

UE INFO 4067 TP : GROUPE 10

# **RAPPORT DESIGN PATTERN : ÉTAT**

SUPERVISE PAR : DR VALERY MONTHE

## MEMBRES DU GROUPE ET POURCENTAGE DE PARTICIPATIONS

Noms & Prénoms	Matricules	Pourcentages
DONGMO GIRESSSE	20U2925	80 %
DONGMO DJOUAKE LEONEL MAKEN	20U2922	100 %
TCHAPTCHET KOUDJO CEDRIC YOHANN	18T2367	80 %
TEGOMO DYVANE DEGAR	20v2299	100 %

# PLAN DU TRAVAIL

- I. Introduction
- II. Design Pattern "État"
- III. Mise en œuvre
- IV. Comparaison avec d'autres design patterns
- V. Cas d'utilisation réel
- VI. Conclusion

# I. Introduction

- **A. Définition des design patterns**

- Les design patterns sont des solutions réutilisables à des problèmes communs rencontrés lors de la conception de logiciels.
- Ils offrent des modèles de conception éprouvés pour résoudre des problèmes spécifiques de manière efficace.

- **B. Importance des design patterns dans la conception logicielle**

- Les design patterns favorisent la réutilisabilité du code.
- Ils améliorent la maintenabilité du code en fournissant des solutions testées.
- Ils facilitent la communication entre les membres de l'équipe de développement.

## II. Design Pattern "État"

- **A. Définition**

- 1. Qu'est-ce que le pattern "État" ?
- Le pattern "État" permet à un objet de modifier son comportement lorsqu'il change d'état interne.
- Il encapsule chaque état dans une classe distincte et permet à l'objet de changer de classe lorsque son état change.
- 2. Son rôle dans la conception logicielle
- Fournir une structure pour la gestion des états d'un objet.
- Permettre une extension facile en ajoutant de nouveaux états ou en modifiant les transitions entre les états.

## II. Design Pattern "État"

- **B. Objectif**

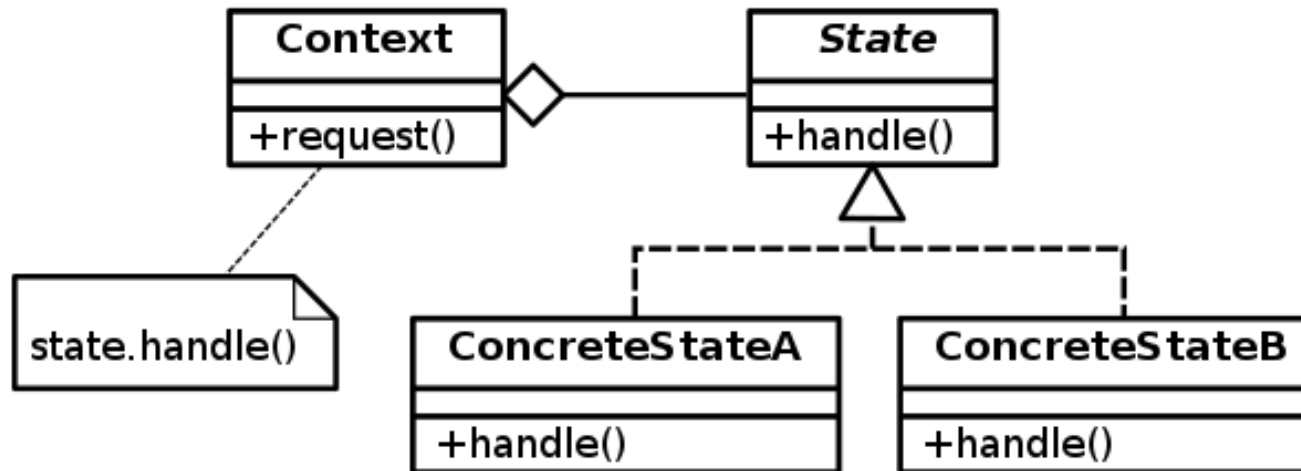
- 1. Pourquoi utiliser le pattern "État" ?
- Simplifier la gestion des états d'un objet.
- Réduire la complexité du code en isolant le comportement lié à chaque état.
- Faciliter l'ajout de nouveaux états sans modifier le code existant.
- 2. Avantages de son utilisation
- Modularité accrue.
- Maintenance simplifiée.
- Meilleure extensibilité.

