

Initiation génie logiciel : Principes SOLID

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

25 septembre 2024

amU Faculté
des sciences
Aix Marseille Université

Section 1

Introduction

Objectif d'une bonne conception logicielle

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

Un programme est “bien conçu” s'il permet de :

- 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

Objectifs :

- Limiter les modules impactés
- Simplifier les tests
- Rester conforme aux spécifications qui n'ont pas changé

Les cinq principes (pour créer du code) SOLID

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

Section 2

Single Responsibility Principle (SRP)

Single Responsibility Principle (SRP)

SRP (Robert C. Martin, 2004)

Un module ne doit être responsable que d'un (et un seul) acteur.

Acteur : groupe de personne pouvant demander un changement (utilisateurs, ...).

Module : partie du logiciel décrit dans un seul fichier

- Une responsabilité est une raison de changer
- Une classe ne doit avoir qu'une seule raison de changer

Pourquoi respecter SRP ?

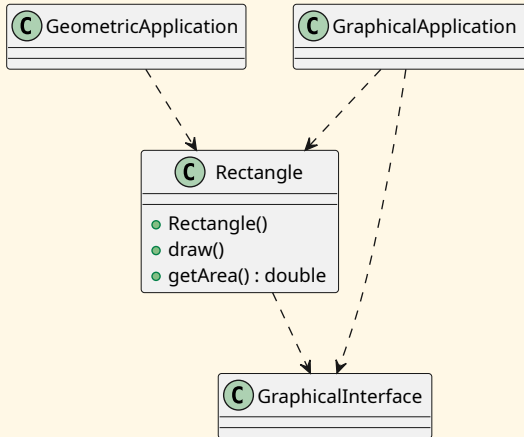
- Si une classe a plusieurs responsabilités, elles sont couplées
- Dans ce cas, la modification d'une des responsabilités nécessite de :
 - ▶ tester à nouveau l'implémentation des autres responsabilités
 - ▶ modifier potentiellement les autres responsabilités
 - ▶ déployer à nouveau les autres responsabilités
- Donc, vous risquez de :
 - ▶ introduire des bugs
 - ▶ de perdre du temps

Avantages SRP

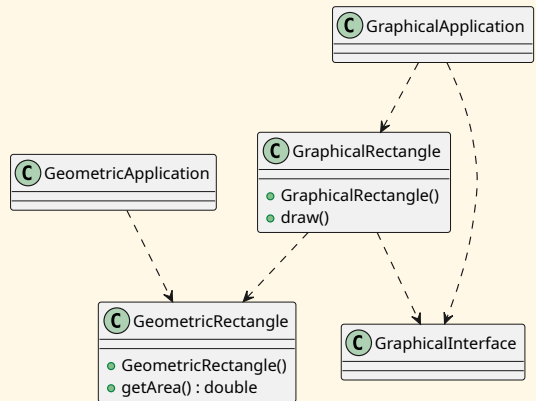
- Diminution de la complexité du code
- Amélioration de la lisibilité du code
- Meilleure organisation du code
- Modification locale lors des évolutions
- Augmentation de la fiabilité
- Les classes ont davantage de chance d'être réutilisables

Violation SRP

Violation de SRP



Séparation des responsabilités



SRP en une image



SINGLE RESPONSIBILITY PRINCIPLE

Just Because You Can, Doesn't Mean You Should

Section 3

Open/Closed Principle (OCP)

Open/Closed Principle (OCP)

OCP (Bertrand Meyer, 1988)

Programme ouvert pour l'extension, fermé à la modification

On doit pouvoir ajouter une nouvelle fonctionnalité :

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

Avantages :

- Le code existant n'est pas modifié \Rightarrow augmentation de la fiabilité
- Les classes ont plus de chance d'être réutilisables
- Simplification de l'ajout de nouvelles fonctionnalités

Exemple avec des classes Circle et Rectangle

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

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

Non-respect d'OCP (1/2)

```
public class GraphicTools {
    static void draw(Graphics graphics, List<Object> objects) {
        for (Object object : objects) {
            if (object instanceof Rectangle) {
                Rectangle rectangle = (Rectangle) object;
                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 (object instanceof Circle) {
                // Voir le slide suivant
            }
        }
    }
}
```

