Román Ardanuy Hamill

Arilson Reynaldo Ventocilla Ramos

Jueves 12:30

PONG

En este proyecto hemos desarrollado una implementación del clásico juego Pong siguiendo una metodología de desarrollo basada en pruebas (TDD) y aplicando diversos patrones y técnicas de testing. El juego ha sido implementado utilizando Java y JavaFX, siguiendo una arquitectura Modelo-Vista-Controlador (MVC) que nos ha permitido mantener una clara separación de responsabilidades y facilitar las pruebas unitarias. Se han implementado diferentes modos de juego como el modo clásico, modo de velocidad incremental, tiempo limitado y primero en llegar a cinco puntos, añadiendo variedad a la experiencia de juego básica.

Para ejecutar el juego, es necesario tener instalado Java JDK 21 y Maven. El proyecto se puede ejecutar clonando el repositorio https://github.com/RomanArdanuy/TQS\_Practica.git) y, desde el directorio del proyecto, ejecutando el comando mvn clean javafx:run. Los controles del juego son W/S para el jugador izquierdo y las flechas ARRIBA/ABAJO para el jugador derecho. Se puede volver al menú principal con la tecla ESC y reiniciar la partida actual con la tecla R. El proyecto se ha desarrollado con un enfoque en la calidad del código y la testabilidad, implementando pruebas de caja negra, caja blanca, design by contract y mocks, alcanzando una cobertura significativa en todos los componentes principales del juego.

**Funcionalidad**: Creación y movimiento de la pelota en un juego de Pong. Se ha implementado:

* Inicialización con posición (x,y) y tamaño (radio)
* Movimiento con velocidad variable
* Cambio de dirección

**Localización**:

* Archivo: src/main/Pong/model/Ball.java
* Clase: Ball
* Métodos principales: Constructor, move(), reverseX(), reverseY()Class: Ball

**Test**:

* Archivo: src/test/Pong/model/BallTest.java
* Clase: BallTest
* Tipo de test: Caja Negra
* Técnicas:

1. Particiones Equivalentes:
   * Valores normales de posición (100.0, 100.0)
   * Velocidades positivas/negativas/cero
2. Valores Límite:
   * Posiciones mínimas (0.0, 0.0)
   * Radio mínimo (1.0)
   * Cambios de dirección (+5.0 a -5.0)

Resumen:  
Hemos realizado pruebas de caja negra para la clase Ball, que implementa la pelota del juego Pong, siguiendo el enfoque TDD (Test-Driven Development). Nuestras pruebas incluyen particiones equivalentes (posiciones normales, velocidades positivas/negativas/cero) y valores límite (posiciones mínimas 0.0, radio mínimo 1.0, cambios de dirección). El informe de cobertura JaCoCo muestra un 100% de cobertura en todos los métodos (11 métodos, 21 líneas de código), verificando la inicialización de la pelota, su movimiento y los cambios de dirección. La estructura sigue el patrón MVC, con nuestra clase Ball formando parte del modelo del juego.

Los ficheros en cuestión se pueden ver en el commit: Initial commit & black box testing

**Funcionalidad**: Detección de colisiones entre la pelota y la pala en nuestro Pong. Se ha implementado:

* Colisiones verticales y horizontales
* Verificación de alineamiento
* Detección de bordes

**Localización**:

* Ball.java: Nuevo método checkCollision(Paddle)
* Paddle.java: Nueva clase para representar las palas
* CollisionTest.java: Tests de caja blanca

**Test de Caja Blanca**:

1. Cobertura de Decisión/Condición (83%):
   * Método: checkCollision()
   * Condiciones probadas:
     + Alineación vertical (y + radius >= paddle.getY())
     + Alineación horizontal (x + radius >= paddle.getX())
     + Combinación de ambas condiciones
2. Cobertura de Caminos:
   * Sin colisión
   * Colisión solo vertical
   * Colisión solo horizontal
   * Colisión completa
   * Casos límite (bordes y esquinas)
3. Resultados de Cobertura:
   * Instrucciones: 100%
   * Ramas: 83%
   * Líneas: 100%
   * Métodos: 100%

