	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor: René Adrián Dávila Pérez

Asignatura: Programación Orientada a Objetos

Grupo: 01

No. de práctica(s): 09 y 10

Integrante(s):
322276824
322258516
425037384
320108116
322221415

No. de brigada: 01

Semestre: 2026-1

Fecha de entrega: 13 de noviembre de 2025

Observaciones:

CALIFICACIÓN: _____

Índice

1. Introducción

- **Planteamiento del problema:**
  Desarrollar un programa que registre distintos tipos de vehículos y calcule el costo de su servicio según sus características, que interactúe con el usuario a través de la consola y valide los datos capturados, para ello se requiere incluir manejo de excepciones para controlar los errores generados por las entradas inválidas, así como otros conceptos teóricos. Además de documentar el funcionamiento mediante diagramas UML.

- **Motivación:**
  La migración hacia el ecosistema de Dart y Flutter es un paso para acercarnos a herramientas de desarrollo modernas y orientadas a entornos multiplataforma. Comprender cómo los conceptos de POO se aplican en un lenguaje diferente amplia la capacidad de adaptación a nuevos lenguajes y entornos. Además, el correcto manejo de excepciones es indispensable en aplicaciones que dependen de la interacción con el usuario. De igual manera, representar el funcionamiento de una aplicación mediante diagramas UML fomenta habilidades de documentación más formales.

- **Objetivos:**
  Interpretar el uso y el propósito específico del manejo de excepciones dentro de un programa, identificar las implementaciones de conceptos fundamentales de la Programación Orientada a Objetos en Dart, así como desarrollar diagramas UML (estático y dinámico) que faciliten la comprensión de la estructura del programa y como parte de la documentación formal del mismo, culminando así el proceso de migración de Java al ecosistema de Dart y Flutter.

2. Marco Teórico

2.1. Abstracción

La abstracción consiste en identificar las características esenciales de un objeto y omitir detalles que no son relevantes para el problema que se desea resolver. Esto nos permite diseñar clases que representan conceptos generales y establecer una base común para clases más específicas.

2.2. Encapsulamiento

El encapsulamiento es el mecanismo que protege los datos internos de un objeto, restringiendo su acceso mediante modificadores de visibilidad. En Dart, el prefijo "_" indica que un atributo es privado a nivel de archivo. El uso de getters y setters permite validar datos y mantener la integridad de los valores asignados.

2.3. Herencia

La herencia permite crear nuevas clases basadas en una clase ya existente, reutilizando sus atributos y métodos. Gracias a este principio, las clases derivadas pueden extender o redefinir comportamientos. En este proyecto, `Auto`, `Moto` y `Camion` heredan de la clase abstracta `Vehiculo`.

2.4. Polimorfismo

El polimorfismo permite que distintas clases implementen métodos con el mismo nombre pero con comportamientos diferentes. Esto facilita procesar objetos heterogéneos de manera uniforme. En esta práctica, cada tipo de vehículo implementa su propia versión de los métodos `calcularServicio()` y `generarReporteServicio()`.

2.5. Interfaces

Una interfaz define un conjunto de métodos que una clase debe implementar sin especificar su comportamiento. En Dart, las interfaces ayudan a establecer contratos claros dentro del diseño del software. La interfaz `ServicioTaller` asegura que todos los vehículos posean los métodos necesarios para calcular costos y generar reportes.

2.6. Manejo de Excepciones

El manejo de excepciones permite identificar y gestionar errores durante la ejecución del programa. Dart ofrece mecanismos como `throw`, `try`, `catch` y `finally` para controlar situaciones inesperadas. En esta práctica, se utilizan para validar datos del usuario y evitar fallos en el flujo del programa.

2.7. UML (Lenguaje de Modelado Unificado)

El Lenguaje de Modelado Unificado (UML) es un estándar utilizado para representar visualmente la estructura y el comportamiento de un sistema. Los diagramas de clases muestran relaciones jerárquicas y atributos, mientras que los diagramas de secuencia representan la interacción dinámica entre objetos. Su uso facilita la documentación y comprensión del diseño del software.

