UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

Práctica 3: Calabozoss

ALUMNOS

Rivera Zavala Javier Alejandro - 311288876
López Diego Gabriela - 31824385
Diego Martínez Calzada - 318275457

PROFESOR

Gilde Valeria Rodríguez Jiménez

AYUDANTES

Rogelio Alcantar Arenas
Gibran Aguilar Zuñiga
Luis Angel Leyva Castillo

ASIGNATURA

Computación Concurrente

1. Cuestionario

1. Tu solución propuesta cumple:

   - Exclusión Mutua
     Sí, nuestra implementación cumple con exclusión mutua pues delega la responsabilidad de gestionar la entrada a la sección critica directamente sobre el candado de tipo *ReentrantLock*. Dado que no hay otra condicional o mecanismo que gestione el acceso a la sección crítica, más allá de una verificación para saber si el vocero ya anunció que pasaron todos y que los candados de tipo *ReentrantLock* garantizan la exclusión mutua. Observemos la porción del código donde se hace uso del lock, donde además se aprecia que unlock siempre se ejecuta y por lo tanto el candado siempre se libera:

```
1     public void run() {
2         final Logger LOG = Logger.getLogger("kass.concurrente.modelos.
              Habitacion");
3         Random r = new Random();
4         Prisionero p = prisioneros.get(Integer.parseInt(Thread.currentThread
              ().getName()));
5         while (!yaDijoLaFrase) {
6         // Variable aleatoria que nos ayuda a darle una pseudoaleatoria a la
               forma en
7         // que pasan los prisioneros
8         int aux = r.nextInt(2);
9         lock.lock();
10        try{
11
12        }finally{
13        lock.unlock;
14        }
15        ...
```

     Supongamos que la implementación presentada no cumple con exclusión mutua y que 2 hilos, hilo1 e hilo2, ingresaron a la sección crítica. Esto nos lleva a una contradicción pues ello supondría que lock dejo pasar a ambos, lo cuál no es posible pues lock garantiza la exclusión mutua.

   - No Deadlock
     Por motivos similares al caso del inciso anterior, podemos afirmar que nuestra implementación es No deadlock. Esto se debe a que, cómo mencionamos antes, no hay estructuras de control u otros candados que le proporcionen a los hilos el acceso a la sección critica al mismo tiempo. Supongamos que al menos 2 hilos se quedan esperando indefinidamente ya que se bloquean entre sí, eso contradice el hecho de que "lock" es No deadlock, por lo tanto no es posible que nuestra implementación produzca un deadlock entre los hilos. en conclusión, nuestro programa es No deadlock.

   - Libre de Hambruna
     Supongamos que hay un hilo que se queda esperando indefinidamente para poder tomar el control del lock y así poder acceder a la sección crítica, eso quiere decir que se quedó esperando en la línea "lock.lock()", pues antes sólo está la verificación de la condición del while y la inicialización de elementos necesarios para la sección crítica. Lo anterior contradice el hecho de que los candados de tipo *ReentrantLock*, son candados de tipo Starvation-free, por lo tanto nuestra suposición ha de ser falsa y entonces nuestra implementación garantiza Starvation-free.

   - En los 2 casos anteriores, la cosa buena que sucede eventualmente y la cosa mala que nunca nunca sucede son inmediatas: nunca 2 hilos o más se bloquean entre sí y todo hilo logra acceder a la sección crítica eventualmente.

   De ser así, demuestra cada propiedad.

2. ¿Tu solución cumple para n prisioneros?
   Sí, basta con modificar el valor PRISIONEROS dentro del archivo de constantes.

3. Si tu programa tarda mucho en terminar, ¿Por qué crees que pasa esto?
   Relativamente sí, dado lo que hemos ido viendo en clase, puede que los tiempos de espera marcados para esta implementación, entre paso y paso de un "prisionero" y otro, sean demasiado largos, lo cuál produce una ligera ralentización. Otra posible causa sería el uso de una estructura inadecuada para guardar a los prisioneros, es decir, usamos una *LinkedList*, misma que nos brinda un tiempo de acceso lineal a sus compartimentos, pues requiere atravesar toda la lista para acceder a una posición aleatoria. Por último, habría que considerar si la idea de simular el acceso a la habitación a través de un objeto es la más eficiente, quizás hacerlo en una forma más procedural podría darnos una ganancia en tiempo.

4. De la pregunta anterior, que podría proponer para mejorar los tiempo.
   De mi respuesta anterior se deducen un poco mis propuestas, pero básicamente habría que reducir el tiempo de espera dentro de la habitación para cada prisionero, también cambiar la *LinkedList* por una *ArrayList* y por último, hacer una simulación más directa del paso por la habitación de los prisioneros, sé que esta es una implementación que sigue mejor el POO, pero quizás para esta ocasión más simple es mejor. Como extra hay que señalar que hicimos uso de Random para generar las posiciones aleatorias, para esta práctica quizás hubiese resultado más conveniente usar ThreadLocalRandom, esto debido a que generar números pseudo-aleatorios con Random es costoso en tiempo y utilizamos esa herramienta constantemente.

