	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor(a): René Adrián Dávila Pérez

Asignatura: Programación Orientada a Objetos

Grupo: 7

No. de práctica(s): Práctica 9 y 10

Integrante(s): 320065570
425133840
423020252
322229916
425032224

No. de lista o brigada: 3

Semestre: 2026-1

Fecha de entrega: 16/11/2025

Observaciones:

CALIFICACIÓN: _____

Índice

1. Introducción

Una parte esencial para comprender los diseños y la arquitectura de ciertos códigos son los diagramas UML(Lenguaje Unificado de Modelado), ya que estos permiten representar de manera visual tanto los elementos estáticos como clases, atributos, relaciones y los elementos dinámicos, entre ellos el flujo de interacción entre los objetos o la secuencia de ejecución. En esta materia revisamos el archivo Dart proporcionado por el Profesor, el cual es un codigo que simulara un sistema de registro de ciertos servicios solicitados, donde se pueden identificar diversos componentes con una organización y funcionamiento que pueden ser explicados mediante diagramas UML estáticos, diagramas de clases, diagramas UML dinámicos y diagramas de secuencia o de actividad. El manejo de excepciones dentro del código es un aspecto esencial para garantizar la robustez y estabilidad de la aplicación. A través de la implementación de estructuras y el lanzamiento de errores personalizados, el programa es capaz de anticipar y detectar situaciones inesperadas durante la ejecución. En este reporte observaremos estos elementos dentro del código Dart estudiado, destacando su estructura, su comportamiento dinámico y la forma en que se implementan las excepciones.

2. Marco Teórico

2.1. Excepciones

Las excepciones indican que ocurrió un imprevisto – son condiciones irregulares que alteran el flujo normal del programa en ejecución. Estos errores pueden ser generados por la lógica del programa, como un índice de un arreglo fuera de su rango, una división entre cero o errores producidos por los mismos objetos que denuncian algún tipo de estado no previsto o condición que no tienen permiso de manipular.

Varios lenguajes –Java, Python, TypeScript, Dart, etc.– tienen la funcionalidad de reconocer la desviación del flujo de ejecución, ocasionado por las excepciones, y realizar alguna acción a partir de ello, sea terminación de la ejecución o, si se trata de excepciones denominadas como "comprobadas", la corrección del flujo.

2.2. Dart

Dart es un lenguaje de programación *open-source* desarrollado en Google con un enfoque orientado hacia la accesibilidad, portabilidad y productividad – esto con el fin de facilitar la creación aplicaciones de alta calidad en cualquier plataforma.

El paradigma de programación principalmente representado en este lenguaje es el de programación orientada a objetos, por lo cual, es posible implementar los conceptos de herencia, polimorfismo, encapsulación, etc. En el caso de la encapsulación, los atributos pueden declararse con acceso privado usando un gión bajo como prefijo del nombre del atributo. Para definir los métodos *setter* y *getter* de un atributo privado, se utilizan las palabras clave `set` y `get` de la siguiente manera:

```
set identificador(tipo_dato nombre_argumento) {
    ...
}


tipo_dato get identificador => _nombre_atributo
```

En Dart todas las excepciones son excepciones no comprobadas. Los métodos no de-

claran qué excepciones pueden lanzar y no es necesario que sean capturadas cuando dichos métodos son llamados.

Dart nos brinda distintos tipos que abstraen las excepciones, como `Exception` y `Error`, además de varios subtipos predefinidos. Es posible definir excepciones propias. Sin embargo, los programas de Dart pueden lanzar cualquier objeto no nulo —no solo objetos de tipo `Exception` y `Error`— como excepción.

En Dart el manejo de excepciones se realiza mediante los mecanismos accesibles a través de las siguientes palabras clave:

- `try`: Encapsula la parte del código que podría causar una excepción. Si ocurre una excepción dentro de este bloque, la ejecución normal se detiene y el control pasa al bloque `catch` u `on`.

