Examen Parcial de Desarrollo de software

Notas:

- Duración de la prueba: 3 horas
- Entrega: construye una carpeta dentro de tu repositorio personal en Github que se llame ExamenParcial-3S2 donde se incluya todas tus respuestas cada una dividida por carpetas. No olvides colocar un Readme para indicar el orden de tus respuestas y procedimientos.
- Construye un proyecto de IntelliJ Idea para todos tus códigos.
- Sube un documento PDF de tus respuestas como respaldo a la plataforma,
- **No se admiten imágenes sin explicaciones.**
- Evita copiar y pegar cualquier información de internet. Cualquier acto de plagio anula la evaluación.

Pregunta 1 (3 puntos) SOLID y refactorización

Tenemos un club de ajedrez que acepta tres tipos de miembros: Premium, VIP y Free. Tenemos una clase abstracta llamada Member que actúa como clase base y tres subclases: PremiumMember, VipMember y FreeMember.

Veamos si cada uno de estos tipos de miembros puede sustituir a la clase base.

La clase Member es abstracta y representa la clase base para todos los miembros del club de ajedrez:

```
public abstract class Member {
private final String nombre;
public Member(String nombre) {
        this.name = nombre;
}
        public abstract void joinTournament();
        public abstract void organizeTournament();
}
```

La clase PremiumMember puede unirse a torneos de ajedrez u organizar tales torneos también. Entonces, su implementación es bastante simple:

```
public class PremiumMember extends Member {
public PremiumMember(String nombre) {
        super(nombre);
}
@Override
```

…. completa

La clase VipMember es más o menos la misma que PremiumMember. Ahora vamos a la clase FreeMember. La clase FreeMember puede unirse a torneos, pero no puede organizar torneos. Este es un problema que debemos abordar en el método OrganizeTournament(). Podemos lanzar una excepción con un mensaje significativo o podemos mostrar un mensaje.

```
public class FreeMember extends Member {
public FreeMember(String name) {
   super(nombre);
 }

 @Override
          public void joinTournament() {
System.out.println(“.....");
  }
    //Este método rompe LSP
@Override
public void organizeTournament() {
System.out.println("... ");
    }
}
```

Pero lanzar una excepción o mostrar un mensaje no significa que sigamos LSP. Dado que un free member no puede organizar torneos, no puede ser un sustituto de la clase base, por lo tanto, rompe el LSP. Consulta la siguiente lista de miembros:

```
List<Member> miembros = List.of(
 PremiumMember("Abejita Azul"),
  new VipMember("Kaperucita Feliz"),
   new FreeMember("Inspectora Motita")
);
```

Escribe código que indica que código no es compatible con LSP porque cuando la clase FreeMember tiene que sustituir a la clase Member, no puede realizar su trabajo ya que FreeMember no puede organizar torneos de ajedrez.

Esta situación es un escándalo. No podemos continuar con la implementación de la aplicación. Rediseña la solución para obtener un código compatible con LSP a través de un proceso de refactorización.

Debes presentar código antes y después de usar la refactorización para tener código compatible con LSP.

Pregunta 2 (6 puntos) RGR en Tic-Tac-Toe

El ejercicio consiste en la creación de una única prueba que corresponda a uno de los requisitos. La prueba es seguida por el código que cumple con las expectativas de esa prueba. Finalmente, si es necesario, el código se refactoriza. El mismo procedimiento debe repetirse con más pruebas relacionadas con el mismo requisito.

Una vez que estemos contentos con las pruebas y la implementación de ese requisito, pasaremos a la siguiente hasta que estén todas listas. En situaciones del mundo real, no obtendría requisitos tan detallados, pero te sumergirá directamente en las pruebas que actuarían como requisitos y validación.
Sin embargo, hasta que se sienta cómodo con TDD, tendremos que definir los requisitos por separado de las pruebas. Lee solo un requisito a la vez y escribe las pruebas y el código de implementación. No hay una y solo una solución, el tuyo podría ser mejor.

Requisito 1: colocación de piezas

Deberíamos comenzar definiendo los límites y lo que constituye una colocación no válida de una pieza. Se puede colocar una pieza en cualquier espacio vacío de un tablero de 3×3. Podemos dividir este requisito en tres pruebas:

- Cuando una pieza se coloca en cualquier lugar fuera del eje x, se lanza RuntimeException
- Cuando una pieza se coloca en cualquier lugar fuera del eje y, se lanza RuntimeException
- Cuando una pieza se coloca en un espacio ocupado, se lanza RuntimeException

Estas pruebas relacionadas con este primer requisito tienen que ver con las validaciones del argumento de entrada. No hay nada en los requisitos que diga qué se debe hacer con esas piezas.