- **C. Structure**

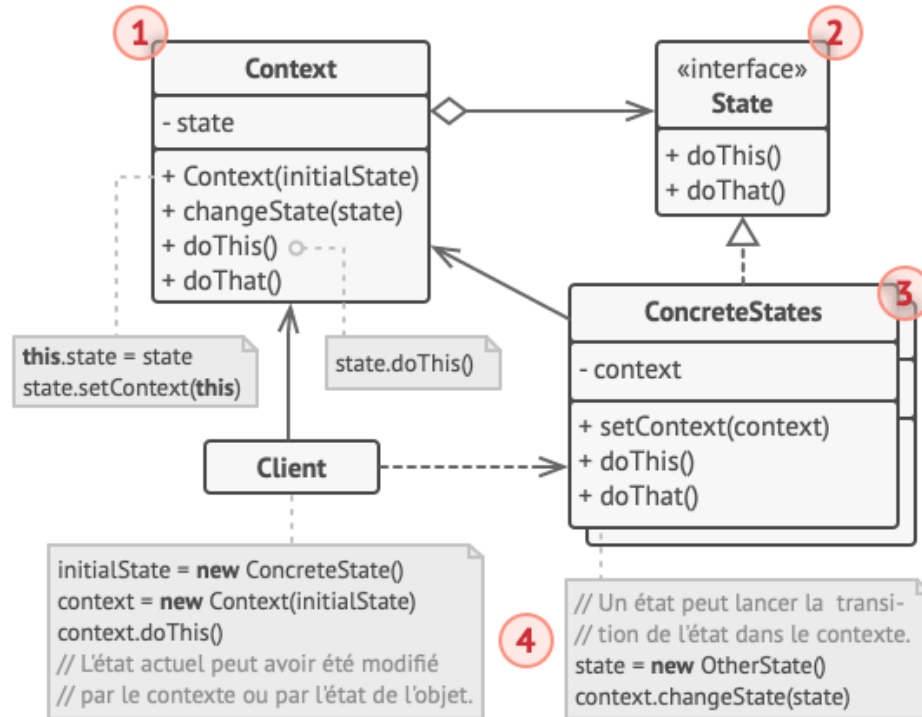
- 1. Les éléments clés du pattern "État"
- Contexte
- États concrets
- Interface d'état

## II. Design Pattern "État"

### Modele generique



## II. Design Pattern "État"





## II. Design Pattern "État"

- 2. Relations entre ces éléments
- Le contexte maintient une référence vers l'objet d'état actuel.
- Les états concrets implémentent l'interface d'état.
- **D. Exemple d'application**
- 1. Illustration d'un cas d'utilisation concret
- Imaginons un distributeur automatique de boissons avec des états tels que "En attente", "Sélectionné", "En préparation", etc.
- 2. Comment le pattern "État" résout les problèmes spécifiques
- Chaque état gère son propre comportement, rendant le code plus lisible.
- Les transitions entre les états sont clairement définies.

# III. Mise en œuvre

## A. Étapes de mise en œuvre du pattern "État"

- 1. Identifier les états possibles

Lors de l'identification des états possibles, il est crucial de comprendre les différentes situations ou conditions sous lesquelles l'objet peut se trouver. Par exemple, si nous prenons l'exemple d'un ordinateur, les états possibles pourraient être "Allumé", "Éteint", "En veille", etc.

- 2. Définir les transitions entre les états

Une fois les états identifiés, déterminez les conditions qui déclenchent le passage d'un état à un autre. Par exemple, pour le cas de l'ordinateur, la transition de "Allumé" à "En veille" pourrait être déclenchée par une période d'inactivité.

- 3. Implémenter les classes d'état

Chaque état identifié doit être représenté par une classe distincte. Ces classes doivent implémenter une interface commune (l'interface d'état) pour garantir une structure cohérente. Prenons l'exemple d'un lecteur de médias avec des états "Lecture", "Pause" et "Arrêt". Nous aurions trois classes : `LectureState`, `PauseState`, et `ArretState`, toutes implémentant une interface `MediaState`.

### III. Mise en œuvre : Code

**// Interface commune pour les états**

```
public interface MediaState {  
    void action();  
}
```

- **// Implémentation pour l'état de lecture**

```
public class LectureState implements MediaState {  
    @Override  
    public void action() {  
        System.out.println("Lecture en cours");  
    }  
}
```

### III. Mise en œuvre : Code

**// Implémentation pour l'état de pause**

- **public class PauseState implements MediaState {  
    @Override  
    public void action() {  
        System.out.println("Lecture en pause");  
    }  
}**

### III. Mise en œuvre : Code

**// Implémentation pour l'état d'arrêt**

- **public class ArretState implements MediaState {  
    @Override  
    public void action() {  
        System.out.println("Lecture arrêtée");  
    }  
}**

### III. Mise en œuvre : Code

#### 4. Intégrer les transitions dans le code

- Le contexte, qui est l'objet ayant un état, doit être capable de changer dynamiquement d'état en fonction des transitions définies. Créez une classe de contexte et ajoutez-y une référence à l'interface d'état. Ensuite, utilisez cette référence pour déléguer le comportement aux différentes classes d'état.

