Universidad de Alcalá

Paradigmas de Programación - E1-

Grado en Ingeniería en Sistemas de Información (GISI)
Curso 2024/2025 - Convocatoria Ordinaria
Paradigmas de Programación
Tutores:

Sergio Caro Álvaro
David de Fitero Domínguez
Antonio García Cabot
Eva García López

03226056F - Serrano Diaz David
03220316V - Palomo Cuenca Rodrigo

Universidad de Alcalá
Madrid, Spain

david.serranod@edu.uah.es
rodrigo.palomo@edu.uah.es

ÍNDICE

1. Introducción

Este proyecto busca modelar de manera realista los procesos internos de la fábrica, como la producción, horneado y empaquetado de galletas, también integrar mecanismos avanzados de sincronización y comunicación entre los distintos elementos del sistema, respetando las limitaciones y particularidades de cada uno.

El programa se organiza en dos grandes apartados:

1. **Programación Concurrente:** Se desarrolla una simulación inicial en la que los procesos de producción (reposteros), horneado (hornos) y empaquetado (empaquetadores) se gestionan mediante hilos. Cada entidad tiene un comportamiento autónomo que se coordina mediante herramientas de sincronización para evitar conflictos o ineficiencias en el uso de recursos como la cafetera, los hornos y el almacén. Los eventos del sistema se registran en un log y se muestran gráficamente para facilitar su análisis.
2. **Programación Distribuida:** Sobre la base del sistema concurrente, se añade un módulo de visualización remota. Esto incluye un servidor que monitorea y expone el estado de la fábrica, y un cliente que permite al usuario observar y controlar aspectos clave como la cantidad de galletas producidas, horneadas, empaquetadas y almacenadas, así como gestionar el comportamiento de los reposteros.

Ambos mecanismos deben funcionar de una forma óptima, realizando los procedimientos que sean necesarios para ello.

2. Análisis de Alto Nivel

2.1. Descripción General del Problema

El sistema debe reflejar la interacción entre los distintos actores (reposteros, hornos y empaquetadores) bajo las siguientes condiciones:

- **Sincronización y gestión de recursos compartidos** como hornos, empaquetadores y el almacén.
- **Restricciones operativas** como tiempos de espera, descansos, capacidad máxima de los hornos y almacén.
- **Gestión eficiente de recursos** para evitar desperdicios o bloqueos, manteniendo el flujo continuo de producción.
  Adicionalmente, el sistema debe incluir un módulo distribuido que permita monitorizar y controlar remotamente la fábrica, asegurando una actualización en tiempo real de los estados y eventos relevantes.

2.2. Objetivos del Sistema

- **Simulación Realista:** Modelar los procesos internos de la fábrica mediante un sistema de hilos concurrentes, respetando las limitaciones y tiempos operativos de cada actor.
- **Sincronización y Comunicación:** Implementar mecanismos eficientes para la sincronización de recursos y comunicación entre hilos.
- **Visualización y Control Remotos:** Desarrollar un módulo distribuido que permita al usuario observar el estado de la fábrica y realizar acciones de control a través de una interfaz gráfica.
- **Registro y Análisis de Eventos:** Generar un log de eventos para rastrear las operaciones y analizar el comportamiento del sistema.
- **Evitar Bloqueos:** Garantizar que el sistema funcione sin interbloqueos o ineficiencias que puedan interrumpir la producción.

2.3. Requisitos Funcionales

RF001	La fábrica debe contar con cinco reposteros que produzcan tandas de galletas con cantidades generadas aleatoriamente entre 37 y 45 unidades.
RF002	Los reposteros deben llenar los hornos en orden y esperar si todos los hornos están ocupados.
RF003	Cada repostero debe realizar descansos para preparar café después de producir entre 3 y 5 tandas de galletas.
RF004	El sistema debe permitir que solo un repostero utilice la cafetera a la vez, tardando 2 segundos en preparar café y descansando entre 3 y 6 segundos antes de reanudar la producción.
RF005	Los hornos deben hornear galletas únicamente cuando estén completamente llenos (200 unidades) y deben tardar 8 segundos en completar el horneado.
RF006	Los empaquetadores deben recoger galletas en lotes de 20
RF007	Una vez acumulen 100 galletas(5 tandas) las trasladan luego al almacén.
RF008	El almacén debe tener una capacidad máxima de 1000 galletas. Si está lleno, los empaquetadores no podrán depositar más galletas hasta que se vacíe parcialmente.
RF009	Los usuarios deben poder consumir 100 galletas del almacén por cada pulsación de un botón en la interfaz gráfica (“Comer”).
RF010	Cada empaquetador debe estar asignado a un horno específico para recoger galletas.
RF011	Los tiempos de producción, horneado, empaquetado y descansos deben ser generados aleatoriamente dentro de los rangos establecidos.

RF012	El sistema debe registrar todos los eventos en un archivo de log con marcas de tiempo detalladas.
RF013	Debe haber una interfaz gráfica que muestre en tiempo real el estado de la producción, hornos, empaquetado y almacén.
RF014	La interfaz gráfica debe permitir la visualización y control remoto del estado de la fábrica a través de un cliente externo.
RF015	El cliente remoto debe actualizar automáticamente la información cada segundo.
RF016	El sistema debe permitir pausar y reanudar individualmente a los reposteros desde el cliente remoto.
RF017	Si un repostero genera más galletas de las necesarias para llenar un horno, las galletas sobrantes deben ser registradas como desperdicio en el log.

2.4. Requisitos No Funcionales

RNF000	El programa deberá estar escrito en Java.
RNF001	El sistema debe garantizar un rendimiento eficiente, evitando tiempos muertos prolongados entre las operaciones de producción, horneado y empaquetado.
RNF002	El diseño debe ser robusto, asegurando que no se produzcan bloqueos o interbloqueos entre los hilos del sistema.
RNF003	La actualización de datos en la interfaz gráfica, tanto local como remota, debe realizarse en menos de un segundo para garantizar la visualización en tiempo real.
RNF004	La solución debe ser escalable, permitiendo la posibilidad de aumentar el número de reposteros, hornos o empaquetadores sin cambios significativos en la arquitectura.
RNF005	El sistema debe desarrollarse siguiendo buenas prácticas de programación, incluyendo nombres descriptivos para las clases, métodos y atributos, así como encapsulación de atributos privados con métodos getter y setter.
RNF006	El log de eventos debe ser fácilmente legible y organizado, mostrando cada acción con un formato uniforme, incluyendo la fecha, hora, minuto y segundo del evento.
RNF007	La interfaz gráfica debe ser intuitiva, mostrando toda la información relevante de manera clara, incluyendo indicadores visuales del estado de los actores (ocupado, esperando, pausado, etc.).

RNF008	El cliente remoto debe ser compatible con múltiples plataformas y redes estándar, garantizando la conexión estable y segura con el servidor de la fábrica.
RNF009	La solución debe ser desarrollada en Java, utilizando el entorno de desarrollo NetBeans, conforme a las restricciones del proyecto.
RNF010	Todo el código debe estar documentado para facilitar su comprensión, mantenimiento y extensibilidad futura.
RNF011	El sistema debe minimizar el uso innecesario de recursos, como memoria y procesamiento, optimizando la ejecución concurrente y distribuida.
RNF012	El código entregado debe ser original y no incluir fragmentos reutilizados de prácticas previas.
RNF013	La solución debe estar diseñada para que su comportamiento sea reproducible y verificable durante la evaluación.

3. Diseño General del Sistema

3.1. Arquitectura del Sistema Concurrente

La arquitectura del sistema está diseñada para simular el funcionamiento de un sistema concurrente en el que múltiples reposteros producen galletas y las depositan en hornos, coordinándose a través de una cafetería y un logger centralizado. Este diseño emula un entorno en el que la sincronización, la gestión de recursos compartidos y la concurrencia son fundamentales para un desempeño eficiente. A continuación, se describen los componentes principales de la arquitectura:

Componentes Principales

1. **Repostero**
    - **Descripción:** Cada repostero es representado por un hilo (`Thread`) que alterna entre tres tareas principales: producir galletas, depositarlas en hornos y descansar.
    - **Responsabilidades:**
        - Simular la producción de galletas en tandas.
        - Coordinar con los hornos para depositar las galletas producidas, verificando disponibilidad.
        - Gestionar estados de pausa y descanso mediante sincronización (`lock`).
        - Registrar eventos relevantes en el logger.
    - **Estados del Repostero:**
        - Produciendo: Cuando está generando galletas.
        - Depositando: Cuando intenta colocar galletas en un horno.
        - Descansando: Simula un descanso utilizando la cafetería.

        - Esperando: Cuando todos los hornos están llenos.
        - Parado: Cuando se pausa manualmente el hilo.
    - **Concurrencia:** El repostero utiliza sincronización explícita para manejar la pausa y evitar colisiones al acceder a recursos compartidos, como hornos.
2. **Horno**
    - **Descripción:** Los hornos son recursos compartidos donde los reposteros depositan las galletas producidas.
    - **Responsabilidades:**
        - Almacenar galletas hasta un límite definido.
        - Verificar si están llenos o en proceso de horneado.
        - Descartar galletas que excedan su capacidad.
    - **Concurrencia:** Los hornos deben ser accedidos de manera controlada para evitar inconsistencias cuando varios reposteros intentan utilizarlos simultáneamente.
3. **Cafetería**
    - **Descripción:** Representa un recurso que los reposteros utilizan para simular descansos. La cafetería asegura que solo se registre el uso y que los descansos sean controlados temporalmente.
    - **Responsabilidades:**
        - Proveer un lugar para que los reposteros simulen descansos.
        - Registrar eventos de entrada y salida al descanso mediante el logger.
    - **Concurrencia:** Manejada de forma que varios reposteros puedan descansar sin colisiones, pero sin compartir recursos físicos.
4. **Logger**
    - **Descripción:** Es un componente central para el registro de eventos y errores en el sistema, permitiendo monitorizar el estado de los procesos.
    - **Responsabilidades:**
        - Almacenar mensajes y eventos generados por los reposteros, hornos y cafetería.
        - Permitir el seguimiento del sistema concurrente en tiempo real o mediante logs históricos.
    - **Concurrencia:** Es accedido por todos los reposteros y componentes del sistema, por lo que debe garantizar consistencia a través de mecanismos de sincronización.
5. **Utilidades**
    - **Descripción:** Proveen funcionalidades auxiliares como generación de números aleatorios y otros métodos comunes utilizados en el sistema.
    - **Concurrencia:** Funciones autónomas sin interacción compartida, no requieren sincronización explícita.

Patrones de Diseño Implementados

1. **Productor-Consumidor:**
    - Implementado en la interacción entre los reposteros y los hornos. Los reposteros producen galletas y los hornos actúan como consumidores al recibirlas.
2. **Monitor (Objeto de Bloqueo):**
    - Utilizado para gestionar el estado de pausa de los reposteros. El objeto `lock` asegura que solo los hilos en pausa queden bloqueados hasta ser notificados.
3. **Locks para Sincronización de Eventos**
    - El sistema utiliza mecanismos de bloqueo (`locks`) para garantizar que los eventos concurrentes se manejen de manera controlada y sin interferencias.
    - **Locks en el Repostero:**
        - Los reposteros usan locks para controlar su actividad y coordinar el cliente con el servidor.
    - **Locks en Hornos y Recursos Compartidos:**
        - Cada horno utiliza un `lock` para controlar el acceso cuando múltiples reposteros intentan introducir galletas simultáneamente. Esto asegura consistencia en los estados de llenado y horneado.
4. **Recurso Compartido (Hornos y Cafetería):**
    - Hornos y la cafetería son recursos compartidos que los reposteros utilizan bajo condiciones de sincronización para evitar conflictos.

Sincronización y Concurrencia

1. **Sincronización del Repostero:**
    - **Pausa y Reanudación:** Cada repostero verifica periódicamente si debe pausar su ejecución mediante el método `checkPausa()`. Esto asegura que se pueda detener temporalmente su actividad sin interrumpir el sistema completo. Se realiza con locks.
    - **Acceso a Hornos:** El acceso al método `introducirGalletas` está sincronizado para evitar colisiones cuando múltiples reposteros intentan depositar galletas simultáneamente.
2. **Gestión de Recursos Compartidos:**
    - **Hornos:** Se verifica la disponibilidad de los hornos mediante el método `todosLlenos()` antes de depositar galletas. Los hornos controlan internamente la cantidad de galletas que pueden almacenar.
    - **Cafetería:** Los descansos son independientes y no bloquean a otros reposteros.
3. **Eventos Concurrentes:**
    - Los reposteros funcionan de manera independiente y se sincronizan solo cuando acceden a recursos compartidos (hornos o logger) y al usar la función `checkPausa()`. Esto minimiza los bloqueos y permite escalabilidad.

