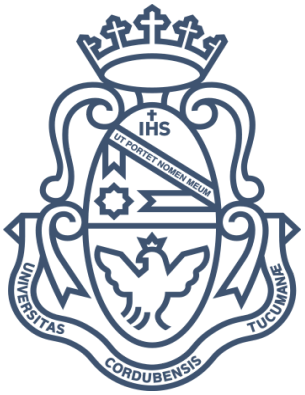


UNIVERSIDAD NACIONAL DE CÓRDOBA

FACULTAD DE CIENCIAS EXACTAS, FÍSICAS Y NATURALES
CÁTEDRA DE PROGRAMACIÓN CONCURRENTE
EQUIPO ***DEADLOCK PARRILLA***

Trabajo Práctico Nº1:

Desarrollo de un sistema concurrente para la gestión de entregas de productos

Docentes:

- Luis Orlando Ventre
- Mauricio Ludemann

Integrantes:

- Dante Maximiliano Ruiz
- Matias Leonel Sacchi
- Facundo Weihmüller
- Agustin Alvarez

Introducción

Mediante este informe se realizará un análisis detallado de un sistema concurrente para simular la gestión de entregas de productos comprados a través de una plataforma de e-commerce, que fue desarrollado por el equipo **DeadLock Parrilla**. El objetivo es explicar la lógica de nuestro programa, que fue implementado en JAVA, y comentar qué decisiones tomamos para cumplir con los siguientes requerimientos.

Nuestro sistema debe realizar la gestión de 500 pedidos almacenado en una matriz de 200 espacio, en donde cada uno de ellos tiene que pasar por 4 etapas:

- Preparación (3 hilos)
- Despacho: (2 hilos)
- Entrega: (3 hilos)
- Verificación :(2 hilos)

El sistema debe contar con un LOG con fines estadísticos, el cual registre cada 200 milisegundos en un archivo:

- Cantidad de pedidos fallidos.
- Cantidad de pedidos verificados.

Por cuenta propia, por motivos de control, optamos por añadir al Log:

- Cantidad de pedidos en preparación.
- Cantidad de pedidos en transición.
- Cantidad de pedidos por entregar.

Al finalizar el LOG debe imprimir una estadística de los casilleros y el tiempo total que demoró el programa, el cual debe demorar un rango de 15 a 30 segundos.

Desarrollo

Primeramente construimos el diagrama de clases, para tener una visión más clara y gráfica de cómo se relacionan las partes de nuestro programa, al cual hemos ido actualizando y modificando en el transcurso del desarrollo del programa hasta obtener una diagrama completo :

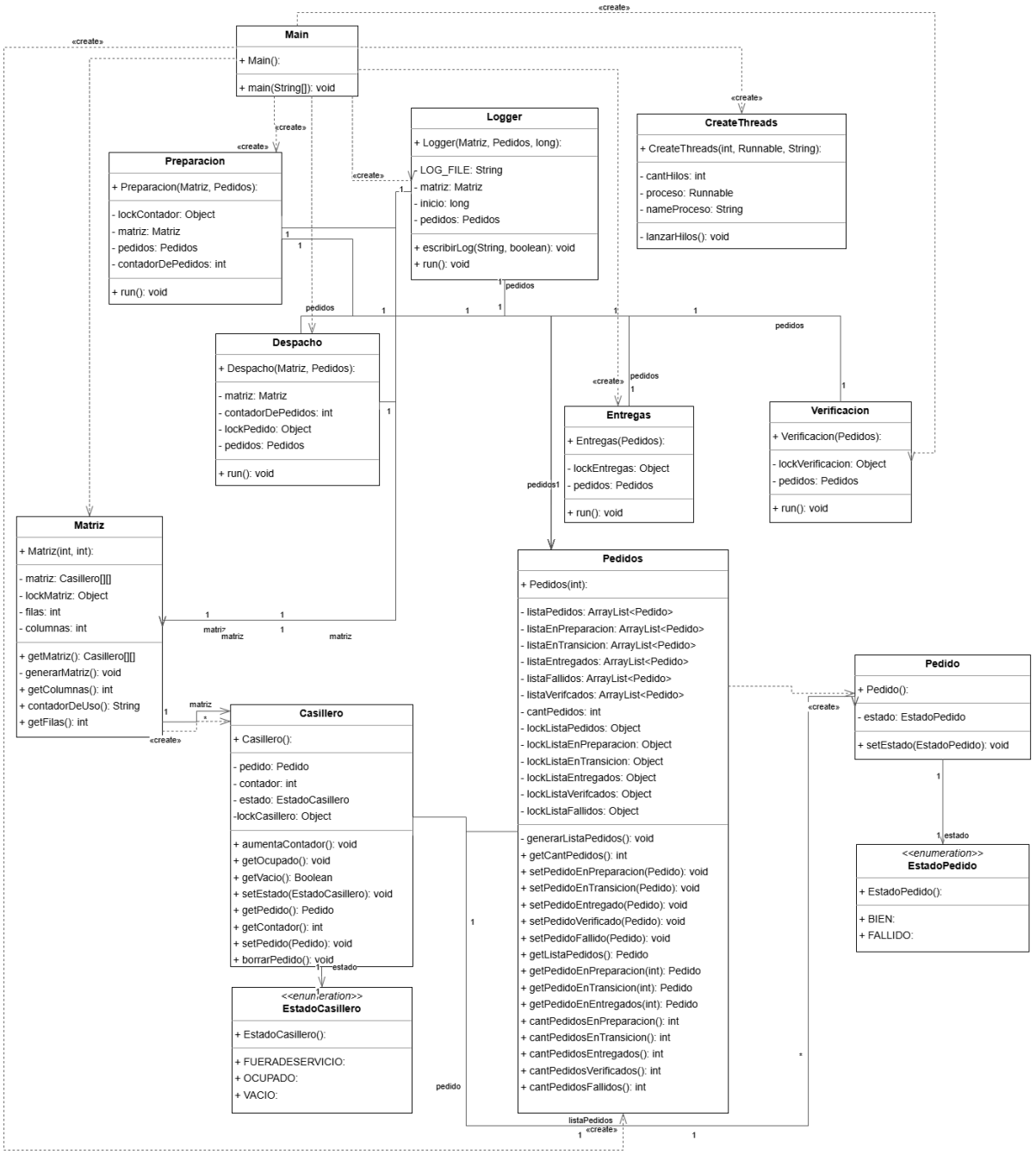


Figura 1: Diagrama de clases

El mismo puede ser observado con mayor detenimiento en su archivo **drive**

Por otro lado, tenemos el diagrama de secuencia, el cual sigue todo el recorrido de un pedido y muestra las interacciones entre los objetos creados durante la ejecución del programa:

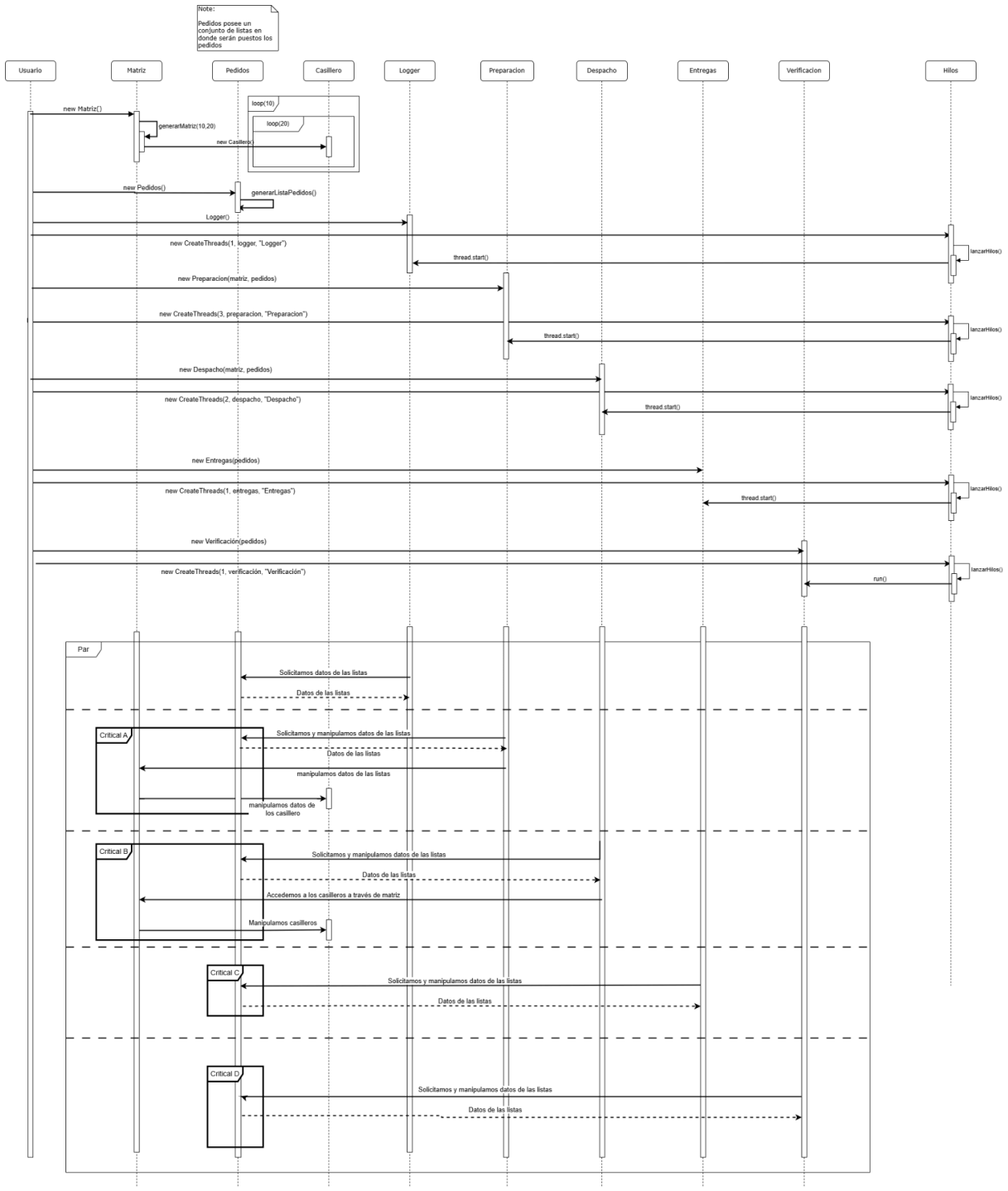


Figura 2: Diagrama de secuencia general

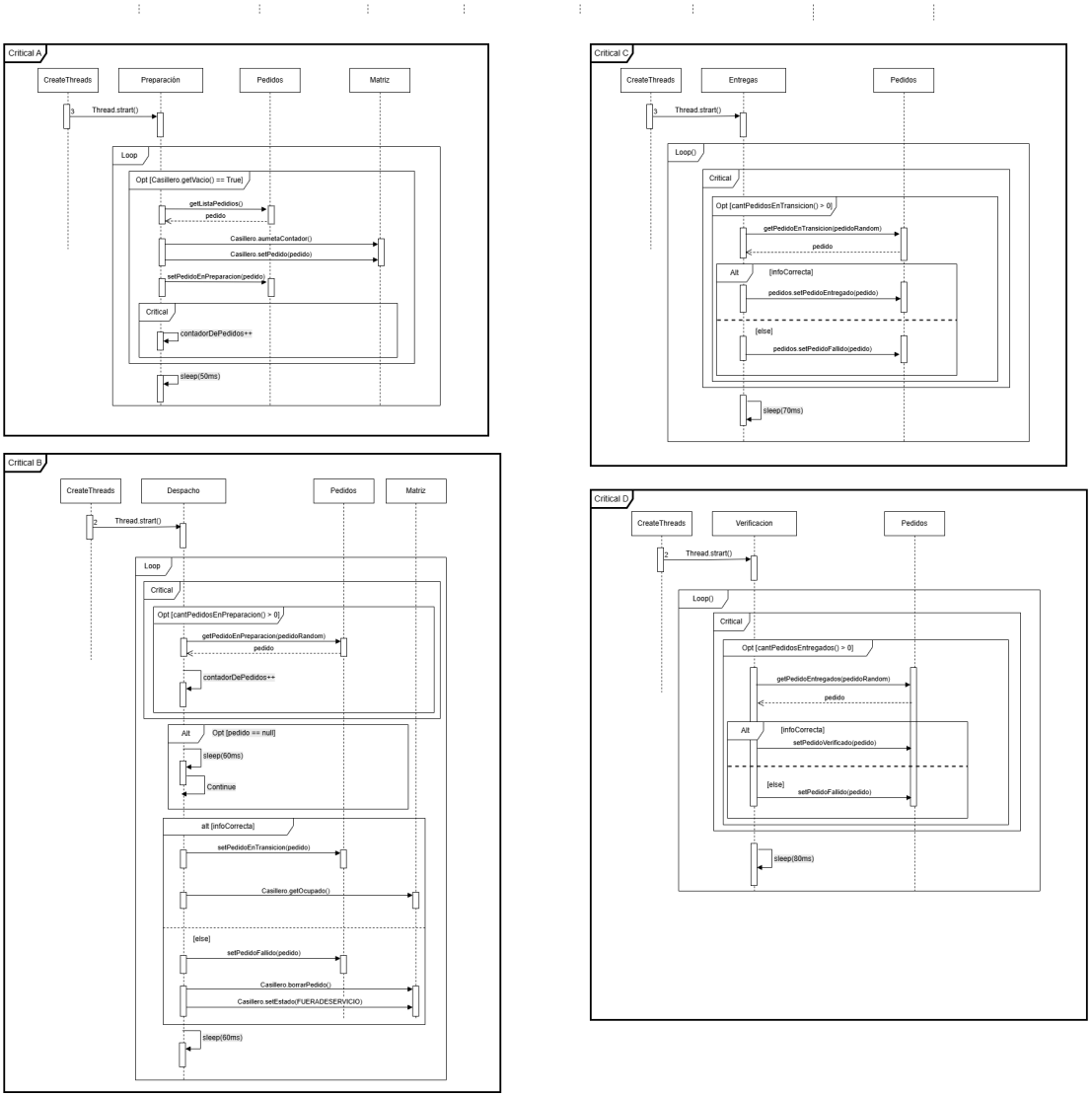


Figura 3: Diagrama de secuencia de procesos.

El diagrama de secuencia puede ser examinado con una mejor calidad en su archivo **drive**

Para explicar el código, comenzamos, con lo que hace el método principal **main()**

```
1   package src;
2   public class Main {
        Run | Debug
3       public static void main(String[] args) {
4
5           long inicio = System.currentTimeMillis();  //inicio del programa
6
7           Matriz matriz = new Matriz(filas:10,columnas:20);
8           Pedidos pedidos = new Pedidos(cantPedidos:500);
9
10          Logger logger = new Logger(matriz, pedidos, inicio);
11          CreateThreads threadLogger = new CreateThreads(cantHilos:1, logger, nameProceso:"Logger");
12
13          Preparacion preparacion = new Preparacion(matriz, pedidos);
14          CreateThreads threadsPreparacion = new CreateThreads(cantHilos:3, preparacion, nameProceso:"Preparacion");
15
16          Despacho despacho = new Despacho(matriz, pedidos);
17          CreateThreads threadsDespacho = new CreateThreads(cantHilos:2, despacho, nameProceso:"Despacho");
18
19          Entregas entregas = new Entregas(pedidos);
20          CreateThreads threadsEntregas = new CreateThreads(cantHilos:3, entregas, nameProceso:"Entregas");
21
22          Verificacion verificacion = new Verificacion(pedidos);
23          CreateThreads threadsVerificacion = new CreateThreads(cantHilos:2, verificacion, nameProceso:"Verificacion");
24      }
25  }
```

Figura 4: Código de clase Main().

Línea 6) La variable “inicio” guarda los milisegundos en tiempo real de la ejecución del programa, que al ser la primera instrucción, su valor es muy pequeño. Para imprimir el tiempo total de ejecución del programa se resta esta variable a otra que llamaremos “fin”.