5. Por ultimo, analiza bien el siguiente enunciado: "Si los candados cumplen con exclusión mutua, no deadlock o libre de hambruna, es decir, con las propiedades para un candado seguro, entonces el sistema donde lo utilicemos también las cumplirá." De esto justifica porque si se cumple o porque no.
   Esto no es cierto, según que estructuras de control se empleen, donde se haga la petición para tomar el control del candado o para liberarlo, e incluso, según cuantos candados utilizamos, el programa puede no ser seguro. Pondremos el siguiente ejemplo que nos muestra como a pesar de utilizar candados seguros, que por ende son No deadlock, se produce un deadlock debido a nuestra implementación:

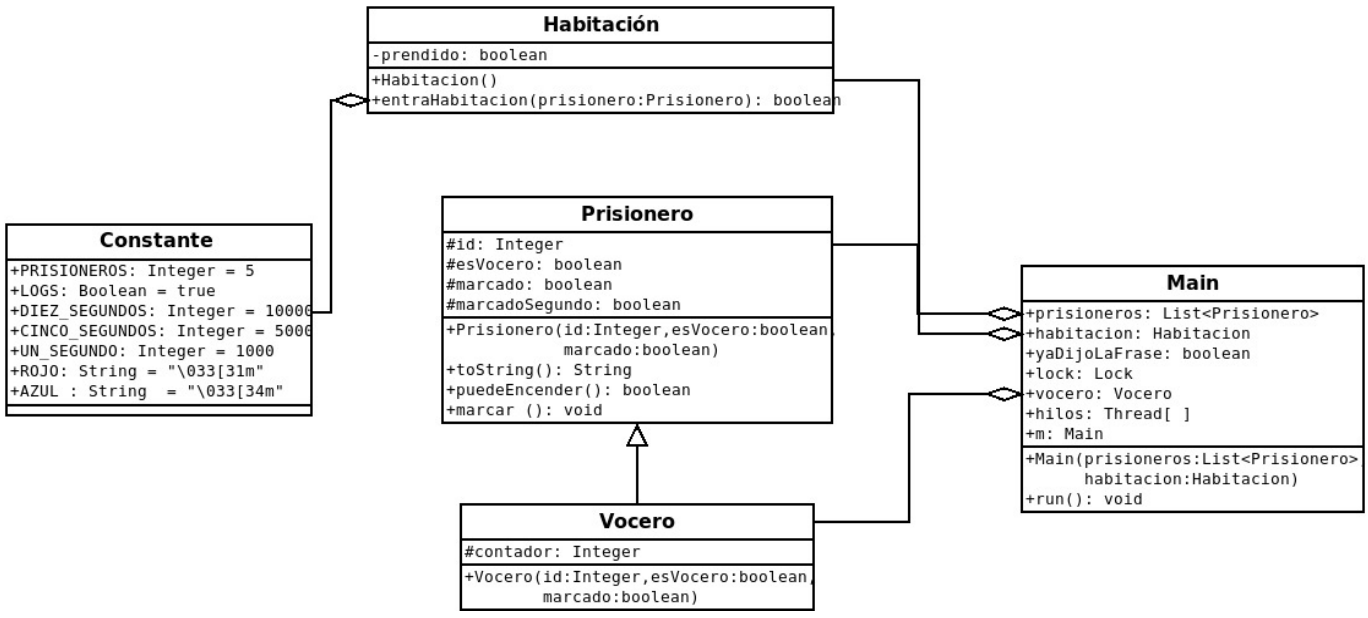
```java
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class MiClase {

    // Candados
    private static final Lock candado1 = new ReentrantLock();
    private static final Lock candado2 = new ReentrantLock();

    public static void main(String[] args) {

        Thread hilo1 = new Thread(() -> {
            tomaCandado(candado1, candado2);
        });

        Thread hilo2 = new Thread(() -> {
            tomaCandado(candado2, candado1);
        });

        // Iniciar los hilos
        hilo1.start();
        hilo2.start();

    }

    // Metodo para adquirir recursos
    public static void tomaCandado(Lock lock1, Lock lock2) {
        try {
            lock1.lock();
            //Alguna operacion, digamos imprimir algo en pantalla.
            lock2.lock();
```

```
32              //Lo mismo que antes, se hacen algunas operaciones con los candados
                    ya tomados.
33          } catch (InterruptedException e) {
34              e.printStackTrace();
35          } finally {
36              lock2.unlock();
37              lock1.unlock();
38          }
39      }
40  }
```

En este caso, supongamos que hilo1 e hilo2 intentan tomar los candados al mismo tiempo, digamos que hilo1 toma el primer candado y entonces realiza algunas operaciones, entonces sucede que hilo2 al no poder tomar al primer candado, toma al segundo y realiza algunas operaciones pero se queda a la espera a que hilo1 libere al primer candado y en cambio hilo1 se queda esperando a que hilo2 libere al segundo, de modo que se bloquean entre sí. Lo anterior sucede ya que cada hilo necesita del candado que tiene el otro para así poder liberar al que tiene, pero no lo libera mientras no pueda tomar el del otro y así sucesivamente. Espero que con este ejemplo quede claro por que el uso de candados seguros no garantiza un acceso seguro a secciones críticas del código, hay que ser precavidos en como se usan.

