MANUAL Tecnico

Simulador de Callcenter

Mario Rodrigo Balam

Correo electrónico: 2908263140708

Carné: 202200147

Curso: Lenguajes Formales

Sección: A-

Ing Vivian Damaris Campos González

Aux: Carlos Javier Cox Bautista

TABLA DE CONTENIDO

INTRODUCCION

El presente proyecto corresponde a la práctica de Lenguajes Formales y de Programación, en la cual se desarrolló una aplicación de consola en JavaScript que simula la gestión de llamadas de un CallCenter.

La aplicación permite cargar registros desde un archivo CSV, procesarlos con funciones nativas del lenguaje, clasificarlos según su calificación y generar diferentes reportes en consola y en formato HTML utilizando Bootstrap para su presentación.
El propósito de esta práctica es fortalecer el uso de la Programación Orientada a Objetos, el manejo de archivos y la lógica de procesamiento de datos.

Este manual describe los aspectos técnicos de la solución desarrollada para la práctica de Lenguajes Formales y de Programación. Incluye detalles de la arquitectura, clases, funcionamiento y consideraciones adicionales.

OBJETIVO GENERAL

Desarrollar una aplicación en consola que simule el funcionamiento de un Call Center mediante la lectura, análisis y reporte de registros de llamadas, aplicando técnicas de programación orientada a objetos en JavaScript.

OBJETIVOS ESPECIFICOS.

- Implementar estructuras de datos que permitan almacenar operadores, clientes y llamadas.
- Procesar archivos de entrada en formato CSV utilizando únicamente funciones nativas del lenguaje.
- Clasificar las llamadas según su calificación y generar estadísticas globales.
- Fortalecer las habilidades de lógica de programación y manipulación de cadenas.

TECNOLOGIAS UTILIZADAS

IV. Especificación técnicas

- Node.js instalado (v14 o superior)
- Lenguaje: JavaScript (Node.js en consola)
- Estilos: Bootstrap 5 (a través de CDN)
- Manejo de archivos: módulo nativo fs
- Interacción por consola: módulo nativo readline
- Manejo de rutas y nombres de archivo: módulo nativo path

V. Estructura del proyecto

Practica/

```
├── index.js        (flujo principal y menú)
├── callcenter.js   (clases y lógica de negocio)
├── data/llamadas.csv (archivo de entrada)
└── reportes/       (reportes generados en HTML)
```

CLASES

Clase llamada:

- Atributos: idOperador, nombreOperador, estrellas, idCliente, nombreCliente
- Método: clasificacion() → devuelve Buena, Media o Mala

```
// Clase Llamada
class Llamada {
    constructor(idOperador, nombreOperador, estrellas, idCliente, nombreCliente) {
        this.idOperador = idOperador;
        this.nombreOperador = nombreOperador;
        this.estrellas = estrellas; // valor numérico (0-5)
        this.idCliente = idCliente;
        this.nombreCliente = nombreCliente;
    }

    clasificacion() {
        if (this.estrellas >= 4) return "Buena";
        if (this.estrellas >= 2) return "Media";
        return "Mala";
    }
}
```

Clase operador:

- Atributos: id, nombre, llamadas[]

- Método:
calcularRendimiento(total): calcula el porcentaje de llamadas atendidas por este operador respecto al total global.
agregarLlamada(llamada): agrega una llamada al historial del operador.

En esta clase modela a un operador del Call Center, permite registrar las llamadas que atendió y calcular su porcentaje de rendimiento respecto al total de llamadas cargadas en el sistema.

```
// ======================
class Operador {
    constructor(id, nombre) {
        this.id = id;
        this.nombre = nombre;
        this.llamadas = [];
    }

    agregarLlamada(llamada) {
        this.llamadas.push(llamada);
    }

    calcularRendimiento(totalLlamadas) {
        if (totalLlamadas === 0) return 0;
        return (this.llamadas.length / totalLlamadas) * 100;
    }
}
```

Clase cliente:

-Atributos: id, nombre, llamadas [ ]

-Método:

agregarLlamada(llamada) : agrega una llamada al historial del cliente.

```
// ========================
class Cliente {
    constructor(id, nombre) {
        this.id = id;
        this.nombre = nombre;
        this.llamadas = [];
    }

    agregarLlamada(llamada) {
        this.llamadas.push(llamada);
    }
}
```

Clase Callcenter

Clase principal que gestiona todo el proyecto.

-Atributos: llamadas[], operadores{}, clientes{}, cargado, directorio Salida.

