	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorios de docencia

Laboratorio de Computación Salas A y B

Profesor(a): M.C. RENE ADRIAN DAVILA PEREZ

Asignatura: PROGRAMACIÓN ORIENTADA A OBJETOS

Grupo: 01

No de Práctica(s): 9 y 10

Integrante(s): 322118311

322094028

322092842

322078673

322067738

No. de lista o brigada: 03

Semestre: 2026-1

Fecha de entrega: 11/11/2025

Observaciones:

CALIFICACIÓN: _____

Índice

1. Introducción

- **Planteamiento del problema.** El problema a resolver en esta práctica es el análisis de una aplicación desarrollada en el lenguaje Dart que simula un sistema de gestión para un taller mecánico. El desafío central de este reporte es documentar la arquitectura de dicha aplicación mediante la creación de diagramas UML y analizar e interpretar cómo se aplica el concepto de manejo de excepciones para asegurar la integridad y validez de los datos ingresados al sistema

- **Motivación.**La documentación de software mediante diagramas UML es una práctica esencial en la ingeniería, ya que permite comunicar visualmente la estructura y el comportamiento de un sistema, facilitando su comprensión, mantenimiento y evolución.

- **Objetivos.** Los objetivos específicos de esta práctica son:

  Elaborar un diagrama UML estático que represente la estructura de las clases del código, sus atributos, métodos y las relaciones de herencia e implementación existentes.

  Diseñar un diagrama UML dinámico que ilustre una interacciones del sistema. Interpretar la aplicación de los conceptos de Programación Orientada a Objetos.

  Proporcionar una interpretación detallada del uso de excepciones en Dart.

2. Marco Teórico

2.1. Abstracción (Clases Abstractas e Interfaces)

La abstracción permite modelar la funcionalidad esencial sin preocuparse de los detalles de implementación.

Clase Abstracta

En Dart, se definen con la palabra clave `abstract`. Sirven como plantillas que no pueden ser instanciadas.

Listing 1: Ejemplo de Clase Abstracta en Dart

```
abstract class Figura {
  // Metodo abstracto (sin implementaci n)
  double calcularArea();
}
```

Interfaces

Una diferencia con Java es que Dart no tiene una palabra clave `interface`. En su lugar, cualquier clase puede actuar como una interfaz. Se utiliza la palabra clave `implements` para obligar a una clase a definir los métodos de otra. [1]

Listing 2: Ejemplo de Interfaz en Dart

```
// Clase que actua como interfaz
abstract class Dibujable {
  void dibujar();
}

// La clase implementa la "interfaz"
class Circulo implements Dibujable {
  @override
  void dibujar() {
    // Implementacion...
  }
}
```

2.2. Herencia

Una diferencia con Java está en el constructor de la subclase, que utiliza una lista de inicializadores (`:`) para llamar al constructor padre, en lugar de `super()` dentro del cuerpo.

Listing 3: Ejemplo de Herencia y constructor `super`

```
class Rectangulo extends Figura {
  double base;
  double altura;

  // Se llama a super() antes del cuerpo del constructor
  Rectangulo(this.base, this.altura) : super();

  @override
  double calcularArea() {
    return base * altura;
  }
}
```

[2]

2.3. Polimorfismo

Al igual que en Java, Dart, utiliza la anotación `@override` para indicar que un método de la subclase está sobrescribiendo al de la superclase.

Esto permite, por ejemplo, tener una lista de un tipo abstracto (`Figura`) que contenga objetos de tipos concretos (`Rectangulo`, `Circulo`) y que Dart ejecute la versión correcta del método (`calcularArea`) para cada uno.

Listing 4: Ejemplo de Polimorfismo en una Lista

```
List<Figura> misFiguras = [];
misFiguras.add(Rectangulo(10, 5));
// ...
// Dart sabe qu  m todo .calcularArea() llamar
for (var fig in misFiguras) {
  print(' rea : ${fig.calcularArea()}');
}
```

[2]

2.4. Encapsulamiento

Dart no usa `public` o `private`. La privacidad se define a nivel de biblioteca usando un guion bajo (`_`) al inicio del nombre del atributo o método.

Getters y Setters

Listing 5: Ejemplo de Encapsulamiento con Getters y Setters

```
class Persona {
  String _nombre; // Atributo privado

  // Getter publico
  String get nombre => _nombre;

  // Setter publico con validacion
  set nombre(String nuevoNombre) {
    if (nuevoNombre.isEmpty) {
      print("El nombre no puede ser vacio");
    } else {
      _nombre = nuevoNombre;
    }
  }
}
```

[3]

2.5. Manejo de Excepciones

El manejo de excepciones es el mecanismo para controlar errores en tiempo de ejecución de forma estructurada, evitando que la aplicación colapse. A diferencia de Java, todas las excepciones en Dart son *unchecked*.

