INFORME PRÁCTICA TQS

JUEGO PARCHIS

Xavi Grau Lirio
Leidy Luzón Velez
Curs 2025 - 2026
09/12/2025

1. INTRODUCCIÓN

En este proyecto hemos implementado el juego del Parchís utilizando el lenguaje Java y una arquitectura basada en el patrón Modelo-Vista-Controlador. El objetivo principal ha sido construir un sistema fácilmente verificable mediante diferentes técnicas de pruebas de software. En el modelo tenemos toda la lógica del juego incluyendo los jugadores, las fichas, el tablero, el dado y el funcionamiento general de las mecánicas del Parchís. En la vista hemos implementado con Swing el tablero visual y en el controlador coordinamos la comunicación entre ambos componentes.

Hemos seguido un enfoque TDD que nos ha permitido mantener un crecimiento progresivo del sistema y garantizar que cada nueva funcionalidad estuviera correctamente validada desde el primer momento. También, hemos aplicado técnicas de prueba de caja negra y caja blanca. Además de mock objects mediante Mockito para simular dependencias y poder comprobar el comportamiento de componentes sin necesidad de llamar directamente a esas clases.

Por otro lado, cada nueva funcionalidad o corrección se ha desarrollado en una rama independiente y posteriormente integrada en main mediante pull requests. El proceso de automatización se ha implementado con GitHub Actions, utilizando archivos yaml que ejecutan los tests y aplican reglas de estilo mediante Checkstyle, garantizando así que únicamente se integra cambios que cumplen los estándares definidos.

Este informe presenta de manera estructurada todo el proceso de pruebas realizado durante el desarrollo, mostrando cómo se han aplicado las distintas técnicas requeridas y aportando las evidencias correspondientes.

2. NUESTRAS REGLAS DEL PARCHÍS

2.1 Salidas de las fichas desde la casa

Las fichas no se pueden incorporar al tablero en cualquier momento. Para que una ficha salga de la casa es necesario obtener un 5 en el dado. Mientras la ficha se encuentra en la casa, su posición se representa mediante un valor negativo. Cuando se obtenga el valor 5, la ficha se colocará en la casilla de salida de su color.

2.2 Movimiento general por el tablero

Cuando una ficha ya está dentro del tablero, avanza tantas casillas como indique el resultado del dado. El recorrido puede requerir ajustes dependiendo del color de la ficha, especialmente al entrar en el tramo final. Si un movimiento supera los límites permitidos para el recorrido específico de un color, la casilla no avanzara y se quedará en la casilla que está. También recalcar que en caso de 6 en el dado, el jugador vuelve a tirar.

2.3 Barreras

Consideramos barrera cuando en una casilla encontramos dos fichas del mismo color. Cuando esto ocurre:

- Ninguna ficha puede atravesar esa casilla
- Las fichas que avanzan deben detenerse antes de la barrera, incluso si el valor del dado les permitiría superarla.
- Una barrera solo desaparece cuando una de las dos fichas se desplaza voluntariamente.

En la implementación, antes de validar cualquier movimiento se comprueba si existe una barrera en alguna de las casillas situadas entre la posición actual de la ficha y la destino. Si se detecta una barrera en el camino, el movimiento queda prohibido.
Para dejar simplificadas las reglas de las barreras y por unificar las diversas normativas existentes hemos establecido que dos fichas de diferentes colores no pueden realizar una barrera y además no se permite juntar dos fichas de diferente color en una casilla segura.

2.4 Casillas seguras

Como su nombre indica son casillas en las que no se puede capturar a la ficha que exista dentro de esa casilla.

2.5 Capturas

La captura ocurre cuando una ficha se coloca en una casilla no segura que ya estaba ocupada por una ficha de otro color. La ficha capturada vuelve a la casa del jugador correspondiente y la ficha que captura avanza, si es posible 20 casillas.

2.6 Victoria

Una vez una ficha, consigue dar toda la vuelta al tablero, llega a las posiciones finales, una vez llega a la ultima casilla final la ficha contabiliza como un punto. Gana el jugador que consiga llevar todas sus fichas a la posición final.

