UNCUYO
UNIVERSIDAD NACIONAL DE CUYO

FACULTAD DE INGENIERÍA

Cátedra: Programación orientada a objetos

Trabajo Práctico N°2: Desarrollo orientado a objetos

Carrera: Ingeniería Mecatrónica

Camila Naser
Legajo:12318
Julieta Rossi
Legajo: 12467
Año 2025

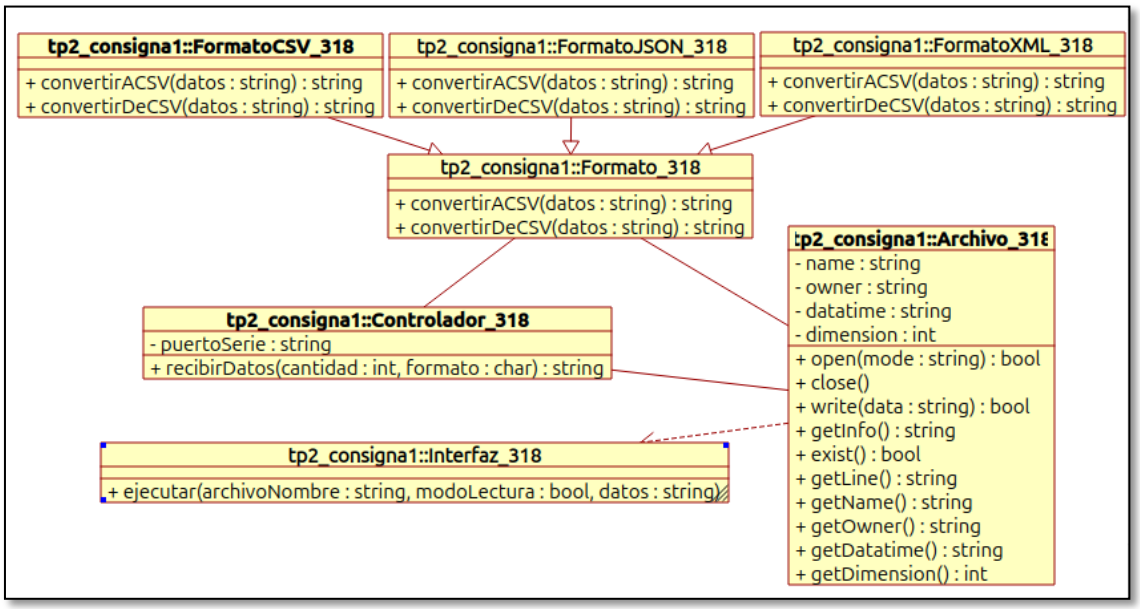
Objetivo general:

Implementar programas orientados a objetos de complejidad básica/media en lenguaje C++.

Actividad 1

Motivación: Esta actividad tiene la finalidad de definir una clase (al menos) que permita resolver un requerimiento sencillo e implementarla en lenguaje C++.

a) Esquema general de la solución:



b) Interfaces de usuario:

Se trabajó en la terminal, el programa esta planteado para recibir comandos de línea al estilo Linux.

Resultado del pedido de datos al Arduino en formato JSON:

```
camila@camila-VMware-Virtual-Platform: ~/uml-generated-code/tp2-desc...
_consigna1$ lsof /dev/ttyUSB0
camila@camila-VMware-Virtual-Platform:~/uml-generated-code/tp2-descomprimido/tp2
_consigna1$ ./programa -w test_json_final -n 1 -t j -p /dev/ttyUSB0 -f j
=== HERRAMIENTA DE DATOS ARDUINO ===
Modo: ESCRITURA
Archivo: test_json_final
========================================
Iniciando adquisición de datos desde Arduino...
Puerto: /dev/ttyUSB0
Cantidad de lecturas: 1
Formato de recepción: j
Limpiando buffer inicial...
Solicitando dato 1/1 (formato: j)
JSON recibido: {  "dispositivo_id": 2,  "porcentaje_valvula": 29,...
Total datos recibidos: 1
Datos recibidos: 1 líneas
Archivo guardado exitosamente: ./datos/test_json_final.csv
```

```
=== INFORMACIÓN DEL ARCHIVO ===
Propiedad           Valor
--------------------------------------------------
Nombre:             ./datos/test_json_final.csv
Propietario:        Rossi-Naser
Fecha mod.:         Fri Sep 26 14:26:30 2025
Tamaño:            96 bytes
==================================================

=== CONTENIDO DEL ARCHIVO ===
Formato de salida: JSON
--------------------------------------------------
[
  "{  "dispositivo_id": 2,  "porcentaje_valvula": 29,  "estado_nivel": "medio",
 "caudal": 72.23,}",
]


Proceso completado exitosamente.
```

