UNIVERSIDAD DE ALCALÁ
Grado en Ingeniería en Sistemas de Información

Paradigmas de Programación
Curso 2020-2021 - Convocatoria Ordinaria

MEMORIA PECL
(SIMULACIÓN DEL FUNCIONAMIENTO DE UN RESTAURANTE DE COMIDA RÁPIDA)

- MURG, ADINA

- ORDENES ORBEGOZO, VICTORIA LORENA

ÍNDICE DE CONTENIDO:

1. Análisis de alto nivel

Descripción detallada del problema

Analizando en detalle el enunciado, nuestro programa deberá funcionar de la siguiente manera:

1. 200 **clientes** (hilos) que dejarán 2 Pedidos (String) en el objeto compartido Mostrador Restaurante, enfocado como un ArrayList de capacidad limitada a 10 elementos. Ejecución limitada a 2 pedidos por cliente.
2. 2 **empleados** (hilos) recogerán los Pedidos del objeto compartido Mostrador Restaurante y los deposita en otro objeto compartido, Mesa de Platos, también un ArrayList de capacidad 20. Ejecución infinita.
3. 3 **cocineros** (hilos) recogerán los Pedidos de Mesa de Platos y los prepararán. Ejecución infinita.

En todo caso hay que comprobar el estado de ocupación de los ArrayList, verificando si están vacíos o llenos y actuar en consecuencia a ello. Por otro lado, los hilos dormirán entre acciones un tiempo determinado por un método aleatorio limitado entre ciertos valores.

Adicionalmente, se requiere guardar un log en un fichero txt del funcionamiento del restaurante, incluyendo una marca de tiempo.

Todo lo anterior, debe reflejarse de forma gráfica en una ventana que ilustre el comportamiento de los empleados (2), cocineros (3) y objetos compartidos (2), Mostrador Restaurante y Mesa de Platos), de forma que podamos observar en todo momento qué pedido es el que se está manejando y en qué fase del ciclo se encuentra.

Esta interfaz gráfica contará con botones que permitan pausar y reanudar un empleado (a elección del usuario) o ambos a la vez.

Además, el programa deberá contar con una ventana gráfica adicional, un módulo de visualización que nos permita observar el contenido de los objetos compartidos de forma distribuida.

2. Diseño general del sistema y herramientas de sincronización utilizadas

Descripción detallada de la solución

- Crearemos una clase principal, **Restaurante,** que será la encargada de lanzar los hilos (cliente, empleado y cocinero), inicializar y crear el log, además de actuar como el servidor y ser la parte gráfica correspondiente al Restaurante.

- En el sistema productor-consumidor, tendremos lo siguiente:
  - Clase **Cliente,** en cuyo método run se producirán los dos Pedidos, siendo estos un *String* que seguirán el patrón *idCliente* + *-PX*. Primero se dejará el Pedido 1 y luego el Pedido 2.
  - Estos Pedidos se almacenarán en un ArrayList de *Strings* en el objeto compartido **Mostrador Pedidos**, de tamaño 10. Se decidió hacer uso de un ArrayList frente a, por ejemplo, un Buffer, debido a la facilidad y comodidad que nos suponía el manejo del mismo. En esta clase tendremos dos métodos, dejarPedido() y cogerPedido(), encargados de añadir y eliminar Pedidos al Array respectivamente.

Para proteger las secciones críticas y evitar problemas de inanición, interbloqueos y condiciones de carrera nos aseguramos de garantizar el acceso a la variable compartida ArrayList mediante la exclusión mutua, asegurando que un único hilo entra dentro de dicha sección, modifica (añade o quita un pedido) y sale sin problemas. Para nuestros objetos compartidos **Mostrador Pedidos** y **Mesa Platos** decidimos emplear cerrojos explícitos o *locks*. Podríamos haber protegido las secciones críticas con otros elementos, como semáforos, monitores o cerrojos implícitos. El motivo de la elección de cerrojos explícitos fue debido únicamente a la comodidad que nos suponía frente a los semáforos, por haberlos tratado con más frecuencia a lo largo de la asignatura.

Adicionalmente, ya que simplemente con los cerrojos no solucionaremos los problemas de sincronización, emplearemos dos condition, lleno y vacío que nos permitirán controlar el estado del ArrayList y decidirán sin un hilo debe esperar para retirar un pedido del mismo (en caso de que esté vacío) o esperar para añadir un nuevo pedido en el caso de que esté lleno.

De esta manera, conseguimos resolver los problemas de comunicación y sincronización.

Explicaremos más en detalle el funcionamiento en el siguiente fragmento de código:

```
public void dejarPedido(String pedido) throws InterruptedException {
        cerrojo.lock(); // El hilo adquiere el lock
        while (numElem == maximo) { // Bucle en el que el hilo entra si el ArrayList está lleno
            lleno.await(); // Si lo está, espera. El hilo se bloquea hasta que alguien lo despierte
        }
        try { // Hacemos uso de try-catch por seguridad
              // Vamos añadiendo los pedidos al ArrayList
            pedidosMostrador.add(pedido);
            numElem++;
            vacio.signal(); //El ArrayList ya no está vacío, desbloquea el await
            text.setText(getpedidosMostrador().toString());//Mostramos la info
gráficamente
        } finally { //Con finally aseguraremos que el cerrojo se libere y no se quede bloqueado
            cerrojo.unlock();//Liberamos el lock
        }
    }
```

- Clase **Empleado** consumirá lo que haya producido el **Cliente** (sacará los Pedidos del ArrayList del objeto compartido **Mostrador Pedidos** de uno en uno) y a su vez será un productor, ya que se encargará de insertar esos Pedidos al segundo objeto compartido, **Mesa Platos.**
- Para finalizar, el **Cocinero** consumirá dicho Pedido previamente insertado en el ArrayList por el **Empleado** sacándolo del ArrayList del objeto compartido y finalizando.

- La clase **Log** se encargará de crear un archivo .txt en caso de no existir, y escribir en él. Protegeremos la sección crítica del método escribir() mediante cerrojos explícitos (ya explicados) para evitar que haya 'interferencias' entre hilos (un hilo escriba antes que otro, cuando no corresponde).
- La clase **Pausar** será la encargada de pausar la ejecución de los hilos. Se decidió proteger las secciones críticas mediante monitores y cerrojos implícitos (*synchronized*).

