INFORME TRABAJO PRACTICO 2

Instituto CEI

Materia: Programacion 2

Federico Goldaracena

Resumen

El programa desarrollado es un sistema de gestión para una sucursal de una empresa de alquiler de vehículos. La aplicación permite llevar un registro detallado de los vehículos disponibles, así como de los alquileres realizados por los clientes.

Contenido

1. Introducción

1.1. Excepciones y control de excepciones

El control de excepciones en el lenguaje C# es fundamental para gestionar situaciones inesperadas durante la ejecución de un programa. Utiliza las palabras clave try, catch y finally para manejar acciones que podrían no completarse correctamente, controlar errores cuando sea necesario y limpiar recursos posteriormente. Las excepciones pueden ser generadas por Common Language Runtime (CLR), .NET, bibliotecas de terceros o el código de la aplicación, y se crean mediante la palabra clave throw. Es importante destacar que una excepción puede ser generada por un método distinto al llamado directamente en el código, lo que implica que el CLR busca un bloque catch adecuado en la pila de llamadas para manejar la excepción específica. En caso de no encontrar un bloque catch apropiado, el proceso finaliza y se muestra un mensaje al usuario. [1]

1.2. Relaciones de clases

En un diagrama de clases, las clases están interconectadas mediante diversas relaciones que proporcionan información sobre cómo diferentes entidades están vinculadas entre sí. Estos vínculos facilitan la comprensión de la conexión entre las distintas entidades. Aquí se explican los tipos de relaciones en un diagrama de clases y su notación en UML. 1. Asociaciones Las asociaciones representan los vínculos estáticos entre clases, conectando estructuralmente dos o más clasificadores y detallando sus atributos, propiedades y relaciones. Están visualizadas mediante una línea sólida entre los clasificadores y se dividen en cuatro tipos: unidireccional, bidireccional, agregación y composición. Asociación unidireccional: También conocida como asociación dirigida, esta relación indica que un objeto contiene otro en su campo, representada por una línea continua y una flecha apuntando hacia el clasificador de contenedores. Asociación bidireccional: Esta asociación se emplea cuando dos clasificadores están estrechamente vinculados y pueden almacenarse mutuamente en sus campos. Se representa mediante una línea continua, siendo el tipo de asociación más común en los diagramas UML. Agregación: Una asociación más específica que muestra la relación "parte de" en los diagramas, representada por una línea sólida y un diamante hueco cerca de la clase contenedora. Composición: Utilizada para representar la dependencia de los objetos de la entidad focal, donde los objetos contenidos se eliminan si se elimina la clase focal. Se representa con una línea sólida y un diamante relleno cerca de la clase contenedora. 2. Generalización/Herencia En el modelado UML, la generalización se utiliza para representar relaciones entre la clase principal y la secundaria. Muestra una relación "especie de" entre los clasificadores, indicando que una entidad hereda atributos, operaciones y relaciones de la otra. En un diagrama UML, las generalizaciones se visualizan con una línea sólida y una flecha vacía apuntando desde la clase secundaria hacia la clase principal. [2]

2. Metodología de estudio

2.1. UML Trabajo Practico 2

El sistema de gestión de alquiler de vehículos desarrollado se basa en un diagrama UML que presenta diversas clases y sus interconexiones. Entre las principales clases se encuentran: Vehiculo: Representa los vehículos disponibles para alquilar. Cada vehículo tiene atributos como número, matrícula, marca, color, kilometraje, estado, precio por día y cantidad de puertas. Está asociado con la clase Sucursal. Alquiler: Representa los contratos de alquiler entre clientes y la empresa. Contiene información como número de alquiler, precio total, datos del cliente (nombre, apellido, documento, teléfono) y una lista de detalles que incluyen vehículos alquilados y sus fechas de alquiler. Está asociado con la clase Sucursal. Detalle: Describe los detalles específicos de un alquiler, incluyendo el vehículo alquilado, la fecha de retiro y la duración del alquiler. Esta clase está compuesta por la clase Alquiler. Sucursal: Representa las sucursales de la empresa de alquiler. Cada sucursal tiene un número y una dirección, y está asociada con colecciones de vehículos y alquileres. Existen relaciones de agregación con las clases Vehiculo y Alquiler. El diagrama UML también muestra relaciones clave entre estas clases: Asociación: Indica que los objetos de una clase están relacionados con objetos de otra clase. Por ejemplo, los objetos Vehiculo y Alquiler están asociados con objetos de la clase Sucursal. Composición: Representa una relación fuerte donde una clase (por ejemplo, Detalle) es parte integral de otra clase (por ejemplo, Alquiler). Los detalles de un alquiler no pueden existir sin el alquiler asociado. Agregación: Indica una relación más débil en la que una clase (por ejemplo, Sucursal) tiene una colección de objetos de otra clase (por ejemplo, Vehiculo y Alquiler). Estos objetos pueden existir independientemente de la clase que los contiene. El diagrama UML proporciona una estructura clara para el sistema de gestión de alquiler de vehículos, mostrando cómo las clases están interrelacionadas y cómo interactúan dentro del programa. (Figura 2.1)

