| | Carátula para entrega de prácticas |
|---|---|
| Facultad de Ingeniería | Laboratorios de docencia |

# Laboratorio de Computación Salas A y B

*Profesor(a):* Dávila Pérez René Adrián

*Asignatura:* Programación Orientada a Objetos

*Grupo:* 1

*No de Práctica(s):* Practica 9 y 10

*Integrante(s):* 322089020

322089817

322151194

425091586

322085390

*No. de lista o brigada:* Equipo 4

*Semestre:* 2026-1

*Fecha de entrega:* 16 de noviembre de 2025

*Observaciones:*

CALIFICACIÓN: _____

# Índice

# 1. Introducción

## 1.1. Planteamiento del Problema

En esta práctica se aborda el análisis y representación estructural y dinámica de una aplicación desarrollada en Dart, con énfasis en la elaboración de diagramas UML y en la interpretación del manejo de excepciones implementando en el código base proporcionado. El problema central consiste en comprender la arquitectura del programa, identificar las relaciones entre las clases, describir el flujo de ejecución y explicar el papel que desempeña el control de errores dentro de la solución.

## 1.2. Motivación

La necesidad de resolver este problema radica en que la programación orientada a objetos exige no solo la correcta implementación del código, sino también la capacidad de modelar y documentar los componentes del sistema de manera clara. La representación UML facilita la lectura, el mantenimiento y la escalabilidad del software, mientras que el análisis del manejo de excepciones permite evaluar la solidez y robustez de la aplicación frente a condiciones inesperadas.

## 1.3. Objetivos

El objetivo principal de esta práctica es generar un reporte técnico que describa la estructura del programa mediante un diagrama UML estático, ilustre su comportamiento mediante un diagrama UML dinámico e interprete los conceptos teóricos aplicados, destacando el mecanismo de excepciones empleado en la solución. Con ello se busca consolidar la habilidad de analizar, modelar y documentar software conforme a los lineamientos establecidos para la elaboración de reportes académicos.

# 2. Marco Teórico

## 2.1. Abstracción

La abstracción es un principio fundamental de la POO que consiste en enfocarse en las características esenciales de un objeto, descartando la información que no es relevante en un contexto particular, para así gestionar la complejidad del sistema. Permite a los desarrolladores construir sistemas complejos de manera más sencilla y organizada, ofreciendo una representación simplificada de la realidad o de un concepto subyacente [2].

## 2.2. Encapsulamiento

El encapsulamiento se refiere a la práctica de agrupar los datos (atributos) y los métodos (comportamientos) que operan sobre esos datos en una única unidad, conocida como clase. Además, permite limitar el acceso directo a los datos (protección de datos), ocultando los detalles de implementación interna de un objeto y exponiendo solo la información necesaria a través de interfaces públicas (métodos), lo que protege la integridad de los datos y facilita la mantenibilidad del código [8].

## 2.3. Herencia

La herencia es un mecanismo de la POO que posibilita la creación de nuevas clases (subclases o clases hijas) a partir de clases existentes (superclases o clases padres). La subclase hereda automáticamente los atributos y métodos de su superclase, permitiendo la reutilización de código, la especialización de objetos y la definición de una jerarquía entre clases [3].

## 2.4. Polimorfismo

El polimorfismo es la habilidad de un objeto de tomar diferentes formas o de realizar una acción de diferentes maneras, dependiendo del contexto o del objeto específico. Permite

que comportamientos diferentes, asociados a objetos distintos, compartan el mismo nombre, logrando flexibilidad y compatibilidad de objetos en el diseño de software. Se implementa comúnmente mediante la sobrecarga y la sobrescritura de métodos [7].

## 2.5. Dart

Dart es un lenguaje de programación moderno, de código abierto y orientado a objetos desarrollado por Google. Es un lenguaje puramente orientado a objetos, basado en clases, diseñado con un enfoque en la facilidad de uso y la familiaridad para la mayoría de los programadores. Es conocido por ser un lenguaje optimizado para el cliente, lo que lo hace ideal para la creación de aplicaciones multiplataforma. Las aplicaciones Dart se ejecutan utilizando la Máquina Virtual Dart, que permite la compilación Just-In-Time (JIT) para desarrollo y la compilación Ahead-Of-Time (AOT) para producción [4].

