Documentación Entregable 2 - Concurrencia

Integrantes: Barberousse Luciana y Sobral Alfonso

Indicaciones de ejecución

El sistema se ejecuta exclusivamente en modo simulación. Para iniciar una partida se compila y ejecuta la clase Main.java, tras lo cual el desarrollo del juego se visualiza en la terminal. Durante la inicialización, el programa solicita los parámetros de configuración o aplica valores predeterminados según la modalidad seleccionada.

Los parámetros principales son: tamaño del tablero (N), tiempo total de juego (T), cantidad de jugadores (M) y nombres de los jugadores. Como valor sugerido, se emplea un tablero de 10×10, una duración de 60 segundos y 3 o 4 jugadores.

El sistema dispone de un mecanismo de configuración controlado por la variable booleana INPUT_DATA. Cuando INPUT_DATA = false, el programa utiliza valores predefinidos (hardcodeados); cuando INPUT_DATA = true, solicita los parámetros por consola. Entre los parámetros configurables se incluyen los tiempos de operación de los robots (buenos y malos), los tiempos de movimiento de los jugadores, la cantidad de vidas iniciales y los límites de elementos a colocar en el tablero.

Mecánicas de juego

Cada jugador inicia la partida con dos vidas y compite en un tablero controlado por tres tipos de robots que actúan de manera concurrente.

Los elementos presentes en el juego son:

- Robot Bueno (Monedas): se encarga de colocar monedas de valores [1, 2, 5 y 10] puntos en hasta el 10% de las casillas del tablero.
- Robot Bueno (Vidas): añade vidas adicionales hasta alcanzar un límite máximo definido al inicio de la partida.
- Robot Malo (Trampas): distribuye trampas venenosas en el tablero, también en un máximo del 10% de las casillas, y cada una de ellas resta una vida al jugador que la pisa.

La partida finaliza cuando se cumple el tiempo total establecido o cuando únicamente queda un jugador con vida. Una vez concluida, el sistema genera un archivo de texto con el registro de resultados y el ranking de jugadores, ordenado según criterios de supervivencia, cantidad de monedas recolectadas, vidas restantes y, en caso de empate, orden alfabético.

Patrones de diseño concurrente usados y justificar su elección en la documentación

La implementación aplica patrones de concurrencia con el objetivo de garantizar la seguridad de hilos, la consistencia del estado y la eficiencia:

1. Monitor: cada celda del tablero actúa como un monitor mediante ReentrantLock, encapsulando la sincronización y asegurando exclusión mutua sobre su estado. En operaciones críticas (por ejemplo, movimientos atómicos entre celdas) se adquieren los locks necesarios y se liberan en orden inverso dentro de bloques try/finally, evitando condiciones de carrera.
2. Producer–Consumer: los robots funcionan como productores de elementos (monedas, vidas y trampas) y los jugadores como consumidores que recogen o activan dichos elementos al desplazarse. El tablero opera como buffer compartido con sincronización por celdas, desacoplando ritmos de producción y consumo y favoreciendo la escalabilidad.

3. Working Threads: el sistema crea hilos dedicados para jugadores y robots, gestionados y coordinados desde el hilo principal. Se establece una terminación ordenada (graceful shutdown) mediante señales de parada para jugadores e interrupciones para robots.
4. Read–Write (patrón de acceso): se distingue entre operaciones de lectura (consultas de ocupación y contenido de celdas, con lecturas volatile) y operaciones de escritura (movimientos y colocación de elementos), que requieren exclusión mediante locks. Este enfoque optimiza el rendimiento permitiendo múltiples lectores concurrentes.
5. Barrier (sincronización de terminación): al finalizar la simulación, el hilo principal actúa como barrera de terminación coordinando la detención de todos los hilos y esperando su finalización mediante join, lo que asegura una clausura limpia del sistema.
6. Maestro–Trabajadores: el hilo principal cumple el rol de maestro, encargado de crear, iniciar y detener a los trabajadores (jugadores y robots), monitorear las condiciones de finalización y consolidar los resultados.

