Curso de desarrollo de software

Práctica Calificada 5

Reglas de la evaluación

- Queda terminantemente prohibido el uso de herramientas como ChatGPT, WhatsApp, o cualquier herramienta similar durante la realización de esta prueba. El uso de estas herramientas, por cualquier motivo y teniendo en cuenta antecedentes estudiantes resultará en la anulación inmediata de la evaluación.
- Las respuestas deben presentarse con una explicación detallada, utilizando términos técnicos apropiados. La mera descripción sin el uso de terminología técnica, especialmente términos discutidos en clase, se considerará insuficiente y podrá resultar en que la respuesta sea marcada como incorrecta.
- Toda respuesta que incluya código debe ser acompañada de una explicación. La ausencia de esta explicación implicará que la pregunta se evaluará como cero. Asegúrense de indicar la parte que se está indica en comentares.
- Utiliza imágenes para explicar o complementar las respuestas, siempre y cuando estas sean de creación propia y relevantes para la respuesta, como es el caso de la verificación de los pasos de pruebas.
- Cada estudiante debe presentar su propio trabajo. Los códigos iguales o muy parecidos entre sí serán considerados como una violación a la integridad académica, implicando una copia, y serán sancionados de acuerdo con las políticas de la universidad.
- La claridad, orden, y presentación general de las evaluaciones serán tomadas en cuenta en la calificación final. Se espera un nivel de profesionalismo en la documentación y presentación del código y las respuestas escritas.

Instrucciones de entrega para la prueba calificada

Para asegurar una entrega adecuada y organizada de su prueba calificada, sigan cuidadosamente las siguientes instrucciones. El incumplimiento de estas pautas resultará en que su prueba no sea revisada, sin importar el escenario.

- Cada estudiante debe tener un repositorio personal en la plataforma designada para el curso (por ejemplo, GitHub, GitLab, etc.). El nombre del repositorio debe incluir su nombre completo o una identificación clara que permita reconocer al autor del trabajo fácilmente.
- Dentro de su repositorio personal, deben crear una carpeta titulada **PracticaCalificada4-CC3S2.** Esta carpeta albergará todas las soluciones de la prueba.
- Todas las respuestas y soluciones deben ser documentadas, completas y presentadas en archivos Markdown (.md) dentro de las subcarpetas correspondientes a cada ejercicio. Los archivos Markdown permiten una presentación clara y estructurada del texto y el código.
- Cada archivo Markdown de respuesta debe incluir el nombre del autor como parte del nombre del archivo. Esto es para asegurar la correcta atribución y reconocimiento del trabajo individual.
- No se aceptarán entregas en formatos como documentos de Word (.docx) o archivos PDF. La entrega debe cumplir estrictamente con el formato Markdown (.md) como se especifica.

- Es crucial seguir todas las instrucciones proporcionadas para la entrega de su prueba calificada. Cualquier desviación de estas pautas resultará en que su prueba no sea considerada para revisión.
- Antes de la entrega final, verifique la estructura de su repositorio, el nombramiento de las carpetas y archivos, y asegúrese de que toda su documentación esté correctamente formateada y completa.

Descripción del proyecto

El juego Tower Defense es un videojuego de consola donde el jugador debe defender su base de oleadas de enemigos colocando torres en lugares estratégicos del mapa. El proyecto incluirá el uso de mocks, stubs y fakes para pruebas unitarias y de integración utilizando Mockito y pruebas de mutación.

Objetivos del ejercicio:

1. Configurar y ejecutar contenedores Docker.

2. Configurar redes y volúmenes en Docker.

3. Usardocker exec para interactuar con contenedores en ejecución.

4. Implementar aplicaciones con Docker Compose.

5. Desplegar aplicaciones en Kubernetes.

6. Realizar pruebas unitarias y de integración utilizando Mockito.

7. Implementar pruebas de mutación para verificar la calidad de las pruebas.

Estructura del proyecto

Clases principales:

1.TowerDefenseGame: Clase principal que maneja la lógica del juego.

