Pattern State

Présenté par Léana RENON, Sylvain FREDIANI, Remi DE PELLEGRIN, Emma NGUYEN VAN GIAU

Sommaire

Les designs patterns

Brève histoire

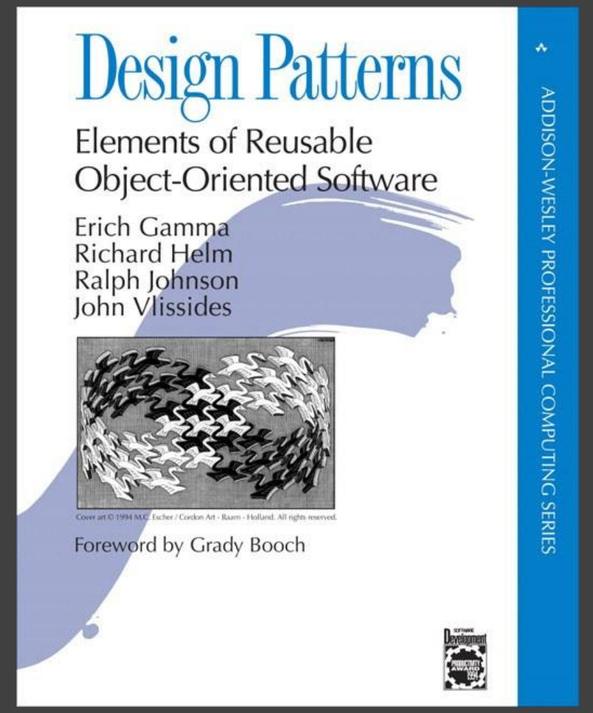
Le Gang of Four

- Début des années 90
- 3 types de design pattern :