Esta arquitectura garantiza que el sistema pueda manejar múltiples reposteros de forma eficiente y robusta, maximizando el aprovechamiento de recursos y permitiendo el monitoreo centralizado del estado del sistema.

3.2. Herramientas de Sincronización y Comunicación Utilizadas

En este sistema concurrente, se han implementado varias herramientas de sincronización y comunicación para garantizar la correcta interacción entre los componentes y evitar conflictos en los recursos compartidos:

Locks

- Descripción: Se utilizan para sincronizar el acceso a recursos compartidos como hornos, cafetería y logger.
- Uso principal:
    - Controlar el acceso concurrente a los hornos para evitar que varios reposteros inserten galletas al mismo tiempo.
    - Sincronizar la escritura en el logger, asegurando que los registros de diferentes hilos no se mezclen ni se pierdan.
    - Gestionar el acceso a la cafetería, de modo que los reposteros puedan descansar sin conflictos.

Monitores

- Descripción: Se emplean para sincronizar el acceso a recursos compartidos y controlar las condiciones de espera de manera eficiente. Un monitor encapsula tanto los datos compartidos como las operaciones que los manipulan, proporcionando una abstracción más sencilla para la sincronización que los semáforos.
- Uso principal:
    - Cafetería: Controlar el acceso de los reposteros a la cafetería mediante un monitor que limita la cantidad de reposteros que pueden descansar simultáneamente. Esto se hace mediante una condición de espera que bloquea a los reposteros hasta que haya espacio disponible.
    - Producción y Horneado: Gestionar el acceso a los hornos para garantizar que solo un repostero pueda insertar galletas a la vez y coordinar la producción de galletas con los hornos.

Condiciones de Bloqueo (`Condition`)

- Descripción: Asociadas a los monitores, permiten suspender y reanudar la ejecución de los hilos bajo ciertas condiciones.
- Uso principal:
    - Controlar el estado de pausa/reanudación de los reposteros. Los hilos que están en pausa esperan hasta recibir una señal de reanudación.

Colas de Trabajo

- Descripción: Mecanismo para implementar el patrón productor-consumidor, permitiendo que los reposteros (productores) entreguen galletas a los hornos

(consumidores). En el caso de los hornos se realiza de forma secuencial iniciando a verificar en orden desde horno1. Mientras que el empaquetador está asociado directamente a un horno lo que facilita su implementación al ser horno directamente un atributo asociado al empaquetador.

- Uso principal:
    - Facilitar la transferencia de lotes de galletas entre los reposteros y los hornos de forma controlada.
    - Usado en la cafetería para controlar el flujo de reposteros (mediante monitores).

3.3. Esquema del Flujo de Trabajo

El flujo de trabajo del sistema concurrente sigue un esquema en el que los reposteros, hornos y la cafetería interactúan de forma sincronizada. A continuación, se describe el flujo:

1. Inicio del Proceso:
    - Los reposteros comienzan produciendo lotes de galletas.
2. Producción de Galletas:
    - Los reposteros trabajan en paralelo para crear galletas.
    - Una vez completado un lote, intentan acceder a un horno disponible secuencialmente.
3. Uso de los Hornos:
    - Cada horno tiene un `lock` que asegura que solo un repostero puede interactuar con él en un momento dado.
    - El repostero introduce el lote de galletas en el horno, lo cual se registra en el logger mediante un `lock` exclusivo para registros.
    - El horno hornea el lote, liberándose cuando termina el proceso.
4. Descanso en la Cafetería:
    - Los reposteros, después de hornear un lote, pueden decidir descansar en la cafetería.
    - La cafetería tiene un monitor que limita el número de reposteros que pueden usarla simultáneamente. Si el número de reposteros excede el límite, los reposteros se bloquean hasta que haya espacio disponible.
5. Gestión de Pausas:
    - Si el sistema pausa a los reposteros, estos quedan bloqueados utilizando una condición de bloqueo hasta recibir una señal de reanudación.
6. Registro de Eventos:
    - Todos los eventos del sistema (producción, horneado, descanso) se registran en el logger. El acceso al logger está sincronizado mediante un `lock` para evitar conflictos.
7. Reinicio del Ciclo:
    - Una vez descansados, los reposteros vuelven a producir más galletas, reiniciando el flujo de trabajo.

Diagrama del Flujo de Trabajo:

- Producción: Repostero → Lote de galletas → Verificación horno.
- Horneado: Horno disponible → Lote horneado → Liberación del horno.
- Descanso: Repostero → Acceso a la cafetería (limitado por monitor).
- Reanudación: Pausa/Reanudación gestionada por condiciones de bloqueo.
- Registro: Logger centralizado para capturar todos los eventos.

4. Clases Principales

4.1. Descripción de Clases y Atributos

4.1.1 Clase: Repostero

- **Descripción:**
  La clase `Repostero` extiende `Thread` y representa a un trabajador que produce galletas, las deposita en hornos y descansa en la cafetería. La clase maneja la producción y la interacción con los hornos de forma sincronizada, además de permitir que el repostero se pause y reanude según sea necesario.

Atributos:

1. **`ID` (int):** Identificador único para cada repostero, utilizado para hacer referencia al repostero en los logs y en otras interacciones dentro del sistema.
2. **`galletasProducidas` (int):**
   Número de galletas producidas en el último ciclo de producción.
3. **`historicoProducidas` (int):**
   Total acumulado de galletas producidas por el repostero a lo largo de su vida en el sistema.
4. **`galletasDesperdiciadas` (int):**
   Número de galletas que no se han podido utilizar debido a la capacidad llena de los hornos.
5. **`tandasProducidas` (int):**
   Número de tandas de galletas producidas en total.
6. **`descansando` (boolean):**
   Indica si el repostero está en estado de descanso. Si es `true`, el repostero está descansando; si es `false`, está trabajando.
7. **`situacion` (String):**
   Descripción del estado actual del repostero, como "Iniciado", "Descansando", "Produciendo", "Esperando", etc.
8. **`logger` (Logger):**
   Instancia del logger que registra los eventos y cambios de estado del repostero en los logs.
9. **`cafeteria` (Cafeteria):**
   Instancia de la clase `Cafeteria`, utilizada para gestionar el descanso del repostero.
10. **`arrayDeHornos` (Horno[]):**
    Un array de hornos disponibles. El repostero interactúa con estos hornos para depositar las galletas producidas.
11. **`tandas` (int):**
    Número de tandas que el repostero producirá antes de descansar.
12. **`enPausa` (boolean):**
    Bandera que indica si el repostero está en pausa. Si es `true`, el repostero está detenido.

13. `Lock` **(Object):**
    Acceso coordinado a los métodos de reanudar y para

Métodos:

1. `getTandasProducidas()`
    - **Descripción:**
      Devuelve el número de tandas producidas hasta el momento.
    - **Retorno:** `int`
2. `getSituacion()`
    - **Descripción:**
      Devuelve el estado actual del repostero, como "Produciendo", "Descansando", "Esperando", etc.
    - **Retorno:** `String`
3. `getTandas()`
    - **Descripción:**
      Devuelve el número de tandas que el repostero producirá antes de descansar.
    - **Retorno:** `int`
4. `getID()`
    - **Descripción:**
      Devuelve el identificador único del repostero.
    - **Retorno:** `String`
5. `getGalletasProducidas()`
    - **Descripción:**
      Devuelve el número de galletas producidas en el último ciclo de producción.
    - **Retorno:** `int`
6. `getGalletasDesperdiciadas()`
    - **Descripción:**
      Devuelve el número total de galletas que fueron desperdiciadas debido a que no había espacio suficiente en los hornos.
    - **Retorno:** `int`
7. `isDescansando()`
    - **Descripción:**
      Verifica si el repostero está descansando.
    - **Retorno:** `boolean`
8. `parar()`
    - **Descripción:**
      Pausa el repostero, cambiando su estado a "PARADO". El hilo se detiene hasta que se le indique reanudarse.
    - **Sin retorno.**
9. `reanudar()`
    - **Descripción:**
      Reanuda la ejecución del repostero después de una pausa. Notifica al hilo bloqueado que puede continuar.
    - **Sin retorno.**
10. `checkPausa()`
    - **Descripción:**
      Bloquea el hilo del repostero si está en pausa. El hilo se detiene hasta que se le indique que se reanude.
    - **Excepción:** `InterruptedException`

11. `todosLlenos()`
    - **Descripción:**
      Verifica si todos los hornos están llenos. Retorna `true` si todos los hornos están llenos y no pueden recibir más galletas.
    - **Retorno:** `boolean`
12. `introducirGalletas(int nGalletas)`
    - **Descripción:**
      Intenta introducir el número de galletas en el primer horno disponible. Si todos los hornos están llenos, el repostero espera.
    - **Parámetro:**
      `nGalletas` (int): El número de galletas a introducir en los hornos.
    - **Excepción:** `InterruptedException`
13. `producirGalletas()`
    - **Descripción:**
      Simula la producción de una tanda de galletas, incrementando el número total de galletas producidas.
    - **Excepción:** `InterruptedException`
14. `depositarEnHorno()`
    - **Descripción:**
      Deposita las galletas producidas en el primer horno disponible. Resetea el contador de galletas producidas después de depositarlas.
    - **Excepción:** `InterruptedException`
15. `descansar()`
    - **Descripción:**
      Simula el descanso del repostero usando la cafetería. El repostero espera hasta que termine de descansar.
    - **Excepción:** `InterruptedException`
16. `run()`
    - **Descripción:**
      Ciclo principal del repostero. Alterna entre la producción de galletas, su depósito en los hornos y el descanso, repitiendo el ciclo de manera indefinida.
    - **Excepción:** `InterruptedException`

4.1.2 Clase: Empaquetador

- **Descripción:**
  La clase `Empaquetador` extiende `Thread` y representa un trabajador en el sistema encargado de recoger las galletas horneadas, empaquetarlas y transportarlas al almacén. Cada empaquetador está asignado a un horno específico para recoger las galletas producidas por el repostero.

Atributos:

1. `id` **(String):**
   Identificador único del empaquetador. Es utilizado para hacer referencia al empaquetador en los logs y en otras interacciones dentro del sistema.
2. `hornoAsignado` **(Horno):**
   Instancia del `Horno` asignado al empaquetador. El empaquetador recoge las galletas del horno asignado para empaquetarlas.

3. **`almacen` (Almacen):**
   Instancia del `Almacen` donde se almacenan las galletas empaquetadas una vez que el empaquetador ha terminado su tarea.
4. **`galletasEmpaquetadas` (int):**
   Contador de las galletas que el empaquetador ha empaquetado hasta el momento.
5. **`logger` (Logger):**
   Instancia de la clase `Logger` que se utiliza para registrar los eventos y acciones realizadas por el empaquetador.
6. **`estado` (String):**
   El estado actual del empaquetador. Los posibles valores incluyen "Esperando", "Recogiendo tanda", "Transportando", entre otros.
7. **`tanda` (int):**
   El número de tandas de galletas empaquetadas hasta el momento.

Métodos:

1. **`getEstado()`**
   - **Descripción:**
     Devuelve el estado actual del empaquetador.
   - **Retorno:** `String`
2. **`getTanda()`**
   - **Descripción:**
     Devuelve el número de tandas que el empaquetador ha procesado hasta el momento.
   - **Retorno:** `int`
3. **`recogerGalletas()`**
   - **Descripción:**
     Simula el proceso de recoger galletas del horno asignado. Si hay galletas disponibles, las recoge y las suma al contador de galletas empaquetadas. Si no hay galletas, el empaquetador espera un tiempo antes de intentar nuevamente.
   - **Excepción:**
     `InterruptedException`
   - **Parámetro:** Ninguno
4. **`transportarAlmacen()`**
   - **Descripción:**
     Si el empaquetador ha empaquetado al menos 100 galletas, simula el transporte de las galletas al almacén. Luego, las galletas empaquetadas se almacenan en el almacén y se reinician los contadores de galletas y tandas.
   - **Excepción:**
     `InterruptedException`
   - **Parámetro:** Ninguno
5. **`run()`**
   - **Descripción:**
     El ciclo principal del empaquetador. Repite indefinidamente el proceso de recoger galletas del horno y transportarlas al almacén.
   - **Excepción:**
     `InterruptedException`

4.1.3 Clase: Horno

- **Descripción:**
  La clase `Horno` representa un horno donde se almacenan las galletas para ser horneadas. Gestiona su capacidad, el proceso de horneado y la retirada de galletas. Utiliza mecanismos de sincronización (con `Locks` y `Conditions`) para asegurar que los reposteros solo puedan interactuar con el horno cuando esté disponible.