### III. Mise en œuvre : Code

#### 4. Intégrer les transitions dans le code

- `public class LecteurMedia {`
- `private MediaState etatActuel;`
- 
- `public void setEtat(MediaState nouvelEtat) {`
- `this.etatActuel = nouvelEtat;`
- `}`
- 
- `public void appuyerSurLecture() {`
- `etatActuel.action();`
- `}`
- `}`

# III. Mise en œuvre : Code

## 4. Intégrer les transitions dans le code

- // Exemple d'utilisation
- `public class Main {`
- `public static void main(String[] args) {`
- `LecteurMedia lecteur = new LecteurMedia();`
- 
- `lecteur.setEtat(new LectureState());`
- `lecteur.appuyerSurLecture(); // Affiche "Lecture en cours"`
- 
- `lecteur.setEtat(new PauseState());`
- `lecteur.appuyerSurLecture(); // Affiche "Lecture en pause"`
- `}`
- `}`



# III. Mise en œuvre : Code

## B. Exemple de code

- 1. Un exemple simple en code pour illustrer la mise en œuvre

Le code ci-dessus est un exemple simple illustrant l'application du pattern "État". Il représente un lecteur de médias avec trois états possibles : lecture, pause et arrêt.

- 2. Explication du code

Dans cet exemple, le lecteur de médias (LecteurMedia) a une référence à l'interface d'état (MediaState). Lorsque la méthode appuyerSurLecture est appelée, elle délègue l'action à l'état actuel. En changeant dynamiquement l'état du lecteur de médias, vous pouvez modifier son comportement sans altérer le code client. Cela illustre comment le pattern "État" permet une gestion flexible des états d'un objet.

## III. Mise en œuvre : Code

### 3- Exemple fonctionnement d'une lampe

On considère un système de gestion du fonctionnement d'une lampe. La lampe peut être allumée ou éteinte suivant le besoin d'un utilisateur. Ainsi donc, on souhaiterait doter le système d'un programme permettant la gestion des différents états de fonctionnement de la lampe suivant un besoin bien précis.

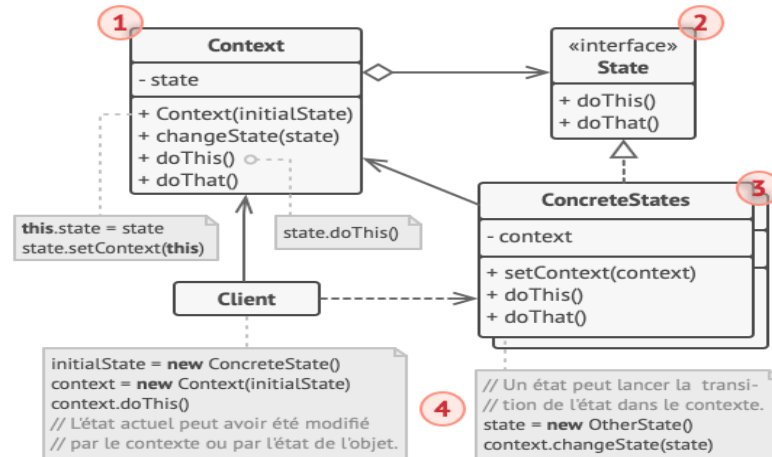
- 1) Quel est le patron de conception le mieux adapté à ce problème
- 2) Donner sa structure générique et décrire les participants
- 3) Proposer une modélisation du problème à l'aide de ce patron
- 4) Donner le code de votre solution