6. Añade lo aprendido en esta practica, así como las dificultades que tuviste para realizarla.
   En esta práctica reforzamos algunos conceptos vistos a lo largo de las clases, en particular aquellos relacionados con las propiedades de los candados seguros. También tuvimos una aproximación al uso adecuado de este tipo de recursos para llevar a cabo implementaciones más seguras a la vez que eficientes. No experimentamos tantas dificultades al tratar de entender como implementar la práctica pues aún contamos con la asistencia del ayudante, pero llegar a entender en la teoría el problema de los reos y el interruptor nos llevó un rato, pues no estamos acostumbrados a pensar de forma concurrente, no aún.

Diagrama UML



Nota: No se emplearon patrones de diseño, por lo tanto no hay justificación al respecto.

SonarLint

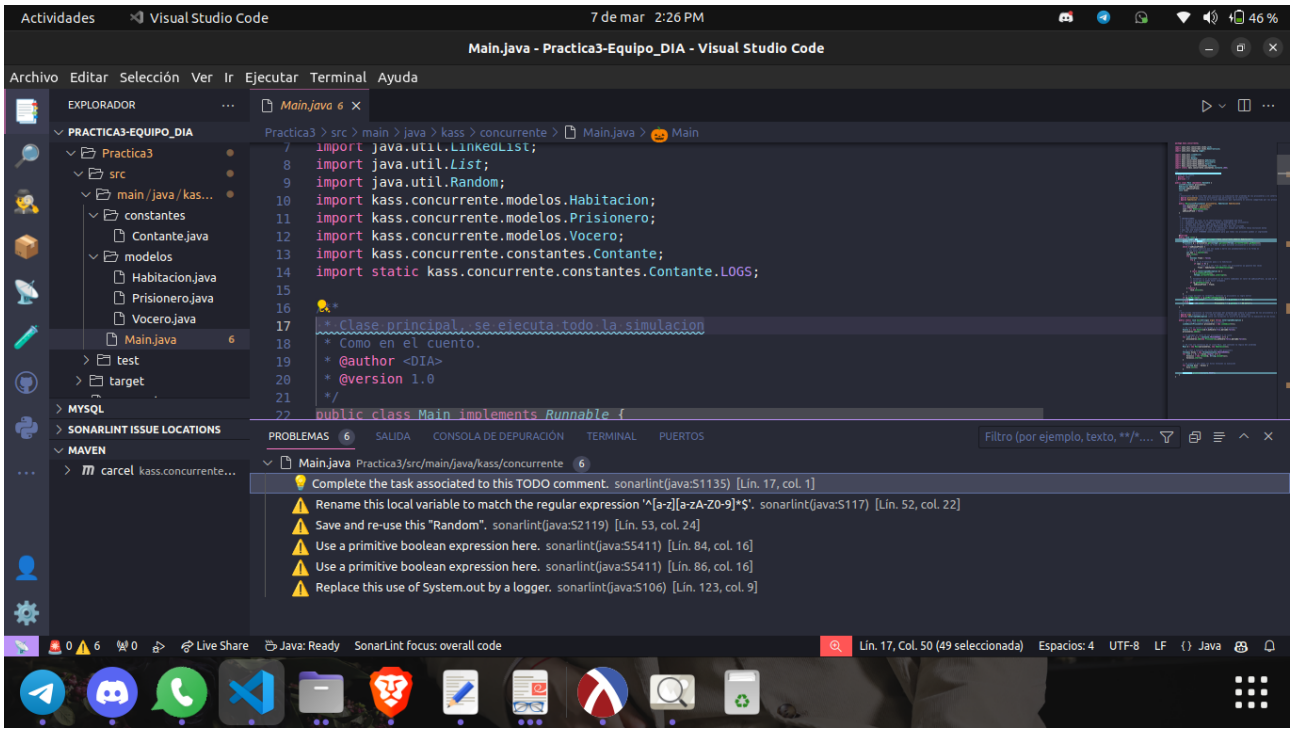


Figura 1: Warnings detectados en nuestro proyecto por SonarLint

Referencias

- Geeks for geeks. (04 de febrero de 2021). Reentrant Lock in Java. (Recuperado el 06 de marzo de 2024) `https://www.geeksforgeeks.org/reentrant-lock-java/`
- Oracle corporation. (s.f.). Class ReentrantLock. `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantLock.html`