Non-respect d'OCP (2/2)

```
public class GraphicTools {
    static void draw(Graphics graphics, List<Object> objects) {
        for (Object object : objects) {
            if (object instanceof Rectangle) {
                /* Voir le slide précédent */
            } else if (object instanceof Circle) {
                Circle circle = (Circle) object;
                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);
            }
        }
    }
}
```

Refactoring du code (1/3)

Première étape : simplifier le code de la méthode.

```
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);
            }
        }
    }
}
```


Refactoring du code (2/3)

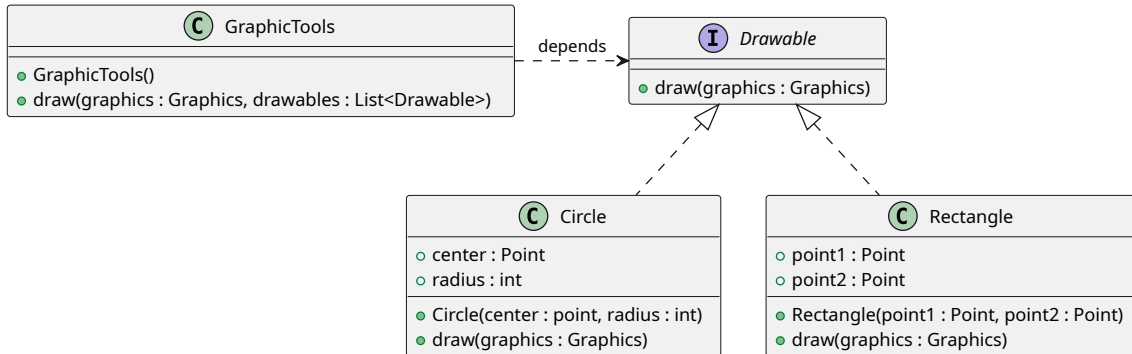
Créer une méthode pour dessiner

```
public class GraphicTools {  
    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);  
    }  
}
```

Refactoring du code (3/3)

```
public class GraphicTools {
    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);
    }
}
```

Solution avec une interface et de l'abstraction



Implémentation du diagramme

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

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

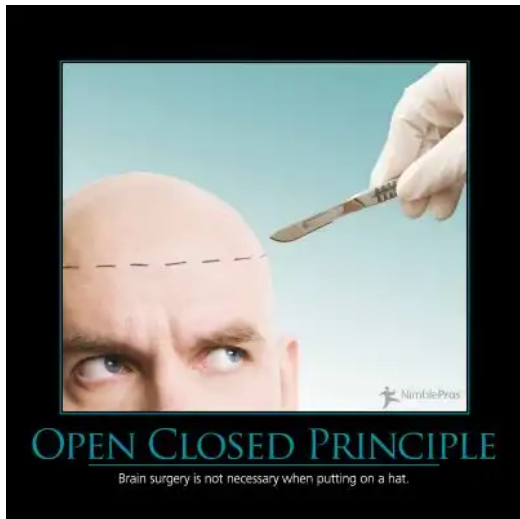
Classe Circle

```
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);
    }
}
```

Classe Rectangle

```
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);
    }
}
```

OCP en une image



Section 4

Liskov Substitution Principle (LSP)

Liskov Substitution Principle (LSP)

LSP (Barbara Liskov 1988)

Les sous-types doivent être substituables par leurs types de base

Si une classe **F** étend une classe **M** alors un programme **P** écrit pour manipuler des instances de type **M** doit avoir le même comportement s'il manipule des instances de la classe **F**.

Avantages :

- Diminution de la complexité du code
- Amélioration de la lisibilité du code
- Meilleure organisation du code
- Modification locale lors des évolutions
- Les classes ont plus de chance d'être réutilisables

Classe Rectangle

Une classe qui permet de représenter un rectangle géométrique :

```
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; }  
}
```

Classe Square extension de Rectangle

Un carré est un rectangle, on devrait pouvoir écrire :

```
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);
    }
}
```

Violation de LSP

```
public void test(Rectangle rectangle) {  
    rectangle.setWidth(2);  
    rectangle.setHeight(3);  
    assertThat(rectangle.getArea()).isEqualTo(3*2);  
}
```

La mauvaise question

Un carré **est**-il un rectangle ?