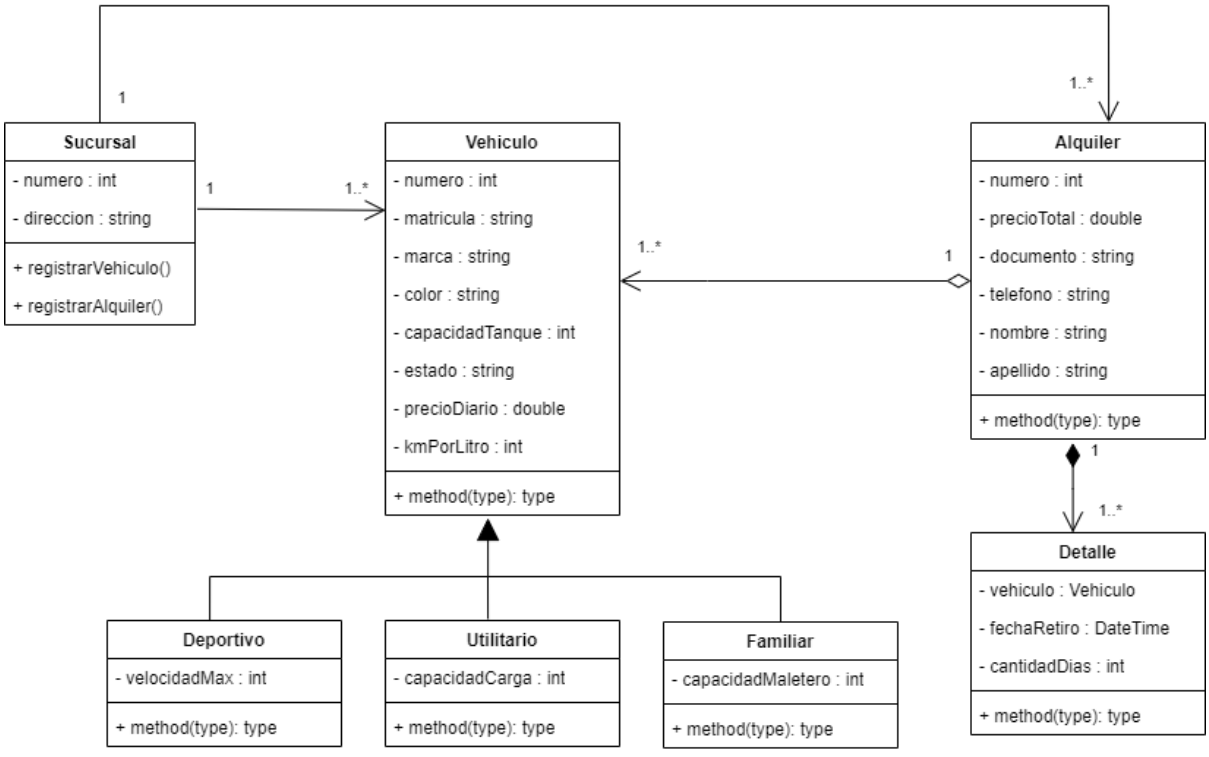

Figura 2.1: Diagrama UML.

