Curso de desarrollo de software

Práctica Calificada 3

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 **PracticaCalificada3-CC3S2.** Esta carpeta albergará todas las soluciones de la prueba.
- Dentro de la carpeta **ExamenParcial-CC3S2,** deben crear dos subcarpetas adicionales, nombradas **Ejercicio1 o Ejercicio2**, respectivamente y dentro de cada parte Ejercicio1 y Ejercicio deben estar Sprint1, Sprint2, Sprint3. Cada una de estas subcarpetas contendrá las soluciones y archivos correspondientes a cada ejercicio de la prueba.

  Ejemplo de estructura:
  /PracticaCalificada1-CC3S2
  /Ejercicio1
  - Sprint1

- Sprin2
- Sprint 3

o

/Ejercicio2
- Sprint1
- Sprin2
- Sprint 3

/Ejercicio3
- Sprint1
- Sprin2
- Sprint 3

- 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.
- Los participantes deben escoger solo una pregunta indicado por el profesor y evitar cualquier inconveniente de copia entre estudiantes.

Pregunta 1: Juego de consola en Java

Descripción del juego

El juego se llama "Aventuras en el Laberinto". El objetivo del juego es que el jugador navegue a través de un laberinto, recolecte tesoros y evite trampas. El jugador se mueve por el laberinto mediante comandos de texto. El laberinto es una cuadrícula de tamaño fijo (por ejemplo, 10x10) con diferentes elementos:

- P: Posición del jugador.

- T: Tesoro.
- X: Trampa.
- .: Espacio vacío.

Instrucciones del juego

1. **Inicio del juego**: El juego inicia con el jugador en una posición aleatoria en el laberinto.
2. **Movimientos**: El jugador puede moverse en cuatro direcciones: norte (N), sur (S), este (E) y oeste (O).
3. **Recolectar tesoro**: Al moverse sobre una celda con un tesoro (T), el jugador lo recoge y su puntaje aumenta.
4. **Evitar trampas**: Si el jugador se mueve sobre una celda con una trampa (X), pierde una vida.
5. **Fin del juego**: El juego termina cuando el jugador recoge todos los tesoros o pierde todas sus vidas.

Metodología de desarrollo

- Aplicación de TDD: Desarrollar pruebas antes de implementar la funcionalidad.
- Refactorización continua: Mejorar el código constantemente para mantener alta calidad.
- Métricas de código: Cohesión, LCOM, LCOM4, CAMC, y complejidad ciclomática.
- Inyección de dependencias: Facilitar pruebas y mantenimiento del código.

Entregables de los Sprints

Sprint 1: Estructura básica y movimiento del jugador (2 puntos)

Clases a a desarrollar:

Laberinto:

- Atributos: matriz de celdas, tamaño del laberinto.
- Métodos: inicializar laberinto, mostrar laberinto.

Jugador:

- Atributos: posición actual, puntaje, vidas.
- Métodos: mover jugador, actualizar posición, verificar colisiones.

Juego:

- Atributos: instancia de laberinto, instancia de jugador.
- Métodos: iniciar juego, procesar comandos, verificar estado del juego.

Salidas:

- Mostrar laberinto inicial.

- Permitir al jugador moverse e imprimir la nueva posición.

Sprint 2: Recolección de tesoros y manejo de trampas (2 puntos)

Clases a desarrollar/modificar:

Celda:

- Atributos: tipo de celda (T, X, .).
- Métodos: constructor, obtener tipo de celda.

Laberinto (modificado):

- Métodos adicionales: colocar tesoros y trampas, actualizar celda.

Jugador (modificado):

- Métodos adicionales: recoger tesoro, verificar trampa.

Juego (modificado):

- Métodos adicionales: actualizar estado del juego, verificar victoria o derrota.

Salidas:

- Imprimir mensajes al recoger un tesoro o caer en una trampa.
- Mostrar puntaje y vidas restantes.

Sprint 3: Refactorización y métricas de calidad (3 puntos)

Actividades:

Refactorización:

- Aplicar principios SOLID.
- Implementar inyección de dependencias.