La bonne question

Pour les utilisateurs, votre carré **a-t-il le même comportement** que votre rectangle ?

La réponse : Dans ce cas, non !

Une solution

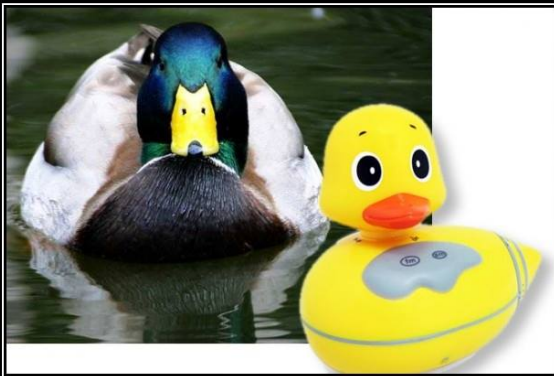
```
public interface RectangularShape {
    public abstract double getWidth();
    public abstract double getHeight();
    default double getArea() { return getWidth()*getHeight(); }
}

public class Rectangle implements 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; }
}
```

Une solution

```
class Square implements RectangularShape {
    private double sideLength;
    public void setSideLength(double sideLength) {
        this.sideLength = sideLength;
    }
    public double getWidth() { return sideLength; }
    public double getHeight() { return sideLength; }
}
```

LSP en une image



LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

Interface Segregation Principle (ISP)

ISP (Robert C. Martin, 2002)

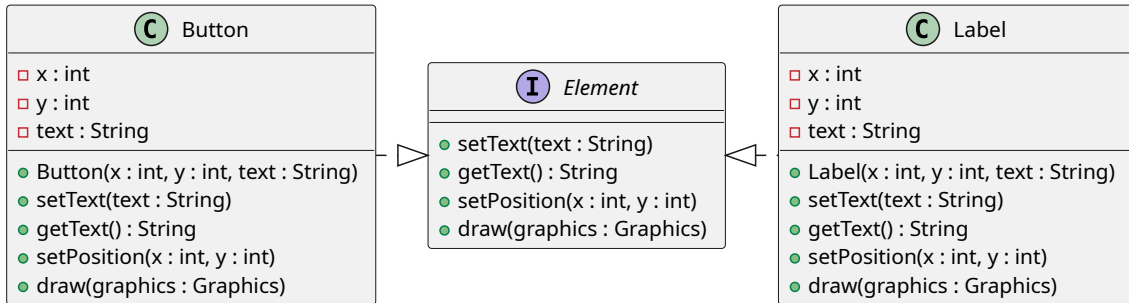
Aucun client ne devrait dépendre de méthodes qu'il n'utilise pas.

- 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

Avantages

- 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
- Simplification de l'ajout de nouvelles fonctionnalités

Exemple de violation de ISP



Implémentation

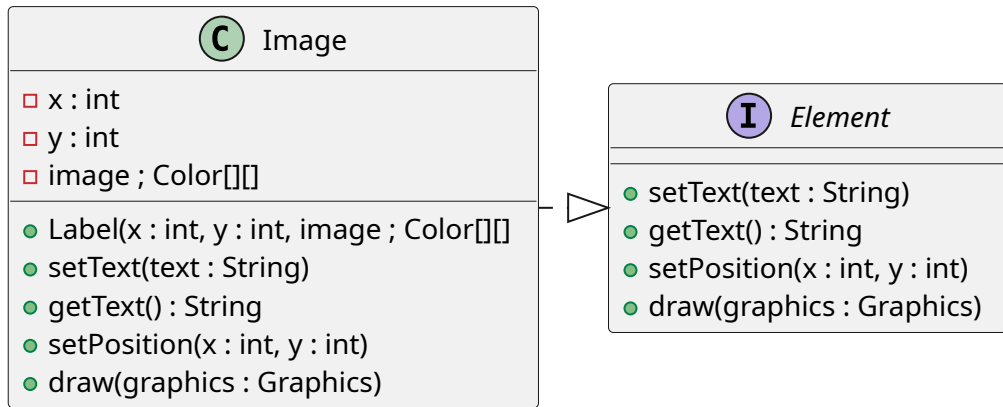
```
public interface Element {  
    public void setText(String text);  
    public String getText();  
    public void setPosition(int x, int y);  
    public void draw(Graphics graphics);  
}
```