2.2. Clase Vehiculo

La clase Vehiculo representa un vehículo en la automotora. Cada vehículo tiene atributos como número, matrícula, marca, color, kilometraje, estado, precio por día y cantidad de puertas. Estos atributos describen las características del vehículo, como su identificación única, detalles físicos y su disponibilidad para alquiler. Esta clase es esencial para llevar un registro de los vehículos disponibles y sus propiedades (Figura 2.2).

```
internal class Vehiculo
{
    private int numero;
    private string matricula;
    private string marca;
    private string color;
    private int capacidadTanque;
    private string estado;
    private double precioDiario;
    private int kmPorLitro;

    public Vehiculo(int numero, string matricula, string marca, string color, int capacidadTanque, string estado, double precioDiario, int kmPorLitro)
    {
        this.numero = numero;
        this.matricula = matricula;
        this.marca = marca;
        this.color = color;
        this.capacidadTanque = capacidadTanque;
        this.estado = estado;
        this.precioDiario = precioDiario;
        this.kmPorLitro = kmPorLitro;
    }

    public void setNumero(int numero) => this.numero = numero;
    public int getNumero() => this.numero;
    public void setMatricula(string matricula) => this.matricula = matricula;
    public string getMatricula() => this.matricula;
    public void setMarca(string marca) => this.marca = marca;
    public string getMarca() => this.marca;
    public void setColor(string color) => this.color = color;
    public string getColor() => this.color;
    public void setCapacidadTanque(int capacidadTanque) => this.capacidadTanque = capacidadTanque;
    public int getCapacidadTanque() => this.capacidadTanque;
    public void setEstado(string estado) => this.estado = estado;
    public string getEstado() => this.estado;
    public void setPrecioDiario(double precioDiario) => this.precioDiario = precioDiario;
    public double getPrecioDiario() => this.precioDiario;
    public void setKmPorLitro(int kmPorLitro) => this.kmPorLitro = kmPorLitro;
    public int getKmPorLitro() => this.kmPorLitro;
}
```

Figura 2.2: Clase Vehiculo con sus atributos y métodos.

2.3. Clase Detalle

La clase Detalle está diseñada para representar los detalles específicos de un alquiler asociado a un vehículo en particular. Almacena información como el vehículo involucrado, la fecha de retiro y la cantidad de días del alquiler. En otras palabras, un Detalle guarda los pormenores de un alquiler específico, lo que facilita el seguimiento de cada transacción de alquiler (Figura 2.3).

```
internal class Detalle
{
    private Vehiculo vehiculo;
    private DateTime fechaRetiro;
    private int cantidadDias;

    public Detalle(Vehiculo vehiculo, DateTime fechaRetiro, int cantidadDias)
    {
        this.vehiculo = vehiculo;
        this.fechaRetiro = fechaRetiro;
        this.cantidadDias = cantidadDias;
    }

    public Vehiculo getVehiculo() => this.vehiculo;
    public void setVehiculo(Vehiculo vehiculo) => this.vehiculo = vehiculo;
    public DateTime getFechaRetiro() => this.fechaRetiro;
    public void setFechaRetiro(DateTime fechaRetiro) => this.fechaRetiro = fechaRetiro;
    public int getCantidadDias() => this.cantidadDias;
    public void setCantidadDias(int cantidadDias) => this.cantidadDias = cantidadDias;
}
```

Figura 2.3: Clase Detalles con sus atributos y métodos.

2.4. Clase Alquiler

La clase Alquiler modela un contrato de alquiler en la automotora. Contiene detalles sobre el número de alquiler, el precio total, la información del cliente (como documento, teléfono, nombre y apellido) y una lista de detalles asociados al alquiler. Esta clase permite gestionar y organizar los contratos de alquiler, proporcionando una estructura para almacenar información relevante sobre cada transacción (Figura 2.4).

```
internal class Alquiler
{
    private int numero;
    private double precioTotal;
    private string documento;
    private string telefono;
    private string nombre;
    private string apellido;
    private List<Detalle> colDetalles; // Relacion de Composicion. Tengo una lista de Detalles 1---> 1..*
    private List<Vehiculo> colVehiculos; // Relacion de Agregacion. Tengo una lista de Vehiculos 1 --> 1..*
    public Alquiler(int numero, double precioTotal, string documento, string telefono, string nombre, string apellido)
    {
        this.numero = numero;
        this.precioTotal = precioTotal;
        this.documento = documento;
        this.telefono = telefono;
        this.nombre = nombre;
        this.apellido = apellido;
        this.colDetalles = new List<Detalle>(); // Composicion
        this.colVehiculos = new List<Vehiculo>(); // Agregacion
    }
    public void AgregarDetalle(Vehiculo vehiculo, DateTime fechaRetiro, int cantidadDias)
    {
        Detalle miDetalle = new Detalle(vehiculo, fechaRetiro, cantidadDias);
        colDetalles.Add(miDetalle);
    }
    public int getNumero() => this.numero;
    public void setNumero(int numero) => this.numero = numero;
    public double getPrecioTotal() => this.precioTotal;
    public void setPrecioTotal(double precioTotal) => this.precioTotal = precioTotal;
    public string getDocumento() => this.documento;
    public void setDocumento(string documento) => this.documento = documento;
    public string getTelefono() => this.telefono;
    public void setTelefono(string telefono) => this.telefono = telefono;
    public string getNombre() => this.nombre;
    public void setNombre(string nombre) => this.nombre = nombre;
    public string getApellido() => this.apellido;
    public void setApellido(string apellido) => this.apellido = apellido;
    public List<Detalle> getColDetalles() => this.colDetalles;
    public void setColDetalles(List<Detalle> colDetalles) => this.colDetalles = colDetalles;
    public List<Vehiculo> getColVehiculos() => this.colVehiculos;
    public void setColVehiculos(List<Vehiculo> colVehiculos) => this.colVehiculos = colVehiculos;
}
```

