Curso de desarrollo de software

Práctica Calificada 4

Instrucciones

- Construye un repositorio llamado PracticaCalificada3 donde coloques tus respuestas - Evita copiar y pegar de otros repositorios. Cualquier evidencia de copia implica la anulación de la evaluación.
- Evita solo copiar imágenes como respuesta y añade un archive Readme en formato markdown que indique como se han desarrollado cada una de las preguntas. La presentación de código sin explicación afecta tu calificación.
- Debes entregar en la plataforma del curso, el URL donde se alojan todas tus respuestas.

Prueba 1 (8 puntos)

Sea el ejemplo "contar clumps", inspirado en una tarea de CodingBat: https://codingbat.com/prob/p193817): El programa debe contar el número de clumps en un arreglo. Un clumps es una secuencia del mismo elemento con una longitud de al menos 2.

nums: el arreglo para la que contar los grupos. El arreglo debe ser no nula y de longitud > 0; el programa devuelve 0 si se viola alguna pre-condición.

El programa devuelve el número de clumps en el arreglo.

Pregunta 1 (2 puntos): Escribe una implementación del problema dado

En la implementación se tiene un arreglo de numeros. Si el arreglo es null o vacío retorna 0. Si el arreglo tiene longitud 1 devuelve 0 porque no puede haber clumps. Luego se evalua el caso de un arreglo de 2 elementos. Finalmente si el arreglo tiene 3 elementos se verifica si los dos primeros elementos son iguales y a partir del tercer elemento se cuenta cada nuevo clump cuando el elemento anterior es el mismo pero al anterior a este diferente.

```
public class ContarClumps {
    public int contarClumps(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }
        if (nums.length == 1) {
            return 0;
        }
        if(nums.length == 2){
            if(nums[0] == nums[1]){
                return 1;
            } else {
```

```
                return 0;
            }
        }
        int clumps = 0;
        if(nums[0] == nums[1]){
            clumps++;
        }
        for (int i = 2; i < nums.length; i++) {
            if (nums[i] == nums[i - 1] && nums[i] != nums[i-2]) {
                clumps++;
            }
        }
        return clumps;
    }
}
```

Supongamos que decidimos no mirar los requisitos. Queremos lograr, digamos, el 100% de cobertura de ramas.
Pregunta 2 (1 puntos) Escribe tres pruebas para hacer eso (T1-T3). También agrega algunas pruebas de límites adicionales (T4):

Se pide realizar pruebas para lograr cobertura de ramas por lo que se debera cubrir que cada la condición en cada if sea al menos una vez verdadera y una vez falsa. Para ello se requiere 6 pruebas:
- T1: longitud de nums es cero por lo que la condición del primer if es verdadera
- T2: longitud de nums es uno por lo que la condición del primer if es falsa y la condición del segundo if es verdadera.
- T3: longitud de nums es dos y tiene elementos iguales. En este caso la condición del segundo if es falsa, la condición del tercer y cuarto if son verdaderas.
- T4: longitud de nums es dos y tiene elementos diferentes. En este caso la condición del tercer if es verdadera y el cuarto if es falsa.
- T5: longitud de nums es tres y los dos primeros elementos son iguales y el tercero diferente. En este caso la condicion del tercer if es falsa y del quinto if es verdadera y del sexto if es falsa.
- T6: longitud de nums es tres y los dos últimos elementos son iguales y el primero diferente. En este caso la condicion del quinto if es falsa y del sexto if es verdadera.

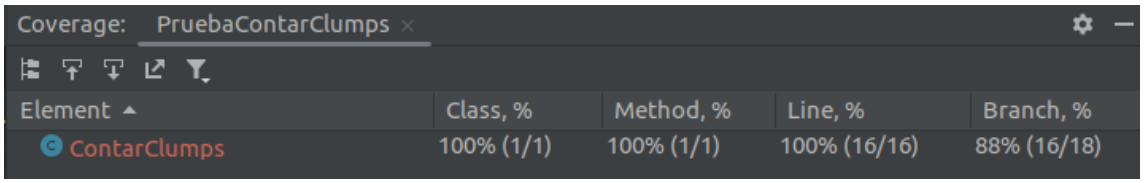
Pregunta 3 (2 puntos) Anota estas tres pruebas como casos de prueba automatizados (JUnit) y ejecuta la herramienta de cobertura de código favorita. Muestra los resultados