  Atributos:

1. **`DURACION_HORNEO` (final int):**
   Define la duración del proceso de horneado, que es de 8 segundos (8,000 ms).
2. **`ID` (String):**
   Identificador único para el horno. Es de solo lectura y sirve para distinguir entre hornos en el sistema.
3. **`capacidadMAX` (int):**
   Capacidad máxima del horno, es decir, la cantidad máxima de galletas que el horno puede almacenar.
4. **`capacidad_actual` (int):**
   Número actual de galletas en el horno. Este valor se actualiza a medida que las galletas son agregadas o retiradas.
5. **`logger` (Logger):**
   Instancia de la clase `Logger`, utilizada para registrar los eventos del horno, como la adición y retirada de galletas.
6. **`horneando` (boolean):**
   Indica si el horno está en proceso de horneado. Si es `true`, el horno está ocupado cocinando las galletas; si es `false`, el horno está disponible.
7. **`horneadas` (boolean):**
   Indica si las galletas han sido horneadas. Si es `true`, las galletas están listas para ser retiradas.
8. **`historicoGalletas` (int):**
   Número total de galletas que se han horneado en el horno a lo largo del tiempo.
9. **`lock` (Lock):**
   Objeto `Lock` utilizado para garantizar que el acceso al horno sea exclusivo en determinadas operaciones.
10. **`lleno` (Condition):**
    `Condition` asociada al `lock`, usada para esperar hasta que el horno se haya llenado con galletas antes de que comience el horneado.
11. **`vacio` (Condition):**
    `Condition` asociada al `lock`, usada para esperar hasta que el horno esté vacío después de retirar las galletas.

Métodos:

1. `estaLleno()`
   - **Descripción:**
     Verifica si el horno está lleno, es decir, si ha alcanzado su capacidad máxima.
   - **Retorno:** `boolean`
2. `getID()`
   - **Descripción:**
     Devuelve el identificador único del horno.
   - **Retorno:** `String`
3. `getHistoricoGalletas()`
   - **Descripción:**
     Devuelve el número total de galletas horneadas en el horno.
   - **Retorno:** `int`
4. `getCapacidadMAX()`
   - **Descripción:**
     Devuelve la capacidad máxima del horno.
   - **Retorno:** `int`
5. `isHorneando()`
   - **Descripción:**
     Verifica si el horno está en proceso de horneado.
   - **Retorno:** `boolean`
6. `setHorneando(boolean horneando)`
   - **Descripción:**
     Establece si el horno está en proceso de horneado o no.
   - **Parámetro:**
     `horneando` (boolean): El estado del horno.
7. `getCapacidad_actual()`
   - **Descripción:**
     Devuelve la cantidad actual de galletas en el horno.
   - **Retorno:** `int`
8. `isHorneadas()`
   - **Descripción:**
     Verifica si las galletas en el horno han sido horneadas.
   - **Retorno:** `boolean`
9. `setHorneadas(boolean horneadas)`
   - **Descripción:**
     Establece si las galletas del horno están horneadas.
   - **Parámetro:**
     `horneadas` (boolean): El estado de las galletas (si han sido horneadas o no).
10. `agregarGalletas(int cantidad)`
    - **Descripción:**
      Agrega galletas al horno, respetando la capacidad máxima del horno. Si se excede la capacidad, se calculan y descartan las galletas sobrantes. Si el horno se llena, se notifica al hilo para que comience el proceso de horneado.
    - **Parámetro:**
      `cantidad` (int): Número de galletas a agregar al horno.
    - **Retorno:**
      `int`: El número de galletas desperdiciadas si la capacidad del horno se excede.

11. **`retirarGalletas(int cantidad)`**
    - **Descripción:**
      Retira galletas del horno. Si la cantidad solicitada es mayor a la disponible, retira todas las galletas. Si el horno queda vacío, se notifica al sistema.
    - **Parámetro:**
      `cantidad` (int): Número de galletas a retirar del horno.
    - **Retorno:**
      `int`: Número de galletas retiradas.
12. **`run()`**
    - **Descripción:**
      El ciclo principal del horno. Este método es ejecutado cuando se inicia el hilo del horno. El horno espera a que esté lleno antes de hornear y, luego, espera a que las galletas sean retiradas antes de volver a estar disponible.
    - **Excepción:**
      `InterruptedException`

4.1.4 Clase: Cafeteria

- **Descripción:**
  La clase `Cafeteria` representa el área donde los reposteros pueden descansar. Gestiona el acceso a la cafetera, asegurando que solo un repostero pueda usarla al mismo tiempo. Si varios reposteros intentan usar la cafetera simultáneamente, se sincronizan para que el acceso sea exclusivo.

Atributos:

1. **`cafeteraOcupada` (boolean):**
   Indica si la cafetera está ocupada o no. Si es `true`, significa que un repostero está usando la cafetera; si es `false`, la cafetera está libre.
2. **`logger` (Logger):**
   Instancia de la clase `Logger` utilizada para registrar los eventos de la cafetera, como cuando un repostero comienza a usarla o cuando termina de usarla.
3. **`idOcupado` (String):**
   Representa el identificador del repostero que está usando la cafetera. Si está vacío (`" "`), significa que la cafetera está libre; si tiene el identificador de un repostero, significa que esa persona está usando la cafetera.

Métodos:

1. **`getIdOcupado()`**
   - **Descripción:**
     Devuelve el identificador del repostero que está usando la cafetera. Si la cafetera está libre, devuelve un valor vacío.
   - **Retorno:**
     `String` - El identificador del repostero o un valor vacío si la cafetera está libre.
2. **`usarCafetera(String idRepostero)`**
   - **Descripción:**
     Permite que un repostero use la cafetera. Si la cafetera está ocupada, el repostero se bloquea hasta que esté disponible. El repostero ocupa la cafetera, simula un

tiempo de descanso (preparación de café) y luego la libera. Cuando termina, la cafetera queda libre y se notifica a los otros reposteros.
    - **Parámetro:**
      `idRepostero` (String) - El identificador del repostero que quiere usar la cafetera.
    - **Excepción:**
      `InterruptedException` - Si ocurre una interrupción mientras el repostero está esperando o descansando.

4.1.4 Clase: Almacen

- **Descripción:**
  La clase `Almacen` gestiona el inventario de galletas almacenadas, asegurando que el número de galletas almacenadas no exceda la capacidad máxima. También proporciona métodos para almacenar y consumir galletas, con mecanismos de exclusión mutua mediante monitores para evitar conflictos en el acceso concurrente.

Atributos:

1. **`logger` (Logger):**
   Instancia de la clase `Logger` utilizada para registrar los eventos relacionados con el almacén, como las operaciones de almacenamiento y consumo de galletas.
2. **`CAPACIDAD_MAXIMA` (int):**
   La capacidad máxima del almacén, es decir, el número total de galletas que puede almacenar el almacén.
3. **`capacidad_actual` (int):**
   El número actual de galletas almacenadas. Es de solo lectura por otros componentes del sistema.
4. **`ID` (String):**
   El identificador único del almacén. Este valor es solo de lectura y se utiliza para distinguir entre diferentes almacenes si existen varios.
5. **`consumidas` (int):**
   El número total de galletas que se han consumido del almacén. Este valor es útil para el cliente RMI que consulta las galletas consumidas.

Métodos:

1. **`getConsumidas()`**
    - **Descripción:**
      Devuelve el número de galletas que han sido consumidas del almacén.
    - **Retorno:** `int`
2. **`getCapacidad_actual()`**
    - **Descripción:**
      Devuelve la cantidad de galletas que hay actualmente en el almacén.
    - **Retorno:** `int`
3. **`getID()`**
    - **Descripción:**
      Devuelve el identificador único del almacén.
    - **Retorno:** `String`
4. **`almacenar(int cantidad, String autor)`**

- **Descripción:**
  Almacena un número determinado de galletas en el almacén. Si no hay suficiente espacio, el hilo se bloquea hasta que se pueda agregar el número deseado de galletas.
- **Parámetros:**
  - `cantidad` (int): Número de galletas a almacenar.
  - `autor` (String): El identificador del hilo que está realizando la operación de almacenamiento.
- **Excepción:**
  `InterruptedException`
- **Sin retorno.**

5. `comer()`
   - **Descripción:**
     Permite consumir 100 galletas del almacén. Si hay suficientes galletas, las consume y actualiza la capacidad del almacén. Si no hay suficientes galletas, no realiza ninguna acción.
   - **Sin parámetros.**
   - **Sin retorno.**
   - **Excepción:**
     `Exception` (en caso de error en el proceso).

4.1.5 Clase: Logger

- **Descripción:**
  La clase `Logger` está diseñada para gestionar el registro de eventos en un archivo de log. Permite almacenar mensajes relacionados con el estado del sistema, incluyendo advertencias y errores, y proporciona una forma organizada de hacer seguimiento de las actividades a lo largo de la ejecución del programa. Además, se encarga de crear el archivo de log, gestionarlo y agregarle entradas.

Atributos:

1. `path` **(String):**
   El archivo en el que se almacenarán los logs. Este atributo contiene la ruta del archivo donde se guardarán los mensajes de log.

Métodos:

1. `Logger(String path, boolean reset)`
   - **Descripción:**
     Constructor de la clase `Logger`. Inicializa el `Logger` y establece la ruta del archivo de log. Si el parámetro `reset` es `true`, borra el archivo de log anterior antes de crear uno nuevo.
   - **Parámetros:**
     - `path` (String): Ruta donde se guardará el archivo de log.
     - `reset` (boolean): Si es `true`, el archivo de log existente se elimina antes de crear uno nuevo.
2. `borrar()`
   - **Descripción:**
     Método privado que elimina el archivo de log si existe.

    - **Sin parámetros.**
3. `checkExist()`
    - **Descripción:**
      Método privado que verifica si el archivo de log existe. Si no existe, lo crea e inserta un banner de fecha de creación.
    - **Sin parámetros.**
4. `write(String msg)`
    - **Descripción:**
      Método privado que escribe el mensaje proporcionado en el archivo de log.
    - **Parámetros:**
        - `msg` (String): El mensaje que se debe registrar en el archivo de log.
5. `add(String author, String msg)`
    - **Descripción:**
      Método público para agregar una entrada al log. La entrada incluirá la fecha actual, el autor y el mensaje proporcionado.
    - **Parámetros:**
        - `author` (String): El autor que genera el mensaje.
        - `msg` (String): El mensaje a registrar en el log.
6. `add(String author, String author2, String msg)`
    - **Descripción:**
      Método público que sobrecarga `add`. Permite agregar dos autores en una sola entrada de log. Incluye la fecha, los dos autores y el mensaje.
    - **Parámetros:**
        - `author` (String): El primer autor que genera el mensaje.
        - `author2` (String): El segundo autor que genera el mensaje.
        - `msg` (String): El mensaje a registrar en el log.
7. `addE(String author, String msg)`
    - **Descripción:**
      Método que sobrescribe el método `add`, pero marca el mensaje como un error, añadiendo los tags de error `[[ERROR]]`. Facilita la identificación de los errores en el archivo de log.
    - **Parámetros:**
        - `author` (String): El autor que genera el error.
        - `msg` (String): El mensaje de error a registrar.

4.1.6 Clase: Utilidades

- **Descripción:**
  La clase `Utilidades` contiene métodos de utilidad general sin propósito específico que pueden ser utilizados en todo el sistema.