Explicando en detalle:

```
public synchronized void comprobarHilo() throws InterruptedException{
//Synchronized para garantizar la exclusión mutua de la sección crítica
//Comprueba el estado del hilo, si la parada está a true, entra en el bucle y espera hasta que cambie a false
        while (parada) {//Mientras que parada == True...
                wait(); // El proceso queda bloqueado hasta que un hilo lo despierte
        }
    }
public synchronized void parar() {//Volvemos a garantizar exclusión mutua
        parada = true;
    }

    public synchronized void seguir() {
        parada = false;
        notifyAll();//Despertaremos a TODOS los hilos que previamente hicieron un wait()
    }
```

Para la parte concurrente distribuida, teníamos la opción de hacer uso de TCP, UDP o RMI. Decidimos hacer uso de RMI dado que es un mecanismo bastante sencillo de implementar y que ofrece la fiabilidad de TCP pero sin la necesidad de preocuparnos de los canales. Tendremos las siguientes clases:

- **Cliente RMI** será una clase principal además de la parte gráfica del Módulo de Visualización. Será la clase encargada de lanzar el hilo **HiloRMI.**

- **HiloRMI** es un hilo encargado con la función 'real' de un cliente RMI. Se encargará de localizar la referencia remota, buscando el objeto remoto en el registro de nombres RMI (*Naming.lookup("//127.0.0.1/ModuloV")*) e invocará los métodos del objeto remoto. Imprimirá en el jTextField el contenido del ArrayList.
- **ObjetoRMI** será la clase que contiene los códigos de los métodos remotos mostrarPedidos() y mostrarPlatos() que se encargará de mostrar en la parte gráfica del Módulo los Array correspondientes. Se recorrerán y extraerán elemento a elemento para su conversión de ArrayList a String.
- **InterfaceRMI** contendrá la cabecera de los métodos remotos mostrarPedidos() y mostrarPlatos() para poder ser llamados de forma remota.
- **Restaurante,** por la parte que actuará como servidor, creará, registrará el objeto remoto **ObjetoRMI** y lo hará visible. Esta clase implementará la interface remota.

3. Clases principales que intervienen

Nuestro proyecto está formado por 12 clases.

3.1 Clase Cliente

Hereda de la clase hilo y tendrá los siguientes atributos que además formarán el constructor de la clase.

- idCliente será de tipo String y nos permitirá identificar al cliente.
- pedido de la clase MostradorPedidos dónde almacenaremos los pedidos de nuestro cliente y será una de las variables compartidas,
- log de la clase Log que nos permitirá almacenar en el mismo log el comportamiento de los clientes.
- pararTodo de la clase Pausar, a través de ese objeto reanudaremos y pararemos la ejecución de nuestros clientes.

Esta clase tiene un método, que será public void run() que contiene el código a ejecutar por el hilo. Primero comprobaremos si el hilo cliente está detenido o no a través del método comprobarHilo. Después en una variable string llamada pedido almacenaremos el idCliente con su número de pedido "-P1" y mediante el método dejarPedido almacenaremos el pedido en la variable compartida. Guardaremos la información de lo que hace ese cliente en date con la fecha y hora del sistema para después almacenarlo en el log a través del método escribir. Ese cliente se irá a "dormir" un tiempo entre 500 ms y 1000 ms. Después volvemos a comprobar el hilo, guardaremos la variable pedido con el nombre del cliente y su número de pedido "-P2" y los demás que suceden son iguales a lo explicado anteriormente.

```
public void run() {
    String pedido;
    // Hay 200 clientes, ejecutarán su tarea y finalizarán
    try {
        pararTodo.comprobarHilo();
        String pedido1 = idCliente + "-P1";
        mpedidos.dejarPedido(pedido1);
        int tiempo = (1000 + (int) (-500 * Math.random()));
        System.out.println("El " + idCliente + " deja un primer plato " + pedido1);
        Date date = Calendar.getInstance().getTime();
        pedido=date+": "+" El " + idCliente + " deja un primer plato " + pedido1;
        log.escribir(pedido);
        sleep(tiempo);

        pararTodo.comprobarHilo();
        String pedido2 = idCliente + "-P2";
        mpedidos.dejarPedido(pedido2);
        System.out.println("El " + idCliente + " deja un segundo plato " + pedido2);
        date = Calendar.getInstance().getTime();
        pedido=date+":"+" El " + idCliente + " deja un segundo plato " + pedido2;
        log.escribir(pedido);

    } catch (InterruptedException e) {
    }

}
```

3.2 Clase Cocinero

Hereda de la clase hilo y tendrá los siguientes atributos que además formarán el constructor de la clase.

- idCocinero será de tipo String y nos permitirá identificar al cocinero.
- mplatos será de la clase MesaPlatos donde estarán almacenados los pedidos de los clientes en cocina y será otra variable compartida.
- text será de la clase JTextField.
- log de la clase Log que nos permitirá almacenar en el mismo log el comportamiento de los cocineros
- pararTodo de la clase Pausar, a través de ese objeto reanudaremos y pararemos la ejecución de nuestro cocinero.