Implémentation

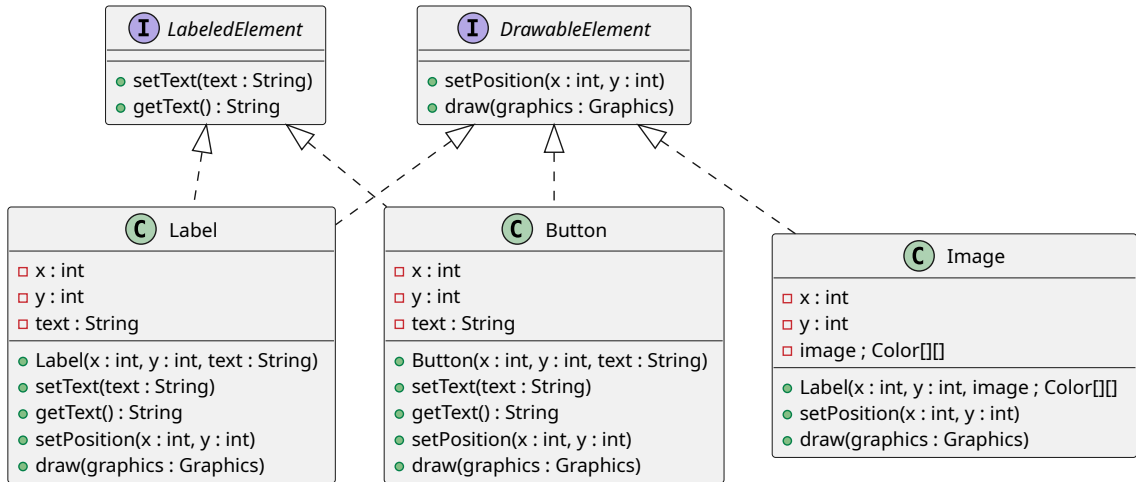
```
public class Label implements Element {
    private int x,y;
    private String text;
    public Label(String text, int x, int y) {
        this.text = text; this.x = x; this.y = y;
    }
    public void setText(String text) { this.text = text; }
    public String getText() { return text; }
    public void setPosition(int x, int y) {
        this.x = x; this.y = y;
    }
    public void draw(Graphics graphics) {
        graphics.drawString(text, x, y);
    }
}
```

Ajout d'une classe Image

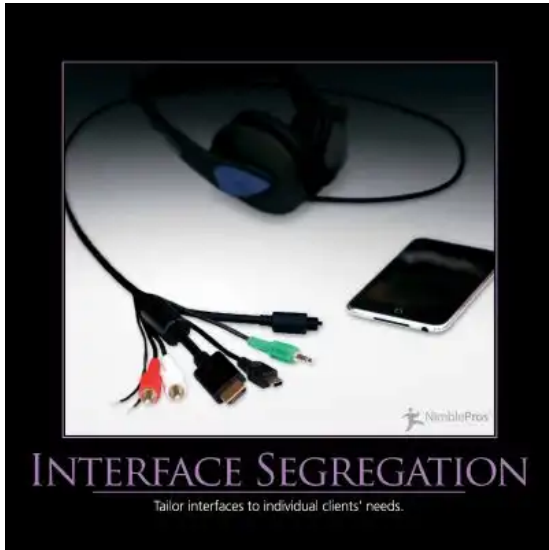
Une image n'a pas de texte : que faire dans setText et getText ?



Solution : découper l'interface



ISP en une image



Dependency Inversion Principle (DIP)

DIP (Robert C. Martin, 2009)

Les modules doivent dépendre d'abstractions (interfaces stables) ou d'éléments concrets stables (comme `String` en Java).

