Parte II

1.) Análisis de concurrencia

- Explica cómo el código usa hilos para dar autonomía a cada serpiente.

  RTA: El juego implementa concurrencia asignando un hilo independiente por cada serpiente. Cada serpiente es ejecutada mediante una instancia de SnakeRunner, la cual implementa Runnable y es enviada a ejecución usando un executor de virtual threads.

  Se logro identificar que cada hilo decide cambios de dirección de forma autónoma, invoca a board.step(snake) lo cual hace que avance en el tablero y controle su propia velocidad que puede ser normal o turbo mediante Thred.sleep. Esto permite que todas las serpientes se muevan de forma concurrente e independiente, sin depender unas de otras para avanzar.

- Posibles condiciones de carrera.
  RTA: Seria el acceso concurrente que no está protegiendo al cuerpo de la serpiente ya que tenemos 2 hilos que están trabajando con el mismo objeto
  Snake:
  1. El hilo del juego SnakeRunner que mueve la serpiente y actualiza su posición.
  2. El hilo de la interfaz gráfica que dibuja la serpiente en la pantalla y procesa las teclas que se presionan.

  Esto hace que la estructura de datos que almacena el cuerpo de la serpiente Deque<Position> body, no tenga protección contra el acceso simultaneo, lo cual causa que mientras la interfaz este dibujando la serpiente y leyendo las posiciones, el hilo del juego podría estar moviendo la serpiente modificando las posiciones. El resultado de esto es que la interfaz podría dibujar una serpiente partida o en un estado raro, mostrando posiciones viejas y algunas nuevas al mismo tiempo.

  Otro problema sería la dirección de la serpiente ya que el atributo de dirección es volatile, lo que garantiza visibilidad. Pero el método turn() y su uso concurrente pueden generar actualizaciones intercaladas, aunque el impacto es menor.

- Colecciones o estructuras no seguras en contexto concurrente.
  RTA: Las colecciones del tablero no son seguras entre ellas ya que oos atributos mice, obstacles, turbo, teleports son HashSet y HashMap, estas colecciones no son thread-safe por lo que no están diseñadas para ser accedidas concurrentemente por múltiples hilos sin sincronización externa.

  Las listas de las serpientes ya que es compartida entre los hilos de ejecución de las serpientes y el hilo de la interfaz gráfica, que la recorre para pintar el estado del juego.

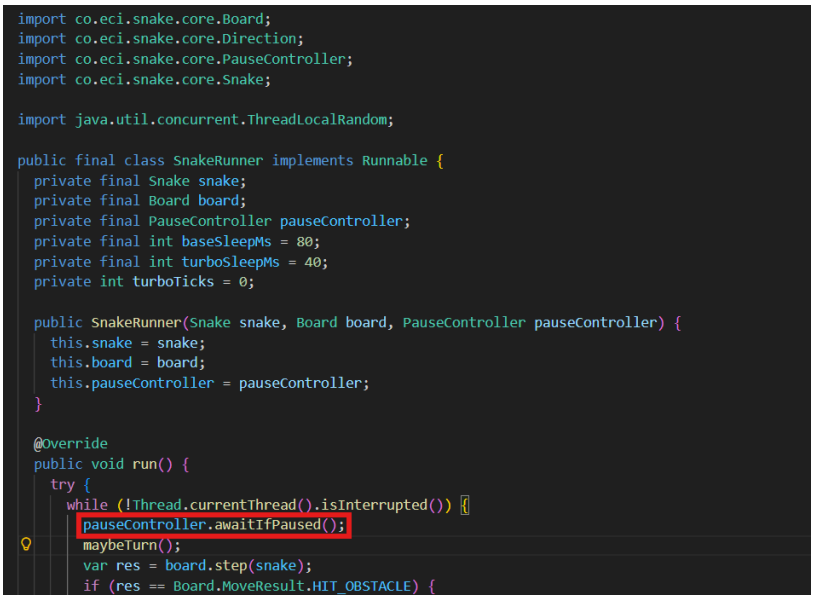
Aunque la lista no se modifica después de la inicialización, sigue siendo accedida concurrentemente por múltiples hilos.

- Ocurrencias de espera activa (busy-wait) o de sincronización innecesaria.
  RTA: SnakeRunner usa Thread.sleep() dentro de un ciclo continuo, lo que hace que las serpientes sigan ejecutándose incluso cuando el juego está pausado. Aunque no hay busy-wait, existe actividad innecesaria durante la pausa. El GameClock solo controla el repintado de la UI y no la lógica de movimiento. Esto provoca un desacople entre la pausa visual y la ejecución real del juego, generando una sincronización incompleta.

