
Patrones de Diseño en Videojuegos
Factory y Builder

Introducción

¿De Qué vamos a hablar?

- Patrones de Diseño: Concepto y Beneficios
- Patrón Factory: Definición, Uso y Ejemplos en Videojuegos
- Patrón Builder: Definición, Uso y Ejemplos en Videojuegos
- Comparación entre Factory y Builder
- Conclusiones y Referencias

Patrones de Diseño en Videojuegos

Los patrones de diseño son soluciones reutilizables a problemas comunes en el desarrollo de software. En el contexto de los videojuegos, estos patrones ayudan a estructurar el código de manera eficiente, facilitando la creación, mantenimiento y escalabilidad del juego.

En el ámbito de los videojuegos, los patrones de diseño son especialmente útiles para manejar la complejidad inherente a los sistemas de juego, como la gestión de entidades, la lógica de juego, la interacción del usuario y la renderización gráfica.

En esta presentación, nos centraremos en dos patrones de diseño fundamentales: el patrón Factory y el patrón Builder, explorando sus definiciones, usos y ejemplos prácticos en el desarrollo de videojuegos.

Tipos de Patrones de Diseño

Si bien no hay un número fijo de patrones de diseño, algunos de los más comunes incluyen:

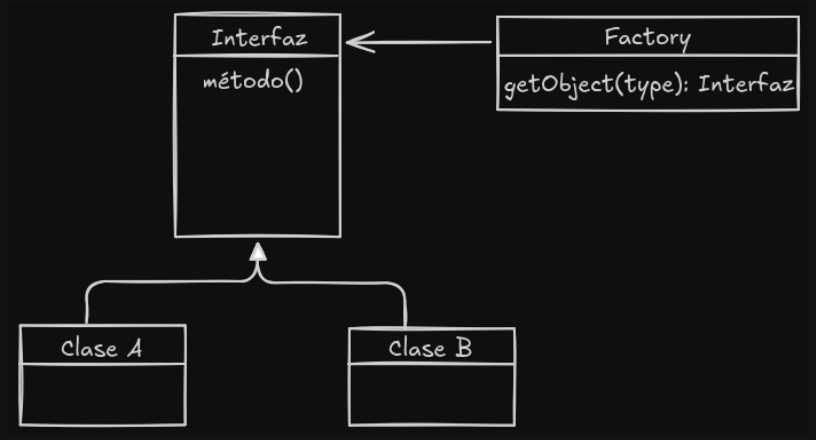
- Singleton
- Factory
- Builder
- Observer
- Strategy
- Decorator
- Command
- Adapter
- ECS (Entity-Component-System)
- etc...

En esta presentación, nos centraremos en los patrones Factory y Builder, explorando sus definiciones, usos y ejemplos prácticos en el desarrollo de videojuegos.

Patrón Factory

Patrón de diseño creacional que proporciona una interfaz para crear objetos en una superclase, pero permite que las subclases alteren el tipo de objetos que se crearán.

Es importante porque promueve la flexibilidad y la reutilización del código, permitiendo que el sistema sea más fácil de mantener y extender. Se basa principalmente en el principio de "programar para una interfaz, no para una implementación".



Ejemplo de Patrón Factory en Videojuegos

Vamos a ver un ejemplo sencillo de cómo implementar el patrón Factory en Python para crear diferentes tipos de enemigos en un videojuego.

```python
class Enemy:
    def attack(self):
        pass
class Goblin(Enemy):
    def attack(self):
        return "Goblin attacks with a club!"
class Troll(Enemy):
    def attack(self):
        return "Troll attacks with a hammer!"
class EnemyFactory:
    @staticmethod
    def create_enemy(enemy_type):
        if enemy_type == "goblin":
            return Goblin()
        elif enemy_type == "troll":
            return Troll()
        else:
            raise ValueError("Unknown enemy type")
```