-**Métodos:**

- cargarCSV(texto) : procesa el archivo CSV y crea objetos Llamada, Operador y Cliente.

```
// Cargar registros desde .csv (cadena ya leída)
cargarCSV(texto) {

    this.llamadas = [];
    this.operadores ={};
    this.clientes = {};

    const lineas = texto.trim().split("\n");

    for (let i = 1; i < lineas.length; i++) { // ignorar cabecera
        const fila = lineas[i].split(",");

        if (fila.length < 5) continue;

        let idOperador = fila[0].trim();
        let nombreOperador = fila[1].trim();
        let estrellasCadena = fila[2].trim();
        let idCliente = fila[3].trim();
        let nombreCliente = fila[4].trim();
```

- historial() : devuelve un listado de todas las llamadas.

```
// Reportes básicos
historial() {
    return this.llamadas.map(l => `${l.idOperador} - ${l.nombreOperador} → ${l.nombreCliente} (${l.estrellas} estrellas)`);
}
```

- listadoOperadores() : devuelve todos los operadores registrados.

```
listadoOperadores() {
    return Object.values(this.operadores).map(o => `${o.id} - ${o.nombre}`);
}
```

- listadoClientes() : devuelve todos los clientes registrados.

```
listadoClientes() {
    return Object.values(this.clientes).map(c => `${c.id} - ${c.nombre}`);
}
```

- rendimientoOperadores() : calcula el rendimiento de cada operador.

```
rendimientoOperadores() {
    const total = this.llamadas.length;
    return Object.values(this.operadores).map(o => ({
        id: o.id,
        nombre: o.nombre,
        rendimiento: o.calcularRendimiento(total).toFixed(2) + "%"
    }));
}
```

- porcentajeClasificacion(): muestra porcentaje de llamadas Buenas, Medias y Malas.

```
porcentajeClasificacion(){
    let buenas=0, medias =0, malas=0;

    for (let llamada of this.llamadas){
        const clasif = llamada.clasificacion();
        if(clasif === "Buena") buenas++;
        else if (clasif ==="Media")medias++;
        else malas++;
    }

    const total = this.llamadas.length;
    return{
        buenas: ((buenas/total)*100).toFixed(2)+ "%",
        medias: ((medias/total)*100).toFixed(2)+ "%",
        malas:((malas/total)*100).toFixed(2)+ "%"

    };

}
```

- cantidadPorCalificacion() : devuelve cantidad de llamadas por calificación (1-5 estrellas).

```
cantidadPorcalificacion(){
    let conteo= {1:0, 2:0, 3:0, 4:0, 5:0};

    for(let llamada of this.llamadas){
        if(llamada.estrellas >=1 && llamada.estrellas <=5){
            conteo[llamada.estrellas]++;
        }
    }
    return conteo;
}
```

- exportarHistorial(), exportarOperadores(), exportarClientes(), exportarRendimiento() : generan reportes en HTML.

Estos métodos permiten generar los reportes en formato HTML, pasando los parámetros a la tabla generada en el código de html. Utilizando el método .map que convierte la lista de datos cargados, en un arreglo.

```
//Reporte 2: Historial llamadas.
exportarHistorial(){

    if(!this.cargado) return console.log("Primero debe cargar el archivo de llamadas.CSV");
    const encabezados =["ID Operador", "Nombre Operador", "Estrellas", "ID Cliente", "Nombre Cliente" ];
    const filas = this.llamadas.map(l => [
        l.idOperador, l.nombreOperador, l.estrellas, l.idCliente, l.nombreCliente
    ]);

    const html= this.generarTablaHTML("Historial de Llamadas", encabezados, filas);
    this.guardarReporte("historial", html);
}
```

```
exportarOperadores(){
    if(!this.cargado) return console.log("Primero debe cargar el archivo de llamadas.CSV");
    const encabezados=["ID Operador", "Nombre Operador"];
    const filas = Object.values(this.operadores).map(o => [o.id, o.nombre]);
    const html = this.generarTablaHTML("Listado de Operadores", encabezados, filas, "operadores");
    this.guardarReporte("operadores", html);
}
```

```
//reporte 4: listado de clientes.
exportarClientes(){
    if (!this.cargado) return console.log("⚠ Primero debe cargar el archivo CSV.");
    const encabezados = ["ID Cliente", "Nombre Cliente"];
    const filas = Object.values(this.clientes).map(c =>[c.id, c.nombre]);
    const html = this.generarTablaHTML("Listado de Clientes", encabezados, filas, "clientes");
    this.guardarReporte("clientes", html);
}
```

```
//Reporte 5 rendimiento de OPERADORES
exportarRendimiento(){
    if (!this.cargado) return console.log("⚠ Primero debe cargar el archivo CSV.");
    const total= this.llamadas.length;
    const encabezados=["ID Operador", "Nombre Operador", "Rendimiento"];
    const filas = Object.values(this.operadores).map(o =>[
        o.id,
        o.nombre,
        o.calcularRendimiento(total).toFixed(2) + "%"
    ]);
    const html = this.generarTablaHTML("Rendimiento de Operadores", encabezados, filas, "rendimiento");
    this.guardarReporte("rendimiento", html);
}
```

- generarTablaHTML() : genera la estructura HTML de las tablas con Bootstrap.

