Patrón Abstract Factory

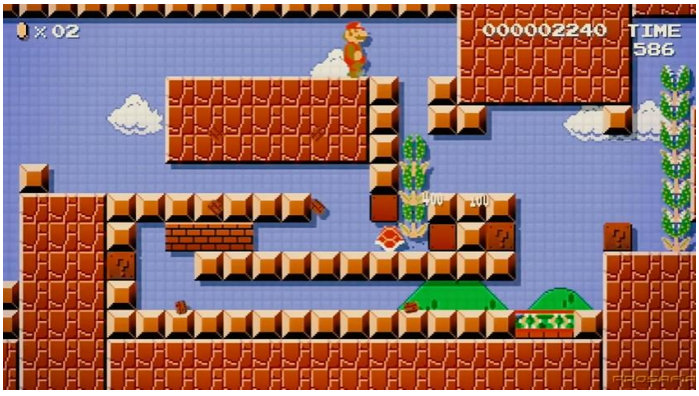
Ejemplo: un video juego, Super Mario Maker

https://www.youtube.com/watch?v=CVlpjFJN17U&list=PLJkcleqxxobUJlz1Cm8WYd-F_kckkDvc8&index=3

Para este patrón vamos a ver que tiene mucha relación con patrón Factory, de ahí que los nombres se parezcan.

Hablando del juego

Para este caso de uso vamos a ver lo siguiente: En Super Mario Maker puedes montar tus propios niveles para diferentes consolas (como GameBoy o NintendoDS), colocando las típicas monedas, bloques, enemigos, tuberías, etc., es decir te permiten hacer y modificar distintos tipo de gráficas. El jugador podrá escoger el estilo que más le guste.



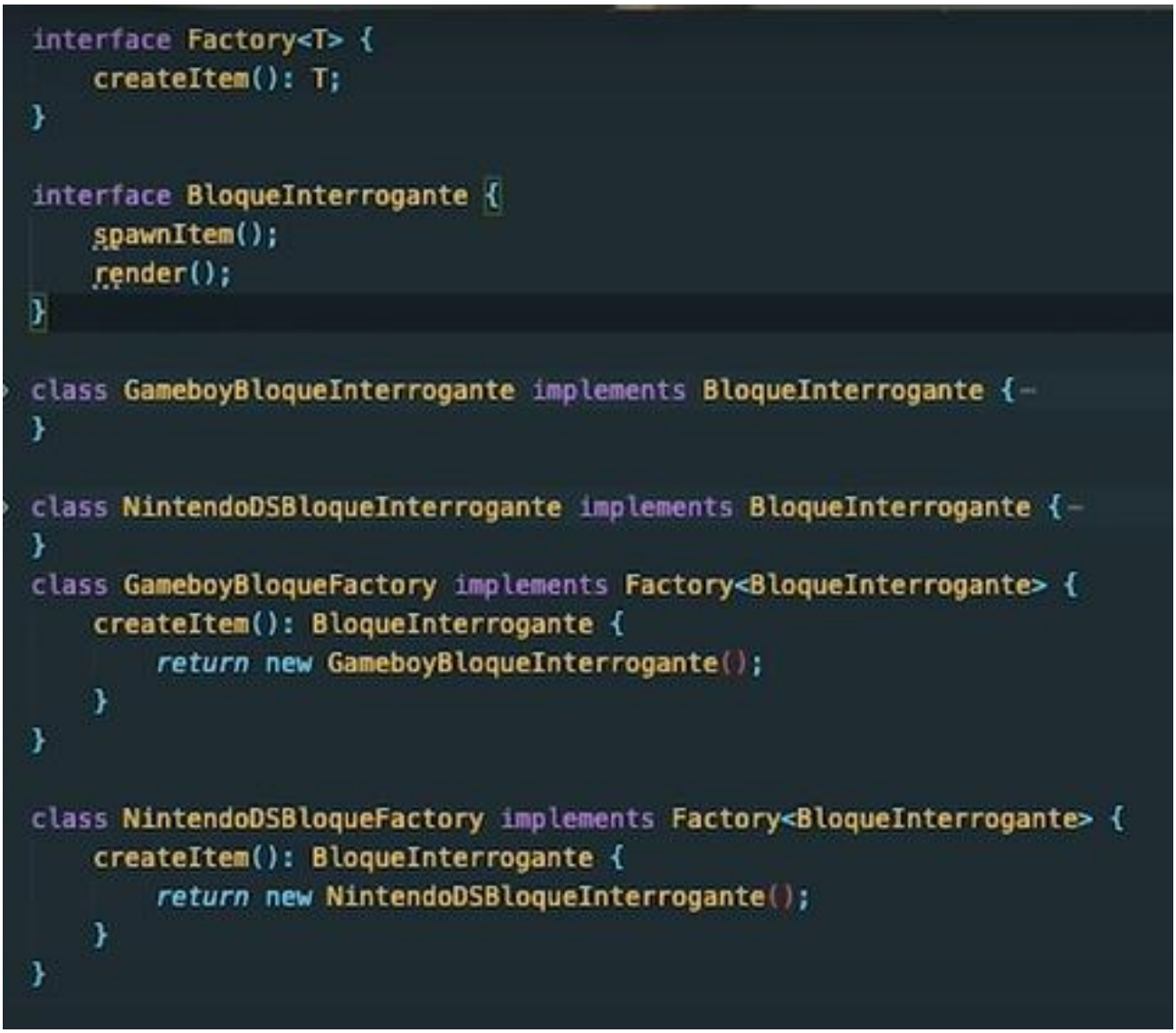
¿Pero cómo se hace esto?