3. TESTS CAJA NEGRA Y MOCK OBJECTS

3.1 DadoTest

El test implementado en DadoTest verifica el comportamiento básico del método **lanzar()**. Para ello, hemos realizado cien lanzamientos consecutivos del dado donde comprobamos que cada valor obtenido se encuentra dentro del rango válido (1 a 6). Con esto no aseguramos que el dado nunca devuelve valores fuera de los límites.

3.3.1 Diseño por contrato e invariantes

El constructor y el método lanzar() incluyen precondiciones que aseguran que el generador aleatorio esté correctamente inicializado, y una postcondición que verifica que el valor obtenido siempre se mantiene dentro del rango permitido. Además, la clase define un invariante mediante checkInvariant(), que asegura que el dado mantiene de forma permanente un rango válido entre 1 y 6. Con ello, la clase garantiza que cualquier lanzamiento cumple siempre las reglas establecidas.

3.2 PosicionTest

La prueba en PosicionTest verifica que el constructor que recibe los parámetros **nCasilla** y **seguro** inicializa correctamente los valores internos del objeto. Con este test comprobamos que los parámetros del constructor se asigna correctamente a los atributos internos y los métodos **getNumero()** y **esSeguro()** devuelven los valores esperados.

3.3.2 Diseño por contrato e invariantes

Establecemos que una casilla sólo puede marcarse como segura si pertenece al conjunto de casillas definidas como seguras en el tablero. Los constructores y los métodos modificadores llaman a checkInvariant() para asegurar que esta regla se cumple en todo momento, incluso después de cualquier cambio en la posición.

3.3 FichaTest

3.3.1 testConstructorInicializaCorrectamente

Para dos fichas creadas con distintos colores y tipos, comprobamos que cada una conserva correctamente los valores pasados como argumento y que la posición inicial es nula cuando así se especifica.

3.3.2 testSetTipoCambiaTipoFicha

Evaluamos el comportamiento del método **setTipo()**. Hemos cambiado el tipo de una ficha previamente inicializada y comprobamos que el cambio se aplica correctamente y queda reflejado mediante el getter.

3.3.3 testSetColorCambiaColorFicha

Verificamos que el método **setColor()** permite modificar el color de una ficha sin inconsistencias. Establecemos un nuevo color para una ficha previamente creada y confirmamos que el getter lo devuelve correctamente.

3.3.4 testGettersRetornaValoresEsperados

Se revisa que los métodos **getColor()**, **getTipo()** y **getPosicion()** devuelven los valores definidos al crear la ficha.

3.3.5 testConstructorVacioNoLanzaErrores

En este último tests comprobamos que el constructor vacío de la clase **Ficha** puede ejecutarse sin producir excepciones, incluso cuando no se inicializan los atributos. Además, verificamos que todos los valores quedan en estado nulo.

3.3.6 Diseño por contrato e invariantes

La clase Ficha aplica diseño por contrato mediante precondiciones e invariantes que garantizan que cada ficha mantenga un estado válido. Las fichas reales deben tener siempre un color y un tipo no nulos, y si son del tipo TIPO_EMPTY no pueden tener posición ni actuar como barrera. El constructor vacío queda excluido de estas restricciones porque representa una ficha especial del tablero. Gracias a estas reglas, la clase asegura un comportamiento coherente sin perder flexibilidad en la representación del juego.

3.4 TableroTest

3.4.1 testInicializarTablero

En este test comprobamos que la inicialización del tablero se realiza correctamente. Primero comprobamos que las dimensiones de la matriz sean 19x19. Luego, el test comprueba que nunca hay celdas nulas, es decir, todas las posiciones contienen una ficha válida, ya sea una ficha de un color o una ficha vacía.