La clase tiene un método, que será public void run() que contiene el código a ejecutar por el hilo. Tenemos un while True ya que los cocineros sólo paran cuando ya no hay más pedidos. Primero comprobaremos si el cocinero se tiene que detener a través del método comprobarHilo(). Después el cocinero retirará el pedido de la variable compartida a través del método cogerPedidoCocina. Al igual que en la clase cliente almacenaremos la fecha y hora y el comportamiento del cocinero en una variable llamada dato que la utilizaremos para guardar la información en el log a través del método escribir. Por último reflejaremos lo que hace el cocinero mediante un setText ya que es lo que necesitamos que muestre en la interfaz gráfica de la práctica. Volveremos a comprobar si el hilo está detenido. El cocinero

se para un tiempo aproximado entre 1500 ms y 2000 ms y después se vuelve a repetir todo los explicado anteriormente.

```
public void run() {
    String dato;
    String pedido;
    while (true) { // Los cocineros nunca cesan su ejecución
        try {
            pararTodo.comprobarHilo();
            pedido = mplatos.cogerPedidoCocina();  //cogen el pedido de la cocina
            System.out.println("El " + idCocinero + " ha recogido el plato "+ pedido);
            Date date = Calendar.getInstance().getTime();
            dato=date+": "+ " El " + idCocinero + " ha recogido el plato "+ pedido;
            log.escribir(dato);

            text.setText("Cocinando pedido " + pedido);
            pararTodo.comprobarHilo();
            int tiempo = (2000 + (int) (-1500 * Math.random()));
            sleep(tiempo);
            System.out.println("El " + idCocinero + " ha terminado y vuelve a por otro plato... ");
            date = Calendar.getInstance().getTime();
            dato=date+": "+" El " + idCocinero + " ha terminado y vuelve a por otro plato... ";
            log.escribir(dato);
        } catch (InterruptedException e) {
        }
    }
}
```

3.3 Clase Empleado

Hereda de la clase hilo y tendrá los siguientes atributos que además formarán el constructor de la clase. El empleado es el único que accede a ambas variables compartidas.

- idEmpleado será un String.
- mpedidos de la clase MostradorPedidos.
- mplatos de la clase MesaPlatos.
- text de la clase JTextField.
- log de la clase Log.
- pararTodo de la clase Pausar.
- pararEmpleado de la clase Pausar.

La clase tiene un método que será public void run() que contiene el código a ejecutar por el hilo. Aquí haremos dos comprobaciones ya que podemos detener todos los hilos (Empleados, cocineros y clientes) o detener un empleado en concreto, y haremos esas comprobaciones a través del método comprobarHilo del que hablaremos más adelante. El empleado retira el pedido a través del método cogerPedido de la variable compartida mpedidos (almacenará los pedidos de los clientes). El comportamiento de lo que está haciendo el empleado más la fecha y hora del sistema se almacenará en una variable llamada dato que posteriormente utilizaremos a través del método escribir para que refleje la acción realizada por el empleado

en el log. Volveremos a comprobar si el hilo está detenido (ya sea porque todos lo están o solo el empleado). Ahora el empleado llevará el pedido a la "cocina" y dejará el pedido en la variable compartida mplatos a través del método dejarPedidoCocina. También hacemos que se refleje esa misma acción en un setText, para que aparezca en la interfaz gráfica del programa.

Registramos ese comportamiento en el log como hemos comentado anteriormente. Después el hilo empleado descansará entre 300 y 700 ms.

```
String pedido;
String dato;
while (true) { // Los empleados nunca cesan su ejecución
    try {
        pararTodo.comprobarHilo();
        pararEmpleado.comprobarHilo();
        pedido = mpedidos.cogerPedido(); //recoger pedido del mostrador
        System.out.println("El " + idEmpleado + " ha recogido el plato del mostrador " + pedido);
        Date date = Calendar.getInstance().getTime();
        dato=date+": "+" El " + idEmpleado + " ha recogido el plato del mostrador " + pedido;
        log.escribir(dato);

        pararTodo.comprobarHilo();
        pararEmpleado.comprobarHilo();
        mplatos.dejarPedidoCocina(pedido); //dejar pedido en la cocina
        System.out.println("El " + idEmpleado + " ha dejado el plato en cocina " + pedido);
        date=Calendar.getInstance().getTime();
        dato=date+": "+" El " + idEmpleado + " ha dejado el plato en cocina " + pedido;
        log.escribir(dato);

        text.setText("Llevando " + pedido + " a la mesa de platos");
        pararTodo.comprobarHilo();
        pararEmpleado.comprobarHilo();
        int tiempo = (700 + (int) (-300 * Math.random()));
        sleep(tiempo);
        System.out.println("El " + idEmpleado + " ha terminado y vuelve a por otro plato... ");
        date=Calendar.getInstance().getTime();
        dato=date+": "+" El " + idEmpleado + " ha terminado y vuelve a por otro plato... ";
        log.escribir(dato);
        pararTodo.comprobarHilo();
        pararEmpleado.comprobarHilo();
    } catch (InterruptedException e) {
    }
```

3.4 Clase Log

Esta clase es la que nos permite guardar en un archivo el comportamiento de todos los hilos del programa. Está formada por 3 atributos.

- escritor de la clase FileWriter.
- archivo de la clase File.
- cerrojo de la clase ReentrantLock.

Distinguimos dos métodos, y hay que aclarar que utilizamos File para crear el archivo y comprobar que existe y como no tiene la funcionalidad de escribir en un archivo, utilizaremos FileWriter para poder escribir.

- ❏ crearArchivo():

Instanciamos un objeto de la clase file con la ruta en paréntesis donde queremos que se guarde el archivo. Después si el archivo no existe creamos uno nuevo y si existe lo eliminamos y creamos uno nuevo.

```
public void crearArchivo(){
        try {
         archivo = new File("evolucionRestaurante.txt");
         if (archivo.createNewFile()){
             System.out.println("El archivo se ha creado");
         }else{
             System.out.println("El archivo ya existe y se va a sobreescribir ");
             archivo.delete();
             archivo.createNewFile();
         }
      } catch (IOException e) {
         System.out.println("No se ha podido ejecutar");
         e.printStackTrace();
      }
}
```

- ❏ escribir(String texto):

  Vamos a proteger este método, ya que va a estar disponible para los distintos hilos y para evitar condiciones de carrera.

  Igualamos nuestra variable escritor a la misma ruta del File.

```
public void escribir(String texto){
    cerrojo.lock();
    try {
        escritor=new FileWriter("evolucionRestaurante.txt",true); //El escritor se
        escritor.write(texto+"\n");
        escritor.flush();
        escritor.close();
        } catch (IOException ex) {
            Logger.getLogger(Log.class.getName()).log(Level.SEVERE, null, ex);
        }
    finally {
        cerrojo.unlock();
    }
}
```

3.5 Clase MesaPlatos

De esta clase haremos el objeto compartido para los objetos de la clase Empleado y Cocinero. Tiene los siguientes atributos:

- pedidosMesa que será un ArrayList de String.
- maximo y numElem de tipo int.
- cerrojo de la clase ReentrantLock.
- lleno y vacío que serán de la clase Condition asociados al cerrojo.
- text que será un JTextField.

