UNIVERSIDAD DE CÓRDOBA

FACULTAD DE INGENIERÍA

INGENIERÍA DE SISTEMAS

Programacion II

Presenta:

Dober Hoyos Ramos

Leonardo Argumedo

Docente:

Alberto Manuel Paternina Leon

Universidad de Cordoba
Monteria-Cordoba
2025

INFORME TÉCNICO

Sistema de Gestión de Atención al Cliente TechClassUC

Asignatura: Programación II
Proyecto: Sistema de Gestión con Estructuras de Datos

---

1. INTRODUCCIÓN

1.1 Contexto del Proyecto

El presente informe documenta la implementación del Sistema de Gestión de Atención al Cliente para **TechClassUC**, empresa que ofrece servicios de soporte técnico y mantenimiento.

1.2 Objetivo del Sistema

Automatizar la gestión del flujo de atención mediante la implementación de tres estructuras de datos fundamentales:

- **Cola (ArrayDeque)**: Gestión de clientes en espera
- **Lista (LinkedList)**: Historial de clientes atendidos
- **Pila (Stack)**: Registro de acciones con capacidad de deshacer

---

2. ARQUITECTURA DEL SISTEMA

2.1 Patrón Modelo-Vista-Controlador (MVC)

El sistema implementa el patrón arquitectónico MVC, que separa las responsabilidades en tres capas:

MODELO (`modelo` package)

Clases:

- `Cliente.java`: Entidad que representa a un cliente del sistema
- `RegistroDeAcciones.java`: Entidad que representa una acción realizada
- `SistemaDeGestion.java`: Lógica de negocio y gestión de estructuras de datos

Responsabilidades:

- Definir las entidades del dominio
- Administrar las estructuras de datos (ArrayDeque, LinkedList, Stack)
- Implementar la lógica de negocio
- Gestionar el estado del sistema

VISTA (`vista` package)

Clase:

- `VentanaTechClassUC.java`: Interfaz gráfica de usuario con 4 pestañas

Responsabilidades:

- Presentar información al usuario
- Capturar eventos de interfaz
- Mostrar resultados de operaciones
- No contiene lógica de negocio

CONTROLADOR (`controlador` package)

Clase:

- `MVCTechClassUC.java`: Coordinador entre modelo y vista

Responsabilidades:

- Procesar eventos de la interfaz
- Invocar métodos del modelo
- Actualizar la vista con resultados
- Validar datos de entrada
- Coordinar el flujo de la aplicación

MAIN (`proyectotechclassuc` package)

Clase:

- `ProyectoTechClassUC.java`: Punto de entrada de la aplicación

java

```
public static void main(String[] args) {
    VentanaTechClassUC ventana = new VentanaTechClassUC();
    MVCTechClassUC controlador = new MVCTechClassUC(ventana);
    controlador.iniciar();
}
```

2.2 Diagrama de Flujo General

Usuario → Vista (Swing) → Controlador (MVC) → Modelo (Sistema) → Estructuras de Datos

---

3. ESTRUCTURAS DE DATOS IMPLEMENTADAS

3.1 COLA (ArrayDeque) - Gestión de Clientes en Espera

Estructura Utilizada

java

```
private ArrayDeque<Cliente> colaClientes;
```

Características

- **Tipo:** Cola FIFO (First In, First Out)
- **Implementación:** `java.util.ArrayDeque`
- **Capacidad:** Dinámica (redimensionamiento automático)

Operaciones Implementadas

1. Agregar Cliente (offer)

java

```
public void agregarCliente(Cliente cliente) {
    colaClientes.offer(cliente);  // O(1)
    pilaAcciones.push(new RegistroDeAcciones("agregar", cliente));
}
```

- **Complejidad:** O(1) - Inserción al final
- **Propósito:** Agregar nuevos clientes al final de la cola
- **Uso:** Cuando un cliente llega a recepción