Resumen:   
Para el testeo de caja blanca, nos hemos centrado en el método checkCollision() que verifica las colisiones entre la pelota y la pala en nuestro juego de Pong. Implementamos tests que cubren la cobertura de decisión/condición y la cobertura de caminos, alcanzando un 100% de cobertura de instrucciones y un 83% de cobertura de ramas. Los tests verifican diferentes escenarios de colisión, incluyendo casos sin colisión, colisiones parciales (solo vertical u horizontal) y colisiones completas, además de casos límite en bordes y esquinas.

Los ficheros se pueden ver en el commit: "White box testing - Collision detection"

**Funcionalidad**: Implementación de Design by Contract en las clases del modelo del juego Pong. Se han definido contratos para asegurar el correcto funcionamiento de las clases Ball y Paddle.

**Localización**:

* Clases: Ball.java y Paddle.java
* Tests: ContractTests.java

**Contratos Implementados**:

1. Invariantes de clase:
   * Radio/dimensiones positivas
   * Posiciones y velocidades finitas
   * Comprobación en todos los métodos
2. Precondiciones:
   * Constructor: dimensiones > 0
   * setVelocity: valores finitos
   * checkCollision: paddle no nulo
3. Postcondiciones:
   * move: posición actualizada correctamente
   * reverse: velocidades invertidas

**Tests**:

* Verificación de contratos del constructor
* Validación de movimiento
* Comprobación de colisiones
* Validación de velocidades
* Cobertura del 100% de los contratos

Los ficheros se pueden ver en la rama “Designbycontract”, commit: "Design by Contract implementation"

Resumen:  
Hemos implementado Design by Contract en las clases principales del modelo (Ball y Paddle), estableciendo contratos que garantizan el correcto funcionamiento del juego. Los contratos incluyen invariantes de clase que mantienen la consistencia de los objetos (dimensiones positivas, posiciones finitas), precondiciones que aseguran entradas válidas (dimensiones correctas, valores finitos) y postcondiciones que verifican el comportamiento esperado de los métodos (actualizaciones de posición, cambios de velocidad). La implementación se ha validado mediante ContractTests, alcanzando una cobertura completa de los contratos definidos.

**Funcionalidad**: Implementación de Mock Objects para testing del juego Pong.

**Proceso de Implementación**:

**Mock de Vista con Mockito (GameView)**

* **Objetivo**: Simular la interfaz gráfica sin necesidad de implementarla
* **Proceso**:
  1. Creación de interfaz GameView con métodos clave:
     + updateBallPosition
     + updatePaddlePositions
     + updateScore
     + render
  2. Implementación de GameControllerTest:
     + Uso de anotación @Mock para crear mocks
     + Configuración en setUp() con MockitoAnnotations
     + Verificación de llamadas a métodos con verify()
* **Beneficios**:
  1. Testing sin dependencia de GUI
  2. Verificación de lógica del controlador
  3. 100% cobertura en GameController

**Mock Manual de ScoreBoard**

* **Objetivo**: Gestionar puntuación de forma testeable
* **Proceso**:
  1. Diseño de interfaz ScoreBoard
  2. Implementación manual en MockScoreBoard:
     + Tracking de puntuación
     + Lógica de victoria
     + Métodos de verificación adicionales
  3. Integración en GameModel
  4. Tests específicos en MockScoreBoardTest
* **Resultados**:
  1. 95% cobertura de instrucciones
  2. Verificación completa de estados del juego

**Mock de GameLogger con Mockito**

* **Objetivo**: Registrar eventos del juego
* **Proceso**:
  1. Definición de interfaz GameLogger
  2. Integración en GameModel:
     + Logging de inicio de juego
     + Registro de colisiones
     + Tracking de puntuación
     + Notificación de fin de juego
  3. Tests en GameLoggerTest
* **Verificaciones**:
  1. Llamadas correctas a métodos
  2. Secuencia de eventos
  3. Interacción con otros componentes