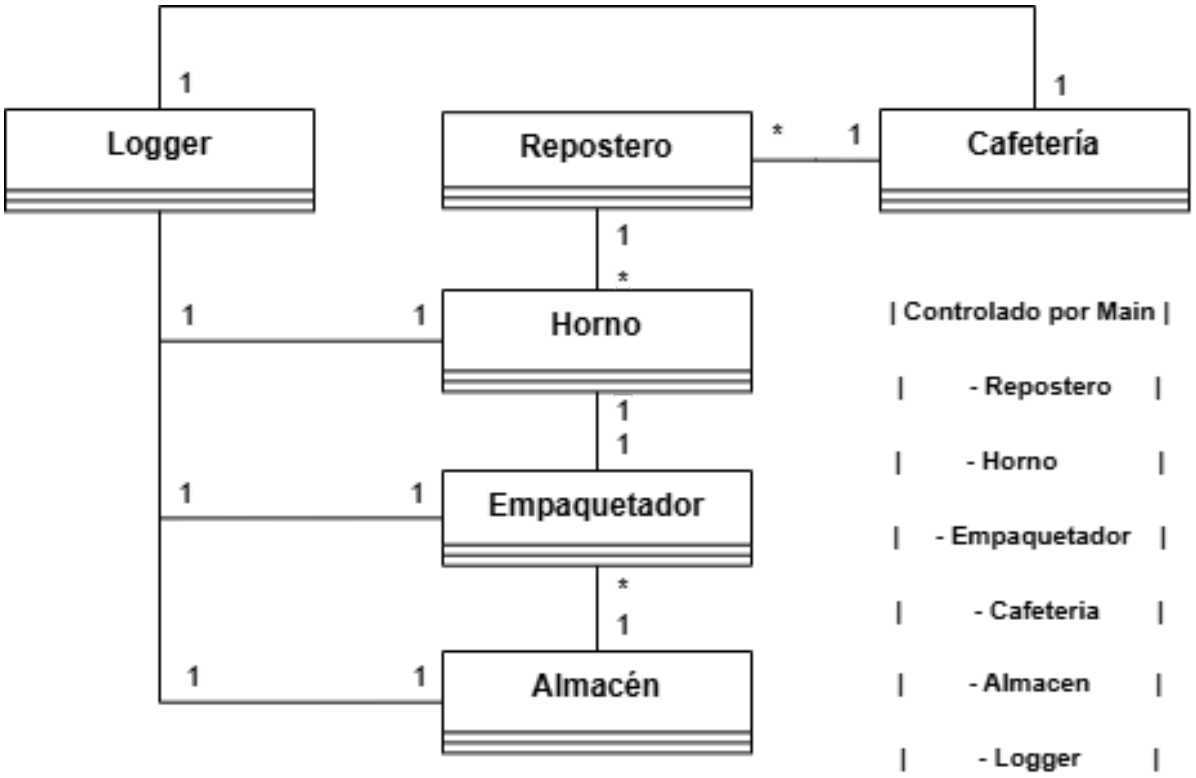
Métodos:

1. `numeroRandom(int minimo, int maximo)`
    - **Descripción:**
      Este método genera y devuelve un número aleatorio comprendido entre los valores `minimo` y `maximo`, ambos inclusive.
    - **Parámetros:**
        - `minimo` (int): El límite inferior del rango del número aleatorio.

- `maximo` (int): El límite superior del rango del número aleatorio.

- **Retorno:**
  `int`: Un número aleatorio dentro del rango [minimo, maximo].
- **Precondición:**
  - `maximo >= minimo`: El valor máximo debe ser mayor o igual al valor mínimo para evitar un comportamiento inesperado.

5. Diagrama de Clases



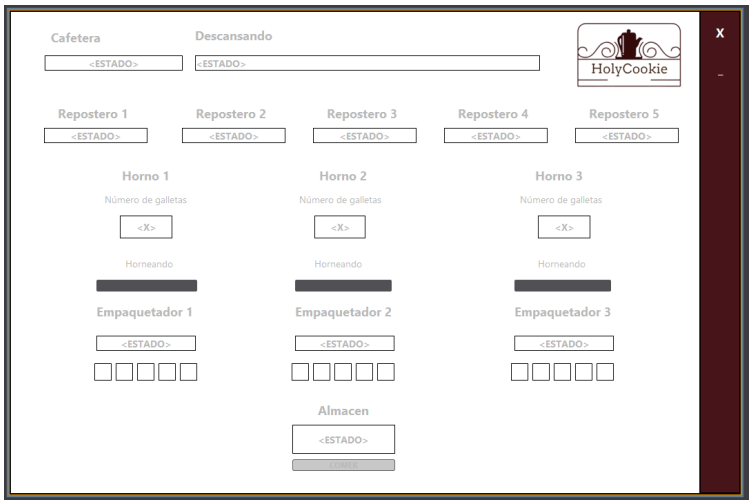
No aparece en el diagrama la clase Utilidades, no la hemos contado al ser miscelánea y auxiliar, pero esta es utilizado tanto en empaquetador, repostero, horno y cafetería.

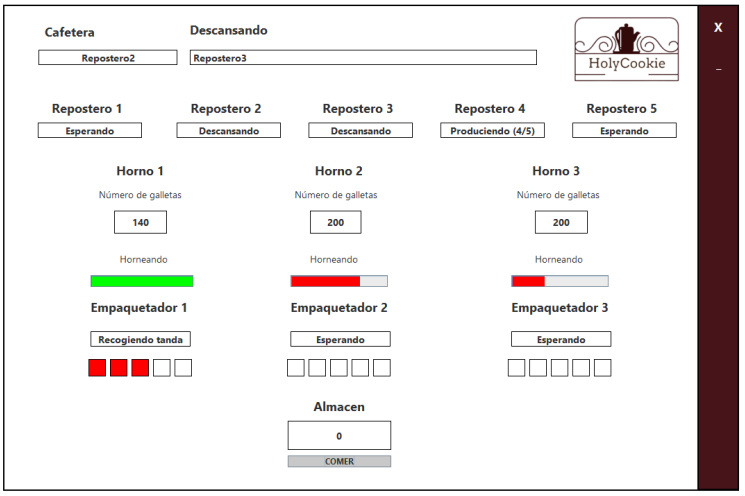
Por último, la clase Main, engloba a todas estas y ejecuta los hilos de los objetos, al igual que existe la clase FabricaGalletasImpl, que envuelve a todas los objetos para enviarlos en el apartado RMI.

6. Interfaz Gráfica

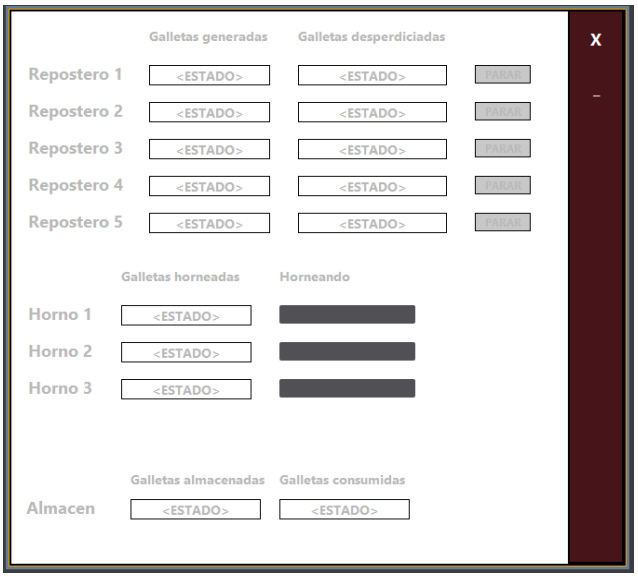
Siguiendo las imágenes orientativas del enunciado de la práctica hemos realizado las siguientes interfaces.

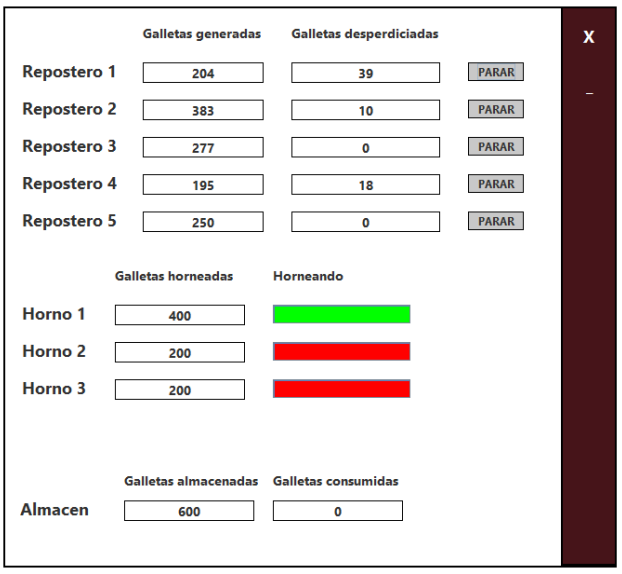
6.1 Interfaz Servidor





6.2 Interfaz Cliente





7. Código Fuente (Anexo)

7.1. Código Completo del Sistema Concurrente

7.1.1 Clase Main (Principal) [Carpeta Server/Main]

```
package Server.Main;

/**************************************************************************************
****
* CODIGO PRINCIPAL, MAIN GLOBAL DEL PROGRAMA (INSTANCIADO DE OBJS + GUI + SERVER
RMI)
**************************************************************************************
***/

import Server.fabrica_galletas.*;
import Server.misc.Logger;
import Server.GUI.Principal;

import java.rmi.Naming;
import java.rmi.registry.LocateRegistry;

public class Main {

	public static void main(String[] args) throws InterruptedException {

	/*****************************************
	*   CONSTANTES
	*****************************************/
	final int MAX_GALLETAS_HORNO = 200;
	final int NUMERO_REPOSTEROS = 5;


	/*****************************************
	*   BOOLEANOS POR SI SE QUIERE DESACTIVAR GUI/RMI
	*****************************************/
	final boolean GUI_ACTIVADO = true;
	final boolean RMI_ACTIVADO = true;

	/*****************************************
	*   INSTANCIAS DE CLASES FABRICA
	*****************************************/

	Logger logger = new Logger("../evolucionGalletas.txt", true);
	Almacen almacen = new Almacen("ALMACEN1", 1000, logger);

	Horno[] hornos = {new Horno("Horno1", MAX_GALLETAS_HORNO, logger),
		new Horno("Horno2", MAX_GALLETAS_HORNO, logger),
		new Horno("Horno3", MAX_GALLETAS_HORNO, logger) };

	for (Horno horno : hornos)
		horno.start();

	Cafeteria cafeteria = new Cafeteria(logger);

	// Indexados en array para no perder acceso a cualquiera de ellos
	Repostero[] reposteros = new Repostero[NUMERO_REPOSTEROS];
```

```
		for (int i = 0; i < reposteros.length; i++) {
			reposteros[i] = new Repostero("Repostero" + (i + 1), hornos,
cafeteria, logger);
			reposteros[i].start();
		}

		// Inicializar los empaquetadores
		Empaquetador[] empaquetadores = new Empaquetador[3];
		for (int i = 0; i < empaquetadores.length; i++) {
			empaquetadores[i] = new Empaquetador("Empaquetador" + (i + 1),
hornos[i], almacen, logger);
			empaquetadores[i].start();
		}

		/**************************************
		*   PARTE DE GUI
		**************************************/
		if (GUI_ACTIVADO) {
			logger.add("Sistema", "GUI Activandose");
			Principal principal = new Principal(cafeteria, reposteros, hornos,
empaquetadores, almacen);
			principal.setVisible(true);
			Thread thread = new Thread(principal); // Pasar Principal como
Runnable a un Thread
			thread.start(); // Inicia el hilo, llamando al método run
			logger.add("Sistema", "GUI listo...");
		}


		/**************************************
		*   PARTE DE RMI SERVIDOR
		**************************************/
		if (RMI_ACTIVADO) {
			try {
			logger.add("Sistema", "RMI INCIANDOSE");

			FabricaGalletasImpl fabrica = new FabricaGalletasImpl(reposteros,
hornos, almacen);

			LocateRegistry.createRegistry(1099);

			Naming.rebind("//localhost/fabrica", fabrica);

			logger.add("Sistema", "Servidor RMI listo...");

			} catch (Exception e) {
			logger.addE("Sistema", "Error en el servidor RMI: " + e.getMessage());
			System.out.println(e.getMessage());
			}
		}
		}
}
```