### III. Mise en œuvre : Code

1) Quel est le patron de conception le mieux adapté à ce problème

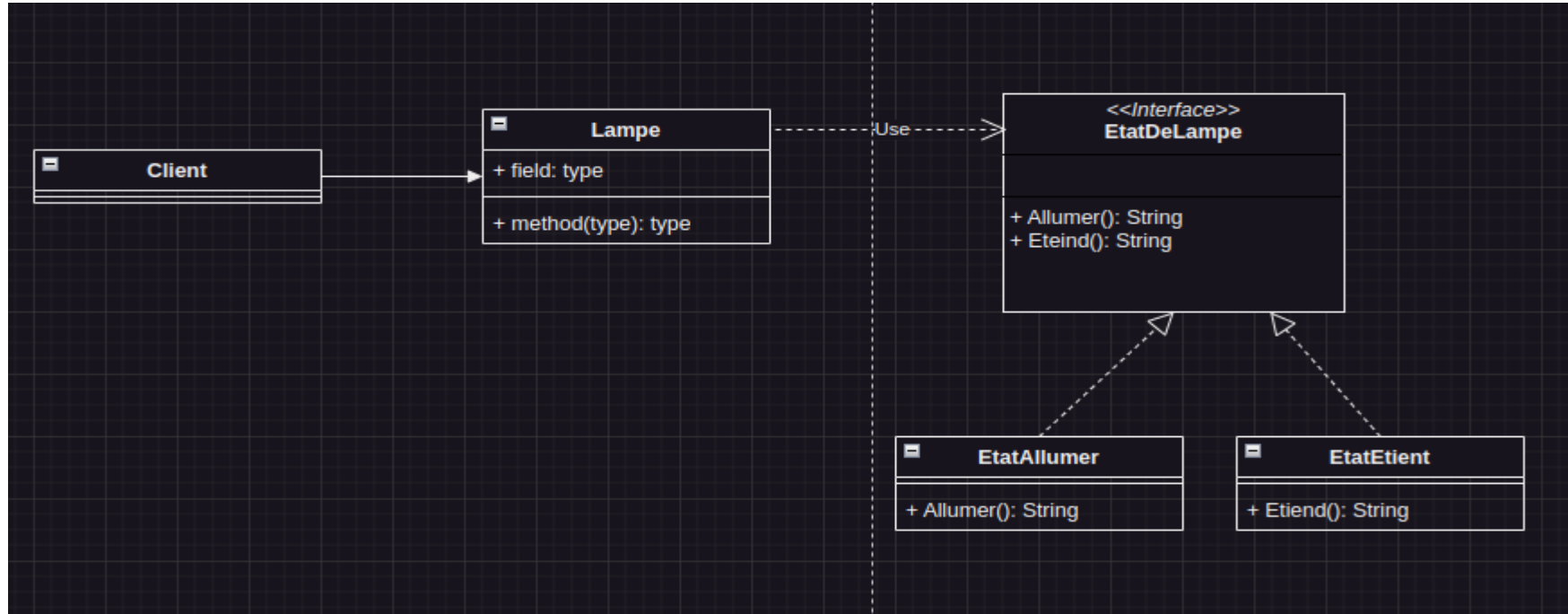
- Le patron d'état

2) Donner sa structure générique et décrire les participants



### III. Mise en œuvre : Code

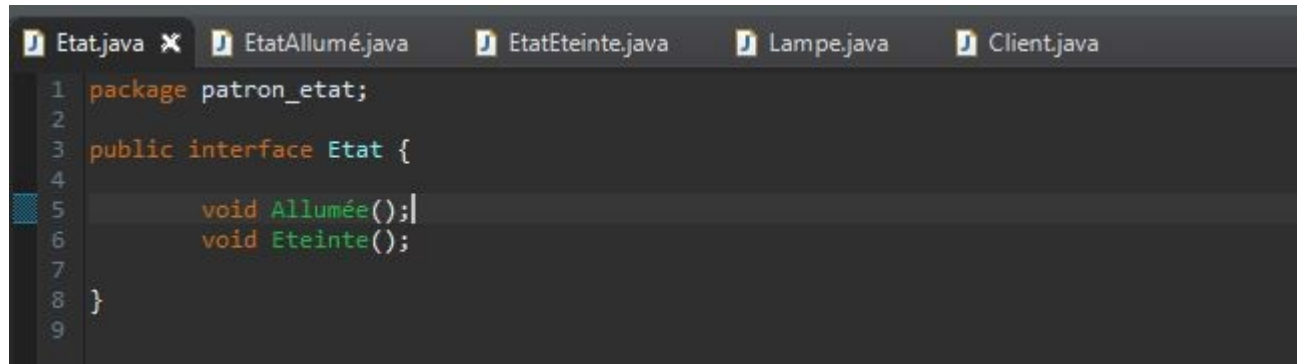
3) Proposer une modélisation du problème à l'aide de ce patron