El constructor de la clase estará formado por max y text e inicializará el arrayList de pedidosMesa.

Distinguimos 4 métodos:

- dejarPedidoCocina(String pedido)

Este método recibe el pedido por parámetro. El hilo que adquiere el lock después tenemos una condición en la que el hilo entra si el ArrayList está lleno y por lo tanto se bloqueará hasta que se despierte (otro hilo que haga un signal). Si no entra, añadimos el pedido al ArrayList, incrementamos la variable numElem. Como ya sabemos que el ArrayList no está vacío, desbloqueamos el await. Con setText, escribimos el contenido del ArrayList a través del método getMesaPlatos del que hablaremos más adelante y finalmente liberamos el lock.

```
public void dejarPedidoCocina(String pedido) throws InterruptedException {
    cerrojo.lock();
    while (numElem == maximo) {
        lleno.await();
    }
    try {
        pedidosMesa.add(pedido);
        numElem++;
        vacio.signal();
        text.setText(getMesaPlatos().toString());
    } finally {
        cerrojo.unlock();
    }
}
```

- cogerPedidoCocina:

Este método devuelve un variable de tipo String. El hilo adquiere el lock y entra en el while si el número de elementos de nuestro ArrayList es 0, es decir que no hay

ningún elemento en la variable pedidosMesa y se queda el hilo bloqueado hasta que otro hilo haga un signal. Si no entra en el while podemos extraer un pedido de pedidosMesa. Disminuimos el número de elementos del ArrayList, desbloqueamos el await.Con setText, escribimos el contenido del ArrayList a través del método getMesaPlatos del que hablaremos más adelante y retornamos la variable pedido. Finalmente liberamos el lock.

```
public String cogerPedidoCocina() throws InterruptedException {
    cerrojo.lock();
    while (numElem == 0) {
        vacio.await();
    }
    try {
        String pedido = pedidosMesa.get(0);
        pedidosMesa.remove(0);
        numElem--;
        lleno.signal();
        text.setText(getMesaPlatos().toString());
        return (pedido);
    } finally {
        cerrojo.unlock();
    }

}
```

- getMesaPlatos:

  Este método nos devuelve un ArrayList de String, en este caso lo utilizamos para que nos devuelva el contenido de los Pedidos del mostrador en la "cocina".

```
public ArrayList<String> getPedidosMostrador() {
    return pedidosMostrador;
}
```

- getMesaPlatos2:

  Este método está protegido mediante cerrojos y nos retorna también un ArrayList de String.

```
public ArrayList<String> getPedidosMostrador2() throws InterruptedException {
    cerrojo.lock();
    try{
       return pedidosMostrador;
    }finally {
        cerrojo.unlock();
    }
}
```

3.6 Clase MostradorPedidos

De esta clase haremos el objeto compartido para los objetos de la Clase Empleado y Cliente. Tiene los siguientes atributos:

- pedidosMostrador será un ArrayList de String.
- maximo y numElem de tipo int.
- cerrojo de tipo ReentrantLock.
- lleno y vacio de tipo Condition.
- text de tipo JTextField.

El constructor de la clase estará formado por max y text e inicializará el arrayList de pedidosMostrador.

Distinguimos 4 métodos:

- dejarPedido(String pedido):

  Este método ya está explicado arriba y se muestra más detalladamente.

- cogerPedido()

  El funcionamiento de este método es exactamente igual al de cogerPedidoCocina() de la clase MesaPlatos.

```
public String cogerPedido() throws InterruptedException {
    cerrojo.lock();
    while (numElem == 0) { //Si el ArrayList está vacio... Espera
        vacio.await();
    }
    try {
        String pedido = pedidosMostrador.get(0);
        pedidosMostrador.remove(0);
        numElem--;
        lleno.signal(); //El ArrayList ya no está lleno
        text.setText(getPedidosMostrador().toString());
        return (pedido);
    } finally {
        cerrojo.unlock();
    }
}
```

- getPedidosMostrador():

  Este método nos devolverá un ArrayList de String.

```
public ArrayList<String> getPedidosMostrador() {
    return pedidosMostrador;
}
```

- getPedidoMostrador2():

  Este método es exactamente igual a getMesaPlatos2 explicado anteriormente.

```
public ArrayList<String> getPedidosMostrador2() throws InterruptedException {
    cerrojo.lock();
    try{
       return pedidosMostrador;
    }finally {
        cerrojo.unlock();
    }
}
```

3.7 Pausar

La clase pausar tiene un atributo llamado parada de tipo boolean inicializado a false. Conforme seleccionemos las opciones de parar(depende del hilo que deseemos detener) en otra clase de la que hablaremos en el siguiente punto, esa variable parada cambiará a true y eso indica que el hilo se va a detener. Consta de 3 métodos: comprobarHilo(), parar() y seguir(). Ya están explicados con detalle en el punto 2.

3.8 Restaurante

Esta clase es la interfaz gráfica de nuestro programa y donde tenemos implementado el servidor.

```
//Servidor Remoto RMI
try {
    //Crea instancia del objeto que implementa la interfaz (objeto a registrar):
    System.out.println("Servidor Arrancado....");
    ObjetoRMI obj = new ObjetoRMI(mplatos, mpedidos);
    Registry registry = LocateRegistry.createRegistry(1099); //Arranca rmiregistry local en el puerto 1099
    Naming.rebind("//127.0.0.1/ModuloV", obj); //Hace visible el objeto para clientes
    System.out.println("El Objeto ModuloV ha quedado registrado");

} catch (Exception e) {
    System.out.println("Error: " + e.getMessage());
    e.printStackTrace();
}
```

Además de instanciar el resto de objetos de las anteriores clases mencionadas (Clientes, Cocinero, Empleado, MostradorPedidos, Mesa platos, Log, Pausar). También tendremos 4 botones:

- BotonReanudarEmpleado1
- BotonReanudarEmpleado2
- BotonPausarEmpleado1
- BotonPausarEmpleado2
- BotonPausarTodo
- BotonReanudarTodo

Los botones de reanudar llaman al objeto de la clase pausar al método seguir() y los botones de pausar llaman al método parar().

