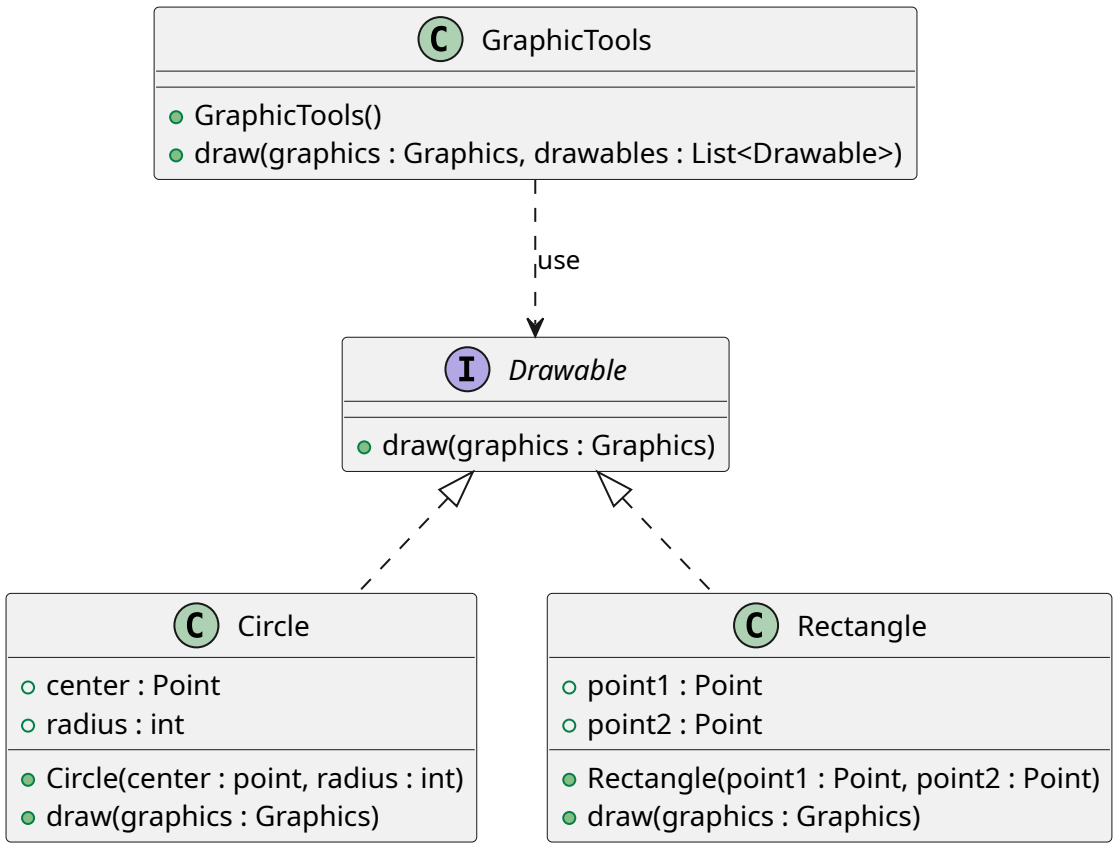
```
  }
}
```

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.

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

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

```
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 :

```
public class Rectangle {
  private double width;
  private double height;

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

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

  public double getWidth() {
    return width;
  }
```

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

  public double getArea() {
    return width*height;
  }
}
```