Figura 2.4: Clase Alquiler con sus atributos, métodos y relaciones.

2.5. Clase Sucursal

La clase Sucursal representa una ubicación específica de la automotora. Tiene atributos como número, dirección, una lista de vehículos y una lista de alquileres. Esta clase actúa como un contenedor para organizar y gestionar los vehículos disponibles y los contratos de alquiler en una sucursal particular. Permite realizar operaciones como registrar vehículos y alquileres, así como listarlos cuando sea necesario (Figura 2.5 y 2.6).

```
internal class Sucursal
{
    private int numero;
    private string direccion;
    private List<Vehiculo> colVehiculos; // Relacion de Asociacion. Tengo una lista de Vehiculos 1 ---> 1..*
    private List<Alquiler> colAlquileres; // Relacion de Asociacion. Tengo una lista de Alquiler 1 ---> 1..*
    public Sucursal(int numero, string direccion, List<Vehiculo> colVehiculos, List<Alquiler> colAlquileres)
    {
        this.numero = numero;
        this.direccion = direccion;
        this.colVehiculos = colVehiculos;
        this.colAlquileres = colAlquileres;
    }

    public int getNumero() => this.numero;
    public void setNumero(int numero) => this.numero = numero;
    public string getDireccion() => this.direccion;
    public void setDireccion(string direccion) => this.direccion = direccion;
    public List<Vehiculo> getVehiculos() => this.colVehiculos;
    public void setColVehiculos(List<Vehiculo> colVehiculos) => this.colVehiculos = colVehiculos;
    public List<Alquiler> GetAlquileres() => this.colAlquileres;
    public void setColAlquileres(List<Alquiler> colAlquileres) => this.colAlquileres = colAlquileres;

    public void RegistrarVehiculo(Vehiculo vehiculo)
    {
        colVehiculos.Add(vehiculo);
        //Console.WriteLine("Vehiculo registrado en la sucursal exitosamente.");
    }

    public void RegistrarAlquiler(Alquiler alquiler)
    {
        colAlquileres.Add(alquiler);
        //Console.WriteLine("Alquiler registrado en la sucursal exitosamente.");
    }

    public void ListarVehiculos()
    {
        Console.WriteLine("Vehículos en la sucursal:");
        foreach (var vehiculo in colVehiculos)
        {
            Console.WriteLine($"Número: {vehiculo.getNumero()}, Matrícula: {vehiculo.getMatricula()}, Marca: {vehiculo.getMarca()}");
        }
    }
```

Figura 2.5: Clase Sucursal con sus atributos, métodos y relaciones.

```
    public void ListarAlquileres()
    {
        Console.WriteLine("\nAlquileres en la sucursal:");
        foreach (var alquiler in colAlquileres)
        {
            Console.WriteLine($"Número de alquiler: {alquiler.getNumero()}, Precio total: {alquiler.getPrecioTotal()}, Cliente: {alquiler.getNombre()} {alquiler.getApellido()}");
            Console.WriteLine("Vehículos en el alquiler:");
            foreach (var detalle in alquiler.getColDetalles())
            {
                var vehiculo = detalle.getVehiculo();
                Console.WriteLine($"   Número: {vehiculo.getNumero()}, Matrícula: {vehiculo.getMatricula()}, Marca: {vehiculo.getMarca()}");
            }
        }
    }

    public Vehiculo BuscarVehiculoPorNumero(int numero)
    {
        foreach (var vehiculo in colVehiculos)
        {
            if (vehiculo.getNumero() == numero)
            {
                return vehiculo;
            }
        }
        return null; // Retorna null si el vehículo no se encuentra en la lista
    }
}
```

Figura 2.6: Continuación de la figura 2.4 del código de la Clase Sucursal.