Para mayor conveniencia, se utiliza la librería de Bootstrap para poder crear un diseño visiblemente mas agradable, para reflejar los datos.

```
generarTablaHTML(titulo, encabezados, filas){
    return `
    <!DOCTYPE html>
    <html lang="es">
    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>${titulo}</title>
        <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.7/dist/css/bootstrap.min.css" rel="stylesheet">
         <link href="https://getbootstrap.com/docs/5.3/assets/css/docs.css" rel="stylesheet">
          <script defer src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.7/dist/js/bootstrap.bundle.min.js"></script>
    </head>
    <body class="bg-secondary-subtle">
        <nav class="navbar shadow-sm p-3 mb-5 rounded bg-secondary">
```

- guardarReporte() : guarda el reporte en la carpeta /reportes/ con nombre único usando Date.now().

```
guardarReporte(nombreBase, html) {
    this.crearDirectorioSalida();
    const nombreArchivo = `${nombreBase}_${Date.now()}.html`; // ID único
    const rutaCompleta = path.join(this.directorioSalida, nombreArchivo);

    fs.writeFileSync(rutaCompleta, html, "utf8");
    console.log(`✅ Reporte guardado en: ${rutaCompleta}`);
    return rutaCompleta;
}
```

Metodos nativos.

.map(): se utiliza este método para transformar cada objeto Llamada en un arreglo con los datos que iran a la tabla html, esto lo realiza con los métodos exportarRendimiento, exportarHistorial, exportarClientes, exportarHistorial.

.forEach: este método se utilizo para poder recorrer operadores, clientes, llamadas y mostrar datos en consola, ya que no fue necesario devolver un nuevo arreglo como se hizo al utilizar .map, solo recorrió el arreglo para exportar su información.

.Split(); Divide una cadena de texto en partes más pequeñas, usando un separador, como se utilizó, para la práctica con la calificación de la llamada (;) y para separar cada columna del documento, usando el carácter (,)

.trim();nos ayuda para eliminar los espacios en blanco al inicio y al final de una cadena, se utilizao para limpiar los datos que venían en el CSV y evitar problemas con espacios.

```
for (let i = 1; i < lineas.length; i++) { // ignorar cabecera
    const fila = lineas[i].split(",");

    if (fila.length < 5) continue;

    let idOperador = fila[0].trim();
    let nombreOperador = fila[1].trim();
    let estrellasCadena = fila[2].trim();
    let idCliente = fila[3].trim();
    let nombreCliente = fila[4].trim();
```

Restricciones.

- El archivo CSV se procesa con split() y trim(), sin librerías externas
- Validación: si no se ha cargado CSV, no se permiten exportar reportes
- Reportes en HTML con Bootstrap
- Cada reporte se guarda con un nombre único usando Date.now(), evitando sobreescrituras
- Se asegura la creación automática del directorio /reportes/

