Informe del Proyecto: Simulación de un Restaurante

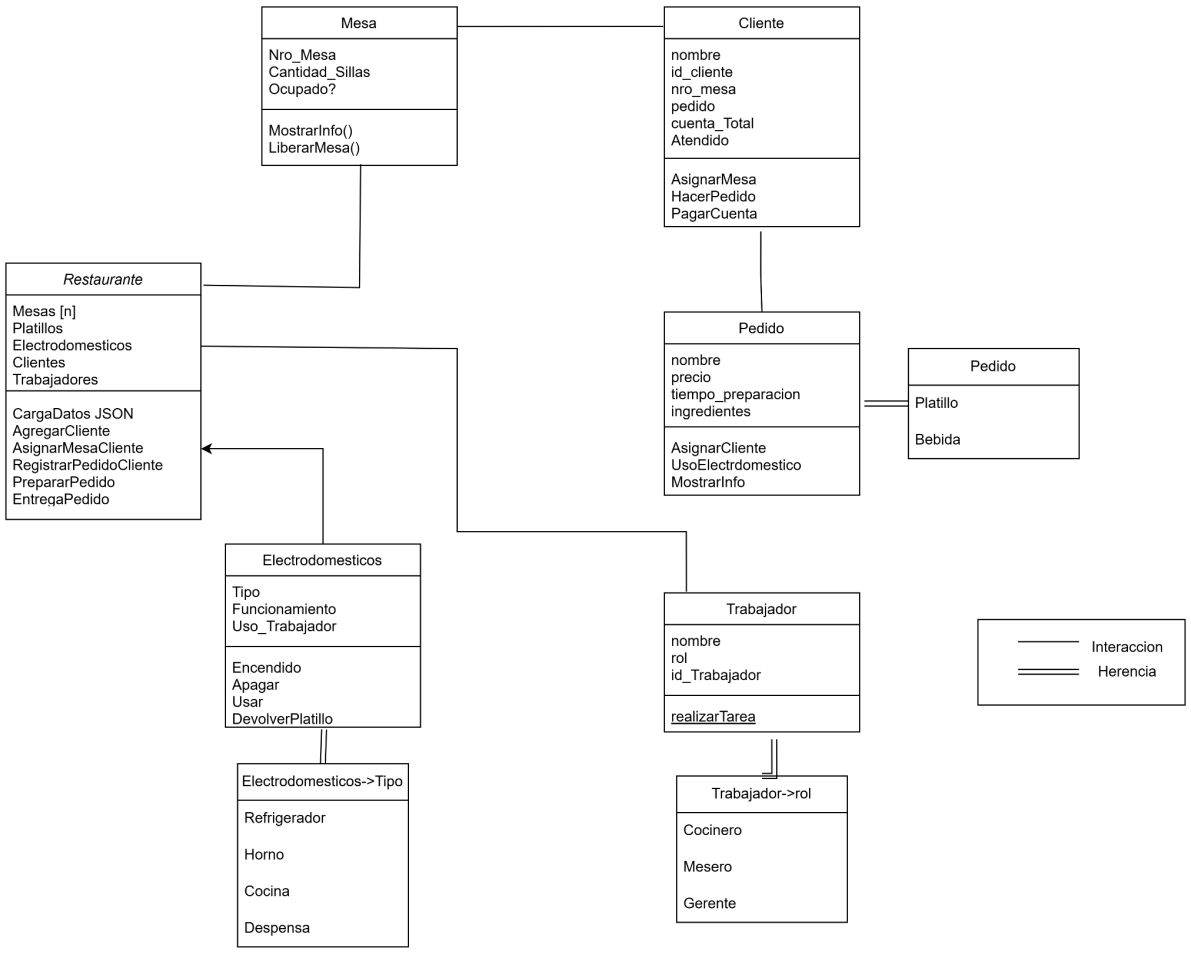
Camila Elena Cossio Tirado
Edson Alejandro Pardo Tambo

Este proyecto final buscó aplicar los cuatro temas principales del curso en una simulación de restaurante. Se enfocó en la Programación Orientada a Objetos (POO), Estructuras de Datos, Algoritmos y Concurrencia para modelar las acciones de Clientes, Cocineros y Meseros. El desafío principal radicó en coordinar sus actividades independientes y simultáneas, garantizando un uso seguro y ordenado de los recursos compartidos.

La clase Restaurante actuó como el núcleo central de la simulación. Su rol fue iniciar y gestionar a los trabajadores, mesas, colas de pedidos y el flujo completo del servicio al cliente. Este enfoque permitió demostrar cómo los conceptos teóricos se aplican en un escenario práctico, evidenciando la importancia de la coordinación y la gestión eficiente en sistemas concurrentes.

