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

# Laboratorio de Computación
# Salas A y B

*Profesor(a):*

*Asignatura:*

*Grupo:*

*No de Práctica(s):* Práctica 9 - 10

*Integrante(s):* 322246320

322267567

322012051

322236688

322330872

*No. de lista o brigada:*

*Semestre*

*Fecha de entrega:*

*Observaciones:*

CALIFICACIÓN: _____

# Índice

# 1. Introducción

- **Planteamiento del Problema:** Se busca la implementación de una aplicación de Dart, que simule los servicios de un taller mecánico mediante diferentes clases en las que se manejarán métodos sobrescritos, excepciones y errores para asegurar un correcto funcionamiento del programa.
- **Motivación:** En esta práctica aprenderemos sobre la gestión de excepciones y errores en Dart para evitar comportamientos inesperados, crasheos del programa o simplemente enviar mensajes claros al usuario cuando ha ocurrido un problema.
- **Objetivos:** Crear un programa que garantice que nuestro programa es estable y sea capaz de responder ante situaciones inesperadas, todo esto con la finalidad de brindar al usuario una mejor experiencia a la hora de ejecutar.

# 2. Marco Teórico

- **Flutter y Dart:** Flutter es un kit de herramientas de interfaz de usuario de código abierto desarrollado por *Google* utilizado para crear aplicaciones. Tiene soporte para seis plataformas: iOS, Android, web, Windows, MacOS y Linux.

  Dart es un lenguaje de programación orientado a objetos desarrollado por *Google*, que también permite la programación estructurada. Este lenguaje fue diseñado para crear aplicaciones rápidas y fáciles de mantener. Sigue una sintaxis similar a C y Java.

  Flutter trabaja con el lenguaje Dart, y de esta forma se complementan para crear aplicaciones completas, robustas y eficientes. [1] [2]

- **Excepciones y errores:** Una excepción es cuando ocurre un evento que altera el flujo normal del programa en ejecución. Las excepciones ocasionan la finalización abrupta de la ejecución del programa, y muestran la razón por la cual se generó la excepción.

  Un error es un problema, del cual no se puede hacer nada para resolverlo, y suele tratarse de problemas mas profundos que el programador no puede abordar; por ejemplo, un error en la maquina virtual de Java.

  Existen mecanismos que permiten controlar y manejar cuando se 'lanza' una excepción; de esta manera se puede continuar con la ejecución del programa tomando en cuenta las posibles excepciones que podrían ser lanzadas.

  El manejo de excepciones se realiza con el bloque *try-catch-finally*. Dentro del bloque *try* se coloca el código que podría generar una excepción. En el bloque *catch* se especifica el tipo de excepción que será capturada y lo que realizará el programa en ese caso. En el bloque *finally* se coloca el código que siempre se ejecutará, independientemente de que ocurra el *try* o el *catch*. Estos bloques

pueden tener mas de un *catch*, y puede haber bloques sin el bloque *try* (*catch-finally*) o sin el bloque *catch* (*try-finally*).

Se puede lanzar una excepción explícitamente, con la palabra reservada *throw*, esto detiene el flujo del programa indicando un problema.

En Java, se puede declarar que un método podría lanzar excepciones con la palabra reservada *throws*. De esta forma no se manejan las excepciones dentro del método y son trabajo de los métodos que llamen a ese método. Si los métodos que llaman a esas funciones tampoco manejan las excepciones, se llama 'propagación de excepciones', donde la excepción es recorrida hasta el método que la maneje; y si no es manejada, se lanza la excepción y se termina la ejecución del programa como pasaría normalmente. [3]

- **UML:** UML, siglas de Lenguaje Unificado de Modelado, son un tipo de diagramas utilizados para modelar visualmente la sintaxis y semántica de un desarrollo de software complejo. Consiste en diferentes diagramas que describen los límites, estructura y comportamiento del sistema y los objetos que contiene. Los diagramas UML guardan una relación directa con la programación orientada a objetos. [4]

# 3. Desarrollo

El código proporcionado trae en un solo archivo distintas clases y funciones independientes, las cuales podrán ser usadas por cualquier otra clase y método del mismo código.