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

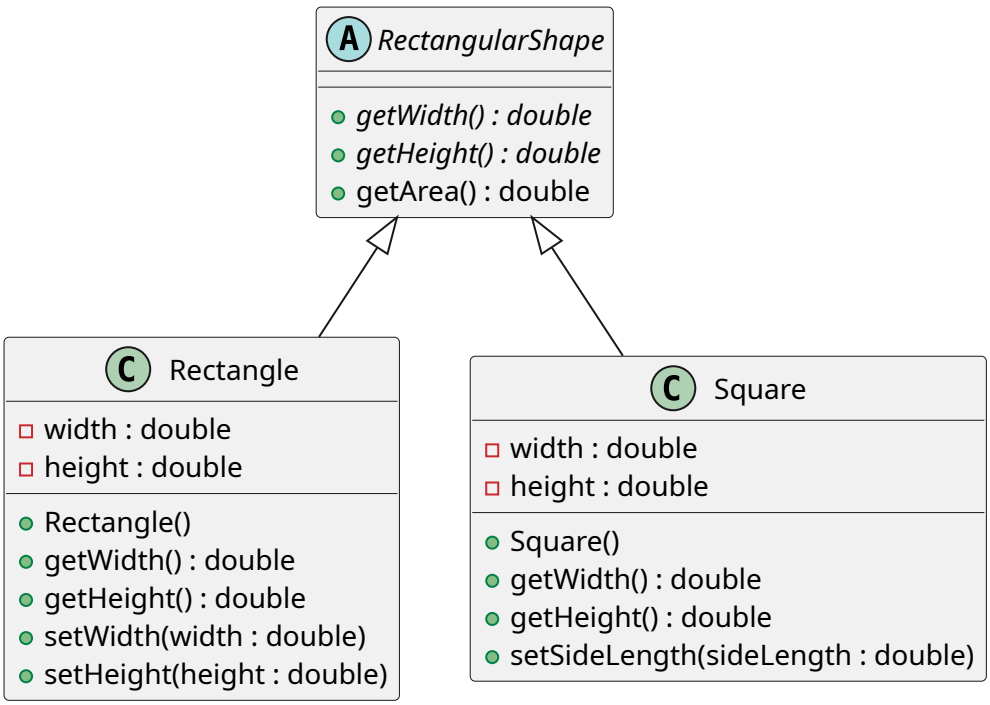
```
public class Square extends Rectangle {

  public void setWidth(double width) {
    super.setWidth(width);
    super.setHeight(width);
  }

  public void setHeight(double height) {
    super.setWidth(height);
    super.setHeight(height);
  }
}
```

