

Entornos de Desarrollo

Práctica 5.2

Refactorización



Nathan

Gonzalez Mercado

1ºDAM

Índice

1. Introducción.....	3
2. Materiales.....	4
3. Ejercicios.....	5
3.1 Cuestiones teóricas.....	5
3.2 Ejercicios Prácticos.....	8
4. Conclusiones y Notaciones.....	26

1. Introducción

En esta práctica se verá el análisis y la refactorización del código que nos permitirá hacerlo más mantenible, comprensible, extraíble y manteniendo la documentación de los proyectos y códigos.

2. Materiales

Para realizar esta práctica, necesitaremos los siguientes materiales:

1. Equipo con JVM instalada.
2. Tener instalado el programa NetBeans
3. Tener instalado el programa Visual Studio Code.

3. Ejercicios

3.1 Cuestiones teóricas

1. Define con tus palabras. ¿Qué es y para qué se realiza la refactorización?

Refactorizar se trata de modificar la estructura del código, lo que permite que sea más simple de entender y a su vez más sencillo de modificar sin cambiar la forma en la que funciona el programa. La refactorización se realiza para hacer que nuestro código sea más simple de entender y de ampliar sin modificar el funcionamiento del mismo, más allá de las velocidades.

2. ¿En qué consiste la técnica de los dos sombreros? ¿Qué debe hacer cada uno de los sombreros? ¿Con un sombrero se puede hacer cosas asignadas a otro sombrero?

La técnica de los 2 sombreros consiste en que existen distintos roles, donde el sombrero representa ese rol y el desarrollador los llevará dependiendo del momento. Encontramos dos tipos de sombrero en esta técnica, los cuales son:

2.1. **Desarrollador:** Su función se basa en crear código y añadir pruebas, haciendo las comprobaciones necesarias para verificar que funcionan. No debe cambiar el código ya escrito.

2.2. **Refactorizador:** Se encarga de reestructurar el código sin añadir nuevo código ni pruebas, aunque debe comprobar que el programa supera las pruebas tras la refactorización.

Con uno de dichos sombreros equipados, exclusivamente podremos hacer uso de las funciones de dicho sombrero, es decir, no podremos añadir nuevo código llevando el sombrero **Refactorizador** y por tanto, llevando el sombrero **Desarrollador** no podremos reestructurar el código y deberemos centrarnos exclusivamente en escribir código **nuevo** y pruebas para dicho código. También existe la opción de que nos encontremos esta técnica incluyendo un sombrero adicional, el cuál es el de escritor de pruebas, quitando esa responsabilidad al sombrero **Desarrollador**.

3. ¿Qué es el “code smell”? ¿Cómo encontrarlo? Relacionar con la refactorización.

Se dice que el code smell es código que puede ser problemático y por ello debe ser necesario que reciba cambios para evitar dichos futuros problemas. Dichos “code smell” se clasifican de la siguiente manera que nos ayudaran a detectarlo:

3.1. **Inflado de código:** Hace referencia a códigos excesivamente grandes, independientemente de si es una clase, método o el número de parámetros entre otras cosas. Aquí encontramos los siguientes tipos:

3.1.1. Métodos demasiado largos**3.1.2. Clases demasiado grandes****3.1.3. Lista de parámetros larga**

3.2. Abuso de OO: Se refiere a usar incorrectamente la potencia de los POO y no seguir los principios del POO. Internamente encontramos:

3.2.1. Declaración de cambio**3.2.2. Legado Rechazado**

3.3. Impedidores del Cambio: Se trata de código que en caso de ser modificado tendremos que modificar otras partes del código, lo que implica que su mantenimiento sería costoso. Internamente nos encontramos:

3.3.1. Cambio divergente

3.4. Código prescindible Fragmentos de código sin utilizad y la eliminación de dichos fragmentos que harían el código más eficiente y entendible. Aquí encontramos:

3.4.1. Código duplicado

3.5. Acopladores. Código que provoca un alto grado de acoplamiento entre clases.

3.6. Otros: Son los que no se pueden clasificar en ninguna clasificación anterior.

Se relaciona con la refactorización ya que la necesidad de la misma se debe a problemas de diseño o a la implementación, la cuál es el caso que nos atañe.

4. Si un fragmento de código no tiene utilidad, ¿qué tipo de “code smell” es?

Se trata de **código prescindible** ya que no esta siendo de utilidad durante la ejecución.

5. Se tiene un método de 250 líneas. Comentar si es candidato a ser “code smell” y las razones para ello.

Se podría considerar candidato a code smell del tipo de **Inflado de código** y dentro de este el tipo de **Métodos demasiado largos**. Esto se debe a que el código cuenta con 250 líneas, y a partir de 10 se comienza a considerar sospechoso. Se trata de code smell debido a su excesiva cantidad de código que lo hace más complejo de entender y de mantener.