3. Desarrollo

La aplicación desarrollada dentro del paquete `mx.unam.fi.poo.p910` permite el registro, cotización y la generación de reportes de vehículos ingresados al sistema implementando los conceptos de herencia, polimorfismo, encapsulamiento y `manejo de errores`. Con el propósito de explicar la lógica y las herramientas usadas en cada parte del código el desarrollo se divide en estas secciones:

3.1. Clases

Las clases de la aplicación definen los métodos que se ocuparán dentro del sistema y determinan el tipo de vehículos que se pueden registrar.

3.1.1. ServicioTaller

`ServicioTaller` es una interfaz que define dos métodos importantes que cada clase que represente a un vehículo se verá obligada a definir. Los métodos en cuestión son **calcularServicio()** y **generarReporteServicio()**.

3.1.2. Vehiculo

`Vehiculo` es una clase abstracta que implementa la interfaz `ServicioTaller` y es la superclase encargada de heredar a las clases que definen cada tipo de vehículo. Los atributos que incorpora son **marca**, **modelo** y **anio**, y al tener un encapsulamiento a nivel de archivo denotado por "_" usado como prefijo en los atributos, define sus getters y setters, haciendo uso del concepto de `manejo de errores` con **throw** para lanzar una excepción y detener el flujo del código implementando así validaciones en los setters que incluyen que **marca** y **modelo** no puedan ser vacíos y que **anio** no pueda ser menor a 1960, manteniendo así integridad y coherencia en los datos ingresados. Por último implementa le método **descripcion()** que devuelve los valores de los atributos ingresados.

3.1.3. Auto

`Auto` es una clase que extiende sus métodos y atributos de la clase **Vehiculo** añadiendo el atributo **_tieneAireAcondicionado** donde "_"marca que solo es accesible desde su archivo, por lo cual se crean getters y setters, además de sobreescribir el método **descripcion()** para agregar el nuevo atributo y definir los métodos **calcularServicio()** el cual calcula y regresa el total del monto a pagar sumando un precio base y un costo extra que depende de si tiene A/C y **generarReporteServicio()** que regresa en modo de cadena los atributos del vehículo y el monto a pagar.

3.1.4. Moto

`Moto` es una clase hija de **Vehiculo**, heredando así sus atributos y métodos, agregando un atributo de tipo entero **_Cilindrada** para el cual se contruyen su get y set, añadiendo en el set un **throw** para lanzar un error y manejar el caso en que se quiera ingresar una clindrada negativa al sistema.

Por otro lado los métodos que sobreescribe son **descripcion()** para agregar el nuevo atributo, **calcularServicio()** en el que se calcula y devuelve el monto total sumando una tarifa base y un costo extra que se suma en el caso de que la moto tenga arriba de 600 de cilindrada y **generarReporteServicio()** en el que se devuelven como String los atributos de la moto y el total a pagar.

3.1.5. Camion

`Camion` es una clase hija de **Vehiculo**, heredando sus métodos y atributos, añadiendo el atributo de tipo double **capacidadToneladas**, para el cual se designan sus respectivos get y set, incluyendo **throw** para arrojar un error en el caso de que se ingrese una capacidad negativa.

Por otra parte los métodos que sobreescribe, al igual que en los clases anteriores, son **descripcion()** para añadir le atributo **capacidad** a la descricpión del vehículo, **calcularServicio()** para calcular y regresar el total a pagar, sumando un costo base más un extra que depende de que la capacidad del camión sea mayor a 10 toneladas y **generarReporteServicio()** que devuelve en forma de String los atributos del vehículo y el total a pagar.

3.2. Funciones

En el código se implementaron nueve funciones con diferentes propósitos que permiten una ejecución más limpia y organizada desde el main:

Cuatro funciones se dedicaron a hacer más clara la lectura y captura de los distintos tipos de datos ingresados desde la interfaz de línea de comandos (CLI): **leerLinea()** que devuelve un String y en caso de ser NULL asigna un String vacío, **leerEntero()** se asegura de que se ingrese un entero convirtiendo la línea a entero con `tryParse` y haciendo un bucle con while hasta que se ingrese un valor válido, **leerDouble()** es el mismo caso de **leerEntero()** pero con double y por último **leerBoolSN()** que hace minúsculas las líneas ingresadas siendo que si es "s"devuelve verdadero, si es "n"devuelve falso y repite el proceso hasta que se ingrese alguna de esas dos opciones válidas.