Usa nombres descriptivos para los métodos de prueba.

Debe quedar claro qué condiciones se establecen antes de la prueba, qué acciones se realizan y cuál es el resultado esperado. Hay muchas maneras diferentes de nombrar los métodos de prueba. El método es nombrarlos usando la sintaxis given/when/then que se usa en los escenarios de BDD. Given describe (pre)condiciones, When describe acciones y Then describe el resultado esperado. Si una prueba no tiene precondiciones (normalmente establecidas mediante las anotaciones @Before y @BeforeClass), se puede omitir Given.

Utiliza Gradle y ejecuta en el símbolo del sistema:

$ gradle test

IntelliJ IDEA proporciona un muy buen modelo de tareas de Gradle al que se puede acceder haciendo clic en View|Tool Windows|Gradle.

Prueba: límites del tablero I

Escribe la prueba para comprobar si una pieza está colocada dentro de los límites del 3x3. Cuando una pieza se coloca en cualquier lugar fuera del eje x, se lanza RuntimeException. En esta prueba, estamos definiendo que se espera RuntimeException cuando se invoca el método ticTacToe.jugar(5, 2).

Todo lo que tenemos que hacer es crear el método jugar y asegurarnos de que arroja RuntimeException cuando el argumento x es menor que 1 o mayor que 3 (el tablero es 3x3). Debes ejecutar esta prueba tres veces. La primera vez, debería fallar porque el método jugar no existe. Una vez que se agrega, debería fallar porque no se lanza RuntimeException. La tercera vez, debería tener éxito porque el código que corresponde a esta prueba está completamente implementado. Muestra esto.

Implementación

Ahora que tienes una definición clara de cuándo debe lanzarse una excepción, realiza la implementación que debería ser sencilla.

Por ahora, estamos haciendo pasos rojo-verde. No hay mucho que podamos hacer para mejorar este código, por lo que nos saltamos la refactorización.

Prueba: límites del tablero II

Escribe esta prueba que es casi igual a la anterior. Esta vez debemos validar el eje y.
Cuando una pieza se coloca en cualquier lugar fuera del eje y, se lanza RuntimeException.

Implementación

Realiza la implementación de esta especificación es casi la misma que la anterior. Todo lo que se tiene que hacer es lanzar una excepción si y no cae dentro del rango definido.

Prueba - lugar ocupado

Ahora que sabemos que las piezas se colocan dentro de los límites del tablero, debemos asegurarnos de que solo se puedan colocar en espacios desocupados. Escribe una prueba para este caso.

Eso es todo; esta fue la última prueba. Una vez terminada la implementación, podemos dar por terminado el primer requisito.

Implementación

Para implementar la última prueba, debemos almacenar la ubicación de las piezas colocadas en un arreglo. Cada vez que se coloca una nueva pieza, debemos verificar que el lugar no esté ocupado, o de lo contrario lanzar una excepción.

Estamos comprobando si un lugar que se jugó está ocupado y, si no lo está, estamos cambiando el valor de entrada del arreglo de vacío (\0) a ocupado (X). Ten en cuenta que aún no almacenamos quién jugó (X u O).

Refactorización

Si bien el código que hemos hecho hasta ahora cumple con los requisitos establecidos por las pruebas, parece un poco confuso. Si alguien lo leyera, no quedaría claro qué hace el método jugar. Refactoriza moviendo el código a métodos separados.

Requisito 2: agregar soporte para dos jugadores

Ahora es el momento de trabajar en la especificación de qué jugador está a punto de jugar su turno.

Debería haber una manera de averiguar qué jugador debería jugar a continuación

Podemos dividir este requisito en tres pruebas:

- El primer turno lo debe jugar el jugador X.
- Si el último turno fue jugado por X, entonces el próximo turno debe ser jugado por O
- Si el último turno fue jugado por O, entonces el próximo turno debe ser jugado por X

Escribe la prueba antes de escribir el código de implementación. Un beneficio adicional es que con las pruebas primero, evitamos el peligro de que las pruebas funcionen como control de calidad en lugar de garantía de calidad.

Prueba – X juega primero

Escribe la prueba para ese caso.