## 2.6. Flutter

Flutter es un kit de desarrollo de software (SDK) de interfaz de usuario (UI) de código abierto creado por Google. Se utiliza para desarrollar aplicaciones multiplataforma para sistemas operativos como Android, iOS, Windows, Mac, Linux y la web, utilizando una única base de código. Flutter se basa en el lenguaje Dart y emplea un framework reactivo moderno con un extenso conjunto de widgets de diseño específico que permiten crear una UI consistente y de alto rendimiento. Su motor de renderizado, escrito en C++, utiliza la biblioteca gráfica Skia de Google para el renderizado de bajo nivel [1].

## 2.7. Excepciones

El manejo de excepciones es un mecanismo que permite gestionar errores y situaciones excepcionales de manera controlada durante la ejecución de un programa. En Dart, todas las excepciones derivan de la clase `Exception` y se utilizan para separar el flujo normal del

programa del manejo de condiciones de error, mejorando así la robustez y mantenibilidad del software [6].

## 2.8. Diagramas UML

El Lenguaje Unificado de Modelado (UML) es un estándar para visualizar, especificar, construir y documentar artefactos de sistemas de software [5].

### 2.8.1. Diagrama UML Estático

El diagrama de clases representa la estructura del sistema, mostrando:

- Interfaz `ServicioTaller` con métodos abstractos
- Clase abstracta `Vehiculo` que implementa la interfaz
- Clases concretas: `Auto`, `Moto`, `Camión`
- Relaciones de herencia y realización

### 2.8.2. Diagrama UML Dinámico

El diagrama de secuencia muestra las interacciones temporales durante el registro de vehículos, incluyendo:

- Flujo normal de creación de objetos
- Propagación y captura de excepciones
- Validaciones en setters y constructores

# 3. Desarrollo

## 3.1. Análisis del Código

### 3.1.1. Interpretación del Concepto de Excepciones

Las excepciones en el programa sirven para manejar situaciones que no estaban planeadas y que pueden provocar que el código falle de manera inesperada. En términos simples, una excepción es como una señal que avisa que algo salió mal y que el programa necesita una forma especial de responder para no detenerse por completo.

En el código analizado, las excepciones ayudan a mantener ordenada la lógica, porque separan lo que hace el programa normalmente de lo que debe hacer cuando ocurre un error. Esto evita estar comprobando manualmente cada posible fallo, ya que el propio lenguaje detecta el problema y permite reaccionar de inmediato.

El uso de bloques como `try-catch` demuestra que el programa está preparado para enfrentar errores. Cuando algo no funciona —por ejemplo, un dato inválido o una operación incorrecta— el `catch` permite que el programa siga funcionando sin cerrarse y muestra un mensaje o realiza otra acción que ayude a manejar la situación.

Además, trabajar con excepciones hace más fácil encontrar de dónde viene el error y corregirlo después, porque el manejo está más organizado y centralizado. En general, este mecanismo hace que el programa sea más estable, más claro y capaz de recuperarse cuando ocurre algo inesperado.

### 3.1.2. Uso de Herencia

El código emplea herencia para reutilizar comportamientos comunes entre clases relacionadas. Esto permite que las clases derivadas extiendan la funcionalidad base sin reescribir código, lo cual mejora la cohesión del sistema. La herencia también actúa como fundamento para otras características, como el polimorfismo y la sobrescritura de métodos.

### 3.1.3. Polimorfismo

El polimorfismo se observa cuando múltiples clases comparten una misma interfaz de comportamiento, permitiendo que un mismo método invoque acciones diferentes dependiendo de la instancia concreta utilizada. Este mecanismo facilita que el programa sea expandible, pues nuevas clases pueden integrarse sin necesidad de modificar el código existente que trabaja sobre tipos más generales.