Línea 8) Crea un objeto Matriz de dimensiones 10x20 destinada a contener objetos de la clase Casillero

Línea 9) El constructor de la clase “Pedidos” que contendrá un conjunto ArrayList de objetos “Pedido” destinados a usarse en las diferentes etapas de la simulación.

Línea 11 y 12) Logger que crea un archivo de estadísticas, donde se escribe cada 200 milisegundos las cantidades de pedidos en preparación, transición, entregados, fallidos y verificados. Al final del archivo se imprime la matriz de casilleros con la cantidad de veces que ha sido ocupado cada lugar, y además, el tiempo total de ejecución. Para ello se lanza un hilo específico instanciando un objeto de la clase CreateThreads.

Línea 14 a 24) Un objeto dedicado a cada etapa del funcionamiento del sistema (preparación, despacho, entregas y verificación) y lanza sus respectivos hilos.

```java
public class CreateThreads {
    private int cantHilos;
    private Runnable proceso;
    private String nameProceso;
    public CreateThreads(int cantHilos, Runnable proceso, String nameProceso){
        this.cantHilos = cantHilos;
        this.proceso = proceso;
        this.nameProceso = nameProceso;
        lanzarHilos();
    }
    private void lanzarHilos(){   //solo se ejecuta cuando se instancia un objeto CreateThreads
        for (int i = 0; i<cantHilos; i++){
            String nameThread = "Thread - "+ i+" - " + nameProceso;
            Thread thread = new Thread(proceso, nameThread);
            thread.start();
        }
    }
}
```

Figura 5: Código de clase CreateThreads()

```java
public class Casillero {
    private EstadoCasillero estado;
    private int contador;
    private Pedido pedido;
    public Casillero() {
        this.estado = EstadoCasillero.VACIO; //se instancia como "vacio"
        this.contador = 0; //se instacia contador en "0"
        this.pedido = null; //pedido asociado nulo
    }
```