Precisamente usando la factoría abstracta

Problemas que nos podemos encontrar

¿Qué pasaría si intentamos hacer esto mismo pero con el patrón Factory?

Supongamos que tenemos una Factory que nos instancia objetos de la interface **BloqueInterrogante**, según el estilo del nivel nos puede instanciar la clase concreta **GameBoyBloqueInterrogante** o **NintendoDSBloqueInterrogante**



```
interface Factory<T> {
    createItem(): T;
}

interface BloqueInterrogante {
    spawnItem();
    render();
}

class GameboyBloqueInterrogante implements BloqueInterrogante {
}

class NintendoDSBloqueInterrogante implements BloqueInterrogante {
}
class GameboyBloqueFactory implements Factory<BloqueInterrogante> {
    createItem(): BloqueInterrogante {
        return new GameboyBloqueInterrogante();
    }
}

class NintendoDSBloqueFactory implements Factory<BloqueInterrogante> {
    createItem(): BloqueInterrogante {
        return new NintendoDSBloqueInterrogante();
    }
}
```

Hasta aquí no tenemos ningún problema, ya que por ahora son los únicos que cambian según el tipo de nivel.

Ahora bien, si queremos que las monedas cambien de estilo, debemos construir otra Factory, es decir, generar la interfaz **Moneda**, y sus respectivas clases concretas **GameBoyMoneda** y **NintendoDSMoneda**

¿Ven por dónde vamos?

Si seguimos por este camino, vamos a tener una Factory por cada elemento del juego que se vea afectado, por lo que nos va a ocurrir lo siguiente:

- Vamos a tener cada vez más factorías por lo que tendríamos que gestionar más y más clases.
- Al ofrecer todas las factorías de forma individual, no hay sistema de control sobre que ítem se utilizan con cuales. Nada evitaría que utilicemos por ejemplo un enemigo 2D en una estilo 3D

Para evitar esto es cuando entra la factoría abstracta que se define como

“Patrón de diseño patrón de diseño creacional que nos permite producir familias de objetos relacionados sin especificar sus clases concretas.”

Ventajas

- En el ejemplo vemos que los objetos están relacionados por temas (familias), en este caso por la visual o tema.
- En este patrón siempre trabajas con las clases abstractas, nunca se depende de las clases concretas, es decir siempre usamos la abstracción Moneda, nos es indiferente que moneda es.
- Con esto conseguimos trabajar en forma independiente a las implementaciones concretas de cada clase
- Nos permite crear relaciones entre objetos de forma que no se puedan utilizar de forma indiscriminada
- Te protege de utilizar sin querer objetos que no deben utilizarse juntos.
- Desde afuera nunca se sabe con qué clases concretas estás trabajando
- Se aplica polimorfismo en su máxima expresión

Implementación

En vez de tener una factoría con un método de crear cierto objeto, tenemos una factoría abstracta que tiene varios métodos que crean diferentes tipos de objetos.

En el ejemplo, la factoría abstracta tendría definido el método de crear el método BloqueInterrogante y Moneda.

Seria responsabilidad de la implementación concreta de esta factoría abstracta implementar estos métodos y crear instancias concretas de estos objetos.

