Creación de una Simulación Interactiva del Problema de los Filósofos con Concurrencia en Java

Fabian Alonso Lopez Gálvez

Universidad de Sonora, Ingeniería en sistemas de información, Hermosillo, Sonora

1. Introducción

Aprender a coordinarse y convivir con otros mientras compartimos recursos limitados es un aspecto fundamental del comportamiento humano y social. Desde pequeños, interactuamos con objetos, personas y reglas implícitas que determinan quién puede usar qué, cuándo y de qué forma. Estos procesos de aprendizaje dónde a veces se fracasa, se compite, se espera turno o se coopera moldean nuestra capacidad para desenvolvernos en entornos donde las decisiones de un individuo afectan directamente a los demás

De forma muy similar, la informática y los sistemas concurrentes enfrentan problemas en los que múltiples entidades deben compartir recursos limitados sin generar caos, bloqueos o comportamientos conflictivos. Uno de los ejemplos clásicos para ilustrar este fenómeno es el **Problema de los Filósofos**, una metáfora desarrollada para enseñar los conceptos fundamentales de concurrencia, sincronización y acceso ordenado a recursos compartidos.

En este trabajo se desarrolla una simulación interactiva en Java del Problema de los Filósofos, donde varios filósofos alternan entre pensar y comer en una mesa circular mientras comparten palillos. Más allá de su apariencia de videojuego y visualmente oriental, esta simulación permite observar en tiempo real cómo se comporta un sistema concurrente, cómo se evita el interbloqueo, y cómo cada hilo (filósofo) coordina el acceso a objetos compartidos (palillos) mediante mecanismos de sincronización.

Primero que todo explicaremos que es un hilo, concurrencia y semáforos, ya que fueron el núcleo para desarrollar este trabajo

¿Qué es un Hilo?

Un hilo (o *thread*) es la unidad básica de ejecución dentro de un proceso. Puede entenderse como un "subproceso" que realiza una tarea específica de manera independiente, pero compartiendo el mismo espacio de memoria y recursos del proceso que lo contiene.

Mientras que un proceso es un programa en ejecución con su memoria, variables y estado propio, un hilo permite dividir ese programa en partes más pequeñas que pueden ejecutarse de manera concurrente. Esto significa que un mismo programa puede realizar varias tareas al mismo tiempo, como escuchar eventos, actualizar una interfaz gráfica y procesar datos sin bloquearse.

¿Por qué son importantes los hilos?

Los hilos permiten que las aplicaciones sean más eficientes y responsivas. Por ejemplo:

-En videojuegos, un hilo puede manejar la física, otro la IA y otro los gráficos.

-En servidores, miles de solicitudes se procesan mediante hilos independientes.

-En simulaciones académicas, como el Problema de los Filósofos, cada filósofo es un hilo que compite por recursos compartidos (los palillos).

Hilos y problemas clásicos

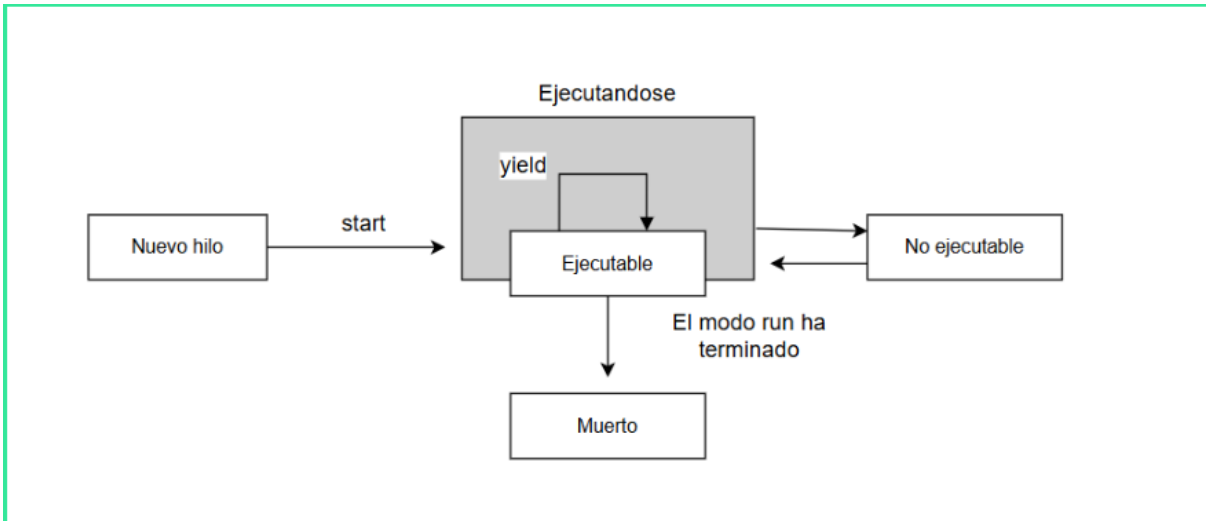
Cuando varios hilos necesitan acceder a los mismos recursos, pueden surgir:

-Condiciones de carrera

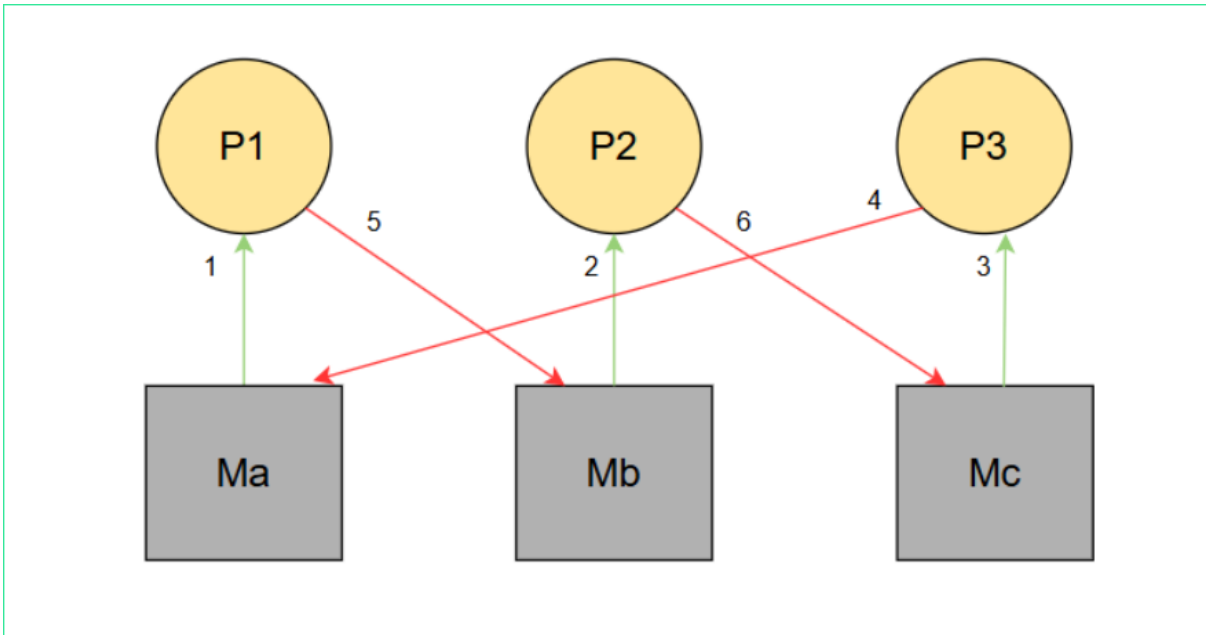
-Interbloqueos

-Inanición

-Problemas de sincronización



En la siguiente imagen se presentara un diagrama con 3 hilos y como debería ser su comportamiento para no haya un interbloqueo



¿Qué es la Concurrencia?

La concurrencia es la capacidad de un sistema para manejar múltiples tareas de manera aparentemente simultánea, compartiendo recursos limitados como CPU, memoria o dispositivos de entrada/salida. Aunque estas tareas pueden no ejecutarse exactamente al mismo tiempo (especialmente en sistemas con un solo procesador), el sistema operativo las alterna con tal rapidez que, desde la perspectiva del usuario o del programa, parece que avanzan paralelamente.

En programación, la concurrencia surge cuando dos o más hilos o procesos deben ejecutarse de forma coordinada mientras comparten datos o estructuras comunes. Esto introduce desafíos como:

-Condiciones de carrera (race conditions): cuando dos hilos acceden y modifican un dato al mismo tiempo, creando resultados impredecibles.

-Interbloqueos (deadlocks): cuando dos tareas se quedan esperando recursos que la otra posee, deteniendo el sistema indefinidamente.

-Inanición (starvation): cuando un proceso nunca recibe los recursos necesarios para continuar.

-Sección crítica: fragmento de código donde se accede a recursos compartidos que requieren protección.

Debido a estos problemas, se utilizan mecanismos como semáforos, bloqueos (locks) y monitores, que aseguran el acceso controlado a los recursos para que la ejecución sea correcta y predecible.

La concurrencia es fundamental en aplicaciones modernas, desde servidores web hasta videojuegos, sistemas de bases de datos y simulaciones. En particular, proyectos académicos como el Problema de los Filósofos sirven como modelo para estudiar los principales retos de la concurrencia y las técnicas para resolverlos.

¿Qué es un Semáforo?

Un semáforo es un mecanismo de sincronización utilizado en programación concurrente para controlar el acceso a recursos compartidos. Su función principal es evitar conflictos, como condiciones de carrera o interbloqueos, cuando varios hilos intentan usar simultáneamente un recurso que no está diseñado para ser accedido por múltiples tareas al mismo tiempo.

¿Por qué son importantes los semáforos?

Los semáforos son fundamentales porque permiten:

-Coordinar el orden en que los hilos acceden a recursos compartidos.

-Evitar accesos simultáneos peligrosos.

-Resolver problemas clásicos como productores-consumidores, lectores-escritores y filósofos.

-Garantizar la integridad de los datos, evitando inconsistencias.

Sin ellos, los hilos ejecutarían instrucciones de manera no coordinada, provocando resultados impredecibles, bloqueos o corrupción de datos.

Semáforos y el Problema de los Filósofos

En este problema clásico de sincronización, los semáforos se utilizan para:

-Controlar el acceso a los palillos.

-Evitar que todos los filósofos tomen un palillo simultáneamente, generando interbloqueo.

-Garantizar que los filósofos puedan alternar entre pensar y comer sin bloquearse indefinidamente.

Los semáforos permiten implementar políticas de sincronización que aseguren que el sistema funcione de forma segura, evitando tanto la inanición como el interbloqueo.

¿Qué es el Problema de los Filósofos y por qué estudiarlo?

El Problema de los Filósofos es una analogía para representar cómo múltiples procesos pueden competir por recursos limitados de forma ordenada y segura. Cada filósofo puede realizar dos acciones principales:

-Pensar

-Comer, para lo cual necesita tomar dos palillos que comparte con sus vecinos.

En términos de sistemas concurrentes:

-Los filósofos representan hilos o procesos.

-Los palillos representan recursos compartidos (locks).

El desafío es evitar:

-Deadlocks (todos se quedan esperando recursos para siempre)

-Starvation (alguien nunca accede al recurso)

-Condiciones de carrera (dos procesos usando el mismo recurso simultáneamente)

Así como en el aprendizaje humano evaluamos la mejor manera de actuar según el entorno, en concurrencia buscamos la mejor forma de coordinar procesos para que todos logren su objetivo sin bloquearse entre sí.