Figura 6: Código de clase Casillero()

```java
public enum EstadoCasillero {
    VACIO, OCUPADO, FUERADESERVICIO
}
```

Figura 7: Código de clase enum EstadoCasillero()

Método run() de la clase **Preparación:**

```
11      private Object lockContador = new Object();
12      @Override
13      public void run() {
14          while (contadorDePedidos < pedidos.getCantPedidos()){
15              int randomFila = ThreadLocalRandom.current().nextInt(origin:0, matriz.getFilas());
16              int randomColumna = ThreadLocalRandom.current().nextInt(origin:0, matriz.getColumnas());
17              if (matriz.getMatriz()[randomFila][randomColumna].getVacio()) { //si esta OCUPADO itera de vuelta, si esta vacio lo pasa a ocupado y entra
18                  Pedido pedido = pedidos.getListaPedidos(); //pedido a settear
19                  if (pedido == null) { // controlo que el pedido no sea null, porque si mas de un hilo
20                      // entro en el while y justo termina la lista de Pedidos me va a devolver un null
21                      continue;
22                  }
23                  matriz.getMatriz()[randomFila][randomColumna].aumentaContador(); //contador del casillero ++
24                  matriz.getMatriz()[randomFila][randomColumna].setPedido(pedido); //setteo el pedido al casillero
25                  pedidos.setPedidoEnPreparacion(pedido);//setteo el pedido a la lista de pedidos en preparación
26                  //---inicio SC---//
27                  synchronized (lockContador) {
28                      contadorDePedidos++; //aumento contadorDePedidos para salir del while()
29                  }//---Fin SC---//
30              }
31              try {
32                  Thread.sleep(millis:50);  //cada iteracion debe tener una demora fija
33              } catch (InterruptedException e) {
34                  throw new RuntimeException(e);
35              }
36          }
37          System.out.println("Termino : " + Thread.currentThread().getName());  //solo para ver que corra
38      }
39  }
```

Figura 8: Código del método run() sobre escrito de preparación()

Línea 11) Objeto lockContador que utilizaremos como llave del synchronized para entrar a la sección crítica

Línea 14) Método run() que ejecutarán los 3 hilos correspondientes a esta clase/etapa. Mientras el contador de pedidos que ya fueron preparados sea menor que la cantidad de pedidos por preparar (500), los hilos volverán a este punto y le buscará a cada pedido un casillero aleatorio (lugar en la matriz) siempre y cuando esté vacío. Al ocupar un lugar, el estado del casillero pasará de VACÍO a OCUPADO.

Línea 17) Se pregunta si el estado de casillero random es VACÍO (el método sincronizado getVacio(), de la clase Casillero, devuelve true si lo está, y false si está ocupado)

Línea 19) Se hace un control de que el pedido obtenido de la listaPedidos no sea nulo porque se puede dar de que cuando quede un solo pedido disponible en la lista haya más de un hilo esperando la llave de la sección crítica, lo que provocaría que la lista le devuelva un pedido nulo* al/los último/s hilo/s que obtengan la llave, corrompiendo la cantidad de pedidos en las listas subsiguientes.

*Todas listas tienen control al solicitar un pedido cuando la lista está vacía, retorna un null.

Línea 25) Se va agregando cada pedido a un ArrayList de pedidos en preparación

Línea 27) Sección crítica para el contador del While, para no corromper la cantidad de pedidos que se están preparando.

Línea 31 a 35) Tiempo de preparación fijo de 50 milisegundos (cada iteración del While)

Línea 37) A medida que se van desocupando los hilos, y no quedan pedidos por preparar, se imprime un mensaje indicando qué hilo terminó y de qué proceso, en este caso el proceso “preparación”.

```java
    private Object lockCasillero = new Object();
    public boolean getVacio() { //metodo para preguntar si esta vacio, y si lo esta cambia de estado
        synchronized (lockCasillero){ //
            if (estado == EstadoCasillero.VACIO){
                setEstado(EstadoCasillero.OCUPADO);
                return true;
            }
            return false;
        }
    }
    public void getOcupado() { //metodo para preguntar si esta ocupado, y si lo esta cambia de estado y borra el pedido asociado
        synchronized (lockCasillero){
            if (estado == EstadoCasillero.OCUPADO){
                setEstado(EstadoCasillero.VACIO);
                borrarPedido();
            }
        }
    }
```

Método run() de la clase **Despacho:**

```java
    private Object lockPedido = new Object();
    @Override
    public void run() {
        while (contadorDePedidos < pedidos.getCantPedidos()) {
            Pedido pedido = null;
            //---inicio SC---//
            synchronized (lockPedido) { ///cambiar
                if (pedidos.cantPedidosEnPreparacion() > 0) { //si no hay pedidos en preparacion pasa a la siguiente itereacion
                    //no hace falta controlar si el pedido es null en esta instancia porque este es el unico proceso
                    //que saca pedidos de la lista anterior
                    int pedidoRandom = ThreadLocalRandom.current().nextInt(origin:0, pedidos.cantPedidosEnPreparacion());
                    pedido = pedidos.getPedidoEnPreparacion(pedidoRandom); //tomo el pedido aleatorio de la lista de pedidos en preparacion, y se borra de la lista
                    contadorDePedidos++; //aumento contadorDePedidos para salir del while()
                }
            }
            //---Fin SC---//
            if (pedido == null) { // controlo que el pedido no sea null, si es null duermo y salgo
                try {
                    Thread.sleep(millis:60);  //cada iteracion debe tener una demora fija
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
                continue;
            }
            boolean infoCorrecta = ThreadLocalRandom.current().nextInt(origin:0, bound:100) < 85;
            if (infoCorrecta) {
                pedidos.setPedidoEnTransicion(pedido); //seteo el pedido a PedidoEnTransicion
                for (Casillero[] casilleros : matriz.getMatriz()) { //forEach anidado para recorrer la matriz y poder manuipular los casilleros
                    for (Casillero casillero : casilleros) {
                        if (casillero.getPedido() == pedido) { //busco el pedido en la matriz
                            casillero.getOcupado();
                        }
                    }
                }
            } else {
                pedido.setEstado(EstadoPedido.FALLIDO);
                pedidos.setPedidoFallido(pedido); //seteo el pedido a PedidoFallido
                for (Casillero[] casilleros : matriz.getMatriz()) { //forEach anidado para recorrer la matriz y poder manuipular los casilleros
                    for (Casillero casillero : casilleros) {
                        if (casillero.getPedido() == pedido) { //busco el pedido en la matriz
                            casillero.borrarPedido(); //borro el pedido asociado al casillero
                            casillero.setEstado(EstadoCasillero.FUERADESERVICIO); //anulo el casillero
                        }
                    }
                }
            }
            try {
                Thread.sleep(millis:60);  //cada iteracion debe tener una demora fija
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
        System.out.println("Termino : " + Thread.currentThread().getName()); //solo para ver que corra
    }
```