2.6. Clase Deportivo

La clase Deportivo representa un tipo específico de vehículo deportivo. Hereda de la clase base Vehiculo y tiene un atributo adicional: Atributo: velocidadMax: Representa la velocidad máxima del vehículo deportivo. Constructor: El constructor de la clase Deportivo recibe parámetros que son utilizados para inicializar los atributos tanto de la clase base (Vehiculo) como del propio (velocidadMax). Métodos: getVelocidadMax(): Retorna la velocidad máxima del vehículo deportivo. setVelocidadMax(int velocidadMax): Establece la velocidad máxima del vehículo deportivo. (Figura 2.7).

```
internal class Deportivo : Vehiculo
{
    private int velocidadMax;

    0 referencias
    public Deportivo(int numero, string matricula, string marca, string color, int capacidadTanque, string estado, double precioDiario, int kmPorLitro, int velocidadMax)
        : base(numero, matricula, marca, color, capacidadTanque, estado, precioDiario, kmPorLitro)
    {
        this.velocidadMax = velocidadMax;
    }
    0 referencias
    public int getVelocidadMax() => this.velocidadMax;
    0 referencias
    public int setVelocidadMax(int velocidadMax) => this.velocidadMax = velocidadMax;
}
```

Figura 2.7: Clase Deportivo con sus atributos, métodos y la clase base Vehiculo.

2.7. Clase Utilitario

La clase Utilitario representa un tipo específico de vehículo utilitario. Al igual que Deportivo, hereda de la clase base Vehiculo y tiene un atributo adicional: Atributo: capacidadCarga: Representa la capacidad de carga del vehículo utilitario. Constructor: El constructor de la clase Utilitario recibe parámetros que son utilizados para inicializar los atributos tanto de la clase base (Vehiculo) como del propio (capacidadCarga). Métodos: getCapacidadCarga(): Retorna la capacidad de carga del vehículo utilitario. setCapacidadCarga(int capacidadCarga): Establece la capacidad de carga del vehículo utilitario (Figura 2.8).

```
internal class Utilitario : Vehiculo
{
    private int capacidadCarga;

    0 referencias
    public Utilitario(int numero, string matricula, string marca, string color, int capacidadTanque, string estado, double precioDiario, int kmPorLitro, int capacidadCarga)
        : base(numero, matricula, marca, color, capacidadTanque, estado, precioDiario, kmPorLitro)
    {
        this.capacidadCarga = capacidadCarga;
    }
    0 referencias
    public int getCapacidadCarga() => this.capacidadCarga;
    0 referencias
    public void setCapacidadCarga(int capacidadCarga) => this.capacidadCarga = capacidadCarga;
}
```

Figura 2.8: Clase Utilitario con sus atributos, métodos y la clase base Vehiculo.

2.8. Clase Familiar

La clase Familiar representa un tipo específico de vehículo familiar. Al igual que las anteriores, hereda de la clase base Vehiculo y tiene un atributo adicional:Atributo:capacidadMaletero: Representa la capacidad del maletero del vehículo familiar.Constructor:El constructor de la clase Familiar recibe parámetros que son utilizados para inicializar los atributos tanto de la clase base (Vehiculo) como del propio (capacidadMaletero).Métodos:getCapacidadMaletero(): Retorna la capacidad del maletero del vehículo familiar.setCapacidadMaletero(int capacidadMaletero): Establece la capacidad del maletero del vehículo familiar (Figura 2.9).

```
internal class Familiar : Vehiculo
{
    private int capacidadMaletero;

    0 referencias
    public Familiar(int numero, string matricula, string marca, string color, int capacidadTanque, string estado, double precioDiario, int kmPorLitro, int capacidadMaletero)
        : base(numero, matricula, marca, color, capacidadTanque, estado, precioDiario, kmPorLitro)
    {
        this.capacidadMaletero = capacidadMaletero;
    }
    0 referencias
    public int getCapacidadMaletero() => this.capacidadMaletero;
    0 referencias
    public void setCapacidadMaletero(int capacidadMaletero) => this.capacidadMaletero = capacidadMaletero;
}
```

Figura 2.9: Clase Familiar con sus atributos métodos y la clase base Vehiculo.