2. Atender Cliente (poll)

java

```
public Cliente atenderCliente() {
    if (colaClientes.isEmpty()) {
        return null;
    }
    Cliente cliente = colaClientes.poll();  // O(1)
    cliente.setHoraAtencion(LocalDateTime.now());
    clienteEnAtencion = cliente;
    historialAtendidos.add(cliente);
    pilaAcciones.push(new RegistroDeAcciones("atender", cliente));
    return cliente;
```

}

- **Complejidad:** O(1) - Extracción del frente
- **Propósito:** Extraer el primer cliente para diagnóstico
- **Uso:** Cuando se presiona "CONTINUAR" en Recepción

3. Eliminar Cliente Específico (remove por ID)

java
```
public boolean eliminarClienteDeCola(String id) {
    Iterator<Cliente> iterator = colaClientes.iterator();
    while (iterator.hasNext()) {
        Cliente c = iterator.next();
        if (c.getId().equals(id)) {
            iterator.remove();  // O(n)
            pilaAcciones.push(new RegistroDeAcciones("eliminar", c));
            return true;
        }
    }
    return false;
}
```

- **Complejidad:** O(n) - Búsqueda lineal
- **Propósito:** Eliminar un cliente específico antes de ser atendido
- **Uso:** Cuando se selecciona un cliente y se presiona "ELIMINAR"

Justificación Técnica

Se eligió `ArrayDeque` sobre `LinkedList` como cola porque:

1. **Rendimiento superior:** Mejor localidad de caché
2. **Menor overhead de memoria:** No almacena referencias next/prev en cada nodo
3. **Operaciones O(1):** offer() y poll() son constantes
4. **Redimensionamiento eficiente:** Duplica capacidad cuando es necesario

Integración en el Sistema

- La cola se visualiza en la **tabla de la pestaña RECEPCIÓN**
- Cada fila representa un cliente en espera
- El orden de la tabla refleja el orden FIFO de la cola
- El contador "Clientes en espera" muestra `colaClientes.size()`

3.2 LISTA (LinkedList) - Historial de Clientes Atendidos

Estructura Utilizada
java

```
private LinkedList<Cliente> historialAtendidos;
```

Características

- **Tipo:** Lista doblemente enlazada
- **Implementación:** `java.util.LinkedList`
- **Ordenamiento:** Secuencial por orden de atención

Operaciones Implementadas

1. Agregar al Historial (add)

java
```
// En el método atenderCliente()
historialAtendidos.add(cliente);  // O(1)
```

- **Complejidad:** O(1) - Inserción al final
- **Propósito:** Registrar cliente atendido
- **Momento:** Automático al finalizar diagnóstico

2. Búsqueda por ID (iteración)

java
```
public Cliente buscarPorId(String id) {
    for (Cliente c : historialAtendidos) {  // O(n)
        if (c.getId().equals(id)) {
            return c;
        }
    }
    return null;
}
```

- **Complejidad:** O(n) - Búsqueda lineal
- **Propósito:** Localizar cliente específico en historial
- **Uso:** Filtro por ID en pestaña INFORME

3. Búsqueda por Tipo de Solicitud (filtrado)

java
```
public LinkedList<Cliente> buscarPorTipoSolicitud(String tipo) {
    LinkedList<Cliente> resultado = new LinkedList<>();
    for (Cliente c : historialAtendidos) {  // O(n)
        if (c.getTipoSolicitud().equalsIgnoreCase(tipo)) {
            resultado.add(c);
        }
    }
    return resultado;
```

```
}
```

- **Complejidad:** O(n) - Recorrido completo
- **Propósito:** Filtrar clientes por categoría
- **Uso:** ComboBox de filtros en INFORME

4. Cálculo de Estadísticas (recorrido)

java

```
public double getPromedioTiempoAtencion() {
    if (historialAtendidos.isEmpty()) {
        return 0.0;
    }

    long totalMinutos = 0;
    int count = 0;

    for (Cliente c : historialAtendidos) {  // O(n)
        if (c.getHoraAtencion() != null) {
            Duration duracion = Duration.between(
                c.getHoraLlegada(),
                c.getHoraAtencion()
            );
            totalMinutos += duracion.toMinutes();
            count++;
        }
    }

    return count > 0 ? (double) totalMinutos / count : 0.0;
}
```