6. Un método tiene 10 parámetros. Explicar si es o no un “code smell”, y de ser así: la razón, como solucionarlo y que se obtiene al solucionarlo.

Un método que cuenta con 10 parámetros se puede considerar code smell del tipo **Inflado de código** y del subtipo **Lista de parámetros larga**. Esto se debe a que manejar un gran número de parámetros hace que sea mas complejo comprenderlo y a su vez lo hace contradictorio a medida que crece.

Para solucionarlo se remplazaran algunos de los parámetros con llamadas a método y si los datos provienen de fuentes diversas pasarlo como un único objeto. Realizando estas acciones obtenemos un código más legible y corto que puede evitar que el código se duplique.

7. En la refactorización, si se encuentra una sentencia “switch”. ¿Puede ser un problema? ¿Cuáles son los posibles tratamientos en caso de serlo?

Una sentencia switch puede ser un problema si está es compleja. Esto se debe a que al agregar una nueva condición se debe localizar todo el código afectado por dicho switch y modificarlo, además de analizar si el mismo puede llevar a cabo polimorfismo.

Los posibles tratamientos que se pueden llevar a cabo son los siguientes:

- 7.1. **Aislar el switch** y colocarlo en la clase correcta.
- 7.2. **Si se basa en un tipo** remplazarlo por subclases y especificarlas.

Lo más común es usar herencia y polimorfismo para solucionar este problema.

8. Se tiene un método muy largo, que es candidato a ser un “code smell”. ¿Qué técnicas se utilizan para resolverlo? Explicarlas.

Dado que, como se ha mencionado, nos encontramos con un método muy largo nos encontramos con un code smell del tipo **Inflado de código** e internamente **Métodos demasiados largos**. Para solventar esto, lo que podremos aplicar es extraer parte de la lógica del método y llevarla a un nuevo método o clases nuevas. Con esto, no solo reduciremos el propio método, si no que haremos que sea más sencillo de mantener y de comprender.

3.2 Ejercicios Prácticos.

NOTA: Los códigos se han adjuntado en la práctica, no obstante, al final del documento se adjuntará un enlace a GitHub que contendrá todos los ejercicios vistos a continuación,

1. Se tiene el siguiente código:

```
public class Shape {  
    public enum ShapeType {  
        Circle,  
        Rectangle,  
        Square,  
        Triangle  
    }  
    private ShapeType type;  
    private int p1;  
    private int p2;  
    public Shape (ShapeType type, int p1, int p2) {  
        this.type=type;  
        this.p1=p1;  
        this.p2=p2;  
    }  
    public double area () {  
        double resultado=0.0d;  
        switch (type) {  
            case Circle:  
                resultado = 2*Math.PI*p1;  
                break;  
            case Rectangle:  
            case Square:  
                resultado = p1*p2;  
                break;  
            case Triangle:
```

```
        resultado= p1*p2/2;
    default:
        break;
    }
    return resultado;
}
}
```

- Analizar si es candidato a la refactorización indicando los síntomas, la razón de que sea un problema, el tratamiento o solución y los beneficios que se obtiene.

Este código proporcionado es candidato a refactorización debido **la estructura del mismo**.

Dado que contamos con un switch que distintos métodos que podemos del cuál podemos evitar hacer uso haciendo que cada forma detecte el suyo propio y haga su operación debida. Además al usar todos los mismos métodos de la clase Shape se podría hacer como clase abstracta y que los tipos de ShapeType hereden de ella.

Para solucionarlo se debe de hacer la clase figuras y método área como abstracto, luego que cada figura herede de estos y de forma independiente cuando se vaya a realizar el calculo se pueda llevar acabo. Al ser abstracto y que cada figura de manera individual herede del mismo, ya no es necesario hacer uso del enum ShapeType ni del switch con las distintas opciones lo que permite eliminarlos. Haciendo esto obtenemos que las figuras se encuentren mejor distribuidas y nos permita modificar cada de manera independiente sin tener que añadir o modificar parámetros de la clase Shape

- Escribir el código para solucionarlo, usando los IDE's

Como hemos mencionado anteriormente, la mejor forma de solucionarlo es convertir la clase Shape en una clase abstracta y de ahí hereden el resto de formas lo que nos permite:

- Eliminar el enum ShapeType
- Eliminar el switch
- Separar las figuras en distintas clases.