Escribiendo las pruebas descritas en la pregunta anterior se obtiene:

```
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;

public class PruebaContarClumps {
    @Test
    public void arregloLongitudCeroONulo() {
        ContarClumps contarClumps = new ContarClumps();
```

```
        int[] nums = new int[0];
        int clumps = contarClumps.contarClumps(nums);
        assertEquals(0, clumps);
    }
    @Test
    public void arregloLongitudUno() {
        ContarClumps contarClumps = new ContarClumps();
        int[] nums = {5};
        int clumps = contarClumps.contarClumps(nums);
        assertEquals(0, clumps);
    }
    @Test
    public void arregloLongitudDosConElementosIguales() {
        ContarClumps contarClumps = new ContarClumps();
        int[] nums = {5, 5};
        int clumps = contarClumps.contarClumps(nums);
        assertEquals(1, clumps);
    }
    @Test
    public void arregloLongitudDosConElementosDiferentes() {
        ContarClumps contarClumps = new ContarClumps();
        int[] nums = {5, 7};
        int clumps = contarClumps.contarClumps(nums);
        assertEquals(0, clumps);
    }
    @Test
    public void
arregloLongitudTresPrimerosDosElementosIgualesTerceroDiferente()
{
        ContarClumps contarClumps = new ContarClumps();
        int[] nums = {5, 5, 7};
        int clumps = contarClumps.contarClumps(nums);
        assertEquals(1, clumps);
    }
    @Test
    public void
arregloLongitudTresUltimosDosElementosIgualesPrimeroDiferente()
{
        ContarClumps contarClumps = new ContarClumps();
        int[] nums = {5, 7, 7};
        int clumps = contarClumps.contarClumps(nums);
        assertEquals(1, clumps);
    }
}
```

Ejecutando la cobertura de código con Jacoco se obtiene:



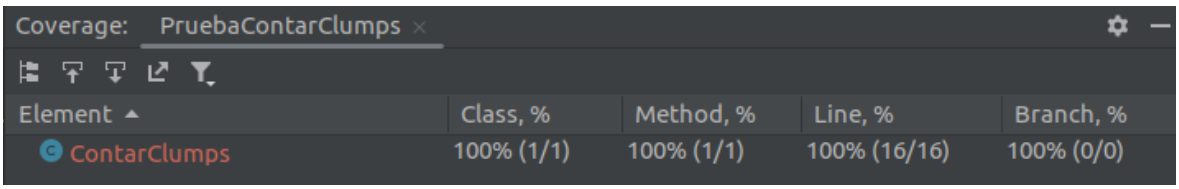
Por lo que se observa que la cobertura de rama ha sido del 88%.

Pregunta 4 (3 puntos) En la resolución de las preguntas dadas anteriormente se pierde muchos casos de prueba interesantes. Incluso sin realizar pruebas de especificación sistemáticas, en un programa que cuenta clumps, es natural probar el programa con varios clumps en lugar de uno solo. Experimenta con el último clump en el último elemento del arreglo o con un conjunto que tiene un clump que comienza en la primera posición. ¿Qué pasa con el código cobertura de ramas?.

Se crea una prueba según lo pedido con un arreglo que tiene varios clumps y el último clump al final del arreglo.

```
@Test
public void arregloConClumpAlFinal() {
    ContarClumps contarClumps = new ContarClumps();
    int[] nums = {1, 2, 2, 3, 4, 4, 7, 9, 9};
    int clumps = contarClumps.contarClumps(nums);
    assertEquals(3, clumps);
}
```

Esta vez se obtiene una cobertura de ramas del 100%:



Pregunta 2 (5 puntos)

En esta pregunta, aplicaremos lo que hemos aprendido escribiendo una prueba para una clase que elegirá una palabra al azar para que el jugador la adivine, de un conjunto de palabras almacenadas. Utiliza la aplicación Wordz realizada en actividades anteriores para crear una interfaz llamada WordRepository para acceder a las palabras almacenadas. Realiza esto con un método fetchWordByNumber(wordNumber), donde wordNumber identifica una palabra. La decisión de diseño aquí es que cada palabra se almacene con un número secuencial a partir de 1 para ayudarnos a elegir una al azar.

Escribe una clase WordSelection, que sea responsable de elegir un número aleatorio y usarlo para obtener una palabra del almacenamiento que está etiquetada con ese número. Para este ejercicio, la prueba cubrirá el caso en el que intentamos obtener una palabra de la interfaz de WordRepository, pero por alguna razón, no está allí.