- **Complejidad:** O(n) - Recorrido completo
- **Propósito:** Calcular tiempo promedio de atención
- **Uso:** Estadísticas en pestaña INFORME

5. Eliminación de Cliente (remove)

java

```
// En deshacerUltimaAccion() cuando tipo = "finalizar"
historialAtendidos.remove(cliente);  // O(n)
```

- **Complejidad:** O(n) - Búsqueda y eliminación
- **Propósito:** Revertir finalización de atención
- **Uso:** Función "DESHACER"

Justificación Técnica

Se eligió `LinkedList` para el historial porque:

1. **Inserción eficiente:** O(1) al final para agregar atendidos
2. **Flexibilidad:** Fácil eliminación en cualquier posición (para deshacer)
3. **Iteración frecuente:** El historial se recorre constantemente para búsquedas y estadísticas
4. **No requiere acceso aleatorio:** No necesitamos get(index) frecuente

Integración en el Sistema

- El historial se muestra en la **pestaña INFORME**
- Permite filtrado por tipo y búsqueda por ID
- Genera estadísticas: total atendidos y tiempo promedio
- Cada cliente incluye: ID, nombre, tipo, prioridad, problema, diagnóstico, tiempos

---

3.3 PILA (Stack) - Registro de Acciones y Sistema Deshacer

Estructura Utilizada

java

```
private Stack<RegistroDeAcciones> pilaAcciones;
```

Características

- **Tipo:** Pila LIFO (Last In, First Out)
- **Implementación:** `java.util.Stack`
- **Propósito:** Historial de operaciones y funcionalidad "Deshacer"

Tipos de Acciones Registradas

1. **"agregar"**: Cuando un cliente se agrega a la cola
2. **"eliminar"**: Cuando un cliente se elimina de la cola
3. **"atender"**: Cuando un cliente pasa de cola a diagnóstico
4. **"finalizar"**: Cuando se completa la atención y se guarda el diagnóstico

Operaciones Implementadas

1. Registrar Acción (push)

java

```
// Se ejecuta automáticamente en cada operación del sistema
pilaAcciones.push(new RegistroDeAcciones("agregar", cliente));  // O(1)
```

- **Complejidad:** O(1) - Inserción en el tope
- **Propósito:** Guardar cada acción realizada
- **Momento:** Automático en agregar, eliminar, atender, finalizar

2. Consultar Última Acción (peek)

java
```
public RegistroDeAcciones getUltimaAccion() {
    if (pilaAcciones.isEmpty()) {
        return null;
    }
    return pilaAcciones.peek();  // O(1)
}
```

- **Complejidad:** O(1) - Consulta del tope sin remover
- **Propósito:** Ver la última acción sin deshacerla
- **Uso:** Validación antes de deshacer

3. Deshacer Última Acción (pop)

java
```
public void deshacerUltimaAccion() {
    if (pilaAcciones.isEmpty()) {
        return;
    }

    RegistroDeAcciones ultimaAccion = pilaAcciones.pop();  // O(1)
    String tipoAccion = ultimaAccion.getTipoAccion();
    Cliente cliente = ultimaAccion.getCliente();

    switch (tipoAccion) {
        case "agregar":
            colaClientes.remove(cliente);  // Deshace agregar
            break;

        case "atender":
            historialAtendidos.remove(cliente);
            colaClientes.offerFirst(cliente);  // Devuelve a la cola
            clienteEnAtencion = null;
            break;

        case "eliminar":
            colaClientes.offer(cliente);  // Restaura cliente eliminado
            break;

        case "finalizar":
            historialAtendidos.remove(cliente);
            clienteEnAtencion = cliente;  // Vuelve a diagnóstico
            break;
    }
}
```

- **Complejidad:** O(1) para pop, O(n) para operaciones de reversión
- **Propósito:** Revertir la última operación realizada
- **Uso:** Botones "DESHACER" en Recepción y Diagnóstico

4. Obtener Historial Completo (conversión)

java
```
public List<RegistroDeAcciones> getAccionesEnOrdenInverso() {
    List<RegistroDeAcciones> acciones = new ArrayList<>(pilaAcciones);
    Collections.reverse(acciones);  // O(n)
    return acciones;
}
```