Para ello, contaremos con los siguientes códigos:

ShapeRef.java

```
package Ejercicio1;

public abstract class ShapeRef {

    public int p1;

    public int p2;

    public ShapeRef(int p1, int p2) {

        this.p1 = p1;

        this.p2 = p2;

    }

    public abstract double area();

}
```

Circle.java

```
package Ejercicio1;

public class Circle extends ShapeRef {

    public Circle(int p1, int p2) {

        super(p1,p2);

    }

    @Override

    public double area() {

        double resultado = 0;

        resultado = 2 * Math.PI * this.p1;

        return resultado;

    }

}
```

Rectangle.java

```
package Ejercicio1;

public class Rectangle extends ShapeRef{

    public Rectangle(int p1, int p2) {

        super(p1, p2);

    }

    @Override

    public double area() {

        double resultado = 0;

        resultado = this.p1 * this.p2;

        return resultado;

    }

}
```

Square.java

```
package Ejercicio1;

public class Square extends ShapeRef {

    public Square(int p1, int p2) {

        super(p1, p2);

    }

    @Override

    public double area() {

        double resultado = 0;

        resultado = this.p1 * this.p2;

        return resultado;

    }

}
```

Triangle.java

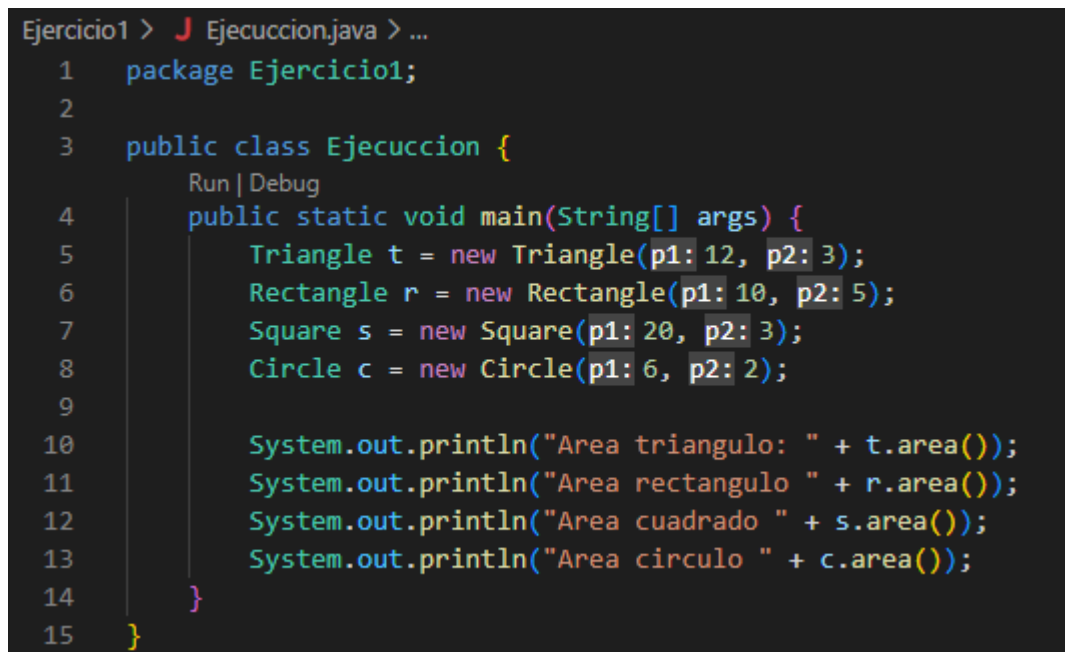
```
package Ejercicio1;

public class Triangle extends ShapeRef {

    public Triangle(int p1, int p2) {
        super(p1, p2);
    }

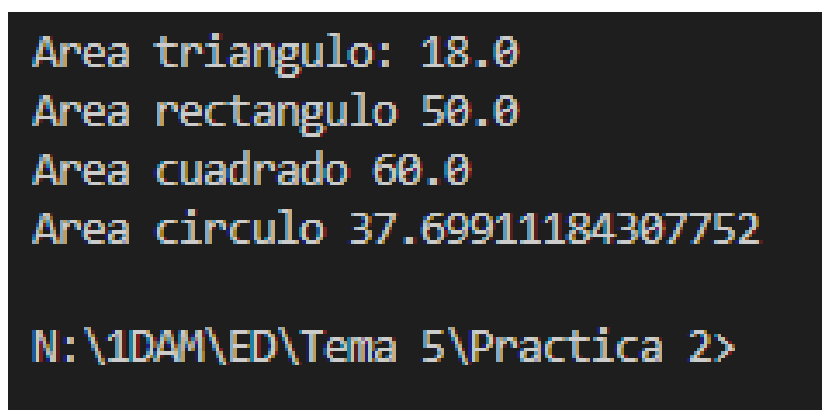
    @Override
    public double area() {
        double resultado = 0;
        resultado = this.p1*this.p2/2;
        return resultado;
    }
}
```