`throw`

Se utiliza para señalar que ha ocurrido una condición de error. Se puede lanzar cualquier objeto, aunque comúnmente se usan `Exception` o `Error`, o tipos más específicos como `ArgumentError`.

Listing 6: Uso de `throw` en un setter

```
// Mejorando el setter de Persona
set edad(int nuevaEdad) {
  if (nuevaEdad < 0) {
    // Lanza una excepcion para senalar el error
    throw ArgumentError("La edad no puede ser negativa.");
  }
  _edad = nuevaEdad;
}
```

`try-catch` **(Capturar)**

Maneja los errores lanzados. El código que podría fallar se coloca en el bloque `try`. Si se lanza una excepción, la ejecución del `try` se detiene y salta al bloque `catch` donde se puede gestionar el error.

Listing 7: Uso de `try-catch` para capturar errores

```
// En el codigo principal
try {
  Persona p = Persona();
  p.edad = -5; // Esto lanzara la excepcion
  print("Edad actualizada"); // Esta linea no se ejecutara
} catch (e) {
  // 'e' captura el objeto lanzado (el ArgumentError)
  print("Error al registrar: $e");
}
print("El programa contin a...");
```

[2]

3. Desarrollo

3.1. Diagrama de Clases:

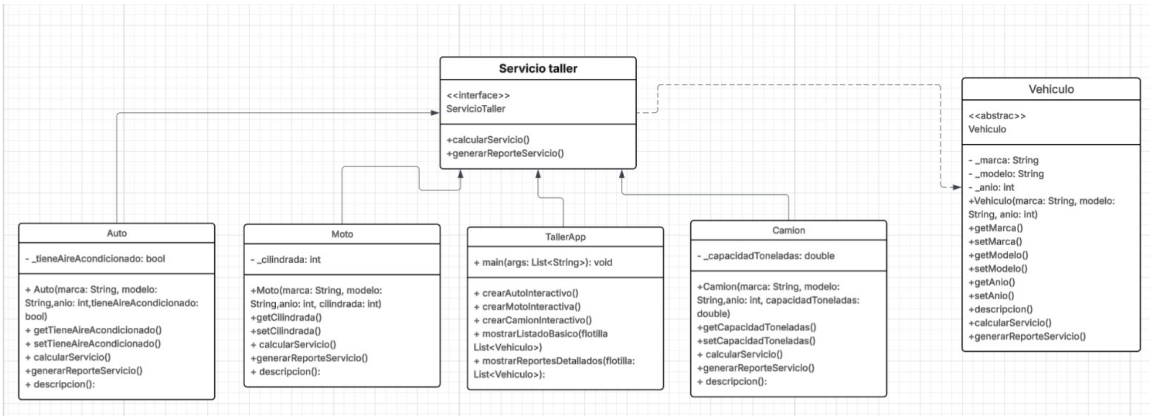

Figura 1: Diagrama de Clases

3.2. Diagrama de Secuencia:

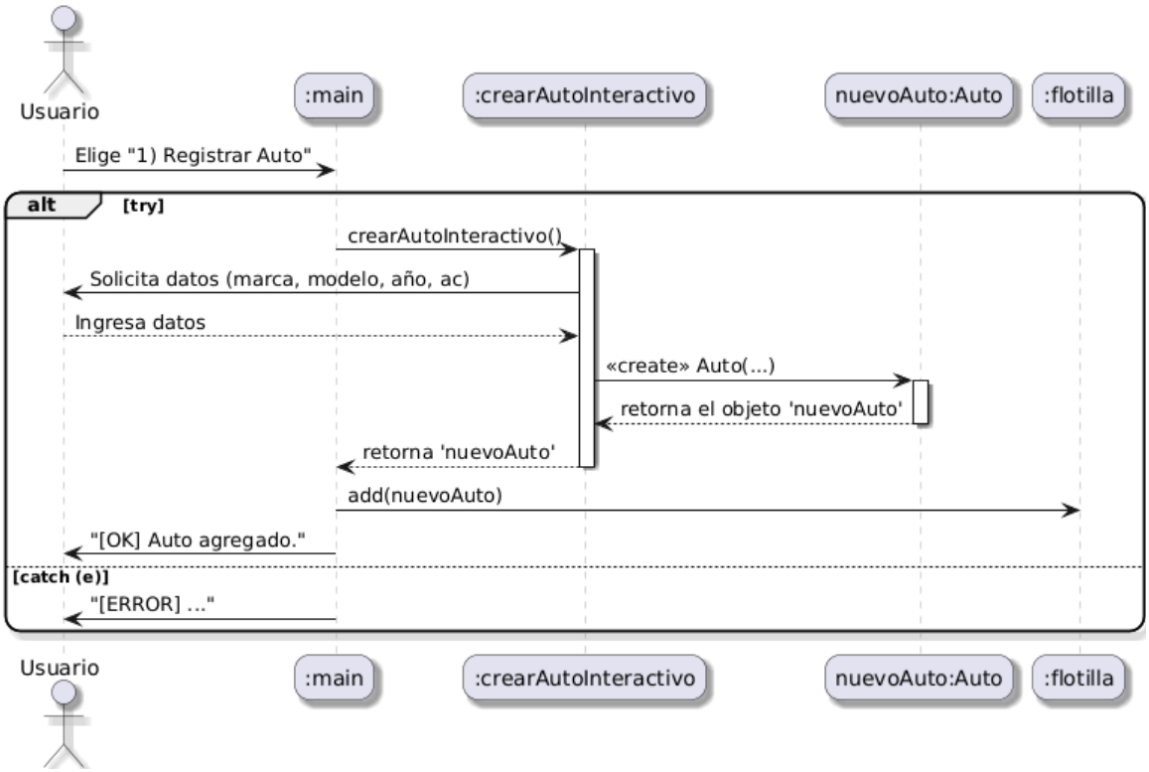

Figura 2: Diagrama de secuencia

3.3. Interfaz `ServicioTaller`

Esta es una clase abstracta. Define los métodos abstractos que establecen el çontrato"para los vehículos del taller: **calcularServicio()** y **generarReporteServicio()**. Esto obliga a que cualquier clase que la implemente deba proporcionar una lógica concreta para estos dos métodos.

3.4. Clase abstracta `Vehiculo`

Esta clase implementa la interfaz `ServicioTaller` y define los atributos principales que tendrán todos los tipos de vehículos. Iniciaremos encapsulando los atributos privados que tendrán todos los vehículos: de tipo String **_marca** y **_modelo**, y de tipo entero **_anio**. Posteriormente, se inicializan estos atributos mediante un método constructor.

Esta clase aplica la lógica de validación de datos directamente en sus métodos *setters*. Por ejemplo, **set anio(int nuevoAnio)** y **set marca(String nuevaMarca)** comprueban que los datos ingresados sean válidos). Si un dato es inválido, el *setter* correspondiente lanza (`throw`) una excepción de tipo **ArgumentError** para detener la asignación y reportar el error.

