Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Ingeniería en Ciencias y Sistemas
Catedrático: Ing. David Estuardo Morales Ajcot
Auxiliar: Kewin Maslovy Patzan Tzun
Curso: Lenguajes formales y de programación
Sección: “B+”

Práctica 1

Nombre:
Alexis José Trujillo Vásquez

Carnet:
202401884

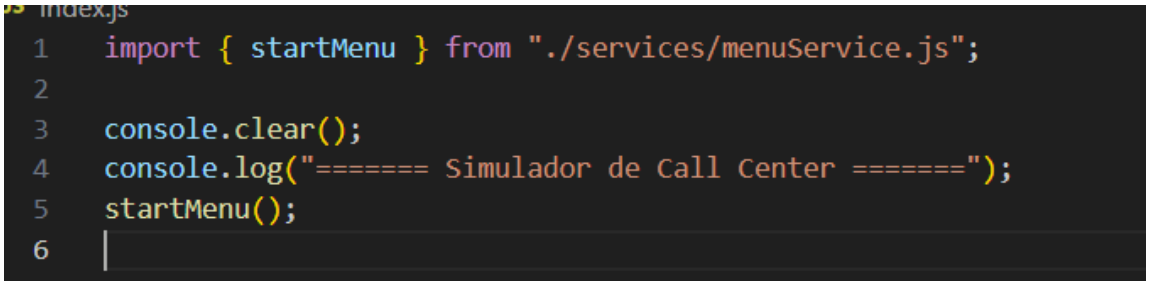
Manual técnico

Este manual proporciona información sobre los aspectos técnicos del programa, explicando en detalle cómo funciona la parte programable y brindando el conocimiento necesario para comprender cómo se ejecutan todas las partes del código.

- Index.js

1. import { startMenu } from "./services/menuService.js";
    - Utiliza la sintaxis de módulos ES6 (habilitada por el campo "type": "module" en package.json) para importar la función startMenu desde services/menuService.js.
    - Al ejecutarse index.js, Node.js resuelve la ruta relativa "./services/menuService.js", carga ese módulo y pone a disposición la función startMenu exportada allí.
2. console.clear();
    - Llama al método incorporado de Node que limpia la consola.
    - Esto borra cualquier salida anterior para que el usuario vea el menú inicial sin “ruido” en pantalla.
3. console.log("======= Simulador de Call Center =======");
    - Imprime un encabezado en la consola para que el usuario sepa que está entrando al simulador.
    - Sirve como presentación visual antes de mostrar las opciones interactivas.
4. startMenu();
    - Invoca la función importada en el primer paso.
    - startMenu crea una interfaz de lectura (readline), muestra el menú de opciones y escucha las entradas del usuario para cargar archivos CSV, generar reportes HTML, ver estadísticas, etc.
    - Mientras el usuario no elija la opción de salida, el menú se vuelve a mostrar y el programa permanece interactivo.

En conjunto, este fragmento prepara la consola, muestra un título y lanza el flujo principal del simulador mediante startMenu.



- menuService.js

- import readline from "readline";
  Carga el módulo nativo readline de Node.js, que permite crear interfaces de entrada/salida por consola (stdin/stdout).
- import { loadCallRecords } from "./fileService.js";
  Importa la función loadCallRecords, responsable de leer un archivo CSV y convertirlo en registros de llamadas.
- import { mostrarHistorialConsola, generarHistorialHTML, generarOperadoresHTML, generarClientesHTML, generarRendimientoHTML, porcentajeClasificacion, llamadasPorEstrellas } from "./reportService.js";
- Trae múltiples funciones de generación de reportes y estadísticas, cada una encargada de una salida específica (HTML, tablas, porcentajes, etc.).

```
import readline from "readline";
import { loadCallRecords } from "./fileService.js";
import {
    mostrarHistorialConsola,
    generarHistorialHTML,
    generarOperadoresHTML,
    generarClientesHTML,
    generarRendimientoHTML,
    porcentajeClasificacion,
    llamadasPorEstrellas
} from "./reportService.js";
```

- **startMenu():**

Punto de entrada del flujo interactivo:

1. let records = [];

Arreglo donde se almacenarán los registros de llamadas cargados desde el CSV.