- **Complejidad:** O(n) - Copia y reversión
- **Propósito:** Mostrar traza completa de acciones
- **Uso:** Pestaña "REPORTE DE PROCESOS"

Clase RegistroDeAcciones

java
```
public class RegistroDeAcciones {
    private String tipoAccion;       // "agregar", "eliminar", "atender", "finalizar"
    private Cliente cliente;         // Cliente involucrado
    private LocalDateTime fechaHora;  // Timestamp automático

    public RegistroDeAcciones(String tipoAccion, Cliente cliente) {
        this.tipoAccion = tipoAccion;
        this.cliente = cliente;
        this.fechaHora = LocalDateTime.now();  // Marca temporal
    }
}
```

Flujo de Deshacer según Tipo de Acción

CASO 1: Deshacer "agregar"

```
Estado Inicial: Cliente en cola
        ↓
Usuario agrega cliente
        ↓
[PILA] push("agregar", cliente)
[COLA] offer(cliente)
        ↓
Usuario presiona DESHACER
        ↓
[PILA] pop() → acción "agregar"
[COLA] remove(cliente)
        ↓
```

Estado Final: Cliente eliminado de cola

CASO 2: Deshacer "atender"

Estado Inicial: Cliente en diagnóstico
↓
Usuario envió cliente a diagnóstico
↓
[PILA] push("atender", cliente)
[COLA] poll() → cliente extraído
[HISTORIAL] add(cliente)
clienteEnAtencion = cliente
↓
Usuario presiona DESHACER
↓
[PILA] pop() → acción "atender"
[HISTORIAL] remove(cliente)
[COLA] offerFirst(cliente) → vuelve al inicio
clienteEnAtencion = null
↓
Estado Final: Cliente de vuelta en cola

CASO 3: Deshacer "eliminar"

Estado Inicial: Cliente eliminado
↓
Usuario eliminó cliente de cola
↓
[PILA] push("eliminar", cliente)
[COLA] remove(cliente)
↓
Usuario presiona DESHACER
↓
[PILA] pop() → acción "eliminar"
[COLA] offer(cliente) → restaura cliente
↓
Estado Final: Cliente restaurado en cola

CASO 4: Deshacer "finalizar"

Estado Inicial: Cliente en historial
↓
Usuario finalizó atención con diagnóstico
↓
[PILA] push("finalizar", cliente)
[HISTORIAL] add(cliente)
clienteEnAtencion = null

```
      ↓
Usuario presiona DESHACER
      ↓
[PILA] pop() → acción "finalizar"
[HISTORIAL] remove(cliente)
clienteEnAtencion = cliente → vuelve a diagnóstico
      ↓
Estado Final: Cliente de vuelta en diagnóstico
```

Justificación Técnica

Se eligió `Stack` porque:

1. **Orden LIFO natural:** La última acción debe deshacerse primero
2. **Operaciones O(1):** push() y pop() son constantes
3. **Simplicidad:** API clara y específica para pilas
4. **Historial completo:** Se mantienen todas las acciones para auditoría

Integración en el Sistema

- La pila se visualiza en **"REPORTE DE PROCESOS"**
- Cada acción incluye: tipo, cliente, fecha/hora
- Botones "DESHACER" en Recepción y Diagnóstico
- Permite reversión ilimitada hasta vaciar la pila
- El controlador sincroniza pila con vista (tabla y áreas de texto)