Pruebas

- Desarrollar pruebas unitarias usando JUnit.
- Aplicar TDD para nuevas funcionalidades.

Métricas de Calidad:

- Calcular LCOM, LCOM4, CAMC y complejidad ciclomática.
- Mejorar cohesión y reducir complejidad según las métricas obtenidas.

Salidas:

- Reporte de métricas de calidad antes y después de la refactorización.

- Código refactorizado con pruebas unitarias exitosas.

Notas importantes

- **Cohesión y complejidad**: Al refactorizar, asegúrate de que cada clase y método tenga una única responsabilidad (SRP de SOLID).
- **Inyección de dependencias**: Utiliza patrones como el de inyección de dependencias para desacoplar las clases, facilitando la prueba y la modificación.
- **Pruebas unitarias**: Escribe pruebas unitarias para cada método público y usa TDD para guiar el desarrollo de nuevas funcionalidades.
- **Métricas de calidad**: Utiliza herramientas como SonarQube o IntelliJ IDEA para calcular las métricas de cohesión y complejidad, y realiza ajustes en el código para mejorar estos valores.

1. Aplica TDD para desarrollar una funcionalidad específica en el juego (3 puntos)

Descripción: Implementa la funcionalidad para que el jugador pueda ver cuántos tesoros le faltan por recolectar.

Escribe un caso de prueba:

- Crea un método en una clase de prueba (por ejemplo, JuegoTest.java) que verifique al iniciar el juego, el número de tesoros restantes se muestra correctamente.

Implementa el código mínimo:

- Modifica la clase Juego para pasar la prueba.

Refactoriza y agrega Funcionalidades:

- Mejora el código para hacerlo más limpio y eficiente, asegurándote de que todas las pruebas sigan pasando.

2. Mide y mejora las métricas de cohesión y acoplamiento del código (2 puntos)

Descripción: Utiliza una herramienta como IntelliJ IDEA o SonarQube para medir las métricas de cohesión (LCOM, LCOM4, CAMC) y acoplamiento (acoplamiento entre clases, acoplamiento a clases externas) del código o en todo caso usar los códigos dados en clase.

Pasos:

Configura la herramienta:

- Configura IntelliJ IDEA o SonarQube para analizar el proyecto Java.

Mide las métricas:

- Analiza el proyecto y obtén las métricas de cohesión y acoplamiento.

Interpreta los resultados:

- Identifica las clases con baja cohesión (alto LCOM) o alto acoplamiento.

Refactoriza el código:

- Refactoriza las clases identificadas para mejorar su cohesión y reducir el acoplamiento. Por ejemplo, dividiendo clases grandes en clases más pequeñas con responsabilidades específicas.

3. Aplica los principios SOLID al diseño del juego (4 puntos)

Descripción: Revisa y refactoriza las clases del juego para cumplir con los principios SOLID.

Pasos:

Single Responsibility Principle (SRP):

- Asegúrate de que cada clase tenga una única responsabilidad.

Ejemplo: La clase Jugador solo maneja el estado y los movimientos del jugador.

Open/Closed Principle (OCP):

- Modifica las clases para que sean abiertas a la extensión, pero cerradas a la modificación.

Ejemplo: Utiliza interfaces o clases abstractas para definir comportamientos extensibles.

Liskov Substitution Principle (LSP):

- Verifica que las subclases puedan sustituir a sus superclases sin alterar el comportamiento esperado.

Ejemplo: Crea una jerarquía de clases de celdas (Celda, CeldaTesoro, CeldaTrampa).

Interface Segregation Principle (ISP):

- Divide interfaces grandes en interfaces más pequeñas y específicas.

Ejemplo: Crea interfaces separadas para las acciones de juego (mover, recoger, etc.).

Dependency Inversion Principle (DIP):

- Dependencia de abstracciones en lugar de clases concretas.

Ejemplo: Usa inyección de dependencias para las dependencias del juego (Laberinto, Jugador).

4. Mide y mejora la cobertura de código usando Jacoco (2 puntos)

Descripción: Configura Jacoco para el proyecto y asegura una cobertura de código del 80% o más.

