Diseño de Software Orientado a Objetos
Práctico 2: Patrón Strategy

Ejercicio 0. Descargue el código provisto en el repositorio git a continuación:

https://github.com/pponzio/oo-design-2024.git

La carpeta `2-strategy-duck-simulator` contiene el código complementario a esta práctica. Se proveen scripts gradle para compilar automáticamente el código y ejecutar los tests (ej. ejecutar *gradle test* en línea de comandos). Se recomienda utilizar algún IDE (ej. IntelliJ Idea) para facilitar la codificación, el refactoring, la ejecución de tests y el debugging.

Ejercicio 1. Utilice el código provisto del Duck Simulator para experimentar y entender el patrón Strategy.

a) Defina tests que creen varios objetos distintos de las subclases de `Duck`, y cambie sus comportamientos en tiempo de ejecución. Agregue algunas subclases nuevas para `Duck`, `FlyBehavior` y `QuackBehavior`.
b) Cree una clase `DucksFlock` que implemente una bandada de patos. La clase debe tener los métodos `fly` y `quack`, que hagan volar y hacer cuac, respectivamente, a todos los miembros de la bandada. Utilice el patrón Strategy para asegurarse que `DucksFlock` esté desacoplado de las implementaciones concretas de las subclases de `Duck`. Desarrolle algunos tests para `DucksFlock`.
c) Agregue un nuevo tipo de pato, y cree una bandada que contenga instancias de este tipo. ¿Tuvo que modificar la implementación de `DucksFlock`, `Duck` y las subclases de `Duck` existentes?

Ejercicio 2. Desarrolle una versión preliminar de un juego de acción y aventuras, utilizando la metodología de TDD. El juego tiene distintos tipos de personajes (ej. caballeros, magos, etc.). Cada personaje puede tener un arma (el arma por defecto son golpes de puño). Durante una pelea entre dos personajes, cada golpe con un arma produce un cierto daño al oponente (ej. un golpe de espada reduce la vida del adversario en 50).

a) Implemente algunos personajes y armas del juego usando el patrón Strategy, de modo que los personajes puedan tener flexibilidad en el uso de armas, e incluso cambiarlas en tiempo de ejecución.
b) Añada una funcionalidad que permita hacer luchar a dos personajes. En una lucha los personajes alternan golpes, con el arma que tenga cada uno actualmente, hasta que uno de ellos muere.

c) Una vez completados los dos puntos anteriores, cree nuevos personajes y armas, y hágalos luchar contra otros personajes existentes. Esto debería poder implementarse sin modificar el código existente (sólo agregando clases).
d) Los personajes ahora se clasifican entre: los que pueden pelear cuerpo a cuerpo, los que pelean a distancia, con magia, etc. Y las armas también tienen esta clasificación. Introduzca restricciones de integridad que aseguren que sólo los personajes que pelean cuerpo a cuerpo puedan usar espadas, los que pelean a distancia puedan usar arcos, etc.

Ejercicio 3. Implemente usando TDD un programa que imprima por pantalla los primeros *n* números primos (para *n* dado).

a) Refactorice usando el patrón Strategy para que su código admita distintas implementaciones del algoritmo de cómputo de primos.
b) Agregue una nueva implementación de un algoritmo de cómputo de primos sin modificar el código existente.
c) Averigüe cómo usar tests parametrizados en JUnit, y modifique sus tests de modo que sirvan para testear las distintas implementaciones de cómputo de primos.
d) Descargue una nueva implementación del algoritmo de cómputo de primos de internet, y hágala encajar en su código actual (y en sus tests) sin modificar el código existente.
e) Utilice el patrón Strategy para refactorizar su código para que admita distintas formas de mostrar la salida.
f) Extienda su programa para que sea posible guardar la salida en un archivo (además de mantener la posibilidad de poder imprimir por pantalla).