

Parcial de OO2 - Curso 2022 - 16/jul/2022

Ejercicio 1 - Patrones

Sea una estación meteorológica hogareña que permite conocer información de varios aspectos del clima. Esta estación está implementada con la clase `HomeWeatherStation` que interactúa con varios sensores para conocer fenómenos físicos. La misma implementa los siguientes métodos:

```
//retorna la temperatura en grados Fahrenheit
public double getTemperaturaFahrenheit()
```

```
//retorna la presión atmosférica en hPa
public double getPresion()
```

```
//retorna la radiación solar
public double getRadiacionSolar()
```

```
//retorna una lista con todas las temperaturas sensadas hasta el momento, en
grados Fahrenheit
public List<Double> getTemperaturasFahrenheit()
```

Esta clase se encuentra implementada por terceros y no se puede modificar.

Nos piden construir una aplicación que además de lo anteriormente descripto pueda obtener:

- La temperatura en grados Celsius ($^{\circ}\text{C} = (^{\circ}\text{F} - 32) \div 1.8$).
- El promedio de las temperaturas históricas en grados Fahrenheit.

Además, la aplicación debe permitir al usuario configurar qué datos mostrar y en qué orden. Esto significa que podría querer ver la información de muchas maneras, por ejemplo:

- Ejemplo 1: "Presión atmosférica: 1008"
- Ejemplo 2: "Presión atmosférica: 1008 Radiación solar: 500"
- Ejemplo 3: "Radiación solar: 500 Temperatura C: 28 Promedio de temperaturas C: 25"

Para ello, usted debe proveer en algún punto de su solución, la implementación del mensaje `public String displayData()` que devuelva los datos elegidos en el orden configurado (dado que la app aun no cuenta con interface de usuario).

Haga uso de la clase `HomeWeatherStation` sin modificarla.

Tareas:

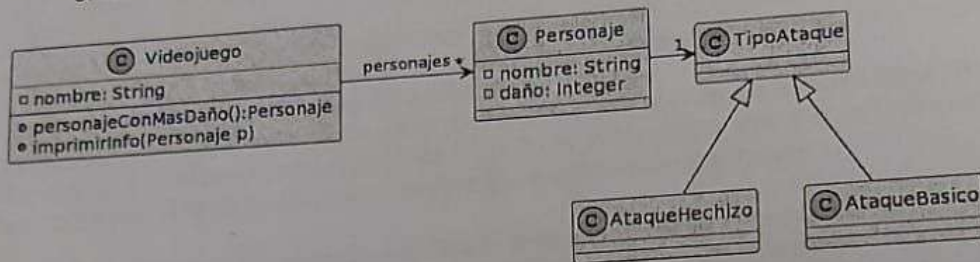
- 1- Modele una solución para el problema planteado. Si utiliza algún patrón, indique cuál
- 2- Implemente en Java
- 3- Implemente un test para validar la configuración del ejemplo 2, asumiendo que en el momento de la ejecución del mismo, los sensores arrojan los valores del ejemplo.

Ejercicio 2 - Refactoring

Para el siguiente código, realice las siguientes tareas:

- indique qué mal olor presenta
- indique el refactoring que lo corrige
- aplique el refactoring mostrando únicamente el código que cambió, detallando cada paso intermedio.

Si vuelve a encontrar un mal olor, retorne al paso (i).



```

public class Videojuego{
    // ...
    //
    public Personaje personajeConMasDaño() {
        Personaje temp = null;
        double max= 0;
        for (Personaje p : personajes) {
            double daño = p.getTipoAtaque().calcularDaño(p.getDaño());
            if (daño > max){
                temp = p;
                max = daño;
            }
        }
        return temp;
    }

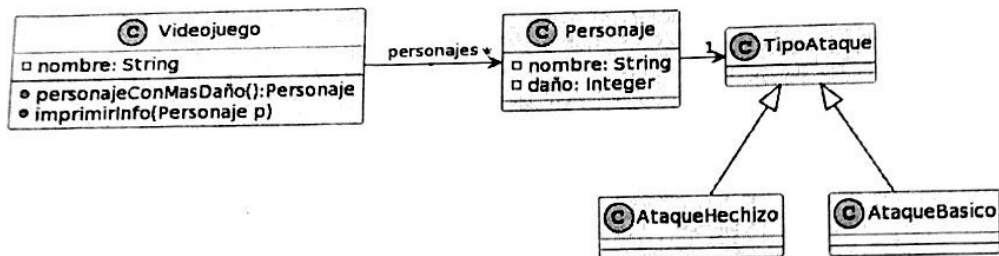
    public void imprimirInfo(Personaje p){
        System.out.println(p.getNombre() + "tiene como daño " + p.getDaño());
        if (p.getTipoAtaque().getClass() == AtaqueHechizo.class) {
            System.out.println("Ataque tipo hechizo");
            System.out.println("Este ataque dobla tu fuerza");
        } else{
            System.out.println("Ataque tipo Ataque Básico");
            System.out.println("Este ataque mantiene tu fuerza");
        }
    }
}
    
```