2.) Correcciones mínimas y regiones críticas

- Elimina esperas activas reemplazándolas por señales/estados o mecanismos de la librería de concurrencia.
  RTA: El riesgo que se identifico es que los SnakeRunner usaban Thread.sleep() sin respetar el estado global de pausa, provocando actividad innecesaria y desincronización entre lógica y UI.

  El cambio para solucionarlo fue implementar la clase PauseController con wait()/notifyAll() asi cada SnakeRunner llama awaitIfPaused() al inicio de su bucle y queda bloqueado eficientemente cuando el juego está en pausa.

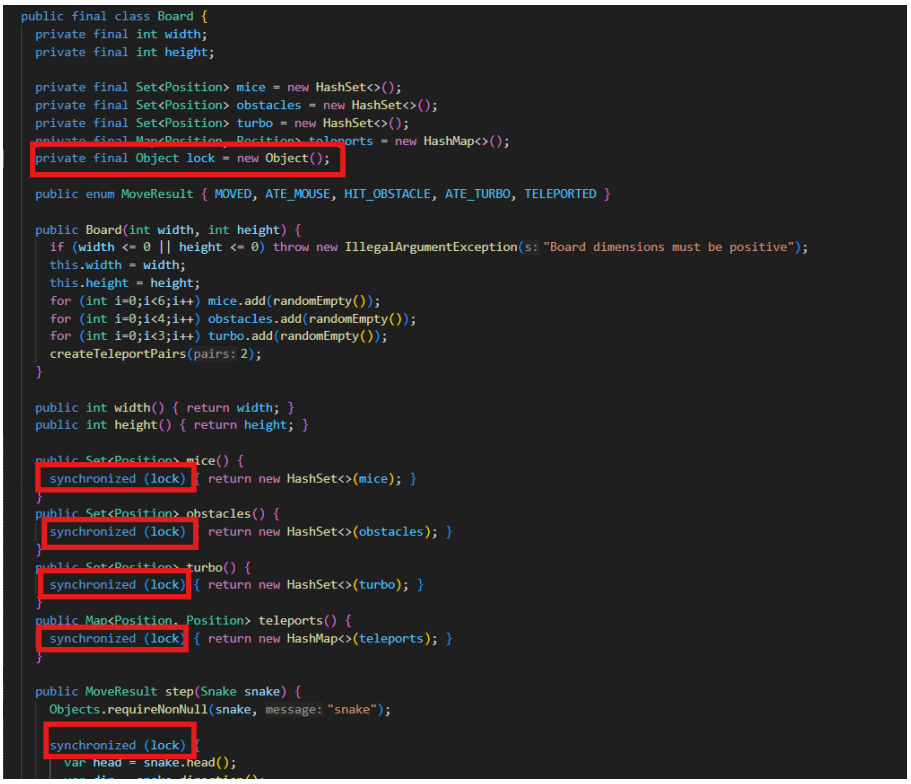


```
import co.eci.snake.core.Board;
import co.eci.snake.core.Direction;
import co.eci.snake.core.PauseController;
import co.eci.snake.core.Snake;

import java.util.concurrent.ThreadLocalRandom;

public final class SnakeRunner implements Runnable {
  private final Snake snake;
  private final Board board;
  private final PauseController pauseController;
  private final int baseSleepMs = 80;
  private final int turboSleepMs = 40;
  private int turboTicks = 0;

  public SnakeRunner(Snake snake, Board board, PauseController pauseController) {
    this.snake = snake;
    this.board = board;
    this.pauseController = pauseController;
  }

  @Override
  public void run() {
    try {
      while (!Thread.currentThread().isInterrupted()) {
        pauseController.awaitIfPaused();
        maybeTurn();
        var res = board.step(snake);
        if (res == Board.MoveResult.HIT_OBSTACLE) {
```

```java
package co.eci.snake.core;

public final class PauseController {
  private volatile boolean paused = false;

  public void pause() {
    synchronized (this) {
      paused = true;
    }
  }

  public void resume() {
    synchronized (this) {
      paused = false;
      this.notifyAll();
    }
  }

  public void awaitIfPaused() throws InterruptedException {
    synchronized (this) {
      while (paused) {
        this.wait();
      }
    }
  }

  public boolean isPaused() {
    return paused;
  }
}
```

El siguiente problema eran las regiones críticas en la clase del tablero Board, ya que no tenían protección las colecciones del tablero esto provocaba que puedan
sufrir corrupciones o ConcurrentModificationException.