### 3.1.4. Sobrescritura de Métodos

En varias clases se emplea la sobrescritura para modificar o especializar el comportamiento definido en clases superiores. Esto permite que cada clase concreta implemente su propia versión de métodos compartidos, respetando la estructura de la superclase pero adaptándose a las necesidades particulares de cada entidad.

### 3.1.5. Encapsulamiento

El encapsulamiento está presente mediante el uso de atributos privados o protegidos, gestionados a través de métodos públicos. Esto evita la manipulación directa del estado interno de los objetos, garantizando mayor seguridad lógica y evitando inconsistencias. La gestión del estado mediante getters y setters también ayuda a centralizar la validación de datos.

### 3.1.6. Modularidad y Separación de Responsabilidades

Cada clase del código está diseñada para cumplir una responsabilidad específica, lo que sigue el principio *Single Responsibility*. Esto permite localizar problemas y modificar componentes sin afectar al resto del sistema. El uso de archivos separados para cada módulo mejora la organización del proyecto.

### 3.1.7. Interacción entre Objetos

Las clases mantienen comunicación mediante composición y agregación; es decir, algunas clases contienen instancias de otras. Esto permite construir estructuras complejas a partir de componentes simples, promoviendo un diseño escalable y flexible.

### 3.1.8. Gestión de Flujo y Lógica de Negocio

La lógica del programa se distribuye de manera ordenada entre las clases, permitiendo que el flujo de ejecución sea claro. Se integran decisiones, operaciones y cálculos que responden al propósito del proyecto sin saturar las clases con tareas excesivas.