Figura 9: Código del método run() sobre escrito de Despacho()

Línea 14) Se ejecuta el While mientras haya pedidos por despachar, debe llegar a 500

Línea 16) Comienza una sección crítica, evitando la posibilidad de que más de un hilo tome el mismo pedido para despachar, sumando una unidad a la variable contadorDePedidos. Si no se logra obtener algún pedido de la lista de preparación, no hay nada que despachar, así que en ese caso devuelve la llave, espera 60 milisegundos (tiempo fijo de la etapa de despacho) y comienza de nuevo el While.

Si hay pedidos en preparación, se busca en esa lista un pedido aleatorio estableciendo una probabilidad de 85% de que la información sea correcta. De ser el caso, se agrega el pedido a un ArrayList de pedidos en transición, se lo busca en la matriz y lo borra, cambiando el estado de ese casillero de OCUPADO a VACÍO. Si la información no es correcta, se selecciona el estado del pedido como FALLIDO, lo agrega a la lista de pedidos fallidos, lo busca en la matriz y lo borra, así el estado de ese casillero pasa de OCUPADO a FUERA DE SERVICIO. Finalmente el hilo espera los 60 milisegundos fijos del proceso y pasa a la siguiente iteración del While, hasta que hayan pasado los 500 pedidos. En ese punto se imprime un mensaje del hilo que terminó. La sección crítica del bloque comprendido entre línea 17-25, evita que más de un hilo no extraiga elementos de un mismo elemento del casillero, provocando una corrupción de los datos, lo que causaría un fallo en el estado real de cada casillero y cada pedido.

Método run() de la clase **Entregas:**

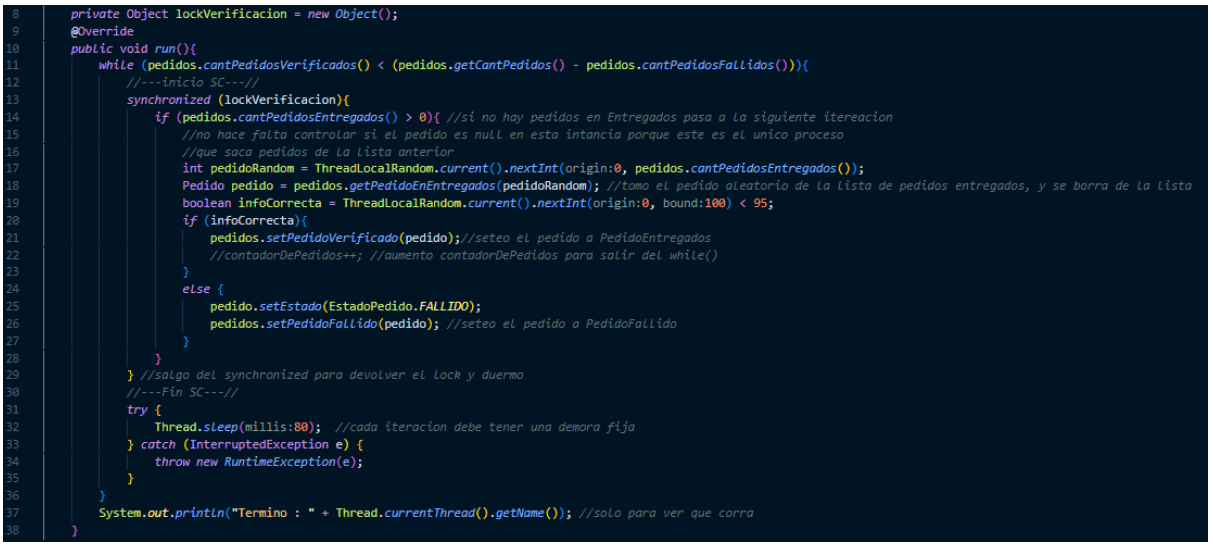
```
9   private Object lockEntregas = new Object();
10  @Override
11  public void run() {
12      while ((pedidos.cantPedidosVerificados() + pedidos.cantPedidosFallidos()) < pedidos.getCantPedidos()){
13          //---inicio SC---//
14          synchronized (lockEntregas){
15              if (pedidos.cantPedidosEnTransicion() > 0){ //si no hay pedidos en transicion pasa a la siguiente itereacion
16                  //no hace falta controlar si el pedido es null en esta intancia porque este es el unico proceso
17                  //que saca pedidos de la lista anterior
18                  int pedidoRandom = ThreadLocalRandom.current().nextInt(origin:0, pedidos.cantPedidosEnTransicion());
19                  Pedido pedido = pedidos.getPedidoEnTransicion(pedidoRandom); //tomo el pedido aleatorio de la lista de pedidos en transicion, y se borra de la lista
20                  boolean infoCorrecta = ThreadLocalRandom.current().nextInt(origin:0, bound:100) < 90;
21                  if (infoCorrecta){
22                      pedidos.setPedidoEntregado(pedido);//seteo el pedido a PedidoEntregados
23                  }
24                  else {
25                      pedido.setEstado(EstadoPedido.FALLIDO);
26                      pedidos.setPedidoFallido(pedido); //seteo el pedido a PedidoFallido
27                  }
28              }
29          } //salgo del synchronized para devolver el lock y duermo
30          //---Fin SC---//
31          try {
32              Thread.sleep(millis:70);  //cada iteracion debe tener una demora fija
33          } catch (InterruptedException e) {
34              throw new RuntimeException(e);
35          }
36      }
37      System.out.println("Termino : " + Thread.currentThread().getName()); //solo para ver que corra
38  }
```

Figura 10: Código del método run() sobre escrito de Entregas()

Línea 12) Mientras la suma de la cantidad de pedidos fallidos y verificados sea menos que la cantidad total de pedidos a procesar (500), se ejecuta el While.