2. const rl = readline.createInterface({...});

- Configura la interfaz de lectura/escritura con process.stdin y process.stdout.
- rl gestiona las preguntas al usuario y las respuestas en consola.

```
export function startMenu() {
    let records = []; // Array para almacenar los registros de llamadas

    const rl = readline.createInterface({ // Crear interfaz de lectura
        input: process.stdin, // Entrada estándar
        output: process.stdout // Salida estándar
    });
```

- **showMenu()**

Función interna que despliega las opciones disponibles cada vez que se debe mostrar el menú:

```
function showMenu() {
    console.log("\n--- Menú de Opciones ---");
    console.log("1. Cargar registros de llamadas");
    console.log("2. Mostrar historial en consola");
    console.log("3. Exportar historial en HTML");
    console.log("4. Listado de Operadores");
    console.log("5. Listado de Clientes");
    console.log("6. Rendimiento de Operadores");
    console.log("7. Porcentaje de Clasificación");
    console.log("8. Llamadas por Calificación");
    console.log("9. Salir");

    rl.question("Seleccione una opción: ", handleMenuOption);
}
```

- Imprime el listado del menú numerado.
- Llama a rl.question para leer la opción del usuario y delegar su procesamiento a handleMenuOption.

- **handleMenuOption()**

Procesa la selección del menú

```
function handleMenuOption(option) {
    switch(option) {
        case '1':
            rl.question("Ingrese la ruta del archivo CSV (ej: ./data/llamadas.csv): ", (ruta) => { // Solicitar ruta del archivo CSV
                records = loadCallRecords(ruta); // Cargar registros desde el archivo CSV
                if (records.length > 0) { // Verificar si se cargaron registros
                    console.log(`Se cargaron ${records.length} registros desde ${ruta}`);
                }
                showMenu();
            });
            return; // Finalizar la función
        case '2':
        mostrarHistorialConsola(records);
            break;
        case '3':
        generarHistorialHTML(records);
            break;
        case '4':
        generarOperadoresHTML(records);
            break;
        case '5':
        generarClientesHTML(records);
            break;
        case '6':
        generarRendimientoHTML(records);
            break;
        case '7':
            porcentajeClasificacion(records);
            break;
        case '8':
            llamadasPorEstrellas(records);
            break;
        case '9':
            console.log("Saliendo... ¡Adiosito!");
            rl.close();
            return;
```

- Caso '1': Pide la ruta del archivo CSV, carga los registros y actualiza records.
- Casos '2' a '8': Ejecutan la función correspondiente de reportes/estadísticas usando los registros cargados.
- Caso '9': Muestra mensaje de salida, cierra la interfaz y termina la aplicación.
- Default: Maneja opciones no válidas.
- Después de cada acción (excepto al salir), vuelve a llamar a showMenu para mostrar el menú nuevamente.

- **showMenu()**

Se invoca al final de startMenu() para presentar el menú inicial cuando se ejecuta el módulo.

En resumen, este fragmento define el núcleo de la interfaz de consola: presenta un menú, maneja la entrada del usuario y delega cada opción a funciones específicas para cargar datos, generar reportes o finalizar el programa.

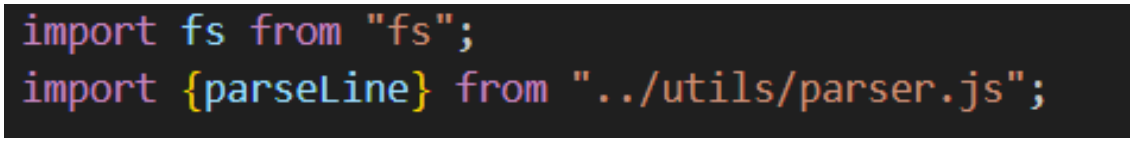
- fileService.js

- import fs from "fs";

Carga el módulo nativo de Node.js para trabajar con el sistema de archivos, en especial para leer el contenido de archivos mediante fs.readFileSync.

- import {parseLine} from "../utils/parser.js";

Trae la función parseLine, encargada de convertir una línea del CSV en un objeto de registro (callRecord) validado.



```
export function loadCallRecords(filePath){
    try{
        const data = fs.readFileSync(filePath, "utf-8"); // Leer el archivo
        const lines = data.split("\n").map(l => l.trim()).filter(l => l.length > 0); // Eliminar líneas vacías
        lines.shift(); // Eliminar encabezado
        return lines.map(parseLine).filter(r => r !== null && r !== undefined); // Analizar líneas y eliminar nulos
    }catch(error){ // Manejar errores
        console.error("Error al cargar el archivo:", error.message); // Imprimir error en consola
        return []; // Retornar arreglo vacío en caso de error
    }
}
```

1. Lectura del archivo

    - fs.readFileSync(filePath, "utf-8") abre el archivo CSV de llamadas y devuelve su contenido como texto.