Pasos:

Configura Jacoco:

- Agrega el plugin de Jacoco a tu archivo build.gradle o pom.xml.

Ejecuta las Pruebas:

- Ejecuta las pruebas y genera el reporte de cobertura con Jacoco.

Ejemplo para Gradle: ./ gradlew test jacocoTestReport

Analiza el reporte:

- Abre el reporte HTML generado para ver la cobertura de código.

Aumenta la cobertura:

- Escribe pruebas adicionales para cubrir las áreas no probadas.
- Asegúrate de cubrir casos extremos y de borde.

5 . Mejora el diseño del código mediante refactorización. (2 puntos)

Descripción: Identifica áreas del código que pueden beneficiarse de la refactorización y aplícale mejoras.

Pasos:

Identifica áreas para Refactorizar:

- Busca código duplicado, métodos largos, y clases con múltiples responsabilidades.

Aplica refactorización:

- Divide métodos largos en métodos más pequeños y específicos.
- Extrae clases para encapsular comportamientos.
- Mejora los nombres de métodos y variables para mayor claridad.

Verifica el comportamiento:

- Asegúrate de que todas las pruebas unitarias pasen después de la refactorización.

Repite el proceso:

- Continúa refactorizando iterativamente para mantener y mejorar la calidad del código.

Pregunta 2: Diseño del videojuego: Juego basado en consola en java

Resumen

El juego será una aventura de texto en la que los jugadores navegan a través de una serie de habitaciones, recogen objetos, resuelven acertijos y enfrentan desafíos. El juego enfatizará la aplicación de métricas de acoplamiento (Ce, Ca, CF), TDD y refactorización.

Mecánica del juego

Objetivo: Navegar por las habitaciones, recoger objetos específicos y resolver acertijos para alcanzar el objetivo final.
Jugabilidad:

- Los jugadores pueden moverse entre habitaciones usando comandos (por ejemplo, "mover norte", "mover sur").
- Los jugadores pueden interactuar con objetos en las habitaciones (por ejemplo, "recoger llave", "usar poción").
- Los jugadores deben resolver acertijos para desbloquear nuevas áreas.
- El juego termina cuando el jugador alcanza el objetivo final o completa la misión principal.

Metodología de desarrollo

- Aplicación de TDD: Desarrollar pruebas antes de implementar la funcionalidad.
- Refactorización continua: Mejorar el código constantemente para mantener alta calidad.
- Métricas de código: Métricas de acoplamiento,
- SOLID, código de cobertura y mantenimiento del código.

Estructura del proyecto

El proyecto se desarrollará en tres sprints, cada uno enfocado en diferentes aspectos del juego.

Desglose de sprints

Sprint 1: Estructura básica del juego y movimiento (3 puntos)

Clases a implementar:

- Juego: Clase principal para iniciar y controlar el flujo del juego.
- Habitación: Representa una habitación en el juego.
- Jugador: Representa al jugador y rastrea su inventario y ubicación actual.

Tareas:

1. **Inicialización del juego:**
    - Inicializa el juego con un conjunto de habitaciones.

    - Define la habitación inicial para el jugador.
2. **Movimiento del jugador:**
    - Permite al jugador moverse entre habitaciones usando comandos.
3. **Interacción básica:**
    - Implementa comandos básicos para el movimiento (por ejemplo, "mover norte").

Enfoque TDD:

- Escribe pruebas para la creación de habitaciones, movimiento del jugador e inicialización del juego.

Salida:

- Navegación funcional entre habitaciones.
- Mostrar la descripción de la habitación actual al entrar.

Sprint 2: Recolección de objetos y resolución de acertijos (3 puntos)

Clases a implementar:

- Objeto: Representa un objeto que puede ser recogido por el jugador.
- Acertijo: Representa un acertijo que necesita ser resuelto en el juego.