Création

Structuration

Comportement



Énoncé du besoin

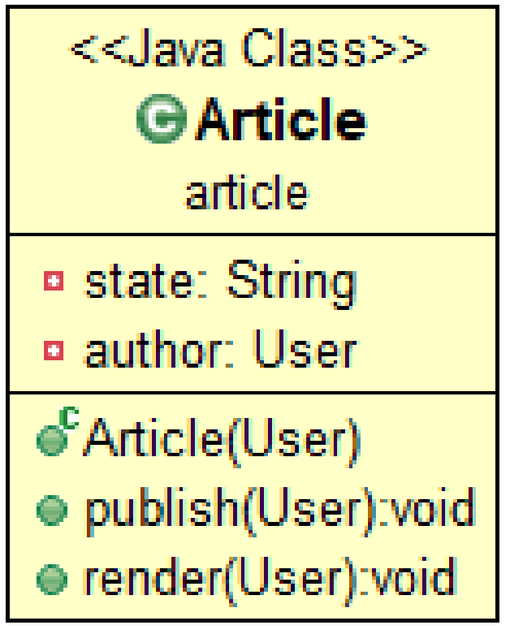
Exemple

Brouillon

En relecture

Publié

Première modélisation

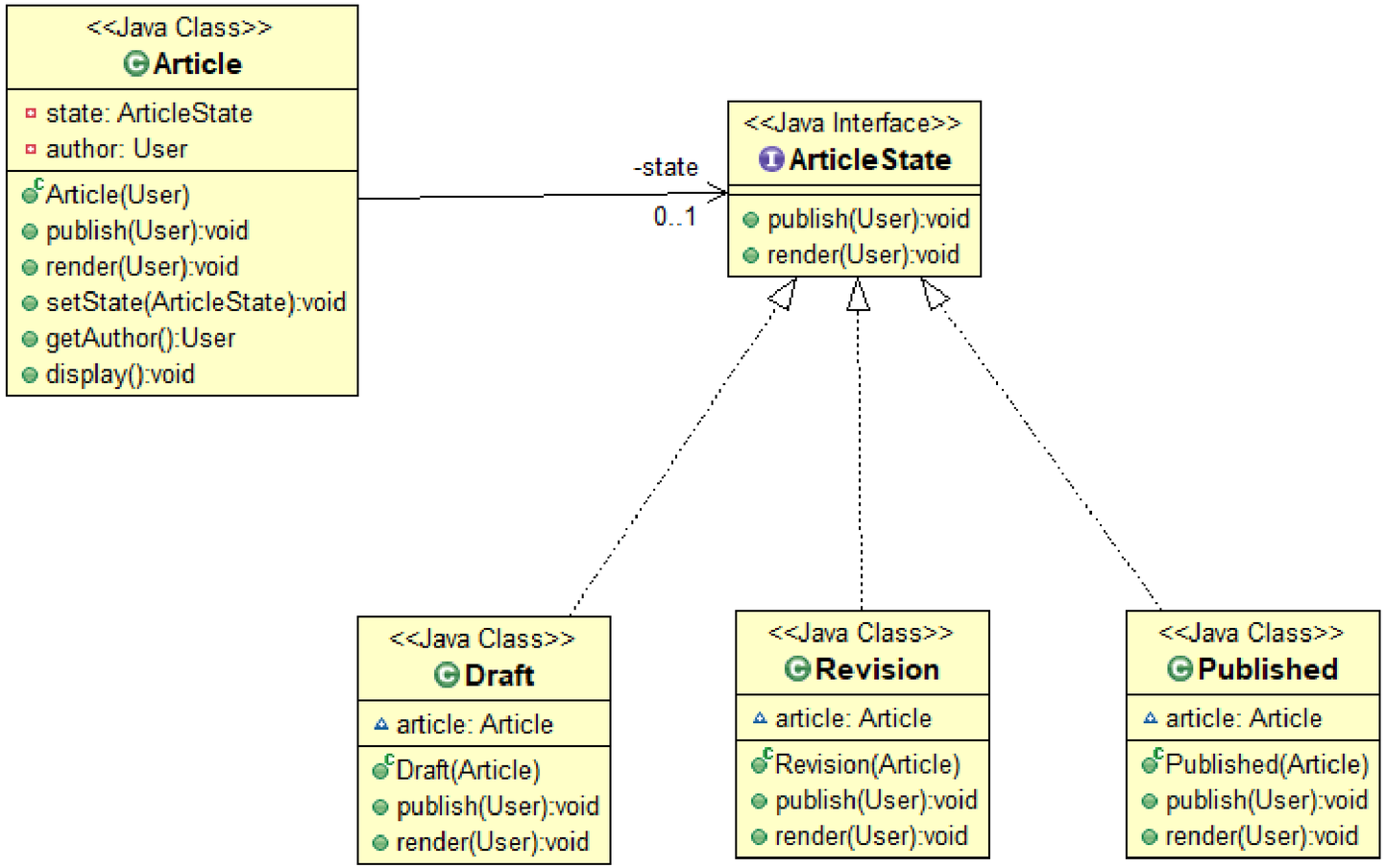


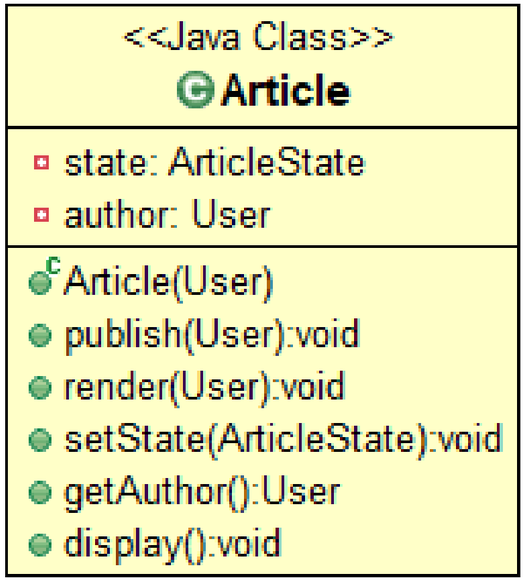

```java
public class Article {

    private String state;
    private User author;

    public Article(User author) {
        super();
        this.author = author;
        this.state = "draft";
    }

    public void publish(User user) {
        switch(state) {
        case "draft" :
            if (user.isModerator()) {
                state = "published";
            }else if(user.equals(author)) {
                state = "revision";
            }
            break;
        case "revision" :
            if (user.isModerator()) {
                state = "published";
            }
            break;
        case "published" :
            System.out.println("L'article est déjà publié");
            break;
        default :
            state = "published";
        }
    }
```