## 3.2. Diagramas UML

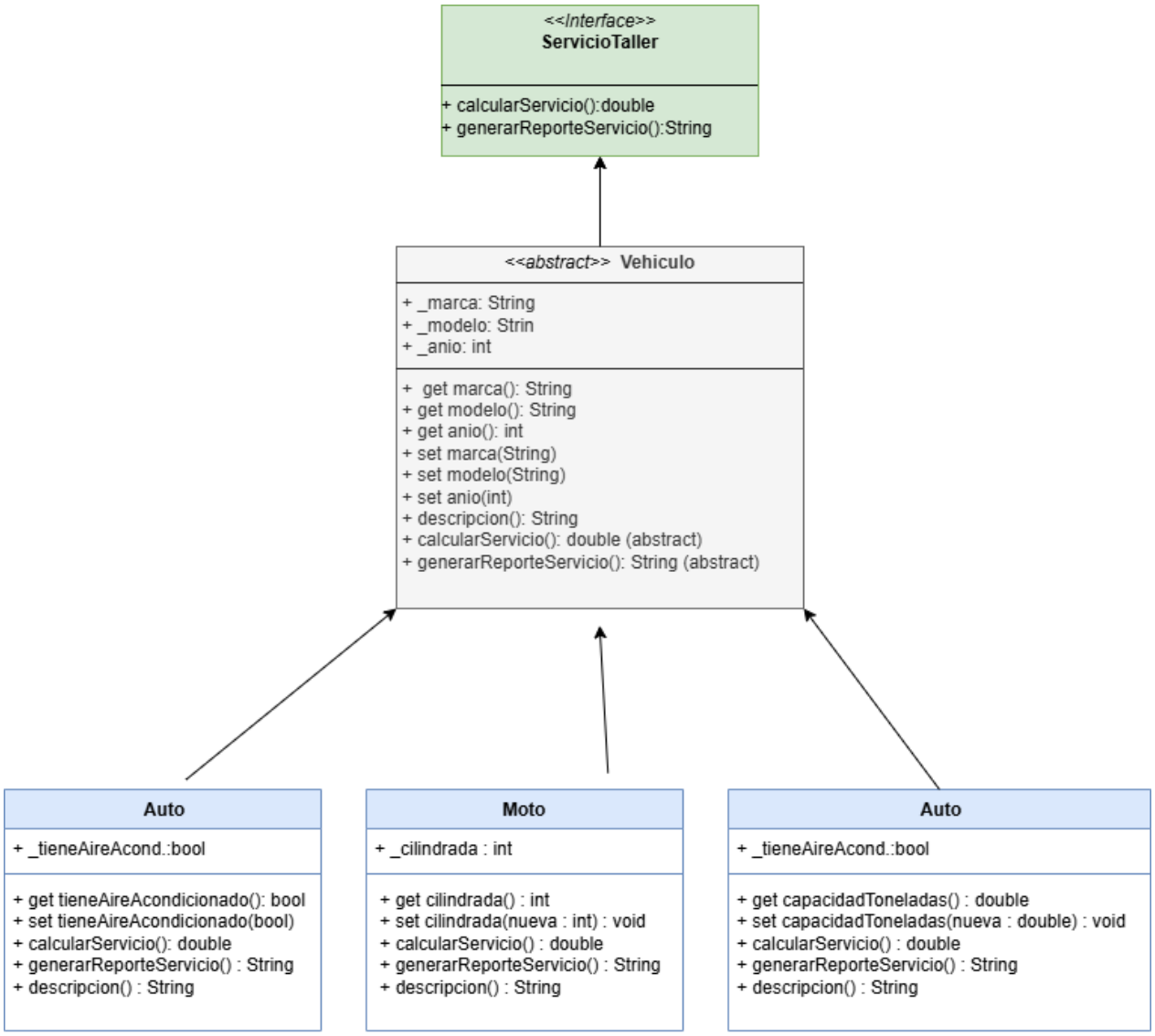


Figura 1: Diagrama de clases

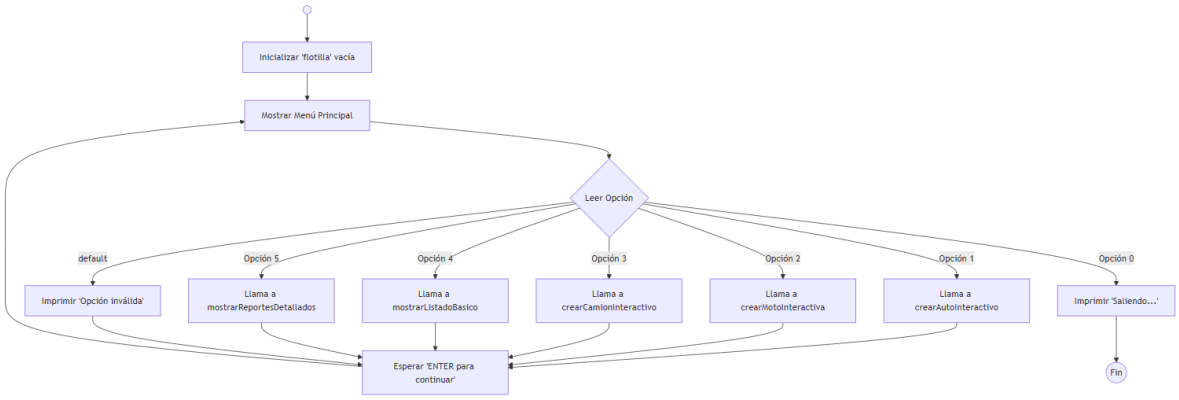


Figura 2: Diagrama de dinamico del main

### 3.3. Pruebas

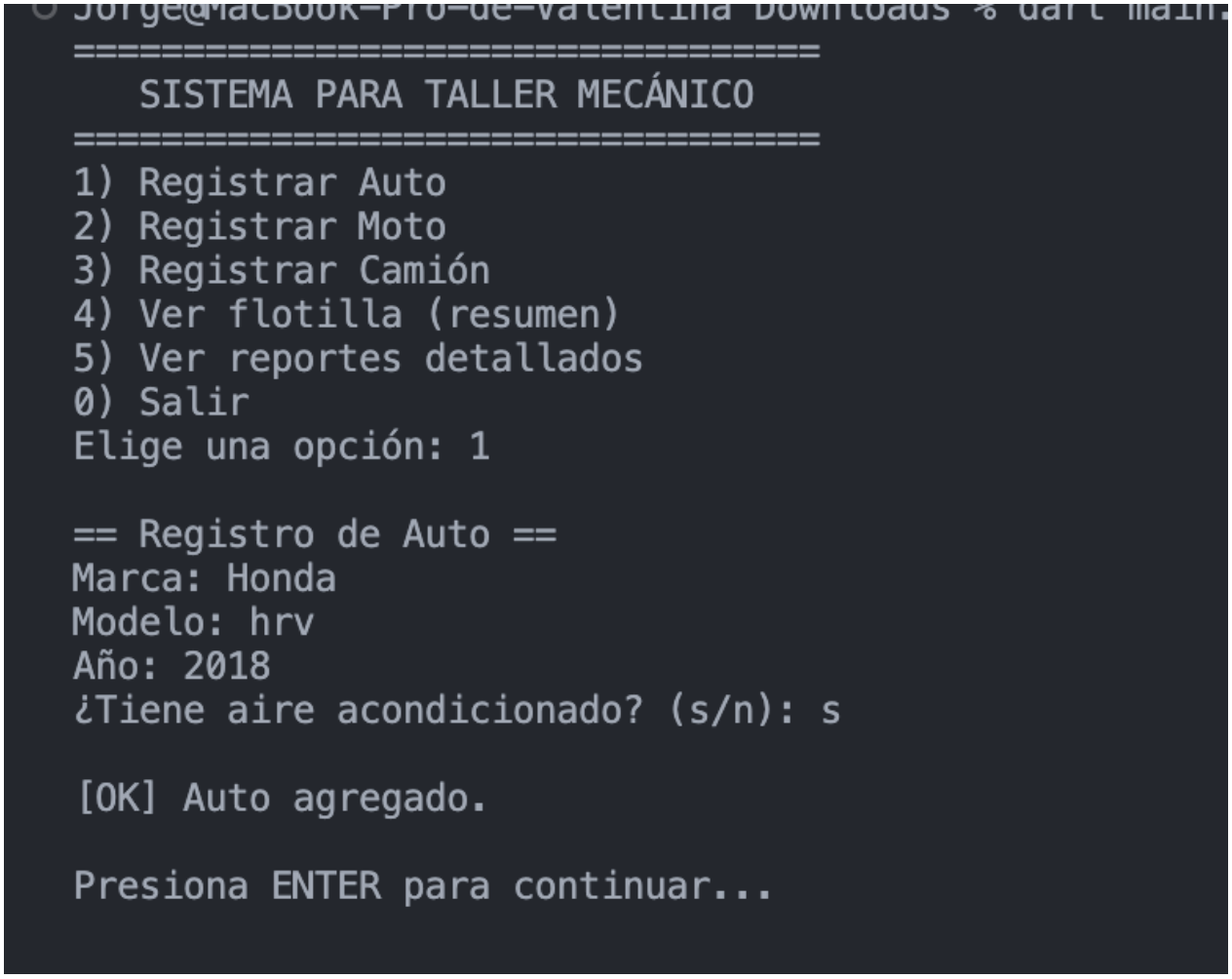


Figura 3: Registro de auto

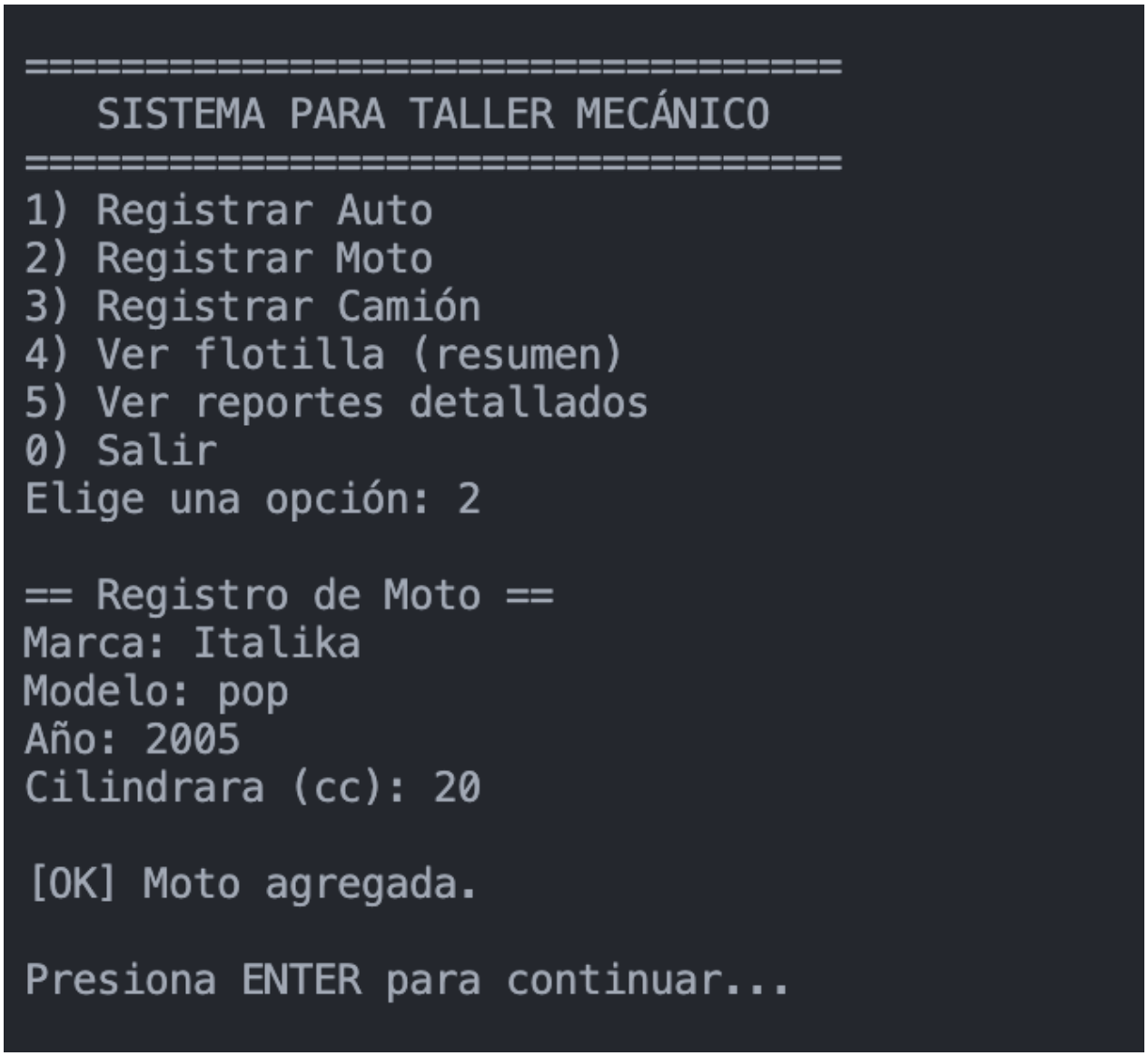