2. Procesamiento de líneas

    - data.split("\n"): separa el texto en un arreglo de líneas.
    - .map(l => l.trim()): elimina espacios en blanco al inicio y final de cada línea.
    - .filter(l => l.length > 0): descarta líneas vacías para evitar registros incompletos.

3. Remover encabezado

    - lines.shift() extrae la primera fila (encabezado del CSV) para que no se procese como registro.

4. Transformación a registros

    - lines.map(parseLine) aplica parseLine a cada línea, devolviendo objetos callRecord o null.
    - .filter(r => r !== null && r !== undefined) descarta los resultados inválidos o nulos.

5. Manejo de errores

    - Si cualquier operación falla (archivo inexistente, permisos, formato inválido), se captura la excepción, se muestra un mensaje con console.error y se devuelve un arreglo vacío.

loadCallRecords lee un CSV de llamadas, limpia y valida cada línea, convierte las válidas en objetos de registro y retorna todos los registros cargados, manejando de forma segura cualquier error que pudiera ocurrir durante el proceso.

- reportService.js

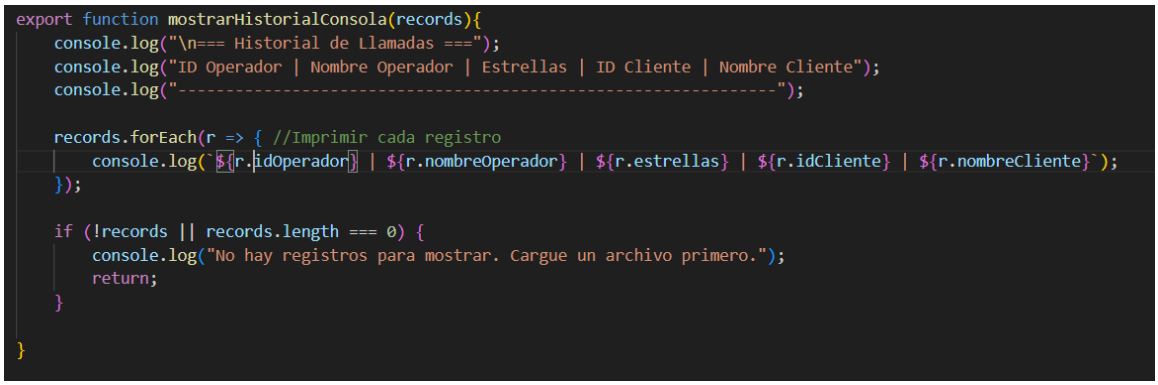
import fs from "fs"; — incorpora el módulo nativo para leer y escribir archivos, utilizado en todas las funciones que generan reportes HTML

mostrarHistorialConsola(records)

1. Imprime un encabezado y la estructura de columnas del historial.

2. Recorre cada registro para mostrar ID y nombre del operador, estrellas y datos del cliente.

3. Si la lista está vacía, advierte al usuario que cargue un archivo antes de continuar.



```
export function mostrarHistorialConsola(records){
    console.log("\n=== Historial de Llamadas ===");
    console.log("ID Operador | Nombre Operador | Estrellas | ID Cliente | Nombre Cliente");
    console.log("-----------------------------------------------------------");

    records.forEach(r => { //Imprimir cada registro
        console.log(`${r.idOperador} | ${r.nombreOperador} | ${r.estrellas} | ${r.idCliente} | ${r.nombreCliente}`);
    });

    if (!records || records.length === 0) {
        console.log("No hay registros para mostrar. Cargue un archivo primero.");
        return;
    }

}
```

generarHistorialHTML(records)

1. Construye una plantilla HTML con una tabla que enumera cada llamada (operador, cliente y calificación).

2. Recorre los registros para añadir filas con sus datos.

3. Garantiza que exista la carpeta ./reportes, escribe el archivo historial.html y notifica su creación.

```
export function generarHistorialHTML(records){
    let html = `
        <head>
            <meta charset="UTF-8">
            <title>Historial de Llamadas</title>
            <style>
                table { border-collapse: collapse; width: 100%; }
                th, td { border: 1px solid #ccc; padding: 8px; text-align: left; }
                th { background-color: #f2f2f2; }
            </style>
        </head>
        <body>
            <h1>Historial de Llamadas</h1>
            <table>
                <tr>
                    <th>ID Operador</th>
                    <th>Nombre Operador</th>
                    <th>Estrellas</th>
                    <th>ID Cliente</th>
                    <th>Nombre Cliente</th>
                </tr>
        `;

    records.forEach(r => {
        html += `
            <tr>
                <td>${r.idOperador}</td>
                <td>${r.nombreOperador}</td>
                <td>${r.estrellas}</td>
                <td>${r.idCliente}</td>
                <td>${r.nombreCliente}</td>
            </tr>
        `;
    });

    html += `
            </table>
```

```
    if (!fs.existsSync("./reportes")) {
        fs.mkdirSync("./reportes");
    }

    fs.writeFileSync("./reportes/historial.html", html);
    console.log("Reporte HTML generado en /reportes/historial.html");
}
```