- `catch`: Captura la excepción generada en el bloque `try`. Recibe el objeto de la excepción.

- `on`: Se usa para controlar tipos de excepciones concretas. Es posible tener varios bloques `on` para distintos tipos de excepciones, seguidos opcionalmente por un bloque `catch` genérico.

- `finally`: Contiene el código que siempre se ejecuta, independientemente de si se generó una excepción o no. Se usa comúnmente para tareas de limpieza, como cerrar archivos o conexiones de red.

- `throw`: Se usa para lanzar una excepción de forma manual. Se puede lanzar cualquier objeto no nulo, aunque se recomienda lanzar objetos de tipo `Exception` o `Error`.

3. Desarrollo

3.1. Funcionalidad

El programa elaborado simula un sistema de registro de servicios solicitados para diferentes tipos de vehículos – específicamente automóviles, motocicletas y camiones. La interfaz del programa está a base de la entrada/salida estándar del sistema operativo del equipo de cómputo. La interacción con el usuario se da de la siguiente manera: se muestran las funciones que el sistema es capaz de llevar a cabo –registrar un automóvil para servicio, registrar una motocicleta para servicio, registrar un camión para servicio, mostrar un resumen del registro de servicios y mostrar un reporte detallado de cada entrada del registro–, el usuario indica la salida del sistema ó la función que desea ejecutar ingresando el número correspondiente a la opción elegida, lo cual lleva al sistema a realizar la acción establecida.

El punto de entrada del programa es el método `main`, donde se lleva a cabo la funcionalidad principal, la cual está contenida en un bucle infinito. Primero, se despliega a la salida estándar el menú con las opciones mencionadas anteriormente, mediante llamadas consecutivas a la función `print`. Después, se instruye leer la entrada estándar, y utilizando la estructura de control `switch` se definen los protocolos a seguir con base en el resultado de la lectura. En los casos cuando el resultado es el número 1, 2 ó 3, se crea un nuevo objeto de la clase que representa un automóvil, motocicleta ó camión (respectivamente) y se agrega a la lista de vehículos. En los casos cuando el resultado es el número 4 ó 5, se muestra ó el resumen del registro de vehículos y los correspondientes servicios ó los reportes detallados de los servicios brindados a los vehículos. Cuando el resultado es el número 0, el bucle se rompe y termina el programa. Los métodos que se encargan de leer la entrada estándar, crear las entradas del registro y exponerlo se describen a continuación.

El método `leerLinea()` recibe como argumento una cadena, la cual es escrita a la salida estándar con la función `stdou.write()`, lee la entrada estándar mediante el método `readLineSync()` y si se logra obtener un resultado, lo regresa. En el caso de los métodos `leerEntero` y `leerDouble`, contienen instrucciones similares a las anteriores y además, intentan extraer un valor de su respectivo tipo de dato del resultado de la lectura, el cual

es regresado si la extracción es exitosa. El método `leerBoolSN()` funciona de manera casi idéntica a los dos métodos anteriores, sin embargo, este regresa `true` si el resultado de la lectura de la entrada estándar es el carácter 's' y `false` si es 'n'.

Cada una de los métodos encargados de crear una entrada en el registro de vehículos –`crearAutoInteractivo()`, `crearMotoInteractiva()`, `crearCamionInteractivo()` – utilizan los métodos de lectura descritos anteriormente para obtener valores con los cuales crean un nuevo objeto de la clase que representa su tipo de vehículo respectivo – posteriormente lo agregan a la lista de vehículos.

En cuanto a la exposición del registro de vehículos, se definen dos métodos, los cuales reciben como argumento la lista de vehículos e iteran a través de ella, si es que o está vacía: `mostrarListadoBasico()` que escribe a la salida estándar, con un formato específico, los datos generales de cada vehículo y su costo de servicio, haciendo una llamada sus métodos `descripcion()` y `calcularServicio()`, y `mostrarReportesDetallados()` el cual escribe a la salida estándar la salida del método `generarReporteServicio()` del vehículo.