3.5. Clase `Auto`

Esta clase hereda de la clase abstracta `Vehiculo`. Define un Auto y añade el atributo privado específico **_tieneAireAcondicionado** de tipo booleano. Utiliza el constructor de la clase padre (vía `super()`) para inicializar los datos que tienen en común (marca, modelo, año).

Posteriormente, se sobrescriben e implementan los métodos abstractos: **calcularServicio()**, que define el ingreso sumando un costo base, alineación y un recargo si `_tieneAireAcondicionado` es verdadero; y **generarReporteServicio()**, que devuelve una cadena que identifica al objeto como "Servicio para AUTO.ªgregando sus detalles y el costo total.

3.6. Clase `Moto`

Esta clase también hereda de la clase abstracta `Vehiculo` añade el atributo privado **_cilindrada** de tipo entero. Su constructor utiliza `super()` para pasar la información personal básica al padre (marca, modelo, año).

Mediante sobrescritura se implementan los métodos **calcularServicio()**, que implementa la lógica de negocio para este tipo de vehículo calculando el pago con base en un costo de ajuste de cadena y un recargo si la cilindrada es alta; y **generarReporteServicio()**, que identifica al objeto como "Servicio para MOTO".

Tambienimplementa su propia lógica de validación en el *setter* **set cilindrada(int nuevaCildrada)** para asegurar que la cilindrada sea un valor mayor a 0, lanzando una `ArgumentError` en caso contrario.

3.7. Clase `Camion`

Esta clase también hereda de `Vehiculo` y añade el atributo privado **`_capacidadToneladas`** de tipo double. Su constructor utiliza `super()` para la inicialización de los datos comunes.

Sobrescribe **calcularServicio()** para incluir costos de revisión de frenos, suspensión y un recargo por carga pesada si `_capacidadToneladas` supera un umbral. También sobrescribe **generarReporteServicio()** para formatear los detalles del camión. Al igual que `Moto`, implementa validación en su propio *setter* para asegurar que la capacidad de toneladas sea un valor positivo.

3.8. Función `main` (Aplicación Principal)

Finalmente, en la función `main` se declara una lista `List<Vehiculo>` llamada `flotilla` para almacenar todos los vehículos registrados. El programa presenta un menú cíclico al usuario para registrar nuevos vehículos (Auto, Moto o Camión) o ver reportes.