3.9 ClienteRMI

Ya está comentado en el apartado 2.

3.10 HiloRMI

Esta clase será un hilo que se encargará de pedirle al servidor RMI los datos de la mesa de platos y la mesa de pedidos. Tiene dos atributos y su constructor estará formado por ellos también.

- textoPedido de la clase JTextField.
- textoPlato de la clase JTextArea.

Solo tiene un método que será public void run() que contiene el código a ejecutar por el hilo.

```
public void run() {
    try {
        //Localiza el objeto distribuido:
        InterfaceRMI obj = (InterfaceRMI) Naming.lookup("//127.0.0.1/ModuloV");
        while (true){
            textoPedido.setText(obj.mostrarPedidos());
            textoPlato.setText(obj.mostrarPlatos());
        }
    } catch (Exception e) {
        System.out.println("Excepcion: " + e.getMessage());
        e.printStackTrace();
    }
}
```

3.11 ObjetoRMI

Esta clase contiene los códigos de los métodos remotos mostrarPedidos() y mostrarPlatos(). A continuación mostramos solo un método ya que el otro sigue la misma estructura. Tenemos un

for que nos permite recorrer mostradorpedidos de forma segura, y según eso vamos sacando los elementos y agregandolo al String pedido.

```
public String mostrarPedidos() throws RemoteException {// Implementación del método remoto
    String pedido = "  ";
    try {
        for (int i=0; i<mostradorpedidos.getPedidosMostrador2().size();i++) {
            pedido = pedido+mostradorpedidos.getPedidosMostrador().get(i)+" , ";
        }
    } catch (InterruptedException ex) {
        Logger.getLogger(ObjetoRMI.class.getName()).log(Level.SEVERE, null, ex);
    }
    return pedido;
}
```

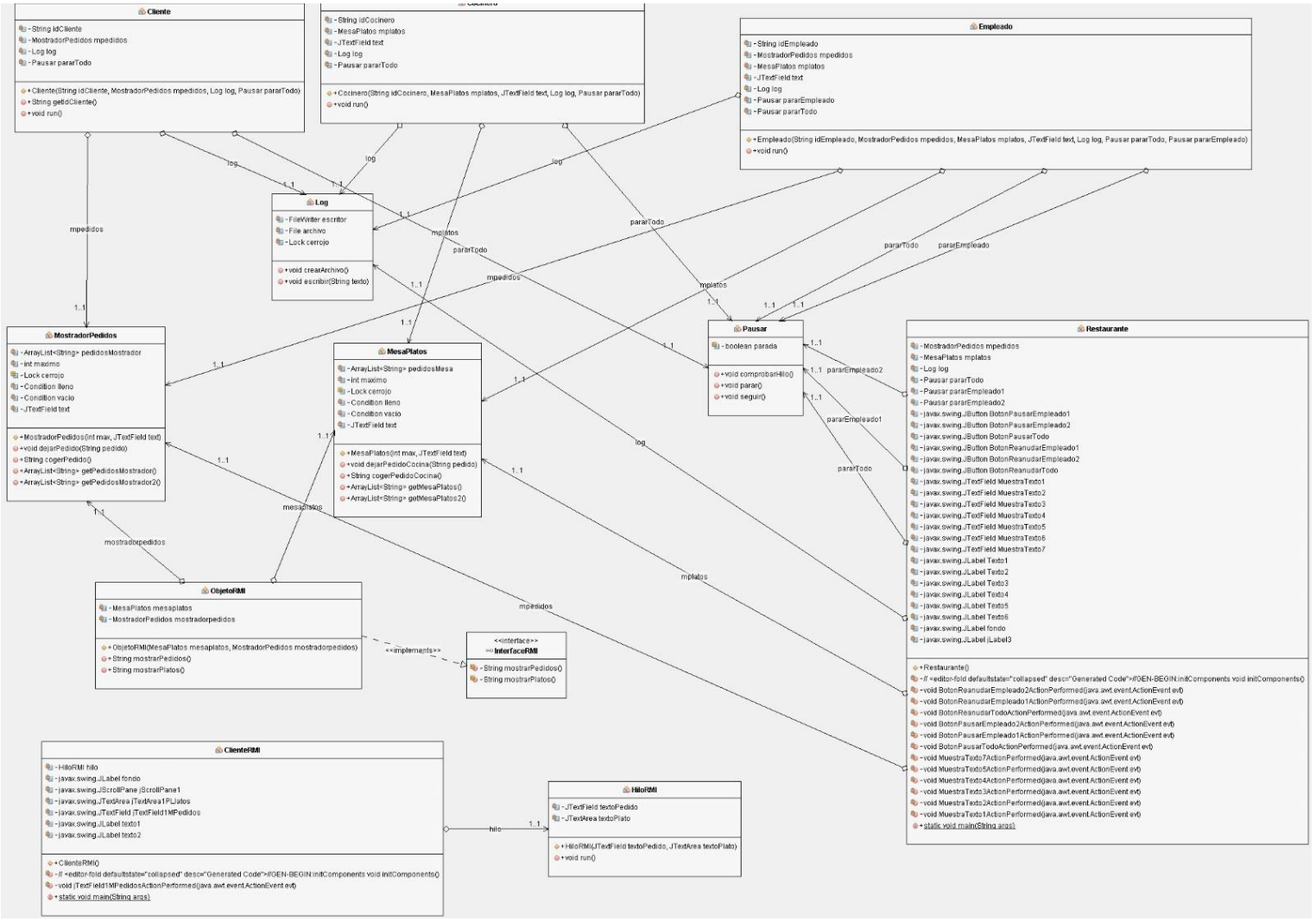
3.12 InterfaceRMI

Esta clase contendrá la cabecera de los métodos remotos mostrarPedidos() y mostrarPlatos() para poder ser llamados de forma remota.

```
public interface InterfaceRMI extends Remote{
    //Método que se publica
    String mostrarPedidos() throws RemoteException;
    String mostrarPlatos() throws RemoteException;

}
```

4. Diagrama UML



Se adjunta la imagen del diagrama por si se quiere ver con mayor claridad.