El esqueleto de la prueba es:

...
public class WordSelectionTest {

private WordRepository repository;

...
public void reportsWordNotFound() {

```
 doThrow(new WordRepositoryException())
 .when(repository)
 .fetchWordByNumber(anyInt());
...
 assertThatExceptionOfType(WordSelectionException.class)
 .isThrownBy(
 ()->selection.getRandomWord());
 }
}
```

Preguntas:

1. Construye el SUT , completa y explica el código de prueba generado.
2. Indica los pasos Act y Assert de la prueba.
3. Ejecuta la prueba. ¿Pasa la prueba?. Agrega la lógica necesaria para que la prueba pase. 4. Indica el código de cobertura obtenido y compara estos resultados con las actividades anteriores de la aplicación Wordz. ¿Qué diferencias encuentras?.
5. Muestra en que parte de tu código podemos usar dobles de prueba y la verificación de condiciones de error con TDD . Indica algunas decisiones de diseño en esta prueba sobre qué excepciones ocurren y dónde se usan.

Pregunta 3 (7 puntos)

Tu tarea es escribir una versión (muy) simplificada de un sistema de reservas. De hecho, se puede escribir como una sola clase, que debería:

- devolver una lista de horas reservadas
- no permitir que una hora en particular se reserve dos veces
- tratar de forma sensata los valores ilegales (proporcionados como

parámetros de entrada). Por el lado de las restricciones (para que la tarea sea

más apropiada para practicar TDD), el sistema:

- tiene solo un recurso que se puede reservar (por ejemplo una mesa de restaurante o cualquier otra cosa)
- se asume que todas las reservas son para hoy
- solo debes permitir la reserva de horas regulares de reloj completas (por ejemplo, no debe permitir una reserva de 4:30 p. m. a 5:30 p. m.)
- no estás obligado a recordar ninguna información adicional relativa a la reserva (quién la reservó, cuándo, etc.).

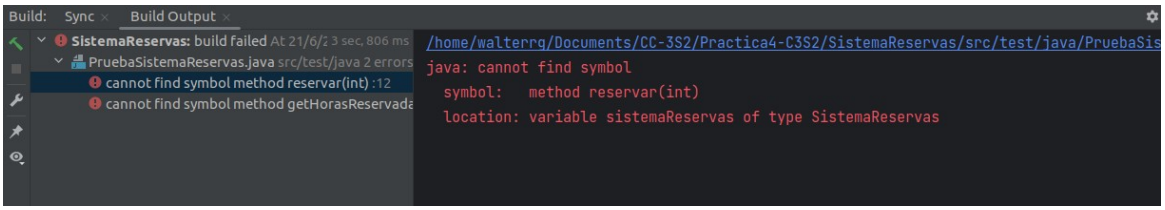
Solución Primera Parte:

Como piden que se realize con TDD primero se escribe la prueba considerando reservar una única hora válida.

```
@Test
public void reservaValida() {
    SistemaReservas sistemaReservas = new SistemaReservas();
    sistemaReservas.reservar(10);

    List<Integer> horasReservadasEsperadas = new
```

```
ArrayList<Integer>();
    horasReservadasEsperadas.add(10);

    List<Integer> horasReservadas =
sistemaReservas.getHorasReservadas();


assertThat(horasReservadas).isEqualTo(horasReservadasEsperadas);
}
```

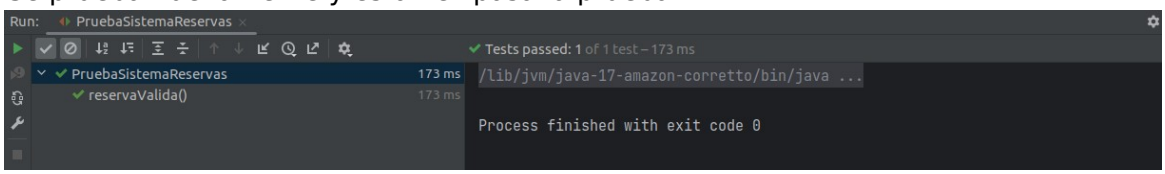
Resultado de la prueba da error porque aún no se implementa:



Se implementa la reserva:

```
public class SistemaReservas {
    List<Integer> horasReservadas = new ArrayList<>();

    public void reservar(int hora) {
        horasReservadas.add(hora);
    }

    public List<Integer> getHorasReservadas() {
        return horasReservadas;
    }
}
```