Además, comprobamos que las casillas donde deberían colocarse las fichas iniciales de cada color coinciden con su ubicación esperada, y que esas fichas están configuradas como ocupadas y con el color correcto. Por último, el test nos confirma que el resto del tablero queda completamente vacío, sin fichas adicionales, lo cual nos asegura que no hay errores en la creación del estado inicial del juego.

3.4.2 testObtenerIndice

En este test analizamos el método encargado de buscar la posición en el tablero (fila y columna) correspondiente a un número de casilla específico. Comprobamos los valores fronteras e inválidos.

En los casos válidos, el test comprueba que las coordenadas devueltas coinciden exactamente con las posiciones en las que la inicialización del tablero asignó a esos números. Además, introducimos valores por debajo del mínimo (-16) y por encima del máximo (120), comprobando que el método devuelve null correctamente cuando el número de casilla no pertenece al tablero.

3.4.5 testMovimentPosible

En este test comprobamos toda la lógica que determina si una ficha puede moverse o no. Comprobamos todos los casos con todas las particiones equivalentes y los valores limite, frontera y pertenecientes a valores esperados

```
//CAS BASIC -> captura normal
assertTrue(tablero.movimientPosible(ficha, numDado: 5));
assertTrue(tablero.isCaptura());
assertTrue(tablero.movimientPosible(ficha, numDado: 6)); //Valors frontera
assertFalse(tablero.isCaptura());
assertTrue(tablero.movimientPosible(ficha, numDado: 4)); //Valors frontera
assertFalse(tablero.isCaptura());
```

- **Captura normal**: Creamos una ficha y un oponente situados de forma que la primera pueda capturar a la segunda. Comprobamos que el movimiento es permitido y que el atributo captura pasa a true.
- **Movimientos sin captura:** Probamos movimientos que no deben generar captura, verificando que el sistema reconoce correctamente situaciones neutras.
- **Interacción con fichas aliadas:** Una ficha no puede capturar ni impedir el paso de otra de su mismo color salvo que formen barrera. En el test revisamos que estos movimientos se permiten y no activan captura.
- **Barrera rival:** Si una ficha encuentra una barrera en su recorrido, no puede pasar. En el test creamos una barrera y comprobamos que el movimiento deja de ser posible.
- **Casilla segura:** Aunque el movimiento sea válido, si la casilla destino es segura y tiene una ficha rival, no se permite captura.
- **Conversión al tramo final:** Verificamos que cada color se convierte correctamente cuando entra en su recta final.
- **Bucle del tablero:** Comprobamos que el tablero se comporta como un circuito.
- **Salida de casa:** Verificamos que al sacar un 5 en el dado una ficha con posición negativa mueve su ficha a la casilla de salida.

3.4.6 Diseño por contrato e invariantes

La clase Tablero emplea diseño por contrato mediante precondiciones que garantizan el uso correcto de sus métodos, como impedir reinicializar el tablero o asegurar que las fichas y posiciones recibidas sean válidas. También incorpora postcondiciones para verificar que los valores resultantes, como casillas calculadas o movimientos, se mantengan dentro del rango permitido. Además, define invariantes que aseguran que el tablero conserve siempre una estructura coherente: las matrices deben ser de 19×19, no puede haber fichas nulas y todos los tipos de ficha deben ser válidos. Con estas reglas, el tablero mantiene un estado consistente durante toda la partida.

3.5 JugadorTest

3.5.1 testMoverFichaElegida

Verificamos el comportamiento principal del método **jugar()**. Dado que necesitamos interacción por consola, el test simula la entrada del usuario utilizando un **ByteArrayInputStream**, lo que nos permite automatizar la selección de opciones. Además, preparamos dos fichas rojas (ficha1 y ficha2) y configuramos el tablero simulado para que ambas puedan moverse con el número de dado 5. Esto provoca que ambas aparezcan en la lista interna de fichas movibles que genera el método **jugar().**

La comprobación clave es verificar que el jugador realmente mueve la ficha seleccionada, y no otra, es decir, que:

- tablero.mouFicha(ficha2, 5) se ejecuta.
- tablero.mouFicha(ficha1, 5) no se ejecuta.