generarOperadoresHTML(records)

1. Prepara una página HTML para listar operadores únicos.

2. Usa un objeto como mapa para deduplicar operadores por ID.

3. Escribe operadores.html dentro de reportes y confirma la generación del reporte.

```
// 3. Exportar listado de Operadores
export function generarOperadoresHTML(records){
    let html = `
        <!DOCTYPE html>
        <html lang="es">
        <head>
            <meta charset="UTF-8">
            <title>Listado de Operadores</title>
            <style>
                table { border-collapse: collapse; width: 100%; }
                th, td { border: 1px solid #ccc; padding: 8px; text-align: left; }
                th { background-color: #f2f2f2; }
            </style>
        </head>
        <body>
            <h1>Listado de Operadores</h1>
            <table>
                <tr><th>ID Operador</th><th>Nombre Operador</th></tr>
    `;

    const operadores = {};
    records.forEach(r => operadores[r.idOperador] = r.nombreOperador);

    Object.entries(operadores).forEach(([id, nombre]) => {
        html += `<tr><td>${id}</td><td>${nombre}</td></tr>`;
    });

    html += `</table></body></html>`;

    if (!fs.existsSync("./reportes")) fs.mkdirSync("./reportes");
    fs.writeFileSync("./reportes/operadores.html", html);
    console.log("Reporte generado en /reportes/operadores.html");
}
```

generarClientesHTML(records)

1. Genera un HTML con la lista de clientes sin duplicados.

2. Recorre la colección para crear un mapa id donde coloca nombre y rellena la tabla.

Guarda clientes.html y muestra un mensaje de éxito.

```
// 4. Exportar listado de Clientes
export function generarClientesHTML(records){
    let html = `
        <!DOCTYPE html>
        <html lang="es">
        <head>
            <meta charset="UTF-8">
            <title>Listado de Clientes</title>
            <style>
                table { border-collapse: collapse; width: 100%; }
                th, td { border: 1px solid #ccc; padding: 8px; text-align: left; }
                th { background-color: #f2f2f2; }
            </style>
        </head>
        <body>
            <h1>Listado de Clientes</h1>
            <table>
                <tr><th>ID Cliente</th><th>Nombre Cliente</th></tr>
    `;

    const clientes = {};
    records.forEach(r => clientes[r.idCliente] = r.nombreCliente);

    Object.entries(clientes).forEach(([id, nombre]) => {
        html += `<tr><td>${id}</td><td>${nombre}</td></tr>`;
    });

    html += `</table></body></html>`;

    if (!fs.existsSync("./reportes")) fs.mkdirSync("./reportes");
    fs.writeFileSync("./reportes/clientes.html", html);
    console.log("Reporte generado en /reportes/clientes.html");
}
```

generarRendimientoHTML(records)

1. Calcula cuántas llamadas atendió cada operador y almacena el conteo.

2. Construye una tabla HTML con ID, nombre, número de llamadas y porcentaje respecto al total.

3. Exporta el archivo rendimiento.html y reporta su ubicación.

```
export function generarRendimientoHTML(records){
    const total = records.length;
    const conteo = {};

    records.forEach(r => {
        if (!conteo[r.idOperador]) conteo[r.idOperador] = { nombre: r.nombreOperador, llamadas: 0 };
        conteo[r.idOperador].llamadas++;
    });

    let html = `
        <!DOCTYPE html>
        <html lang="es">
        <head>
            <meta charset="UTF-8">
            <title>Rendimiento de Operadores</title>
            <style>
                table { border-collapse: collapse; width: 100%; }
                th, td { border: 1px solid #ccc; padding: 8px; text-align: left; }
                th { background-color: #f2f2f2; }
            </style>
        </head>
        <body>
            <h1>Rendimiento de Operadores</h1>
            <table>
                <tr><th>ID Operador</th><th>Nombre Operador</th><th>Llamadas</th><th>Porcentaje</th></tr>
    `;

    Object.entries(conteo).forEach(([id, info]) => {
        const porcentaje = ((info.llamadas / total) * 100).toFixed(2);
        html += `<tr><td>${id}</td><td>${info.nombre}</td><td>${info.llamadas}</td><td>${porcentaje}%</td></tr>`;
    });

    html += `</table></body></html>`;

    if (!fs.existsSync("./reportes")) fs.mkdirSync("./reportes");
    fs.writeFileSync("./reportes/rendimiento.html", html);
    console.log("Reporte generado en /reportes/rendimiento.html");
```

porcentajeClasificacion(records)

