

LOG725 - Ingénierie et conception de jeux vidéo

Labo 2 - Programmation et patrons pour les jeux vidéo

Gabriel C. Ullmann

École de Technologie Supérieure, Hiver 2024



Le génie pour l'industrie

Objectifs d'apprentissage

- Identifier les patrons de conception
- Identifier les *code smells*
- Implémenter le patron Command
- Implémenter le patron Observer
- Comprendre l'utilisation des patrons dans le contexte des jeux vidéo

Activités

Discussion

Trouver les patrons de conception
+ code smells dans un jeu

Activités

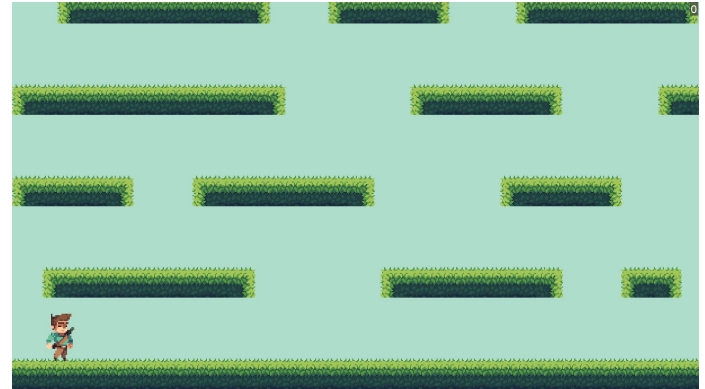
Implémenter les patrons
Command et Observer

Cas d'étude : Jungle Climb



Nous commencerons par analyser le jeu Jungle Climb, créé par Elijah Lopez. Actions disponibles:

- Se déplacer (touches K_LEFT et K_RIGHT)
- Sauter (touche K_UP)
- Mettre le jeu en pause (touche K_ESCAPE)
- Quitter le jeu (touches ALT + F4)



Brise-glace: patrons de conception

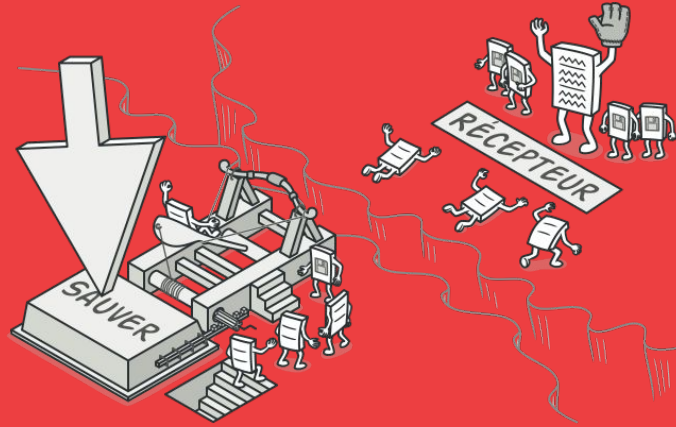
- Est-ce que vous pouvez identifier les **patrons** déjà implémentés dans le code ?

Par exemple:

- Update
- Game loop
- State
- Est-ce que pouvez-vous identifier **d'autres patrons**?
- **À quoi servent-ils** dans le contexte de Jungle Climb ?
- Sont-ils **bien** ou **mal** implémentés ?

Brise-glace: Identifier les *code smells*

- Est-ce que vous pouvez identifier les *code smells* dans le code ?
 - Méthode trop longue
 - Liste de paramètres trop longue
 - Code 'mort'
 - Code commenté
 - Code dupliqué
 - Commentaires non clairs
- Comment pouvons-nous **régler** ces problèmes dans le code ?
- Et comment pouvons-nous les **éviter** dès le départ?
- Nos references: [Game Programming Patterns](#) et [Refactoring Guru](#)



Le patron Command

Command : un cas d'utilisation

La logique de vérification des entrées est implémentée à partir de la ligne 504 du fichier `climber_game.py`.

```
for event in pygame.event.get():
    pressed_keys = pygame.key.get_pressed()
    alt_f4 = (event.type == KEYDOWN and event.key == K_F4
              and (pressed_keys[K_LALT] or pressed_keys[K_RALT]))
    if event.type == QUIT or alt_f4:
        sys.exit()
    if event.type == KEYDOWN and action == -1:
        if event.key == K_RIGHT:
            self.player.go_right()
        elif event.key == K_LEFT:
            self.player.go_left()
```