Problèmes

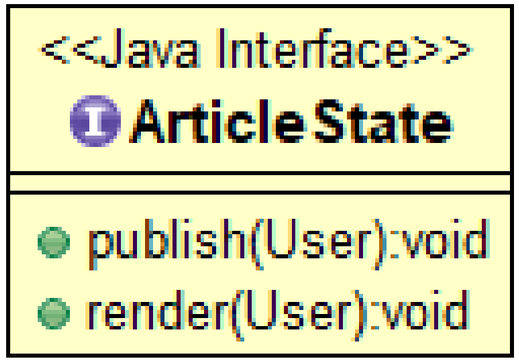
- Difficulté d'implémentation lors de l'ajout d'états ou de méthodes
- Multiplication des méthodes dans une seule classe
- Difficulté de relecture du code, ainsi que pour trouver les « bugs »

Solution

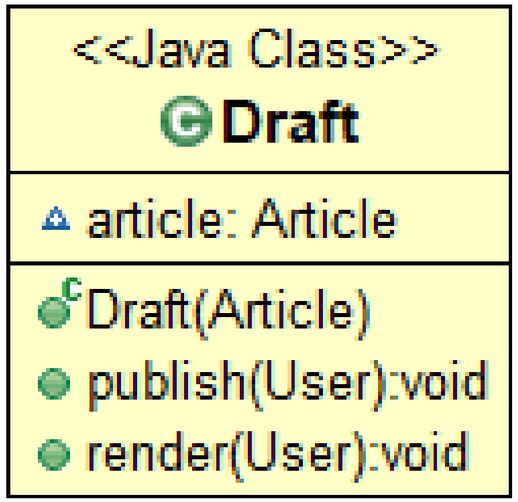

<<Java Class>>
Article
state: ArticleState
author: User
Article(User)
publish(User):void
render(User):void
setState(ArticleState):void
getAuthor():User
display():void
-state
0..1
<<Java Interface>>
ArticleState
publish(User):void
render(User):void
<<Java Class>>
Draft
article: Article
Draft(Article)
publish(User):void
render(User):void
<<Java Class>>
Revision
article: Article
Revision(Article)
publish(User):void
render(User):void
<<Java Class>>
Published
article: Article
Published(Article)
publish(User):void
render(User):void




```java
public class Article {

    private ArticleState state;
    private User author;

    public Article(User author) {
        super();
        this.author = author;
        this.state = new Draft(this);
    }

    public void publish(User user) {
        this.state.publish(user);
    }

    public void render(User user) {
        this.state.render(user);
    }

    public void setState(ArticleState state) {
        this.state = state;
    }
```




```java
public interface ArticleState {

    void publish(User user);

    void render(User user);

}
```




```java
public class Draft implements ArticleState {

    Article article;

    public Draft(Article article) {
        this.article = article;
    }

    @Override
    public void publish(User user) {
        if (user.isModerator()) {
            article.setState(new Revision(this.article));
        }else if(user.equals(article.getAuthor())) {
            article.setState(new Published(this.article));
        }
    }

    @Override
    public void render(User user) {
        if (user.isModerator() || user.equals(article.getAuthor())) {
            article.display();
        }else {
            System.out.println("Vous n'êtes pas autorisé à accèder à cet article");
        }
    }

}
```