Pedido de datos en formato XML:

```
camila@camila-VMware-Virtual-Platform:~/uml-generated-code/tp2-descomprimido/tp2
_consigna1$ ./programa -w test_xml_final -n 1 -t x -p /dev/ttyUSB0 -f x
=== HERRAMIENTA DE DATOS ARDUINO ===
Modo: ESCRITURA
Archivo: test_xml_final
========================================
Iniciando adquisición de datos desde Arduino...
Puerto: /dev/ttyUSB0
Cantidad de lecturas: 1
Formato de recepción: x
Limpiando buffer inicial...
Solicitando dato 1/1 (formato: x)
XML recibido: <?xml version = "1.0" ?> <registro>  <dispositivo_...
Total datos recibidos: 1
Datos recibidos: 1 líneas
Archivo guardado exitosamente: ./datos/test_xml_final.csv
```

```
=== INFORMACIÓN DEL ARCHIVO ===
Propiedad           Valor
-------------------------------------------------
Nombre:             ./datos/test_xml_final.csv
Propietario:        Rossi-Naser
Fecha mod.:         Fri Sep 26 14:39:12 2025
Tamaño:            192 bytes
=================================================

=== CONTENIDO DEL ARCHIVO ===
Formato de salida: XML
-------------------------------------------------
<datos>
  <linea><?xml version = "1.0" ?> <registro>  <dispositivo_id>2</dispositivo_id>
   <porcentaje_valvula>82</porcentaje_valvula>   <estado_nivel>medio</estado_niv
el>   <caudal>42.22</caudal> </registro></linea>
</datos>


Proceso completado exitosamente.
```

Y por último, si se solicita en formato CSV:

```
=== INFORMACIÓN DEL ARCHIVO ===
Propiedad           Valor
-------------------------------------------------
Nombre:             ./datos/test_csv_multi2.csv
Propietario:        usuario
Fecha mod.:         Thu Sep 25 11:14:25 2025
Tamaño:            17 bytes
=================================================

=== CONTENIDO DEL ARCHIVO ===
Formato de salida: CSV
-------------------------------------------------
2,92,medio,92.74


Proceso completado exitosamente.
```

Esto es lo que observa el usuario por consola, pero los archivos con la información se guardan en la carpeta “datos”, con una extensión fija (en este caso .csv)

c) Recursos adicionales:
Para esta actividad se utilizó el Firmware de Arduino, proporcionado por la cátedra. El mismo es capaz de devolver datos en formato CSV, JSON o XML enviándole los parámetros “c”, “j”, “x” respectivamente.

d) Uso:

El aplicativo desarrollado (“programa”) permite almacenar y recuperar información en distintos formatos de texto plano (CSV, JSON, XML) junto con sus metadatos (nombre, fecha, propietario y cantidad de líneas).

Consideraciones de uso por parte del usuario final

1.Compilación previa

El programa debe ser compilado con un compilador compatible con C++17 o superior. Una vez generado el ejecutable, puede utilizarse directamente desde la consola. Como se cuenta con un Makefile simplemente ejecutando “make all” se consigue la compilación de todos los archivos.