Figura 4: Registro de moto

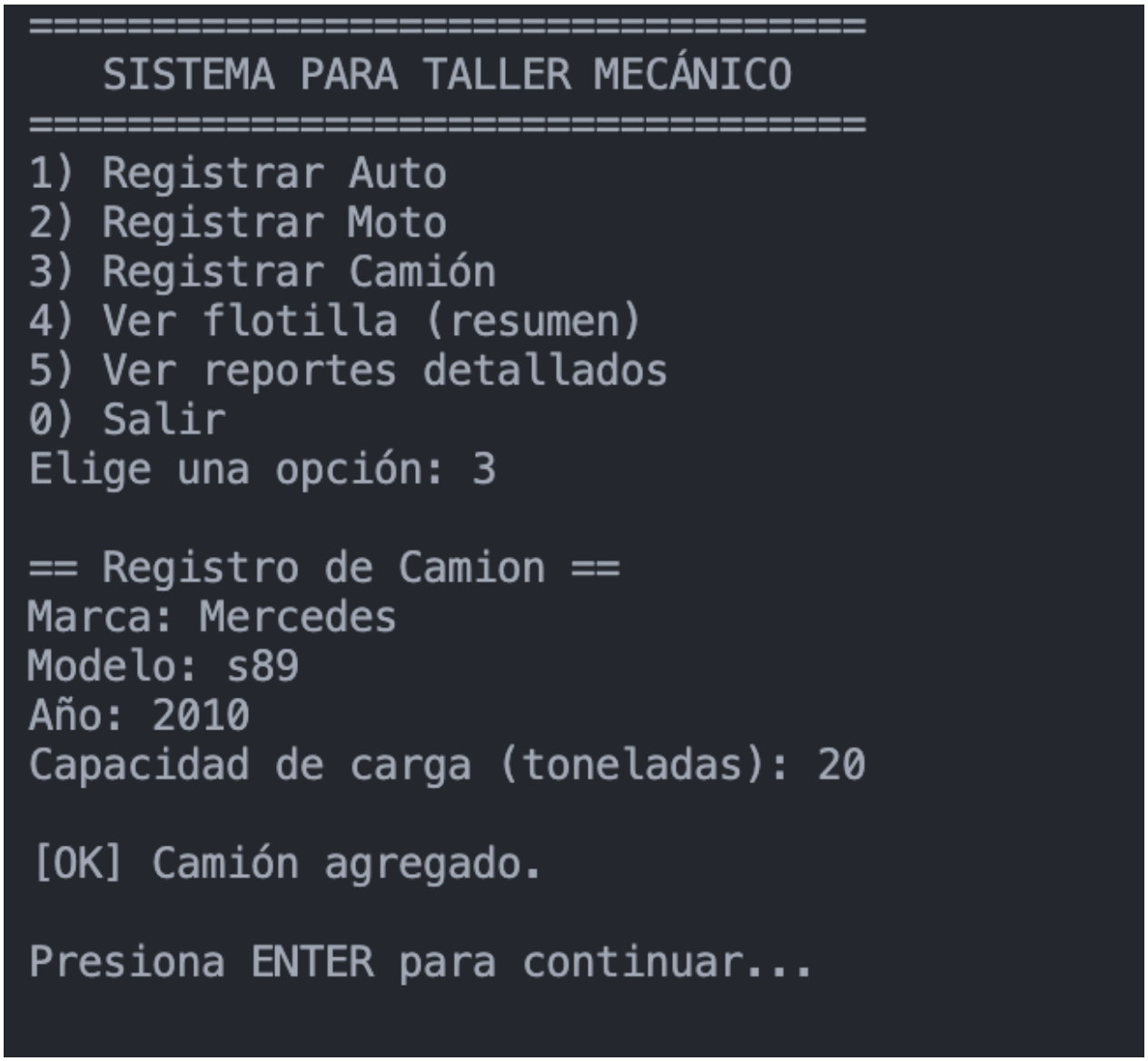


Figura 5: Registro de camion

```
=== Flotilla registrada ===

[0] Auto: Honda hrv (2018) - A/C: sí | Servicio: $1250.00
[1] Moto: Italika pop (2003) - 12cc | Servicio: $450.00
[2] Moto: Italika 2005 (23) - 23cc | Servicio: $450.00
[3] Moto: Italika pop (2005) - 20cc | Servicio: $450.00
[4] Camión: Mercedes s89 (2010) - Capacidad: 20.0 toneladas | Servicio: $3600.00
```

Figura 6: Resumen de flotilla

```
=== Flotilla registrada ===

Servicio para AUTO Honda hrv:
- Año: 2018
- A/C: sí
- Total: $1250.00

Servicio para MOTO Italika pop:
- Año: 2003
- Cilindrada: 12cc
- Total: $450.00

Servicio para MOTO Italika 2005:
- Año: 23
- Cilindrada: 23cc
- Total: $450.00

Servicio para MOTO Italika pop:
- Año: 2005
- Cilindrada: 20cc
- Total: $450.00

Servicio para CAMIÓN Mercedes s89:
- Año: 2010
- Capacidad: 20.0 toneladas
- Total: $3600.00
```