5. ANEXO - Código del programa

Cliente

```
import java.util.Calendar;
import java.util.Date;

public class Cliente extends Thread {

    private String idCliente;
    private MostradorPedidos mpedidos;
    private Log log;
    private Pausar pararTodo;

    public Cliente(String idCliente, MostradorPedidos mpedidos,Log log, Pausar
pararTodo) {
        this.idCliente = idCliente;
        this.mpedidos = mpedidos;
        this.log=log;
        this.pararTodo = pararTodo;
    }

    public void run() {
        String pedido;
        // Hay 200 clientes, ejecutarán su tarea y finalizarán
        try {
            pararTodo.comprobarHilo();
            String pedido1 = idCliente + "-P1";
            mpedidos.dejarPedido(pedido1);
            int tiempo = (1000 + (int) (-500 * Math.random()));
            System.out.println("El " + idCliente + " deja un primer plato " +
pedido1);
            Date date = Calendar.getInstance().getTime();
            pedido=date+": "+" El " + idCliente + " deja un primer plato " +
pedido1;
            log.escribir(pedido);
            sleep(tiempo);

            pararTodo.comprobarHilo();
            String pedido2 = idCliente + "-P2";
            mpedidos.dejarPedido(pedido2);
            System.out.println("El " + idCliente + " deja un segundo plato " +
pedido2);
            date = Calendar.getInstance().getTime();
            pedido=date+":"+" El " + idCliente + " deja un segundo plato " +
pedido2;
            log.escribir(pedido);

        } catch (InterruptedException e) {
        }

    }
}
```

Cocinero

```
import java.util.Calendar;
import java.util.Date;
import javax.swing.JTextField;

public class Cocinero extends Thread {

    private String idCocinero;
    private MesaPlatos mplatos;
    private JTextField text;
    private Log log;
    private Pausar pararTodo;

    public Cocinero(String idCocinero, MesaPlatos mplatos, JTextField text,Log
log, Pausar pararTodo) {
        this.idCocinero = idCocinero;
        this.mplatos = mplatos;
        this.text = text;
        this.log=log;
        this.pararTodo = pararTodo;
    }

    public void run() {
        String dato;
        String pedido;
        while (true) { // Los cocineros nunca cesan su ejecución
            try {
                pararTodo.comprobarHilo();
                pedido = mplatos.cogerPedidoCocina();  //cogen el pedido de la
cocina
                System.out.println("El " + idCocinero + " ha recogido el plato
"+ pedido);
                Date date = Calendar.getInstance().getTime();
                dato=date+": "+ " El " + idCocinero + " ha recogido el plato "+
pedido;
                log.escribir(dato);

                text.setText("Cocinando pedido " + pedido);
                pararTodo.comprobarHilo();
                int tiempo = (2000 + (int) (-1500 * Math.random()));
                sleep(tiempo);
                System.out.println("El " + idCocinero + " ha terminado y vuelve
a por otro plato... ");
                date = Calendar.getInstance().getTime();
                dato=date+": "+" El " + idCocinero + " ha terminado y vuelve a
por otro plato... ";
                log.escribir(dato);
            } catch (InterruptedException e) {
            }
        }
```

Empleado

```
import java.util.Calendar;
import java.util.Date;
import javax.swing.JTextField;

public class Empleado extends Thread {

    private String idEmpleado;
    private MostradorPedidos mpedidos;
    private MesaPlatos mplatos;
    private JTextField text;
    private Log log;
    private Pausar pararEmpleado;
    private Pausar pararTodo;

    public Empleado(String idEmpleado, MostradorPedidos mpedidos, MesaPlatos
mplatos, JTextField text, Log log, Pausar pararTodo, Pausar pararEmpleado) {
        this.idEmpleado = idEmpleado;
        this.mpedidos = mpedidos;
        this.mplatos = mplatos;
        this.text = text;
        this.log= log;
        this.pararTodo = pararTodo;
        this.pararEmpleado = pararEmpleado;
    }

    public void run() {
        String pedido;
        String dato;
        while (true) { // Los empleados nunca cesan su ejecución
            try {
                pararTodo.comprobarHilo();
                pararEmpleado.comprobarHilo();
                pedido = mpedidos.cogerPedido(); //recoger pedido del mostrador
                System.out.println("El " + idEmpleado + " ha recogido el plato
del mostrador " + pedido);
                Date date = Calendar.getInstance().getTime();
                dato=date+": "+" El " + idEmpleado + " ha recogido el plato del
mostrador " + pedido;
                log.escribir(dato);

                pararTodo.comprobarHilo();
                pararEmpleado.comprobarHilo();
                mplatos.dejarPedidoCocina(pedido); //dejar pedido en la cocina
                System.out.println("El " + idEmpleado + " ha dejado el plato en
cocina " + pedido);
                date=Calendar.getInstance().getTime();
                dato=date+": "+" El " + idEmpleado + " ha dejado el plato en
cocina " + pedido;
                log.escribir(dato);
```

```
                text.setText("Llevando " + pedido + " a la mesa de platos");
                pararTodo.comprobarHilo();
                pararEmpleado.comprobarHilo();
                int tiempo = (700 + (int) (-300 * Math.random()));
                sleep(tiempo);
                System.out.println("El " + idEmpleado + " ha terminado y vuelve
a por otro plato... ");
                date=Calendar.getInstance().getTime();
                dato=date+": "+" El " + idEmpleado + " ha terminado y vuelve a
por otro plato... ";
                log.escribir(dato);
                pararTodo.comprobarHilo();
                pararEmpleado.comprobarHilo();
            } catch (InterruptedException e) {
            }
        }
    }

}
```