7.1.2 Clase Cafeteria [Carpeta Server/fabrica_galletas]

```
package Server.fabrica_galletas;

import Server.misc.Logger;

/*
*   Objetivo: Manejar el uso de la cafetera
 */


public class Cafeteria {

	private boolean cafeteraOcupada;
	private Logger logger;
	private String idOcupado = "  ";

	public Cafeteria(Logger logger) {
	this.cafeteraOcupada = false;
	this.logger = logger;
	}

	public String getIdOcupado() {
	return idOcupado;
	}

	// Método para acceder a la cafetera
	public synchronized void usarCafetera(String idRepostero) throws
InterruptedException {
	while (cafeteraOcupada) {
		logger.add("Cafeteria", "Cafeteria ocupada");
		wait(); // Espera hasta que la cafetera esté libre
	}

	// Ocupa la cafetera
	cafeteraOcupada = true;
	idOcupado = idRepostero;
	logger.add("Cafeteria", idRepostero + " usando cafetera");
	Thread.sleep(2000); // Simula el tiempo para preparar café
	logger.add("Cafeteria",idRepostero + " terminó de usar la cafetera.");

	// Libera la cafetera
	cafeteraOcupada = false;
	idOcupado = "  ";
	notifyAll(); // Notifica a otros reposteros que pueden usar la cafetera
	}
}
```

7.1.3 Clase Repostero [Carpeta Server/fabrica_galletas]

```
package Server.fabrica_galletas;

/**
 * ****************************
 * IMPLEMENTACION DE REPOSTERO
 ******************************
 */
import Server.misc.Utilidades;
import Server.misc.Logger;

public class Repostero extends Thread {

	private String ID;
	private int galletasProducidas = 0;
	private int historicoProducidas = 0;
	private int galletasDesperdiciadas = 0;

	private int tandasProducidas = 0;

	private boolean descansando = false;
	private String situacion;

	private Logger logger;
	private Cafeteria cafeteria;
	private Horno[] arrayDeHornos;
	private int tandas = 0;

	private boolean enPausa = false; // Flag para controlar la pausa
	private final Object lock = new Object(); // Objeto de bloqueo para
sincronización

	public Repostero(String ID, Horno[] arrayDeHornos, Cafeteria cafeteria,
Logger logger) {
	this.ID = ID;
	this.logger = logger;
	this.arrayDeHornos = arrayDeHornos;
	this.cafeteria = cafeteria;
	situacion = "Iniciado";
	}

	public int getTandasProducidas() {
	return tandasProducidas;
	}

	public String getSituacion() {
	return situacion;
	}

	public int getTandas() {
	return tandas;
	}

	public String getID() {
	return ID;
	}
```

```
public int getGalletasProducidas() {
	return historicoProducidas;
	}

	public int getGalletasDesperdiciadas() {
	return galletasDesperdiciadas;
	}


	public boolean isDescansando() {
	return descansando;
	}

	// Método para pausar el hilo
	public void parar() {
	synchronized (lock) {
		enPausa = true; // Cambia el estado a pausa
	}
	situacion = "PARADO";
	}

	// Método para reanudar el hilo
	public void reanudar() {
	synchronized (lock) {
		enPausa = false; // Cambia el estado a reanudado
		lock.notify(); // Despierta al hilo bloqueado en el wait()
	}
	situacion = "REANUDANDO";
	}

	/*
	OBJ: Bloquea hilo hasta que se le indique que reanude
	PRE: -
	POST: -
	*/
	public void checkPausa() throws InterruptedException {
	synchronized (lock) {
		while (enPausa) {
		lock.wait();
		}
	}
	}

	/*
	OBJ: Verificar si todos los hornos están llenos.
	PRE: El array de hornos debe estar inicializado.
	POST: Devuelve true si todos los hornos están llenos, false en caso
contrario.
	*/
	private boolean todosLlenos() {
	for (Horno horno : arrayDeHornos) {
		if (!horno.estaLleno() && !horno.isHorneadas()) {
		return false;
		}
	}
	return true;
	}
```

```
	/*
	OBJ: Introducir galletas en el primer horno disponible, manejando
desperdicios si es necesario.
	PRE: El array de hornos debe estar inicializado y no estar vacío.
	POST: Las galletas se depositan en el horno disponible o el hilo espera si
todos los hornos están llenos.
	*/
	public synchronized void introducirGalletas(int nGalletas) throws
InterruptedException {
	int galletas = nGalletas;

	while (todosLlenos()) {
		situacion = "Esperando";
		logger.add(ID, "Todos los hornos llenos. Esperando...");
		Thread.sleep(Utilidades.numeroRandom(1000, 1500));
	}

	for (Horno horno : arrayDeHornos) {
		if (!horno.estaLleno() && !horno.isHorneadas()) {
		galletasDesperdiciadas += horno.agregarGalletas(galletas);

		logger.add(ID, galletas + " colocadas en " + horno.getID());
		break;
		} else {
		logger.add(ID, horno.getID() + " Lleno, pasando al siguiente");
		}
	}
	}

	/*
	OBJ: Simular la producción de una tanda de galletas.
	PRE: -
	POST: Incrementa el contador de galletas producidas y tandas realizadas.
	*/
	public void producirGalletas() throws InterruptedException {
	tandasProducidas++;
	situacion = "Produciendo (" + tandasProducidas + "/" + tandas + ")";
	Thread.sleep(Utilidades.numeroRandom(2000, 4000)); // Simula entre 2 y 4
segundos de producción

	int galletas = Utilidades.numeroRandom(37, 45); // Genera entre 37 y 45
galletas
	galletasProducidas = galletas;
	historicoProducidas += galletas;

	logger.add(ID, " Produjo " + galletas + " galletas. Total producidas: " +
historicoProducidas);
	}

	/*
	OBJ: Depositar las galletas producidas en un horno disponible.
	PRE: Debe haber galletas producidas y al menos un horno debe tener espacio
disponible.
	POST: Las galletas producidas se depositan en un horno y el contador se
reinicia.
	*/
```

```
public synchronized void depositarEnHorno() throws InterruptedException {
	situacion = "Depositando";
	logger.add(ID, " Intenta depositar " + galletasProducidas + " galletas.");
	introducirGalletas(galletasProducidas);
	galletasProducidas = 0; // Resetea la cantidad tras depositar
	}

	/*
	OBJ: Simular el descanso del repostero usando la cafetería.
	PRE: La cafetería debe estar inicializada.
	POST: El hilo descansa durante un tiempo aleatorio antes de retomar la
producción.
	*/
	public void descansar() throws InterruptedException {
	descansando = true;
	situacion = "Descansando";
	logger.add(ID, " Está descansando.");
	cafeteria.usarCafetera(ID);
	Thread.sleep(Utilidades.numeroRandom(3000, 6000)); // Descanso entre 3 y 6
segundos
	logger.add(ID, " Terminó de descansar.");
	descansando = false;
	}

	/*
	OBJ: Ciclo principal del repostero, alternando entre producción, depósito y
descanso.
	PRE: Todos los objetos asociados deben estar inicializados.
	POST: Ejecuta el ciclo de forma indefinida hasta que el hilo sea
interrumpido.
	*/
	@Override
	public void run() {
	try {
		while (true) {
		tandas = Utilidades.numeroRandom(3, 5); // De 3 a 5 tandas antes de
descansar

		for (int i = 0; i < tandas; i++) {
			// Antes de cada tarea, comprobamos si el hilo está en pausa
			checkPausa();
			producirGalletas();  // Produce una tanda
			checkPausa();
			depositarEnHorno(); // Deposita las galletas
			// Pausar después de cada tarea, si es necesario
			checkPausa();
		}

		descansar();

		// Pausar al final de cada ciclo de tandas, si es necesario
		checkPausa();

		tandasProducidas = 0;
		}
	} catch (InterruptedException e) {
		logger.addE(ID, " fue interrumpido.");
	}
	}
```

```
}
```

7.1.4 Clase Horno [Carpeta Server/fabrica_galletas]

```
package Server.fabrica_galletas;

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

import Server.misc.Logger;

/*
*   Objetivo: Implementacion del horno
       Notas:
       + Aproximación mediante locks
       + Se intentó con monitores pero las condiciones con locks lo hacen
             mucho mas flexible con los signal
*   Autor: Rodrigo Palomo
 */
public class Horno extends Thread {

       // Variables y constantes
       private final int DURACION_HORNEO = 8 * 1000;   // Constante
       private String ID;                              // READONLY
       private int capacidadMAX;                       // READONLY
       private int capacidad_actual = 0;
       private Logger logger;
       private boolean horneando = false;
       private boolean horneadas = false;
       private int historicoGalletas = 0;

       // Lock y conditions
       private final Lock lock = new ReentrantLock();
       private final Condition lleno = lock.newCondition();
       private final Condition vacio = lock.newCondition();

       public Horno(String ID, int capacidadMAX, Logger logger) {
       this.ID = ID;
       this.capacidadMAX = capacidadMAX;
       this.logger = logger;

       }

       public boolean estaLleno() {
       return capacidad_actual == capacidadMAX;

       }


       public String getID() {
       return ID;
       }

       public int getHistoricoGalletas() {
       return historicoGalletas;
       }
```

```
	public int getCapacidadMAX() {
	return capacidadMAX;
	}

	public boolean isHorneando() {
	return horneando;
	}

	public void setHorneando(boolean horneando) {
	this.horneando = horneando;
	}

	public int getCapacidad_actual() {
	return capacidad_actual;
	}

	public boolean isHorneadas() {
	return horneadas;
	}

	public void setHorneadas(boolean horneadas) {
	this.horneadas = horneadas;
	}

	/*
		OBJ: Agrega galletas al horno hasta llegar a 200, si eso ocurre
			entonces libera el condition que inicia el horneado
		PRE: -
		POST: -
		NOTA: Retorna desperdicio para poder luego redirigirlo a donde
convenga
	*/
	public int agregarGalletas(int cantidad) {
	int desperdicio = 0;
	lock.lock();
	try {
		desperdicio = (capacidad_actual + cantidad) - capacidadMAX;

		if (desperdicio > 0) {
		logger.add(ID, "No caben todas, galletas desechadas: " + desperdicio);
		capacidad_actual = capacidadMAX;
		} else {
		capacidad_actual += cantidad;
		desperdicio = 0;

		}

		logger.add(ID, " Se han agregado " + (cantidad - desperdicio) + "
galletas. Total: " + capacidad_actual);

		// Si el horno se llena, notificar al hilo del horno
		if (capacidad_actual == capacidadMAX) {
		lleno.signal();
		}

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

```
}
return desperdicio;
}

/*
	OBJ: Retira galletas, siempre post-horneado, cuando las retire
	todas entonces dará la senal para seguir recibiendo nuevas
	galletas.
	PRE: -
	POST: -
	NOTA: Si la cantidad a retirar es mayor a la disponible se sacaran
		todas las que haya.
*/
public int retirarGalletas(int cantidad) {
int retiradas = 0;
lock.lock();
try {
	if (isHorneadas()){
	// Caluclar retiradas para que no salga negativo
	if (cantidad > capacidad_actual) {
		retiradas = capacidad_actual;
		capacidad_actual = 0;
	} else {
		retiradas = cantidad;
		capacidad_actual -= cantidad;

	}

	// Logger + signal de lock
	String mensaje = " Se han retirado " + retiradas + " galletas.";

	if (capacidad_actual == 0) {
		mensaje += " El horno está vacío.";
		vacio.signal(); // Señalar que el horno está vacío
	} else {
		mensaje += " Galletas restantes: " + capacidad_actual;
	}

	logger.add(ID, mensaje);
	}
} finally {
	lock.unlock();
}
return retiradas;
}



/*
	OBJ: Ciclo principal del propio hilo, bloqueo de lock para esperar
	horneado y para esperar retirada completa
	PRE: -
	POST: -
*/
@Override
public void run() {
logger.add(ID, " Encendido!");
boolean seguir = true;
```

```
	while (seguir) {
		lock.lock();
		try {
		// ESTADIO 1: A espera de signal que horno este lleno
(agregarGalletas)
		while (capacidad_actual < capacidadMAX) {
			setHorneadas(false);
			logger.add(ID, " Esperando a llenarse. Galletas actuales: " +
capacidad_actual);
			lleno.await();
		}

		// ESTADIO 2: Proceso de horneado
		logger.add(ID, " Horneando...");
		setHorneando(true);
		Thread.sleep(DURACION_HORNEO);
		logger.add(ID, " Horneado completado.");
		historicoGalletas += capacidad_actual;
		setHorneando(false);
		setHorneadas(true);