Línea 14) Nueva sección crítica. Si no hay pedidos en transición, no hay nada que entregar, así que en ese caso espera 70 milisegundos (tiempo fijo de la etapa de Entrega) y comienza de nuevo el While. Si hay pedidos en transición, busca uno aleatoriamente en esa lista, con una probabilidad del 90% de que la información sea correcta. Si es correcta se agrega el pedido a la lista de entregados, y en caso contrario, su estado pasa a ser FALLIDO y lo suma a esta lista. Al salir de la sección crítica, espera los 70 milisegundos y continúa con la siguiente iteración siempre que hayan pedidos por verificar. Debido a que los recursos del objeto pedidos son compartidos por los hilos, es necesaria la SC para no corromper el estado de los pedidos y no agregar el mismo elemento a diferentes listas o duplicado en una misma lista (debido a la probabilidad).

Método run() de la clase **Verificación:**



```
8   private Object lockVerificacion = new Object();
9   @Override
10  public void run(){
11      while (pedidos.cantPedidosVerificados() < (pedidos.getCantPedidos() - pedidos.cantPedidosFallidos())){
12          //---inicio SC---//
13          synchronized (lockVerificacion){
14              if (pedidos.cantPedidosEntregados() > 0){ //si no hay pedidos en Entregados pasa a la siguiente itereacion
15                  //no hace falta controlar si el pedido es null en esta intancia porque este es el unico proceso
16                  //que saca pedidos de la lista anterior
17                  int pedidoRandom = ThreadLocalRandom.current().nextInt(origin:0, pedidos.cantPedidosEntregados());
18                  Pedido pedido = pedidos.getPedidoEnEntregados(pedidoRandom); //tomo el pedido aleatorio de la lista de pedidos entregados, y se borra de la lista
19                  boolean infoCorrecta = ThreadLocalRandom.current().nextInt(origin:0, bound:100) < 95;
20                  if (infoCorrecta){
21                      pedidos.setPedidoVerificado(pedido);//seteo el pedido a PedidoEntregados
22                      //contadorDePedidos++; //aumento contadorDePedidos para salir del while()
23                  }
24                  else {
25                      pedido.setEstado(EstadoPedido.FALLIDO);
26                      pedidos.setPedidoFallido(pedido); //seteo el pedido a PedidoFallido
27                  }
28              }
29          } //salgo del synchronized para devolver el lock y duermo
30          //---Fin SC---//
31          try {
32              Thread.sleep(millis:80);  //cada iteracion debe tener una demora fija
33          } catch (InterruptedException e) {
34              throw new RuntimeException(e);
35          }
36      }
37      System.out.println("Termino : " + Thread.currentThread().getName()); //solo para ver que corra
38  }
```

Figura 11: Código del método run() sobre escrito de Verificación()

Línea 11) Se establece como criterio del ciclo, que mientras la cantidad de pedidos ya verificados sea menor a la cantidad restante (sin incluir los fallidos) se ejecuta el While.

Línea 13) Se abre una sección crítica de la misma manera que en la clase Entregas. Si hay pedidos entregados, se tiene una probabilidad del 95% de que sea información correcta, en cuyo caso se agrega el pedido a la lista de verificados. En caso contrario se indica como FALLIDO y lo suma a esta lista. Posteriormente realiza la espera fija de 80 milisegundos correspondiente a la etapa de verificación. Observar que si no hay pedidos entregados, el hilo solo espera este tiempo para seguir iterando.

Clase **Matriz:**

```java
package src;
public class Matriz {
    private final int filas;
    private final int columnas;
    private Casillero[][] matriz;
    public Matriz(int filas, int columnas){
        this.filas = filas;
        this.columnas = columnas;
        this.matriz = new Casillero[filas][columnas];
        generarMatriz();
    }
    private void generarMatriz(){
        for (int i = 0; i < filas; i++) {
            for (int j = 0; j < columnas; j++) {
                matriz[i][j] = new Casillero();  //en cada posicion de la matriz Casillero[][] instancio un Casillero
            }
        }
    }
    private Object lockMatriz = new Object();
    public Casillero[][] getMatriz() {
        synchronized (lockMatriz){ //se protege porque se puede devolver una matriz desactualizada
            return matriz;
        }
    }
    public int getColumnas() {
        return columnas;
    }
    public int getFilas() {
        return filas;
    }
    public String contadorDeUso() {
        String show="";
        for (int i = 0; i < filas; i++) {
            show += "\n";
            for (int j = 0; j < columnas; j++) {
                show += " "+ matriz[i][j].getContador();
            }
        }
        return show;
    }
}
```

Figura 12: Código de la clase Matriz()

En esta clase tenemos el método getMatriz(), que contiene una SC que devuelve la matriz. Es importante esta sección para asegurar que no se devuelva una matriz que está siendo modificada en ese momento, ya que hay más de un proceso manipulando casilleros.