2.Map: Representa el mapa del juego.

3.Enemy: Clase base para todos los enemigos.

4.Tower: Clase base para todas las torres.

5.Wave: Maneja las oleadas de enemigos.

6.Player: Representa al jugador y sus estadísticas.

Entrada y salida

Entrada:

- Comandos del usuario para colocar torres, iniciar oleadas, etc.
- Datos iniciales del mapa y configuración de juego.

Salida:

- Estado del juego después de cada comando.
- Puntuación y estado de salud de la base.

Código en java para el juego

1. Clase PrincipalTowerDefenseGame

```
import java.util.*;


public class TowerDefenseGame {
    private Map map;
    private Player player;
    private List<Wave> waves;

    public TowerDefenseGame() {
        this.map = new Map();
        this.player = new Player();
        this.waves = new ArrayList<>();
    }

    public void placeTower(Tower tower, int x, int y) {
        map.placeTower(tower, x, y);
    }

    public void startWave() {
        Wave wave = new Wave();
        waves.add(wave);
        wave.start();
```

```
    }

    public void gameState() {
        System.out.println(map);
        System.out.println("Puntuación: " + player.getScore());
        System.out.println("Vida de la base: " + player.getBaseHealth());
    }
}
```

2. ClaseMap

```
public class Map {
    private char[][] grid;

    public Map() {
        grid = new char[5][5];
        for (int i = 0; i < 5; i++) {
            for (int j = 0; j < 5; j++) {
                grid[i][j] = ' ';
            }
        }
    }

    public void placeTower(Tower tower, int x, int y) {
        grid[x][y] = tower.getSymbol();
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
```

```
        for (char[] row : grid) {
            for (char cell : row) {
                sb.append("[").append(cell).append("]");
            }
            sb.append("\n");
        }
        return sb.toString();
    }
}
```

3. ClasePlayer

```
public class Player {
    private int score;
    private int baseHealth;

    public Player() {
        this.score = 0;
        this.baseHealth = 100;
    }

    public int getScore() {
        return score;
    }

    public int getBaseHealth() {
        return baseHealth;
    }
}
```

4. ClaseTower

```
public class Tower {
    private char symbol;

    public Tower(char symbol) {
        this.symbol = symbol;
    }

    public char getSymbol() {
        return symbol;
    }
}
```

5. ClaseWave

```
public class Wave {
    public void start() {
        System.out.println("Oleada iniciada!");
    }
}
```

Teoría de conceptos clave

Docker

Docker es una plataforma que permite a los desarrolladores automatizar la implementación de aplicaciones como contenedores portátiles, autónomos y ligeros. Los contenedores Docker pueden ejecutar aplicaciones en cualquier entorno, desde una computadora personal hasta servidores en la nube. A continuación se presentan algunos conceptos clave:

docker exec

El comandodocker exec permite ejecutar comandos dentro de un contenedor Docker en ejecución. Esto es útil para depuración, administración y mantenimiento. Por ejemplo:

docker exec -it tower-defense-container /bin/bash

Este comando abre una sesión interactiva de shell dentro del contenedor llamadotower-defense-container.

Redes en Docker

Docker proporciona la capacidad de crear redes aisladas para que los contenedores se comuniquen entre sí de manera segura y eficiente. La creación de una red personalizada se puede hacer con:

docker network create game-network

Y los contenedores pueden unirse a esta red mediante la opción--network al ejecutardocker run.