Por otra parte tres funciones tienen como objetivo la creación de objetos de los tres tipos de vehículos aceptados por el sistema: **crearAutoIntercativo()**, **crearMotoIntercativa()** y **crearCamionInteractivo()**, cada uno muestra la leyenda "Ṙegistro" y pide llenar los atributos correspondientes de cada tipo de vehículo, los captura con ayuda de las funciones anteriores y regresa un objeto de tipo vehículo con las especificaciones de cada objeto.

Por último, hay dos funciones que cumplen el propósito de mostrar un listado básico y los reportes de todos lo vehículos ingresados al sistema, estas son **mostrarListadoBasico()** y **mostrarReportesDetallados()**, ambos son de tipo void y recorren una lista que se les entrega como argumento, pero la primera funcion imprime una descripcion más el monto a pagar, mientras que la segunda genera y muestra un reporte de cada objeto contenido en la lista.

3.3. Main

El Main o la función principal es donde todo toma forma y las opciones que puede realizar el usuario se estructuran en un `switch`:

1)Registrar Auto
2)Registrar Moto
3)Registrar Camión
4)Ver flotilla (resumen)
5)Ver reportes detallados
0)Salir

Las primeras tres opciones implementan un **try/catch** para manejo de errores, cada uno, dentro del **try**, genera un nuevo objeto vehículo con la ayuda de su función crear respectiva y se agrega a la lista `flotilla`, en caso de ser exitoso se manda un mensaje de confirmación, pero en caso de haber algún error, este se almacena en la variable e que es lanzada y manejada por el **catch** imprimiendo el error que no permitió el registro.

Para las opciones 4) y 5) solo hacen uso de las funciones **mostrarListadoBasico()** y **mostrarReportesDetallados()**, implementando el concepto de polimorfismo pues dentro de estos métodos cada uno aplica su propia versión de **calcularServicio()** y **generarReporteServicio()**. Por último el programa termina una vez que el usuario decide presionar 0 en el menú para salir del sistema.

Diagrama de clases (UML estático)