3.5.2 testNingunaFichaSePuedeMover

En este test analizamos un escenario donde ninguna de las fichas asignadas se puede mover con el número obtenido en el dado. Para ello, hemos simulado el tablero para que el método **movimentPosible()** devuelve siempre false, en este caso y que nunca llame a **tablero.mouFicha()**.

3.5.3 testFicha1EnBarrera

En este test evaluamos el caso de barreras. Cuando una ficha la marcamos como barrera, su comportamiento se modifica y puede impedir el movimiento incluso si es una ficha propia.

En el test, la ficha1 está en barrera y la ficha 2 no está en barrera. El usuario elige la ficha1 para mover pero no es posible moverla porque se encuentra con una barrera y por lo tanto debe elegir la ficha 2 que sí se puede mover. Con el test confirmamos que la ficha2 se mueve y la ficha1 nunca se mueve en este caso.

3.5.4 testAñadirFicha

Comprobamos que las cuatro fichas iniciales añadidas manualmente están correctamente almacenadas en la lista interna del jugador. Después, en el test intentamos añadir una ficha cuyo color no coincide con el del jugador, lo que debe provocar una excepción.

3.5.6 testHaGanado

Verificamos que para que un jugador sea marcado como ganador, todas sus fichas deben encontrarse en la casilla final correspondiente a su color. En el test creamos un jugador verde y le asignamos cuatro fichas simuladas en la posición 92 que es el número de meta del color verde, verificamos que el método **haGanado()** devuelve true para este jugador y también comprobamos que el jugador rojo que no ha alcanzado la meta devuelva false.

3.5.7 Diseño por contrato e invariante

La clase *Jugador* aplica diseño por contrato mediante precondiciones que garantizan que el nombre y el color sean válidos y que las fichas añadidas pertenezcan al jugador. Además, define un invariante que asegura que el jugador mantiene siempre un estado coherente: su nombre no es vacío, su color nunca es nulo y todas las fichas de la lista son válidas y coinciden con su color. Estas comprobaciones se ejecutan en los métodos que pueden modificar el estado interno, asegurando un comportamiento estable durante toda la partida.

3.6 JuegoTest

3.6.1 testPrimerTurnosEsPrimerJugador

En este test validamos que el juego inicializa correctamente el turno, asignándole al primer jugador de la lista. Al crear la instancia de **Juego**, se espera que **turnoActual** empiece en cero, lo que implica que el método **getJugadorActual()** debe devolver el primer jugador en la lista que es j1. Con este test garantizamos que el juego comienza de forma consistente y que el turno inicial está bien definido antes de realizar cualquier acción.

3.6.2 testCambioDeTurnoConTiradaNormal

Simulamos que el dado devuelve un valor de 3 y se ejecuta **jugarTurno().** Dado ese valor el juego debería avanzar el turno hacia el siguiente jugador. El test nos confirma que el jugador activo después del turno es j2.

3.6.3 testJugadorSaca6YRepiteTurno

Comprobamos que cuando a un jugador le sale 6 en el dado vuelve a jugar un turno. Para simular este escenario, el dado se programa para devolver primero un 6 y luego un 3. En la primera tirada, j1 debe mantener el turno activo. En la segunda tirada, como ya no es un 6, el turno debe pasar a j2.

3.6.4 testJugadorNoPuedeMoverYPasaTurno

En este analizamos el comportamiento del juego cuando el jugador de turno no puede mover ninguna de sus fichas. Para ello, forzamos el método **puedoMover()** para que devuelva false, independientemente de las posiciones reales de sus fichas.

Cuando se simula la tirada y se ejecuta el turno, el test verifica que el juego pasa al siguiente jugador sin realizar ningún movimiento. Por lo tanto, este test nos asegura continuidad del juego incluso ante escenarios de inmovilidad.