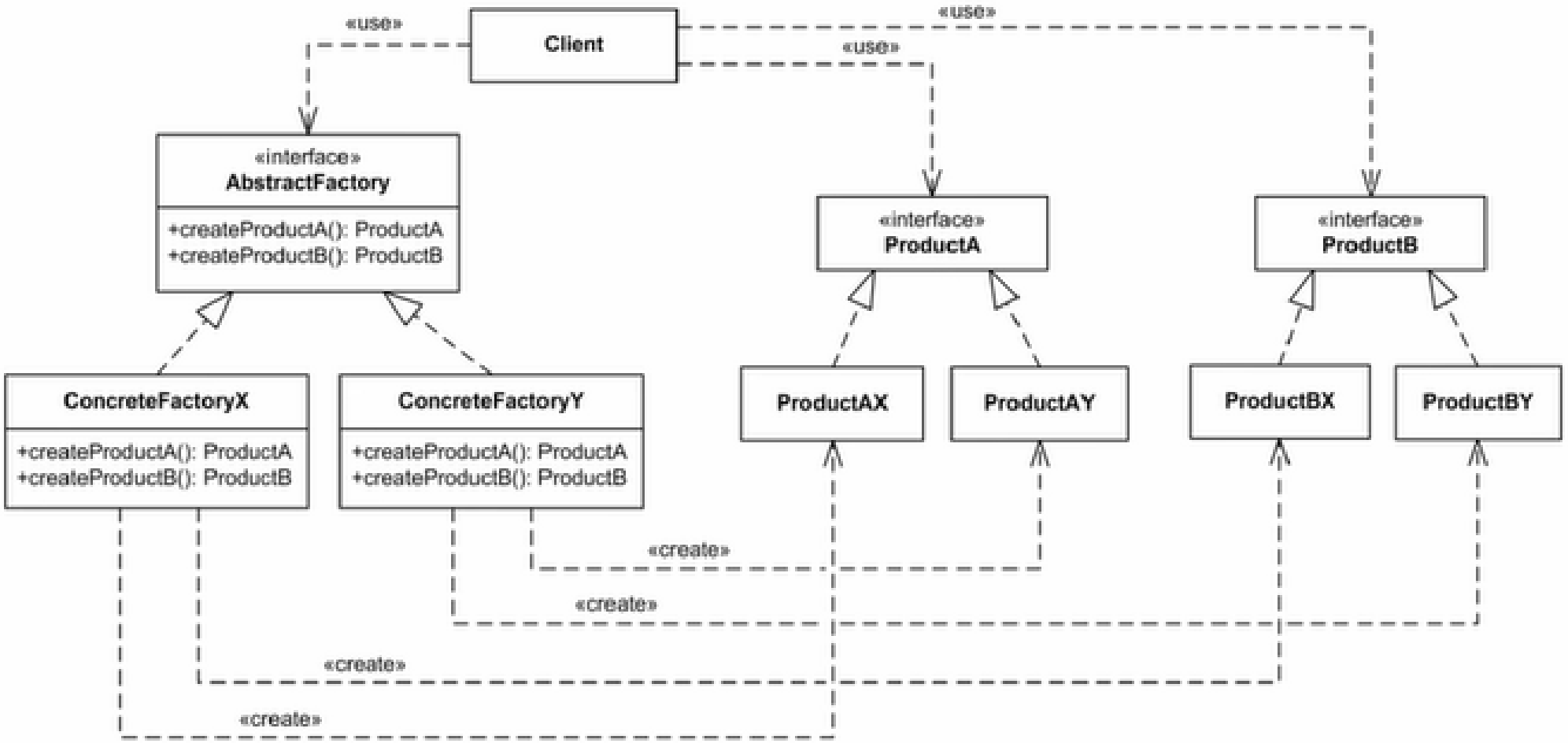
Por ejemplo tendríamos la factoría abstracta **GameBoyItemFactory** que instancia siempre las clases concretas que implementan los gráficos de Game Boy como así también la factoría abstractata **NintendoDSItemFactory** que implementan estos mismos métodos pero instancian los métodos las clases concretas de Nintendo

```
import {
    Moneda,
    BloqueInterrogante,
    GameBoyMoneda,
    GameboyBloqueInterrogante,
    NintendoDSMoneda,
    NintendoDSBloqueInterrogante
} from './abstract_factory';

export interface AbstractFactory {
    createMoneda(): Moneda;
    createBloqueInterrogante(): BloqueInterrogante;
}

export class GameBoyItemFactory implements AbstractFactory {
    createMoneda(): Moneda {
        return new GameBoyMoneda();
    }
    createBloqueInterrogante(): BloqueInterrogante {
        return new GameboyBloqueInterrogante();
    }
}

export class NintendoDSItemFactory implements AbstractFactory {
    createMoneda(): Moneda {
        return new NintendoDSMoneda();
    }
    createBloqueInterrogante(): BloqueInterrogante {
        return new NintendoDSBloqueInterrogante();
    }
}
```


«use»
Client
«use»
«use»
«interface»
AbstractFactory
+createProductA(): ProductA
+createProductB(): ProductB
«interface»
ProductA
«interface»
ProductB
ConcreteFactoryX
+createProductA(): ProductA
+createProductB(): ProductB
ConcreteFactoryY
+createProductA(): ProductA
+createProductB(): ProductB
ProductAX
ProductAY
ProductBX
ProductBY
«create»
«create»
«create»
«create»