Tareas:

1. **Gestión de Objetos:**
    - Permitir a los jugadores recoger y usar objetos.
2. **Integración de acertijos:**
    - Implementar acertijos que los jugadores necesiten resolver para progresar.
3. **Mejoras en la interacción:**
    - Extender los comandos del jugador para incluir interacciones con objetos y acertijos.

Enfoque TDD:

- Escribir pruebas para la recogida de objetos, gestión de inventario y mecánicas de resolución de acertijos.

Salida:

- Recolección y uso funcional de objetos.
- Acertijos resolubles integrados en el flujo del juego.

Sprint 3: Refinamiento del juego y aplicación de métricas (4 puntos)

Tareas de refinamiento:

- Refactoriza el código para reducir el acoplamiento.
- Analiza y aplica métricas de acoplamiento (Ce, Ca, CF) para mejorar el diseño.
- Mejora las mecánicas del juego basándose en la retroalimentación de las pruebas.

Tareas:

Refactorización para bajo acoplamiento:

- Asegura que cada clase tenga dependencias mínimas con otras.
- Calcula y documenta las métricas Ce, Ca y CF.

Toques finales:

- Añade acertijos y objetos finales.
- Mejora la retroalimentación del usuario y la narrativa del juego.

Enfoque TDD:

- Escribe pruebas adicionales para cubrir las nuevas características añadidas y el código refactorizado.

Salida:

- Diseño del juego mejorado con métricas de acoplamiento más bajas.
- Juego finalizado con una narrativa completa y mecánicas funcionales.

Cómo jugar

Iniciar el juego:

- Lanza el juego a través de la consola.
- El juego mostrará la descripción inicial de la habitación.

Movimiento:

- Usa comandos como "mover norte", "mover sur", "mover este" y "mover oeste" para navegar.

Interacciones:

- Usa comandos como "recoger [objeto]", "usar [objeto]" y "resolver [acertijo]" para interactuar con el entorno.

Objetivo:

- Resolver acertijos y recoger objetos para progresar en el juego.
- Alcanzar el objetivo final para ganar el juego.

Ejemplos de salidas

Sprint 1
Estás en una habitación oscura. Las salidas están al norte y al este.
> mover norte
Te mueves al norte. Estás en una biblioteca.

Sprint 2
Ves una llave en el suelo.
> recoger llave
Has recogido la llave.

Sprint 3

Encuentras una puerta cerrada.
> usar llave
La puerta se abre y procedes a la siguiente habitación.

Ejercicios

1. Refactoriza el código del juego para adherirse a los principios SOLID (5 puntos)

Single Responsibility Principle (SRP):

- Identifica clases que realizan múltiples responsabilidades.
- Refactoriza estas clases para que cada una tenga una única responsabilidad.

Ejemplo: Si la clase Jugador maneja tanto el inventario como el movimiento, separar estas responsabilidades en dos clases distintas (Inventario y Movimiento).

Open/Closed Principle (OCP):

- Asegura que las clases estén abiertas para extensión pero cerradas para modificación.

Ejemplo: Crea una interfaz Comando para los comandos del jugador y extenderla para implementar nuevos comandos sin modificar las clases existentes.

Liskov Substitution Principle (LSP):

- Verifica que las subclases puedan sustituir a sus superclases sin alterar el comportamiento del programa.

Ejemplo: Si hay una clase HabitaciónEspecial que extiende Habitación, asegurar que HabitaciónEspecial se pueda usar en lugar de Habitación sin causar errores.

Interface Segregation Principle (ISP):

- Divide interfaces grandes en interfaces más específicas y pequeñas.

Ejemplo: Si hay una interfaz Interacción, dividirla en InteracciónConObjeto y InteracciónConHabitación.

Dependency Inversion Principle (DIP):

Depender de abstracciones en lugar de concretas.

Ejemplo: Usa interfaces o clases abstractas para definir dependencias, y luego implementar estas abstracciones.

2 . Implementar Jacoco para medir la cobertura de código del proyecto del juego y asegurarse de que las pruebas cubran un alto porcentaje del código (5 puntos).

Configurar Jacoco:

- Añade Jacoco al proyecto de Java.
- Configura el build script (por ejemplo, en Maven o Gradle) para generar informes de cobertura de código.