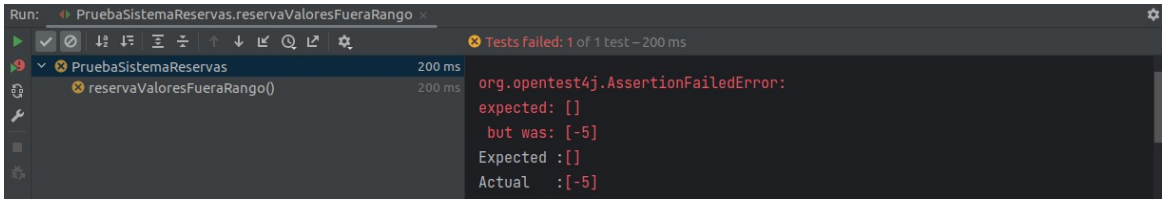
Se prueba nuevamente y esta vez pasa la prueba:



Ahora se prueba valores fuera del rango apropiado:

```
@Test
public void reservaValoresFueraRango() {
    SistemaReservas sistemaReservas = new SistemaReservas();
    sistemaReservas.reservar(-5);

    List<Integer> horasReservadasEsperadas = new ArrayList<>();

    List<Integer> horasReservadas =
sistemaReservas.getHorasReservadas();


assertThat(horasReservadas).isEqualTo(horasReservadasEsperadas);
}
```

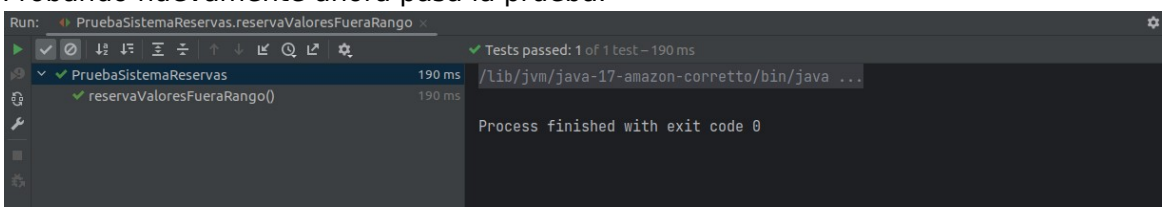
La prueba falla



Ahora se implementa que verifique las horas al reservar que se encuentren en el rango de 8 a 20.

```
public class SistemaReservas {
    List<Integer> horasReservadas = new ArrayList<>();

    public void reservar(int hora) {
        if(hora >= 8 && hora <= 20){
            horasReservadas.add(hora);
        }
    }

    public List<Integer> getHorasReservadas() {
        return horasReservadas;
    }
}
```

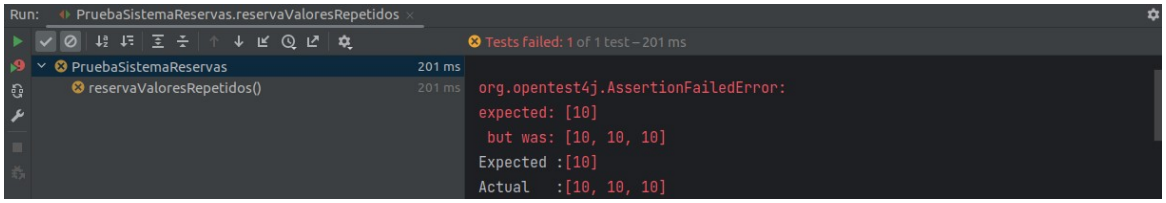
Probando nuevamente ahora pasa la prueba:



Ahora se prueba valores repetidos:

```
public void reservaValoresRepetidos() {
    SistemaReservas sistemaReservas = new SistemaReservas();
    sistemaReservas.reservar(10);
    sistemaReservas.reservar(10);
    sistemaReservas.reservar(10);

    List<Integer> horasReservadasEsperadas = new ArrayList<>();
    horasReservadasEsperadas.add(10);

    List<Integer> horasReservadas =
sistemaReservas.getHorasReservadas();


assertThat(horasReservadas).isEqualTo(horasReservadasEsperadas);
}
```

