ARSW — (Java 21): **Immortals & Synchronization** — con UI Swing

Escuela Colombiana de Ingeniería – Arquitecturas de Software
Laboratorio de concurrencia: condiciones de carrera, sincronización, suspensión cooperativa y *deadlocks*, con interfaz **Swing** tipo *Highlander Simulator*.

Asignatura: Arquitectura de Software
Estudiantes:

- Alexandra Moreno
- Alison Valderrama
- Jeisson Sánchez
- Valentina Gutierrez

Requisitos

- **JDK 21** (Temurin recomendado)
- **Maven 3.9+**
- SO: Windows, macOS o Linux

Parámetros

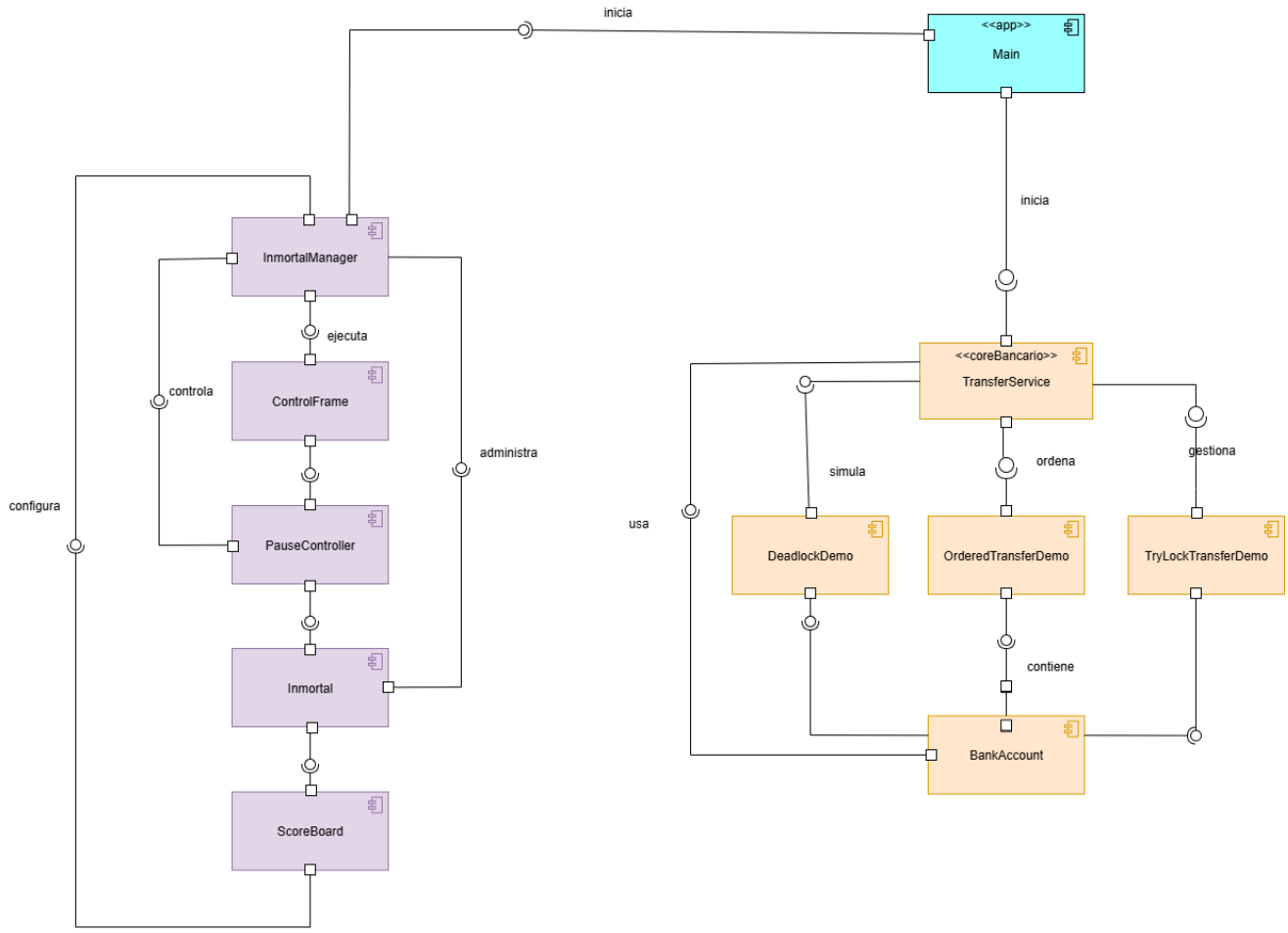
- `-Dcount=N` → número de inmortales (por defecto 8)
- `-Dfight=ordered|naive` → estrategia de pelea (`ordered` evita *deadlocks*, `naive` los puede provocar)
- `-Dhealth`, `-Ddamage` → salud inicial y daño por golpe

Controles en la UI

- **Start**: inicia una simulación con los parámetros elegidos.
- **Pause & Check**: pausa **todos** los hilos y muestra salud por inmortal y **suma total** (invariante).
- **Resume**: reanuda la simulación.
- **Stop**: detiene ordenadamente.

Invariante: con N jugadores y salud inicial H, la **suma total** de salud debe permanecer constante (salvo durante un update en curso). Usa **Pause & Check** para validarlo.