3.2. Clases

La interfaz `ServicioTaller` define el contrato que deben cumplir todos los vehículos registrados en el sistema. Se declaran los métodos abstractos `calcularServicio()` para determinar el costo del servicio y `generarReporteServicio()` para crear un reporte detallado.

En la clase abstracta `Vehiculo` se definen los atributos privados **marca**, **modelo** y **anio**, tal como los métodos *getter* y *setter* para cada uno de ellos. Todos los métodos *setter* lanzan una excepción de tipo `ArgumentError` si el atributo correspondiente contiene una cadena vacía – en el caso del atributo **anio**, esto pasa si su valor es un número menor que 1900. Se define un método adicional, `descripcion()`, el cual regresa una cadena que contiene los valores de los atributos de esta clase. Además, se utiliza la palabra clave `implements` con el fin de establecer que todas las subclases heredan la obligación de implementar los métodos declarados en la interfaz `ServicioTaller`.

A partir de la clase anterior, se define una subclase `Auto`, la cual define un atri-

buto privado `tieneAireAcondicionado` –de tipo `bool`–, define un método *setter* para este e implementa los métodos de la interfaz `ServicioTaller`. Dentro del método `calcularServicio()` se determina el costo de servicio, sumando 500.0 por la cuota base, 350.0 por el concepto de "alineación" y un recargo condicional de 400.0 si el atributo mencionado tiene el valor `true`.

La clase `Moto` es otra subclase de `Vehiculo`. Además de los atributos heredados define otro con el nombre **cilindrada**. El método *setter*, encargado de actualizar el valor del atributo mencionado, realiza una validación que comprueba que el valor nuevo sea un número positivo, lanzando una excepción de tipo `ArgumentError` con el mensaje "La cilindrada debe ser positiva..." si este no es el caso. También contiene una implementación del método `calcularServicio()` que computa el costo de servicio tomando como base un monto de 300.0 y sumando 150.0 por "ajuste de cadena". También aplica un recargo condicional de 200.0 cuando el atributo **cilindrada** tiene un valor mayor o igual a 600.

La última subclase, `Camion`, de manera similar a la subclase anterior, define un atributo privado adicional – **capacidadToneladas** , cuyo método *setter* correspondiente verifica que el valor a asignar al atributo sea un número positivo y lanza una excepción de tipo `ArgumentError` si no lo es (en este caso, el mensaje acompañante es "La capacidad debe ser positiva..."). Su implementación del método `calcularServicio()` calcula el costo de servicio a partir de una cuota base de 1200.0, sumando 800.0 por el concepto de "revisión de frenos" y 600 por "revisión de suspensión". También se toma en consideración un recargo condicional de 200.0 cuando el atributo **capacidadToneladas** tiene un valor mayor a 10.

Todas las subclases mencionadas sobre-escriben el método `descripcion()`, heredado de la clase `Vehiculo`, para añadir información acerca del atributo adicional que definen a la descripción definida en la clase padre. Igualmente, en todas las subclases la implementación del método `generarReporteServicio()` construye un reporte detallado, con un formato específico, que muestra la marca, modelo, año, el costo total de proveer servicio al vehículo calculado a dos cifras decimales significativas e información acerca de la propiedad del vehículo representada por el atributo único a cada subclase. Otra cosa que tienen en común las subclases mencionadas es el hecho de que definen un método *getter* para sus atributos únicos.

4. Resultados

En el siguiente diagrama se presenta la relación entre la clase `Vehiculo` y sus subclases `Auto`, `Moto` y `Camion`, descritas en el apartado anterior.