Ejercicio 2 - Refactoring

Para el siguiente código, realice las siguientes tareas:

- (i) indique qué mal olor presenta
- (ii) indique el refactoring que lo corrige
- (iii) aplique el refactoring mostrando únicamente el código que cambió, detallando cada paso intermedio.

Si vuelve a encontrar un mal olor, retorne al paso (i).

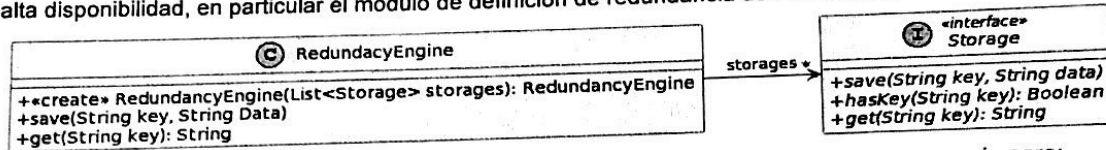


```
public class Videojuego{
    // ...
    //
    public Personaje personajeConMasDaño() {
        Personaje temp = null;
        double max= 0;
        for (Personaje p : personajes) {
            double daño = p.getTipoAtaque().calcularDaño(p.getDaño());
            if (daño > max){
                temp = p;
                max = daño;
            }
        }
        return temp;
    }

    public void imprimirInfo(Personaje p){
        System.out.println(p.getNombre() + "tiene como daño " + p.getDaño());
        if (p.getTipoAtaque().getClass() == AtaqueHechizo.class) {
            System.out.println("Ataque tipo hechizo");
            System.out.println("Este ataque dobla tu fuerza");
        } else{
            System.out.println("Ataque tipo Ataque Básico");
            System.out.println("Este ataque mantiene tu fuerza");
        }
    }
}
```

Ejercicio 3 - Frameworks

Considere el siguiente extracto de código y diagrama de clases UML de un framework para sistemas de alta disponibilidad, en particular el módulo de definición de redundancia de información.



Este framework provee una clase, *RedundancyEngine*, que define la lógica invariante necesaria para:

- (i) almacenar la información en más de un espacio de almacenamiento, esto es, el almacenamiento de cada clave/valor en todos los *Storage* provistos, y,
- (ii) recuperar el valor correspondiente a una clave desde el primer storage que posea dicha información.

Por otro lado, la interface *Storage*, también provista por el framework, define el protocolo necesario para que *RedundancyEngine* pueda almacenar y recuperar la información solicitada.

Es responsabilidad de las personas que utilizan el framework definir el comportamiento específico para almacenar la información en, por ejemplo, un archivo, en memoria, o cualquier otro soporte. Siempre mediante la implementación de la interface *Storage*.

Considerando que, como usuarios/as del framework, ya hemos definido dos clases que implementan la interface *Storage* llamadas *FileStorage* e *InMemoryStorage*, deberíamos usar *RedundancyEngine* para acceder y almacenar información de la siguiente manera:

```
List<Storage> storages = new ArrayList<Storage>();
storages.add(new InMemoryStorage(...));
storages.add(new FileStorage(...));

RedundancyEngine redundancyEngine = new RedundancyEngine(storages);

redundancyEngine.save("name", "Diego Maradona");

// La línea siguiente debería retornar el String "Diego Maradona"
redundancyEngine.get("name");

public class RedundancyEngine{
    List<Storage> storages;
    public RedundancyEngine(List<Storage> storages){
        super();
        this.storages = storages;
    }
    public void save(String key, String data){
        for (InformationStorage storage : this.getStorages()){
            storage.save(key, data)
        }
    }
    public String get(String key){
        for (InformationStorage storage : this.getStorages()){
            if (storage.hasKey(key)){
                return storage.get(key);
            }
        }
        return null;
    }
}

public interface Storage{
    public void save(String key, String data);
    public Boolean hasKey(String key);
    public String get(String key);
}
```

Responda las siguientes preguntas, basándose en el subconjunto de clases y métodos del framework presentado anteriormente:

1. ¿El comportamiento variable del framework (hotspots), está implementado mediante herencia o composición? Justifique su respuesta.
2. ¿Cuáles son los hook methods?
3. ¿Cuál es el Frozen Spot?