- Découpler les différents modules de votre programme
- Les lier en utilisant des interfaces
- Décrire correctement le comportement de chaque module
- Permet de remplacer un module par un autre module plus facilement

Avantages

- Les modules sont plus facilement réutilisables
- Simplification de l'ajout de nouvelles fonctionnalités
- L'intégration est rendue plus facile

Exemple Thermostat (1/2)

```
public class Thermostat {
    BufferedReader t;
    OutputStreamWriter furnace;
    void regulate(double minTemp, double maxTemp)
        throws IOException, InterruptedException{
    for(;;) {
        while (Double.parseDouble(t.readLine()) > minTemp)
            Thread.sleep(1000);
        furnace.write("On");
        while (Double.parseDouble(t.readLine()) < maxTemp)
            Thread.sleep(1000);
        furnace.write("Off");
    }
}
```


Exemple Thermostat (2/2)

Dépendance technique lecture/écriture de fichier

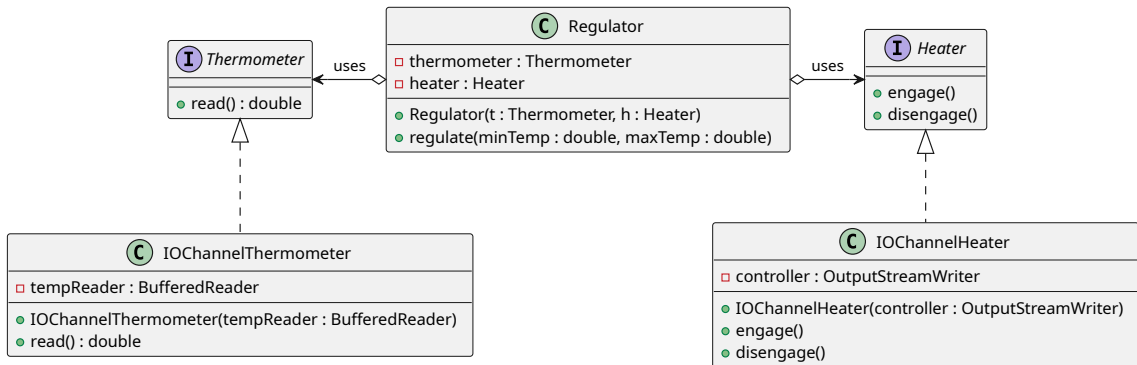
- lecture de la température via lecture fichier
- contrôle chauffage via écriture fichier

- Dépendance à une interface spécifique
- Impossible de réutiliser l'algorithme de régulation pour d'autres méthodes de contrôle

Solution

Extraire les méthodes d'accès requises et les mettre dans des interfaces.

Respect DIP (1/2)



Respect DIP (2/2)

```
public class Thermostat {  
    void Regulate(double minTemp, double maxTemp)  
        throws InterruptedException {  
        for(;;) {  
            while (thermometer.read() > minTemp)  
                Thread.sleep(1000);  
            heater.engage();  
            while (thermometer.read() < maxTemp)  
                Thread.sleep(1000);  
            heater.desengage();  
        }  
    }  
}
```

DIP en une image



DEPENDENCY INVERSION PRINCIPLE

Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

Section 5

Patron de conception *State*

Classe avec état

```
public class Writer {
    private int state = 0;
    public void write(char character) {
        switch (state) {
            case 0 -> {
                if (character=='{') state = 1;
                else System.out.print(character);
            }
            case 1 -> {
                if (character=='}') state = 0;
                else System.out.print(Character.toUpperCase(character));
            }
        }
    }
}
```

Utilisation de la classe

```
Writer writer = new Writer();  
String string = "abc{def}ghi";  
for (int index = 0; index < string.length(); index++)  
writer.write(string.charAt(index));
```

Le code précédent génère la sortie suivante : abcDEFghi

Notez que si nous souhaitons ajouter de nouveaux états, nous devons modifier les méthodes de la classe Writer

→ violation d'OCP

De plus, le fait qu'une classe implémente un grand nombre d'états peut être considéré comme une violation de SRP.