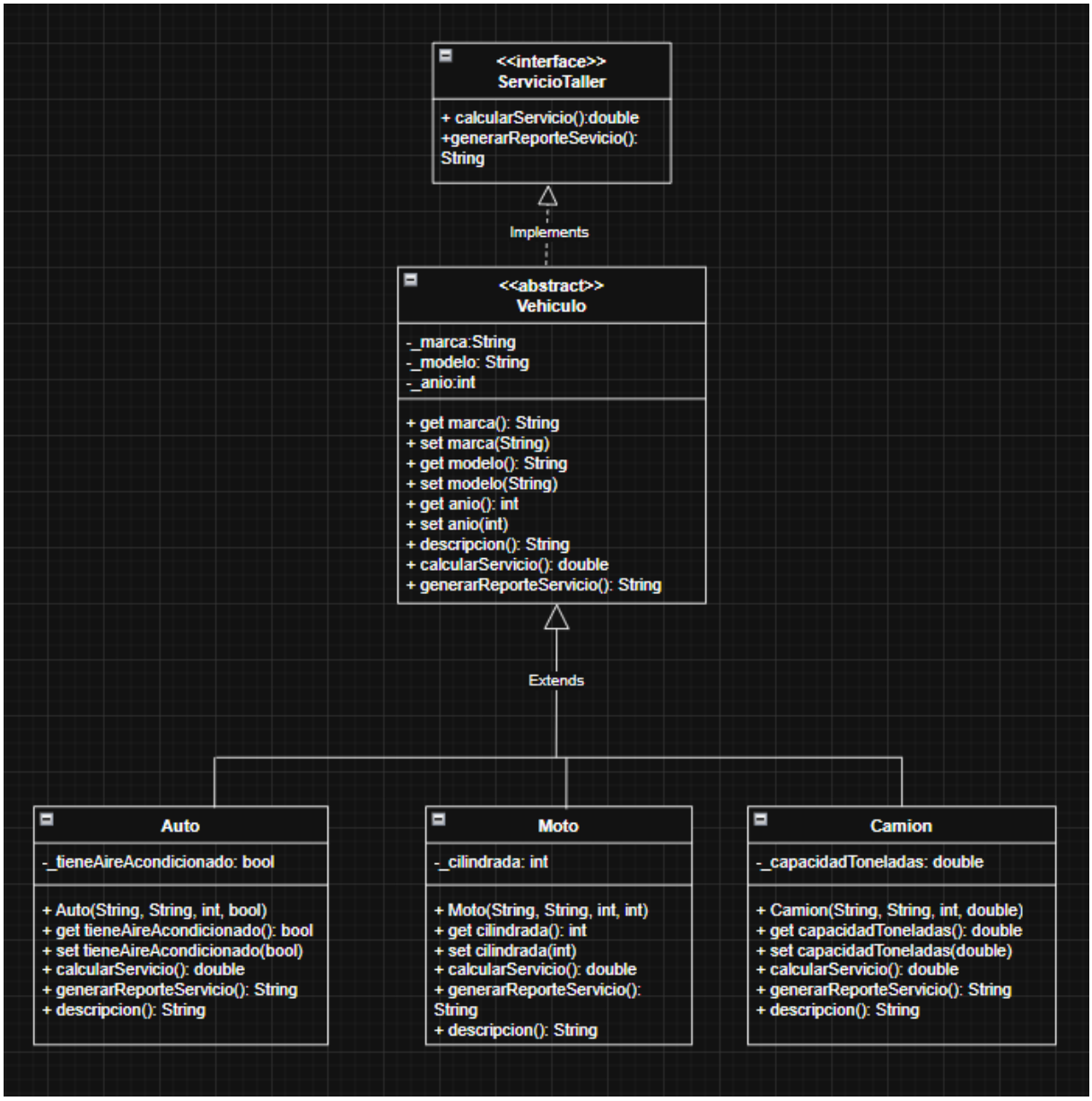

Figura 1: UML:Diagrama de clases.

Diagrama de secuencia (UML dinámico)