Esta prueba debe explicarse por sí misma. Esperamos que el método proximoJugador devuelva X. Si intentas ejecutar esto, verás que el código ni siquiera se compila. Eso es porque el método proximoJugador ni siquiera existe.Tu trabajo es escribir el método y asegurarte de que devuelva el valor correcto.

Implementación

No hay una necesidad real de verificar si realmente es el primer turno del jugador o no. Tal como está, esta prueba se puede cumplir devolviendo siempre X. Las pruebas posteriores nos obligarán a refinar este código.

Prueba: O juego justo después de X

Escribe la prueba para asegurarte de que los jugadores estén cambiando. Después de que X haya terminado, debería ser el turno de O, luego nuevamente X, y así sucesivamente.
Si el último turno fue jugado por X, entonces el próximo turno debe ser jugado por O.

Implementación

Escribe código para rastrear quién debería jugar a continuación y almacenar quién jugó por última vez.

Las pruebas son pequeñas y fáciles de escribir. Con suficiente experiencia, debería tomar un minuto, si no segundos, escribir una prueba y tanto tiempo o menos escribir la implementación.

Prueba: X juega justo después de O

Finalmente, podemos verificar si el turno de X llega después de que O jugó. Si el último turno fue jugado por O, entonces el próximo turno debe ser jugado por X.
No hay nada que hacer para cumplir con esta prueba y, por lo tanto, la prueba es inútil y debe desecharse. Si escribes esta prueba, descubrirás que es un falso positivo. ¿como=
Pasaría sin cambiar la implementación; Pruébalo. Escriba esta prueba y, si tienes éxito sin escribir ningún código de implementación, deséchela. Muestra esto.

Requisito 3: agregar condiciones ganadoras

Es hora de trabajar para ganar de acuerdo con las reglas del juego. Esta es la parte donde, en comparación con el código anterior, el trabajo se vuelve un poco más tedioso. Deberíamos comprobar todas las posibles combinaciones ganadoras y, si se cumple una de ellas, declarar al ganador.

Un jugador gana al ser el primero en conectar una línea de piezas amigas de un lado o esquina del tablero al otro.

Para verificar si una línea de piezas amigas está conectada, debemos verificar las líneas horizontales, verticales y diagonales.

Prueba: por defecto no hay ganador

Comencemos definiendo la respuesta predeterminada del método jugar: Si no se cumple ninguna condición ganadora, entonces no hay ganador. Realiza esta prueba.

Implementación

Los valores de retorno predeterminados siempre son los más fáciles de implementar y este no es una excepción.

Prueba – condición ganadora I

Ahora que hemos declarado cuál es la respuesta predeterminada (No ganador), es hora de comenzar a trabajar en diferentes condiciones ganadoras.

El jugador gana cuando toda la línea horizontal está ocupada por sus piezas. Realiza esta prueba.

Implementación

Para cumplir con esta prueba, debemos verificar si alguna línea horizontal está llena con la misma marca que el jugador actual. Hasta este momento, no nos importaba lo que se pusiera en el arreglo del tablero.
Ahora, necesitamos introducir no solo qué casillas del tablero están vacías, sino también qué jugador las jugó. Realiza la implementación.

Refactorización

El código anterior satisface las pruebas, pero no es necesariamente la versión final. Cumplió su propósito de obtener cobertura de código lo más rápido posible. Ahora, como tenemos pruebas que garantizan la integridad del comportamiento esperado, podemos refactorizar el código. Realiza esa refactorización.

Esta solución refactorizada se debe ver mejor.

Prueba – condición ganadora II

También debemos verificar si hay una ganancia llenando la línea vertical.

El jugador gana cuando toda la línea vertical está ocupada por sus piezas. Realiza esta prueba.

Implementación

Esta implementación debe ser similar a la anterior. Ya tenemos verificación horizontal y ahora necesitamos hacer lo mismo verticalmente.

Prueba – condición ganadora III

Ahora que las líneas horizontales y verticales están cubiertas, debemos centrar la atención en las combinaciones diagonales.

El jugador gana cuando toda la línea diagonal desde la parte superior izquierda hasta la parte inferior derecha está ocupada por sus piezas.
Implementación

Dado que solo hay una línea que puede constituir el requisito, podemos verificarlo directamente sin bucles. Realiza esta implementación.

Prueba – condición ganadora IV

Finalmente, existe la última condición ganadora posible para abordar: El jugador gana cuando toda la línea diagonal desde la parte inferior izquierda hasta la parte superior derecha está ocupada por sus piezas.

