Ingeniería de Software
Año 2025
Práctico 2: Patrón Strategy

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

https://github.com/ingenieria-de-software-unrc/practicos-ingenieria-25

La carpeta `2-strategy` contiene el código complementario a esta práctica. Se proveen scripts gradle para construir (*build*) automáticamente el proyecto (`./gradlew build`), ejecutar el main (`./gradlew run`), y ejecutar los tests (`./gradlew test`).
Se recomienda utilizar algún IDE (por ejemplo, IntelliJ IDEA) para facilitar la codificación, el refactoring, la ejecución de tests y el debugging.

Importante: Tenga en cuenta los siguientes requisitos al realizar los ejercicios. Piense en un diseño para su aplicación y realice un diagrama de clases del diseño propuesto antes de comenzar a codificar. Mantenga el diagrama de clases actualizado con el diseño de su aplicación durante todo el ejercicio. Asegúrese que su diseño adhiere (en la medida de lo posible) a los principios de diseño vistos en la teoría. Utilice la metodología de TDD.

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?
d) Defina la clase `DuckSimulator`, que tenga un método `simulate` que primero pone a volar a todos los patos del simulador, y luego hace que todos hagan cuac.

Ejercicio 2. 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 al menos dos implementaciones distintas del algoritmo de cómputo de primos.
b) Averigüe cómo usar tests parametrizados en JUnit, y modifique sus tests de modo que sirvan para testear las distintas implementaciones de cómputo de primos.
c) 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.
d) Utilice el patrón Strategy para refactorizar su código para que admita distintas formas de mostrar la salida, incluyendo mostrar la salida por pantalla y guardarla en un archivo.
e) Cree un mock para la salida para testear que el programa produce la salida correcta.
f) Refactorice usando el patrón Strategy para que su código admita al menos dos formas distintas de recibir la entrada, incluyendo obtenerla del usuario por consola, y leer la entrada desde un archivo.
g) Cree un mock para la entrada y testee que para algunas entradas dadas el programa produce la salida correcta.

Ejercicio 3. Desarrolle una aplicación para jugar al Conway's Game of Life. La descripción inicial del juego se puede encontrar en [0].

Notar que, las reglas del Juego de la Vida estándar se representan como un String como B3/S23. Esto es, una célula nace si tiene exactamente tres vecinos (B3), sobrevive si cuenta con dos o tres vecinos vivos (S23), y muere en cualquier otro caso. Es decir, el primer número (o lista de números) indica los requisitos para que una célula muerta nazca. El segundo conjunto define las condiciones para que una célula viva sobreviva a la siguiente generación.

a) Ahora queremos soportar en la misma aplicación distintas variantes de reglas del Game of Life. Aplique el patrón Strategy a la implementación de las reglas. Luego implemente la versión del juego B36/S23, denominada High Life, sólo agregando nuevas clases. Implemente además alguna otra variante de su elección. Puede tomar ejemplos de [2].
b) La próxima iteración consiste en soportar distintos colores de células en el Game of Life. Aplique el patrón Strategy a la implementación de los colores de células. Modifique la implementación para que las células puedan tener distintos esquemas de colores. Agregue nuevas células con nuevos colores agregando nuevas clases para las mismas.

c) Finalmente, deseamos soportar variantes del Game of Life donde las células puedan cambiar de color, y los colores de las células cercanas determinan el color de las células que nacen. Por ejemplo, Immigration es una variante con las mismas reglas del Juego de la Vida clásico (B3/S23), con la diferencia clave de que utiliza dos estados de activación (comúnmente representados mediante dos colores distintos). Cuando una nueva célula nace, adopta el estado de activación que sea mayoritario entre las tres células progenitoras que determinaron su nacimiento. Aplique el patrón Strategy para permitir cambiar la estrategia de selección de colores de las células que nacen. Implemente la versión Immigration del Game of Life, sólo agregando clases para la estrategia de selección de colores.

d) QuadLife es otra variante similar que utiliza cuatro estados de activación distintos. Cuando una nueva célula nace a partir de tres vecinas con estados diferentes, adopta el cuarto valor disponible. En otro caso (vecinas con el mismo estado) hereda el valor mayoritario. Implemente QuadLife sólo agregando nuevas clases.

[0] https://codingdojo.org/kata/GameOfLife/
[1]https://github.com/fabricejeannet/kataGameOfLife/blob/master/src/main/java/GameOfLife.java
[2] https://en.wikipedia.org/wiki/Life-like_cellular_automaton