Dentro del alcance del curso dimos otros patrones que no fueron utilizados. No se implementan Future–Promise ni Actor: el caso de uso no requiere retorno de valores asíncronos individuales ni comunicación por buzones; la arquitectura se basa en memoria compartida con sincronización mediante locks.

Herramientas y técnicas de seguridad de hilos

Se eligió ReentrantLock en lugar de synchronized por ofrecer tryLock, soporte de interrupción y mayor control sobre la adquisición/liberación, incluidos timeouts y condiciones. Para contadores compartidos se utilizaron AtomicInteger, lo que habilita operaciones atómicas lock-free con menor sobrecarga que la sincronización explícita. Las variables volátiles se emplearon como flags y para estado simple, garantizando visibilidad sin incurrir en el coste de secciones críticas cuando no es necesario.

Para evitar deadlocks se aplicó Lock Ordering (orden global de adquisición de locks por identificador de celda) y, cuando correspondía, tryLock con degradación elegante en casos de contención alta. El diseño utiliza un lock por celda, que reduce la contención y permite un alto paralelismo. La generación de aleatoriedad se realiza con ThreadLocalRandom, evitando cuellos de botella por sincronización.

Decisiones de diseño e implementación

La arquitectura es modular y orientada a la separación de responsabilidades:

- Board centraliza el estado compartido y expone operaciones thread-safe.
    - *randomFreeCell():* Encuentra una celda vacía sin jugadores ni elementos
    - *tryPlaceCoin():* Coloca moneda con valor específico en celda libre
    - *tryPlaceHeal():*Coloca vida en celda libre
    - *tryPlaceTrap():* Coloca trampa venenosa en celda libre
    - *movePlayerSafe():* Mueve jugador de forma completamente thread-safe entre celdas capturando tanto bonificaciones como trampas en ellas
    - *placePlayerAtRandom():* Coloca jugador en posición aleatoria al inicio del juego
    - *isOccupied():* Verifica si celda tiene jugador
    - *clearAllPlayerPositions():* Limpia jugador específico de todo el tablero (cuando muere)
    - *cellContent():* Consulta qué elemento hay en una celda
    - *cellCoinAmount():* Sirve para consultar el valor de moneda que hay en la celda
    - *snapshotWithColors():*

- Booster enumeración define los tipos de elementos que pueden aparecer en el tablero del juego y que los jugadores pueden encontrar y recoger durante su movimiento. Un enum es como crear tu propio "tipo de dato personalizado" con un conjunto limitado y predefinido de valores válidos, lo que hace tu código más seguro, legible y mantenible.

- Cell constituye la unidad mínima de sincronización.
    - *lock():* Proporcionar acceso al candado de la celda
    - *content()* y *coinAmount():* Consultar el tipo de elemento y su valor
    - *hasPlayer()* y *player():* Verifica y obtiene el jugador
    - *setPlayer()* y *clearPlayer():* Establece o remueve un jugador de la celda
    - *setCoin() - setHeal() - setPoison():* Coloca cada elemento en la celda
    - *clearContent():* Limpia la celda

- PathPlanner aplica una estrategia de tipo BFS (Breadth First Search) con heurística greedy para orientar desplazamientos hacia recursos valiosos minimizando conflictos.
    - *plan():* Calcula el mejor camino posible desde una posición inicial en un número limitado de pasos.

      Es un algoritmo de planificación de rutas inteligente que combina la exploración sistemática BFS (Breadth-First Search) con heurísticas avanzadas para encontrar el camino óptimo que un jugador debe seguir. Este método recibe una posición inicial (fila y columna), el estado actual del tablero, y un número limitado de pasos, para retornar una secuencia de posiciones que maximiza la puntuación del jugador. El algoritmo utiliza una cola para explorar todos los caminos posibles nivel por nivel, manteniendo un array tridimensional de visitados que previene ciclos y permite revisitar celdas en diferentes momentos del tiempo. Para cada posición candidata, el algoritmo calcula una puntuación sofisticada que considera múltiples factores: el valor de elementos presentes (monedas dan 1-10 puntos, curaciones dan 3 puntos, veneno se ignora), la densidad de jugadores cercanos (penaliza áreas congestionadas con -6 puntos por jugador y bonifica áreas vacías con +12 puntos), un bonus de exploración que incentiva alejarse de la posición inicial, un factor de aleatoriedad que introduce variabilidad comportamental, y una penalización por distancias excesivas que mantiene cohesión en el movimiento. Al final, el algoritmo selecciona y retorna el camino completo que obtuvo la mayor puntuación, convirtiendo cada jugador en un agente inteligente capaz de tomar decisiones estratégicas basadas en el estado dinámico del juego y la presencia de otros jugadores.