¿Qué vamos a hacer?

Como el título sugiere, vamos a simular el Problema de los Filósofos, pero no solo como un ejercicio teórico, sino como un proyecto visual e interactivo donde podamos observar cómo funciona un sistema concurrente en tiempo real. En lugar de quedarnos únicamente con modelos abstractos matemáticos, vamos a construir una aplicación gráfica donde cada filósofo sea representado por un personaje, con expresiones, animaciones y comportamientos que cambian dependiendo del estado en el que se encuentren.

La idea es trasladar un concepto clásico de concurrencia a un entorno visual accesible, comprensible e incluso entretenido, donde podamos analizar:

-Cómo múltiples hilos intentan acceder a recursos compartidos.

-Qué estrategias podemos usar para evitar errores de sincronización.

-Cómo se comportan los procesos cuando compiten por recursos limitados.

-Qué ocurre cuando un proceso “come”, espera, piensa o bloquea a otros.

-Cómo se ve un deadlock y cómo podemos evitarlo.

En esta simulación:

-Cada filósofo será un hilo independiente.

-Cada palillo será un recurso compartido, protegido por un semáforo.

Los tazones cambiarán de color según el estado:

-Blanco → Antes de comer

-Verde → Filósofo comiendo

-Negro → Filósofo terminó de comer

El dibujo completo incluirá:

-Filósofos animados

-Sombreros, bigotes, ojos, boca, rostro, plato y palillos

-Un fondo oriental

-Una interfaz para elegir entre **5 y 15 filósofos**

Además, los palillos se colocan de forma dinámica dependiendo del número de filósofos, y se ajustan automáticamente en posición para no chocar con los sombreros. La interfaz también permite al usuario iniciar la simulación con cuántos filósofos desee.

2. Primeros pasos

Es necesario entender cuáles son los elementos fundamentales del proyecto y cómo iniciaremos la arquitectura del sistema. En esta etapa vamos a establecer las bases técnicas necesarias para poder representar correctamente el Problema de los Filósofos y adaptarlo a un entorno visual interactivo.

Preparación del entorno

Para este proyecto utilizamos:

-Java 8+

-Swing para la interfaz gráfica

-Threads para representar filósofos concurrentes

-Semáforos y monitores para el control de recursos

-Imágenes y gráficos 2D para los elementos visuales

Estructura Inicial del Proyecto

Creamos los siguientes componentes base:

Clase—Palillos(contructor)
Representa cada recurso compartido. Contiene un semáforo o estado de disponibilidad.

Clase—Filosofos
Cada filósofo es un hilo independiente con tres comportamientos:

-Pensar

-Tomar palillos

-Comer
Además, interactúa directamente con el panel para actualizar su estado visual.

Clase—PanelFilosofos

-Encargada del dibujo completo:

-Filósofos

-Bigotes, sombreros, ojos, boca

-Palillos

-Tazones

-Fondo oriental

Clase—FilosofosGUI
Ventana principal.
Incluye:

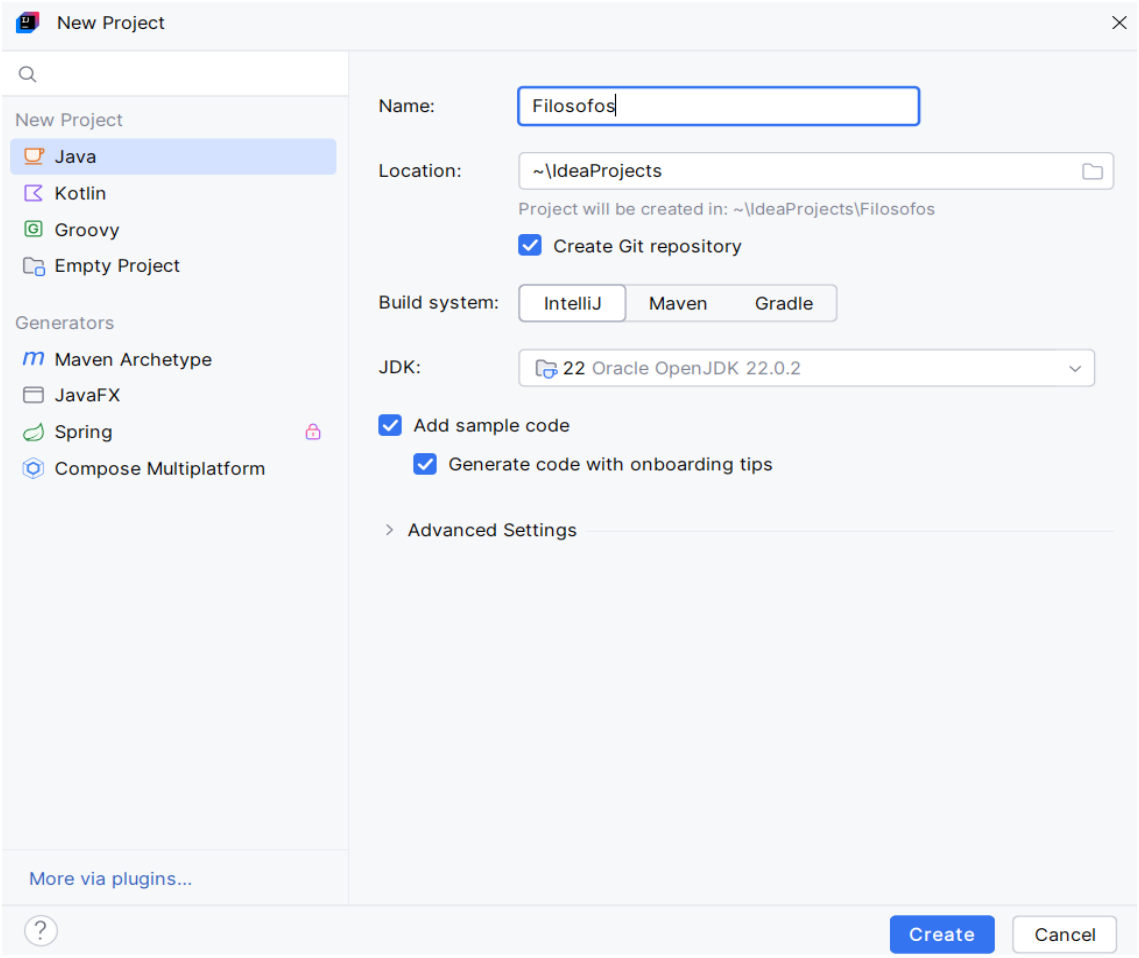
-Entrada para elegir cantidad de filósofos (5 a 15)