3.6.5 testVictoriaJugador

Evaluamos la condición final de la partida, para comprobarlo creamos un espía del jugador **j1** que siempre devuelve **true** al llamar a **haGanado()**. Así, al ejecutar **jugarTurno()**, el juego detecta inmediatamente una condición de victoria. El test nos confirma que el estado del

juego pasa a **terminado = true**, que el ganador registrado corresponde al jugador que cumple la condición y que no se produce avianca de turno después.

3.6.6 Diseño por contrato e invariante

Las precondiciones garantizan que existan jugadores, tablero y dado antes de ejecutar acciones, mientras que las postcondiciones verifican que los resultados del turno (como la tirada o el avance del jugador) sean coherentes. El invariante refuerza esta estabilidad comprobando que el turno esté dentro del rango permitido, que todos los jugadores sean válidos y que, si el juego ha finalizado, el ganador sea correcto. De este modo, la clase mantiene una lógica interna consistente durante toda la partida.

4. TESTS CAJA BLANCA

4.1 Statement coverage

Los tests realizados ofrecen un statement coverage porque obligan a ejecutar todas las instrucciones del código. En **Juego** cubrimos los distintos resultados de un turno; comienzo de partida, cambio de jugador, repetición por sacar un 6, imposibilidad de mover y finalización del juego. En **Jugador**, las pruebas recorren todos los caminos del método **jugar**, incluyendo selección de ficha, entradas inválidas, barreras, colores, incorrectos, posiciones nulas y diferentes iteraciones del bucle. Finalmente, en **Tablero** se ejercitan todas las situaciones de movimiento: capturas, bloqueos, casillas seguras, vueltas completas según color, conversión a casilla final, búsqueda interna de fichas y movimientos iniciales. En conclusión, estas pruebas ejecutan las sentencias del sistema y proporcionan una cobertura muy sólida.

4.2 Decision coverage and Condition coverage

Los tests nos proporcionan un decision coverage muy completo, ya que ejecutamos prácticamente todas las rama de decisión presentes en el juego: en **Juego**, comprobamos las bifurcaciones asociadas a repetir turno, pasar turno, detectar victoria y detener partida. En **Jugador**, evaluamos las decisiones sobre si una ficha puede moverse, si la entrada del usuario es válida, si existe una barrera o si alguna condición impide seleccionar una ficha; y en **Tablero**, recorremos todas las ramas vinculadas a capturas, movimientos bloqueados, casillas seguras, vueltas completas, movimiento inicial o conversión a casilla final.

En cuanto a la condition coverage, las pruebas fuerzan la evaluación independiente de condiciones múltiples dentro de las decisiones: fichas aliadas u oponentes, estados de barrera, posiciones fuera de rango, valores frontera, casillas seguras, movimientos imposibles o retornos al inicio.

Además hay casos donde añadimos un enum con los diferentes resultados que pueden ocurrir para poder hacer el test correctamente y comprobar el resultado exacto y testear que en las condiciones concretas obtenemos los resultados esperados.

```
public enum ResultadoTurno {
    NO_PUEDE_MOVER,
    MOVIMIENTO_REALIZADO,
    REPITE_TURNO,
    JUGADOR_GANA,
    TURNO_AVANZA,
    JUEGO_TERMINADO
}
```

4.3 Path coverage

En **Juego**, se cubren rutas que incluyen turnos normales, turnos repetidos por sacar un 6, caminos donde el jugador no puede mover y caminos que llevan directamente a la victoria. En **Jugador**, se exploran rutas que van desde la selección correcta de ficha hasta recorridos más complejos con entradas inválidas, barreras, colores no permitidos o posiciones nulas. En **Tablero**, los tests obligan a atravesar diferentes caminos completos del movimiento: capturas, bloqueos, casillas seguras, pasos por la casilla inicial y las rutas especiales según cada color. Aunque el número total de caminos posibles es muy elevado, las pruebas cubren los más representativos y los críticos para el funcionamiento del juego.