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

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

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 :

```
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.

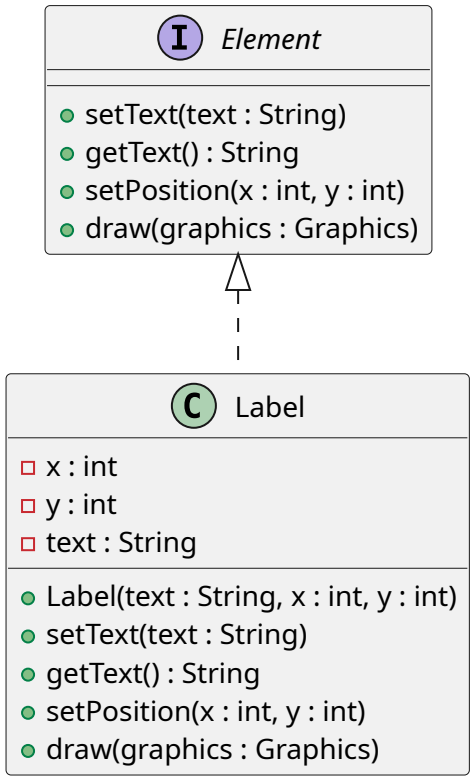
Il faut donc éviter qu'un client ne voit une interface qu'il n'utilise pas. Dans la plupart des cas, appliquer ce principe revient à éviter les interfaces qui contiennent beaucoup de méthodes :

- 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 :

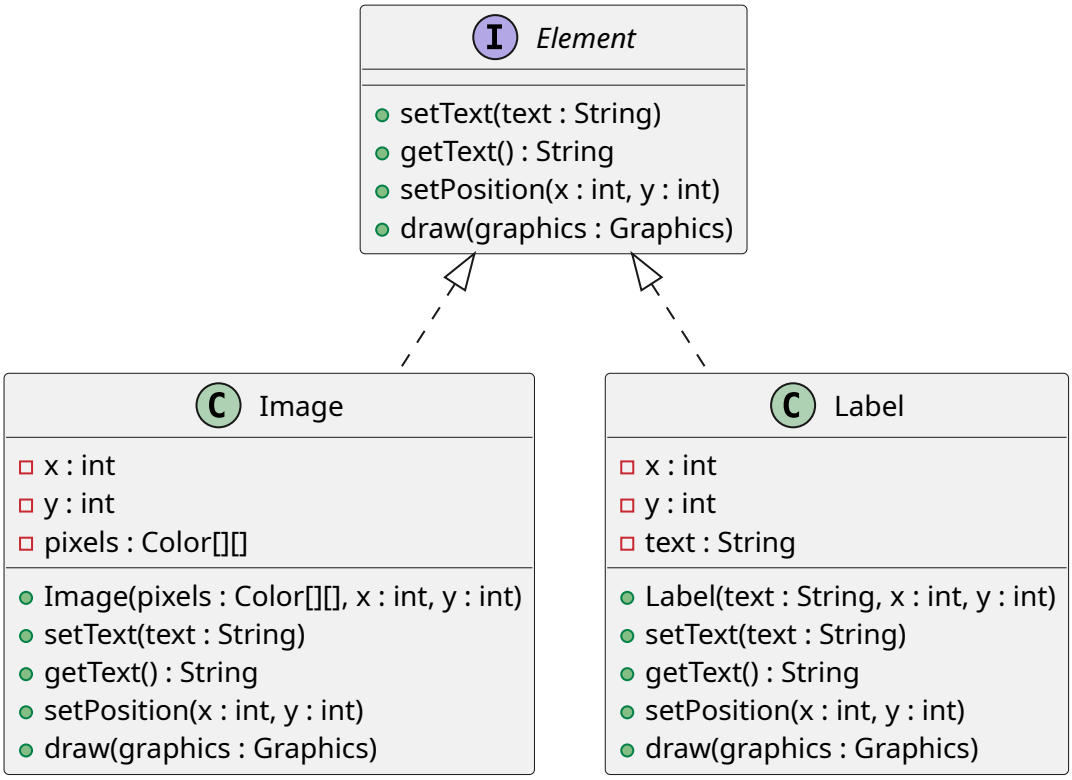


Cela nous donne le code suivant :

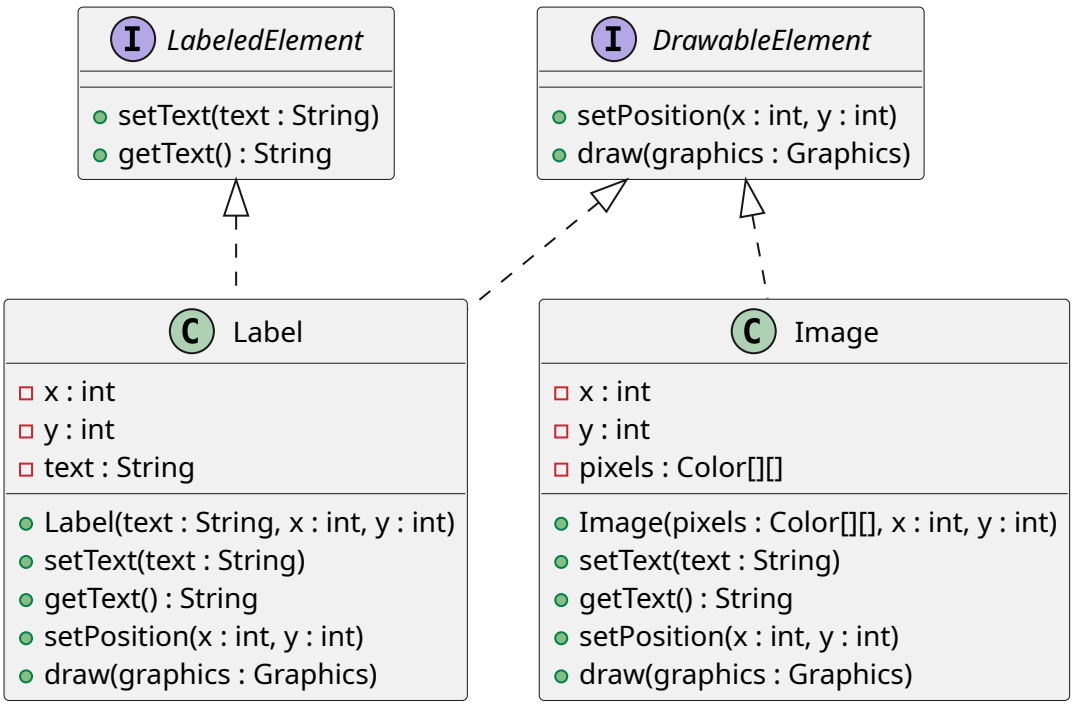
```
public interface Element {
  public void setText(String text);
  public String getText(String text);
  public void setPosition(int x, int y);
  public void draw(Graphics graphics);
}

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 :

```
public class Thermostat {
  enum IOChannel{
    THERMOMETER, FURNACE
  }
  enum Action{
    ENGAGE, DISENGAGE
  }

  int read(IOChannel channel){
    // TODO : add code for reading on a channel
    return 0;
  }

  void write(IOChannel channel, Action action){
    // TODO : add code for writing on a channel
  }

  void Regulate(double minTemp, double maxTemp)
    throws InterruptedException{
    for(;;) {
      while (read(IOChannel.THERMOMETER) > minTemp)
        wait(1);
      write(IOChannel.FURNACE, Action.ENGAGE);
      while (read(IOChannel.THERMOMETER) < maxTemp)
        wait(1);
      write(IOChannel.FURNACE, Action.DISENGAGE);
    }
  }
}
```

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.