Diagrama de Flujo

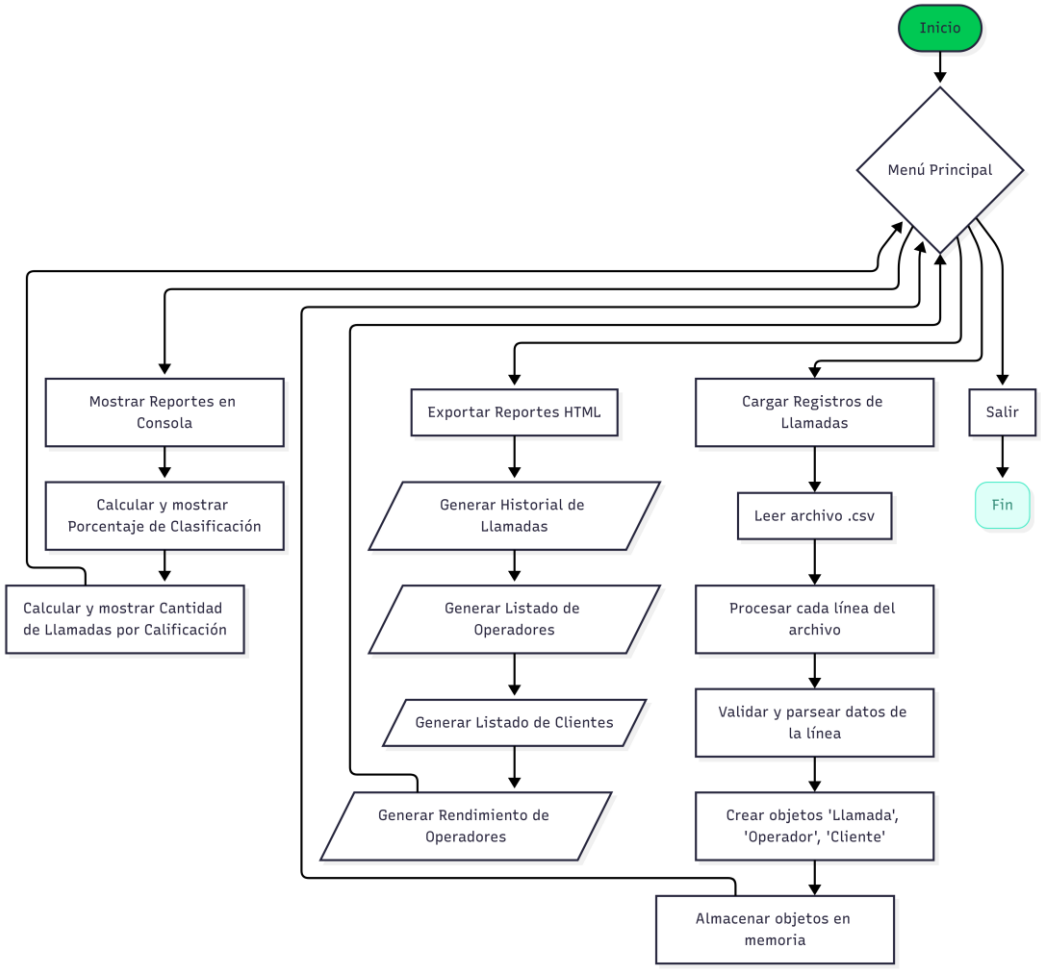
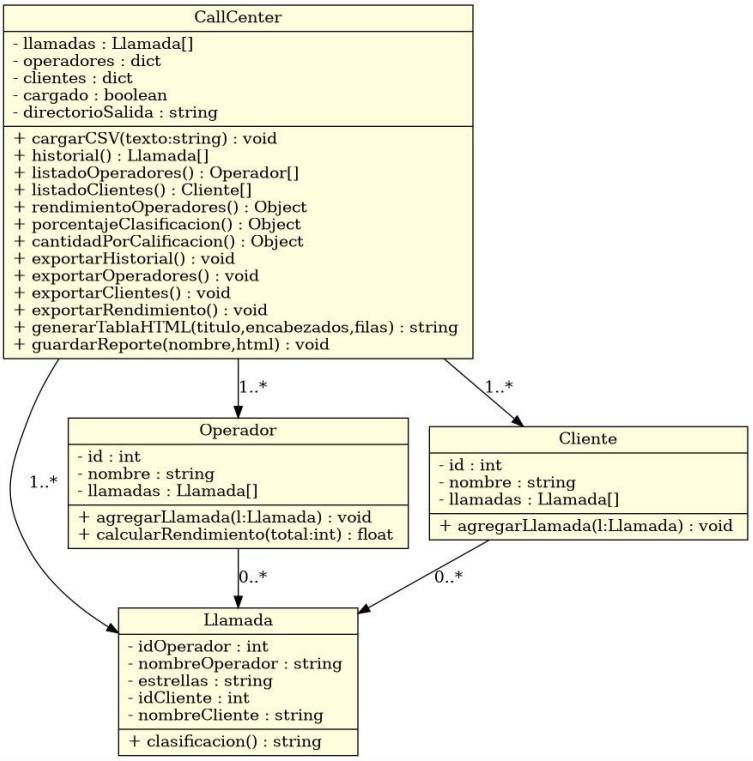


Diagrama de clases:



Conclusiones

- Se logró desarrollar un sistema modular y orientado a objetos que procesa correctamente la información de un Call Center.

- El uso exclusivo de funciones nativas de JavaScript para el análisis del CSV permitió comprender mejor la manipulación de cadenas y arreglos.

- La generación de reportes en consola e HTML facilita la interpretación de la información, ofreciendo tanto una salida rápida como una presentación estructurada.

- La práctica permitió aplicar conceptos fundamentales de POO, manejo de archivos y visualización de datos, reforzando competencias clave para el desarrollo de software.

- El proyecto puede expandirse en el futuro con visualizaciones gráficas y validaciones adicionales para hacer el sistema más robusto.