		// ESTADIO 3: A espera de signal que horno este vacio
(retirarGalletas)
		while (capacidad_actual > 0) {
			logger.add(ID, " Esperando a que el horno se vacíe. Galletas
restantes: " + capacidad_actual);
			vacio.await();
		}

		// FIN DEL CICLO
		logger.add(ID, " Horno vacío y listo para recibir nuevas galletas.");

		} catch (InterruptedException e) {
		Thread.currentThread().interrupt();
		logger.addE(ID, "Horno detenido.");
		seguir = false;
		} finally {
		lock.unlock();
		}
	}
	}
}
```

7.1.5 Clase Empaquetador [Carpeta Server/fabrica_galletas]

```
package Server.fabrica_galletas;

/****************************************
 * Implementacion clase empaquetador
 ****************************************/
import Server.misc.Logger;
import Server.misc.Utilidades;

public class Empaquetador extends Thread {

	private String id;
	private Horno hornoAsignado;
	private Almacen almacen;
	private int galletasEmpaquetadas;
	private Logger logger;
	private String estado;
	private int tanda = 0;

	public Empaquetador(String id, Horno hornoAsignado, Almacen almacen, Logger
logger) {
	this.id = id;
	this.hornoAsignado = hornoAsignado;
	this.almacen = almacen;
	this.galletasEmpaquetadas = 0;
	this.logger = logger;
	estado = "Iniciado";
	}

	public String getEstado() {
	return estado;
	}

	public int getTanda() {
	return tanda;
	}

	/**
	* OBJ: Recoger galletas del horno asignado y actualizarlas al contador del
	* empaquetador. PRE: El horno debe estar inicializado y asignado
	* correctamente. POST: Si hay galletas disponibles en el horno, las recoge
	* y las suma al total del empaquetador. Si no hay galletas, espera un
	* tiempo antes de reintentar.
	*/
	public void recogerGalletas() throws InterruptedException {
	estado = "Esperando";
	int galletasRetiradas = hornoAsignado.retirarGalletas(20);
	if (galletasRetiradas > 0) {
		tanda++;
		estado = "Recogiendo tanda";

		galletasEmpaquetadas += galletasRetiradas;
		String mensaje = id + " recogio " + galletasRetiradas + " galletas del
" + hornoAsignado.getID()
			+ ". Total empaquetadas: " + galletasEmpaquetadas;
		logger.add(id, mensaje);
		Thread.sleep(Utilidades.numeroRandom(500, 1000));
	}
```

```
	else {
		estado = "Esperando";
		String mensaje = id + " esperando porque el horno esta vacio.";
		logger.add(id, mensaje);
		Thread.sleep(Utilidades.numeroRandom(1000, 1500));
	}
	}

	/**
	* OBJ: Transportar las galletas empaquetadas al almacén cuando se alcanzan
	* 100 unidades. PRE: El almacén debe estar inicializado y el contador de
	* galletas empaquetadas debe ser igual o mayor a 100. POST: Las galletas
	* empaquetadas son almacenadas y el contador del empaquetador se reinicia.
	*
	* @throws InterruptedException si ocurre una interrupción durante el
	* transporte.
	*/
	public void transportarAlmacen() throws InterruptedException {
	if (galletasEmpaquetadas >= 100) {
		estado = "Transportando";
		String mensaje = id + " transporta " + galletasEmpaquetadas + "
galletas al almacen.";
		logger.add(id, mensaje);
		Thread.sleep(Utilidades.numeroRandom(2000, 4000));
		almacen.almacenar(galletasEmpaquetadas, id);
		galletasEmpaquetadas = 0;
		tanda = 0;
	}
	}

	/**
	* OBJ: Ciclo principal del empaquetador que recoge galletas y las
	* transporta al almacén. PRE: Debe haberse inicializado correctamente el
	* horno, el almacén y el logger. POST: La operación se realiza
	* indefinidamente hasta que el hilo sea interrumpido.
	*/
	@Override
	public void run() {
	try {
		while (true) {
		recogerGalletas();
		transportarAlmacen();
		}
	} catch (InterruptedException e) {
		String mensaje = id + " fue interrumpido.";
		logger.addE(id, mensaje);
	}
	}
}
```

7.1.6 Clase Almacen [Carpeta Server/fabrica_galletas]

```
package Server.fabrica_galletas;

/*
   Objetivo: Implementacion del almacen
       Notas:
       + Se ha usado monitores para una implementación más entendible
 */

import Server.misc.Logger;


public class Almacen {

       // Atributos
       private Logger logger;
       private int CAPACIDAD_MAXIMA;
       private int capacidad_actual = 0; //READONLY por el resto
       private String ID; //READONLY
       private int consumidas = 0; // Para cliente RMI



       public Almacen(String ID, int CAPACIDAD_MAXIMA, Logger logger) {
       this.CAPACIDAD_MAXIMA = CAPACIDAD_MAXIMA;
       this.ID = ID;
       this.logger = logger;
       }

       /*
              OBJ: getter galletas consumidas (Cliente rmi)
              PRE: -
              POST: -
       */
       public int getConsumidas() {
       return consumidas;
       }

       /*
              OBJ: getter capacidad total
              PRE: -
              POST: -
              EXTRA: Usado synchronized porque se lee mejor que lock con try y
finally
       */
       public synchronized int getCapacidad_actual() {
       return capacidad_actual;
       }

       /*
              OBJ: getter ID
              PRE: -
              POST: -
       */
       public String getID() {
       return ID;
       }
```

```
	/*
		OBJ: Almacena en capacidad actual en numero de galletas indicado
		Usa exclusion mutua mendiante  monitores
		PRE: -
		POST: Capacidad actual actualizada o hilo en espera para ello
	*/
	public void almacenar(int cantidad, String autor) {
	try {
		// Bloque sincronizado para manejar la exclusión mutua
		synchronized (this) {
		// Esperar si no hay suficiente espacio en el almacén
		while (getCapacidad_actual() + cantidad > CAPACIDAD_MAXIMA) {
			logger.add(ID, autor, "Capacidad máxima alcanzada, esperando
turno...");
			wait();
		}
		// Agregar las galletas al almacén
		capacidad_actual += cantidad;


		logger.add(ID, autor, "Se almacenaron " + cantidad + " galletas.
Capacidad actual: " + capacidad_actual);

		// Notificar a todos los hilos que podrían estar esperando
		notifyAll();
		}

	} catch (InterruptedException e) {
		Thread.currentThread().interrupt();
		logger.addE(ID,"Hilo interrumpido: " + e.getMessage());
	}
	}
	/*
		OBJ: Consume galletas del almacen
		Usa exclusion mutua mendiante locks y conditions
		PRE: Pensada para interaccion por boton en interfaz "Comer"
		\-> Por eso no hay waits ni ningun tipo de mecanismo de exclusion,
			si no hay suficientes galletas el consumo es 0
		POST: Capacidad actual actualizada o hilo en espera para ello
	*/
	public synchronized void comer() {
	String autor = "Usuario";
	try {
		if (capacidad_actual >= 100) {
		capacidad_actual -= 100;
		logger.add(ID, autor, "Se consumieron 100 galletas. Capacidad actual:
" + capacidad_actual);

		consumidas += 100;
		// Notificar a los hilos en espera que la capacidad ha cambiado
		notify();

		} else {
		logger.add(ID, autor, "No hay suficientes galletas para consumir 100.
Capacidad actual: " + capacidad_actual);
		}
	} catch (Exception e) {
		logger.addE(autor,"Error en comer: " + e.getMessage());
```

```
	}}}
```

7.1.7 Clase Logger [Carpeta Server/misc]

```
package Server.misc;

/*
*   Objetivo: Clase especifica para poder annadir eventos al log de ejecución
	Notas:
	+ Opcion habilitada para que a cada arranque se resetee el log
		+ Seleccionable que lo haga o no en instanciación
	+ Pasar como parametro en la instancia de cualquier clase el logger
	instanciado en Main.
	USO GENERAL: logger.add("ADMIN", "Esto es una ");
 */
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

public class Logger {

	private String path;

	public Logger(String path, boolean reset) {
	this.path = path;

	// Borrar el antiguo log segun usuario decida
	if (reset) {
		borrar();
	}

	// Checkea si el archivo existe para crear el banner
	checkExist();
	}

	private void borrar() {
	/*
		OBJ: Procedimineto privado que borra el archivo si existe
		PRE: -
		POST: Archivo borrado
	*/

	File archivo = new File(this.path);
	if (archivo.exists()) {
		if (!archivo.delete()) { // Error si se bloquea el borrado
		System.err.println("Error al eliminar el archivo.");
		}
	}
	}

	private void checkExist() {
	/*
		OBJ: Procedimiento privado que si el archivo existe introduce en el
			un banner de fecha de creación del mismo
		PRE: -
		POST: Archivo creado con banner nuevo
```

```
	*/

	File archivo = new File(this.path);
	try {
		if (archivo.createNewFile()) { // Devuelve true si  existe y entra en
if
		DateTimeFormatter formato = DateTimeFormatter.ofPattern("dd/MM/yyyy
HH:mm:ss");
		write("========== ARCHIVO LOG CREADO [" +
LocalDateTime.now().format(formato) + "] ==========");
		}
	} catch (IOException e) {
		System.err.println(e);
	}
	}

	private void write(String msg) {
	/*
		OBJ: Metodo privado exclusivamente dedicado a cargar el archivo y
			volcar los datos en él
		PRE: Archivo existe
		POST: Datos escritos
	*/

	try (BufferedWriter escritor = new BufferedWriter(new FileWriter(this.path,
true))) {
		escritor.write(msg);
		escritor.newLine();
	} catch (IOException e) {
		System.err.println(e);
	}
	}


	public void add(String author, String msg) {
	/*
		OBJ: Metodo publico para que cualquiera pueda introducir linea nueva
			al log.
			Nota: Se introduce la fecha de creacion)
		PRE: Introducir Autor y el propio mensaje
		POST:  Datos escritos
	*/

	String cadena;
	DateTimeFormatter formato = DateTimeFormatter.ofPattern("dd/MM/yyyy
HH:mm:ss");
	String date = LocalDateTime.now().format(formato);

	cadena = "[" + author + "] [" + date + "]: " + msg;

	write(cadena);
	}

	// Sobrecarga de metodo para aceptar dos autores
	public void add(String author, String author2, String msg) {
	/*
		OBJ: Metodo publico para que cualquiera pueda introducir linea nueva
			al log.
			Nota: Se introduce la fecha de creacion)
```

```
		PRE: Introducir Autor (2) y el propio mensaje
		POST:  Datos escritos
	*/

	String cadena;
	DateTimeFormatter formato = DateTimeFormatter.ofPattern("dd/MM/yyyy
HH:mm:ss");
	String date = LocalDateTime.now().format(formato);

	cadena = "[" + author + "] [" + author2 + "] [" + date + "]: " + msg;

	write(cadena);
	}

	// Pseudo-sobrecarga de metodo, es un add pero con el código de error
	// Para buscar más facilmente en el log errores.
	public void addE(String author,String msg){
	add(author,"[[ERROR]]--"+ msg + "--[[ERROR]]");
	}
}
```

7.1.8 Clase Utilidades [Carpeta Server/misc]

```
package Server.misc;

/*
*   Objetivo: Metodos para funciones sin un contexto especifico y potencialmente
		muy usadas.
 */

import java.util.Random;

public class Utilidades {

	public static int numeroRandom (int minimo,int maximo){
	/*
		OBJ: Devuelve numero entero comprendido entre minimo y maximo
			(ambos inclusive)
		PRE: maximo >= minimo
		POST: -
	*/
	Random random = new Random();
	return random.nextInt((maximo - minimo) + 1) + minimo;
	}

}
```

7.1.9 Clase Interfaz1 [Carpeta Server/GUI]

```
package Server.GUI;

import Server.fabrica_galletas.*;
import java.awt.Color;
import javax.swing.JProgressBar;

/***********************************
 * INTERFAZ PRINCIPAL DEL SERVIDOR
 ***********************************/

public class Interfaz1 extends javax.swing.JFrame implements Runnable {

	Horno[] hornos;
	Cafeteria cafe;
	Repostero[] reposteros;
	Almacen almacen;
	Empaquetador[] empaquetadores;