-Botón para iniciar la simulación

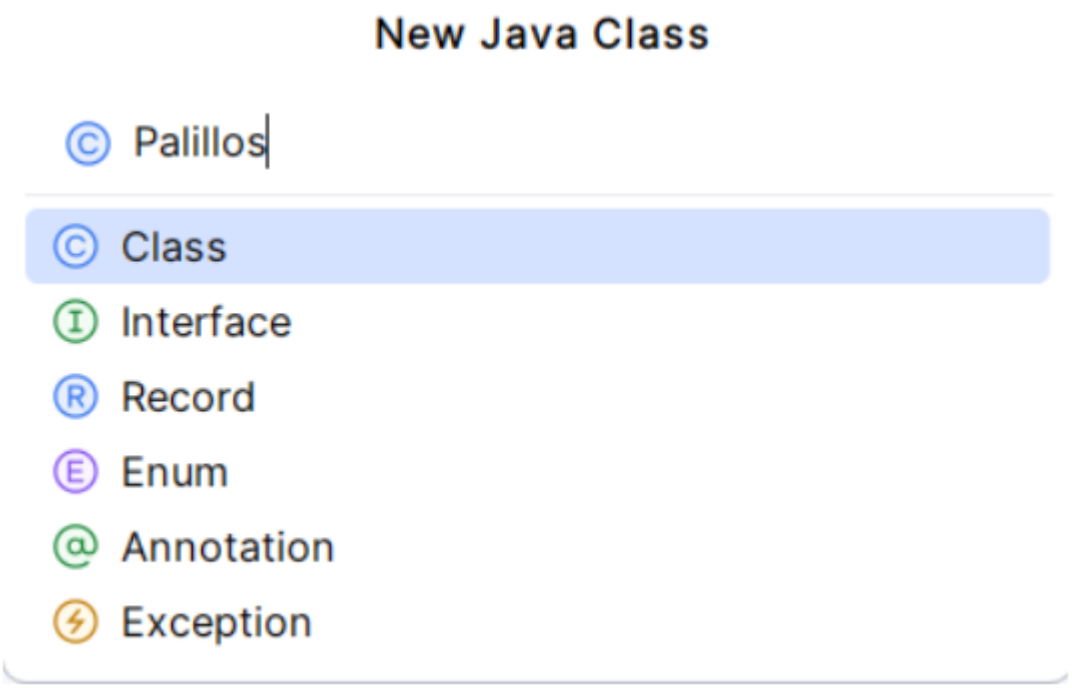
-Panel donde se dibuja todo

explicado el propósito de cada clase se mostrara como se creara el proyecto

-es en java en este ejemplo Intellij (como se muestra en la imagen)



-se creara un paquete con cualquier nombre deseable y adentro de ese paquete crear las clases (java)



3. Explicando las clases una por una

3.1 Clase Palillos

La clase Palillos representa un recurso compartido en el Problema de los Filósofos: un palillo

```
public class Palillos {
    public final int id;
    public volatile boolean enUso;

    public Palillos(int id) {
        this.id = id;
        this.enUso = false;
    }

    public void usar(String lado, int filosofoId) {
        enUso = true;
        System.out.println("Filósofo " + filosofoId + " tomó el
palillo " + lado + " (" + id + ") ");
    }
```

```
}
```

```
public final int id;
```

-Identificador único del palillo

-Es final quiere decir que nunca cambia después de crearse

```
public volatile boolean enUso;
```

Volatile significa que:

-Si un hilo modifica el valor

-Los otros hilos lo leerán INMEDIATAMENTE

-Evita que una lectura quede cacheada en un hilo

Sirve para que todos los filósofos sepan si el palillo está en uso.

Contructor

```
public Palillos(int id) {
        this.id = id;
        this.enUso = false;
    }
```

-Guarda el id

-Indica que el palillo está libre

Método (usar)

```
public void usar(String lado, int filosofoId) {
        //Pone el palillo como ocupado
        enUso = true;
        //Imprime que filosofo lo esta utilizando
        System.out.println("Filósofo " + filosofoId + " tomó el
palillo " + lado + " (" + id + ") ");
    }

}
```