MesaPlatos

```
import java.util.ArrayList;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import javax.swing.JTextField;

public class MesaPlatos {

    private ArrayList<String> pedidosMesa; //Array de pedidos
    private int maximo = 0, numElem = 0;
    Lock cerrojo = new ReentrantLock();
    private Condition lleno = cerrojo.newCondition();
    private Condition vacio = cerrojo.newCondition();
    private JTextField text;

    public MesaPlatos(int max, JTextField text) {
        this.maximo = max;
        this.pedidosMesa = new ArrayList<>();
        this.text = text;

    }

    public void dejarPedidoCocina(String pedido) throws InterruptedException {
        cerrojo.lock();
        while (numElem == maximo) {
            lleno.await();
        }
        try {
            pedidosMesa.add(pedido);
            numElem++;
            vacio.signal();
            text.setText(getMesaPlatos().toString());
```

```
        } finally {
            cerrojo.unlock();
        }
    }

    public String cogerPedidoCocina() throws InterruptedException {
        cerrojo.lock();
        while (numElem == 0) {
            vacio.await();
        }
        try {
            String pedido = pedidosMesa.get(0);
            pedidosMesa.remove(0);
            numElem--;
            lleno.signal();
            text.setText(getMesaPlatos().toString());
            return (pedido);
        } finally {
            cerrojo.unlock();
        }

    }
```

MostradorPedidos

```
import java.util.ArrayList;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import javax.swing.JTextField;

public class MostradorPedidos {

    private ArrayList<String> pedidosMostrador; //Array de pedidos
    private int maximo = 0, numElem = 0;
    Lock cerrojo = new ReentrantLock();
    private Condition lleno = cerrojo.newCondition();
    private Condition vacio = cerrojo.newCondition();
    private JTextField text;


    public MostradorPedidos(int max, JTextField text) {
        this.maximo = max;
        this.pedidosMostrador = new ArrayList<>();
        this.text = text;
    }

    public void dejarPedido(String pedido) throws InterruptedException {
        cerrojo.lock();
        while (numElem == maximo) { //Si el ArrayList está lleno... Espera
            lleno.await();
        }
```

```
        try {
            //Vamos añadiendo los pedidos de la clase productor-cocinero
            pedidosMostrador.add(pedido);
            numElem++;
            vacio.signal(); //El ArrayList ya no está vacío
            text.setText(getPedidosMostrador().toString());
        } finally {
            cerrojo.unlock();
        }
    }

    public String cogerPedido() throws InterruptedException {
        cerrojo.lock();
        while (numElem == 0) { //Si el ArrayList está vacio... Espera
            vacio.await();
        }
        try {
            String pedido = pedidosMostrador.get(0);
            pedidosMostrador.remove(0);
            numElem--;
            lleno.signal(); //El ArrayList ya no está lleno
            text.setText(getPedidosMostrador().toString());
            return (pedido);
        } finally {
            cerrojo.unlock();
        }
    }

    public ArrayList<String> getPedidosMostrador() {
        return pedidosMostrador;
    }

    public ArrayList<String> getPedidosMostrador2() throws InterruptedException
{
        cerrojo.lock();
        try{
           return pedidosMostrador;
        }finally {
            cerrojo.unlock();
        }
    }
}
```

Log

```
import java.io.*;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Log {
```

```
    private FileWriter escritor;
    private File archivo;
    private Lock cerrojo = new ReentrantLock();

    public void crearArchivo(){
           try {
            archivo = new File("evolucionRestaurante.txt");
            if (archivo.createNewFile()){
                System.out.println("El archivo se ha creado");
            }else{
                System.out.println("El archivo ya existe y se va a
sobreescribir ");
                archivo.delete();
                archivo.createNewFile();
            }
        } catch (IOException e) {
            System.out.println("No se ha podido ejecutar");
            e.printStackTrace();
        }
    }

    public void escribir(String texto){
        cerrojo.lock();
        try {
            escritor=new FileWriter("evolucionRestaurante.txt",true); //El
escritor se inicializa con la misma ruta de antes para estar listo para
escribir
            escritor.write(texto+"\n");
            escritor.flush();
            escritor.close();
            } catch (IOException ex) {
                Logger.getLogger(Log.class.getName()).log(Level.SEVERE, null,
ex);
            }
        finally {
            cerrojo.unlock();
        }
    }
}
```

Pausar

```
public class Pausar {
    boolean parada = false;

    public synchronized void comprobarHilo() throws InterruptedException {
        //Comprueba el estado del hilo, si la parada está a true, entra en el
bucle y espera hasta que cambie a false
        while (parada) {
                wait();
        }
    }
     public synchronized void parar() {
```

```
        parada = true;
    }

    public synchronized void seguir() {
        parada = false;
        notifyAll();
    }
}
```