Le pattern State

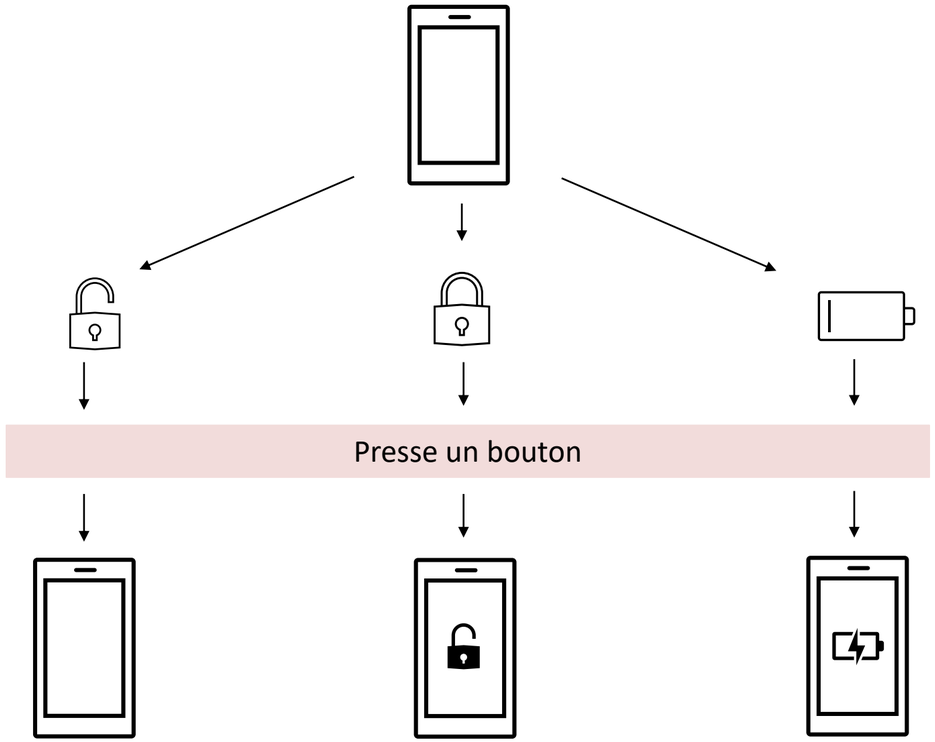
Pattern State

- Modèle de conception comportemental
- Permet à un objet de changer son comportement quand son état interne change

Pattern State

- Modèle de conception comportemental
- Permet à un objet de changer son comportement quand son état interne change