Lo primero es una clase abstracta **ServicioTaller**, que se utilizará como interfaz con dos métodos a implementar. Posteriormente, se tiene otra clase abstracta, **Vehiculo**, que implementa la primera clase haciendo uso de la palabra reservada *implements*. Dentro de esta clase abstracta se tienen tres atributos privados: *marca* (String), *modelo* (String) y *anio* (int). Se asignan como métodos privados con la barra baja antes del nombre, al igual que. La clase consta del constructor, los getters, los setters y un método **descripcion** que retorna lo que se obtiene de los getters, interpolando el valor que devuelven a Strings gracias al uso de '$'.

Los setters de la clase abstracta son donde se implementan las excepciones, ya que dentro de cada uno de estos setters se hace una evaluación, como que la marca y el modelo no estén vacíos o que el año sea mayor a 1900, en caso de no cumplir alguna de estas condiciones, se utilizará la palabra reservada *throw* para 'lanzar' una excepción del tipo *ArgumentError*, con un mensaje personalizado dependiendo del setter en el que se haya detectado un error. En caso de que no se cumpla con la condición para lanzar el *ArgumentError*, se asignará la variable de forma normal.

Luego se encuentra la clase **Auto** la cual hereda de **Vehiculo**, esta añade otro atributo privado de tipo *booleano*, el cual es el aire acondicionado. Su constructor ademas de inicializar su atributo único, se apoya del constructor de la clase padre, haciendo uso de *super*, teniendo ademas su propio set y get para su atributo único.

La clase consta ademas con la sobrescritura de los dos métodos declarados en la interfaz y del método **descripcion** de la clase padre: **calcularServicio** cuenta con dos variables con valores fijos y un tercero el cual cambiará su valor dependiendo de lo que contenga el atributo que representa el aire acondicionado, evaluandolo con una condición ternaria y devolviendo la suma de los tres atributos. **generarReporteServicio** solo devuelve de forma ordenada los atributos del objeto y el valor que regresa **calcularServicio**, limitándose a solo mostrar 2 dígitos luego del punto decimal. **descripcion** hace uso del método del mismo nombre de la clase padre y le añade ademas una parte de texto para mostrar si el auto cuenta con aire acondicionado o no, para devolver todo en una sola cadena de caracteres.

Luego se tiene a la clase **Moto** la cual también hereda de **Vehiculo**, realiza lo mismo que **Auto** pero intercambiando el aire acondicionado por el cilindrado de tipo *int*, lo que lleva a ciertos cambios, en el set se agrega la evaluación del cilindrado, para en caso de que se intente ingresar un cilindrado negativo, lanzar una excepción del tipo *ArgumentError* con el mensaje de que el cilindrado debe ser positivo, la sobrescritura de **calcularServicio** es bastante parecida a la de **Auto** ya que cuenta con valores fijos y evalúa en una condición ternaria el atributo único de la clase, y en **generarReporteServicio** y **descripcion** se añade la impresión del atributo único.

La ultima clase que se tiene es **Camion** la cual hereda de **Vehiculo** y es muy parecida a **Moto** cambiando el cilindrado por la capacidad de toneladas de tipo *double*, su propio constructor que también se apoya del de la clase padre y su propio set y get; teniendo en el set una evaluación la cual arrojará un *ArgumentError* con mensaje personalizado en caso de que la capacidad ingresada sea igual o menor que 0, la sobrescritura de **calcularServicio** cuenta con 3 variables con un valor fijo y una cuarta variable cuyo valor dependerá de la capacidad de toneladas. La sobrescritura de **generarReporteSericio** y **descripcion** es muy parecida a la de **Moto**.