2.Ejecución en consola

La interacción se realiza a través de línea de comando estilo Linux, con las opciones:

-r: modo lectura del archivo existente

-w: modo escritura desde Arduino

-f: formato de salida (donde c=CSV, j=JSON, x=XML)

-n: cantidad de lecturas

-p: puerto serie

-t: tipo de recepción desde Arduino (c,j,x)

Así, el formato a utilizar para su ejecución, en caso de querer por ejemplo una lectura de datos en formato JSON, es:

./programa -w [Nombre del archivo] -n 1 -t j -p /dev/ttyUSB0 -f j

Consideraciones respecto a la comunicación serie:

Es importante tener en cuenta en cuál puerto se conecto efectivamente nuestro Arduino. Normalmente aparece como:

• /dev/ttyUSB0 si usa un chip conversor USB a serie como CH340 o FTDI.

• /dev/ttyACM si usa el chip nativo de la familia Atmel (como el Arduino Uno).

Con el comando ls -la /dev/tty* podemos observar en cuál puerto se encuentra

Tambien es necesario verificar los permisos del puerto, esto se puede hacer con el comando ls -l /dev/ttyUSB0. Generalmente se necesita que el usuario pertenezca al grupo “dialout” , con el comando groups podemos verificar a qué grupo pertenece nuestro usuario.

Por último tambien es bueno verificar que no haya ningun proceso ejecutándose en el puerto, que impida la comunicación, esto lo podemos saber con lsof /dev/ttyUSB0. Si hay algun proceso abierto, exterminarlo con sudo kill [numero de proceso].

A continuacion algunos de los pasos mencionados anteriormente, realizados para asegurar una correcta comunicación:

Consideraciones para la reutilización del código

1. Estructura modular
   La clase Archivo_318 encapsula toda la lógica de manejo de archivos y metadatos. Esto facilita su reutilización en otros proyectos simplemente incluyendo los archivos Archivo_318.hpp y Archivo_318.cpp.
2. Portabilidad
   El código utiliza únicamente librerías estándar de C++17 (<string>, <vector>, <fstream>, <filesystem>), por lo cual puede compilarse en otros sistemas operativos, cambiando el compilador y el manejo del fin de entrada.
3. Extensibilidad: El diseño permite agregar fácilmente:
   - Validaciones de formato más estrictas (ej. validar XML contra un esquema).
   - Conversiones automáticas entre formatos.

e) Conclusiones:

En este trabajo es de especial importancia el correcto manejo del puerto ya que si no se toman todas las precauciones para confirmar que el puerto se encuentra limpio y disponible se producen errores en la lectura de los datos (fácilmente identificables, de cualquier manera, gracias al manejo de excepciones)