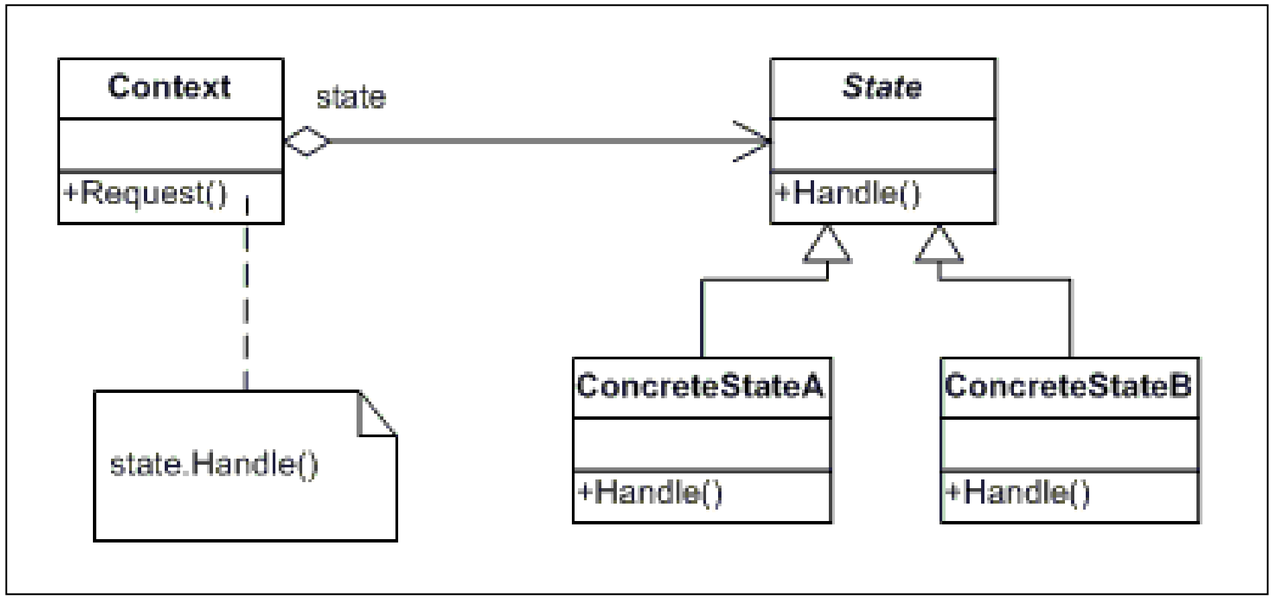
Problématique :



Solution du GoF

Le pattern State est codé à l'aide de 3 éléments principaux :

- la classe Context
- l'interface State
- les classes ConcreteState1, ConcreteState2, etc.



SOLID

- **SRP** une classe n'est responsable que d'une et une seule fonction
- **OCP** une classe est extensible mais pas modifiable
- **LSP** une classe parente peut être remplacée par une classe enfant
- **ISP** on préfère plusieurs interfaces spécifiques
- **DIP** utiliser des abstractions plutôt que des dépendances