Se tienen las funciones que podrá usar el código, primero se tiene **leerLinea** la cual recibe un *String* el cual muestra en consola, para en la misma linea leer lo que ingrese el usuario y almacenarlo en una variable linea de tipo *final* para terminar devolviendo lo que se guardó en linea o no devolviendo nada si es que el usuario no ingreso ningún dato.

**leerEntero** también espera recibir un *String* para dentro de un ciclo while el cual no se puede romper, imprimirlo en consola, leer lo que escriba el usuario y guardarlo en una variable *linea*, la cual en caso de ser nula se pasara a la siguiente iteración del ciclo; en caso contrario se creará la variable *valor* y se igualará a lo que almacena *linea* pero casteado a *int*. En caso de que la variable *valor* no sea nula, se devolverá lo que contiene, rompiendo el ciclo, y en cualquier otro caso se imprimirá que el valor ingresado es inválido; lo que realiza este método se replica en **leerDouble** pero cambiando el casteo de linea a tipo *int* por un casteo a tipo *double*.

**leerBoolSN** realiza algo parecido a los últimos dos métodos, pero en vez de crear una variable *valor*, crea una variable *l* y la iguala a lo que contenga linea pero sin espacios y en minúsculas, para posteriormente evaluar si contiene una 's' o una 'n' para devolver true o false respectivamente, o en todo caso imprimir que se ingrese una 's' o una 'n'.

Luego están las funciones **crearAutoInteractivo**, **crearMotoInteractiva** y **crear-**

**CamionInteractivo**, las cuales crean un objeto de tipo **Auto**, **Moto** y **Camion** respectivamente, haciendo uso de las funciones leer, para almacenar los valores que necesita cada objeto y terminar devolviendo el objeto creado con los valores dados.

Luego se tienen las funciones **mostrarListadoBasico** y **mostrarReportesDetallados**, ambas esperan recibir una lista de objetos de tipo **Vehiculo** y evalúan si esta vacía para imprimir que aun no se tienen registros y no hacer nada más; pero en caso de tener registros, ambas funciones recorrerán la lista dada, y en el caso de la primera función, imprimirá lo que devuelva el método **descripcion** y **calcularServicio** de cada objeto de la lista; mientras que en el caso de la segunda, imprimirá lo que devuelva el método **generarReporteServicio**.

Y por último, la función **main** crea una lista de objetos de tipo **Vehiculo** llamada flotilla, y mediante un ciclo while se imprimirá un menú de opciones, el cual hace uso de **leerEntero** para almacenar la opción del usuario. En las opciones 4, 5 y 0, solo se llama a su función respectiva o se termina el ciclo; sin embargo, las opciones 1, 2 y 3 se encargan de registrar un nuevo tipo de vehículo y tienen una evaluación de excepciones.

Las tres opciones, dentro de un *try*, crean un tipo de objeto **Vehiculo** igualado a lo que devuelva la función crear interactivo correspondiente, para luego añadirlo a la lista e indicar que el vehículo se agregó. Luego, dentro de un *catch*, atrapará lo que se haya lanzado en el primer *throw* ejecutado y se nombrará la excepción como *e*, para luego imprimir que ocurrió un error junto al mensaje personalizado que tenga *e*; por lo que de esta manera si se ingresó algún dato incorrecto al momento de registrar el vehículo, se imprimirá hasta que se hayan ingresado todos los datos, indicando donde ocurrió el error al momento de registrar, pero mostrando unicamente el primer error encontrado.

