Proyecto Programación Concurrente y Paralela (Manual de Uso)

1. Introducción

El proyecto **ProyectoPCyP** es un laboratorio gráfico para explorar problemas clásicos de concurrencia. Cada problema cuenta con:

- Una vista animada en Swing que muestra el estado de los hilos involucrados.
- Una familia de estrategias de sincronización que implementan mutexe, semáforos, variables de condición, monitores y barreras.
- Una capa común encargada de dibujar el grafo de asignación de recursos, recopilar métricas y coordinar el panel activo.

2. Arquitectura general

Carpeta	Contenido
`core/`	Paneles compartidos (`DrawingPanel`), clases base (`ProyectoPCyP`, `SimPanel`) y utilidades para el grafo/estadísticas.
`problemas/`	Panel Swing por problema (productor-consumidor, filósofos, barbero, fumadores, lectores-escritores, asistentes virtuales). Cada panel crea entidades visuales y expone `startWith`, `stopSimulation` y `setDrawingPanel`.
`synch/`	Estrategias concretas por problema. Cada estrategia recibe el panel correspondiente y notifica avances mediante callbacks.

3. Uso general de la aplicación

1. Abrir el proyecto en NetBeans.
2. Seleccionar un problema desde la lista lateral.
3. En el panel derecho elegir el método de sincronización (Mutex, Semáforos, Variable de condición, Monitores o Barreras). Cada cambio detiene la simulación actual y crea una nueva estrategia.
4. Observar la animación y el grafo en vivo. Los contadores inferiores muestran entidades activas/en espera.
5. Si el problema soporta métricas, activar la pestaña de gráficas para comparar métodos.

4. Resumen de problemas disponibles

- **Tanque de Agua**: simula un depósito con bombas de entrada/salida. Uso educativo del productor-consumidor.
- **Cena de los Filosofos**: representa los estados Pensando/Hambriento/Comiendo con diferentes protocolos de sincronización.
- **Barbero Dormilón**: visualiza sillas de espera, la silla del barbero y la cola de clientes.
- **Fumadores**: muestra al agente y los tres fumadores alrededor de la mesa, incluyendo animación de humo.

- **Lectores/Escritores**: limita simultáneamente actores lectores/escritores y refleja prioridades en el grafo.
- **Asistentes Virtuales**: modelo híbrido productor-consumidor con prioridades; se detalla en la siguiente sección.

5. Problema "Asistentes Virtuales"

5.1 Objetivo

Representa un equipo de **6 asistentes virtuales** que compiten por **tokens de prioridad** y **slots del servidor** para responder solicitudes. Hay dos niveles de prioridad (alta/baja) y cinco métodos de sincronización intercambiables.

5.2 Interfaz y controles

- El panel (`VirtualAssistantsSim`) dibuja colas separadas por prioridad, la bandeja de tokens, el servidor y los agentes.
- Al pulsar un método, `startWith` instancia la estrategia apropiada y reinicia los agentes. Si el método no aplica, se muestra un `JOptionPane`.
- La sección "Tokens de prioridad" se anima con flechas (`tokenPulses`) cuando un token es asignado.
- Desde el `DrawingPanel` se puede habilitar una gráfica temporal (línea o barras) para comparar rendimiento entre métodos.

5.3 Flujo interno de un asistente

Cada agente corre en su propio hilo (`runAssistant`). Su ciclo completo es:

1. Espera aleatoria en estado `IDLE`.
2. Se coloca en la cola correspondiente (`WAITING_TOKEN`).
3. Solicita un token al método `acquirePriorityToken` de la estrategia.
4. Con token en mano (`HAS_TOKEN`), pide un slot (`acquireServerSlot`).
5. Procesa la solicitud (`PROCESSING`), responde al usuario (`RESPONDING`) y luego descansa (`RESTING`).
6. Libera token y slot mediante `releaseResources` e inicia un nuevo ciclo.

Código relevante (`src/problemas/VirtualAssistantsSim.java`):

```
public void startWith(SyncMethod method) {
    if (!supports(method)) {
        JOptionPane.showMessageDialog(this,
            "Este método de sincronización no aplica a los Asistentes
Virtuales.",
            "Método no disponible", JOptionPane.WARNING_MESSAGE);
        return;
    }
    stopSimulation();
    methodTitle = describeMethod(method);
    currentStrategy = instantiateStrategy(method);
    if (currentStrategy == null) {
        JOptionPane.showMessageDialog(this,
            "La estrategia " + method + " aún no está implementada.",
```

```
                "Sin implementar", JOptionPane.ERROR_MESSAGE);
            return;
        }
        resetAgentsToIdle();
        currentStrategy.start();
        running.set(true);
        startAgents();
        animationTimer.start();
    }
```

```
    private void runAssistant(AssistantAgent agent) {
        Random local = new Random(agent.id * 31L + System.nanoTime());
        while (running.get() && currentStrategy != null) {
            transition(agent, AssistantState.WAITING_TOKEN);
            notifyGraphQueued(agent);
            int tokenIndex = currentStrategy.acquirePriorityToken(agent);
            agent.assignedToken = tokenIndex;

            transition(agent, AssistantState.WAITING_SLOT);
            notifyGraphRequestingSlot(agent);
            int slotIndex = currentStrategy.acquireServerSlot(agent);
            agent.assignedSlot = slotIndex;

            transition(agent, AssistantState.PROCESSING);
            notifyGraphProcessing(agent);
            Thread.sleep(450 + local.nextInt(agent.isHighPriority() ? 450 :
    700));

            transition(agent, AssistantState.RESPONDING);
            Thread.sleep(260 + local.nextInt(240));

            currentStrategy.releaseResources(agent, tokenIndex, slotIndex);
            notifyGraphFinished(agent);
            transition(agent, AssistantState.RESTING);
        }
        transition(agent, AssistantState.IDLE);
    }
```

Estos métodos actualizan el dibujo y el grafo en cada transición para mantener sincronizados animación, contadores y panel de recursos.

5.4 Estrategias de sincronización

Todas las implementaciones comparten la interfaz `VirtualAssistantsStrategy`:

```
    public interface VirtualAssistantsStrategy extends SynchronizationStrategy
    {
        int acquirePriorityToken(AssistantAgent agent) throws
    InterruptedException;
```

```
    int acquireServerSlot(AssistantAgent agent) throws
InterruptedException;
    void releaseResources(AssistantAgent agent, int tokenIndex, int
slotIndex);
}
```

Cada variante (mutex, semáforos, condiciones, monitores o barreras) sigue el mismo contrato:

- `acquirePriorityToken` bloquea hasta obtener uno de los `PRIORITY_TOKENS`.
- `acquireServerSlot` evita starvation equilibrando `SERVER_SLOTS` según prioridad.
- `releaseResources` libera ambos recursos y despierta a los hilos en espera.

5.5 Gráficas y métricas

Cuando el usuario activa una gráfica en `DrawingPanel`, el `ChartSimulationPool` lanza simulaciones "fantasma" que producen muestras periódicas sin afectar la animación principal. Estas simulaciones reutilizan las mismas estrategias para contar cuántas respuestas completan por ventana de tiempo y alimentar la gráfica comparativa.

6. Buenas prácticas de ejecución

- Siempre detenga una simulación antes de iniciar otra.
- NetBeans/Ant es el entorno previsto.
- Debido a que todas las entidades son hilos en vivo, cierre la ventana principal o llame a `stopSimulation` para liberar recursos antes de cerrar la JVM.