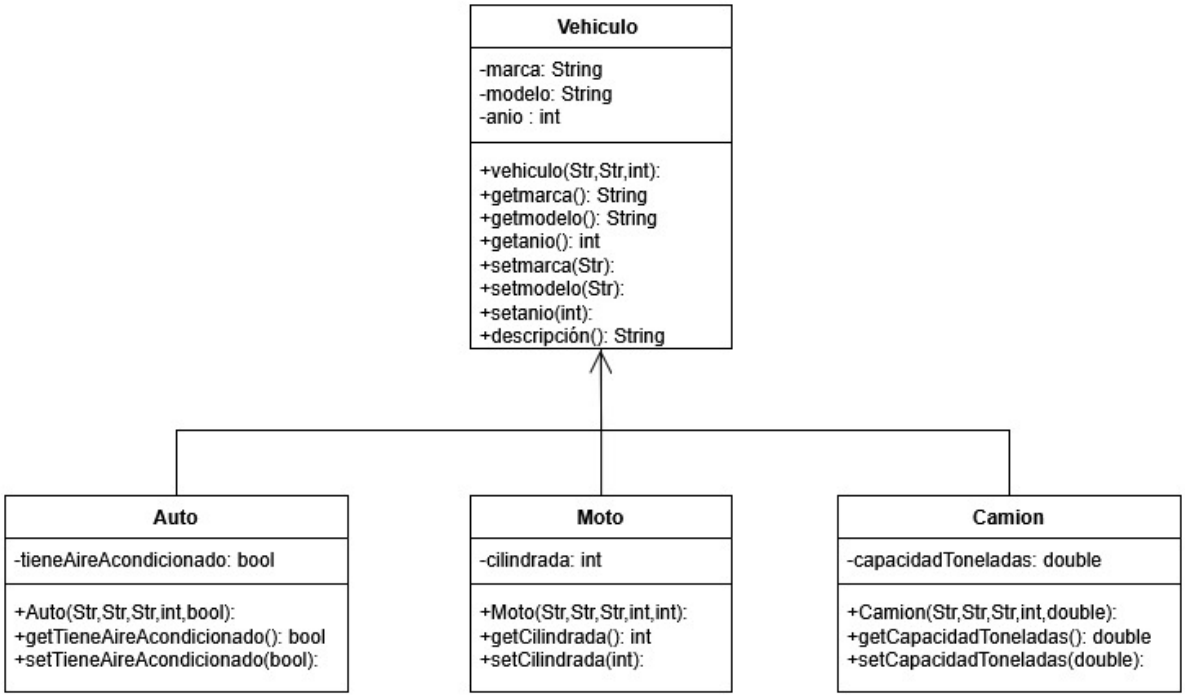

Figura 1: Diagrama de clases

Igualmente, en la figura 2 se presenta un esquema de la interfaz de usuario, con todas las opciones que puede tomar y la secuencia de estas dentro del código.