3.2 Clase Filosofos

La clase filosofos representa a un filósofo en el clásico problema de los filósofos comensales, modelado como un hilo (thread).
Cada filósofo piensa, intenta comer y luego termina su actividad, usando semáforos y palillos compartidos para evitar conflictos.

```
package mx.unison.trabajadores;

//Dependencias
import javax.swing.*;
import java.util.concurrent.Semaphore;
import java.util.concurrent.ThreadLocalRandom;

public class Filosofos extends Thread{
private final Semaphore semaforo; // Controla cuántos filósofos
pueden intentan comer
    //Cada filósofo tiene referencias a los dos palillos que
necesita para comer.
    private final Palillos palilloIzq;
    private final Palillos palilloDer;
    //Identificador único del filósofo.
    private final int id;
    //Panel gráfico donde el filósofo se dibuja.
    private final PanelFilosofos panel;
    //Contador de cuántas veces comió. Se usa más adelante cuando
agregamos platos blancos → verdes → negros.
    public int comidas = 0;
   //Indica el estado actual del filósofo:
   //False : pensando
    //True : comiendo
    public volatile boolean comiendo = false;


    //Contrusctor donde recibe lo necesario
    public Filosofos(int id, Semaphore semaforo, Palillos
palilloIzq, Palillos palilloDer, PanelFilosofos panel) {
        this.id = id;
        this.semaforo = semaforo;
        this.palilloIzq = palilloIzq;
        this.palilloDer = palilloDer;
```

```
        this.panel = panel;

    }

    //Actualiza la interfaz gráfica en el hilo de Swing. Esto es
importante porque Swing NO es thread-safe; por eso se manda el
repintado al hilo correcto.
    private void actualizarPanel() {
        try {
            SwingUtilities.invokeAndWait(panel::repaint);
        } catch (Exception ignored) {}
    }


    //Metodo pensar que se ejecuta al inicio del programa
    private void pensar() {
        //Cambia al estado NO comiendo
        comiendo = false;
        //Actualiza los futuros dibujos
       actualizarPanel();
        //Muestra el mensaje en la consola y tarda entre 3 a 4
segundos para cada accion
        System.out.println("Filósofo " + id + " está pensando ");
        try {
            Thread.sleep(ThreadLocalRandom.current().nextInt(3000,
4000));
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }
    }


    //Este metodo es la principal funcion del programa ya que es el
que controla los palillos y la accion de comer
    private void comer() {
        try {
            //Espera el semaforo
            semaforo.acquire();

            // Límite global, evita bloqueo total

            //Tomar palillos
```

```
            //Marca los palillos como enUso=true
            palilloIzq.usar("izquierdo", id);
            palilloDer.usar("derecho", id);
            actualizarPanel();
            Thread.sleep(4000); // Pequeña pausa para simular acción
            comiendo = true;
            actualizarPanel();
            //Simula el tiempo comiendo y que filosofo esta comiendo
            System.out.println("Filósofo " + id + " está comiendo
");
            Thread.sleep(ThreadLocalRandom.current().nextInt(3000,
6000));

            //Marca los palillos como sueltos igual con comiendo
            palilloIzq.enUso = false;
            palilloDer.enUso = false;
            comiendo = false;
            //Se incrementa las comidas para un futuro (para que el
tazon se ponga blanco)
            comidas++;
            actualizarPanel();
            System.out.println("Filósofo " + id + " terminó de comer
");

        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        //Libera el semaforo
        } finally {
            semaforo.release();
        }
    }

    //Aqui ejecuta todos los metodos y en que orden, primero piensa
y despues come
    //Al finalizar estas 2 acciones se imprime que el filosofo
termino
    @Override
    public void run() {
            pensar();
            comer();
            System.out.println("Filósofo " + id + " ha terminado su
jornada ");
```

```
    }

    }
```