3. Resultados

3.1. Menú

El programa inicia mostrando un menú interactivo para gestionar una automotora. El menú se ejecuta en un bucle que permite al usuario realizar diversas operaciones. Aquí está el flujo del programa: Creación de Instancias: Se crean instancias de las listas colVehiculos, colAlquileres, y colDetalles. Se instancia un objeto de la clase Sucursal llamado sucursal con listas de vehículos y alquileres vacías. Creación de Vehículos, Alquileres y Detalles: Se crean instancias de cinco vehículos (vehiculo1 a vehiculo5) con características específicas. Se crean instancias de tres alquileres (alquiler1 a alquiler3) con información asociada. Se crean instancias de detalles (detalle1 a detalle3) asociados a vehículos y alquileres. Registro de Vehículos y Alquileres en la Sucursal: Se registran los vehículos en la sucursal mediante el método RegistrarVehiculo. Se registran los alquileres en la sucursal mediante el método RegistrarAlquiler. Menú Principal (Bucle do-while): Se inicia un bucle do-while que presenta al usuario un menú interactivo. El usuario selecciona una opción (número o letra) según la operación que desee realizar. Opciones del Menú (Switch-Case): Case 1 (Registrar Vehículo): El usuario proporciona información para registrar un nuevo vehículo. Se manejan posibles errores (por ejemplo, formato inválido). Se crea una instancia de Vehiculo y se registra en la sucursal. Case 2 (Registrar Alquiler): El usuario proporciona información para registrar un nuevo alquiler y vehículos asociados. Se manejan posibles errores (por ejemplo, formato inválido). Se crean instancias de Alquiler y se registran en la sucursal. Case 3 (Listar Vehículos): Se muestra una lista de vehículos disponibles en la sucursal. Case 4 (Listar Alquileres): Se muestra una lista de alquileres registrados en la sucursal. Case 5 (Buscar Vehículo por Número): El usuario ingresa un número de vehículo y se muestra su información. Se manejan posibles errores (por ejemplo, vehículo no encontrado). Case 'x' (Salir): Se muestra un mensaje de despedida y se sale del bucle do-while. Default: Se maneja cualquier otra opción no válida. Mensaje de Continuación: Después de cada operación, se muestra un mensaje pidiendo al usuario que presione cualquier tecla para continuar. Finalización del

Programa: El programa continúa ejecutándose en el bucle hasta que el usuario elige salir (x). Al salir, se muestra un mensaje de agradecimiento. Este enfoque modular permite una gestión eficiente de vehículos, alquileres y detalles en la automotora, proporcionando al usuario un medio fácil de interactuar con el sistema (Figura 3.1 y 3.2).

```
internal class Program
{
    static void Main(string[] args)
    {
        char opcion;
        List<Vehiculo> colVehiculos = new List<Vehiculo>();
        List<Alquiler> colAlquileres = new List<Alquiler>();
        List<Detalle> colDetalles = new List<Detalle>();

        // Crear una instancia de la clase Sucursal
        Sucursal sucursal = new Sucursal(1, "Dirección de la Sucursal", colVehiculos, colAlquileres);

        // Crear instancias de Vehiculo
        Vehiculo vehiculo1 = new Vehiculo(1, "ABC123", "Toyota", "Rojo", 50, "Disponible", 50.0, 15);
        Vehiculo vehiculo2 = new Vehiculo(2, "DEF456", "Ford", "Azul", 45, "Disponible", 45.0, 18);
        Vehiculo vehiculo3 = new Vehiculo(3, "GHI789", "Chevrolet", "Verde", 55, "Disponible", 55.0, 14);
        Vehiculo vehiculo4 = new Vehiculo(4, "JKL012", "Honda", "Amarillo", 48, "Disponible", 48.0, 17);
        Vehiculo vehiculo5 = new Vehiculo(5, "MNO345", "Nissan", "Negro", 52, "Disponible", 52.0, 16);

        // Crear instancias de Alquiler
        Alquiler alquiler1 = new Alquiler(1, 200.0, "12345678", "123456789", "Pepe", "Lopez");
        Alquiler alquiler2 = new Alquiler(2, 300.0, "23456789", "234567890", "Gonzalo", "Sommaruga");
        Alquiler alquiler3 = new Alquiler(3, 150.0, "34567890", "345678901", "Lola", "Mento");

        // Crear instancias de Detalle y asociarlas con Alquiler
        Detalle detalle1 = new Detalle(vehiculo1, DateTime.Now, 3);
        Detalle detalle2 = new Detalle(vehiculo2, DateTime.Now, 5);
        Detalle detalle3 = new Detalle(vehiculo3, DateTime.Now, 2);

        // Registrar vehículos en la sucursal
        sucursal.RegistrarVehiculo(vehiculo1);
        sucursal.RegistrarVehiculo(vehiculo2);
        sucursal.RegistrarVehiculo(vehiculo3);
        sucursal.RegistrarVehiculo(vehiculo4);
        sucursal.RegistrarVehiculo(vehiculo5);

        // Registrar alquileres en la sucursal
        sucursal.RegistrarAlquiler(alquiler1);
        sucursal.RegistrarAlquiler(alquiler2);
        sucursal.RegistrarAlquiler(alquiler3);

        //alquiler1.AgregarDetalle(detalle1);
        //alquiler2.AgregarDetalle(detalle2);
        //alquiler3.AgregarDetalle(detalle3);

        do
        {
            Console.Clear(); // Limpia la consola

            // Mostrar el encabezado del menú
            Console.WriteLine("\t              AUTOMOTORA        ");
            Console.WriteLine("\n\t*************************************");
            Console.WriteLine("\t*                MENU                 *");
            Console.WriteLine("\t*************************************");
            Console.WriteLine("\t* 1.        REGISTRAR VEHICULO        *");
            Console.WriteLine("\t* 2.        REGISTRAR ALQUILER        *");
            Console.WriteLine("\t* 3.        LISTA VEHICULOS           *");
            Console.WriteLine("\t* 4.        LISTAR ALQUILERES         *");
            Console.WriteLine("\t* 5.        BUSCAR VEHICULO POR NUMERO *");
            Console.WriteLine("\t* X.        SALIR                     *");
            Console.WriteLine("\t*************************************");
            Console.Write("\tSeleccione una opción: ");
```

