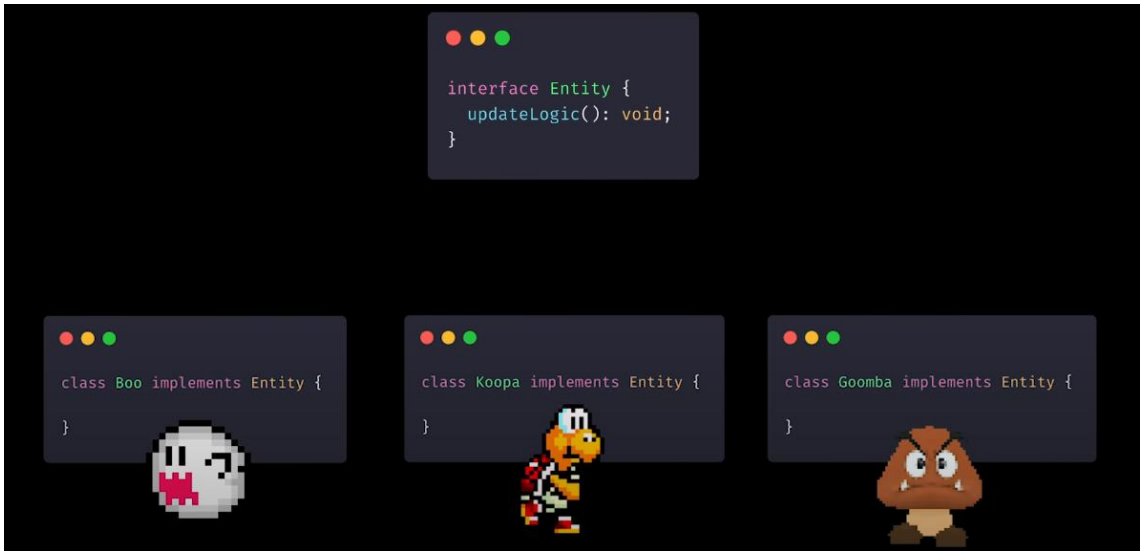
Patrón Factory

Ejemplo: un video juego

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

Los enemigos: son una parte importante del video juego. Para ello vamos a tener diferentes tipos de enemigos y dificultades según se vaya avanzando en el juego para sorprender al jugador

Estas clases “enemigos” son entidades lógicas del juego, es decir todas hacen algo como interactuar con el jugador, con el mapa con otros enemigos por lo tanto debemos definir una clase “entidad”



Esto nos permite aplicar POLOFORMISMO, por lo que podemos tratar a todas las entidades por igual cada vez que actualicemos el estado. Queremos que el jugador no memorice jugadas o que los diferentes enemigos te sorprendan, por lo que hay que generarlos de forma aleatoria. En algún punto del código debemos tener una función para crear nuestros enemigos

Y no podemos instanciar new Koopa, o new Goomba, porque no es una clase concreta, entonces usamos esta función nos permite generar enemigos de forma aleatoria. ¿Pero cómo los construimos, como decidimos cuál enemigo?

A través de una CLASE ABSTRACTA

```
function gameLogic() {
   ...
   //More code above
   if (shouldSpawnEnemy()) {
     let randomNum = Math.random();
     let enemy;
     if (randomNum > 0.66) {
         enemy = new Koopa();
     } else if (randomNum > 0.33) {
         enemy = new Goomba();
     } else {
         enemy = new Boo();
     }
   }
   ...
   //More code below, use enemy
}
```

Pero aquí nos encontramos que la dificultad siempre es la misma ya que las probabilidades siempre son las mismas, ¿Pero cómo mejoramos esto? ¿Cómo lo hacemos sin afectar a todos los jugadores que ya están disfrutando el juego?

```
function gameLogic() {
   ...
   //More code above
   if (shouldSpawnEnemy()) {
     let randomNum = Math.random();
     let enemy;
     if (randomNum > 0.66) {
         enemy = new Koopa();
     } else if (randomNum > 0.33) {
         enemy = new Goomba();
     } else {
         enemy = new Boo();
     }
   }
   ...
   //More code below, use enemy
}
```

Debemos encapsular la lógica de crear los enemigos, es decir, encapsular las estrategias, y para ello vamos a emplear el patrón FACTORY, la responsable de construir dicha lógica para construir enemigos (clase que podremos reutilizar y usarla de forma polimórfica).