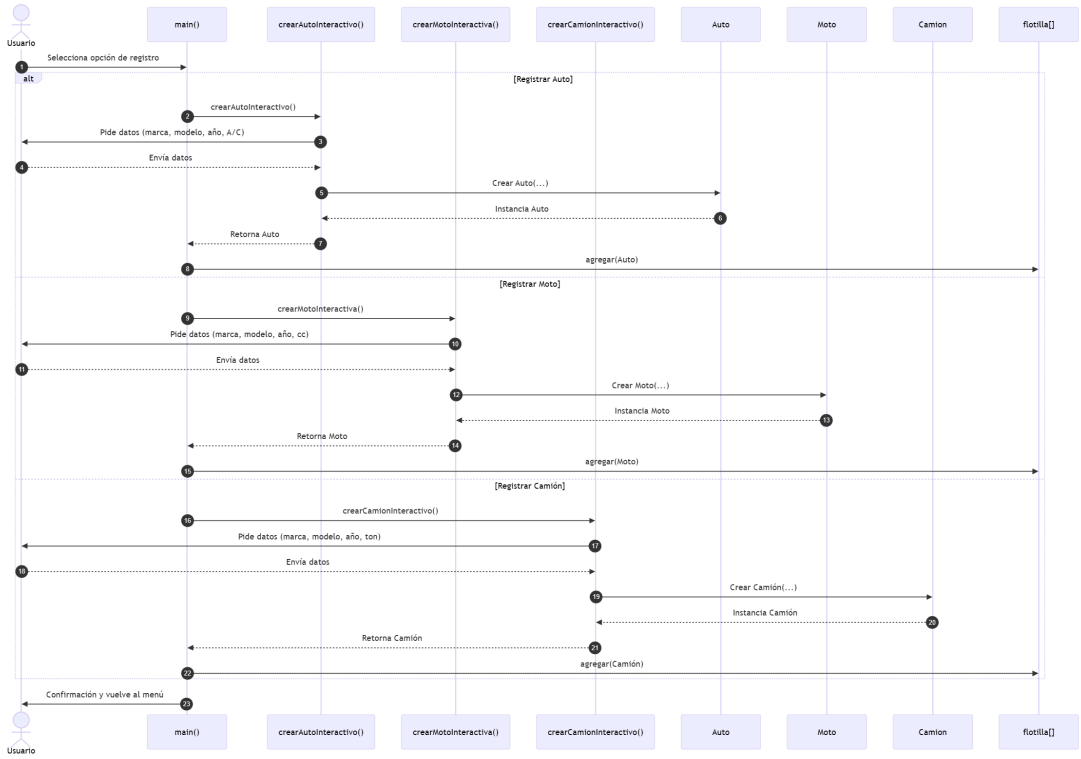

Figura 2: UML: Diagrama de secuencia.

4. Resultados

```
==================================
   SISTEMA PARA TALLER MECÁNICO
==================================
1) Registrar Auto
2) Registrar Moto
3) Registrar Camión
4) Ver flotilla (resumen)
5) Ver reportes detallados
0) Salir
Elige una opción: 1

== Registro de Auto ==
Marca: Dodge
Modelo: Challenger SRT
Año: 2023
¿Tiene aire acondicionado? (s/n): s

[OK] Auto agregado.

Presiona ENTER para continuar...
```

Figura 3: Agregando un auto al sistema de taller mecánico.

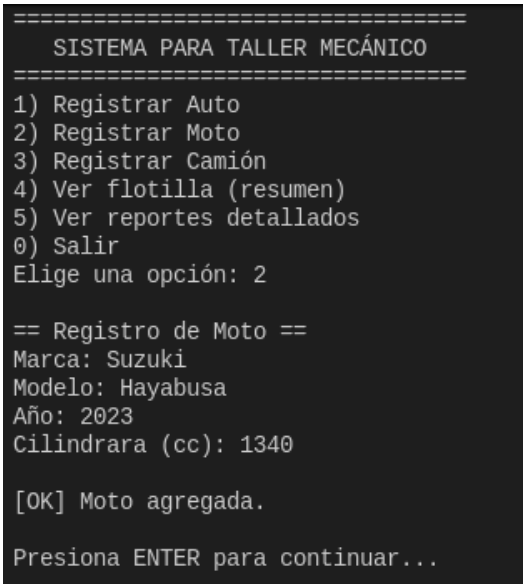

Figura 4: Agregando una moto al sistema de taller mecánico.

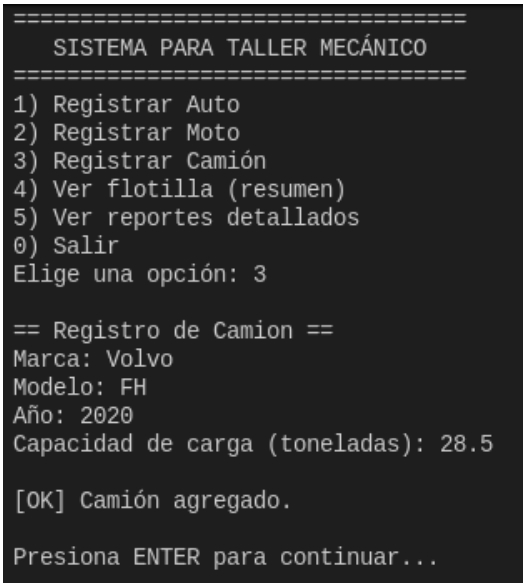

Figura 5: Agregando un camión al sistema de taller mecánico.

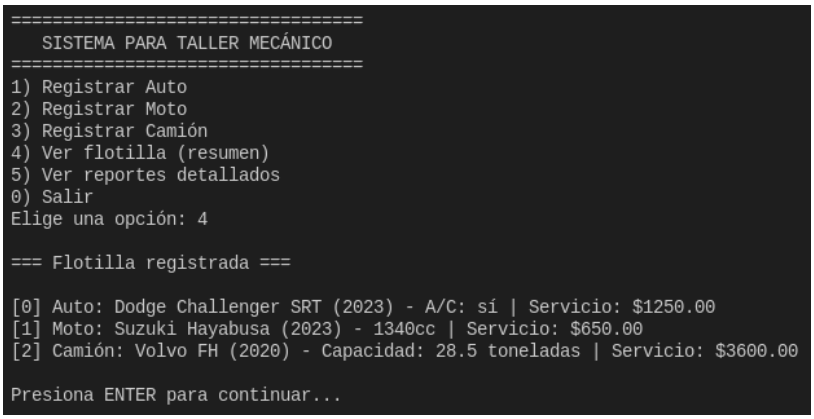

Figura 6: Imprimiendo un resumen de la flotilla.