Figura: 3.1: Parte del código del Menú y sus instancias (se muestra una parte del código porque no cabe la captura de pantalla con el código completo).

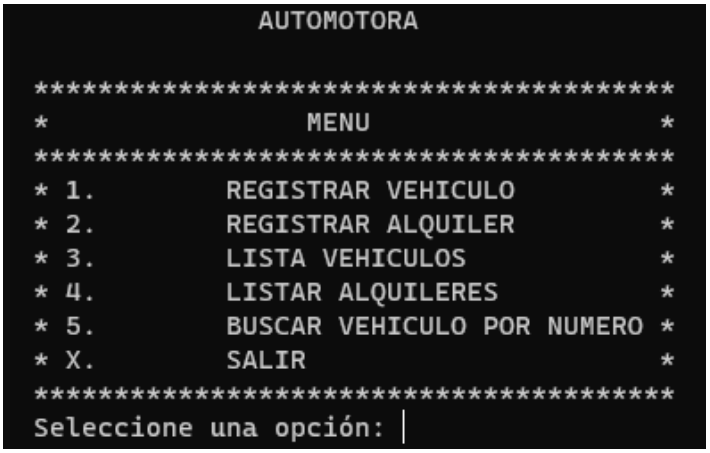

Figura: 3.2: Menú mostrado por consola.

3.2. Instrucción try catch

El código que puede generar una excepción se coloca dentro del bloque try. En este caso, la creación de instancias de vehículo o alquiler, así como la llamada a métodos que podrían lanzar excepciones. Catch (TipoDeExcepcion): Después del bloque try, hay uno o más bloques catch. Cada bloque catch especifica el tipo de excepción que manejará. En el caso de FormatException, se trata de errores de formato al convertir tipos de datos. El bloque catch (Exception ex) maneja cualquier otra excepción que no se haya manejado explícitamente. Código en el Catch: Dentro del bloque catch, se especifica qué hacer si se captura la excepción. En este caso, se imprime un mensaje de error indicando el tipo de error. El uso del bloque try-catch permite que el programa maneje graciosamente situaciones de error, evitando que el programa se cierre inesperadamente. Proporciona una forma de recuperarse de situaciones excepcionales y brinda información útil sobre el error para su posterior diagnóstico y corrección (Figura 3.3, 3.4 y 3.5).

```
case '1':
    Console.Clear();
    Console.WriteLine("\n*****************************************");
    Console.WriteLine("Registrar Vehículo:");

    try
    {
        Console.Write("Número: ");
        int numero = int.Parse(Console.ReadLine());

        Console.Write("Matrícula: ");
        string matricula = Console.ReadLine();

        Console.Write("Marca: ");
        string marca = Console.ReadLine();

        Console.Write("Color: ");
        string color = Console.ReadLine();

        Console.Write("Kilometraje: ");
        int kilometraje = int.Parse(Console.ReadLine());

        Console.Write("Estado: ");
        string estado = Console.ReadLine();

        Console.Write("Precio por día: ");
        double precioPorDia = double.Parse(Console.ReadLine());

        Console.Write("Cantidad de puertas: ");
        int cantidadPuertas = int.Parse(Console.ReadLine());

        // Crear instancia de Vehiculo con los detalles proporcionados
        Vehiculo nuevoVehiculo = new Vehiculo(numero, matricula, marca, color, kilometraje, estado, precioPorDia, cantidadPuertas);

        // Registrar el nuevo vehículo en la sucursal
        sucursal.RegistrarVehiculo(nuevoVehiculo);

        Console.WriteLine("Vehículo registrado exitosamente.");
    }
    catch (FormatException)
    {
        Console.WriteLine("Error: Por favor, ingrese un número válido para el número del vehículo.");
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Error: {ex.Message}");
    }
    break;
```