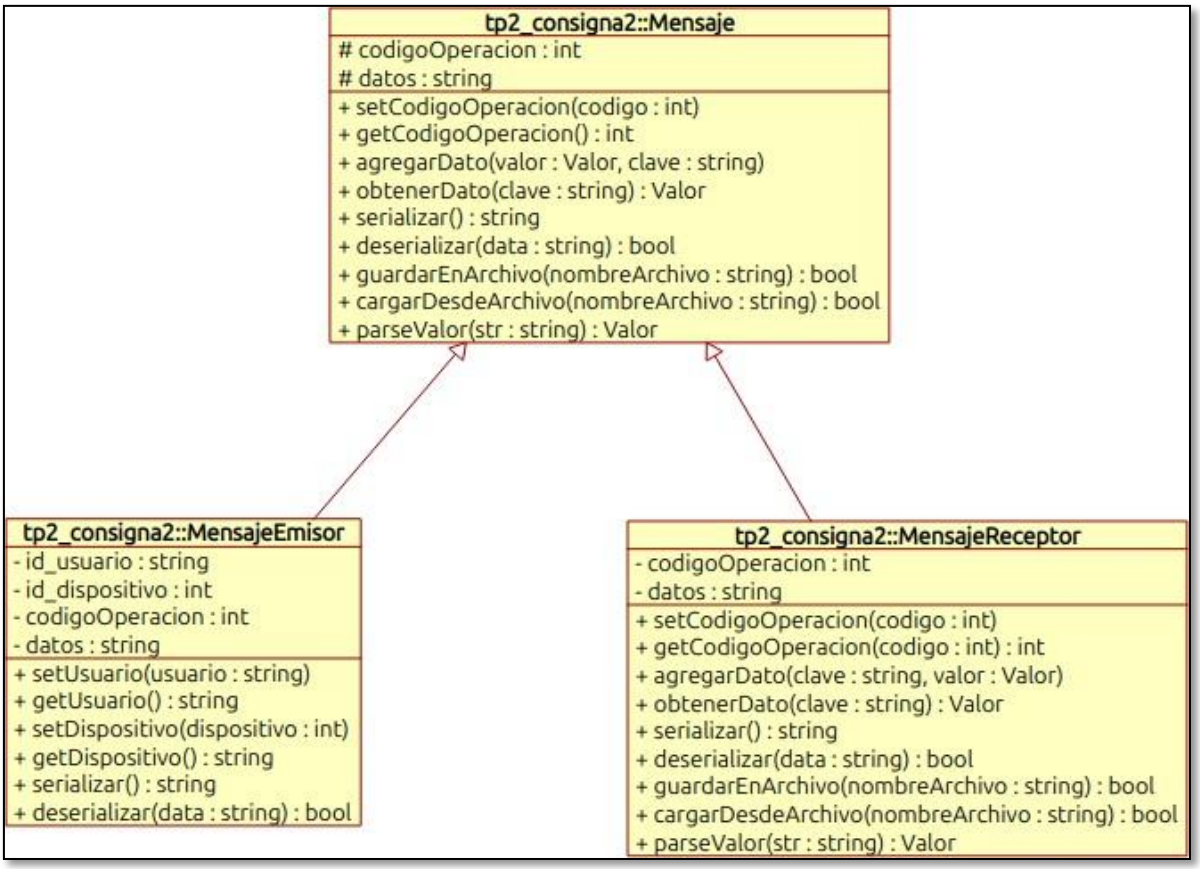
f) Referencias consultadas:

C++ Manejo de archivos: cómo abrir, escribir, leer y cerrar archivos en C++

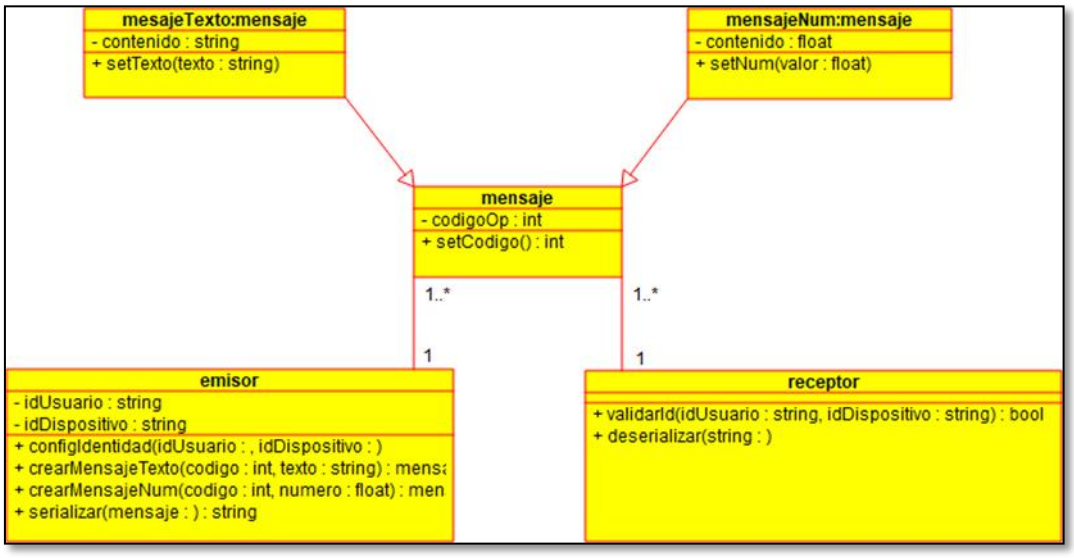
Actividad 2

Motivación:
Esta actividad tiene la finalidad de definir varias clases con relaciones entre ellas, de modo de resolver un requerimiento sencillo e implementarla en lenguaje C++, considerando aspectos de comunicación entre objetos.