	//Hilos de animación de cada barra declarados vacios
	// Solucion a que si los crea la interfaz (hilo GUI) se congela toda esta
esperando
	// Relacionado con metodo: animacionBarra
	private Thread hiloAnimacionBarraH1 = null;
	private Thread hiloAnimacionBarraH2 = null;
	private Thread hiloAnimacionBarraH3 = null;


	public Interfaz1(Cafeteria cafe, Repostero[] reposteros, Horno[] hornos,
Empaquetador[] empaquetadores, Almacen almacen) {
	initComponents();

	// Centra la ventana en la pantalla
	setLocationRelativeTo(null);

	this.cafe = cafe;
	this.hornos = hornos;
	this.reposteros = reposteros;
	this.almacen = almacen;
	this.empaquetadores = empaquetadores;
	}

	/*
		OBJ: Devuelve string con los nombres de los reposteros que estan
		descansando pero no usando la cafetera
		PRE: -
		POST: String listo para poner en stdout
	*/
	public String getEstadoCafetera2() {
	String resultado = " ";

	for (Repostero rep : reposteros) {
		if (rep.isDescansando() && !rep.getID().equals(cafe.getIdOcupado())) {
		resultado += rep.getID() + ", ";
		}
	}

	// Eliminar la última coma y espacio si la longitud es mayor que 1
```

```
	// Puramente estetico
	if (resultado.length() > 1) {
		resultado = resultado.substring(0, resultado.length() - 2);
	}

	return resultado;
	}

	/*
		OBJ: Retorna color de las casillas del estado empaquetador
		PRE: -
		POST:
		NOTA:   Hemos hecho esto para no tener que repetir
			constantemente el mismo codigo para 5 casillas
			por cada empaquetador
	*/
	public Color colorTandaEmpaquetador(int indice, int pos) {
	// Se puede quitar si se quiere que cuando transporte se quede en rojo
	boolean blankLlenado = !hornos[indice].isHorneando();


	boolean blankTransporte = true; //Anteriormente:
!empaquetadores[indice].getEstado().equals("Transportando");
	if (empaquetadores[indice].getTanda() >= pos && blankLlenado &&
blankTransporte) {
		return Color.red;
	} else {
		return Color.white;
	}
	}

	/*
		OBJ: Animacion de cada una de las barras del horno comprobando su
situacion
		PRE: -
		POST:
		NOTA: Cada barra es en si un hilo, para que el therad.sleep no afecte
			al hilo principal, en este caso la interfaz completa.
			Si no se hace esto la interfaz va a trompicones y todas las
barras
			van iguales al mismo tiempo.
	*/

	public void animacionBarra(JProgressBar barra, Horno horno, int hornoIndex)
{

	// Verificar si ya existe un hilo para esta barra, y si no, crear uno nuevo
	// Si no se hace esto se crean infinitos hilos una vez salta

	if (horno.isHorneando()) {
		// Comprobar si el hilo ya está en ejecución para no creear uno nuevo
		if ((hornoIndex == 0 && hiloAnimacionBarraH1 == null)
			|| (hornoIndex == 1 && hiloAnimacionBarraH2 == null)
			|| (hornoIndex == 2 && hiloAnimacionBarraH3 == null)) {

		// Crear el hilo solo si no está en ejecución
		Thread animacionThread = new Thread(new Runnable() {
			@Override
			public void run() {
```

```
                        try {
                        // Tiempo total de la animación (8 segundos)
                        int tiempoTotal = 8000;
                        int unidadesPorAvance = 1;  // Aumentar la barra en 1
unidad por vez (a mas unidades menos fluido)
                        int totalAvances = 100 / unidadesPorAvance;  // Número
de avances (100 avances)

                        // Configurar la barra para que empiece en 0
                        barra.setValue(0);
                        barra.setForeground(Color.red);

                        // Ciclo para actualizar la barra de progreso
                        for (int i = 0; i <= 100; i += unidadesPorAvance) {
                              barra.setValue(i);  // Actualiza el valor de la
barra
                              Thread.sleep(tiempoTotal / totalAvances);  //
Espera para cada avance (400 ms) (en este caso)
                        }
                        } catch (InterruptedException ex) {
                        ex.printStackTrace();
                        } finally {
                        // Al terminar, liberamos el hilo para que se pueda
crear uno nuevo la próxima vez
                        if (hornoIndex == 0) {
                              hiloAnimacionBarraH1 = null;
                        }
                        if (hornoIndex == 1) {
                              hiloAnimacionBarraH2 = null;
                        }
                        if (hornoIndex == 2) {
                              hiloAnimacionBarraH3 = null;
                        }
                        }
                  }
            });

            // Iniciar el hilo de animación
            if (hornoIndex == 0) {
                  hiloAnimacionBarraH1 = animacionThread;
            }
            if (hornoIndex == 1) {
                  hiloAnimacionBarraH2 = animacionThread;
            }
            if (hornoIndex == 2) {
                  hiloAnimacionBarraH3 = animacionThread;
            }

            animacionThread.start();
            }
      } else if (horno.isHorneadas()) {
            // Si el horno ha terminado, llenamos la barra (verde)
            barra.setValue(100);  // La barra se llena
            barra.setForeground(Color.green);
      } else if (!horno.isHorneadas() && !horno.isHorneando()) {
            // Se  vacia si esta cargandose
            barra.setValue(0);
      }
      }
```

[@SuppressWarnings("unchecked") Generated Code][1]

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
almacen.comer();
}
private int xMouse, yMouse;
private void jPanel1MousePressed(java.awt.event.MouseEvent evt) {

xMouse = evt.getX();
yMouse = evt.getY();
}

private void jPanel1MouseDragged(java.awt.event.MouseEvent evt) {

int x = evt.getXOnScreen();
int y = evt.getYOnScreen();
setLocation(x - xMouse, y - yMouse);
}

private void botonCerrarActionPerformed(java.awt.event.ActionEvent evt) {

// TODO add your handling code here:
}

private void botonCerrarMouseClicked(java.awt.event.MouseEvent evt) {

System.exit(0);
}

private void botonCerrarMouseEntered(java.awt.event.MouseEvent evt) {


botonCerrar.setBackground(Color.red);
}

private void botonCerrarMouseExited(java.awt.event.MouseEvent evt) {

botonCerrar.setBackground(new Color(72,24,29));
}

private void botonMinMouseClicked(java.awt.event.MouseEvent evt) {

setState(Interfaz1.ICONIFIED);
}

private void botonMinMouseEntered(java.awt.event.MouseEvent evt) {

botonMin.setBackground(new Color(124,24,29));
}

private void botonMinMouseExited(java.awt.event.MouseEvent evt) {


botonMin.setBackground(new Color(72,24,29));
```

[1] Las partes en amarillo representan codigo generado automaticamente por netbeans al crear los objetos de swing de manera gráfica. Hemos decidido no ponerlos para evitar agrandar el documento innecesariamente.

```
        }

        private void botonMinActionPerformed(java.awt.event.ActionEvent evt) {

        // TODO add your handling code here:
        }

        public void run() {
        while (true) {
                estado_cafetera.setText(cafe.getIdOcupado());
                estado_cafetera2.setText(getEstadoCafetera2());

                estado_repostero.setText(reposteros[0].getSituacion());
                estado_repostero2.setText(reposteros[1].getSituacion());
                estado_repostero3.setText(reposteros[2].getSituacion());
                estado_repostero4.setText(reposteros[3].getSituacion());
                estado_repostero5.setText(reposteros[4].getSituacion());


numero_galletas.setText(String.valueOf(hornos[0].getCapacidad_actual()));

numero_galletas2.setText(String.valueOf(hornos[1].getCapacidad_actual()));

numero_galletas3.setText(String.valueOf(hornos[2].getCapacidad_actual()));

                animacionBarra(barraH1, hornos[0], 0);
                animacionBarra(barraH2, hornos[1], 1);
                animacionBarra(barraH3, hornos[2], 2);

                estado_emp.setText(empaquetadores[0].getEstado());
                estado_emp2.setText(empaquetadores[1].getEstado());
                estado_emp3.setText(empaquetadores[2].getEstado());

                tanda1_emp1.setBackground(colorTandaEmpaquetador(0, 1));
                tanda2_emp1.setBackground(colorTandaEmpaquetador(0, 2));
                tanda3_emp1.setBackground(colorTandaEmpaquetador(0, 3));
                tanda4_emp1.setBackground(colorTandaEmpaquetador(0, 4));
                tanda5_emp1.setBackground(colorTandaEmpaquetador(0, 5));

                tanda1_emp2.setBackground(colorTandaEmpaquetador(1, 1));
                tanda2_emp2.setBackground(colorTandaEmpaquetador(1, 2));
                tanda3_emp2.setBackground(colorTandaEmpaquetador(1, 3));
                tanda4_emp2.setBackground(colorTandaEmpaquetador(1, 4));
                tanda5_emp2.setBackground(colorTandaEmpaquetador(1, 5));

                tanda1_emp3.setBackground(colorTandaEmpaquetador(2, 1));
                tanda2_emp3.setBackground(colorTandaEmpaquetador(2, 2));
                tanda3_emp3.setBackground(colorTandaEmpaquetador(2, 3));
                tanda4_emp3.setBackground(colorTandaEmpaquetador(2, 4));
                tanda5_emp3.setBackground(colorTandaEmpaquetador(2, 5));

                estado_almacen.setText(String.valueOf(almacen.getCapacidad_actual()));

                // Descanso minimo para no saturar memoria
                try { Thread.sleep(100); } catch (InterruptedException ex) {
System.out.println(ex); }
        }}
[Variables declaration - do not modify]
```

7.2. Código Completo del Sistema Distribuido

Este apartado consta de las siguientes partes:

- Las anteriormente mencionadas en [**8.1. Código Completo del Sistema Concurrente**] bajo la carpeta madre “Server” al que hay que añadirle la siguiente clase: **FabricaGalletasImpl**.
- La interfaz compartida entre cliente y servidor [**FabricaGalletasRemote**]
- **Main** e **interfaz** del Cliente

7.2.1 Clase FabricaGalletasImpl [Carpeta Server/Main]

```
package Server.Main;

/*****************************************************************************
****
* IMPLEMENTACION DE INTERFAZ RMI (para pasar todos los objs y poner metodos)
*****************************************************************************
***/

import Server.fabrica_galletas.*;
import RMI.FabricaGalletasRemote;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;

public class FabricaGalletasImpl extends UnicastRemoteObject implements
FabricaGalletasRemote {

	private Repostero[] reposteros;
	private Horno[] hornos;
	private Almacen almacen;

	public FabricaGalletasImpl(Repostero[] reposteros, Horno[] hornos, Almacen
almacen) throws RemoteException {
	this.hornos = hornos;
	this.reposteros = reposteros;
	this.almacen = almacen;
	}

	// Reposteros ================================================================
	public int getGalletasGeneradas(int indiceRepostero) throws RemoteException
{
	return reposteros[indiceRepostero].getGalletasProducidas();
	}

	public int getGalletasDesperdiciadas(int indiceRepostero) throws
RemoteException {
	return reposteros[indiceRepostero].getGalletasDesperdiciadas();
	}

	public void parar(int indiceRepostero) throws RemoteException {
	reposteros[indiceRepostero].parar();
	}

	public void reanudar(int indiceRepostero) throws RemoteException {
	reposteros[indiceRepostero].reanudar();
	}
```

```
        // Hornos ============================================================
        public boolean isHorneando(int indiceHornos) throws RemoteException {
        return hornos[indiceHornos].isHorneando();
        }

        public boolean isHorneado(int indiceHornos) throws RemoteException {
        return hornos[indiceHornos].isHorneadas();
        }

        public int getGalletasHorneadas(int indiceHornos) throws RemoteException {
        return hornos[indiceHornos].getHistoricoGalletas();
        }

        // Almacen ===========================================================
        public int getCapacidadAlmacen() throws RemoteException {
        return almacen.getCapacidad_actual();
        }

        public int getGalletasConsumidas() throws RemoteException {
        return almacen.getConsumidas();
        }

}
```