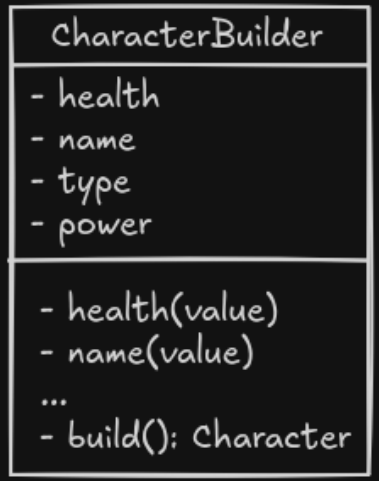
Ejemplo de Patrón Factory en Videojuegos (Continuación)

```
# Uso del Factory para crear enemigos
goblin = EnemyFactory.create_enemy("goblin")
troll = EnemyFactory.create_enemy("troll")
print(goblin.attack())  # Output: Goblin attacks with a club!
print(troll.attack())   # Output: Troll attacks with a hammer!
```

Patrón Builder

Patrón de diseño creacional que separa la construcción de un objeto complejo de su representación, permitiendo que el mismo proceso de construcción pueda crear diferentes representaciones.

Es importante porque facilita la creación de objetos complejos paso a paso, promoviendo la claridad y la flexibilidad en el código. Se basa en el principio de "separar la construcción de un objeto de su representación".



Ejemplo de Patrón Builder en Videojuegos

Vamos a ver un ejemplo sencillo de cómo implementar el patrón Builder en Python para crear diferentes tipos de personajes en un videojuego.

```
class Character:
    def __init__(self):
        self.name = ""
        self.armor = ""
        self.weapon = ""
    def __str__(self):
        return f"Character: {self.name}, Armor: {self.armor}, Weapon: {self.weapon}"
class CharacterBuilder:
    def __init__(self):
        self.character = Character()
    def set_name(self, name):
        self.character.name = name
        return self
    def set_armor(self, armor):
        self.character.armor = armor
        return self
    def set_weapon(self, weapon):
        self.character.weapon = weapon
        return self
    def build(self):
        return self.character
```

Ejemplo de Patrón Builder en Videojuegos (Continuación)

```
# Uso del Builder para crear personajes
warrior = CharacterBuilder()
          .set_name("Warrior")
          .set_armor("Plate")
          .set_weapon("Sword")
          .build()
mage = CharacterBuilder()
          .set_name("Mage")
          .set_armor("Robe")
          .set_weapon("Staff")
          .build()
print(warrior)  # Output: Character: Warrior, Armor: Plate, Weapon: Sword
print(mage)     # Output: Character: Mage, Armor: Robe, Weapon: Staff
```

Comparación entre Factory y Builder

Característica	Patrón Factory	Patrón Builder
Propósito	Crear objetos sin especificar la clase exacta	Construir objetos complejos paso a paso
Uso típico	Cuando el sistema debe ser independiente de cómo se crean los objetos	Cuando se necesita crear objetos complejos con múltiples configuraciones
Complejidad del objeto	Generalmente crea objetos simples	Crea objetos complejos con múltiples atributos
Flexibilidad	Menos flexible en términos de configuración del objeto	Más flexible, permite configuraciones detalladas

Conclusiones

- Los patrones de diseño son herramientas esenciales para estructurar el código de manera eficiente en el desarrollo de videojuegos.
- El patrón Factory es ideal para crear objetos sin especificar la clase exacta, promoviendo la flexibilidad y reutilización del código.
- El patrón Builder es útil para construir objetos complejos paso a paso, facilitando la claridad y flexibilidad en la creación de objetos.
- La elección entre Factory y Builder depende de la complejidad del objeto a crear y los requisitos específicos del proyecto.
- Ambos patrones contribuyen a un código más limpio, mantenible y escalable en el desarrollo de videojuegos.

Referencias

- Desarrollo Homebrew para 16 bits - V. Suárez
- Design Patterns: Elements of Reusable Object-Oriented Software
- Documentación de Python: https://docs.python.org/3/