De manera que al tener los códigos de dicha manera, si los casteamos en un main podremos ver que todos y cada uno realizan las operaciones pertinentes.



```
Ejercicio1 > J Ejecucion.java > ...
1  package Ejercicio1;
2
3  public class Ejecucion {
4      Run | Debug
5      public static void main(String[] args) {
6          Triangle t = new Triangle(p1: 12, p2: 3);
7          Rectangle r = new Rectangle(p1: 10, p2: 5);
8          Square s = new Square(p1: 20, p2: 3);
9          Circle c = new Circle(p1: 6, p2: 2);
10
11          System.out.println("Area triangulo: " + t.area());
12          System.out.println("Area rectangulo " + r.area());
13          System.out.println("Area cuadrado " + s.area());
14          System.out.println("Area circulo " + c.area());
15      }
16  }
```

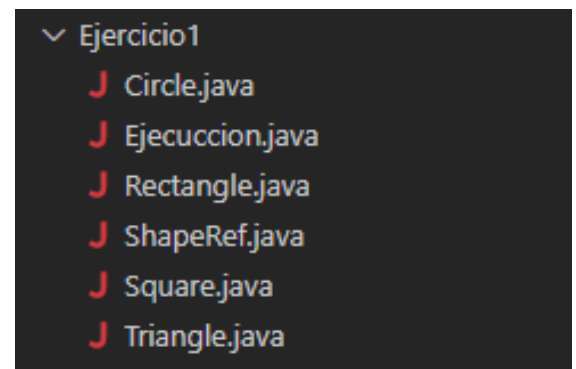
Figura 1: Programa Ejecución (código adjunto debajo de estas capturas.)



```
Area triangulo: 18.0
Area rectangulo 50.0
Area cuadrado 60.0
Area circulo 37.69911184307752

N:\1DAM\ED\Tema 5\Practica 2>
```

Figura 3: Ejecución de dicho programa



```
▼ Ejercicio1
  J Circle.java
  J Ejecucion.java
  J Rectangle.java
  J ShapeRef.java
  J Square.java
  J Triangle.java
```

Figura 2: Estructura final de Ejercicio 1.

Ejecucion.java

```
package Ejercicio1;

public class Ejecucion {
    public static void main(String[] args) {
        Triangle t = new Triangle(12, 3);
        Rectangle r = new Rectangle(10, 5);
        Square s = new Square(20, 3);
        Circle c = new Circle(6, 2);

        System.out.println("Area triangulo: " + t.area());
        System.out.println("Area rectangulo " + r.area());
        System.out.println("Area cuadrado " + s.area());
        System.out.println("Area circulo " + c.area());
    }
}
```

De esta forma cada figura contará con su propia clase donde todas heredaran de ShapeRef de manera que cada una actuará de manera independiente pero obteniendo una estructura inicial proveniente de ShapeRef, la cual incluye el constructor y el metodo abstracto area que luego podremos modificar para cada figura de forma independiente, lo que nos permitirá hacer uso del polimorfismo y de la herencia.

2. Se tiene un juego en el que el personaje puede tener diferentes elementos que le dan ciertas habilidades, por ejemplo, un escudo que decrementa el poder de ataque del enemigo en un 25% o una espada que incrementa su poder de ataque en un 40%.

El código es el siguiente:

```
public class Warrior{  
    private double life;  
    private String name;  
    private double swordpower=1.0d;  
    private String swordname = "apprentice sword";  
    private String shieldname = "apprentice shield";  
    private double shieldpower = 0.25d;  
    public Warrior() {  
        this.life = 100d;  
        this.name="Undefined";  
    }  
    public Warrior (String name) {  
        this.name = name;  
    }  
    public double attack() {  
        return this.swordpower;  
    }  
    public void defense (Warrior w) {  
        this.life -= w.attack()*shieldpower;  
    }  
    public String toString() {  
        return "Name:" + this.name + " life:" + this.life + " Shield:" + shieldname + " Sword: " + swordname;  
    }  
}
```


- ¿Qué sucede si ahora se tiene una espada que además de atacar también sirve para defenderse?
¿Y si se tiene un nuevo escudo con poderes mágicos que al defenderse en vez de decrementar la vida incrementa?

Si contáramos con una espada que además nos permite defendernos el poder de escudo debería incrementar en base a lo que nos permita defendernos dicha espada. En el caso del escudo que incrementa la vida la formula de defensa cambiaría a aumentar la vida en lugar de restarla.

- Si se añaden nuevos escudos/espadas con cualidades nuevas a lo largo del proyecto, ¿es necesario estar cambiando y modificando el código?

Dependiendo de las cualidades de dichas espadas o escudos si, ya que como vimos anteriormente con el escudo que aumenta la vida en lugar de restarla, tendríamos que modificar el código de forma que cumpla nuestros requisitos.