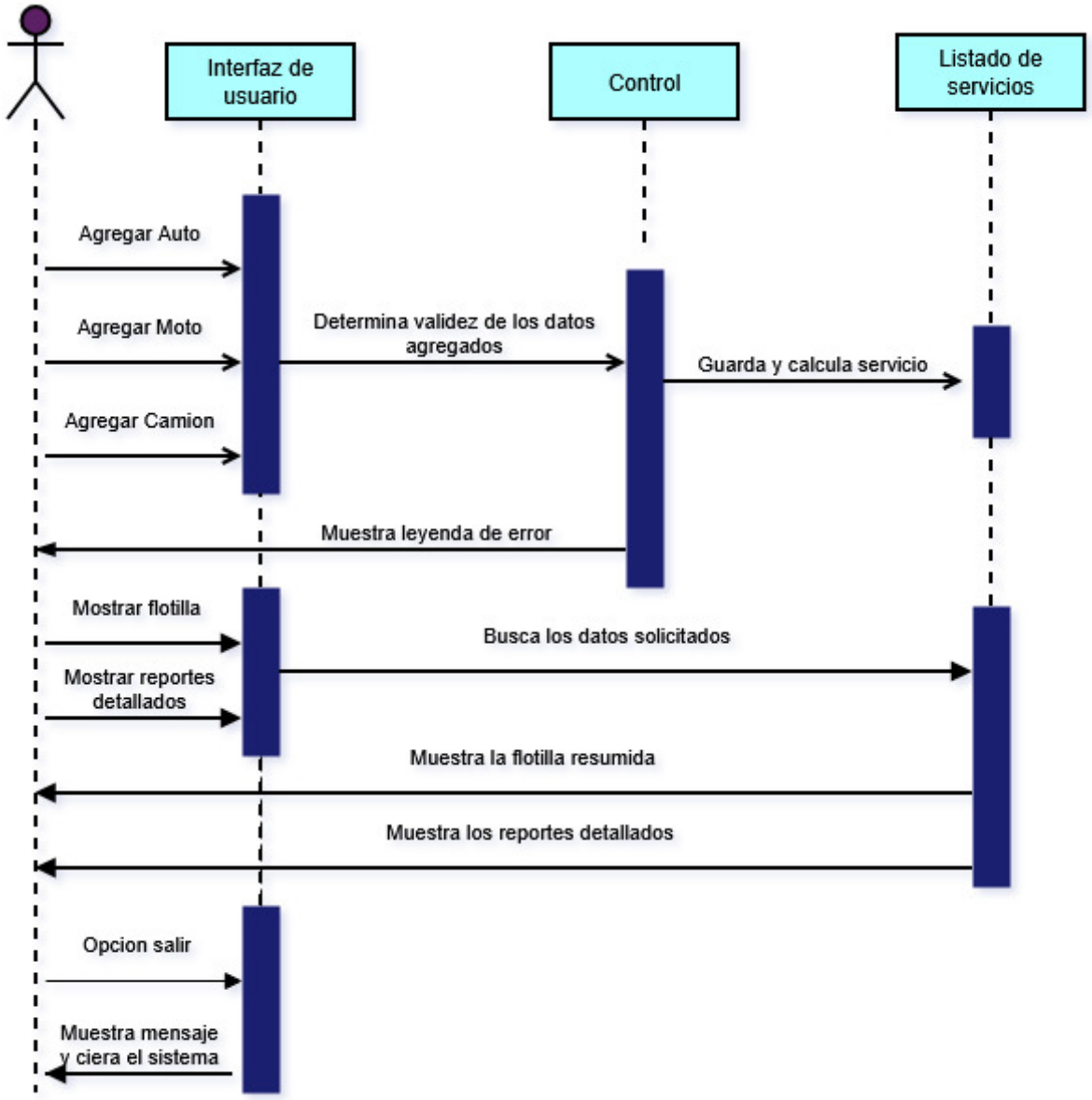

Figura 2: Diagrama de secuencia

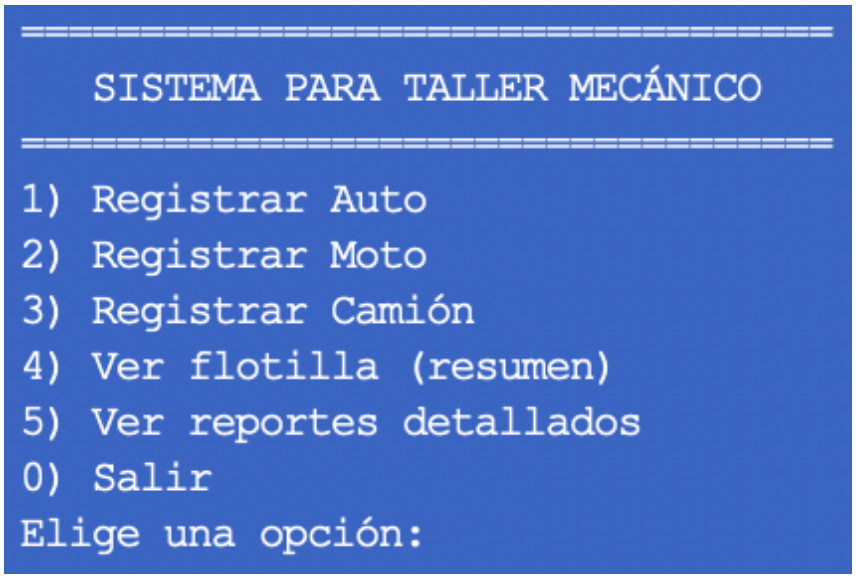

Figura 3: Menú inicial de funcionalidades

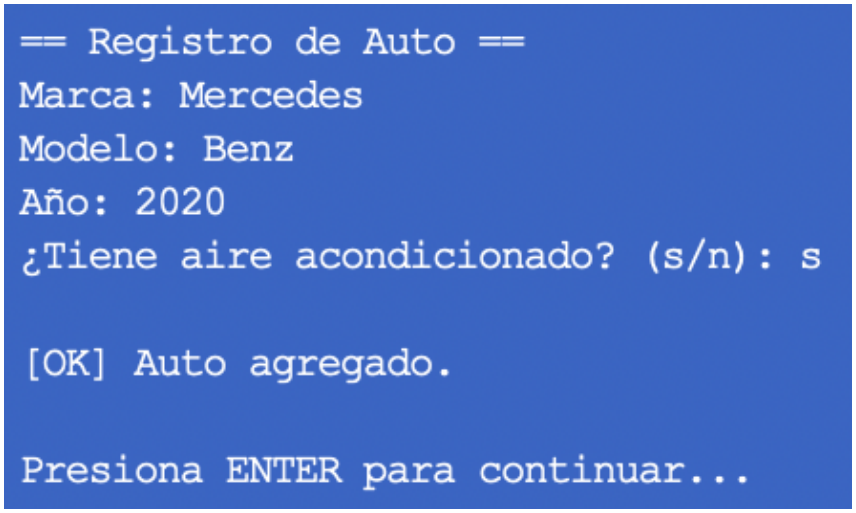

Figura 4: Registro de automóvil

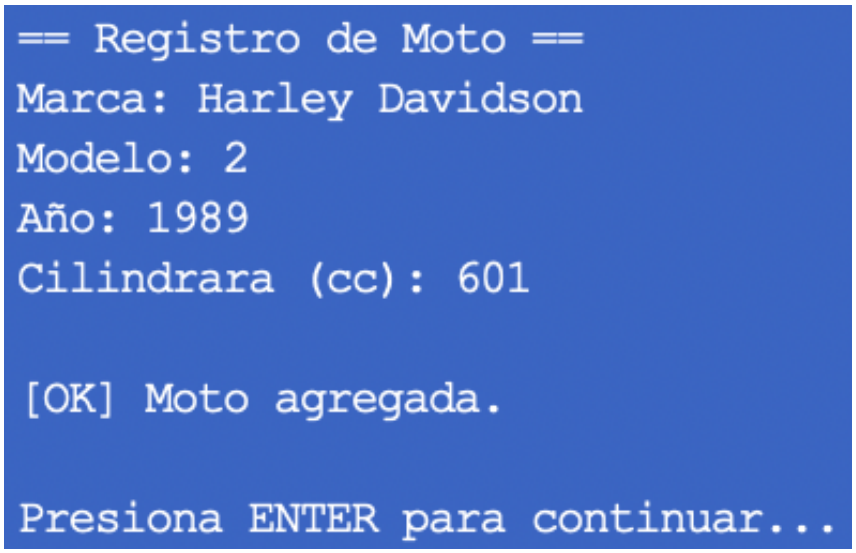

Figura 5: Registro de motocicleta

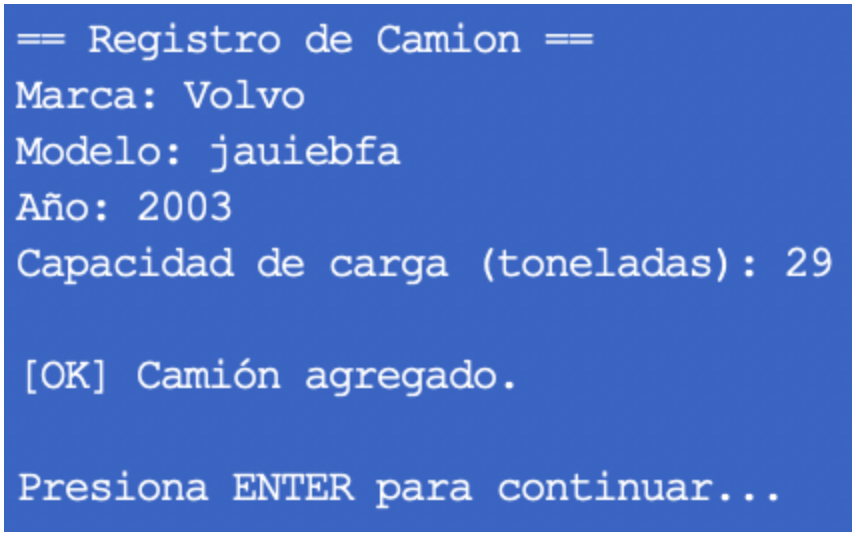

Figura 6: Registro de camión

```
=== Flotilla registrada ===

[0] Auto: Mercedes Benz (2020) - A/C: sí | Servicio: $1250.00
[1] Moto: Harley Davidson 2 (1989) - 601cc | Servicio: $650.00
[2] Camión: Volvo jauiebfa (2003) - Capacidad: 29.0 toneladas | Servicio: $3600.00
```

Figura 7: Resumen de vehículos

```
=== Flotilla registrada ===

Servicio para AUTO Mercedes Benz:
- Año: 2020
- A/C: sí
- Total: $1250.00

Servicio para MOTO Harley Davidson 2:
- Año: 1989
- Cilindrada: 601cc
- Total: $650.00

Servicio para CAMIÓN Volvo jauiebfa:
- Año: 2003
- Capacidad: 29.0 toneladas
- Total: $3600.00