3.3 Clase PanelFilosofos

Es la clase encargada de dibujar toda la parte gráfica de la simulación de los filósofos. Calcula las posiciones en círculo, dibuja la mesa, el fondo oriental, los filósofos, los palillos y los tazones. También cambia los colores y animaciones (como vapor) dependiendo si un filósofo está pensando, comiendo o ya comió. Cada actualización visual se refleja cuando los filósofos llaman a repaint().

```
package mx.unison.trabajadores;

//Dependencias
import javax.swing.*;
import java.awt.*;
import java.awt.geom.AffineTransform;
import java.awt.geom.Path2D;
import java.awt.Image;
import java.util.Objects;
import javax.swing.ImageIcon;

//Esta clase extiende JPanel, lo que significa que es un panel
gráfico de Swing, donde se dibujan los filósofos, sus palillos, los
tazones y el fondo.
public class PanelFilosofos extends JPanel {
    //Atributos principales
private final Filosofos[] filosofos;//Arreglo que representa cada
filosofo
    private final Palillos[] palillos;//Arreglo que representa cada
palillo
    private final int RADIO_FILOSOFO = 30; //Tamaño de la cabeza del
filosofo
    private final int numFilosofos; //Total de filosofos en el
```

```
sistema
    private final Point[] posiciones; //Coordenadas donde se dibuja
cada filosofo
    private Image fondo; //El fondo del panel

    //Constructor donde guarda los arreglos de cada dato
    public PanelFilosofos(Filosofos[] filosofos, Palillos[]
palillos, int numFilosofos) {
        this.filosofos = filosofos;
        this.palillos = palillos;
        this.numFilosofos = numFilosofos;
        //Arreglo de posiciones donde luego se calculará dónde
dibujar a cada filósofo.
        this.posiciones = new Point[numFilosofos + 1];
        //Fondo gris por si no carga la imagen
       setBackground(Color.gray);
        //La imagen del fondo en este caso se llama imagen_oriental,
donde tiene que meterse en el resorces del proyecto de java
        this.fondo = new
ImageIcon(Objects.requireNonNull(getClass().getResource("/imagen_ori
ental.jpg"))).getImage();
        //Calcula la ubicación de los filósofos formando un círculo.
        calcularPosiciones();
    }

    private void calcularPosiciones() {
        //Coordenadas del centro en base a los filosofos
        int centroX = 780;
        int centroY = 430;

        //Radio dinámico según cantidad
        //Ajuste del radio según cuántos filósofos haya
        //Entre más filósofos, más grande el círculo para que se
separen correctamente.
        int RADIO = switch (numFilosofos) {
            case 5, 6 -> 260;
            case 7, 8 -> 280;
            case 9, 10 -> 300;
            case 11, 12 -> 330;
            case 13, 14 -> 360;
            case 15 -> 380;
            default -> 260;
```

```
        };

        //Calculo de posiciones
       //Divide un círculo en partes iguales dependiendo a la
cantidad de los numeros de filosofos
        double paso = 360.0 / numFilosofos;
        for (int i = 1; i <= numFilosofos; i++) {
            //Asigna a cada filósofo un ángulo
            double angulo = Math.toRadians(90 + (i - 1) * paso);
            //Calcula su posición en el círculo con cosenos y senos
            int x = (int) (centroX + RADIO * Math.cos(angulo));
            int y = (int) (centroY - RADIO * Math.sin(angulo));
            //Guarda esas coordenadas en el arreglo posiciones
            posiciones[i] = new Point(x, y);
        }

    }

    @Override
//Swing llama a este método cada vez que hay que redibujar la
pantalla.
/*
Dentro se dibuja todo
-Fondo
-Palillos
-Filósofos
-Tazones
-Bigotes, ojos, sombreros
-Animación de vapor cuando comen
*/
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        //Convierte Graphics a Graphics2D para tener mas control
para que no se vea pixelado
        Graphics2D g2 = (Graphics2D) g;
        g2.setStroke(new BasicStroke(3));
        g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);

        //Dibuja la imagen cargada como fondo del panel.
        if (fondo != null) {
            g2.drawImage(fondo, 0, 0, getWidth(), getHeight(),
```

```
this);
        }


        //Dibujar palillos entre filósofos
        for (int i = 1; i <= numFilosofos; i++) {
            Point p1 = posiciones[i];
            Point p2 = posiciones[(i == numFilosofos) ? 1 : i + 1];
            Palillos palillo = palillos[i];

            // Calcular punto medio entre filósofos
            int midX = (p1.x + p2.x) / 2;
            int midY = (p1.y + p2.y) / 2;

            // Ángulo del palillo (entre los dos filósofos)
            double angulo = Math.atan2(p2.y - p1.y, p2.x - p1.x);

            // Dibujar palillo como rectángulo inclinado
            dibujarPalillo(g2, midX, midY, angulo, palillo.enUso);
        }

        //Dibujar Filosofos
        for (int i = 1; i <= numFilosofos; i++) {
            Point p = posiciones[i];
            Filosofos f = filosofos[i];
            boolean comiendo = f.comiendo;


            // Cabeza del filósofo que cambia a color cremita a
color verde cuando come
            g2.setColor(comiendo ? new Color(255, 230, 180) : new
Color(255, 200, 150));
            g2.fillOval(p.x - RADIO_FILOSOFO, p.y - RADIO_FILOSOFO,
RADIO_FILOSOFO * 2, RADIO_FILOSOFO * 2);
           //Se pinta el contorno en negro
            g2.setColor(Color.BLACK);
            g2.drawOval(p.x - RADIO_FILOSOFO, p.y - RADIO_FILOSOFO,
RADIO_FILOSOFO * 2, RADIO_FILOSOFO * 2);

            // Sombrero (tipo chino) que es un triangulo sobre la
cabeza de un color amarillo un poco dorado
            int[] xSombrero = {p.x - 35, p.x, p.x + 35};
```

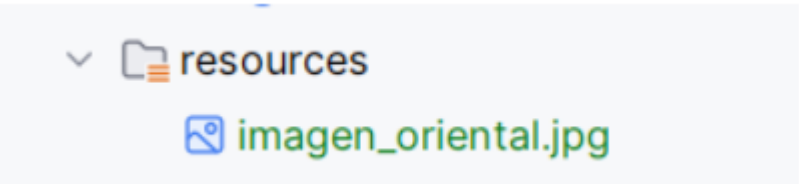
```
            int[] ySombrero = {p.y - 25, p.y - 48, p.y - 25};
            g2.setColor(new Color(180, 140, 30));
            g2.fillPolygon(xSombrero, ySombrero, 3);
            g2.setColor(Color.BLACK);
            g2.drawPolygon(xSombrero, ySombrero, 3);

            // Ojos rasgados sobre los filosofos
            g2.setStroke(new BasicStroke(2));
            g2.drawLine(p.x - 10, p.y - 5, p.x - 2, p.y - 5);
            g2.drawLine(p.x + 2, p.y - 5, p.x + 10, p.y - 5);


            //Bigotes con curvas con Path2D
            g2.setColor(Color.BLACK);
            g2.setStroke(new BasicStroke(2f, BasicStroke.CAP_ROUND,
BasicStroke.JOIN_ROUND));

            //Bigote izquierdo
            Path2D bigoteIzq = new Path2D.Double();
            bigoteIzq.moveTo(p.x - 6, p.y + 4); // inicio más arriba
(casi a la altura de los ojos)
            bigoteIzq.curveTo(
                    p.x - 18, p.y + 5,    //Primer control (curva
hacia afuera)
                    p.x - 1, p.y + 50,    //Segundo control (curva
hacia abajo)
                    p.x - 17, p.y + 35   //Punto final (abajo y
ligeramente afuera)
            );
            g2.draw(bigoteIzq);

            // Bigote derecho que es lo mismo con el izquierdo pero
al lado opuesto
            Path2D bigoteDer = new Path2D.Double();
            bigoteDer.moveTo(p.x + 6, p.y + 4);
            bigoteDer.curveTo(
                    p.x + 18, p.y + 5,
                    p.x + 1, p.y + 50,
                    p.x + 17, p.y + 35
            );
            g2.draw(bigoteDer);
```

```
            //Boca más pequeña y más abajo
            g2.drawArc(p.x - 6, p.y + 8, 12, 6, 0, -180);



            //Tazon asiatico enfrente de los filosofos
            double anguloPlato = Math.toRadians(90 + (i - 1) *
(360.0 / numFilosofos));
            int px = (int) (p.x - 55 * Math.cos(anguloPlato));
            int py = (int) (p.y + 90 * Math.sin(anguloPlato));



            //Color del tazón según estado
            Color colorTazon = comiendo ? Color.GREEN.darker() :
Color.white;



            //Borde superior (boca del tazón)
            g2.setColor(Color.white);
            g2.fillOval(px - 18, py + 1, 36, 12);
            g2.setColor(colorTazon);
            g2.fillOval(px - 18, py - 8, 36, 12); //Borde superior

            g2.setColor(Color.white);
            g2.drawOval(px - 18, py - 8, 36, 12); //Contorno

            //Interior del tazón (óvalo más pequeño)
            if (f.comiendo) {
                g2.setColor(new Color(0, 150, 0)); //Verde mientras
come
            } else if (f.comidas > 0) {
                g2.setColor(Color.white); //Negro cuando termina
            } else {
                g2.setColor(Color.black); //Blanco antes de comer
            }
            g2.fillOval(px - 14, py - 4, 28, 8);
            g2.setColor(Color.black);
            g2.drawOval(px - 14, py - 4, 28, 8);
```

```
            //Vapor (si está comiendo)
            if (comiendo) {
                //Tres humitos cuando comen
                g2.setColor(Color.BLACK);
                for (int s = 0; s < 3; s++) {
                    int sx = px - 8 + s * 8;
                    g2.drawArc(sx, py - 22, 5, 15, 25, 165);
                }
            }

            //Nombre o etiqueta del filósofo bajo a cada filosofo
            //Etiqueta F#
            g2.setColor(Color.BLACK);
            g2.drawString("F" + i, p.x - 6, p.y + RADIO_FILOSOFO +
15);
        }

    }

    //Dibuja un palillo entre filósofos
   private void dibujarPalillo(Graphics2D g2, int x, int y, double
angulo, boolean enUso) {
        int largo = 60;
        int ancho = 6;
        //Dibuja los palillos rotándolos según dirección entre
filósofos.
        AffineTransform at = g2.getTransform();
        g2.translate(x, y);
        g2.rotate(angulo);
        //Cuando se usa el palillo cambia a color verde oscuro para
que se note
        g2.setColor(enUso ? Color.GREEN.darker() : new Color(139,
69, 19)); // verde si en uso, café si no
        g2.fillRect(-largo / 2, -ancho / 2, largo, ancho);
        g2.setColor(Color.BLACK);
        g2.drawRect(-largo / 2, -ancho / 2, largo, ancho);

        g2.setTransform(at);
    }
}
```