Figura 3.3: Parte del código del Menú con la instrucción de try catch.

```
try
{
    Console.Write("Número de Alquiler: ");
    int numeroAlquiler = int.Parse(Console.ReadLine());

    Console.Write("Precio Total: ");
    double precioTotal = double.Parse(Console.ReadLine());

    Console.Write("Documento: ");
    string documento = Console.ReadLine();

    Console.Write("Teléfono: ");
    string telefono = Console.ReadLine();

    Console.Write("Nombre: ");
    string nombre = Console.ReadLine();

    Console.Write("Apellido: ");
    string apellido = Console.ReadLine();

    // Crear instancia de Alquiler con los detalles proporcionados
    Alquiler nuevoAlquiler = new Alquiler(numeroAlquiler, precioTotal, documento, telefono, nombre, apellido);

    // Agregar vehículos al alquiler
    Console.WriteLine("Agregar vehículos al alquiler (presione enter para finalizar):");

    while (true)
    {
        Console.Write("Número de Vehículo: ");
        int numeroVehiculo = int.Parse(Console.ReadLine());

        // Encontrar el vehículo en la lista de vehículos de la sucursal
        Vehiculo vehiculo = sucursal.BuscarVehiculoPorNumero(numeroVehiculo);

        if (vehiculo != null)
        {
            Console.Write("Fecha de Retiro (yyyy-mm-dd): ");
            DateTime fechaRetiro = DateTime.Parse(Console.ReadLine());

            Console.Write("Cantidad de Días: ");
            int cantidadDias = int.Parse(Console.ReadLine());

            // Agregar detalle al alquiler
            nuevoAlquiler.AgregarDetalle(vehiculo, fechaRetiro, cantidadDias);
        }
        else
        {
            Console.WriteLine("Error: Vehículo no encontrado.");
        }

        Console.Write("¿Desea agregar otro vehículo? (s/n): ");
        if (Console.ReadKey().KeyChar.ToString().ToLower() != "s")
        {
            break;
        }
    }

    // Registrar el nuevo alquiler en la sucursal
    sucursal.RegistrarAlquiler(nuevoAlquiler);

    Console.WriteLine("\nAlquiler registrado exitosamente.");
}
catch (FormatException)
```

Figura 3.4: Parte del código del Menú con la instrucción de try catch.

```
catch (FormatException)
{
    Console.WriteLine("Error: Por favor, ingrese un formato válido.");
}
catch (Exception ex)
{
    Console.WriteLine($"Error: {ex.Message}");
}
break;
```

Figura 3.5: Continuación de la figura 3.4.

4. Conclusiones

La implementación de bloques try-catch, junto con la representación de relaciones de clases mediante diagramas UML, simplifica y organiza el diseño de la aplicación antes de la fase de codificación. Estas prácticas no solo contribuyen a un código más claro y estructurado, sino que también mejoran la capacidad del sistema para gestionar excepciones de manera efectiva. La inclusión de bloques try-catch proporciona un mecanismo para anticipar y manejar errores durante la ejecución del programa, mejorando la robustez y la capacidad de respuesta del sistema. Los mensajes descriptivos en los bloques catch facilitan la identificación y solución de problemas, mejorando la experiencia tanto para los desarrolladores como para los usuarios. Por otro lado, la representación visual de las relaciones de clases mediante diagramas UML ofrece una visión holística de la estructura del sistema. Esto permite a los desarrolladores comprender de manera más clara las interconexiones entre las entidades del programa, facilitando una planificación más efectiva antes de la implementación. En conclusión, la combinación de bloques try-catch y diagramas UML en el proceso de desarrollo proporciona una metodología que simplifica y organiza el diseño del software, mejorando su calidad y mantenibilidad.

5. Referencias

[1] https://learn.microsoft.com/es-es/dotnet/csharp/fundamentals/exceptions/

[2] https://www.edrawsoft.com/es/article/class-diagram-relationships.html