Ejecutar Jacoco:

- Ejecuta las pruebas del proyecto y generar el informe de cobertura de Jacoco.
- Analiza el informe para identificar áreas del código que no están cubiertas por las pruebas.

Mejorar la cobertura:

- Escribe alguna prueba adicionale para las partes del código que tienen baja cobertura.
- Asegura de que las pruebas cubran diferentes escenarios y casos límite.
- Genera un informe final de cobertura de código y presentarlo.

Aplicar TDD y mejorar la cobertura de código del proyecto mientras se refactoriza para adherirse a los principios SOLID. Usar Jacoco para medir la cobertura de código después de cada cambio.

Pregunta 3: Creación de un videojuego en consola en Java

Descripción del juego

El videojuego será una versión de "adivina la palabra" donde el jugador debe adivinar una palabra oculta basada en pistas dadas. Se jugará en la consola, sin interfaz gráfica. El juego proporcionará pistas y el jugador tendrá que adivinar la palabra correcta dentro de un número limitado de intentos.

Mecánicas del juego

1. El juego selecciona una palabra oculta de una lista.
2. El jugador recibe una pista inicial sobre la palabra.
3. El jugador ingresa una palabra para adivinar.
4. El juego proporciona retroalimentación sobre la adivinanza (por ejemplo "La palabra tiene X letras correctas en la posición correcta").
5. El jugador tiene un número limitado de intentos para adivinar la palabra correcta.

Metodología de desarrollo

- Aplicación de TDD: Desarrollar pruebas antes de implementar la funcionalidad.
- Refactorización continua: Mejorar el código constantemente para mantener alta calidad.
- Métricas de código: Cohesión, LCOM, LCOM4, CAMC, y complejidad ciclomática.
- Inyección de dependencias: Facilitar pruebas y mantenimiento del código.

Estructura del proyecto

El proyecto se desarrollará en tres sprints, cada uno enfocado en diferentes aspectos del juego.

Desglose de sprints

Sprint 1: configuración básica y estructura del juego (2 puntos)

Objetivo:

- Configurar el proyecto en Java.
- Crear la estructura básica del juego.
- Implementar la lógica de selección de palabras y pistas.

Clases:

- Game: Clase principal del juego.
- WordSelector: Clase responsable de seleccionar palabras.
- HintGenerator: Clase que genera pistas.

Tareas:

- Configurar el entorno de desarrollo.
- Implementar las pruebas básicas.
- Crear la estructura inicial de clases y métodos.
- Aplicar métricas de cohesión y LCOM.

Salida:

- Estructura básica del juego con selección de palabras y generación de pistas.
- Pruebas unitarias iniciales.

Notas finales para el sprint 1

- **Refactorización y TDD:** Desarrollar pruebas unitarias para cada método implementado. Refactorizar el código para mantenerlo limpio y entendible.
- **Métricas de código:** Utilizar herramientas como SonarQube o el calculadorLCOM para medir la cohesión y el LCOM de las clases y refactorizar si es necesario.

Sprint 2: Lógica de juego y retroalimentación (3 puntos)

Objetivo:

- Implementar la lógica de juego y retroalimentación.
- Desarrollar la interacción del jugador con el juego.

Clases:

- Game: Ampliar la funcionalidad.
- FeedbackGenerator: Clase que genera la retroalimentación basada en la adivinanza del jugador.

Tareas:

- Implementa la lógica para recibir adivinanzas del jugador.
- Genera y muestra retroalimentación.
- Realiza pruebas unitarias y de integración.
- Refactoriza el código basado en métricas CAMC y LCOM4.

Salida:

- Juego funcional que acepta adivinanzas y proporciona retroalimentación.
- Pruebas adicionales para la nueva funcionalidad.

Notas finales para el sprint 2

- **Refactorización y TDD:** Asegurar que cada nueva funcionalidad esté cubierta por pruebas unitarias y de integración. Refactorizar el código para mejorar la cohesión y reducir la complejidad.
- **Métricas de código:** Medir y optimizar CAMC y LCOM4. Realizar refactorizaciones necesarias para mejorar estas métricas.