Aqui se establece la imagen, si lo tienes en otro lugar solo pones la ruta de la imagen aqui

```
this.fondo = new
ImageIcon(Objects.requireNonNull(getClass().getResource("/imagen_ori
ental.jpg"))).getImage();
```



3.4 Clase FilosofosGUI

Esta es la clase principal de tu aplicación. Es la encargada de:

-Crear la ventana.

-Mostrar los controles.

-Crear filósofos, palillos y panel visual.

-Iniciar los hilos.

-Permitir elegir cuántos filósofos mostrar.

+Cada vez que el usuario inicia la simulación, esta clase recrea todo el entorno y comienza nuevamente con los hilos trabajando en paralelo.

```
package mx.unison.trabajadores;

//Dependencias
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.util.concurrent.Semaphore;

//Extiende JFrame
public class FilosofosGUI extends JFrame {

    //Atributos principales
    //Se dibuja la simulacion
    private PanelFilosofos panel;
    //Campo para ingresar el numero de filosofos
    private JTextField inputFilosofos;
```

```
    //Boton que reinicia la simulacion dependiendo del numero que
ponemos
    private JButton btnIniciar;
    private Filosofos[] filosofos;
    private Palillos[] palillos;
    //Controla concurrencia
    private Semaphore semaforo;


    //Constructor configuracion general de la ventana
    public FilosofosGUI() {
        //Titulo del interfaz
        setTitle("Problema de los Filósofos Orientales ");
        //Tamaño de la interfaz
        setSize(900, 700);
        //El cierre
       setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        //Layout principal
        setLocationRelativeTo(null);
        setLayout(new BorderLayout());

        //Panel de control arriba que en este caso es de 5 a 15
(puede ser a cualquier numero de filosofos)
        JPanel panelControl = new JPanel();
        panelControl.add(new JLabel("Número de Filósofos (5 -
15):"));
        inputFilosofos = new JTextField("5", 5);
        btnIniciar = new JButton("Iniciar");
        //Un campo para ingresar numero de filosofos
        panelControl.add(inputFilosofos);
        //Un boton para iniciar
        panelControl.add(btnIniciar);
        add(panelControl, BorderLayout.NORTH);


        //Panel inicial (por defecto con 5 filósofos, esto se puede
poner a como tu quieras)
        iniciarSimulacion(5);

        //Acción del botón
        btnIniciar.addActionListener((ActionEvent e) -> {
            try {
```

```
                //Aqui es de 5 a 15, facilmente puede ser de otros
                rangos
                //Pero ten en cuenta que debes de cambair las
     dimensiones del panel, porque no pueden caber todos los
filosofos
                int n =
Integer.parseInt(inputFilosofos.getText().trim());
                if (n < 5 || n > 15) {
                    //Mensaje de error por si no pones un numero
dentro del rango
                    JOptionPane.showMessageDialog(this, "El número
debe ser entre 5 y 15");
                    return;
                }
                iniciarSimulacion(n);
            } catch (NumberFormatException ex) {
                JOptionPane.showMessageDialog(this, "Introduce un
número válido");
            }
        });
        //Se muestra el panel
        setVisible(true);

    }


    private void iniciarSimulacion(int n) {
        //Limpiar el panel anterior cuando se actualiza
        if (panel != null) remove(panel);
        //Crear los nuevos palillos y es un palillo para cada
filosofo
        semaforo = new Semaphore(n);
        palillos = new Palillos[n + 1];
        for (int i = 1; i <= n; i++) {
            palillos[i] = new Palillos(i);
        }

        filosofos = new Filosofos[n + 1];
        //Se crea el panel visual dependiendo los filosofos y los
palillos
        panel = new PanelFilosofos(filosofos, palillos, n);
        add(panel, BorderLayout.CENTER);
```

```
        //Se crean los filosofos con las propiedades necesarias
       for (int i = 1; i <= n; i++) {
            Palillos izq = palillos[i];
            Palillos der = (i == n) ? palillos[1] : palillos[i + 1];
            filosofos[i] = new Filosofos(i, semaforo, izq, der,
panel);
        }
        //Actualiza la interfaz
        revalidate();
        repaint();

        //Iniciar hilos
        SwingUtilities.invokeLater(() -> {
            for (int i = 1; i <= n; i++) {
                filosofos[i].start();
            }
        });
    }


    //El metodo main para que el sistema funcione
    public static void main(String[] args) {
        SwingUtilities.invokeLater(FilosofosGUI::new);
    }

    }
```