Un ejemplo a aplicar sería tener dos clases Factory que podremos llamar:

```
class RandomEnemyFactory implements EnemyFactory {
  createEnemy(): Entity {
    //returns enemies randomly
  }
}
```

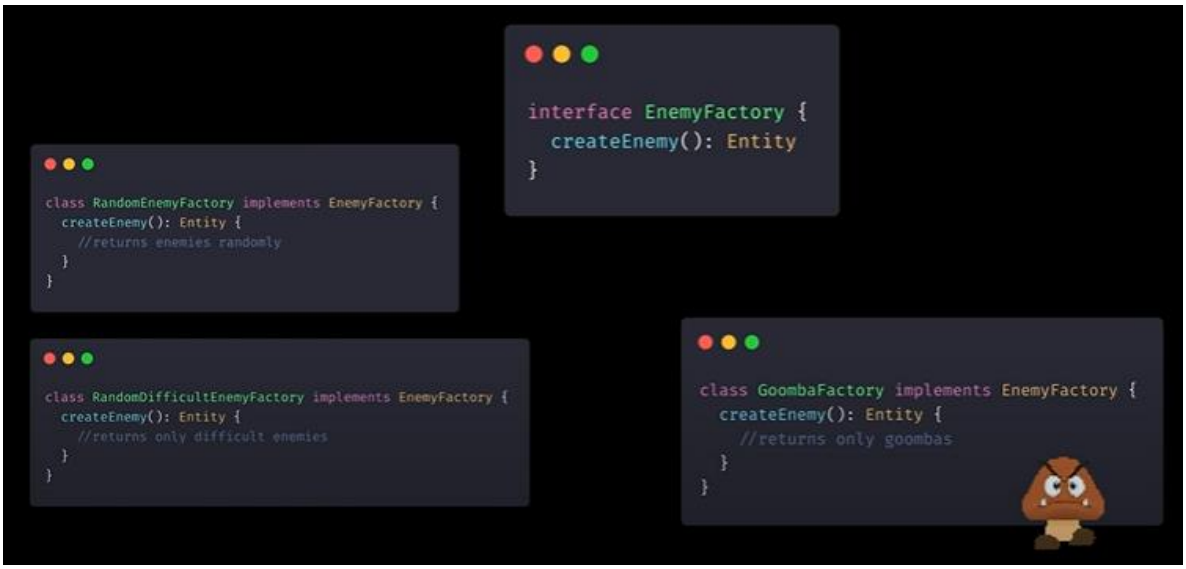
```
class RandomDifficultEnemyFactory implements EnemyFactory {
  createEnemy(): Entity {
    //returns only difficult enemies
  }
}
```

La primera clase nos devuelve enemigos de forma aleatoria y la segunda nos devuelve enemigos solamente poderosos

Todas estas clases siguen a la misma interfaz, porque todas heredan de una abstracción que define el método abstracción de enemigos a la que podemos llamar CreateEnemy(), pero

¿Podríamos hacer más factories?

Claro que sí, porque heredan de la misma abstracción por ejemplo



Esto es muy importante porque cada vez que necesitemos crear enemigos vamos a usar la clase padre a través del polimorfismo, por lo que podremos modificar las implementaciones concretas cuando querramos.

