**Programación Orientada a Objetos**

**Universidad Nacional de Cuyo - Facultad de Ingeniería**

**TP2 – Actividad 1**

# TRABAJO PRACTICO Nº 2 – Desarrollo Orientado a Objetos

F. Barrios Retta

Septiembre 2025

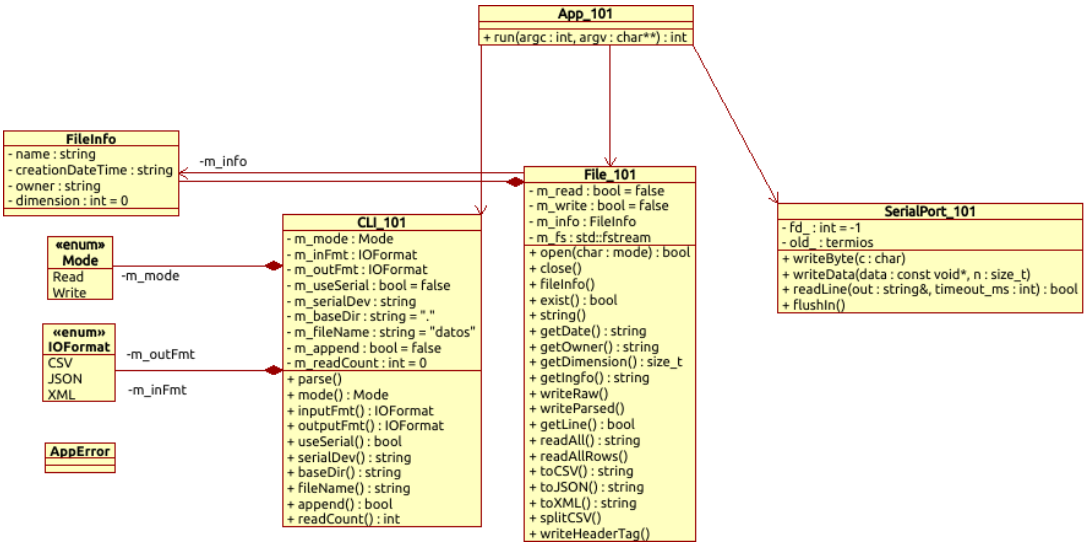
---

*Incluye código probado en hardware.*

## 1 Consideraciones Generales

## Contents

# 2 Esquema general de la solución




# 3 Interfaces de usuario

Imagenes del código que anda

# 4 Recursos adicionales

No se hace uso de ningun componente de software no estandar del lenguaje ni de la plataforma para codificar el programa. A continuación, se listan las librerias estandar de C++ y de Linux que se utilizan en los diferentes archivos `.h` y `.cpp`.

```
#include <string>
#include <vector>

#include <fstream>
#include <stdexcept>
#include <termios.h>
#include <chrono>
#include <ctime>
#include <sstream>
#include <iomanip>
#include <iostream>
#include <cstdlib>
#include <sys/stat.h>
#include <unistd.h>
#include <pwd.h>
#include <filesystem>
#include <system_error>
#include <algorithm>
```

## 4.1 ¿Para qué sirven los `#include`?

- `<string>`: `std::string`, cadenas decentes, no arrays C.

- `<vector>`: `std::vector<T>`, arrays dinámicos.
- `<fstream>`: `std::ifstream/ofstream/fstream` para leer/escribir archivos.
- `<stdexcept>`: excepciones estándar (`std::runtime_error`, `std::invalid_argument…`).
- `<termios.h>`: control POSIX de puertos/terminales. Configurar serial: baudrate, paridad, flags.
- `<chrono>`: tiempo moderno C++: steady_clock, milliseconds, sleep_for.
- `<ctime>`: tiempo estilo C: `time_t`, `std::localtime`, `std::strftime-like`.
- `<sstream>`: `std::stringstream` para parsear/armar strings como si fueran streams.
- `<iomanip>`: formateo de streams: `std::setw`, `std::setprecision`, `std::put_time`.
- `<iostream>`: `std::cout`, `std::cin`, `std::cerr`. El trío inevitable.
- `<cstdlib>`: utilidades varias: `std::getenv`, `std::system`, `std::strtol`, `std::rand`.
- `<sys/stat.h>`: stat, permisos y mkdir con modos; funciones del Sistema Operativo.
- `<unistd.h>`: POSIX: read, write, close, usleep, access, isatty.
- `<pwd.h>`: info de usuario: getpwuid para home, etc.
- `<filesystem>`: C++17 paths y archivos: `std::filesystem::path`, `exists`, `create_directories`.
- `<system_error>`: `std::error_code` y `std::system_error` para reportes de errores del Sistema Operativo.
- `<algorithm>`: `std::sort`, `std::find`, `std::transform`, `std::accumulate…` herramientas útiles.