La lógica de esta sección es el manejo de errores: cada vez que se intenta registrar un nuevo vehículo la llamada se envuelve en un bloque **try**. Si el usuario ingresa un dato inválido el *setter* correspondiente lanza la `ArgumentError`. Esta excepción es capturada por el bloque **catch**, que imprime un mensaje de error (`[ERROR] $e`) y permite que el programa continúe su ejecución y regrese al menú principal.

4. Resultados

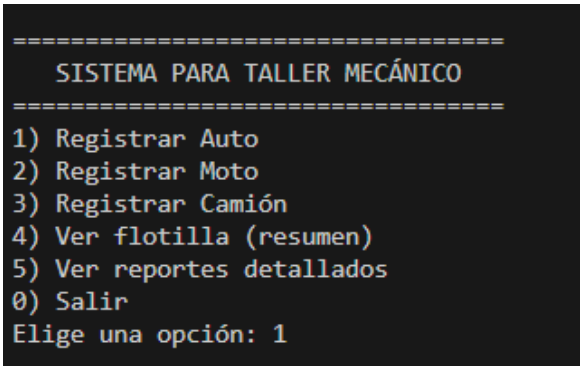

Figura 3: Menu

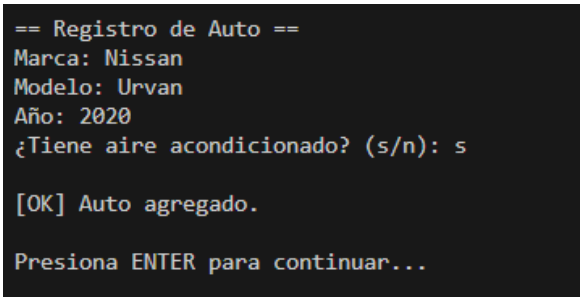

Figura 4: Ejemplo de vehiculo agregado

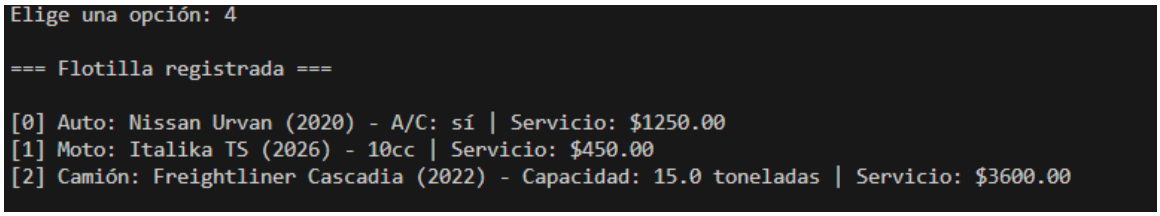

Figura 5: Salida de opcion 4

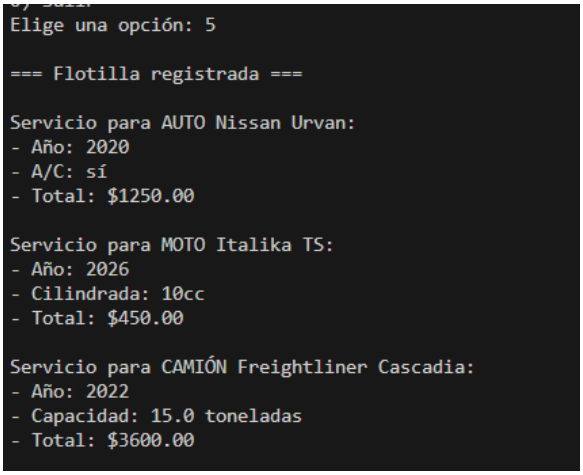

Figura 6: Salida de opcion 5

5. Conclusiones

Analizar el codigo proporcionado nos permite concluir que la aplicación de los conceptos teóricos de la Programación Orientada a Objetos es fundamental para resolver problemas de gestión de datos de una forma estructurada, escalable y mantenible. El uso del manejo de excepciones resulta eficiente al implementar las validaciones

Esta estrategia demuestra una clara separación de responsabilidades: las clases son responsables de definir y hacer cumplir sus propias reglas de negocio, mientras que la interfaz de usuario (la función main se encarga de capturar y gestionar dichos errores de una manera amigable para el usuario. Esto nos permite que nuestros programas eviten que datos inválidos corrompan el estado del sistema.

6. Bibliografía

- **[1]**E. Gamma, R. Helm, R. Johnson, y J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley, 1995.
- **[2]** Google, .Exceptions,"Dart Language Tour. [En línea]. Disponible: https://dart.dev/language/exceptions. (Accedido: 16 de noviembre de 2025).
- **[3]** Google, Çlasses,"Dart Language Tour. [En línea]. Disponible: https://dart.dev/language/classes. (Accedido: 16 de noviembre de 2025).