Command : un cas d'utilisation

```
if event.type == KEYDOWN and action == -1:
    if event.key == K_RIGHT:
        self.player.go_right()
    elif event.key == K_LEFT:
        self.player.go_left()
```

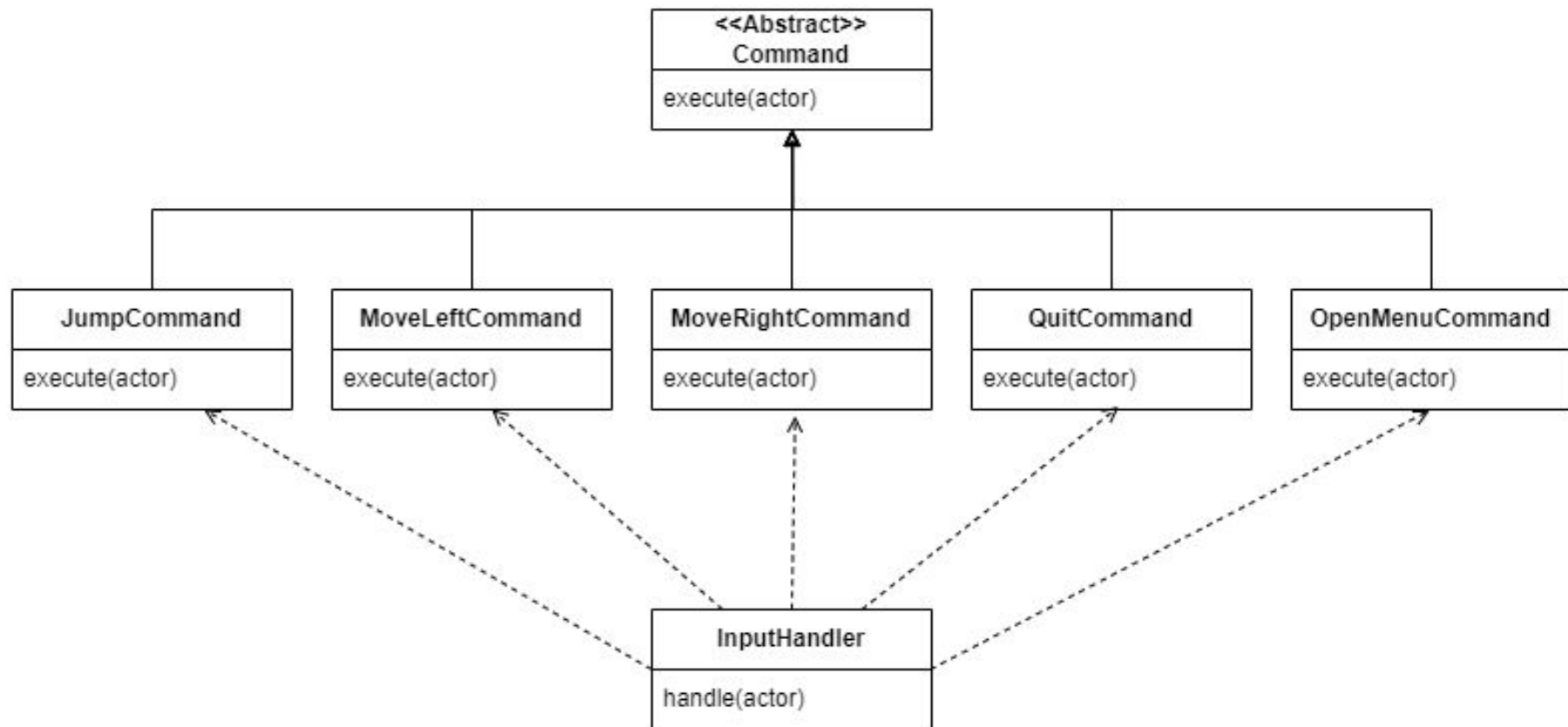
Cette structure présente plusieurs problèmes :

- Nous ne pouvons pas facilement changer la relation entre une touche et une action (remapping)
- Nous ne pouvons pas facilement utiliser la même commande pour plusieurs touches ou plusieurs objets du jeu

Exercice 1 : Créer des classes

Command et Input Handler

1. Ouvrez `climber_game.py` et **extrayez** le code de chaque action - sauter, se déplacer, pauser et quitter.
2. **Placez** le code extrait dans la classe correspondante dans le répertoire "patrons".
3. Complétez le code dans `patrons/input_handler.py`. **Conseil** : le diagramme UML peut vous aider à comprendre comment les différentes parties du code travaillent ensemble.
4. **Importez** et **utilisez** la classe `InputHandler` pour gérer les entrées dans `climber_game.py`.