# 5 Manual de instrucciones de la aplicación

## 5.1 Diseño de la solución

Se adoptó una arquitectura simple orientada a objetos, respetando el diagrama de clases provisto en la consigna. El diagrama actualizado se ilustra a continuación:

- **App** coordina la ejecución general y delega en `CLI` la interpretación de argumentos.
- **CLI** encapsula la lectura de los parámetros y expone *getters* tipados (`Mode`, `IOFormat`).
- **File** maneja la persistencia en CSV, mantiene los metadatos (`FileInfo`) y ofrece conversores a JSON/XML.
- **SerialPort** abstrae la comunicación por puerto serie usando `termios` para trabajar en modo crudo a 115200 bauds.
- **Types** concentra tipos comunes (`AppError`, `Mode`, `IOFormat`) y las funciones utilitarias `parseLine` y `tryParseHeaderTag`.

### 5.1.1 Flujo en modo escritura

1. El operador define el formato esperado mediante `-i`.
2. `App` abre/crea el CSV en el directorio indicado y realiza, si es necesario, la cabecera `#h:`.
3. Para cada muestra se envía el byte de *handshake* (`c`, `j` o `x`) y se espera la respuesta del µC.

4. `parseLine` detecta el formato real, normaliza la estructura y devuelve un vector de campos.
5. `File::writeParsed` escapa los valores y los persiste siempre en CSV.
6. Se contabilizan las lecturas correctas y, al finalizar, se informa la tabla de metadatos.

#### 5.1.2 Flujo en modo lectura

1. `CLI` determina el formato deseado (`-o`).
2. `File` abre el CSV, actualiza los metadatos y expone `getInfoTable()`.
3. En función del formato elegido se reutiliza el CSV original (`toCSV()`), se arma un arreglo JSON (`toJSON()`) o un documento XML (`toXML()`).
4. Cuando no existen encabezados explícitos se generan identificadores genéricos (`c1`, `c2`, …).

### 5.2 Serialización y robustez

- Se configuran 115200 bauds, modo 8N1 sin control de flujo hardware.
- El `timeout` de lectura se fijó en 2000 ms para compensar jitter en la transmisión.
- Las líneas incompletas o vacías se descartan sin afectar el contador de lecturas exitosas.
- `File::countRowsOnDisk()` y `readAllRows()` ignoran comentarios y filas vacías para mantener una dimensión realista del dataset.

### 5.3 Pruebas realizadas

- Compilación con `make` (usar desde `Codigos/`).
- Captura de 20 muestras desde el µC con `./app -m w -i c -n 20 -s /dev/ttyACM0 -d logs -f sensor`.
- Visualización en los tres formatos disponibles: `./app -m r -o c`, `./app -m r -o j`, `./app -m r -o x`.
- Verificación manual de la consistencia del CSV resultante (`logs/sensor.csv`).

## 6 Conclusiones

La herramienta cumple con la consigna del Trabajo Práctico Nº 2 al brindar una solución modular, extensible y alineada con los principios de diseño orientado a objetos. Se lograron separar responsabilidades, normalizar formatos heterogéneos y facilitar la interacción con el µC mediante un protocolo sencillo de handshakes.

## 7 Referencias consultadas

## 8 Anexo – Comandos útiles

```
# Compilación
make

# Captura desde el µC (CSV)
./app -m w -i c -n 10 -s /dev/ttyACM0 -d logs -f sensor

# Lectura en distintos formatos
./app -m r -o c -d logs -f sensor   # CSV
./app -m r -o j -d logs -f sensor   # JSON
```

```
./app -m r -o x -d logs -f sensor   # XML

# Generar PDF del informe
jupyter nbconvert --to pdf "Trabajo Práctico 2 - Actividad N°2.ipynb"
```