# 4. Diagramas UML

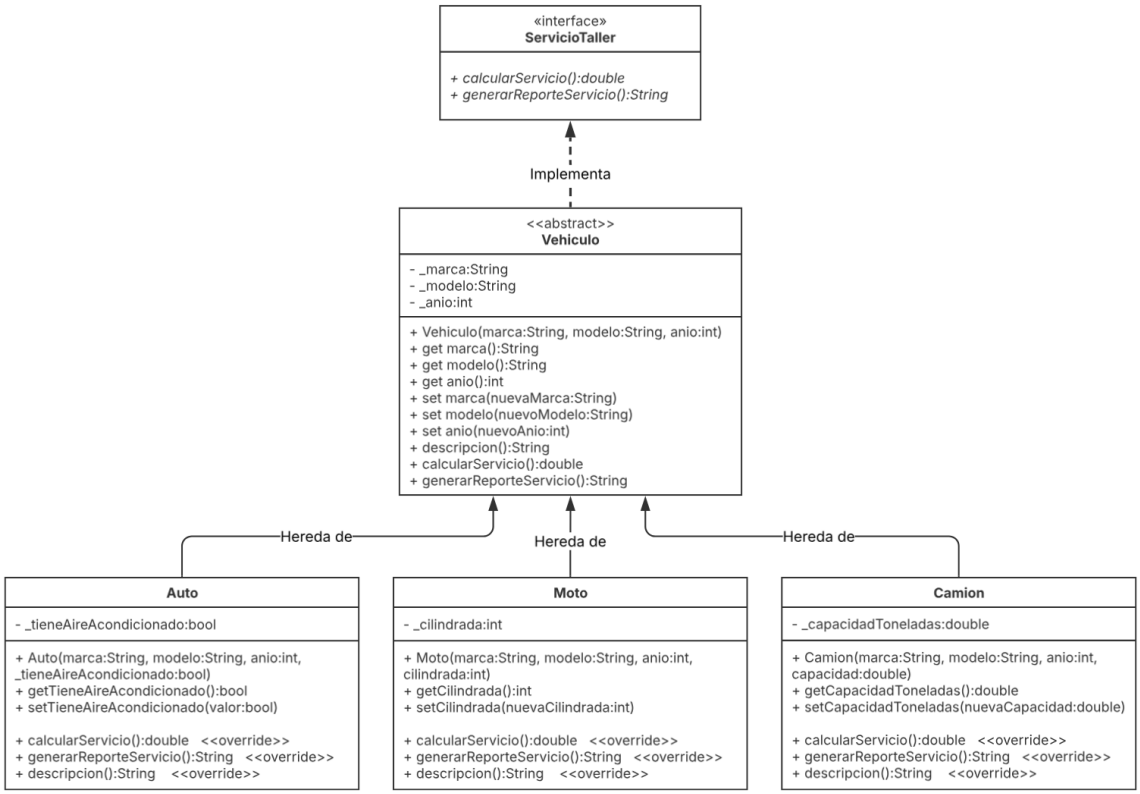



Diagrama UML de Clases

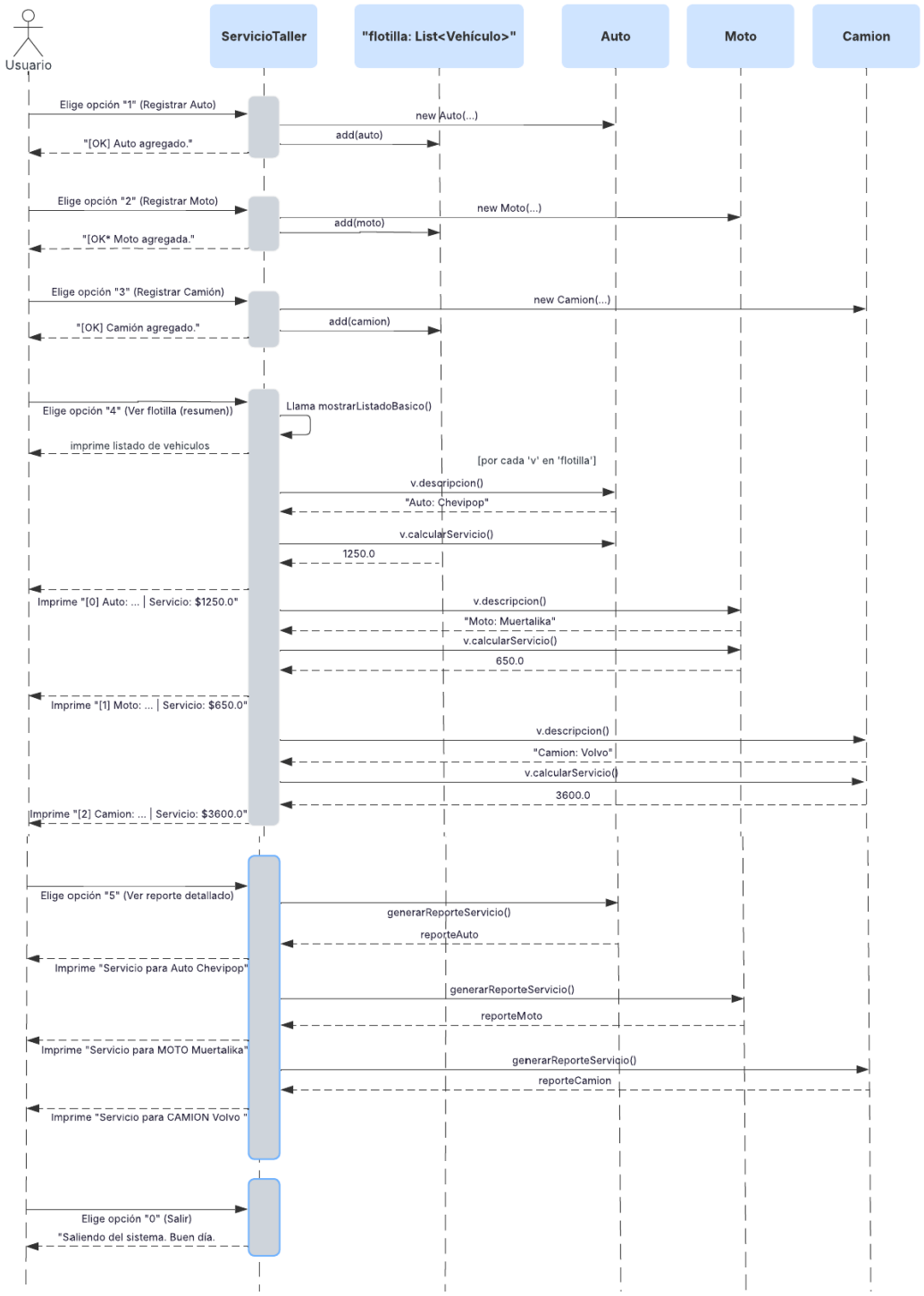



Diagrama UML de Secuencia

# 5. Resultados

```
PS C:\Users\PC\OneDrive\Desktop\VisualCode\Dart> dart run main.dart
==================================
   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: Renault
Modelo: Kwid
Año: 2025
¿Tiene aire acondicionado? (s/n): s

[OK] Auto agregado.

Presiona ENTER para continuar...
```

registro de un automóvil

```
==================================
   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: 2

== Registro de Moto ==
Marca: Yamaha
Modelo: Tracer 9 GT
Año: 2025
Cilindrara (cc): 125

[OK] Moto agregada.

Presiona ENTER para continuar...
```