```

Figura 8: Reportes detallados sobre servicios

5. Conclusiones

En esta practica, se aplicaron los fundamentos de el paradigma de la programación orientada a objetos en el lenguaje Dart, donde la transición desde en el lenguaje Java fue bastante intuitiva, debido a que las bases de un lenguaje a otro son muy similares.

Como ejemplo de implementación, la herencia mediante la clase abstracta Vehículo, y sus subclases `Auto`, `Moto` y `Camión`, así como el polimorfismo al sobrescribir los métodos, por ejemplo `calcularServicio()`, mantienen la misma lógica que en Java.

También la encapsulación y el manejo de excepciones están en ambos lenguajes, que aunque existen algunas diferencias en la sintaxis, son similares en implementación, pudiendo adaptarse fácilmente al uso en uno y otro lenguaje.

Esta práctica nos permitió reforzar la idea que los paradigmas de programación son transferibles desde un lenguaje a otro, ya que habiendo aplicado los mismos conceptos y estrategias anteriormente utilizadas en Java, pero ahora en el lenguaje Dart, es un ejemplo perfecto de la importancia del paradigma más que el lenguaje en sí mismo, donde un programador puede cambiarse de lenguaje manteniendo el paradigma.

Referencias

[1] https://repositorio-uapa.cuaed.unam.mx/repositorio/moodle/pluginfile.php/3085/mod_resource/content/1/UAPA-Excepciones-Errores/index.html

[2] *Error handling* s.f. Disponible en: https://dart.dev/language/error-handling

[3] *What's an Exception and Why Do I Care?*. s.f. Disponible en: https://www.cs.princeton.edu/courses/archive/spr96/cs333/java/tutorial/java/exceptions/definition.html

6. Datos de trabajo

Trabajo en el reporte:

- Introducción: Colaborada por 322229916
- Marco teórico: Colaborado por 425133840
- Desarrollo y resultados: Colaborado por 425032224
- Conclusiones: Colaborado por 423020252
- Diagramas: Colaborado por 320065570