- Buscar entre los “code smells” de tipo “change-preventers” el que mejor casa con el problema anterior (justificar) y aplicar alguna de las técnicas propuestas (con código) usando las herramientas proporcionadas por los IDE’s (NetBeans o VSC)

El tipo de change-preventers más adecuado en esta situación se corresponde con el “**Divergent Change**”. Esto se debe a que para poder aplicar lo anteriormente mencionado de cambiar funcionalidades de espadas y escudos deberemos estar cambiando operaciones y métodos, siendo el ejemplo más claro el del escudo que en lugar de restar vida, nos la aumentaría, que para ello deberíamos cambiar completamente el funcionamiento del método defensa.

Además, **Divergent Change** es utilizado cuando en una clase debemos alterar distintos métodos que no están relacionados para que estos funcionen, cosa que localizamos en la espada que nos permite defendernos (debe alterar los datos del escudo y también la defensa) y también el escudo que nos permite curarnos (se debe modificar por completo el funcionamiento del método defensa.)

Para solucionar este problema de nuestro código, debemos separar de la clase Warrior la espada y el escudo y con ello sus atributos. Siguiendo con la lógica de los enunciados anteriores, de que tendremos una espada que nos permita defendernos y un escudo que nos permita curarnos tendremos lo siguiente en los códigos:

Clase Warrior

```
public class Warrior {
    private double life;
    private String name;
    private Sword sword;
    private Shield shield;
    public Warrior() {
        this.life = 100d;
        this.name = "Undefined";
        this.sword = new Sword();
        this.shield = new Shield();
    }
    public Warrior(String name) {
        this.name = name;
        this.life = 100d;
        this.sword = new Sword();
        this.shield = new Shield();
    }
    public double attack() {
        return this.sword.getSwordpower();
    }
    public void defense(Warrior w) {
        if (shield.isCurativo()) {
            shield.defense(w);
        } else if (sword.getSworddefense() > 0) {
            sword.defensa(w);
        } else {
            shield.defense(w);
        }
    }
    public String toString() {
        return "Name: " + this.name + " life: " + this.life +
            " Shield: " + shield.getShieldname() + " Sword: " +
            sword.getSwordname();
    }
    public double getLife() {
        return life;
    }
    public void setLife(double life) {
        this.life = life;
    }
    public Sword getSword() {
        return sword;
    }
}
```

```
}  
  
public void setSword(Sword sword) {  
    this.sword = sword;  
}  
public Shield getShield() {  
    return shield;  
}  
public void setShield(Shield shield) {  
    this.shield = shield;  
}  
}
```

Clase Sword

```
public class Sword {  
    private double swordpower;  
    private String swordname;  
    private double sworddefense;  
    public Sword() {  
        this.swordname = "ESP Inicial";  
        this.swordpower = 1.0d;  
        this.sworddefense = 0.0d;  
    }  
    public Sword(double swordpower, String swordname) {  
        this.swordpower = swordpower;  
        this.swordname = swordname;  
        this.sworddefense = 0;  
    }  
    public Sword(double swordpower, String swordname, double sworddefense) {  
        this.swordpower = swordpower;  
        this.swordname = swordname;  
        this.sworddefense = sworddefense;  
    }  
    public void defensa(Warrior w) {  
        w.setLife(w.getLife() - getSworddefense());  
    }  
    public double getSwordpower() {  
        return swordpower;  
    }  
    public void setSwordpower(double swordpower) {  
        this.swordpower = swordpower;  
    }  
}
```

```
public String getSwordname() {  
    return swordname;  
}  
public void setSwordname(String swordname) {  
    this.swordname = swordname;  
}  
public double getSworddefense() {  
    return sworddefense;  
}  
public void setSworddefense(double sworddefense) {  
    this.sworddefense = sworddefense;  
}  
}
```