Infraestructura



Actividades del laboratorio

Parte I — (Antes de terminar la clase) `wait/notify`: Productor/Consumidor

1. Ejecuta el programa de productor/consumidor y monitorea CPU con **jVisualVM**. ¿Por qué el consumo alto? ¿Qué clase lo causa?
2. Ajusta la implementación para **usar CPU eficientemente** cuando el **productor es lento** y el **consumidor es rápido**. Valida de nuevo con VisualVM.
3. Ahora **productor rápido** y **consumidor lento** con **límite de stock** (cola acotada): garantiza que el límite se respete **sin espera activa** y valida CPU con un stock pequeño.

> Usa monitores de Java: `synchronized` + `wait()` + `notify/notifyAll()`, evitando *busy-wait*.

Parte II — (Antes de terminar la clase) Búsqueda distribuida y condición de parada

Reescribe el **buscador de listas negras** para que la búsqueda **se detenga tan pronto** el conjunto de hilos detecte el número de ocurrencias que definen si el host es confiable o no (`BLACK_LIST_ALARM_COUNT`). Debe:

- **Finalizar anticipadamente** (no recorrer servidores restantes) y **retornar** el resultado.

- Garantizar **ausencia de condiciones de carrera** sobre el contador compartido.

> Puedes usar `AtomicInteger` o sincronización mínima sobre la región crítica del contador.

Parte III — Respuestas y cambios implementados

Para habilitar una pausa cooperativa consistente, permitir snapshots seguros sin bloquear la simulación y mejorar la parada ordenada. A continuación se resumen las modificaciones

- Archivos modificados:
  - `src/main/java/edu/eci/arsw/concurrency/PauseController.java`
    - Se realiza conteo de hilos pausados (`pausedThreads`) y método `waitForAllPaused(int expected, long timeoutMillis)` para que la UI espere hasta que los hilos alcancen la pausa cooperativa.
  - `src/main/java/edu/eci/arsw/immortals/ImmortalManager.java`
    - `population` ahora es `CopyOnWriteArrayList<Immortal>` para permitir snapshots seguros y remociones sin sincronización global.
    - `stop()` mejora la parada cooperativa y limpia futuros.
  - `src/main/java/edu/eci/arsw/highlandersim/ControlFrame.java`
    - `onPauseAndCheck` espera (hasta 2s) a que todos los hilos lleguen a la pausa antes de tomar el snapshot; si expira el timeout se informa en la UI.
- Cómo validar (pasos rápidos):
  1. Compilar y ejecutar UI:

     ```
     mvn -DskipTests exec:java -
     Dexec.mainClass=edu.eci.arsw.highlandersim.ControlFrame -Dcount=8 -
     Dfight=ordered -Dhealth=100 -Ddamage=10
     ```

  2. Click `Start`.
  3. Click `Pause & Check` — la UI esperará hasta 2s a que los hilos lleguen al punto de pausa cooperativa; luego muestra la salud por inmortal, la suma total y el contador de fights.
  4. Click `Resume` para reanudar, `Stop` para detener ordenadamente.
- Invariante:
  - La suma total de salud debe permanecer constantE. Sin embargo, en el código actual la operación de pelea está implementada como:

    ```
    other.health -= damage;
    this.health += damage / 2;
    ```

por lo que cada pelea provoca un cambio neto en la suma total igual a `-damage/2` (la suma total decrece). Por tanto, con la implementación actual la suma NO se mantiene constante.
    - La suma del invariante debe cambiar la regla de pelea a una transferencia cero-suma — por ejemplo:

```
other.health -= damage;
this.health += damage; // transferencia total
```

      o cualquier regla donde lo restado al oponente sea exactamente lo sumado al atacante.
    - Con la implementación original del enunciado (si asumimos transferencia total) el valor esperado es `Total = N * H` (donde N = número de inmortales, H = salud inicial). En el estado actual, la suma esperada decrece con cada pelea en `damage/2`.

- Resumen del estado de requisitos (Parte III):
    - Pausa correcta (esperar antes de leer): Done — `PauseController.waitForAllPaused(...)` y espera en `ControlFrame`.
    - Resume: Preservado (ya existía) — Done.
    - Regiones críticas / deadlocks: El código mantiene `fightNaive` (potencial deadlock) y `fightOrdered` (evita deadlocks por orden total). No se cambiaron las estrategias de pelea — Done (existing).
    - Diagnóstico de deadlocks: Mantener `jps`/`jstack` como herramienta — Deferred (herramienta externa).
    - STOP (apagado ordenado): Mejorado — `stop()` solicita parada y apaga el executor; se puede añadir `awaitTermination` para esperar completitud — Partial/Done.
    - Para validar invariante con `Pause & Check` en modo experimental, usamos `-Dfight=ordered` para evitar deadlocks. Si la suma no se mantiene, se revisa la regla de pelea
    - Para remover inmortales muertos de forma no bloqueante en simulaciones grandes se implementa una tarea que periódicamente recorra la `CopyOnWriteArrayList` y elimine los que `!isAlive()` (la operación de eliminación en `CopyOnWriteArrayList` es segura, aunque costosa).

Cómo correr pruebas

```
mvn clean verify
```

Incluye compilación y ejecución de pruebas JUnit. Si tienes análisis estático, ejecútalo en `verify` o `site` según tu `pom.xml`.

Créditos

Este laboratorio es una adaptación modernizada del ejercicio **Immortals &&Synchronization** de ARSW. El enunciado de actividades se conserva para mantener los objetivos pedagógicos del curso.

Base construida por el Ing. Javier Toquica.

ECI-ARSW Team

Empowering well-being through technology