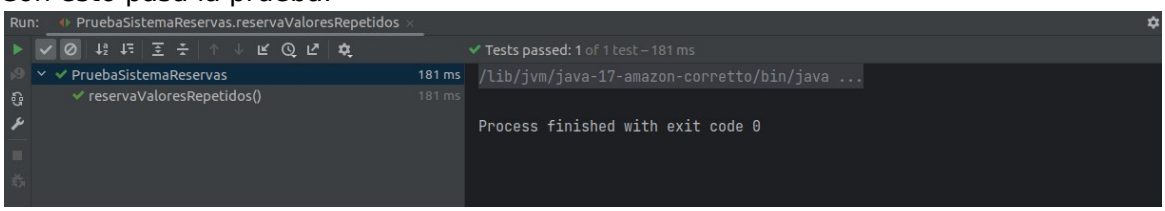
La prueba falla:



Se agrega la lógica para que no reserve horas repetidas:

```
public void reservar(int hora) {
    if(hora >= 8 && hora <= 20){
        if(!horasReservadas.contains(hora)){
            horasReservadas.add(hora);
        }
    }
}
```

Con esto pasa la prueba:



Ya ha escrito un sistema de reservas. Esta vez, se te pide que implementes una aplicación similar, pero probando con dobles de prueba. A continuación, encontrarás una descripción y algunos requisitos que te ayudarán a comenzar a codificar. Si se omiten algunos detalles, simplemente invéntelos durante el desarrollo.

Este sistema de reservas permite reservar aulas. Cada aula tiene una capacidad determinada (por ejemplo, para 20 personas) y puede equiparse con retroproyector, micrófono y pizarra. También tiene un "nombre" determinado (ID, número, como quieras llamarlo...). La API del sistema debe permitir:

- enumerar todas las aulas existentes
- enumerar todas las aulas disponibles (para un día determinado y un intervalo de tiempo por hora), - reservar un aula específica por nombre (p. ej., "Quiero reservar el aula A1": book("A1")) - reservar un aula específica especificando algunas restricciones (por ejemplo, "Quiero reservar un aula con un proyector para 20 personas": book(20, Equipment.PROJECTOR)).

(Opcional) Aquí hay algunas restricciones adicionales, por lo que el sistema no es demasiado complejo al principio:

- solo se admite la reserva periódica, lo que significa que puedes reservar, por ejemplo, el aula A1 para todos los viernes de 10 a 11 am, pero no solo para el viernes 13 de mayo.
- cada reserva tiene una duración de 1 hora; no se permiten períodos más largos o más cortos.

Una vez que haya implementado el sistema especificado anteriormente, usa tu imaginación y agrega algo más de complejidad. Por ejemplo:

- cada operación de reserva debe escribirse en los registros
- cada aula tiene una "hora de limpieza", cuando no está disponible
- el tiempo de reserva ya no está limitado a 1 hora

Solución Segunda Parte:

Se implementa la clase Aula:

```
import java.util.List;

public class Aula {
    String ID;
    int capacidad;
    List<Equipo> equipos;

    public Aula(String ID, int capacidad, List<Equipo> equipos) {
        this.ID = ID;
        this.capacidad = capacidad;
        this.equipos = equipos;
    }

    public String getID() {
        return ID;
    }

    public int getCapacidad() {
        return capacidad;
    }

    public List<Equipo> getEquipos() {
        return equipos;
    }
}
```

Se implementa la clase Reserva que representa una reserva:

```
public class Reserva {
    Aula aula;
    String dia;
    int hora;

    public Reserva(Aula aula, String dia, int hora) {
        this.aula = aula;
        this.dia = dia;
        this.hora = hora;
    }
}
```

Se implementa la enumeración Equipo:

```
public enum Equipo {
    PROYECTOR,
    MICROFONO,
    PIZARRA
```

```
}
```

Se implementa laclase SistemaReservaAulas, solo se ha implementado los métodos enumerarAulasExistentes y enumerarAulasDisponibles.

```
import java.util.List;

public class SistemaReservaAulas {
    List<Aula> aulasExistentes;
    List<Reserva> reservas;

    public SistemaReservaAulas(List<Aula> aulasExistentes) {
        this.aulasExistentes = aulasExistentes;
    }

    public String enumerarAulasExistentes() {
        StringBuilder resp = new StringBuilder();
        for (Aula aula : aulasExistentes) {
            resp.append(aula.getID()).append("-");
        }
        return resp.toString();
    }

    public String enumerarAulasDisponibles(String dia, int hora)
{
        StringBuilder resp = new StringBuilder();
        for (Aula aula : aulasExistentes) {
            if (!reservas.contains(new Reserva(aula, dia, hora)))
{
                resp.append(aula.getID()).append("-");
            }
        }
        return resp.toString();
    }

}
```