registro de una motocicleta

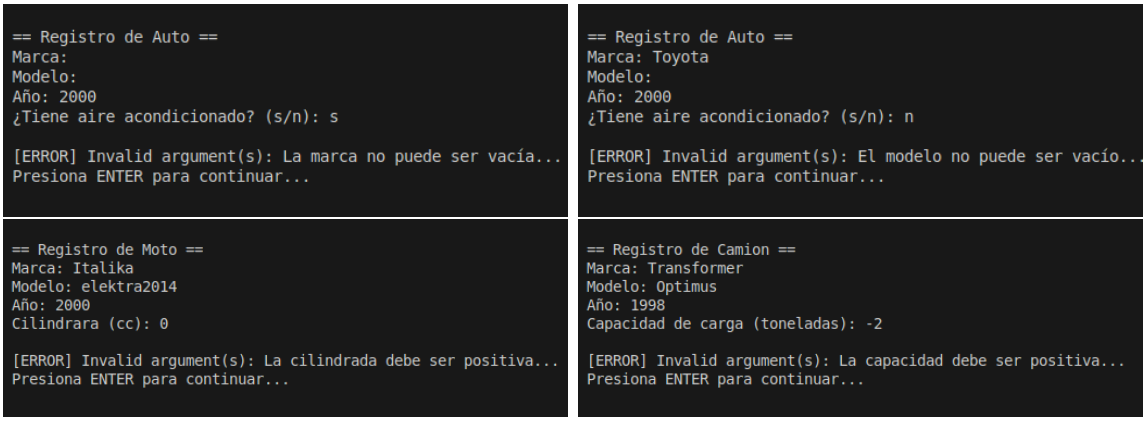
```
==================================
   SISTEMA PARA TALLER MECÁNICO
==================================
1) Registrar Auto
2) Registrar Moto
3) Registrar Camión
4) Ver flotilla (resumen)
5) Ver reportes detallados
0) Salir
Elige una opción: 3

== Registro de Camion ==
Marca: Caterpillar
Modelo: 797F
Año: 2020
Capacidad de carga (toneladas): 400

[OK] Camión agregado.

Presiona ENTER para continuar...
```

Registro de un camión

```
==================================
   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: 4

=== Flotilla registrada ===

[0] Auto: Renault Kwid (2025) - A/C: sí | Servicio: $1250.00
[1] Moto: Yamaha Tracer 9 GT (2025) - 125cc | Servicio: $450.00
[2] Camión: Caterpillar 797F (2020) - Capacidad: 400.0 toneladas | Servicio: $3600.00

Presiona ENTER para continuar...
```

Vista de todos los vehículos registrados


Reportes detallados para cada vehículo




Muestras de algunos errores posibles

# 6. Conclusiones

En ésta práctica logramos adquirir una correcta comprensión y gestión de excepciones y errores en Dart, entendiendo cómo éstas herramientas contribuyen a mantener un flujo de ejecución estable y controlado incluso ante situaciones inesperadas. A través del uso de bloques **try - catch** y la creación de excepciones personalizadas, se

observó cómo es posible anticipar fallos y manejar condiciones especiales. Además, se puede destacar que un buen manejo de errores mejora la legibilidad y la estabilidad del código o aplicación, haciéndolas más profesionales. Por lo que en esta práctica se subrayó la relevancia de diseñar clases y métodos que consideren explícitamente los posibles escenarios de error para lograr sistemas más organizados, seguros y eficientes.

# Referencias

[1] Radosław H. *Flutter vs Dart: Diferencias clave y casos de uso ideales explicados.* miquido. 2025-05-14. URL: https://www.miquido.com/blog/flutter-vs-dart/ (visitado 15-11-2025).

[2] *¿Qué es Flutter?* AWS. 2024. URL: https://aws.amazon.com/es/what-is/flutter/ (visitado 15-11-2025).

[3] J. A. Lozano. *Excepciones y Errores. Unidades de Apoyo para el Aprendizaje.* CUAED/Facultad de Ingeniería-UNAM. 2020. URL: https://repositorio-uapa.cuaed.unam.mx/repositorio/moodle/pluginfile.php/3085/mod_resource/content/1/UAPA-Excepciones-Errores/index.html (visitado 16-11-2025).

[4] *¿Qué es el lenguaje unificado de modelado (UML).* Lucidchart. 2025-07-18. URL: https://www.lucidchart.com/pages/es/que-es-el-lenguaje-unificado-de-modelado-uml (visitado 16-11-2025).