4.Desafios y soluciones en la implementacion

La simulación del problema de los Filósofos implica manejar varios retos relacionados con la concurrencia, el uso compartido de recursos y la actualización segura de la interfaz gráfica. A continuación se describen los principales desafíos encontrados y las soluciones implementadas.

1-Desafio: Los palillos (objetos compartidos) pueden ser tomados por varios filósofos de manera simultánea si no se controla adecuadamente el acceso.

Esto puede causar comportamientos inconsistentes, como dos filósofos usando el mismo palillo al mismo tiempo.

1-Solucion: Cada palillo tiene un indicador enUso y, más importante, se controla su acceso a través del semáforo global, que limita cuántos filósofos pueden intentar comer simultáneamente.
Esto reduce enormemente la probabilidad de colisiones en recursos compartidos.

2-Desafio: En la versión clásica del problema, todos los filósofos podrían tomar su palillo izquierdo al mismo tiempo y quedarse esperando el palillo derecho, deteniendo completamente el sistema.

2-Solucion: Se usa un semáforo global con capacidad = número de filósofos, que controla el número de hilos que pueden estar en la zona crítica (comer).
Esto garantiza que siempre haya un filósofo que pueda tomar ambos palillos y libera el sistema de un posible deadlock.

3-Desafios: La animación (representar tazones llenos o vacíos, palillos en uso, etc.) debe mantenerse sincronizada con las acciones reales de los filósofos, sin retrasos o saltos visuales.

3-Solucion: La variable comiendo se marca como volatile, lo que asegura que todos los hilos (incluido el que pinta la interfaz) vean inmediatamente el nuevo estado.
Además, cada estado relevante invoca un repaint inmediato para mantener la animación fluida.

5.Ejecucion del programa y resultado

ejecución con 5 filósofos sin consola

5 filosofos consola

10 filosofos sin consola

6.Conclusiones

La implementación del problema de los Filósofos permitió comprender de manera práctica y visual los retos fundamentales de la concurrencia, la sincronización de recursos y la interacción segura entre hilos y GUI en Java. A lo largo del desarrollo se presentaron diversos desafíos técnicos que condujeron a aprendizajes significativos.

En primer lugar, se evidenció la importancia de los mecanismos de control de concurrencia, como los semáforos, para evitar condiciones de carrera y asegurar un uso ordenado de los recursos compartidos (los palillos). Asimismo, se comprobó que un diseño adecuado puede prevenir problemas clásicos como el deadlock y la inanición, demostrando la relevancia de la sincronización global mediante un semáforo que regula cuántos filósofos pueden intentar comer simultáneamente.

Otro aprendizaje clave fue comprender que el manejo de interfaces gráficas en entornos concurrentes requiere una atención especial. Swing no es thread-safe, por lo que fue necesario utilizar métodos como SwingUtilities.invokeAndWait para garantizar que todas las actualizaciones visuales se realizaran desde el hilo apropiado, manteniendo estabilidad y consistencia visual.

Finalmente, este proyecto demostró cómo la abstracción de un problema clásico puede aplicarse para reforzar conceptos fundamentales en programación concurrente, diseño de sistemas interactivos y visualización de procesos. La mezcla de lógica, sincronización y diseño gráfico ofreció una experiencia formativa completa que integra teoría y práctica de manera significativa.

7. Referencias Bibliográficas

Repositorio UAPA-UNAM:
UAPA-UNAM. (s. f.). *Hilos*. Recuperado de https://repositorio-uapa.cuaed.unam.mx/repositorio/moodle/pluginfile.php/3084/mod_resource/content/1/UAPA-Hilos/index.html
Laurel, DATS-FI UPM:
UPM, Grupo de Sistemas Operativos. (s. f.). *Capítulo: Interbloqueos*. Recuperado de https://laurel.datsi.fi.upm.es/_media/docencia/asignaturas/soa/soa-capitulo-interbloqueos.pdf
Paradigmas de Programación (Teoría de concurrencia):
Ferestrepoca. (s. f.). *Programación concurrente: Teoría*. Recuperado de https://ferestrepoca.github.io/paradigmas-de-programacion/progconcurrente/concurrente_teoria/index.html
GeeksforGeeks—Dining Philosophers con semáforos:
GeeksforGeeks. (s. f.). *Dining Philosopher Problem Using Semaphores*. Recuperado de https://www.geeksforgeeks.org/operating-systems/dining-philosopher-problem-using-semaphores
GeeksforGeeks—Swing / JPanel:
GeeksforGeeks. (s. f.). *Java Swing—JPanel With Examples*. Recuperado de https://www.geeksforgeeks.org/java-swing-jpanel-with-examples/

GeeksforGeeks—Introducción a Swing:
GeeksforGeeks. (s. f.). *Introduction to Java Swing*. Recuperado de https://www.geeksforgeeks.org/java/introduction-to-java-swing/