## III. Mise en œuvre : Code

4) Donner le code de votre solution

\*Interface Etat.java

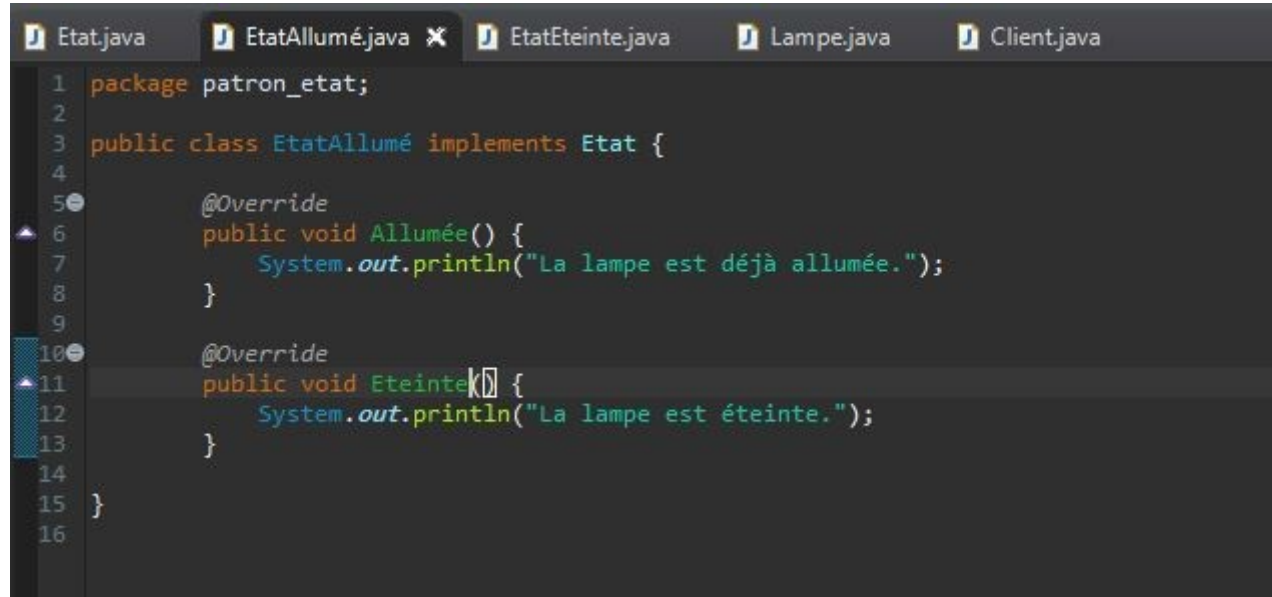
A screenshot of a Java IDE with a dark theme. The top of the window shows five tabs: 'Etat.java' (active), 'EtatAllumé.java', 'EtatEteinte.java', 'Lampe.java', and 'Client.java'. The main editor area displays the code for the 'Etat' interface. The code is as follows:

```
1 package patron_etat;
2
3 public interface Etat {
4
5     void Allumée();|
6     void Eteinte();
7
8 }
9
```

## III. Mise en œuvre : Code

4) Donner le code de votre solution

**\*Classe EtatAllumé implémentant l'interface Etat**



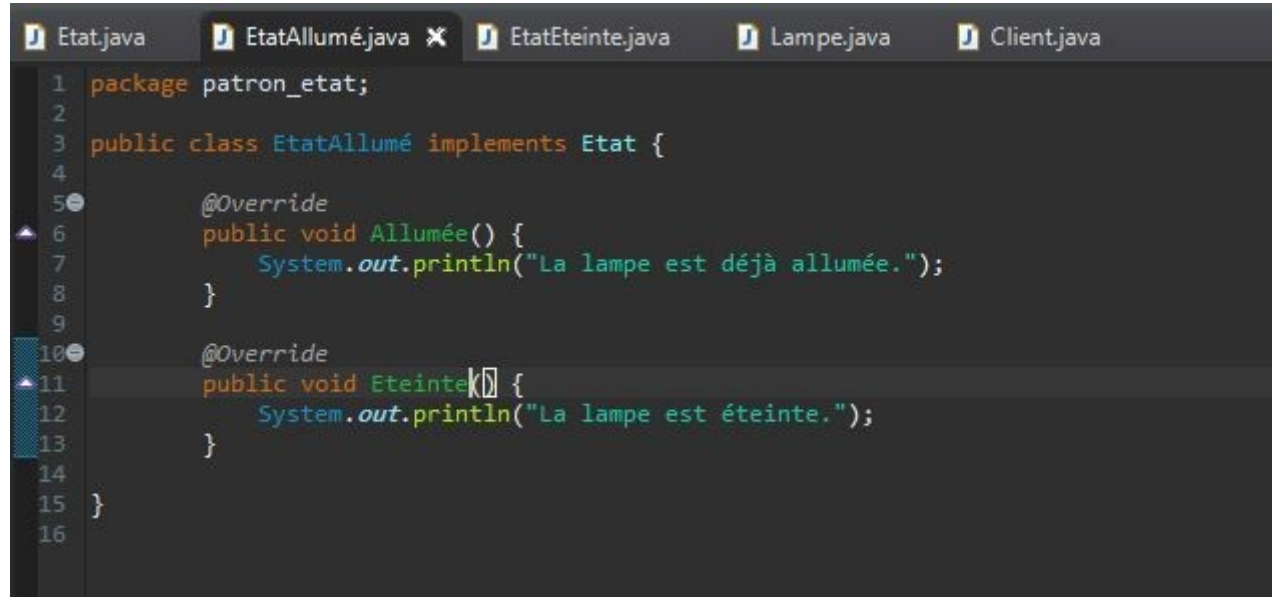
```
Etat.java  EtatAllumé.java  EtatEteinte.java  Lampe.java  Client.java

1 package patron_etat;
2
3 public class EtatAllumé implements Etat {
4
5     @Override
6     public void Allumée() {
7         System.out.println("La lampe est déjà allumée.");
8     }
9
10    @Override
11    public void Eteinte() {
12        System.out.println("La lampe est éteinte.");
13    }
14
15 }
16
```