---

4. INTEGRACIÓN DE LAS TRES ESTRUCTURAS

4.1 Flujo Completo de un Cliente

```
┌──────────────────────────────────────────────────────────────┐
│                CICLO DE VIDA DE UN CLIENTE                   │
└──────────────────────────────────────────────────────────────┘

1. LLEGADA (Recepción)
   Usuario: Ingresa datos del cliente
   Sistema:
   ├─ Crea objeto Cliente con LocalDateTime.now()
   ├─ [COLA] colaClientes.offer(cliente)
   ├─ [PILA] pilaAcciones.push("agregar", cliente)
   └─ [VISTA] Actualiza tabla de espera

2. LLAMADO A DIAGNÓSTICO (Recepción → Diagnóstico)
   Usuario: Presiona "CONTINUAR"
```

```
   Sistema:
   ├─ [COLA] cliente = colaClientes.poll()  ← Extrae primero de la cola
   ├─ [LISTA] historialAtendidos.add(cliente)
   ├─ [PILA] pilaAcciones.push("atender", cliente)
   ├─ clienteEnAtencion = cliente
   ├─ cliente.setHoraAtencion(LocalDateTime.now())
   └─ [VISTA] Actualiza área de diagnóstico

3. FINALIZACIÓN (Diagnóstico)
   Usuario: Ingresa diagnóstico y presiona "ATENDER CLIENTE"
   Sistema:
   ├─ cliente.setDiagnostico(texto)
   ├─ Verifica: if (!historialAtendidos.contains(cliente))
   │            historialAtendidos.add(cliente)
   ├─ [PILA] pilaAcciones.push("finalizar", cliente)
   ├─ clienteEnAtencion = null
   └─ [VISTA] Actualiza informe y estadísticas

4. CONSULTA (Informe)
   Usuario: Filtra o busca en historial
   Sistema:
   ├─ [LISTA] Recorre historialAtendidos
   ├─ Aplica filtros (tipo o ID)
   ├─ Calcula estadísticas (promedio tiempo)
   └─ [VISTA] Muestra resultados
```

4.2 Interacción entre Estructuras

Operación: Agregar Cliente

java

```java
// Controlador
private void agregarCliente() {
    // 1. Validar datos de entrada
    String id = vista.getCampoId().getText().trim();
    String nombre = vista.getCampoNombre().getText().trim();
    // ... más validaciones

    // 2. Crear cliente
    Cliente nuevoCliente = new Cliente(id, nombre, tipo, prioridad, problema, fecha);

    // 3. Agregar al MODELO (activa ArrayDeque y Stack)
    sistema.agregarCliente(nuevoCliente);
    // Dentro del modelo:
    //   colaClientes.offer(nuevoCliente);  ← COLA
    //   pilaAcciones.push(...);            ← PILA

    // 4. Actualizar VISTA
```

```
    modeloTabla.addRow(new Object[]{...});
    actualizarAreaEspera();       // Muestra colaClientes.size()
    actualizarInformeAcciones();  // Muestra pilaAcciones
}
```

Operación: Atender Cliente

java

```
// Controlador
private void continuarADiagnostico() {
    // 1. Validar que no haya cliente en atención
    if (sistema.getClienteEnAtencion() != null) {
        JOptionPane.showMessageDialog(...);
        return;
    }

    // 2. Extraer de COLA y agregar a LISTA
    Cliente cliente = sistema.atenderCliente();
    // Dentro del modelo:
    //   Cliente c = colaClientes.poll();     ← COLA (extrae)
    //   historialAtendidos.add(c);           ← LISTA (agrega)
    //   pilaAcciones.push("atender", c);     ← PILA (registra)

    // 3. Actualizar VISTA
    modeloTabla.removeRow(0);  // Quita de tabla de espera
    actualizarAreaDiagnostico(cliente);
    actualizarAreaEspera();
    actualizarInformeAcciones();
}
```