Se resolvió implementando un objeto lock y sincronización únicamente sobre la porción de código que se accede y modifica estas colecciones y llama snake.advance y los getters devuelven copias bajo el mismo lock.



El tercer riesgo era que los métodos snapshot() y advance() leían la variable body el cuerpo de la serpiente sin protección. Esto causaba que cuando se dibujaba la serpiente en pantalla, a veces veías estados raros o incompletos porque un hilo estaba actualizando body mientras otro lo leía.

Para solucionarlo solo protegemos con synchronized los 3 métodos que acceden a body head(), snapshot() y advance() asi evitamos que se lean estados inconsistentes al dibujar. La variable direction la dejamos como volatile que se actualiza sola entre hilos.

```
public Position head() { synchronized (body) { return body.peekFirst(); } }

public Deque<Position> snapshot() { synchronized (body) { return new ArrayDeque<>(body); } }

public void advance(Position newHead, boolean grow) {
  synchronized (body) {
    body.addFirst(newHead);
    if (grow) maxLength++;
    while (body.size() > maxLength) body.removeLast();
  }
 }
}
```

3.) Control de ejecución seguro (UI)
RTA: La UI llama a pause() para detener todos los hilos de serpientes. Y Esperar sincronización de modo que un coordinador espera a que todas las serpientes alcancen su punto de pausa, y solo cuando todo este detenido muestra la serpiente viva más larga y la primera serpiente que murió.

Mejoramos la clase PauseController para soportar conteo de runners y el método waitForAllPaused()

```
package co.eci.snake.core;

public final class PauseController {
  private volatile boolean paused = false;
  private int totalRunners = 0;
  private int pausedCount = 0;

  public synchronized void setTotalRunners(int totalRunners) {
    this.totalRunners = totalRunners;
    this.notifyAll();
  }

  public synchronized void pause() {
    paused = true;
  }


  public synchronized void resume() {
    paused = false;
    this.notifyAll();
  }
```

```java
public void awaitIfPaused() throws InterruptedException {
  synchronized (this) {
    if (!paused) return;
    pausedCount++;
    this.notifyAll();
    try {
      while (paused) {
        this.wait();
      }
    } finally {
      pausedCount--;
      this.notifyAll();
    }
  }
}

public synchronized void waitForAllPaused() throws InterruptedException {
  while (paused && pausedCount < totalRunners) {
    this.wait();
  }
}


public boolean isPaused() {
  return paused;
}
}
```

- Añadimos una detección de auto colisión cuando la cabeza entra en una posición ya ocupada por su cuerpo, por medio de 2 atributos para marcar alive=false y guardamos deathTime la primera vez que mueren.

```java
public final class Snake {
  private final Deque<Position> body = new ArrayDeque<>();
  private volatile Direction direction;

  private volatile boolean alive = true;
  private volatile long deathTime = 0L;
```

```java
public Position head() { synchronized (body) { return body.peekFirst(); } }

public Deque<Position> snapshot() { synchronized (body) { return new ArrayDeque<>(body); } }

public boolean isAlive() { return alive; }

public long getDeathTime() { return deathTime; }

public void advance(Position newHead, boolean grow) {
  synchronized (body) {
    if (!alive) return;
    boolean selfCollision = body.contains(newHead);
    body.addFirst(newHead);
    if (grow) maxLength++;
    while (body.size() > maxLength) body.removeLast();
    if (selfCollision && alive) {
      alive = false;
      deathTime = System.currentTimeMillis();
    }
  }
}
}
```