### III. Mise en œuvre : Code

4) Donner le code de votre solution

**\*Classe EtatAllumé implémentant l'interface Etat**

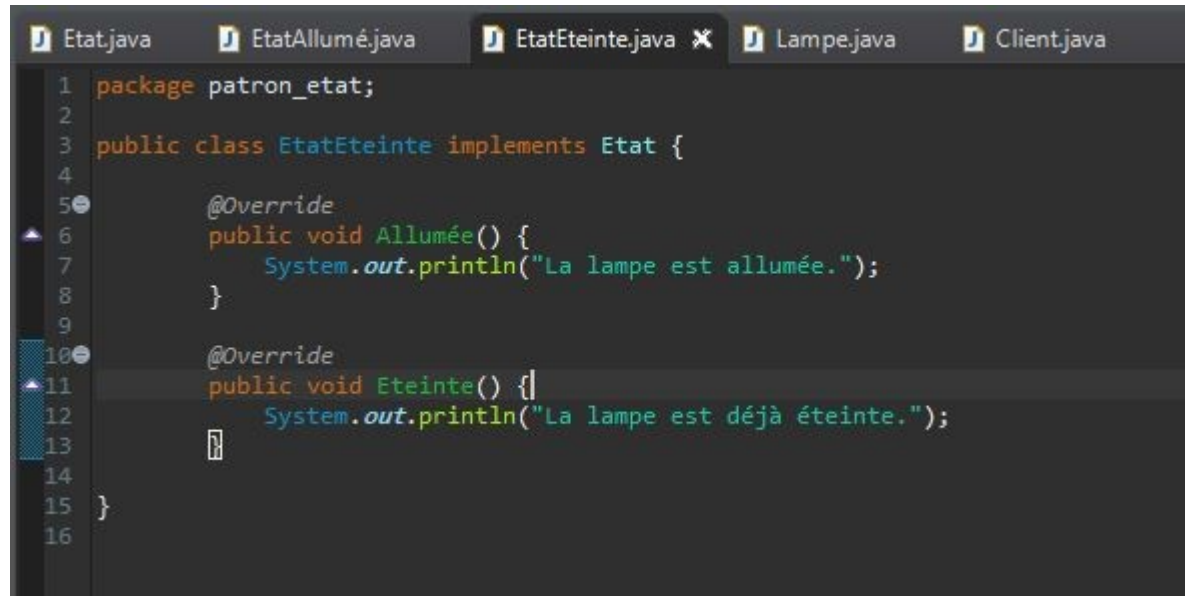


```
Etat.java  EtatAllumé.java  EtatEteinte.java  Lampe.java  Client.java
1 package patron_etat;
2
3 public class EtatAllumé implements Etat {
4
5     @Override
6     public void Allumée() {
7         System.out.println("La lampe est déjà allumée.");
8     }
9
10    @Override
11    public void Eteinte() {
12        System.out.println("La lampe est éteinte.");
13    }
14
15 }
16
```