**Resultados de Cobertura**:

**Cobertura por Componente**:

* **Model Layer**:
  + GameModel: 64% inst., 34% branches
  + Ball: 100% inst., 91% branches
  + Paddle: 84% instrucciones
  + MockScoreBoard: 95% inst., 62% branches
* **Controller Layer**:
  + GameController: 100% cobertura total
* **View Layer**:
  + Cubierto por mocks

**Métricas de Testing**:

* **Cantidad de Tests**:
  + GameControllerTest: 2 tests
  + BallTest: 7 tests
  + CollisionTest: 5 tests
  + GameLoggerTest: 3 tests
  + GameModelTest: 3 tests
  + MockScoreBoardTest: 3 tests
* **Total**: 23 tests ejecutados

**Estadísticas Globales**:

* **Líneas de Código**:
  + Total: 229 líneas
  + Testeadas: 146 líneas
* **Métodos**: 63 métodos
* **Clases**: 8 clases
* **Cobertura General**:
  + Instrucciones: 55%
  + Ramas: 38%
  + Métodos: 100%

**Cumplimiento de Objetivos**:

* ✓ Mock de Vista para Controller
* ✓ Mock Manual implementado
* ✓ Mock adicional con Mockito
* ✓ Tests pasando correctamente
* ✓ Cobertura significativa

Resumen:

Para el testing con Mock Objects, hemos implementado tres enfoques diferentes que cubren distintos aspectos del juego. Primero, utilizamos Mockito para simular la Vista en GameControllerTest, permitiendo verificar la lógica del controlador sin depender de la interfaz gráfica. Segundo, desarrollamos un mock manual del ScoreBoard para gestionar y testear la puntuación del juego de forma aislada, alcanzando un 95% de cobertura. Finalmente, implementamos un mock del GameLogger con Mockito para verificar el registro de eventos del juego. Los tests ejecutados (23 en total) demuestran una cobertura significativa (55% de instrucciones, 38% de ramas) y la correcta interacción entre los componentes del sistema.

En cuanto a la integración continua hemos creado el repositorio en git, donde se pueden ver los primeros testeos en forma de commits, pero en cuanto hemos visto que habían que hacer ramas a parte lo hemos realizado así.

**Resumen** **Final del Proyecto**

Durante el desarrollo de nuestro juego Pong, hemos implementado con éxito todos los requisitos establecidos para alcanzar una nota de entre 7 y 10, además de características adicionales que enriquecen la experiencia del juego. A continuación se detallan nuestros logros:

1. Implementación MVC:
   * Model: Ball, Paddle, GameModel y ScoreBoard
   * View: GameView y GameViewImpl con JavaFX
   * Controller: GameController para la lógica del juego
   * Separación clara de responsabilidades
2. Testing Completo:
   * Black Box Testing (75% particiones equivalentes y valores límite)
   * White Box Testing (cobertura de código, decisiones y caminos)
   * Design by Contract en clases del modelo
   * Mock Objects:
     + Mock manual de ScoreBoard
     + Mock con Mockito para GameView
     + Mock adicional de GameLogger
3. Características del Juego:
   * Menú principal interactivo
   * Múltiples modos de juego:
     + Modo Clásico
     + Modo Velocidad Incremental
     + Modo Contrarreloj
     + Modo Primero a 5
   * Sistema de puntuación
   * Controles intuitivos
4. Métricas de Calidad:
   * Cobertura de código: 77% instrucciones en model
   * Tests unitarios: 23 tests implementados
   * Integración continua configurada
   * Documentación completa

El proyecto no solo cumple con los requisitos técnicos de testing y calidad de código, sino que también ofrece un producto final jugable y entretenido, demostrando la aplicación práctica de los conceptos de testing en un proyecto real.

Los ficheros se pueden ver en los diferentes commits del repositorio, cada uno enfocado en una fase específica del desarrollo: testing de caja negra, testing de caja blanca, design by contract, implementación de mocks y características adicionales del juego.