Implementación

La implementación de esta prueba debería ser casi la misma que la anterior.

Refactorización

Por la forma en que manejamos las posibles victorias en diagonal, el cálculo no parece correcto. Quizás la reutilización del bucle existente tendría más sentido. Realiza la refactorización.

Ahora, repasemos el último requisito.

Requisito 4: condiciones de empate

Lo único que falta es cómo abordar el resultado del empate. El resultado es un empate cuando se llenan todas las casillas.

Prueba: manejo de una situación de empate

Podemos probar el resultado del juego llenando todas las casillas del tablero. Realiza esta prueba.

Implementación

Comprobar si es un empate es bastante sencillo. Todo lo que tenemos que hacer es verificar si todas las casillas del tablero están llenas. Podemos hacer eso iterando a través de un arreglo del tablero.

Refactorización

No necesitamos verificar todas las combinaciones, sino solo aquellas relacionadas con la posición de la última pieza tocada. Realiza la refactorización. Utiliza un método esGanador, pese que no está al alcance de la última prueba.

La refactorización se puede realizar en cualquier parte del código en cualquier momento, siempre que todas las pruebas sean exitosas. Si bien a menudo es más fácil y rápido refactorizar

el código que se acaba de escribir, volver a algo que se escribió el otro día, el mes anterior o incluso hace años es más que bienvenido.

El mejor momento para refactorizar algo es cuando alguien ve la oportunidad de mejorarlo. No importa quién lo escribió o cuándo; mejorar el código siempre es algo bueno.

Cobertura de código

No usamos herramientas de cobertura de código a lo largo de la clase. La razón es que queríamos que te centraras en el modelo Red-Green-Refactor. Has escrito una prueba, vio que fallaba, escribió el código de implementación, vio que todas las pruebas se ejecutaron con éxito, refactorizaste el código cada vez que se vio la oportunidad de mejorarlo y luego repitió el proceso. ¿Nuestras pruebas cubrieron todos los casos?

Eso es algo que las herramientas de cobertura de código como JaCoCo pueden responder. ¿Deberías usar esas herramientas? Probablemente, sólo al principio.

Cuando comiences con TDD, probablemente te perderás algunas pruebas o implementarás más de lo que definieron las pruebas. En esos casos, usar la cobertura de código es una buena manera de aprender de sus propios errores. Más adelante, cuanto más experiencia tengas con TDD, menos necesidad tendrás de tales herramientas.

Escribirás pruebas y solo lo suficiente del código para que pasen. Tu cobertura será alta con o sin herramientas como JaCoCo. Habrá una pequeña cantidad de código que no cubrirán las pruebas porque tomará una decisión consciente sobre lo que no vale la pena probar.

Las herramientas como JaCoCo se diseñaron principalmente como una forma de verificar que las pruebas escritas después del código de implementación brindan suficiente cobertura. Con TDD, estamos adoptando un enfoque diferente con el orden invertido (pruebas antes de la implementación).

Para habilitar JaCoCo dentro de Gradle, agrega lo siguiente a build.gradle:

apply plugin: 'jacoco'

A partir de ahora, Gradle recopilará métricas de JaCoCo cada vez que ejecutemos pruebas. Esas métricas se pueden transformar en un buen informe utilizando el jacocoTestReport. Ejecuta tus pruebas nuevamente indica cuál es la cobertura del código.

Los resultados variarán dependiendo de la solución que hayas hecho para este ejercicio.

Pregunta 3 (5 puntos)

Aplica el proceso RGR dado anteriormente en el juego SOS implementado en los cuatro sprints hasta el momento..

Pregunta 4 (6 puntos)

(a) ¿Qué son las pruebas efectivas y sistemáticas? Referencia: https://www.effective-software-testing.com/effective-and-systematic **(1 punto)**

(b) Pruebas (2 puntos)

Dada esta especificación:

```
/**
* Dividir una cadena en un carácter delimitador.
*
*  @param texto          una cadena
* @param delimitador          un delimitador por el cual dividir la cadena
 * @param límite                un límite superior en el número de elementos a devolver:
                          si límite < 0, no hay límite superior; límite != 0
* @retorna  una  lista de cadenas [s1, s2, ..., sN] tales que:
*          - texto = s1 + delimitación + s2 + delimitación + ... + delimitación + sN
*           - N <= límite si límite > 0
*          - none de s1, s2, ..., sN contiene delimitador
*  @throws IllegalArgumentException si límite > 0
*
*          y hay más de limite-1 ocurrencias de delimitador en el texto.
* /
```

public static List<String> split(String text, char delim, int limit);

(a) Comienza a implementar una estrategia de prueba sistemática para esta función escribiendo una buena partición del espacio de entrada solo en el límite de entrada, es decir, la partición no debe mencionar ni el texto ni el delimitador.