Operación: Deshacer

java

```
// Controlador
private void deshacerAccion() {
    // 1. Obtener última acción de PILA
    RegistroDeAcciones ultimaAccion = sistema.getUltimaAccion();
    if (ultimaAccion == null) return;

    String tipo = ultimaAccion.getTipoAccion();
    Cliente cliente = ultimaAccion.getCliente();

    // 2. Revertir en MODELO (afecta COLA, LISTA y PILA)
    sistema.deshacerUltimaAccion();
    // Dentro del modelo:
    //   pilaAcciones.pop();              ← PILA (extrae acción)
    //   switch(tipo):
    //     "agregar"   → colaClientes.remove(cliente)
```

```
    //     "atender"   → colaClientes.offerFirst(cliente) + historialAtendidos.remove(cliente)
    //     "eliminar"  → colaClientes.offer(cliente)
    //     "finalizar" → historialAtendidos.remove(cliente)

    // 3. Actualizar VISTA según tipo
    switch (tipo) {
        case "agregar":
            // Quitar de tabla
            for (int i = 0; i < modeloTabla.getRowCount(); i++) {
                if (modeloTabla.getValueAt(i, 0).equals(cliente.getId())) {
                    modeloTabla.removeRow(i);
                    break;
                }
            }
            break;
        case "atender":
            // Devolver a tabla en primera posición
            modeloTabla.insertRow(0, new Object[]{...});
            vista.getAreaDeDiagnostico().setText("No hay cliente...");
            break;
        // ... otros casos
    }

    actualizarAreaEspera();
    actualizarInformeAcciones();
    actualizarReporteAtendidos();
}
```

4.3 Sincronización Modelo-Vista

El controlador mantiene la consistencia entre las estructuras de datos y la interfaz:

Estructura	Vista Asociada	Método de Actualización
ArrayDeque (cola)	Tabla en Recepción	`actualizarAreaEspera()` → Muestra `colaClientes.size()`
LinkedList (historial)	Área de texto en Informe	`actualizarReporteAtendidos()` → Itera `historialAtendidos`
Stack (acciones)	Área de texto en Reporte	`actualizarInformeAcciones()` → Convierte Stack a List invertida
Cliente en atención	Área de texto en Diagnóstico	`actualizarAreaDiagnostico(cliente)` → Muestra datos del cliente

---

5. ANÁLISIS DE COMPLEJIDAD ALGORÍTMICA

5.1 Complejidad Temporal

Operación	Estructura	Complejidad	Justificación
Agregar cliente	ArrayDeque	**O(1)**	`offer()` inserta al final en tiempo constante (amortizado)
Atender siguiente	ArrayDeque	**O(1)**	`poll()` extrae del frente en tiempo constante
Eliminar por ID	ArrayDeque	**O(n)**	Requiere recorrido completo con Iterator
Agregar atendido	LinkedList	**O(1)**	`add()` inserta al final en tiempo constante
Buscar por ID	LinkedList	**O(n)**	Recorrido lineal hasta encontrar coincidencia
Filtrar por tipo	LinkedList	**O(n)**	Recorrido completo filtrando elementos
Calcular promedio	LinkedList	**O(n)**	Recorrido completo sumando duraciones
Registrar acción	Stack	**O(1)**	`push()` inserta en el tope instantáneamente
Deshacer acción	Stack	**O(1)** + O(n)	`pop()` es O(1), pero reversión puede ser O(n)
Ver última acción	Stack	**O(1)**	`peek()` consulta el tope sin modificar
Historial completo	Stack	**O(n)**	Convierte Stack a ArrayList y reversa

5.2 Complejidad Espacial

Estructura	Espacio	Justificación
ArrayDeque	O(n)	Array interno que crece dinámicamente, n = clientes en espera
LinkedList	O(m)	Nodos doblemente enlazados, m = clientes atendidos totales
Stack	O(k)	Array interno, k = número de acciones registradas

Total Sistema	**O(n + m + k)**	Suma de todas las estructuras

Nota: En un día típico de operación:

- n ≈ 10-50 clientes en espera simultáneamente
- m ≈ 100-500 clientes atendidos por día
- k ≈ 200-1000 acciones registradas
- **Espacio total estimado:** ~500KB - 2MB

5.3 Optimizaciones Implementadas

1. **ArrayDeque vs LinkedList para Cola:**
    - ArrayDeque tiene mejor localidad de caché
    - Menor overhead por elemento (sin punteros next/prev)
    - Operaciones offer/poll más rápidas en la práctica
2. **Validación Temprana:**

java
```
if (colaClientes.isEmpty()) {
    return null;  // Evita operaciones innecesarias
}
```

3. **Búsqueda con Short-Circuit:**

java
```
for (Cliente c : historialAtendidos) {
    if (c.getId().equals(id)) {
        return c;  // Termina al encontrar
    }
}
```

4. **Reutilización de Objetos Cliente:**
    - Los clientes se mueven entre estructuras sin crear copias
    - Referencias compartidas (mismo objeto en memoria)

---

6. FUNCIONAMIENTO DEL SISTEMA DE REVERSIÓN (UNDO)

6.1 Arquitectura del Sistema Deshacer

El sistema de deshacer se basa en el **patrón Command** implícito, donde cada acción se encapsula en un objeto `RegistroDeAcciones` que contiene toda la información necesaria para revertirla.