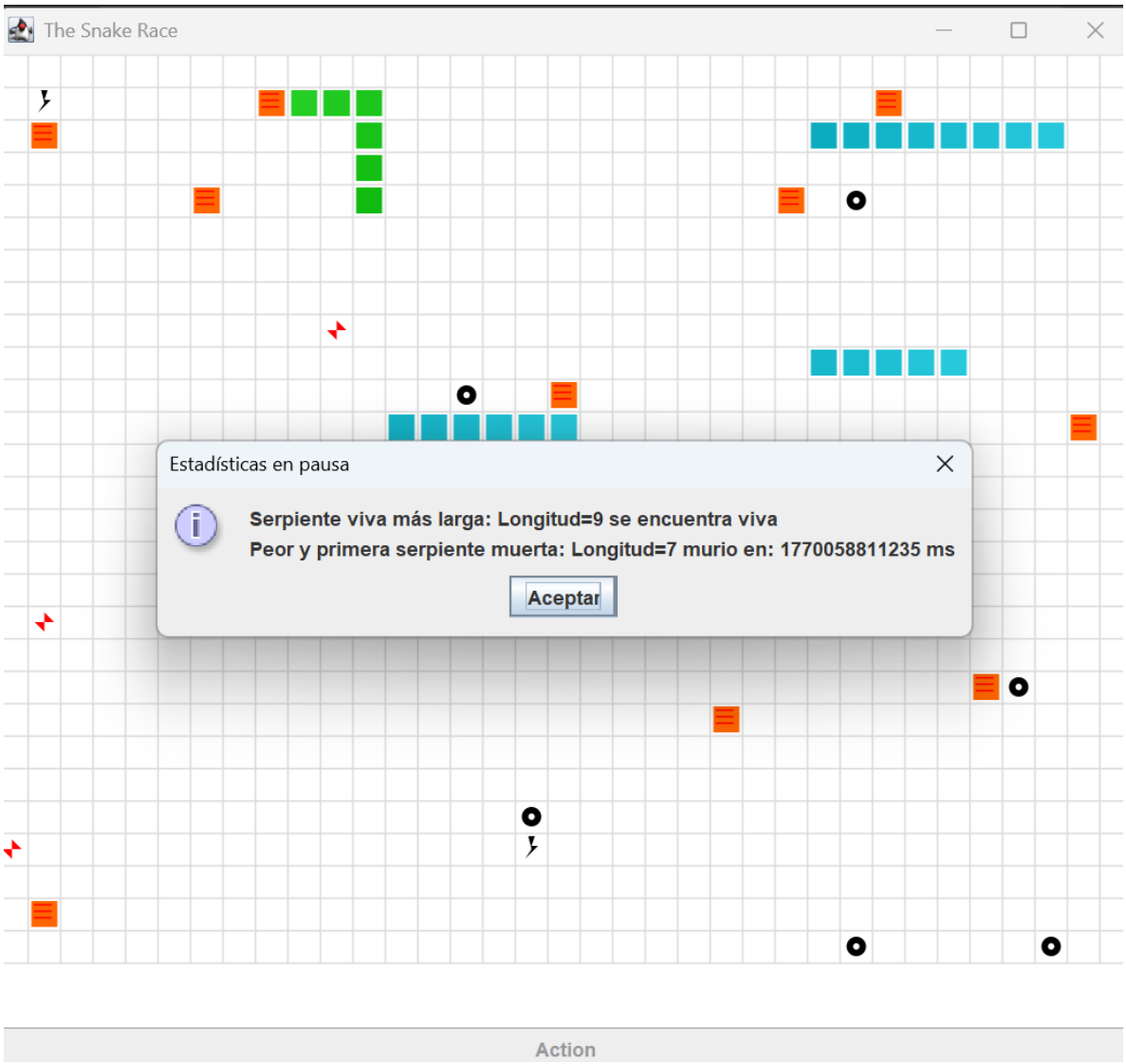
- Luego modificamos SnakeApp para coordinar pausa y mostrar estadísticas de forma segura.

```
private void togglePause() {
  if ("Action".equals(actionButton.getText())) {
    actionButton.setEnabled(b: false);
    new Thread(() -> {
      pauseController.pause();
      try {
        pauseController.waitForAllPaused();
      } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
      }
      var longest = computeLongestAliveSnake();
      var worst = computeFirstDiedSnake();
      SwingUtilities.invokeLater(() -> {
        clock.pause();
        String msg = "Serpiente viva más larga: " + (longest != null ? formatSnakeInfo(longest) : "N/A")
                   + "\nPeor y primera serpiente muerta: " + (worst != null ? formatSnakeInfo(worst) : "N/A");
        JOptionPane.showMessageDialog(this, msg, title: "Estadísticas en pausa", JOptionPane.INFORMATION_MESSAGE);

        actionButton.setText(text: "Resume");
        actionButton.setEnabled(b: true);
      });
    }, name: "pause-coordinator").start();

  } else {
    pauseController.resume();
    clock.resume();
    actionButton.setText(text: "Action");
  }
}

  private String formatSnakeInfo(Snake s) {
  var body = s.snapshot();
  return "Longitud=" + body.size() + (s.isAlive() ? " se encuentra viva" : " murio en: " + s.getDeathTime() + " ms");
}
```

```
private Snake computeLongestAliveSnake() {
  Snake best = null;
  int bestLen = -1;
  for (Snake s : snakes) {
    if (!s.isAlive()) continue;
    int len = s.snapshot().size();
    if (len > bestLen) {
      bestLen = len;
      best = s;
    }
  }
  return best;
}


private Snake computeFirstDiedSnake() {
  Snake worst = null;
  long minDeath = Long.MAX_VALUE;
  for (Snake s : snakes) {
    long dt = s.getDeathTime();
    if (dt > 0 && dt < minDeath) {
      minDeath = dt;
      worst = s;
    }
  }
  return worst;
}
```

- Evidencia:

Los resultados observados confirman que el sistema de pausa implementado sincroniza correctamente la lógica del juego y la interfaz gráfica.