## III. Mise en œuvre : Code

4) Donner le code de votre solution

**\*Classe EtatEteinte implémentant l'interface Etat**



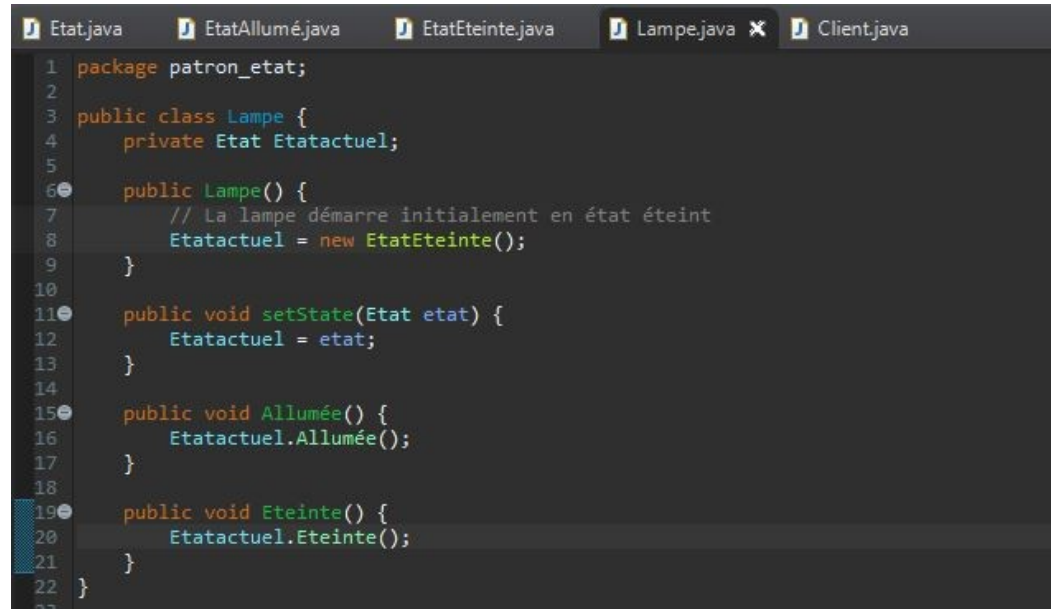
```
1 package patron_etat;
2
3 public class EtatEteinte implements Etat {
4
5     @Override
6     public void Allumée() {
7         System.out.println("La lampe est allumée.");
8     }
9
10    @Override
11    public void Eteinte() {
12        System.out.println("La lampe est déjà éteinte.");
13    }
14
15 }
16
```



# III. Mise en œuvre : Code

4) Donner le code de votre solution

**Classe Lampe pour l'implémentation des différents états**



```
1 package patron_etat;
2
3 public class Lampe {
4     private Etat Etatactuel;
5
6     public Lampe() {
7         // La lampe démarre initialement en état éteint
8         Etatactuel = new EtatEteinte();
9     }
10
11     public void setState(Etat etat) {
12         Etatactuel = etat;
13     }
14
15     public void Allumée() {
16         Etatactuel.Allumée();
17     }
18
19     public void Eteinte() {
20         Etatactuel.Eteinte();
21     }
22 }
```

# III. Mise en œuvre : Code

4) Donner le code de votre solution

**Classe Lampe pour l'implémentation des différents états**

```
public class Main {
    Run | Debug
    public static void main(String[] args) {
        Lampe lampe = new Lampe();
        Scanner scanner = new Scanner(System.in);

        while (true) {
            System.out.println("1. Allumer la lampe");
            System.out.println("2. Éteindre la lampe");
            System.out.println("3. Quitter");

            System.out.print("Choisissez une option : ");
            int choix = scanner.nextInt();

            switch (choix) {
                case 1:
                    lampe.allumer();
                    break;
                case 2:
                    lampe.eteindre();
                    break;
                case 3:
                    System.out.println("Au revoir !");
                    System.exit(0);
                default:
                    System.out.println("Option invalide. Veuillez réessayer.");
            }
        }
    }
}
```

# IV. Comparaison avec d'autres design patterns

## A. Différences avec d'autres patterns

### • 1. Différences avec le pattern "Stratégie"

#### **Pattern "État" :**

- Se concentre sur la gestion des états internes d'un objet.
- L'objet peut changer d'état dynamiquement pendant son cycle de vie.
- Les classes d'état encapsulent le comportement spécifique à chaque état.
- Les transitions entre les états sont généralement définies dans le contexte.

#### **Pattern "Stratégie" :**

- Se concentre sur la définition d'une famille d'algorithmes interchangeables.
- Les algorithmes sont encapsulés dans des classes distinctes, appelées stratégies.
- Le client peut choisir dynamiquement la stratégie à utiliser.
- Les changements de stratégie n'impliquent pas nécessairement un changement d'état.

## IV. Comparaison avec d'autres design patterns

### 2. Différences avec le pattern "Observateur"

#### **Pattern "État" :**

- Se concentre sur la gestion des états internes d'un objet.
- Les changements d'état peuvent être observés, mais le pattern ne se concentre pas spécifiquement sur la notification des observateurs.