Clase **Pedidos:**

```java
    public void setPedidoEnPreparacion(Pedido enPreparacion) {
        synchronized (lockListaEnPreparacion){
            this.listaEnPreparacion.add(enPreparacion);
        }
    }
    public void setPedidoEnTransicion(Pedido enTransicion) {
        synchronized (lockListaEnTransicion) {
            this.listaEnTransicion.add(enTransicion);
        }
    }
    public void setPedidoEntregado(Pedido entregado) {
        synchronized (lockListaEntregados){
            this.listaEntregados.add(entregado);
        }
    }
    public void setPedidoVerificado(Pedido entregado) {
        synchronized (lockListaVerifcados){
            this.listaVerifcados.add(entregado);
        }
    }
    public void setPedidoFallido(Pedido fallido) {
        synchronized (lockListaFallidos){
            this.listaFallidos.add(fallido);
        }
    }
    ///getters listas->pedido
    public Pedido getListaPedidos() {
        synchronized (lockListaPedidos){
            if (!(listaPedidos.isEmpty())){ //me aseguro de que la lista no este vacia
                return listaPedidos.remove(index:0); //saco el primer pedido de la lista y se borran
            }
        }
        return null;
    }
    public Pedido getPedidoEnPreparacion(int index) {
        synchronized (lockListaEnPreparacion){
            if (!(listaEnPreparacion.isEmpty())){ //me aseguro de que la lista no este vacia
                return listaEnPreparacion.remove(index); //si se saca se borra
            }
        }
        return null;
    }
    public Pedido getPedidoEnTransicion(int index) {
        synchronized (lockListaEnTransicion){
            if (!(listaEnTransicion.isEmpty())){ //me aseguro de que la lista no este vacia
                return listaEnTransicion.remove(index); //si se saca se borra
            }
        }
        return null;
    }
```

```java
    public Pedido getPedidoEnEntregados(int index) {
        synchronized (lockListaEntregados){
            if (!(listaEntregados.isEmpty())){ //me aseguro de que la lista no este vacia
                return listaEntregados.remove(index); //si se saca se borra
            }
        }
        return null;
    }
    //no hace falta getPedidoEnVerificados()

    ///getters de cantidades de las listas
    public int cantPedidosEnPreparacion(){
        synchronized (lockListaEnPreparacion){ //se protege porque se puede modificar la cantidad en paralero y dar un size erroneo
            return listaEnPreparacion.size();
        }
    }
    public int cantPedidosEnTransicion(){
        synchronized (lockListaEnTransicion){ //se protege porque se puede modificar la cantidad en paralero y dar un size erroneo
            return listaEnTransicion.size();
        }

    }
    public int cantPedidosEntregados(){
        synchronized (lockListaEntregados){ //se protege porque se puede modificar la cantidad en paralero y dar un size erroneo
            return listaEntregados.size();
        }
    }
    public int cantPedidosVerificados(){
        synchronized (lockListaVerifcados){ //se protege porque se puede modificar la cantidad en paralero y dar un size erroneo
            return listaVerifcados.size();
        }
    }
    public int cantPedidosFallidos(){
        synchronized (lockListaFallidos){ //se protege porque se puede modificar la cantidad en paralero y dar un size erroneo
            return listaFallidos.size();
        }
    }
}

```

Figura 13 y 14: métodos getters y setters de la clase Pedidos()

Los Locks de esta clase son también muy importantes para garantizar, otra vez, que los hilos de diferentes procesos NO accedan a las listas de pedidos al mismo tiempo. Podría suceder, por ejemplo, que los While presentes a los largo del programa hagan los cálculos con datos que no son reales, porque existiría la posibilidad de que las variables que necesita se estén modificando al mismo tiempo que las lee. Otro ejemplo sería que un hilo de la clase Despacho esté modificando la lista de pedidos en transición al mismo tiempo que es leída por otro hilo de la clase Entregas, por lo que podría generar conflictos en el tamaño de la lista.

Salida de una **ejecución del programa:**

```
Termino : Thread - 2 - Preparacion
Termino : Thread - 1 - Preparacion
Termino : Thread - 0 - Preparacion
Termino : Thread - 0 - Despacho
Termino : Thread - 1 - Despacho
Termino : Thread - 0 - Entregas
Termino : Thread - 1 - Entregas
Termino : Thread - 2 - Entregas
Termino : Thread - 0 - Verificacion
Termino : Thread - 1 - Verificacion
Termino : Thread - 0 - Logger
```

Figura 15: Mensaje impreso de la finalización de los hilos()

Podemos observar que, por más que los hilos trabajen de manera concurrente, tiene sentido que vayan terminando aproximadamente en ese orden, ya que los hilos de un proceso dependen del recurso que les brindan los hilos del proceso anterior. En esta ejecución, el último hilo en terminar es el del Logger, pero no necesariamente es así siempre, dependerá de la espera final de cada uno del resto de hilos, esos milisegundos de diferencia hacen que la impresión en pantalla de la finalización de los hilos se desordene.

Otra **Ejecución:**

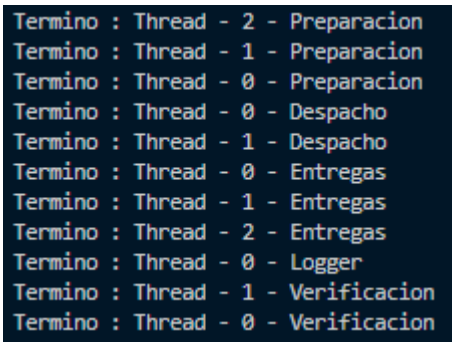


Figura 16: Mensaje impreso de la finalización de los hilos()

Por lo mencionado anteriormente, tiene sentido que los hilos de Verificación terminen últimos, ya que son los que mayor espera tienen.