Componentes del Sistema Undo

1. **Registro de Acción**

java
```
public class RegistroDeAcciones {
    private String tipoAccion;       // Tipo de operación
    private Cliente cliente;         // Snapshot completo del cliente
    private LocalDateTime fechaHora;  // Timestamp para auditoría
}
```

2. **Pila de Historial**

java
```
private Stack<RegistroDeAcciones> pilaAcciones;
```

3. **Motor de Reversión**

java
```
public void deshacerUltimaAccion() {
    if (pilaAcciones.isEmpty()) return;

    RegistroDeAcciones accion = pilaAcciones.pop();
    // Lógica de reversión según tipo
}
```

6.2 Matriz de Reversión Detallada

Acción Original	Estado Antes	Operación Realizada	Estado Después	Reversión al Deshacer
agregar	Cola vacía	`offer(C1)` → PILA push	C1 en cola	`remove(C1)` → C1 eliminado
atender	C1 en cola	`poll()` → C1 a diag.	C1 en atención	`offerFirst(C1)` → C1 vuelve a cola
eliminar	C1 en cola	`remove(C1)` por ID	C1 eliminado	`offer(C1)` → C1 restaurado
finalizar	C1 en diag.	Guarda diagnóstico	C1 en historial	`remove(C1)` → C1 vuelve a diag.

6.3 Casos de Uso Detallados

ESCENARIO 1: Error en Registro

Situación: Usuario agregó cliente con ID incorrecto

Paso 1: Agregar cliente
  Input: ID="123A", Nombre="Juan"
  COLA: [Cliente{123A, Juan}]
  PILA: [agregar → Cliente{123A}]
  TABLA: Fila con 123A visible

Paso 2: Usuario detecta error y presiona DESHACER
  PILA: pop() → obtiene acción "agregar"
  COLA: remove(Cliente{123A})
  TABLA: removeRow() para fila de 123A

Estado Final: Sistema limpio para reingresar correctamente

ESCENARIO 2: Cliente Llamado por Error

Situación: Se llamó al cliente equivocado a diagnóstico

Paso 1: Estado Inicial
  COLA: [C1, C2, C3] (C1 es el siguiente)

Paso 2: Continuar a Diagnóstico
  COLA: poll() → [C2, C3] (C1 extraído)
  LISTA: add(C1)
  clienteEnAtencion = C1
  PILA: [atender → C1]

Paso 3: Usuario se da cuenta que era C2, presiona DESHACER
  PILA: pop() → obtiene acción "atender"
  LISTA: remove(C1)
  COLA: offerFirst(C1) → [C1, C2, C3] (C1 vuelve al inicio)
  clienteEnAtencion = null

Estado Final: C1 de vuelta en primera posición

ESCENARIO 3: Diagnóstico Guardado Incorrectamente

Situación: Se guardó diagnóstico equivocado

Paso 1: Cliente en diagnóstico
  clienteEnAtencion = C1

Paso 2: Finalizar con diagnóstico
  Input: "Requiere formateo completo"

C1.setDiagnostico("Requiere formateo completo")
LISTA: contains(C1) ? add(C1) : skip
PILA: [finalizar → C1]
clienteEnAtencion = null

Paso 3: Usuario presiona DESHACER
PILA: pop() → obtiene acción "finalizar"
LISTA: remove(C1)
clienteEnAtencion = C1 (vuelve a diagnóstico)
Vista: Restaura área de diagnóstico con datos de C1
Restaura campo diagnóstico con texto anterior

Estado Final: C1 en diagnóstico, usuario puede corregir