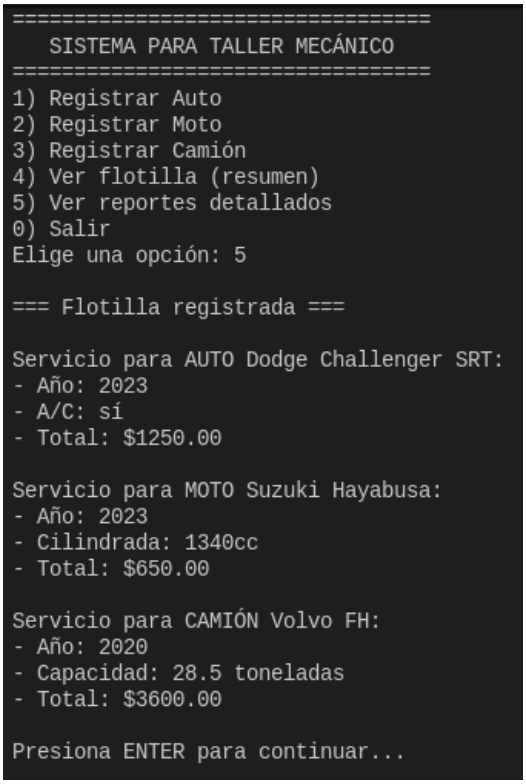

Figura 7: Imprimiendo reporte detallado de la flotilla.

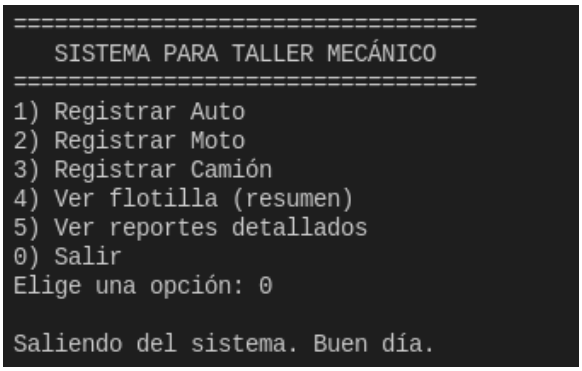

Figura 8: Saliendo del sistema de taller mecánico.

5. Conclusiones

Esta práctica supuso la migración de Java al ecosistema de Dart y Flutter, lo que permitió identificar similitudes y diferencias clave en la aplicación de los principios de la Programación Orientada a Objetos entre ambos lenguajes. El programa emplea principios fundamentales como las clases abstractas, interfaces, herencia y polimorfismo, lo cual permitió establecer una conexión con los conceptos teóricos abordados en clases, particularmente para el caso de las excepciones, se logró evidenciar su importancia en un programa para garantizar un flujo correcto de operaciones durante la ejecución.

Además, como parte de la documentación de la práctica, los diagramas UML permitieron visualizar la estructura general del programa con el diagrama estático de clases y el flujo de interacción entre el programa y el usuario con el diagrama dinámico de secuencia de una manera práctica e intuitiva. Esto demostró la utilidad del UML para la comprensión del diseño de software y reafirmó la importancia de mantener una documentación formal en los proyectos.

Con esta práctica se pudo comprobar que los principios esenciales de la Programación Orientada a Objetos se mantienen constantes y aplicables, independientemente del entorno utilizado.

Referencias

[1] Google Developers, *Dart Language Tour*. Disponible en: `https://dart.dev/language` [Consultado: 13-nov-2025].

[2] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3rd ed., Addison-Wesley, 2003.

[3] D. van der Linden, *Effective Dart: Usage*. Disponible en: `https://dart.dev/guides/language/effective-dart/usage` [Consultado: 13-nov-2025].