1. Clasifica cada llamada como buena (≥4 estrellas), media (2–3) o mala (≤1) y contabiliza cada categoría.

2. Calcula el porcentaje de cada tipo respecto al total de registros y los muestra en la consola.

```
// 6. Porcentaje de clasificación
export function porcentajeClasificacion(records) {
    let buenas = 0, medias = 0, malas = 0; // Contadores para cada tipo de clasificación
    records.forEach(r => { // Iterar sobre los registros
        if (r.estrellas >= 4) buenas++; // Contar buenas
        else if (r.estrellas >= 2) medias++; // Contar medias
        else malas++; // Contar malas
    });

    const total = records.length; // Total de registros
    console.log("\n=== Porcentaje de Clasificación ===");
    console.log(`Buenas: ${(buenas / total * 100).toFixed(2)}%`); // Calcular porcentaje de buenas
    console.log(`Medias: ${(medias / total * 100).toFixed(2)}%`); // Calcular porcentaje de medias
    console.log(`Malas: ${(malas / total * 100).toFixed(2)}%`); // Calcular porcentaje de malas
}
```

llamadasPorEstrellas(records)

1. Inicializa un arreglo de conteo para las calificaciones de 1 a 5 estrellas.

2. Incrementa el contador correspondiente a la calificación de cada registro.

3. Imprime en consola el número de llamadas que recibió cada nivel de estrellas.

```
// 7. Llamadas por calificación
export function llamadasPorEstrellas(records){
    const conteo = [0, 0, 0, 0, 0, 0];
    records.forEach(r => conteo[r.estrellas]++); // Iterar sobre los registros
    console.log("\n=== Llamadas por Estrellas ===");
    for(let i = 1; i <= 5; i++){ // Iterar sobre las posibles calificaciones
        console.log(`${i} estrellas: ${conteo[i]}`); // Imprimir el conteo de llamadas por calificación
    }
}
```

- callRecord.js

- class callRecord

Contenedor básico de información para cada llamada. Se exporta como la clase principal del modelo.

- Constructor

Recibe cinco parámetros (idOperador, nombreOperador, estrellas, idCliente, nombreCliente) y los asigna a propiedades de instancia, manteniendo juntos los datos del operador, la calificación y el cliente.

```
export default class callRecord{ // Clase para almacenar los registros de llamadas
    constructor(idOperador, nombreOperador, estrellas, idCliente, nombreCliente){ // Constructor de la clase
        this.idOperador = idOperador; // ID del operador
        this.nombreOperador = nombreOperador; // Nombre del operador
        this.estrellas = estrellas; // Estrellas del operador
        this.idCliente = idCliente; // ID del cliente
        this.nombreCliente = nombreCliente; // Nombre del cliente
    }
}
```

- parser.js

1. Importación

- import CallRecord from "../models/callRecord.js";

Trae la clase CallRecord, usada para instanciar objetos representando cada fila del CSV.

2. Definición de la función

```
export function parseLine(line){
    const parts = line.split(",");
```

- Toma una cadena line y la separa por comas, obteniendo un arreglo parts con cada columna.

3. Validación básica

```
if (parts.length < 5) {
    return null;
}
```

- Verifica que la línea tenga al menos cinco columnas; de lo contrario, la considera inválida y devuelve null.

4. Procesamiento de estrellas

```
// separar la columna de estrellas por ;
const estrellasArray = parts[2].split(";");
const estrellasCount = estrellasArray.filter(e => e.trim() === "x").length;
```

- La tercera columna (índice 2) puede tener valores separados por ; para representar estrellas.
- Se cuentan cuántas subcadenas son exactamente "x"—ese número corresponde a la calificación en estrellas (0–5).

