Xideral

Academia Java

Semana 1 -Dia 5

Inyección de dependencias

Presentado por:

Edgar Itzak Sánchez Rogers

Monterrey Nuevo León México a 16 de agosto de 2024

Introducción:

La inyección de dependencias es un patrón de diseño orientado a objetos. Es una clase a la que se le entregan los objetos de una clase en vez de ser la clase la que los crea. Se le llama dependencia por los contratos que nuestras clases requieren para poder funcionar. Nuestras clases no fabrican los objetos que requieren, en lugar de eso, reciben una clase 'contenedora' que incorporará la implementación que desean en nuestro contrato.

En resumen, la inyección de dependencias es un patrón de diseño que permite transferir a una clase la responsabilidad de la creación de instancias de un elemento a otro.

Demostración de Inyección de dependencias en Java:

A continuación, se muestra un código en Java de demostración que implementa el concepto de inyección de dependencias:

Interfaz arma:

```java
package com.edgaritzak.herogenerator;

public interface Weapon {

    String attack();
    String defend();
    String getWeaponName();

}
```

Objetos de Armas con diferentes estadísticas:

```java
package com.edgaritzak.herogenerator;

public class Sword implements Weapon {
    private int attackBonus;
    private int defenseBonus;

    public Sword() {
        attackBonus = 12;
        defenseBonus = 8;
    }
    @Override
    public String attack() {
        return "slashes with the sword. (Deals "+attackBonus+" damage)";
    }
    @Override
    public String defend() {
        return "blocks with the sword. (Blocks "+defenseBonus+" damage)";
    }
    @Override
    public String getWeaponName() {
        return "Sword";
    }
}
```

```java
package com.edgaritzak.herogenerator;

public class Shield implements Weapon {
    private int attackBonus;
    private int defenseBonus;

    public Shield() {
        attackBonus = 3;
        defenseBonus = 20;
    }
    @Override
    public String attack() {
        return "pushes the enemy with the shield (Deals "+attackBonus+" damag
    }
    @Override
    public String defend() {
        return "blocks with the shield. (Blocks "+defenseBonus+" damage)";
    }
    @Override
    public String getWeaponName() {
        return "Shield";
    }
}
```

```java
package com.edgaritzak.herogenerator;

public class Bow implements Weapon {
    private int attackBonus;
    private int defenseBonus;

    public Bow() {
        attackBonus = 17;
        defenseBonus = 4;
    }
    @Override
    public String attack() {
        return "shoots an arrow (Deals "+attackBonus+" damage)";
    }
    @Override
    public String defend() {
        return "blocks with the bow. (Blocks "+defenseBonus+" damage)";
    }
    @Override
    public String getWeaponName() {
        return "Bow";
    }

}
```

```java
package com.edgaritzak.herogenerator;

public class Spellbook implements Weapon {
    private int attackBonus;
    private int defenseBonus;

    public Spellbook() {
        attackBonus = 22;
        defenseBonus = 1;
    }
    @Override
    public String attack() {
        return "casts a spell (Deals "+attackBonus+" damage)";
    }
    @Override
    public String defend() {
        return "creates a magical barrier. (Blocks "+defenseBonus+" damage)";
    }
    @Override
    public String getWeaponName() {
        return "Spellbook";
    }
}
```

Clase Hero:

```java
package com.edgaritzak.herogenerator;

public class Hero {
    private String name;
    private Weapon weapon;
    private int healthPoints;

    public Hero(String name, Weapon weapon, int healthPoints) {
        this.name = name;
        this.weapon = weapon;
        this.healthPoints = healthPoints;
    }

    public String attack() {
        return name+" "+weapon.attack();
    }
    public String defend() {
        return name+" "+weapon.defend();
    }

    @Override
    public String toString() {
        return "Hero [name=" + name + ", healthPoints=" + healthPoints + ", weapon=" + weapon + "]";
    }
    //getters and setter
    public String getWeaponName() {
```

La función de attack() y defend() imprime el nombre del héroe y concatena con la acción del arma

Clase HeroGenerator (Inyector de dependencias)

```java
package com.edgaritzak.herogenerator;

public class HeroGenerator {

    public static Hero generate(String name, String weapon ,int healthPoints) {
        if (weapon.equals("sword")) return new Hero(name, new Sword(),healthPoints);
        if (weapon.equals("shield")) return new Hero(name, new Shield(),healthPoints);
        if (weapon.equals("spellbook")) return new Hero(name, new Spellbook(),healthPoints);
        if (weapon.equals("bow")) return new Hero(name, new Bow(),healthPoints);
        throw new UnsupportedOperationException("Could not generate hero");
    }

}

```

Contiene un método generate() que es el encargado de recibir todos los parámetros del héroe y arma para generar ambas y regresar el objeto héroe con su propia arma, aquí se crea el inyector de dependencias.

Clase Main:

```java
package com.edgaritzak.herogenerator;

public class Main {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

        Hero[] party = new Hero[]{
                HeroGenerator.generate("Ito", "sword", 8370),
                HeroGenerator.generate("Kondor", "bow", 7840),
                HeroGenerator.generate("Shauna ", "shield", 9999),
                HeroGenerator.generate("Krystallia", "spellbook", 7820)
            };

        for(Hero hero:party) {
            System.out.println(hero.toString());
            System.out.println(hero.attack());
            System.out.println(hero.defend());
            System.out.println("------------");
        }
    }
}
```

Genera héroes con nombres, armas y puntos de vida diferentes (aquí se ejecuta el inyector de dependencias), luego se muestra información cada uno de los héroes y ejecutan la acción de atacar y defender.

Ejecución:

```
Hero [name=Ito, healthPoints=8370, weapon=com.edgaritzak.herogenerator.Sword@5c8da962]
Ito slashes with the sword. (Deals 12 damage)
Ito blocks with the sword. (Blocks 8 damage)
------------
Hero [name=Kondor, healthPoints=7840, weapon=com.edgaritzak.herogenerator.Bow@5fe5c6f]
Kondor shoots an arrow (Deals 17 damage)
Kondor blocks with the bow. (Blocks 4 damage)
------------
Hero [name=Shauna , healthPoints=9999, weapon=com.edgaritzak.herogenerator.Shield@6979e8cb]
Shauna  pushes the enemy with the shield (Deals 3 damage)
Shauna  blocks with the shield. (Blocks 20 damage)
------------
Hero [name=Krystallia, healthPoints=7820, weapon=com.edgaritzak.herogenerator.Spellbook@763d9750]
Krystallia casts a spell (Deals 22 damage)
Krystallia creates a magical barrier. (Blocks 1 damage)
------------
```

Se muestran la información de cada héroe.

Conclusión:

La inyección de dependencias promueve el desacoplamiento entre las clases. Al inyectar dependencias en lugar de crearlas dentro de una clase, se reduce la dependencia directa entre componentes, lo que facilita la modificación y el mantenimiento del código. Al permitir la inyección de dependencias, se puede reemplazar fácilmente las implementaciones reales con implementaciones simuladas en las pruebas unitarias. Esto hace que las pruebas sean más fáciles de escribir y mantener

Referencias:

[1] Qué es la Inyección de Dependencias y cómo funciona | campusMVP.es

[2] Inyección de dependencias - Wikipedia, la enciclopedia libre

[3] Inyección de Dependencia y su utilidad - Arquitectura Java