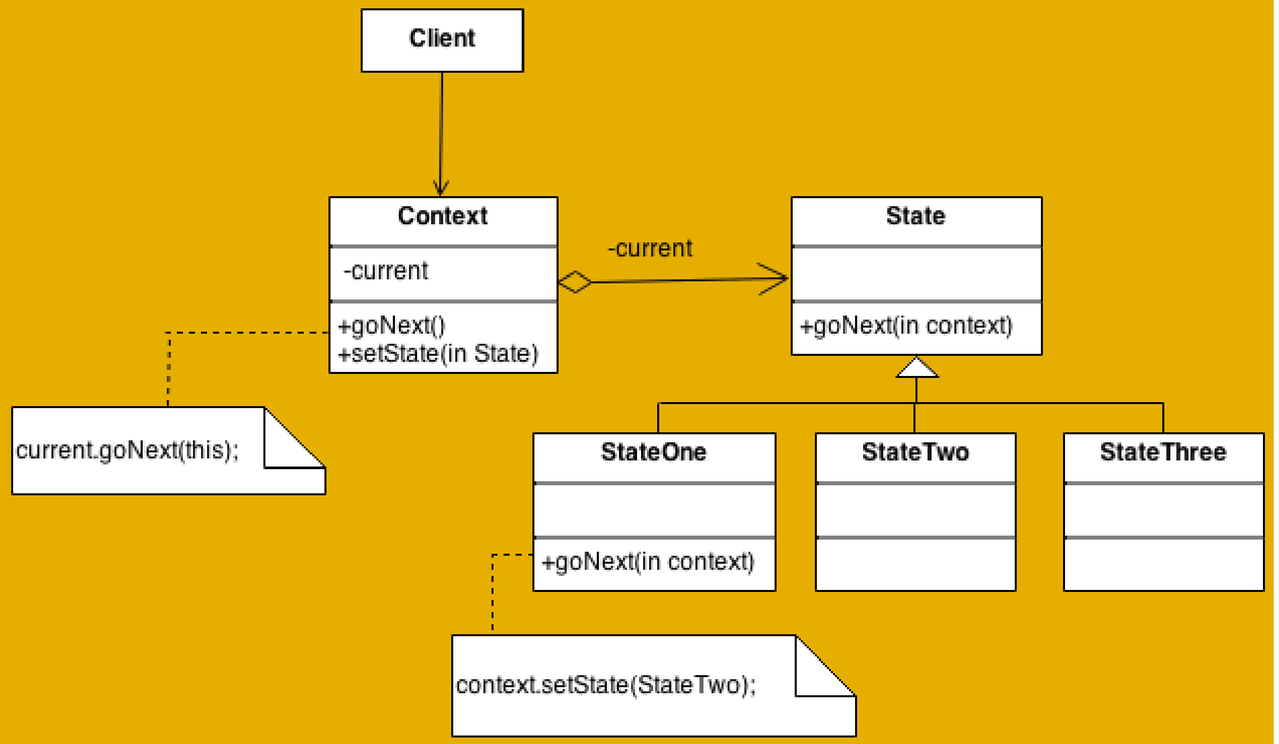
Limites

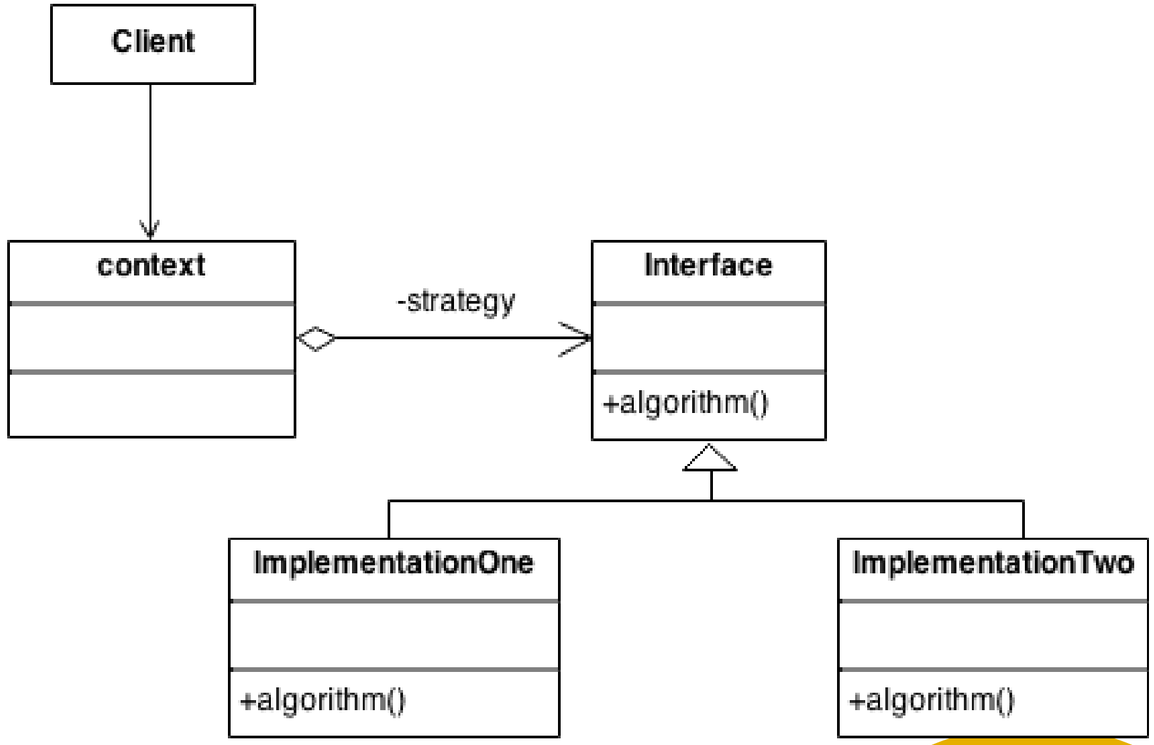
Plus il y a de classes, plus il y a de code et plus le nombre d'erreurs potentielles augmente !