- Player implementa la lógica de movimiento y consumo de elementos.
    - *getInitials():* Genera iniciales del nombre para mostrar en el tablero
    - *getColorCode():* Asignar un color único a cada jugador para visualización
    - *setPos() - addCoins() - addLife() - loseLife():* Todos métodos para actualizar los atributos de los jugadores
    - *run():* Es el bucle principal de vida del jugador. Como implementa Runnable, este método se ejecuta cuando el jugador se inicia como un hilo separado. Convierte al jugador en un independiente para jugar automáticamente, tomando decisiones estratégicas y compitiendo con otros jugadores en tiempo real.
    - *stopGracefully():* Detener el jugador de manera controlada
    - *clearFromBoard():* Limpiar todas las referencias del jugador en el tablero

- Pos es un record de Java que representa una posición o coordenada en el tablero del juego. Aunque es extremadamente simple, cumple un papel fundamental en el sistema. Un record es una clase especial de de Java que permite crear datos inmutables. Es perfecta para representar datos que no cambian después de su creación. La clase Pos actúa como un objeto de valor inmutable que:
    - Representa coordenadas del tablero de manera type-safe
    - Se pasa entre hilos sin riesgo de modificación
    - Simplifica el código al encapsular las coordenadas y mejora la legibilidad al hacer explícito qué representa cada parámetro

Pos es un record porque es inmutable, ligero y perfecto para concurrencia, reduce código repetitivo y evita bugs de estado compartido.

- Robot representa robots productores que implementan el patrón Producer-Consumer. Cada robot es un hilo independiente que se encarga de generar y colocar elementos específicos (COIN, HEAL, POISON) en el tablero del juego.
    - *run():* implementa un ciclo de producción continua donde el robot genera elementos específicos en el tablero de manera automática e independiente. En cada iteración, el robot intenta colocar su elemento especializado: si es de tipo COIN genera monedas con valores aleatorios (1, 2, 5, o 10 puntos), si es HEAL produce curaciones, y si es POISON crea trampas, utilizando los métodos thread-safe correspondientes del tablero. Si la colocación es exitosa, el robot entra en una pausa aleatoria entre tmin y tmax milisegundos simulando el tiempo de "fabricación", pero si falla (tablero lleno), implementa una estrategia diferenciada: los robots de monedas persisten reintentando cada 200ms porque las monedas son críticas para el gameplay, mientras que los robots de curaciones y veneno se rinden y terminan su ejecución pues estos elementos tienen límites naturales en el juego. Todo el proceso se ejecuta en un bucle infinito que solo termina cuando el hilo es interrumpido, manejando correctamente las excepciones para garantizar un shutdown elegante del sistema.

La encapsulación de concurrencia dentro de cada clase evita filtrar detalles de sincronización hacia la API pública, lo que simplifica el mantenimiento y las pruebas. Se incorporó un hilo de visualización independiente para imprimir periódicamente el estado del tablero, utilizando códigos ANSI para diferenciar jugadores.

En cuanto a configuración, como ya mencionamos antes, el sistema admite un conmutador (INPUT_DATA) que facilita el paso entre valores predeterminados y entrada por consola. Se implementaron rutinas de validación de entrada con valores por defecto de respaldo, y un mecanismo de terminación controlada a través de flags volátiles.

Estas rutinas de validación de entrada lo que hacen es que en lugar de lanzar los errores o crashea, utiliza valores por defecto cuando la entrada es inválida según el Input. Esto se debe a que nos interesaba mantener los valores dentro de cierto márgen lógico para el juego.
Luego podemos mencionar algo acerca del algoritmo greedy para definir los caminos a tomar por cada jugador.