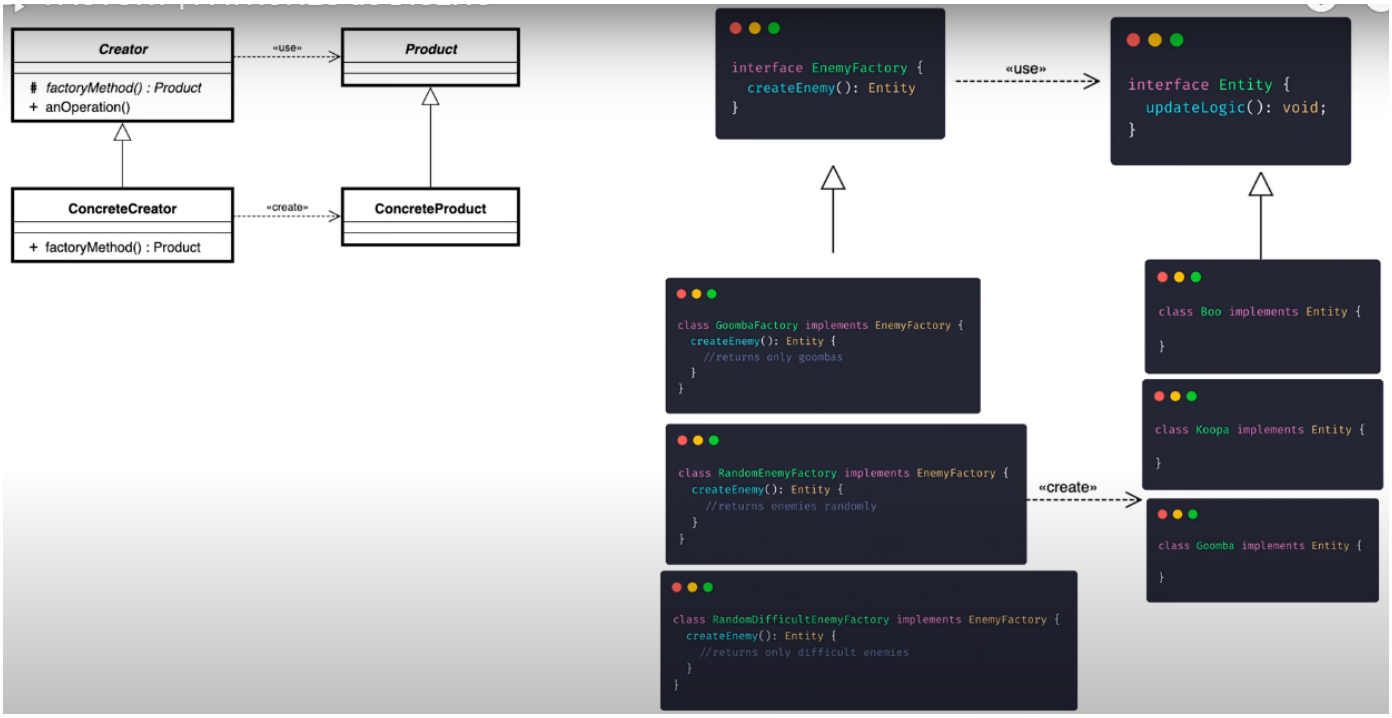
```
class Game {
  private enemyFactory: EnemyFactory;

  constructor(enemyFactory: EnemyFactory) { //Dat DI
   this.enemyFactory = enemyFactory;
  }

  function gameLogic() {
    ...
    //More code above
    if (shouldSpawnEnemy()) {
      // this.enemyFactory is of type EnemyFactory
      let enemy = this.enemyFactory.createEnemy();
    }
    ...
    //More code below
  }
}
```

- Es decir queremos que todos sean Goomba pasamos GoombaFactory() o si queremos todos los enemigos de forma aletoria, pasamos RandomEnemyFactorty()
- Aquí hemos generalizado el mecanismo de pasar enemigos a una interfaz general.
- Aquí ya no tenemos límites: podemos pasar parámetros a una factory por ejemplo porcentajes de dificultad, niveles mínimos, cantidades, etc.
- Lo más importante es que no afecta a otra parte del código, ya que la independencia es entre interfaces

Ahora bien, como lo representamos en UML



- Clase abstracta Product
- Clase entidad ContreteProduct (los enemigos)
- Clase padre de las factories Creator
- Clase ConcreteCreator (por ejemplo el ramdomFactory o goombaFactory)

¿Qué diferencia hay con Abtract Factory?

- Son similares pero hacen cosas diferentes
- Las abstract Factory permiten crear diferentes tipos de objetos (familias de objetos) relacionados entre sí.
- No tienen un único método de creación, tienen varios