4.) Robustez bajo carga

Ejecuta con **N alto** (-Dsnakes=20 o más) y/o aumenta la velocidad.

El juego **no debe romperse**: sin ConcurrentModificationException, sin lecturas inconsistentes, sin *deadlocks*.

Si habilitas **teleports** y **turbo**, verifica que las reglas no introduzcan carreras.

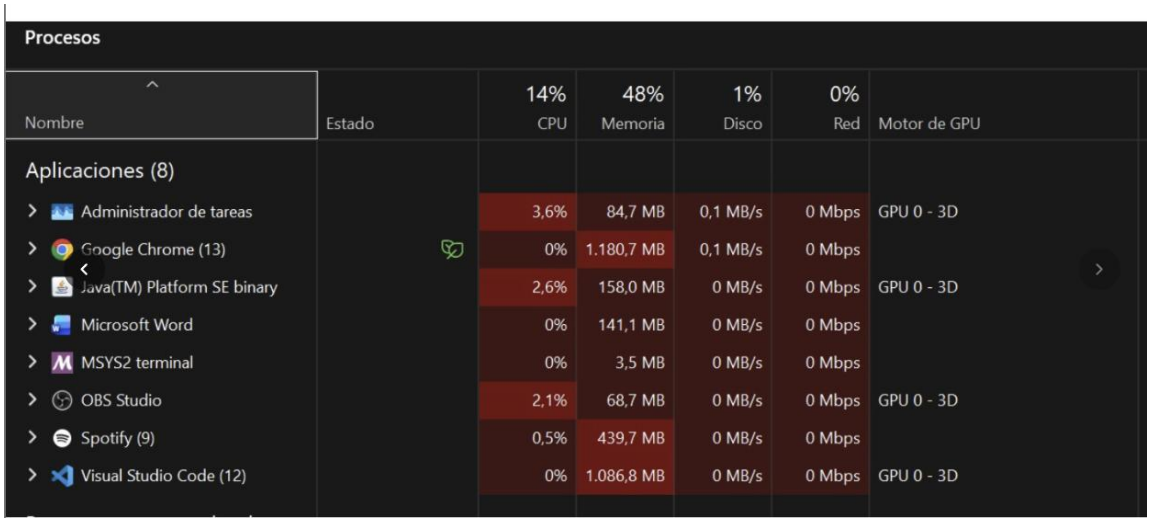
Respuesta:

- https://youtu.be/mC0HA0rPofo

El sistema fue ejecutado con 60 serpientes concurrentes, cada una controlada por un hilo virtual independiente.
Se redujo el tiempo de espera entre movimientos a 20 ms y, bajo efecto turbo, a 10 ms, incrementando significativamente la frecuencia de acceso concurrente al estado compartido del tablero.

Durante las pruebas se mantuvieron activas todas las reglas del juego: teletransportadores, turbo, obstáculos dinámicos y wrap-around.
El sistema fue ejecutado durante varios minutos sin presentar excepciones de concurrencia, bloqueos, ni inconsistencias visibles en el estado del juego o en la interfaz gráfica.

Estos resultados evidencian que las regiones críticas fueron correctamente protegidas y que el diseño concurrente es estable bajo condiciones de alta carga.



```java
// Teleports (flechas rojas)
Map<Position, Position> tp = board.teleports();
g2.setColor(Color.RED);
for (var entry : tp.entrySet()) {
  Position from = entry.getKey();
  int x = from.x() * cell, y = from.y() * cell;
  int[] xs = { x + 4, x + cell - 4, x + cell - 10, x + cell - 10, x + 4 };
  int[] ys = { y + cell / 2, y + cell / 2, y + 4, y + cell - 4, y + cell / 2 };
  g2.fillPolygon(xs, ys, xs.length);
}

// Turbo (rayos)
g2.setColor(Color.BLACK);
for (var p : board.turbo()) {
  int x = p.x() * cell, y = p.y() * cell;
  int[] xs = { x + 8, x + 12, x + 10, x + 14, x + 6, x + 10 };
  int[] ys = { y + 2, y + 2, y + 8, y + 8, y + 16, y + 10 };
  g2.fillPolygon(xs, ys, xs.length);
}
```