#### **Pattern "Observateur" :**

- Permet à un objet (sujet) de maintenir une liste d'objets dépendants (observateurs) qui sont informés des changements d'état.
- Les observateurs sont notifiés lorsque le sujet subit un changement d'état.
- Le pattern "Observateur" est plus axé sur la communication entre objets que sur la gestion des états internes.

# IV. Comparaison avec d'autres design patterns

## B. Quand choisir le pattern "État" par rapport à d'autres patterns similaires

### 1. Choix du pattern "État"

- Situation adaptée au pattern "État" :
- 
- L'objet a un comportement qui change de manière significative en fonction de son état interne.
- Il y a un ensemble défini d'états et des transitions claires entre ces états.
- L'objectif est de rendre les transitions d'état transparentes pour le client.
- Exemple d'application :
- 
- Gestion de l'état d'une machine, d'un processus, ou d'un système complexe où le comportement varie en fonction des conditions internes.

### 2. Choix du pattern "Stratégie"

#### Situation adaptée au pattern "Stratégie" :

- 
- Le client a besoin de choisir dynamiquement entre plusieurs algorithmes interchangeables.
- Il existe des familles d'algorithmes qui peuvent être encapsulées de manière indépendante.
- Exemple d'application :
- 
- Calcul de tarifs d'expédition avec différentes stratégies en fonction du type de produit ou du mode d'expédition.

## V. Cas d'utilisation réel

### A. Exemples d'utilisation du pattern "État" dans des projets logiciels bien connus

- **1. Application mobile de gestion de tâches**

Dans de nombreuses applications mobiles de gestion de tâches, le pattern "État" est utilisé pour gérer l'état des tâches individuelles. Chaque tâche peut être dans des états tels que "À faire", "En cours", ou "Terminée". La transition entre ces états est gérée de manière transparente, permettant aux utilisateurs de suivre l'avancement de leurs tâches.

- 
- **2. Système de réservation de billets en ligne**

Dans un système de réservation de billets en ligne, le pattern "État" peut être appliqué pour gérer l'état des sièges dans une salle de cinéma. Les états possibles incluent "Disponible", "Réservé" et "Acheté". Les transitions entre ces états se produisent en fonction des actions des utilisateurs, telles que la réservation ou l'achat de billets.

## V. Cas d'utilisation réel

### B. Retours d'expérience et avantages observés

- **1. Simplification de la gestion des états**

Dans un projet de gestion de commandes en ligne, l'application du pattern "État" a simplifié la gestion des différentes phases de traitement des commandes. Chaque état représente une phase du processus, comme "En attente de paiement", "En préparation", "Expédié", etc. Les avantages observés incluent une meilleure lisibilité du code et une facilité d'extension du système pour prendre en charge de nouveaux états.

- **2. Flexibilité accrue dans les simulations de jeux**

Dans le domaine des simulations de jeux, le pattern "État" a été utilisé pour gérer les comportements des personnages en fonction de situations changeantes. Par exemple, dans un jeu de stratégie en temps réel, les unités peuvent passer d'états comme "Attaque" à "Défense" en fonction des circonstances. Cela offre une flexibilité significative dans la modélisation des interactions complexes entre les unités du jeu.

## V. Cas d'utilisation réel

### 3. Gestion des états d'une machine industrielle

- Dans un système de contrôle industriel, le pattern "État" a été appliqué pour gérer les différents états d'une machine de production. Les états incluent "Arrêt", "En marche", "Maintenance nécessaire", etc. La transition entre ces états est déterminée par des capteurs et des événements externes. Les retours d'expérience ont montré une meilleure maintenance du code et une facilité de débogage grâce à la clarté des transitions d'état.



## VI. Conclusion

- En conclusion, le pattern "État" offre une approche élégante pour gérer les états internes des objets dans la conception logicielle. En encapsulant les différents comportements dans des classes d'état distinctes, il favorise une maintenance aisée, une meilleure compréhension du code, et une adaptabilité accrue aux changements. En intégrant ce pattern, les développeurs peuvent concevoir des systèmes plus flexibles et robustes, améliorant ainsi la qualité globale du logiciel.

# SOURCES

## **Livres :**

- **"Design Patterns: Elements of Reusable Object-Oriented Software"**
- **Auteurs: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides**

**Ce livre, également connu sous le nom de "Gang of Four" (GoF), est une référence classique pour les design patterns. Il couvre le pattern "État" ainsi que d'autres.**

**"Head First Design Patterns"**

- **Sites Web :**

**Ce site offre une explication détaillée du pattern "État" avec des exemples de code.**

**[Google Scholar](#)**