Logger:

```java
package src;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
public class Logger implements Runnable{
    private static final String LOG_FILE = "Registro de Pedidos.log";
    private Matriz matriz;
    private Pedidos pedidos;
    private long inicio;
    public Logger(Matriz matriz, Pedidos pedidos, long inicio){
        this.matriz = matriz;
        this.pedidos = pedidos;
        this.inicio = inicio;
    }
    public void escribirLog(String mensaje, boolean borrar) { //si borrar es true borra lo que habia antes en el archivo .log
        try (FileWriter fw = new FileWriter(LOG_FILE, !borrar);
             PrintWriter pw = new PrintWriter(fw)) {
            pw.println(mensaje);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    @Override
    public void run() {
        escribirLog(mensaje:"----Inicio de Registro de Pedidos----", borrar:true);
        while ((pedidos.cantPedidosVerificados() + pedidos.cantPedidosFallidos()) < pedidos.getCantPedidos()){
            String log = "-----------------------------------";
            log += "\n Cantidad de Pedidos en Preparacion: " + pedidos.cantPedidosEnPreparacion();
            log += "\n Cantidad de Pedidos en Transicion: " + pedidos.cantPedidosEnTransicion();
            log += "\n Cantidad de Pedidos por Entregar: " + pedidos.cantPedidosEntregados();
            log += "\n Cantidad de Pedidos Fallidos: " + pedidos.cantPedidosFallidos();
            log += "\n Cantidad de Pedidos Verificados: " + pedidos.cantPedidosVerificados();
            escribirLog(log, borrar:false);
            try {
                Thread.sleep(millis:200);  /// duermo 200 ms
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
        //Tiene que haber una última escritura fuera del while() para que figure como terminaron las variables
        String log = "-----------------------------------";
        log += "\n Cantidad de Pedidos en Preparacion: " + pedidos.cantPedidosEnPreparacion();
        log += "\n Cantidad de Pedidos en Transicion: " + pedidos.cantPedidosEnTransicion();
        log += "\n Cantidad de Pedidos por Entregados: " + pedidos.cantPedidosEntregados();
        log += "\n Cantidad de Pedidos Fallidos: " + pedidos.cantPedidosFallidos();
        log += "\n Cantidad de Pedidos Verificados: " + pedidos.cantPedidosVerificados();
        log += "\n-----------------------------------";
        escribirLog(log, borrar:false);
        escribirLog(matriz.contadorDeUso(), borrar:false); //imprimo contadores de los casilleros
        long fin = System.currentTimeMillis();
        escribirLog( "\n------Tiempo de ejecución: " + (fin - inicio) + " ms------", borrar:false);
        System.out.println("Termino : " + Thread.currentThread().getName()); //solo para ver que corra
    }
}
```

Figura 17: Implementación de la clase Logger()

Línea 6) Crea un archivo ".log" con el nombre "Registro de Pedidos", que contiene elementos de la clase String.

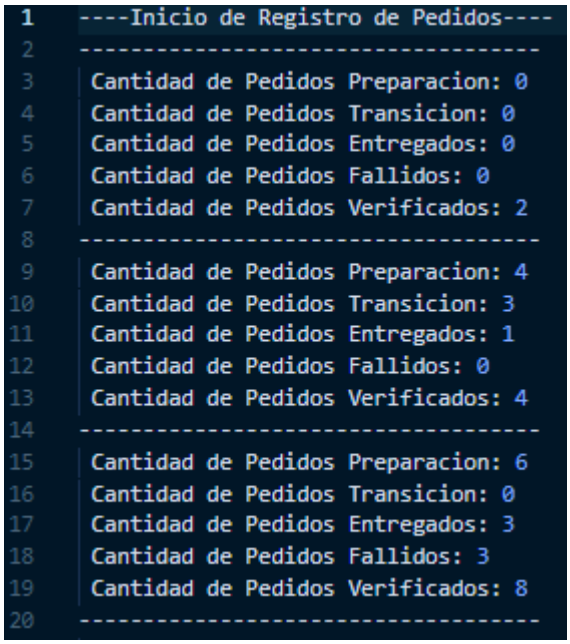
Línea 24 a 53) Método run(). En primer lugar borra el archivo de la ejecución anterior si es que existe. Luego escribe cada 200 milisegundos (fijos) los valores actuales de las cantidades de pedidos en:

- preparación
- transición
- entregados
- fallidos
- verificados

Y lo hace mientras la cantidad de pedidos totales (500) no se haya completado aún. La espera fija mencionada se hace desde dentro de este While. Finalmente, se vuelven a imprimir para ver cómo quedaron dichas cantidades, seguido de la matriz, con la cantidad de veces que se usó cada casillero y

por debajo el tiempo total de ejecución restando el tiempo final y el inicial como se comentó al principio en la primera instrucción del método main().

Registro estadístico generado por el **Logger:**



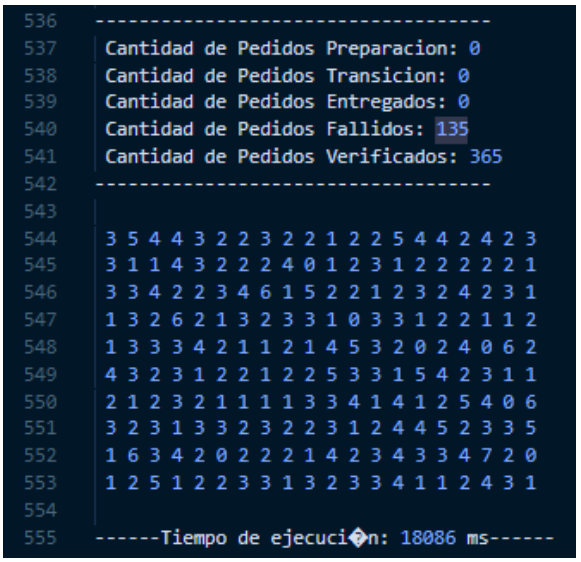


Figura 18 y 19: resultados finales impresos por el logger()

Conclusión

Al finalizar la ejecución del programa se observó un correcto funcionamiento del mismo, sin datos que quedarán pendientes en listas que no sean de pedidos fallidos o verificados, ni presenciando pérdida de pedidos.

Realizándose una serie de ejecuciones de prueba en busca de fallas, se observa que la simulación mantiene una duración media de 15.924 milisegundos con una desviación estándar de 476,56 milisegundos. La cantidad de pedidos verificados tendrán una media 371 elementos con desviación estándar de 6,80. La cantidad de pedidos fallidos posee una media de 129 con desviación estándar de 6,80.

Se determina que los requisitos planteados en las consignas iniciales fueron cumplidos satisfactoriamente y el desarrollo del programa queda finalizado.