5. Construcción del registro

```
    return new CallRecord(
        parseInt(parts[0]),
        parts[1].trim(),
        estrellasCount,   // número entre 0 y 5
        parseInt(parts[3]),
        parts[4].trim()
    );
}
```

- Crea y retorna una instancia de CallRecord con:
  - ID de operador (parseInt(parts[0]))
  - Nombre de operador (parts[1].trim())
  - Número de estrellas (estrellasCount)
  - ID de cliente (parseInt(parts[3]))
  - Nombre de cliente (parts[4].trim())
- Los campos numéricos se convierten con parseInt, y los de texto se limpian con trim para remover espacios innecesarios.

Diagrama de flujo

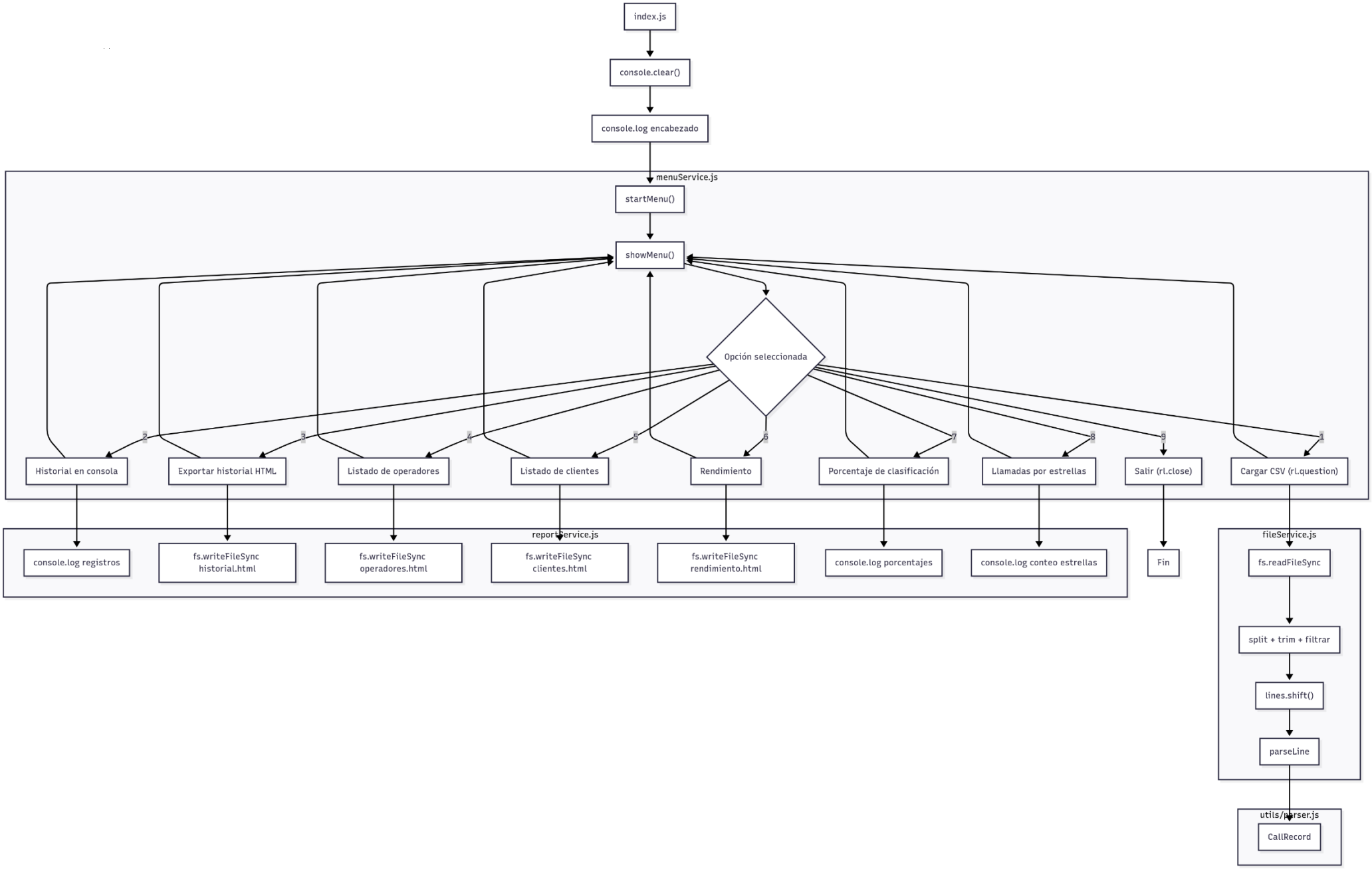


Diagrama de clases