Clase Escudo

```
public class Shield {  
    private String shieldname;  
    private double shieldpower;  
    private boolean curativo;  
    public Shield(String shieldname, double shieldpower, boolean curativo) {  
        this.shieldname = shieldname;  
        this.shieldpower = shieldpower;  
        this.curativo = curativo;  
    }  
    public Shield() {  
        this.shieldname = "ESC Inicial";  
        this.shieldpower = 0.25d;  
        this.curativo = false;  
    }  
    public void defense(Warrior w) {  
        if (curativo) {  
            w.setLife(w.getLife() + this.shieldpower);  
        } else {  
            w.setLife(w.getLife() - w.attack() * this.shieldpower);  
        }  
    }  
    public String getShieldname() {  
        return shieldname;  
    }  
    public void setShieldname(String shieldname) {  
        this.shieldname = shieldname;  
    }  
}
```

```
}  
public double getShieldpower() {  
    return shieldpower;  
}  
public void setShieldpower(double shieldpower) {  
    this.shieldpower = shieldpower;  
}  
public boolean isCurativo() {  
    return curativo;  
}  
public void setCurativo(boolean curativo) {  
    this.curativo = curativo;  
}  
}
```

```

public class Shield {
    private String shieldname;
    private double shieldpower;
    private boolean curativo;

    /**
     * Constructor del escudo sobrecargado
     * @param shieldname Indica el nombre del escudo obtenido
     * @param shieldpower Indica la potencia del escudo.
     */
    public Shield(String shieldname, double shieldpower, boolean curativo) {
        this.shieldname = shieldname;
        this.shieldpower = shieldpower;
        this.curativo = curativo;
    }

    /**
     * Constructor del escudo sin cargar.
     */
    public Shield() {
        this.shieldname = "ESC Inicial";
        this.shieldpower = 0.25d;
        this.curativo = false;
    }

    public void defense(Warrior w) {
        if (curativo) {
            w.setLife(w.getLife() + this.shieldpower);
        } else {
            w.setLife(w.getLife() - w.attack() * this.shieldpower);
        }
    }

    //Getters y Setters
    public String getShieldname() {
        return shieldname;
    }

    public void setShieldname(String shieldname) {
        this.shieldname = shieldname;
    }

    public double getShieldpower() {
        return shieldpower;
    }

    public void setShieldpower(double shieldpower) {
        this.shieldpower = shieldpower;
    }

    public boolean isCurativo() {
        return curativo;
    }

    public void setCurativo(boolean curativo) {
        this.curativo = curativo;
    }
}

```

Figura 5: Clase Escudo

```

public class Sword {
    private double swordpower;
    private String swordname;
    private double sworddefense;

    /**
     * Constructor de la espada sin cargar
     */
    public Sword() {
        this.swordname = "ESP Inicial";
        this.swordpower = 1.0d;
        this.sworddefense = 0.0d;
    }

    /**
     * Constructor de la espada sobrecargado.
     * No permite defenderse con ella.
     * @param swordpower Potencia de la espada creada
     * @param swordname Nombre de la espada creada.
     */
    public Sword(double swordpower, String swordname) {
        this.swordpower = swordpower;
        this.swordname = swordname;
        this.sworddefense = 0;
    }

    /**
     * Constructor de la espada sobrecargada.
     * Permite la defensa con la misma.
     * @param swordpower Potencia de la espada al atacar.
     * @param swordname Nombre de la espada.
     * @param sworddefense Potencia de la espada al defenderse.
     */
    public Sword(double swordpower, String swordname, double sworddefense) {
        this.swordpower = swordpower;
        this.swordname = swordname;
        this.sworddefense = sworddefense;
    }

    /**
     * Método para defenderse con la espada si esta lo permite.
     * @param w Guerrero que va a defenderse.
     */
    public void defensa(Warrior w) {
        w.setLife(w.getLife() - getSworddefense());
    }

    public double getSwordpower() {
        return swordpower;
    }

    public void setSwordpower(double swordpower) {
        this.swordpower = swordpower;
    }

    public String getSwordname() {
        return swordname;
    }

    public void setSwordname(String swordname) {
        this.swordname = swordname;
    }

    public double getSworddefense() {
        return sworddefense;
    }

    public void setSworddefense(double sworddefense) {
        this.sworddefense = sworddefense;
    }
}

```

Figura 4: Clase Sword

```
public class Warrior {
    private double life;
    private String name;
    private Sword sword;
    private Shield shield;
    /**
     * Constructor del Guerrero vacío.
     */
    public Warrior() {
        this.life = 100d;
        this.name = "Undefined";
        this.sword = new Sword();
        this.shield = new Shield();
    }
    /**
     * Constructor del guerrero sobrecargado
     * @param name Nombre de nuestro guerrero.
     */
    public Warrior(String name) {
        this.name = name;
        this.life = 100d;
        this.sword = new Sword();
        this.shield = new Shield();
    }
    /**
     * Método que nos muestra el ataque de la espada actual
     * @return Poder de ataque de nuestra espada actual.
     */
    public double attack() {
        return this.sword.getSwordpower();
    }
    /**
     * Método que hará que nuestro guerrero se defienda
     * Si su espada perm
     * @param w Guerrero que va a defenderse de un ataque.
     */
    public void defense(Warrior w) {
        /*
         * this.life -= w.attack() * this.shield.getShieldpower();
         */
        if (shield.isCurativo()) {
            shield.defense(w);
        } else if (sword.getSworddefense() > 0) {
            sword.defensa(w);
        } else {
            shield.defense(w);
        }
    }
    public String toString() {
        return "Name: " + this.name + " life: " + this.life +
            " Shield: " + shield.getShieldname() + " Sword: " + sword.getSwordname();
    }
    public double getLife() {
        return life;
    }
    public void setlife(double life) {
        this.life = life;
    }
    public Sword getSword() {
        return sword;
    }
    public void setSword(Sword sword) {
        this.sword = sword;
    }
    public Shield getShield() {
        return shield;
    }
    public void setShield(Shield shield) {
        this.shield = shield;
    }
}
```