4.4 Loop testing

En el método **jugar()** de **Jugador**, evaluamos casos de 0 iteraciones, 1 iteración y múltiples iteraciones consecutivas. Esto nos garantiza que el bucle de selección de ficha funciona correctamente ante cualquier secuencia de entrada del usuario. Además, en **Tablero,** las operaciones que implican búsquedas internas o recorridos del tablero son verificadas bajo escenarios normales y límite, asegurando que los bucles internos gestionan adecuadamente los casos en los que la ficha existe, no existe o debe recorrer el tablero para actualizar su posición. En concreto, analizamos los dos bucles simples en los bucles de jugar() y los loops aniuats en la clase Tablero. En conclusión, el loop testing nos confirma que los ciclos se finalizan correctamente sin caer en iteraciones inesperadas.

```
@Test
void loopTest_oneIteration() {
    when(tablero.movimientPosible(ficha1, numDado: 4)).thenReturn(value: true);

    InputStream original = System.in;
    try {
        // Primer input inválido (0), segundo válido (1)
        System.setIn(new ByteArrayInputStream("0\n1\n".getBytes()));

        boolean res = jugador.jugar(numDado: 4, tablero);

        assertTrue(res);
        verify(tablero).mouFicha(ficha1, numDado: 4);
    } finally {
        System.setIn(original);
    }
}
```

```
@Test
void loopTest_twoIterations() {
    when(tablero.movimientPosible(ficha1, numDado: 4)).thenReturn(value: true);

    InputStream original = System.in;
    try {
        // Dos entradas inválidas -> la tercera válida
        System.setIn(new ByteArrayInputStream("abc\n0\n1\n".getBytes()));

        boolean res = jugador.jugar(numDado: 4, tablero);

        assertTrue(res);
        verify(tablero).mouFicha(ficha1, numDado: 4);
    } finally {
        System.setIn(original);
    }
}
```

5. TESTS CONTROLADOR

En el test del controlador hemos verificado que su comportamiento coordina correctamente la interacción entre el modelo y la vista. Comprobamos que, al iniciar la partida, se ejecutan en orden las acciones de limpiar la interfaz, mostrar el mensaje inicial y actualizar el estado del tablero. También comprobamos los distintos escenarios posibles del método **jugarTurno()**: cuando se desarrolla un turno normal y cuando la partida finaliza cuando hay un jugador. En cada caso , comprobamos que el controlador llama a los métodos correctos del modelo y de la vista, mostrando el mensaje correcto y manteniendo la coherencia del juego. Con todo esto nos aseguramos que el controlador gestiona de forma correcta todas las situaciones y que la comunicación entre capas es la adecuada.

6. COMO JUGAR

Al ejecutar el método `main`, el sistema solicitará por consola la configuración inicial de la partida. En primer lugar, se pedirá introducir el número de jugadores y, posteriormente, el nombre y el color asignado a cada uno de ellos. Una vez completado este proceso de inicialización, se abrirá automáticamente la interfaz gráfica desarrollada en **Swing**, donde se mostrará el tablero de juego.

Aunque la visualización se realiza mediante la interfaz gráfica, **la interacción y el flujo de la partida continúan gestionándose a través de la terminal**. El desarrollo del turno de cada jugador sigue los siguientes pasos:

1. **Tirada del dado:**
   El jugador activo debe presionar *Enter* para efectuar la tirada del dado.

2. **Comprobación de movimientos posibles:**
   Con el valor obtenido, el sistema verifica qué fichas pueden realizar un movimiento válido.

3. **Presentación de opciones:**
   En caso de existir movimientos disponibles, la terminal mostrará una lista de fichas que pueden desplazarse. Cada opción se presenta con un índice y el número de casilla en la que se encuentra la ficha correspondiente.

4. **Selección de la ficha a mover:**
   El jugador introduce el índice de la ficha que desea mover. El sistema ejecutará el movimiento en el tablero Swing.

5. **Cambio de turno:**
   Tras finalizar el movimiento, el turno pasa automáticamente al siguiente jugador, quien deberá presionar *Enter* para efectuar una nueva tirada y repetir el proceso.