State Vs Strategy

State Vs Strategy

Structure du State Pattern
(sourcemaking.com)





Structure du Strategy Pattern
(sourcemaking.com)

Plutôt similaires, n'est-ce pas… ?

Alors comment les différencier ?

State Pattern	Strategy Pattern
Plusieurs états, plusieurs comportements	Une tâche, plusieurs stratégies
Context indique currentState à State qui fournit ConcreteState	Context demande Implementation à Interface
ConcreteState peut influencer Context	Implementation n'influence pas Context
Liaison des ConcreteStates	Implementations indépendantes

State Vs Null Object

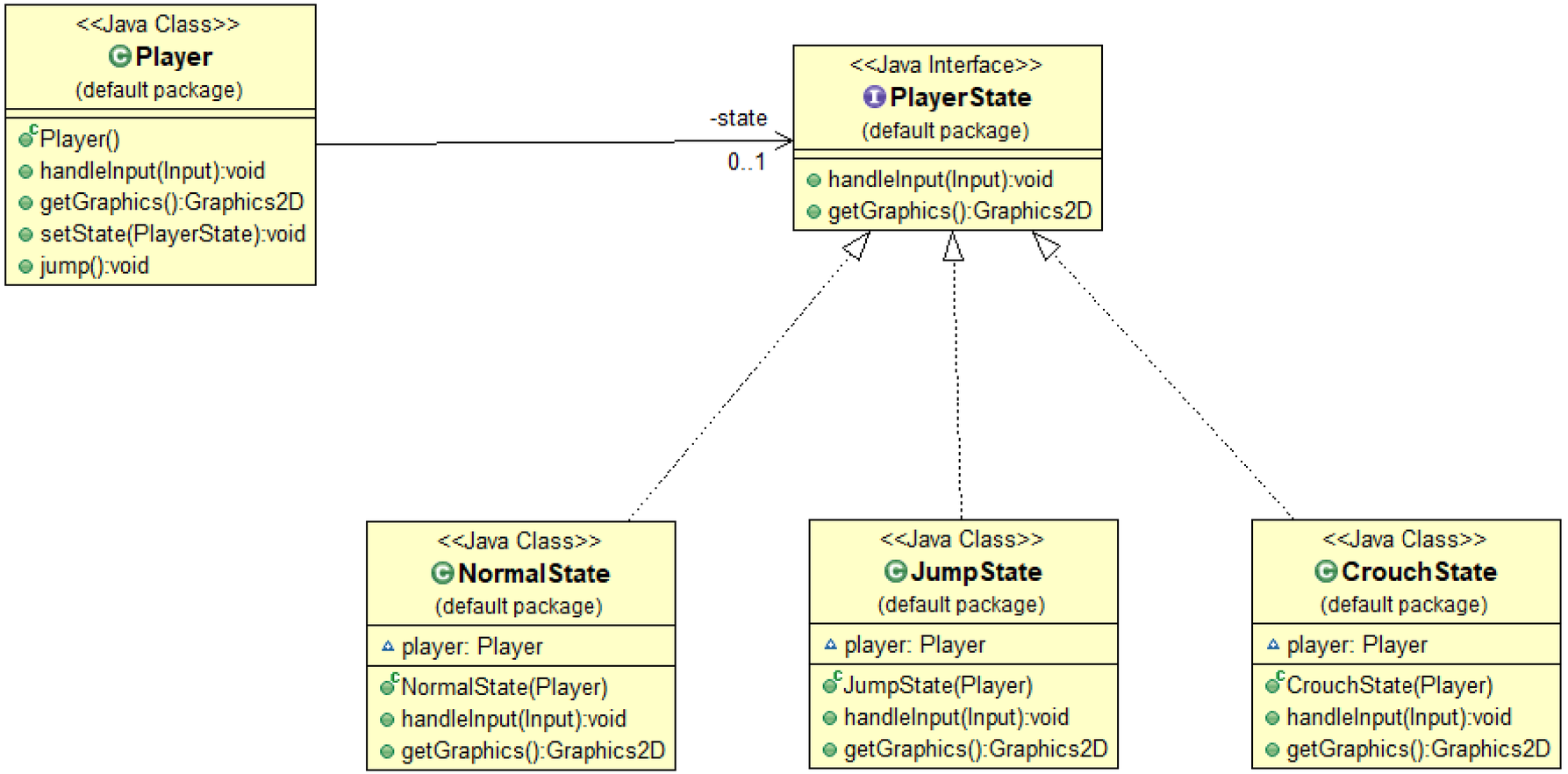
Nouveau Contexte

Nouveau Contexte : un perso de Jeu Vidéo

État normal

Saut

Accroupis


<<Java Class>>
Player
(default package)
Player()
handleInput(Input):void
getGraphics():Graphics2D
setState(PlayerState):void
jump():void
-state
0..1
<<Java Interface>>
PlayerState
(default package)
handleInput(Input):void
getGraphics():Graphics2D
<<Java Class>>
NormalState
(default package)
player: Player
NormalState(Player)
handleInput(Input):void
getGraphics():Graphics2D
<<Java Class>>
JumpState
(default package)
player: Player
JumpState(Player)
handleInput(Input):void
getGraphics():Graphics2D
<<Java Class>>
CrouchState
(default package)
player: Player
CrouchState(Player)
handleInput(Input):void
getGraphics():Graphics2D

Player

```
public class Player {

    private PlayerState state;

    public Player() {
        super();
        this.state = new NormalState(this);
    }

    public void handleInput(Input input) {
        this.state.handleInput(input);
    }

    public Graphics2D getGraphics() {
        return this.state.getGraphics();
    }

    public void setState(PlayerState state) {
        this.state = state;

    }
```

PlayerState

```
public interface PlayerState {