Esquema general de la solución:



Solución alternativa:



Completamos el primer modelo, la interfaz de usuario para una ejecución completa es la siguiente:

```
==============================================
== SIMULACION: FLUJO EMISOR -> RECEPTOR ==
==============================================

--- 1. MensajeEmisor (Envia) ---
EMISOR: Mensaje serializado y guardado en 'transmision_simulada.txt'.
   Serializado completo:
   Usuario:Rossi-Naser;Dispositivo:467-318;Operacion:201;Status=Listo;Temperatur
a=25.6;

--- 2. MensajeReceptor (Recibe) ---
RECEPTOR: Archivo cargado exitosamente.
   -> Codigo de Operacion: 201
   -> Datos recibidos:
  Dato 'Temperatura': 25.6 (double)
  Dato 'Status': "Listo" (string)
```

El aplicativo simula el flujo emisor/receptor escribiendo y leyendo archivos de texto. El mensaje se guarda en el archivo “transmision_simulada.txt”, el formato es texto con pares clave valor separados por punto y coma.

Consideraciones de uso por parte del usuario final

1.Compilación previa

El programa debe ser compilado con un compilador compatible con C++17 o superior con el siguiente comando por consola

```
>g++ -std=c++17 -O2 main.cpp Mensaje.cpp MensajeEmisor.cpp MensajeReceptor.cpp -o programa
```

Una vez compilado se ejecuta:

```
>./programa
```

Al ejecutarse el MensajeEmisor construye el mensaje y le agrega Id de usuario y dispositivo, luego MensajeReceptor recibe el mensaje desde el archivo y reconstruye codigoOperacion.

Consideraciones para reutilizar código

Formato de intercambio: texto plano Operacion:###;clave=valor; Este contrato es estable para Emisor/Receptor y facilita depurar con cualquier editor de texto

Tipos de datos: al deserializar, el sistema infiera automáticamente int, double o string (en ese orden) según el valor textual

Vale aclarar que se uso un Alias para tipos de datos soportados mediante la sentencia:

using Valor = std::variant<int, double, std::string>;

Con esto se crea el alias Valor para el tipo std::variant<int, double, std::string> y así se crea un nuevo nombre mas simple para un dato complejo. Esto lo utilizamos para poder manejar datos que pueden ser variables en su tipo.

std::variant es una clase de la biblioteca estándar de C++ (introducida en C++17) que permite que una variable pueda contener uno de varios tipos de datos posibles en un momento dado, pero solo uno a la vez. Es una forma de tener un tipo de dato que puede ser polimórfico en valor.

En este caso: La variable Valor puede contener: un entero (int), o un número de punto flotante (double), o una cadena de texto (std::string).

Extensibilidad: para agregar campos “de cabecera” (metadatos), sobrescribí serializar() / deserializar() en una subclase (como hace MensajeEmisor con Usuario y Dispositivo) y delegá el resto a Mensaje.

Justificación del modelo adoptado:

Para está implementación especifica elegimos aquella que nos resultaba más simple tanto en programación y como para prueba.

Aun asi el modelo adoptado tiene capacidad de crecimiento porque podemos agregar atributos o nuevos tipos de mensajes

Conclusión

Una dificultad encontrada fue el manejo de distintos tipos de datos leídos desde texto plano. Una posible extensión es la incorporación de otras formas de persistencia (JSON, CSV) similar a lo ya realizado en el requerimiento 1, también podemos agregar más metadatos que nos ayuden en la trazabilidad de mensajes.

Una valoración personal de este trabajo es que nos ayudó a comprender que hay más de una manera válida de resolver un problema, la manera elegida depende de la implementación y recursos.

Referencias consultadas:

https://www-geeksforgeeks-org.translate.goog/cpp/serialize-and-deserialize-an-object-in-cpp/?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=tc