Figura 7: Detalle de flotila

# 4. Conclusiones

El análisis realizado permitió identificar con claridad la estructura interna del programa y representar sus componentes mediante un diagrama UML estático, lo que evidencia las clases definidas, sus atributos, métodos y relaciones. Asimismo, el diagrama UML dinámico permitió describir el flujo de ejecución y la interacción entre objetos durante la operación de la aplicación. A partir de esta representación, fue posible interpretar con precisión los conceptos de programación orientada a objetos aplicados en el código dart.main aportado.

El estudio del manejo de excepciones mostró su relevancia como mecanismo para mantener la estabilidad del programa ante situaciones no previstas durante la ejecución. La forma en que se implementaron dentro del código ejemplifica cómo el lenguaje Dart permite capturar y gestionar errores de forma controlada, garantizando un funcionamiento seguro y predecible. En conjunto, los resultados obtenidos confirman la importancia de emplear modelos UML y técnicas adecuadas de control de errores para desarrollar aplicaciones estructuradas, comprensibles y resistentes a fallos.

# Referencias


[1] Varios Autores. "Blockchain-based Framework for Secure and Transparent Object-Oriented Systems". En: *2023 IEEE International Conference on Blockchain*. [En línea]. Disponible en: https://ieeexplore.ieee.org/document/10156999. IEEE, 2023, págs. 1-8. DOI: 10.1109/Blockchain.2023.10156999. URL: https://ieeexplore.ieee.org/document/10156999 (visitado 14-10-2025).

[2] Ecosistema BUAP. *Principios Básicos de la Programación Orientada a Objetos*. [En línea]. Disponible en: https://ecosistema.buap.mx/forms/files/dspace-23/1_principios_bsicos_de_la_poo.html. 2023. URL: https://ecosistema.buap.mx/forms/files/dspace-23/1_principios_bsicos_de_la_poo.html (visitado 14-10-2025).

[3] CodersLink. *¿Qué es la Programación Orientada a Objetos (POO) y cuáles son sus principios fundamentales?* [En línea]. Disponible en: https://coderslink.com/talento/blog/que-es-la-programacion-orientada-a-objetos-poo-y-cuales-son-sus-principios-fundamentales/. 2023. URL: https://coderslink.com/talento/blog/que-es-la-programacion-orientada-a-objetos-poo-y-cuales-son-sus-principios-fundamentales/ (visitado 14-10-2025).

[4] Creapolis. *Introducción a Dart: Lenguaje de Programación de Google*. [En línea]. Disponible en: https://creapolis.dev/introduccion-a-dart-lenguaje-de-programacion-de-google/. 2023. URL: https://creapolis.dev/introduccion-a-dart-lenguaje-de-programacion-de-google/ (visitado 14-10-2025).

[5] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. 3.ª ed. Boston, MA: Addison-Wesley, 2004.

[6] Google. *Exceptions in Dart*. [En línea]. Disponible en: https://dart.dev/language/error-handling. 2025. URL: https://dart.dev/language/error-handling (visitado 14-10-2025).

[7] OpenWebinars. *Introducción a POO en Java: Herencia y Polimorfismo*. [En línea]. Disponible en: https://openwebinars.net/blog/introduccion-a-poo-en-java-herencia-y-polimorfismo/. 2023. URL: https://openwebinars.net/blog/introduccion-a-poo-en-java-herencia-y-polimorfismo/ (visitado 14-10-2025).

[8] Felipe Restrepo. *Paradigmas de Programación: Programación Orientada a Objetos*. [En línea]. Disponible en: https://ferestrepoca.github.io/paradigmas-de-programacion/poo/poo_teoria/concepts.html. 2023. URL: https://ferestrepoca.github.io/paradigmas-de-programacion/poo/poo_teoria/concepts.html (visitado 14-10-2025).