Solution

Pour corriger ces défauts, nous déclarons la classe et l'interface suivantes :

```
public class WriterContext {
    private WriterState state = new WriterState0();
    public void write(char character) {
        state.write(this, character);
    }
    public void changeState(WriterState state) {
        this.state = state;
    }
}

public interface WriterState {
    public void write(WriterContext context, char character);
}
```


Nous pouvons maintenant définir les différents états :

```
public class WriterStateIdentity implements WriterState {
    public void write(WriterContext context, char c) {
        if (c=='{') {context.changeState(new WriterStateUpper());}
        else System.out.print(c);
    }
}

public class WriterStateUpperCase implements WriterState {
    public void write(WriterContext context, char c) {
        if (c=='}') {context.changeState(new WriterStateIdentity());}
        else System.out.print(Character.toUpperCase(c));
    }
}
```

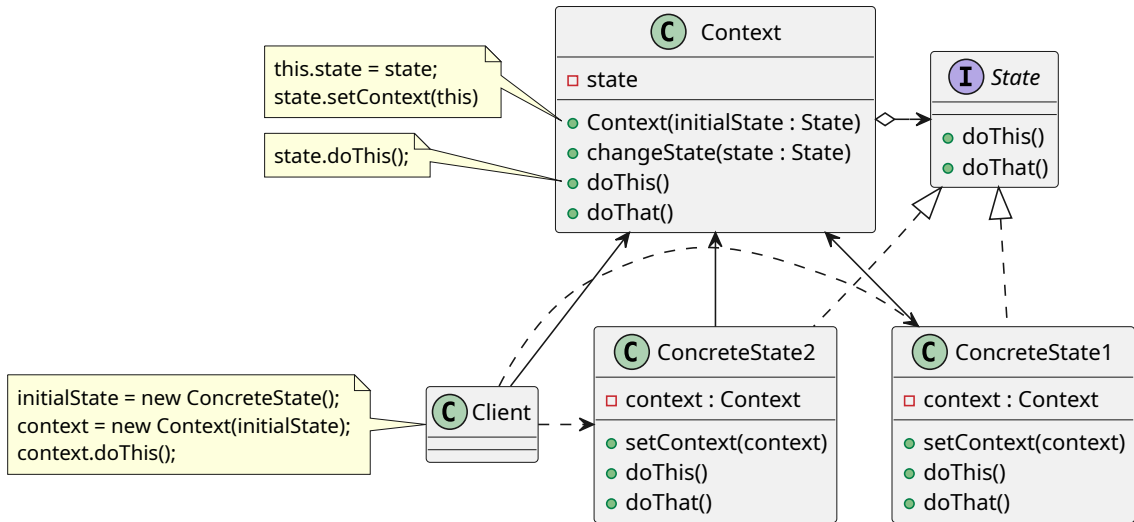
Un exemple d'utilisation de la classe précédente :

```
WriterContext writer = new WriterContext();  
String string = "abc{def}ghi";  
for (int index = 0; index < string.length(); index++)  
writer.write(string.charAt(index));
```

Ce code génère la sortie suivante : abcDEFghi

Notez que l'ajout d'un nouvel état s'effectue en ajoutant une nouvelle classe qui implémente l'interface `WriterState` (respect d'OCP).

Diagramme du patron *state*



Patron de conception *state*

Intention

Permet de modifier le comportement d'un objet lorsque son état interne change.

Solution

Implémenter un automate fini en créant :

- 1 une interface état contenant les méthodes que doit faire chaque état ;
- 2 pour chaque état possible une classe pour cet état qui gère les opérations et les transitions ;
- 3 une classe contexte contenant un état et déléguant les opérations à l'état courant.

Avantages et inconvénients

Avantages :

- Organiser le code des différents états dans des classes séparées (SRP).
- Ajouter de nouveaux états sans modifier les classes état ou le contexte existants (OCP).
- Simplifiez le code du contexte en éliminant les gros blocs conditionnels de l'automate.

Désavantages :

- Créer de la complexité inutile (patron surdimensionné pour un automate ayant peu d'états et de transitions)