Restaurante

```
import java.awt.Dimension;
import java.rmi.Naming;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;


public class Restaurante extends javax.swing.JFrame {

    /**
     * Creates new form Restaurante
     */
    private MostradorPedidos mpedidos;
    private MesaPlatos mplatos;
    private Log log;
    private Pausar pararTodo;
    private Pausar pararEmpleado1;
    private Pausar pararEmpleado2;


    public Restaurante() {
        initComponents();
        this.setSize(new Dimension(1200,800));
        this.setLocationRelativeTo(null);
        this.setTitle(" RESTAURANTE ");

        //Mostrador aceptará máximo 10 pedidos y la mesa 20 pedidos
        mpedidos = new MostradorPedidos(10, MuestraTexto1);
        mplatos = new MesaPlatos(20, MuestraTexto4);
        log = new Log(); //Inicializamos el log
        log.crearArchivo();//Creamos el log
        pararEmpleado1 = new Pausar(); //Inicializamos Pausar
        pararEmpleado2 = new Pausar();
        pararTodo = new Pausar();

        // Crearemos los hilos
        //200 Clientes  + su identificador + variable compartida Mostrador de
Pedidos
        for (int i = 1; i <= 200;i++) {
            Cliente cli = new Cliente("Cliente" + i, mpedidos,log,pararTodo);
            cli.start();
        }
        //2 Empleados + variables compartidas Mesa de Platos y Mostrador de
```

```
Pedidos
        Empleado emp1 = new Empleado("Empleado1", mpedidos, mplatos,
MuestraTexto2,log, pararTodo, pararEmpleado1);
        Empleado emp2 = new Empleado("Empleado2", mpedidos, mplatos,
MuestraTexto3,log, pararTodo, pararEmpleado2);

        //3 Cocineros + variable compartida Moesa de platos
        Cocinero co1 = new Cocinero("Cocinero1", mplatos, MuestraTexto5,log,
pararTodo);
        Cocinero co2 = new Cocinero("Cocinero2", mplatos, MuestraTexto6,log,
pararTodo);
        Cocinero co3 = new Cocinero("Cocinero3", mplatos, MuestraTexto7,log,
pararTodo);
        //Lanzamos los hilos
        emp1.start();
        emp2.start();
        co1.start();
        co2.start();
        co3.start();

        //Servidor Remoto RMI
        try {
            //Crea instancia del objeto que implementa la interfaz (objeto a
registrar):
            System.out.println("Servidor Arrancado....");
            ObjetoRMI obj = new ObjetoRMI(mplatos, mpedidos);
            Registry registry = LocateRegistry.createRegistry(1099); //Arranca
rmiregistry local en el puerto 1099
            Naming.rebind("//127.0.0.1/ModuloV", obj); //Hace visible el objeto
para clientes
            System.out.println("El Objeto ModuloV ha quedado registrado");

        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
            e.printStackTrace();
        }
    }

…

private void BotonReanudarEmpleado2ActionPerformed(java.awt.event.ActionEvent
evt) {//GEN-FIRST:event_BotonReanudarEmpleado2ActionPerformed
        // TODO add your handling code here:
        pararEmpleado2.seguir();
    }//GEN-LAST:event_BotonReanudarEmpleado2ActionPerformed

    private void
BotonReanudarEmpleado1ActionPerformed(java.awt.event.ActionEvent evt)
{//GEN-FIRST:event_BotonReanudarEmpleado1ActionPerformed
        // TODO add your handling code here:
        pararEmpleado1.seguir();

    }//GEN-LAST:event_BotonReanudarEmpleado1ActionPerformed

    private void BotonReanudarTodoActionPerformed(java.awt.event.ActionEvent
```

```
evt) {//GEN-FIRST:event_BotonReanudarTodoActionPerformed
        // TODO add your handling code here:
        pararTodo.seguir();
    }//GEN-LAST:event_BotonReanudarTodoActionPerformed

    private void BotonPausarEmpleado2ActionPerformed(java.awt.event.ActionEvent
evt) {//GEN-FIRST:event_BotonPausarEmpleado2ActionPerformed
        // TODO add your handling code here:
        pararEmpleado2.parar();
    }//GEN-LAST:event_BotonPausarEmpleado2ActionPerformed

    private void BotonPausarEmpleado1ActionPerformed(java.awt.event.ActionEvent
evt) {//GEN-FIRST:event_BotonPausarEmpleado1ActionPerformed
        // TODO add your handling code here:
        pararEmpleado1.parar();

    }//GEN-LAST:event_BotonPausarEmpleado1ActionPerformed

    private void BotonPausarTodoActionPerformed(java.awt.event.ActionEvent evt)
{//GEN-FIRST:event_BotonPausarTodoActionPerformed
        // TODO add your handling code here:
        pararTodo.parar();
    }//GEN-LAST:event_BotonPausarTodoActionPerformed
```

ClienteRMI

```
import java.awt.Dimension;

public class ClienteRMI extends javax.swing.JFrame {
    private HiloRMI hilo;

    public ClienteRMI(){
        initComponents();
        this.setSize(new Dimension(1100,300));
        this.setLocationRelativeTo(null);
        this.setTitle(" MÓDULO DE VISUALIZACIÓN ");

        hilo = new HiloRMI(jTextField1MPedidos,jTextArea1PLlatos);
        hilo.start();
     }
```

InterfaceRMI

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface InterfaceRMI extends Remote{
    //Método que se publica
    String mostrarPedidos() throws RemoteException;
    String mostrarPlatos() throws RemoteException;
```

```
}
```

ObjetoRMI

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.logging.Level;
import java.util.logging.Logger;


public class ObjetoRMI extends UnicastRemoteObject implements InterfaceRMI{
    private MesaPlatos mesaplatos;
    private MostradorPedidos mostradorpedidos;

    public ObjetoRMI(MesaPlatos mesaplatos, MostradorPedidos mostradorpedidos)
throws RemoteException {
        this.mesaplatos = mesaplatos;
        this.mostradorpedidos = mostradorpedidos;
    }
    public String mostrarPedidos() throws RemoteException {// Implementación
del método remoto
        String pedido = "  ";
        try {
            for (int i=0; i<mostradorpedidos.getPedidosMostrador2().size();i++)
{
                pedido = pedido+mostradorpedidos.getPedidosMostrador().get(i)+"
, ";
            }
        } catch (InterruptedException ex) {
            Logger.getLogger(ObjetoRMI.class.getName()).log(Level.SEVERE, null,
ex);
        }
        return pedido;
    }
    public String mostrarPlatos() throws RemoteException {
        String plato = "  ";
        try {
            for (int i=0; i<mesaplatos.getMesaPlatos2().size();i++) {
                plato = plato+mesaplatos.getMesaPlatos().get(i)+" , ";
            }
        } catch (InterruptedException ex) {
            Logger.getLogger(ObjetoRMI.class.getName()).log(Level.SEVERE, null,
ex);
        }
        return plato;
    }
}
```

HiloRMI

```
import java.rmi.Naming;
import java.util.ArrayList;
import javax.swing.JTextArea;
import javax.swing.JTextField;


public class HiloRMI extends Thread {

     private JTextField textoPedido;
     private JTextArea textoPlato;

    public HiloRMI(JTextField textoPedido, JTextArea textoPlato) {
        this.textoPedido = textoPedido;
        this.textoPlato = textoPlato;
    }
     public void run() {
        try {
            //Localiza el objeto distribuido:
            InterfaceRMI obj = (InterfaceRMI)
Naming.lookup("//127.0.0.1/ModuloV");
            while (true){
                textoPedido.setText(obj.mostrarPedidos());
                textoPlato.setText(obj.mostrarPlatos());
            }
        } catch (Exception e) {
            System.out.println("Excepcion: " + e.getMessage());
            e.printStackTrace();
        }
    }

}
```