    public void handleInput(Input input);
    public Graphics2D getGraphics();
}
```

NormalState

```
public class NormalState implements PlayerState{

    Player player;

    public NormalState(Player player) {
        this.player = player;
    }

    @Override
    public void handleInput(Input input) {
        if(input == Input.SPACE_BAR) {
            player.jump();
            player.setState(new JumpState(this.player));
        }else if(input == Input.PRESS_DOWN) {
            player.setState(new CrouchState(this.player));
        }
    }

    @Override
    public Graphics2D getGraphics() {
        return null;
    }

}
```

CrouchState

```
public class CrouchState implements PlayerState {

    Player player;

    public CrouchState(Player player) {
        this.player = player;
    }

    @Override
    public void handleInput(Input input) {
        if(input == Input.RELEASE_DOWN) {
            this.player.setState(new NormalState(player));
        }
    }

    @Override
    public Graphics2D getGraphics() {
        return null;
    }

}
```

CONCLUSION

QCM

https://cutt.ly/2hlt5EL

Merci !

Sitographie

iluwatar. (s. d.-a). *Null Object*. Java Design Patterns. https://java-design-patterns.com/patterns/null-object/

iluwatar. (s. d.-b). *State*. Java Design Patterns. https://java-design-patterns.com/patterns/state/

Robert Nystrom, R. N. (s. d.). *State · Design Patterns Revisited · Game Programming Patterns*. Game Programming Patterns. https://gameprogrammingpatterns.com/state.html

Source Making. (s. d.-a). *Null Object Design Pattern*. https://sourcemaking.com/design_patterns/null_object

Source Making. (s. d.-b). *State Design Pattern*. https://sourcemaking.com/design_patterns/state

Source Making. (s. d.-c). *Strategy Design Pattern*. https://sourcemaking.com/design_patterns/strategy