Figura 6: Clase Warrior

Como podemos apreciar, en las clases sword y shield contamos con el método defensa y en la clase warrior contamos con un if que comprobará primero que el escudo pueda curarse haciendo uso del boolean curativo y ejecutará el método defensa del escudo, que a su vez comprobará si el escudo permite curarnos de manera que ejecutará una operación, si no ejecutará la estándar de restarnos vida. Si el escudo no es curativo miraremos si la espada tiene la capacidad de defendernos mediante el double sworddefense. Si es mayor a 0 nuestra espada nos permitirá defendernos. Si ninguna de estas se cumple nos defenderemos normalmente con nuestro escudo. De manera que si ejecutamos realizando cambios de la espada y escudo se vería así:

```
Name: Nathan life: 100.0 Shield: ESC Inicial Sword: ESP Inicial
Poder de ataque: 1.0

Has obtenido la Espada Demoniac
Tu escudo se ha transformado en Escudo roto

Name: Nathan life: 99.75 Shield: Escudo roto Sword: Espada Demoniac
Potencia actual de ataque: 666.0

Has obtenido el Escudo roto
Al defenderte te curarás

Name: Nathan life: 100.0 Shield: Escudo Sangriento Sword: Espada Demoniac

¡Ahora puedes defenderte con tu espada!
Poder de defensa de tu espada: 0.25

Name: Nathan life: 99.75 Shield: Escudo roto Sword: Espada Demoniac
```

Donde se puede apreciar que se empieza con unos armamentos iniciales y que se puede ver que al inicio no nos curamos al defendernos, no obstante, al cambiar de escudo a uno que tenga la capacidad si lo hará. Por ultimo podremos apreciar que la espada también cuenta con la capacidad de defenderse o no dependiendo de sus atributos.

3. Un compañero ha realizado una prueba técnica para un trabajo y lo han descartado ya que su código huele mal. El no entiende la razón de que haya sido descartado para conseguir el empleo y nos pide que revisemos el código entregado para darle nuestra opinión. Aunque le han dicho que tiene que ver con el acoplamiento de las clases (<https://refactoring.guru/refactoring/smells/couplers>). Analizar el código y explicar las razón de que su código huela mal, decidir que técnica de refactorización usar para solucionarlo y qué se gana al realizar esta refactorización.

Como bien le han dicho, el código huele mal debido al acoplamiento de las clases. Para ser más exactos, del tipo “Feature Envy”, también conocido como Envidia de funciones, ya que estaremos accediendo más a los datos de la bala que al propio barco desde la clase barco. Para solucionarlo, deberemos mover las clases que son pensadas exclusivamente para ellas, en este caso son las siguientes:

1. update()
2. collision(Ship s)

De forma que pasaran de pertenecer a **Ship** a pertenecer a **Bullet**, la cual es la clase más adecuadas para estos. Con esto obtendremos que haya menos código duplicado, siempre y cuando el código se coloque en una posición central, y también obtendremos una mejor organización del código, ya que los datos que vamos a manejar se localizan al lado de los datos reales.

- Realizar la refactorización en el código usando los entornos de desarrollo.

Clase Bullet

```
public class Bullet {
    private int x;
    private int y;
    /**
     * Constructor de la bala con sus coordenadas
     * @param x: coordenada X de la bala.
     * @param y: coordenada Y de la bala.
     */
    public Bullet(int x, int y) {
        this.x = x;
        this.y = y;
    }
    /**
     * Actualiza la posición de la bala actual en la posición X
     */
    public void update() {
        this.setX(getX() + 1);
    }
}
```

```
/**
 * Booleano que indicará si un barco esta colisionando con una bala
 * @param s Barco a comprobar colisiones
 * @return devolverá verdadero si la posición de la bala
 * coincide con la posición actual del barco
 * y devolverá falso si las balas y el barco no coinciden
 * en posiciones.
 */
public boolean collision(Ship s) {
    if (s.getX() == this.getX() && s.getY() == this.getY()) {
        return true;
    } else {
        return false;
    }
}

//Getters y Setters de posicion de balas.
public int getX() {
    return x;
}

public void setX(int x) {
    this.x = x;
}

public int getY() {
    return y;
}

public void setY(int y) {
    this.y = y;
}
}
```

Clase Ship

```
public class Ship {
    private int x;
    private int y;
    public Bullet bullet;
    //Constructor Barco
    public Ship(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