7.2.2 Clase MenuCliente [Carpeta Cliente/Main]

```
package ClienteRMI.GUI;

/************************************************
 *            Interfaz de cliente RMI
 ************************************************/


import RMI.FabricaGalletasRemote;
import java.awt.Color;
import java.rmi.RemoteException;
import javax.swing.JButton;
import javax.swing.JProgressBar;

public class MenuCliente extends javax.swing.JFrame implements Runnable {

        // Clase de RMI
        private FabricaGalletasRemote fabrica;

        // Booleano por cada boton de parar/resume
        private boolean toggle1 = false, toggle2 = false, toggle3 = false, toggle4 =
false, toggle5 = false;

        /*
        * Constructor
        */
        public MenuCliente(FabricaGalletasRemote fabrica) {
        initComponents();
        // Centra la ventana en la pantalla
        setLocationRelativeTo(null);
        this.fabrica = fabrica;
        }
```

```
	/*
		OBJ: Animacion cambio de aspecto y texto de un boton en funcion de
		una variable booleana
		PRE: -
		POST:
		NOTA: Parte solo estética
	*/
	private void toggleButton(JButton boton, boolean toggle) {
	if (toggle) {
		boton.setText("RESUME");
		boton.setBackground(Color.red); // Cambiar el fondo a rojo
		boton.setForeground(Color.white);
	} else {
		boton.setText("PARAR");
		boton.setForeground(Color.black);
		boton.setBackground(new Color(204, 204, 204)); // Volver al color
original
	}
	}

	/*
		OBJ: Segun el estado del booleano perteneciente a su boton
parar/resume
		correspondiente para o reanuda el repostero indicado
		PRE: Indice repostero en limites existentes [0-2]
		POST: Repostero parado/reanudado
	*/
	private void toggleAction(boolean toggle, int indiceRep) throws
RemoteException {
	if (toggle) { // Parar
		fabrica.parar(indiceRep);
	} else { // Resume
		fabrica.reanudar(indiceRep);
	}
	}
```

[@SuppressWarnings("unchecked") Generated Code][2]

```
	private void boton_rep3ActionPerformed(java.awt.event.ActionEvent evt) {

	try {
		// Cambio sistematico del valor del booleano (switch)
		toggle3 = !toggle3;
		toggleButton(boton_rep3, toggle3);
		toggleAction(toggle3, 2);
	} catch (RemoteException ex) {
		System.out.println(ex);
	}

	}

	private void boton_rep2ActionPerformed(java.awt.event.ActionEvent evt) {
```

[2] Las partes en amarillo representan codigo generado automaticamente por netbeans al crear los objetos de swing de manera gráfica. Hemos decidido no ponerlos para evitar agrandar el documento innecesariamente.

```
try {
                // Cambio sistematico del valor del booleano (switch)
                toggle2 = !toggle2;
                toggleButton(boton_rep2, toggle2);
                toggleAction(toggle2, 1);
        } catch (RemoteException ex) {
                System.out.println(ex);
        }
        }

        private void boton_rep1ActionPerformed(java.awt.event.ActionEvent evt) {

        try {
                // Cambio sistematico del valor del booleano (switch)
                toggle1 = !toggle1;
                toggleButton(boton_rep1, toggle1);
                toggleAction(toggle1, 0);
        } catch (RemoteException ex) {
                System.out.println(ex);
        }
        }

        private void boton_rep4ActionPerformed(java.awt.event.ActionEvent evt) {

        try {
                // Cambio sistematico del valor del booleano (switch)
                toggle4 = !toggle4;
                toggleButton(boton_rep4, toggle4);
                toggleAction(toggle4, 3);
        } catch (RemoteException ex) {
                System.out.println(ex);
        }
        }

        private void boton_rep5ActionPerformed(java.awt.event.ActionEvent evt) {

        try {
                // Cambio sistematico del valor del booleano (switch)
                toggle5 = !toggle5;
                toggleButton(boton_rep5, toggle5);
                toggleAction(toggle5, 4);
        } catch (RemoteException ex) {
                System.out.println(ex);
        }
        }

        private void botonCerrarMouseClicked(java.awt.event.MouseEvent evt) {

        System.exit(0);
        }

        private void botonCerrarMouseEntered(java.awt.event.MouseEvent evt) {

        botonCerrar.setBackground(Color.red);
        }
```

```
private void botonCerrarMouseExited(java.awt.event.MouseEvent evt) {

botonCerrar.setBackground(new Color(72, 24, 29));
}

private void botonCerrarActionPerformed(java.awt.event.ActionEvent evt) {

// TODO add your handling code here:
}

private void botonMinMouseClicked(java.awt.event.MouseEvent evt) {

setState(MenuCliente.ICONIFIED); // Minimizar
}

private void botonMinMouseEntered(java.awt.event.MouseEvent evt) {

botonMin.setBackground(new Color(124, 24, 29));
}

private void botonMinMouseExited(java.awt.event.MouseEvent evt) {


botonMin.setBackground(new Color(72, 24, 29));
}

private void botonMinActionPerformed(java.awt.event.ActionEvent evt) {

// TODO add your handling code here:
}

private void botonCerrar1MouseClicked(java.awt.event.MouseEvent evt) {

System.exit(0);
}

private void botonCerrar1MouseEntered(java.awt.event.MouseEvent evt) {

botonCerrar.setBackground(Color.red);
}

private void botonCerrar1MouseExited(java.awt.event.MouseEvent evt) {

botonCerrar.setBackground(new Color(72, 24, 29));
}

private void botonCerrar1ActionPerformed(java.awt.event.ActionEvent evt) {

// TODO add your handling code here:
}

private void botonMin1MouseClicked(java.awt.event.MouseEvent evt) {

setState(MenuCliente.ICONIFIED);
}
```

```
private void botonMin1MouseEntered(java.awt.event.MouseEvent evt) {

botonMin.setBackground(new Color(124, 24, 29));
}

private void botonMin1MouseExited(java.awt.event.MouseEvent evt) {


botonMin.setBackground(new Color(72, 24, 29));
}

private void botonMin1ActionPerformed(java.awt.event.ActionEvent evt) {

// TODO add your handling code here:
}

private void botonCerrar2MouseClicked(java.awt.event.MouseEvent evt) {

System.exit(0);
}

private void botonCerrar2MouseEntered(java.awt.event.MouseEvent evt) {


botonCerrar2.setBackground(Color.red);
}

private void botonCerrar2MouseExited(java.awt.event.MouseEvent evt) {

botonCerrar2.setBackground(new Color(72, 24, 29));
}

private void botonCerrar2ActionPerformed(java.awt.event.ActionEvent evt) {

// TODO add your handling code here:
}

private void botonMin2MouseClicked(java.awt.event.MouseEvent evt) {

setState(MenuCliente.ICONIFIED);
}

private void botonMin2MouseEntered(java.awt.event.MouseEvent evt) {

botonMin2.setBackground(new Color(124, 24, 29));
}

private void botonMin2MouseExited(java.awt.event.MouseEvent evt) {


botonMin2.setBackground(new Color(72, 24, 29));
}

private void botonMin2ActionPerformed(java.awt.event.ActionEvent evt) {

// TODO add your handling code here:
}
```

```
        private int xMouse, yMouse;
        private void jPanel1MousePressed(java.awt.event.MouseEvent evt) {

        xMouse = evt.getX();
        yMouse = evt.getY();
        }

        private void jPanel1MouseDragged(java.awt.event.MouseEvent evt) {

        int x = evt.getXOnScreen();
        int y = evt.getYOnScreen();
        setLocation(x - xMouse, y - yMouse);
        }


        /*
                OBJ: Animacion barra de situacion de cada horno
                PRE: indiceHorno dentro de valores existentes en array hornos [0-2]
                POST: -
                NOTA: Como dice el enunciado, 1 segundo de refresco, demasiado para
                        que la animacion de la barra no de problemas de latencia.
                        Se ha decidido que sea una barra estática de un color u otro
                        dependiendo de la situacion del horno.
        */

        public void BarraSimple(JProgressBar barra, int indiceHorno) throws
RemoteException {
        boolean horneando = fabrica.isHorneando(indiceHorno);
        boolean horneadas = fabrica.isHorneado(indiceHorno);
        if (horneando) {
                // Estado horneado en proceso
                barra.setValue(100);
                barra.setForeground(Color.red);
        } else if (horneadas) {
                // Si el horno ha terminado, llenamos la barra
                barra.setValue(100);
                barra.setForeground(Color.green);
        } else if (!horneadas && !horneando) {
                // Se  vacia si esta cargandose
                barra.setValue(0);
        }
        }

        @Override
        public void run() {
        final int SEGUNDOS_UPDATE = 1;
        while (true) {
                try {

estado_almacenadasAlm.setText(String.valueOf(fabrica.getCapacidadAlmacen()));

estado_consumidasAlm.setText(String.valueOf(fabrica.getGalletasConsumidas()));

                estado_gen1.setText(String.valueOf(fabrica.getGalletasGeneradas(0)));
                estado_gen2.setText(String.valueOf(fabrica.getGalletasGeneradas(1)));
                estado_gen3.setText(String.valueOf(fabrica.getGalletasGeneradas(2)));
                estado_gen4.setText(String.valueOf(fabrica.getGalletasGeneradas(3)));
                estado_gen5.setText(String.valueOf(fabrica.getGalletasGeneradas(4)));
```

```
estado_desperdicio1.setText(String.valueOf(fabrica.getGalletasDesperdiciadas(0)));

estado_desperdicio2.setText(String.valueOf(fabrica.getGalletasDesperdiciadas(1)));

estado_desperdicio3.setText(String.valueOf(fabrica.getGalletasDesperdiciadas(2)));

estado_desperdicio4.setText(String.valueOf(fabrica.getGalletasDesperdiciadas(3)));

estado_desperdicio5.setText(String.valueOf(fabrica.getGalletasDesperdiciadas(4)));


estado_horno1.setText(String.valueOf(fabrica.getGalletasHorneadas(0)));

estado_horno2.setText(String.valueOf(fabrica.getGalletasHorneadas(1)));

estado_horno3.setText(String.valueOf(fabrica.getGalletasHorneadas(2)));

              BarraSimple(barraH1, 0);
              BarraSimple(barraH2, 1);
              BarraSimple(barraH3, 2);

              } catch (RemoteException ex) {

              // Ventana de error en caso de perdida de conexión con el objeto
remoto
              javax.swing.JOptionPane.showMessageDialog(this, "Se perdió la conexión
con el servidor. La aplicación se cerrará.", "Error de conexion RMI",
javax.swing.JOptionPane.ERROR_MESSAGE);

              // Salida de la aplicación
              System.exit(0);
              }
              // Descanso minimo para no saturar memoria
              try { Thread.sleep(SEGUNDOS_UPDATE * 1000); }catch
(InterruptedException ex) {}
       }
       }
```

[Variables declaration - do not modify]

7.2.3 Clase Main [Carpeta Cliente/Main]

```
package ClienteRMI.Main;

/*
*   Objetivo: Clase principal del cliente encargada de recibir el objeto
              remoto y de iniciar la interfaz con él
 */

import ClienteRMI.GUI.MenuCliente;
import RMI.FabricaGalletasRemote;

import java.rmi.Naming;
import java.net.MalformedURLException;
```

```
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
```

```
public class Main {

	public static void main(String[] args) throws NotBoundException,
MalformedURLException, RemoteException {

	// Objeto remoto
	FabricaGalletasRemote fabrica = (FabricaGalletasRemote)
Naming.lookup("//localhost/fabrica");

	// Instancia de una interfaz
	MenuCliente menuCliente = new MenuCliente(fabrica);
	menuCliente.setVisible(true);
	Thread thread = new Thread(menuCliente);
	thread.start();
	}
}
```

7.2.4 Clase FabricaGalletasRemote [Carpeta RMI]

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

/**
 * Interfaz compartida entre Cliente y Servidor
 * (Por eso está en un paquete separado del resto)
 */
public interface FabricaGalletasRemote extends Remote {


	// Funciones de repostero
	//NOTA: Trabaja sobre array reposteros, de ahi el indice
	int getGalletasGeneradas(int indiceRepostero) throws RemoteException;
	int getGalletasDesperdiciadas(int indiceRepostero)throws RemoteException;
	void parar(int indiceRepostero) throws RemoteException;
	void reanudar(int indiceRepostero) throws RemoteException;

	// Funciones de hornos
	//NOTA: Trabaja sobre array hornos, de ahi el indice
	int getGalletasHorneadas(int indiceHornos) throws RemoteException;
	boolean isHorneado(int indiceHornos) throws RemoteException;
	boolean isHorneando(int indiceHornos) throws RemoteException;

	// Funciones de almacen
	int getCapacidadAlmacen() throws RemoteException;
	int getGalletasConsumidas() throws RemoteException;

}
```