- Busca recursos valiosos como monedas o vidas en un rango de movimiento según lo sacado en el dado.
- Evita áreas con múltiples jugadores para evitar concurrencia innecesaria.
- Penaliza proximidad a otros jugadores para reducir conflictos.
- Optimiza trayectorias considerando el número de pasos del dado (1-6).

Visualización de bonificaciones en el tablero: Es importante tener en cuenta que las bonificaciones (monedas, vidas adicionales y trampas) pueden tardar en visualizarse correctamente en el tablero debido a que el sistema de renderizado gráfico se actualiza más rápidamente que el sistema de impresión en consola. Esta diferencia de velocidad puede crear un desfase temporal entre el momento en que un jugador recoge una bonificación y cuando esta desaparece visualmente del tablero. El comportamiento del juego sigue siendo correcto a nivel lógico, ya que las bonificaciones se procesan inmediatamente cuando un jugador las pisa, pero la representación visual puede mostrar un ligero retraso. Adicionalmente, las trampas (representadas como "T" en el tablero) reducen únicamente 1 vida por cada paso que el jugador realiza sobre ellas, manteniendo un equilibrio en la mecánica de juego.

Las bonificaciones pueden aparecer y ser recolectadas por los jugadores entre los intervalos de actualización visual del tablero, lo que puede crear la impresión de que los jugadores recogen elementos que no se alcanzaron a visualizar en pantalla.

Análisis de rendimiento y corrección

El uso de locks por celda permite paralelizar operaciones independientes, lo que se traduce en alta concurrencia y baja contención. La latencia de operaciones puntuales se mantiene acotada, y el rendimiento escala con el tamaño del tablero y el número de hilos.
Desde la perspectiva de corrección, el orden global de adquisición de locks elimina esperas circulares y garantiza ausencia de deadlocks por construcción. La combinación de volatile, tipos atómicos y secciones críticas con ReentrantLock preserva la consistencia de memoria, asegurando que los hilos observen un estado coherente del sistema.

Conclusiones

El proyecto materializa un juego concurrente funcional y robusto, capaz de coordinar múltiples hilos con seguridad y eficiencia. Destacan la aplicación sistemática de patrones de concurrencia (Monitor, Producer–Consumer, Working Threads, Read–Write access, Barrier y Maestro–Trabajadores), la ausencia de deadlocks gracias al ordenamiento de locks, y una arquitectura modular que favorece la extensibilidad y el mantenimiento.

La robustez del sistema se fundamenta en una arquitectura de sincronización multicapa que previene sistemáticamente los problemas típicos de concurrencia mediante estrategias complementarias y bien coordinadas. El diseño implementa fine-grained locking donde cada celda del tablero posee su propio ReentrantLock, maximizando el paralelismo al permitir operaciones simultáneas en diferentes regiones del tablero, mientras que el lock ordering (ordenamiento consistente de candados por coordenadas) elimina completamente la posibilidad de deadlocks al garantizar que todos los hilos adquieren múltiples locks en la misma secuencia. Los campos volatile (running, lives, coins, content) aseguran la visibilidad inmediata de cambios críticos entre hilos sin necesidad de sincronización explícita, complementados por el uso de ThreadLocalRandom para operaciones atómicas thread-safe y un mecanismo de graceful shutdown que coordina la terminación elegante de todos los hilos mediante flags de control e interrupciones controladas. La delegación de responsabilidades thread-safe al Board como hub central, combinada con el patrón de external synchronization donde los métodos de modificación requieren sincronización externa, crea un ecosistema donde múltiples productores (robots) y consumidores (jugadores) pueden interactuar de manera segura y eficiente, resultando en un sistema concurrente que mantiene la integridad de datos, previene condiciones de carrera, evita deadlocks y garantiza terminación correcta bajo cualquier escenario de ejecución.

En términos formativos, se consolidaron competencias en prevención de condiciones de carrera, diseño de componentes thread-safe, uso de operaciones atómicas y balance entre rendimiento y corrección. Estos conocimientos resultan transferibles a sistemas reales de alta concurrencia, como servidores web, sistemas distribuidos, aplicaciones de simulación y entornos de tiempo real.