```
//Crea Bala
public void shoot() {
    this.bullet = new Bullet(x, y);
}

//Movimientos Barco
public void moveLeft() {
    this.x--;
}

public void moveRigth() {
    this.x++;
}

public void moveUp() {
    this.y--;
}

public void Down() {
    this.y++;
}

//Getters y Setters de las coordenadas Barco
public int getX() {
    return x;
}

public void setX(int x) {
    this.x = x;
}

public int getY() {
    return y;
}

public void setY(int y) {
    this.y = y;
}

//Getters y Setters de la bala.
public Bullet getBullet() {
    return bullet;
}

public void setBullet(Bullet bullet) {
    this.bullet = bullet;
}

//toString de coordenadas del barco y coordenadas del bullet.
public String toString(){
    return "X:"+this.x+" Y:"+this.y+"Bullet>X:"+this.bullet.getX()+" Y:"+this.bullet.getY();
}
}
```

```

public class Ship {
    private int x;
    private int y;
    public Bullet bullet;

    //Constructor Barco
    public Ship(int x, int y) {
        this.x = x;
        this.y = y;
    }

    //Crea Bala
    public void shoot() {
        this.bullet = new Bullet(x, y);
    }

    //Movimientos Barco
    public void moveLeft() {
        this.x--;
    }

    public void moveRigth() {
        this.x++;
    }

    public void moveUp() {
        this.y--;
    }

    public void Down() {
        this.y++;
    }

    //Getters y Setters de las coordenadas Barco
    public int getX() {
        return x;
    }

    public void setX(int x) {
        this.x = x;
    }

    public int getY() {
        return y;
    }

    public void setY(int y) {
        this.y = y;
    }

    //Getters y Setters de la bala.
    public Bullet getBullet() {
        return bullet;
    }

    public void setBullet(Bullet bullet) {
        this.bullet = bullet;
    }

    //toString de coordenadas del barco y coordenadas del bullet.
    public String toString(){
        return "X:"+this.x+" Y:"+this.y+"Bullet>X:"+this.bullet.getX()+" Y:"+this.bullet.getY();
    }
}

```

Figura 7: Clase Ship

```

1 package Ejercicio3;
2
3 public class Bullet {
4     private int x;
5     private int y;
6
7     /**
8      * Constructor de la bala con sus coordenadas
9      * @param x: coordenada X de la bala.
10     * @param y: coordenada Y de la bala.
11     */
12     public Bullet(int x, int y) {
13         this.x = x;
14         this.y = y;
15     }
16
17     /**
18     * Actualiza la posición de la bala actual en la posición X
19     */
20     public void update() {
21         this.setX(this.getX() + 1);
22     }
23
24     /**
25     * Booleano que indicará si un barco esta colisionando con una bala
26     * @param s Barco a comprobar colisiones
27     * @return devolverá verdadero si la posición de la bala
28     * coincide con la posición actual del barco
29     * y devolverá falso si las balas y el barco no coinciden
30     * en posiciones.
31     */
32     public boolean collision(Ship s) {
33         if (s.getX() == this.getX() && s.getY() == this.getY()) {
34             return true;
35         } else {
36             return false;
37         }
38     }
39
40     //Getters y Setters de posición de balas.
41     public int getX() {
42         return x;
43     }
44
45     public void setX(int x) {
46         this.x = x;
47     }
48
49     public int getY() {
50         return y;
51     }
52
53     public void setY(int y) {
54         this.y = y;
55     }
56 }

```

Figura 8: Clase Bullet

Al hacer esto, haremos que ship no acceda tanto a Bullet y que el propio bullet haga dichas operaciones.

4. Conclusiones y Notaciones

Esta práctica a mi parecer ha sido una práctica que en la teoría ha resultado bastante sencilla de comprender mientras que la práctica ha resultado ser más compleja de lo esperado encontrando algunos códigos que, pese a saber el tipo de code smell que contiene, no encontraba la forma de solucionar dicho code smell. Quitando ese pequeño inconveniente, la práctica se puede realizar correctamente y entenderla sin problemas.

A continuación, adjunto enlace al repositorio de GitHub que contiene todos los códigos de los programas anteriores. En dicho repositorio se encontrará:

1. Carpetas correspondientes a cada ejercicio. Cada una de ellas contendrá:
 - 1.1. Códigos ya solucionados.
 - 1.2. Códigos originales otorgados en los enunciados de los ejercicios en una subcarpeta.
2. El PDF que contiene el enunciado de este ejercicio.
3. Una copia de este PDF.
4. Fichero README.md que contendrá lo mismo expresado en este último apartado.

Enlace en cuestión: <https://github.com/Nathan-GM/Refactorizacion>