Volúmenes en Docker

Los volúmenes se utilizan para persistir datos generados y utilizados por los contenedores Docker. Los volúmenes son gestionados por Docker y pueden ser compartidos entre contenedores. Crear un volumen se hace con:

docker volume create game-data

Y se puede montar en un contenedor con la opción-v:

docker run -it --name tower-defense-container --network game-network -v game-data:/app/data tower-defense-game

Docker Compose

Docker Compose es una herramienta para definir y ejecutar aplicaciones Docker multi-contenedor. Con un archivo YAML, se pueden configurar los servicios, redes y volúmenes necesarios para la aplicación. Por ejemplo, un archivo docker-compose.yml para nuestro juego:

```
version: '3'

services:
  game:
    image: tower-defense-game
    networks:
      - game-network
    volumes:
      - game-data:/app/data

networks:
  game-network:
    driver: bridge

volumes:
  game-data:
```

```
  driver: local
```

Para iniciar los servicios definidos, se usa el comando:

docker-compose up -d

Docker Swarm

Docker Swarm es una herramienta nativa de Docker para la orquestación de contenedores, permitiendo la gestión de un clúster de Docker. Con Docker Swarm, se puede implementar, escalar y administrar aplicaciones distribuidas. A diferencia de Kubernetes, Docker Swarm es más fácil de configurar y manejar para proyectos más pequeños.

Kubernetes

Kubernetes es una plataforma de orquestación de contenedores de código abierto diseñada para automatizar la implementación, escalado y operación de aplicaciones en contenedores. Kubernetes permite gestionar clústeres de máquinas virtuales o físicas de manera eficiente. Los componentes principales incluyen:

Pods

Un Pod es la unidad de despliegue más pequeña en Kubernetes, que puede contener uno o varios contenedores que comparten almacenamiento y red.

Deployment

Un Deployment asegura que una cantidad especificada de réplicas de una aplicación estén corriendo en cualquier momento. Ejemplo de deployment.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: tower-defense-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: tower-defense-game
  template:
```

```
  metadata:
    labels:
      app: tower-defense-game
  spec:
    containers:
    - name: tower-defense-game
      image: tower-defense-game
      ports:
      - containerPort: 8080
```

Service

Un Service en Kubernetes expone una aplicación corriendo en uno o más Pods como un servicio de red. Ejemplo de service.yaml:

```
apiVersion: v1
kind: Service
metadata:
  name: tower-defense-service
spec:
  selector:
    app: tower-defense-game
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
  type: LoadBalancer
```

Ejercicio 1: Configuración y uso de docker (3 puntos)

Teoría:

- Describe los principios fundamentales de los contenedores Docker y su arquitectura interna. Explica cómo Docker maneja la seguridad y el aislamiento de contenedores.
- Compara y contrasta Docker con soluciones de virtualización tradicionales, como VMware y VirtualBox. Discute las ventajas y desventajas de cada enfoque.

Práctico:

- Escribe un Dockerfile para la aplicación Tower Defense que incluya la instalación de todas las dependencias necesarias. Asegúrate de optimizar el Dockerfile para reducir el tamaño de la imagen final.
- Construye y ejecuta el contenedor Docker utilizando el Dockerfile creado. Utiliza docker exec para acceder al contenedor y verificar que la aplicación funcione correctamente.
- Configura una red personalizada para la aplicación Tower Defense. Implementa múltiples contenedores que interactúen entre sí a través de esta red personalizada.

Ejercicio 2: Redes y volúmenes en Docker (3 puntos)

Teoría:

- Explica en detalle cómo Docker maneja las redes y los volúmenes. Discute los diferentes tipos de redes (bridge, host, overlay) y cuándo es apropiado usar cada una.
- Describe los mecanismos de persistencia de datos en Docker, incluyendo volúmenes y bind mounts. Explica las diferencias entre ellos y las mejores prácticas para su uso.

Práctico:

- Crea una red personalizada para el proyecto Tower Defense y configura los contenedores para que utilicen esta red.
- Implementa un volumen Docker para almacenar los datos del juego de forma persistente. Asegúrate de que el volumen se monte correctamente y que los datos persistan después de reiniciar el contenedor.
- Utiliza docker-compose para definir los servicios de la aplicación Tower Defense, incluyendo redes y volúmenes. Escribe un archivo docker-compose.yml que configure estos servicios y despliega la aplicación utilizando Docker Compose.

Ejercicio 3: Orquestación con Kubernetes (4 puntos)

Teoría:

- Describe la arquitectura de Kubernetes y sus componentes principales, incluyendo el API server, etcd, scheduler, y kubelet. Explica cómo estos componentes interactúan para gestionar un clúster de Kubernetes.
- Discute las estrategias de escalabilidad y alta disponibilidad en Kubernetes. Explica cómo Kubernetes maneja la recuperación de fallos y la gestión de réplicas.

Práctico:

- Escribe un archivo deployment.yaml para la aplicación Tower Defense. Asegúrate de definir los recursos necesarios (CPU, memoria) y las políticas de escalabilidad.
- Implementa un Service en Kubernetes para exponer la aplicación Tower Defense a través de una IP pública. Utiliza un LoadBalancer para distribuir el tráfico entre múltiples réplicas de la aplicación.
- Despliega la aplicación Tower Defense en un clúster de Kubernetes. Utiliza kubectl para gestionar el despliegue y verificar que la aplicación funcione correctamente en el clúster.

Ejercicio 4: Pruebas unitarias y de integración con Mockito (4 puntos)

Teoría:

- Explica los conceptos de mocks, stubs y fakes. Discute cuándo y cómo se deben utilizar estos patrones en las pruebas unitarias.
- Describe el proceso de creación de pruebas unitarias con Mockito. Explica cómo se pueden simular dependencias y verificar comportamientos en las pruebas.

Práctico:

- Escribe pruebas unitarias para la clase TowerDefenseGame utilizando Mockito para simular las dependencias de Map, Player y Wave.
- Implementa pruebas de integración que verifiquen la interacción entre las clases principales (TowerDefenseGame, Map, Player, Wave). Utiliza Mockito para controlar y verificar el comportamiento de las dependencias en estas pruebas.
- Configura un pipeline de integración continua (CI) que ejecute automáticamente las pruebas unitarias e informe sobre los resultados. Utiliza herramientas como Jenkins o GitHub Actions para implementar este pipeline (opcional).

Ejercicio 5: Pruebas de mutación (4 puntos)

Teoría:

- Define qué son las pruebas de mutación y cómo contribuyen a la mejora de la calidad del software. Explica los tipos de operadores de mutación y su propósito.
- Discute las métricas utilizadas para evaluar la efectividad de las pruebas de mutación, como la tasa de mutación (mutation score) y la cobertura de mutación.

Práctico:

- Configura una herramienta de pruebas de mutación, como PIT, en el proyecto Tower Defense. Asegúrate de integrar la herramienta en el pipeline de CI (opcional).
- Implementa pruebas de mutación para la clase Map y analiza los resultados. Asegúrate de identificar y corregir las pruebas unitarias que no detecten mutaciones.
- Realiza un informe detallado sobre la calidad de las pruebas del proyecto Tower Defense, basado en los resultados de las pruebas de mutación. Incluye recomendaciones para mejorar la cobertura y efectividad de las pruebas.

Ejercicio 6: Diseño por contrato (Design by Contract) (2 puntos)

Teoría:

- Explica el concepto de diseño por contrato y cómo se aplica en el desarrollo de software. Discute las diferencias entre precondiciones, postcondiciones e invariantes.
- Describe cómo el diseño por contrato puede mejorar la robustez y mantenibilidad del código.

Práctico:

- Aplica el diseño por contrato a la clase Tower. Define las precondiciones, postcondiciones e invariantes de los métodos principales de la clase.
- Escribe pruebas unitarias que verifiquen el cumplimiento de los contratos definidos para la clase Tower. Utiliza herramientas como Java Assertions para implementar estas verificaciones.
- Realiza una revisión de código para asegurarte de que todas las clases del proyecto Tower Defense siguen los principios del diseño por contrato. Documenta cualquier ajuste o mejora necesaria en el código.