(b) Ahora, escriba una buena partición del espacio de entrada sobre la relación entre el límite y las ocurrencias del delimitador en el texto. Tu partición debe mencionar las tres entradas.

Refactorización avanzada (3 puntos)

El objetivo es cambiar la clase actual para que coincida con la nueva API y luego crear una clase que envuelva la primera y proporcione la API anterior. La estrategia no es tan diferente de lo que hicimos anteriormente, solo que esta vez trataremos con clases en lugar de métodos. Con un estupendo esfuerzo de mi imaginación nombré a la nueva clase NDataStats.

Lo primero, como sucede muy a menudo con la refactorización es duplicar el código y cuando insertamos código nuevo necesitamos tener pruebas que lo justifiquen.

Las pruebas serán las mismas que antes, ya que la nueva clase proporcionará las mismas funcionalidades que la anterior, así que solo crea un nuevo archivo, llamado test_ndatastats.py y allí se coloca la primera prueba test_init().

```
import json

from datastats import NDataStats
test_data = [
        {
        "id": 1,
        "nombre": "Irene",
        "apodo": "Lara",
        "edad": 68,
        "salario": "$27888"
        },

        {
        "id": 2,
        "nombre": "Claudio",
        "apodo": "Avila",
        "edad": 49,
        "salario": "$67137"
        },
        {
        "id": 3,
        "nombre": "Tomo",
        "apodo": "Frugs",
        "edad": 70,
        "salario": "$70472"
        }

]

def test_init():
  ds = NDataStats(test_data)
  assert ds.data == test_data
```

Pregunta: Escribe la nueva clase NdataStats en el archivo datastats.py y comprueba si la prueba anterior pasa.

Ahora iniciamos un proceso iterativo:

- Copia una de las pruebas de `DataStats` y adapta a `NDataStats`
- Copia algo de código de `DataStats` a `NDataStats`, adaptándolo a la nueva API y haciendo pasar una prueba.

En este punto, eliminar iterativamente métodos de `DataStats` y reemplazarlos con una llamada a `NDataStats` es una exageración.

Un ejemplo de las pruebas resultantes para `NDataStats` es el siguiente:

```
#def test_edad():
# ds = NDataStats(test_data)
# assert ds._edades() == [68, 49, 70]
```

Y el código que pasa la prueba para este caso es:

```
#def _edades(self):
#    return [d['edad'] for d in self.data]
```

Los métodos como `_edades()` ya no requieren un parámetro de entrada y se pueden convertir convertirlos en propiedades, cambiando las pruebas con `@property`.

Es hora de reemplazar los métodos de `DataStats` con llamadas a `NDataStats`. Podríamos hacerlo método por método, pero en realidad lo único que realmente necesitamos es reemplazar `stats()`. Así que el nuevo código es:

```
#class DataStats:
# def stats(self, data, iedad, isalario):
#     nds = NDataStats(data)
#     return nds.stats(iedad, isalario)
```

Pregunta: Edita los archivos necesarios para ejecutar la prueba. ¿Cuál es el código de cobertura para este ejemplo ?.

Pregunta: La refactorización es un proceso iterativo, a menudo sucederá que crees que hiciste todo lo posible, solo para descubrir más tarde que te perdiste algo. En este caso, el paso faltante es una pequeña duplicación de código.

Las dos funciones comparten la misma lógica, por lo que definitivamente podemos aislar esto y llamar al código común en cada función.

```
#def _salario_promedio(self):
        # return math.floor(sum(self._salarios)/len(self.data))

#def _edad_promedio(self):
        # return math.floor(sum(self._edades)/len(self.data))

#def _promedio_floor(self, suma_de_numberos):
        # return math.floor(suma_de_numeros / len(self.data))

#def _salario_promedio(self):
```

```
        # return self._promedio_floor(sum(self._salarios))

#def _edad_promedio(self):
        # return self._promedio_floor(sum(self._edades))
```

Edita los archivos necesarios para ejecutar la prueba. ¿Cuál es el código de cobertura para este ejemplo.?