Sprint 3: Refinamiento y finalización (4 puntos)

Objetivo:

- Refinar el juego, optimizar el código y aplicar inyección de dependencias.
- Asegurar la alta calidad del código y la mantenibilidad.

Clases:

- Game: Integrar todas las funcionalidades y refinar la lógica.
- DependencyInjector: Clase para manejar la inyección de dependencias.

Tareas:

- Integrar y refinar todas las partes del juego
- Optimizar el código aplicando métricas de complejidad ciclomática.
- Implementar inyección de dependencias.
- Realizar pruebas finales y refactorización.

Salida:

- Juego completamente funcional y refinado.
- Código optimizado y bien estructurado.
- Cobertura completa de pruebas.

Notas finales para el sprint 3

- **Refactorización y TDD:** Continuar aplicando TDD y refactorización continua. Asegurarse de que todas las clases y métodos estén cubiertos por pruebas unitarias y de integración.

- **Métricas de código:** Medir la complejidad ciclomática de los métodos y refactorizar para mantenerla baja. Implementar y revisar la inyección de dependencias para mejorar la mantenibilidad del código.

Ejercicios

1. Implementa la inyección de dependencias para las clases WordSelector y HintGenerator. (2 puntos)

Tareas:

- Crea una interfaz IWordSelector y una implementación concreta WordSelector.
- Crea una interfaz IHintGenerator y una implementación concreta HintGenerator.
- Modifica la clase Game para usar estas interfaces a través de inyección de dependencias.

```
// Interfaz IWordSelector
public interface IWordSelector {
   String selectWord();
}

// Implementación WordSelector
public class WordSelector implements IWordSelector {
   public String selectWord() {
      return "example";
   }
}

// Interfaz IHintGenerator
public interface IHintGenerator {
   String generateHint(String word);
}

// Implementación HintGenerator
public class HintGenerator implements IHintGenerator {
   public String generateHint(String word) {
      return "The word has " + word.length() + " letters.";
   }
}

// Clase Game modificada
public class Game {
   private IWordSelector wordSelector;
   private IHintGenerator hintGenerator;

   // Constructor con inyección de dependencias
   public Game(IWordSelector wordSelector, IHintGenerator hintGenerator) {
      this.wordSelector = wordSelector;
      this.hintGenerator = hintGenerator;
   }
```

```
    public void start() {
        String word = wordSelector.selectWord();
        String hint = hintGenerator.generateHint(word);
        System.out.println("Hint: " + hint);
    }

    public static void main(String[] args) {
        IWordSelector wordSelector = new WordSelector();
        IHintGenerator hintGenerator = new HintGenerator();
        Game game = new Game(wordSelector, hintGenerator);
        game.start();
    }
}
```

2. Implementa un contenedor de inyección de dependencias simple para gestionar las dependencias del juego (2 puntos)

Tareas:

- Crea una clase DependencyInjector para gestionar las dependencias.
- Modifica la clase Game para obtener las dependencias a través del contenedor.

3. Aplica principios SOLID al código del juego (3 puntos)

Tareas:

- Refactoriza el código para cumplir con el principio de responsabilidad única (SRP) separando la lógica de selección de palabras y generación de pistas en clases separadas.
- Implementa el principio de abierto/cerrado (OCP) permitiendo la extensión de la lógica de generación de pistas sin modificar las clases existentes.

4. Aplicar TDD y Jacoco para desarrollar una nueva funcionalidad y refactorizar el código existente (4 puntos)

Tareas:

- Implementa pruebas unitarias para una nueva funcionalidad, como permitir al jugador obtener una nueva pista después de un número específico de intentos fallidos.
- Refactoriza el código para implementar la nueva funcionalidad siguiendo TDD.
- Utiliza una herramienta de análisis de cobertura como JaCoCo.
- Escribe pruebas unitarias adicionales para alcanzar una mejor cobertura.
- Analice los resultados de cobertura y refactorizar el código si es necesario.