1. Diagrama Simplificado de Clases

Este diagrama muestra las relaciones más importantes entre las clases del sistema.



2. Descripción de Estructuras y Algoritmos

El proyecto utiliza estructuras de datos estándar de C++ y algoritmos fundamentales para gestionar el estado y el flujo de la simulación.

Estructuras de Datos

- **std::vector<std::unique_ptr<T>>**: Es la estructura principal para almacenar los recursos del restaurante como mesas, trabajadores y electrodomésticos. Su uso es clave para los siguientes propósitos:
    - **Gestión de memoria automática**: No es necesario invocar a delete para liberar la memoria de los objetos creados.
    - **Polimorfismo**: Permite almacenar punteros de clases derivadas (como Cajero o Cocinero) en un contenedor de la clase base (Trabajador).
    - **Manejo de Mutex**: Facilita el almacenamiento de objetos que no se pueden mover, como aquellos que contienen un std::mutex, dentro de un vector.
- **std::queue<Pedido*>**: Esta estructura es el núcleo del patrón productor-consumidor -implementado en la simulación. Se utilizan dos colas distintas:
    - **pedidosPendientes**: Los clientes actúan como productores añadiendo pedidos a esta cola, mientras que los cocineros actúan como consumidores extrayéndolos para su preparación.
    - **pedidosListos**: Los cocineros son los productores que añaden pedidos ya terminados, y los meseros son los consumidores que los retiran para entregarlos a los clientes.

Algoritmos

- **Flujo de simulación de eventos discretos**: Este es el "algoritmo" principal que organiza la simulación, orquestado mediante hilos y mecanismos de sincronización.
- **Búsqueda Lineal**: Se emplean bucles for para realizar búsquedas simples. Ejemplos de su uso incluyen el método obtenerCajero() para encontrar al trabajador con ese rol y asignarMesaACliente() para localizar la primera mesa disponible.
- **Patrón Productor-Consumidor**: Es el patrón de concurrencia más relevante del proyecto. Se implementa dos veces a través de las colas de pedidos para coordinar el trabajo entre clientes y cocineros, y luego entre cocineros y meseros.

3. Estrategia de Concurrencia y Ejemplos

La estrategia de concurrencia se centra en modelar cada agente activo (cliente, cocinero, mesero) como un std::thread independiente. Los recursos compartidos entre estos hilos se protegen mediante primitivas de sincronización para evitar conflictos.

Recursos Compartidos Clave

- La **Caja/Atención del Cajero**, protegida por el mutex cajero->caja_mtx.

- La cola de **Pedidos Pendientes**, protegida por restaurante->mtxPedidosPendientes.
- La cola de **Pedidos Listos**, protegida por restaurante->mtxPedidosListos.
- Cada **Mesa** individual, que está protegida por su propio mtx interno.

Mecanismos y Ejemplos

- **std::mutex y std::lock_guard**: Se utilizan para asegurar el acceso exclusivo a un recurso y prevenir condiciones de carrera (race conditions). El std::lock_guard es particularmente útil porque libera el mutex de forma automática cuando sale del ámbito, previniendo bloqueos permanentes.

    - **Ejemplo**: Un cliente debe bloquear la caja del cajero antes de poder realizar y pagar su pedido.

C++
```
// Dentro de Cliente::esperando()
{
   std::lock_guard<std::mutex> lock(cajero->caja_mtx); // Bloquea la caja
   hacerPedido();
   cajero->cobrarPedido(pedidoActual);
} // El mutex se libera aquí, el siguiente cliente puede pagar.
```

- **std::condition_variable**: Se usa para lograr una comunicación eficiente entre los hilos productores y consumidores. Este mecanismo evita el "busy-waiting", que es el gasto innecesario de CPU por parte de los hilos consumidores que revisan constantemente si hay trabajo disponible.

    - **Ejemplo**: Un cocinero espera a que un cliente encole un pedido en la cola de pedidosPendientes.

C++
```
// Dentro de Cocinero::trabajar()
std::unique_lock<std::mutex> lock(restaurante->mtxPedidosPendientes);
// El cocinero se "duerme" hasta que un cliente le notifique.
// La variable de condición libera el mutex mientras duerme.
restaurante->cvPedidosPendientes.wait(lock, [this] {
   return !restaurante->pedidosPendientes.empty() || !restaurante->simulacionActiva;
});
// Cuando despierta, el cocinero ya tiene el bloqueo y puede tomar el pedido.
pedido = restaurante->pedidosPendientes.front();
```

- Cuando un cliente añade un nuevo pedido, invoca a restaurante->cvPedidosPendientes.notify_one(); para "despertar" a uno de los cocineros en espera. Este mismo patrón de notificación se aplica entre los cocineros (productores) y los meseros (consumidores) con la cola de pedidosListos.