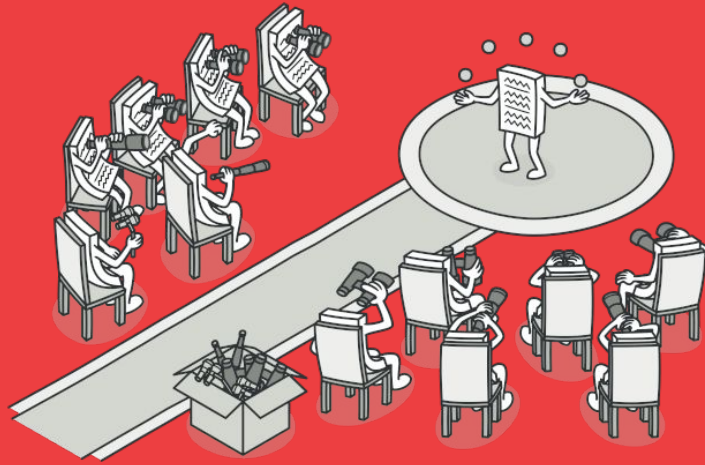


Observer : un cas d'utilisation

Nous pouvons imaginer un jeu vidéo comme une machine à états (FSM).

Les changements d'état d'un acteur peuvent déclencher des changements dans plusieurs autres parties de notre jeu. Par exemple, dans un jeu plateforme:

```
if player.not_on_ground:
    physics.gravity(player)
    sound.play("fall")
    player.set_animation("falling")
    enemy.ai.change_behavior("fall")
```



Le patron Observer

Le patron Observer

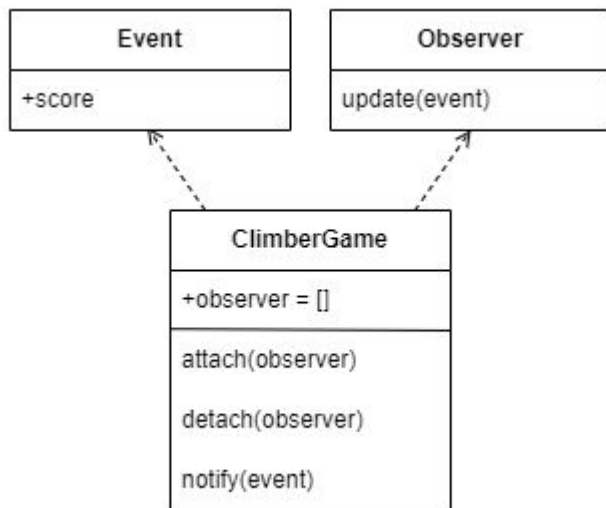
```
if player.not_on_ground:
    physics.gravity(player)
    sound.play("fall")
    player.set_animation("falling")
    enemy.ai.change_behavior("fall")
```

Cette structure présente plusieurs problèmes :

- Le **déclenchement de l'événement** est couplé avec ses **effets**. Ceux-ci doivent être séparés dans les différents fichiers pour faciliter la maintenance.
- Cette structure a plusieurs dépendances: la physique, le son, la IA, etc.
- Tester cette structure est difficile, car tout est dans le même fichier, et il faut trouver les bonnes conditions pour « accéder » à une partie du code.

Exercice 2 : Implémenter le patron Observer

1. **Implémentez** les classes **Observer** et **Event**. **Conseil** : suivez les informations données par le diagramme UML. Il décrit toutes les méthodes et attributs nécessaires.
2. Ouvrez `climber_game.py` et **implémentez les méthodes** *attach* et *detach* dans la classe `ClimberGame`.
3. Lorsque le joueur atteint les 1000 points, faites `print("achievement unlocked")`.



Conclusion

- Patrons de conception et *code smells*
- **Command** : gestion d'entrées
- **Observer** : gestion d'événements
- Les techniques de génie logiciel 'traditionnel' sont également **utiles** dans le métier du jeu vidéo

LOG725 - Ingénierie et conception de jeux vidéo

Labo 2 - Programmation et patrons pour les jeux vidéo

Gabriel C. Ullmann

École de Technologie Supérieure, Hiver 2024



Le génie pour l'industrie