



UNIVERSITAT  
POLITÈCNICA  
DE VALÈNCIA



GENERACIÓN  
ESPONTÁNEA  
UPV DESIGN FACTORY



Generació Espontànea

TARS

Universitat Politècnica de València

21 de Septiembre de 2025

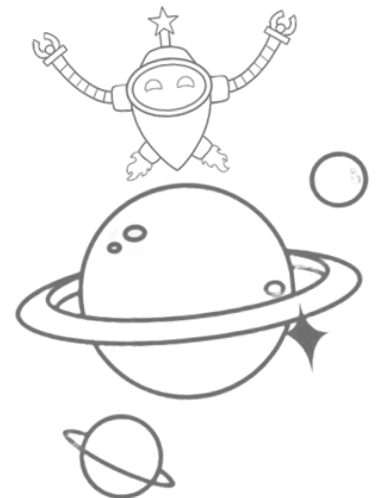
# Buenas Prácticas de Programación

DEPARTAMENTO DE SOFTWARE | TARS

TARS Robotics

**Autor:**

Jose Luis Galán Avilés | [jlgalavi@etsinf.upv.es](mailto:jlgalavi@etsinf.upv.es)



# TABLA DE CONTENIDOS

---

<b>TABLA DE CONTENIDOS.....</b>	<b>1</b>
<b>Introducción.....</b>	<b>2</b>
¿Por qué lo hacemos?.....	2
¿A quién va dirigido?.....	2
<b>Principios de diseño y estilo comunes.....</b>	<b>2</b>
Nombres de variables, funciones y clases.....	3
Variables.....	3
Funciones y métodos.....	3
Clases.....	4
Constantes.....	4
Booleanos.....	4
Comentarios y documentación.....	5
<b>Principios de diseño y estilo comunes.....</b>	<b>6</b>
Branching para features, bugs y hotfixes.....	11
Mensajes de commit.....	12
Política de Pull Requests y revisiones.....	12
📄 sensor_distancia.h.....	13
🔧 Compilación.....	14
Paquetes.....	15
Nodos.....	15
Tópicos.....	15
Parámetros.....	15
Servicios y acciones.....	16
QoS y comunicación.....	16
Frames y TF.....	16
Archivos de lanzamiento.....	16

# Introducción

Este documento recoge las **buenas prácticas de programación** que vamos a seguir en el equipo de **software de TARS Robotics** durante el proyecto **European Rover Challenge (ERC)**.

La idea es que, aunque seamos un grupo de estudiantes con niveles diferentes de experiencia, **todo el código se entienda igual y sea fácil de mantener**.

👉 Piensa en este documento como en un “manual de estilo”: igual que en un idioma seguimos reglas gramaticales para entendernos, aquí seguimos unas pautas para que cualquier persona del equipo pueda leer, modificar o ampliar el código sin problemas.

## ¿Por qué lo hacemos?

- Porque un código **ordenado y homogéneo** evita errores y malentendidos.
- Porque así, si alguien nuevo entra en el equipo, puede entender rápidamente lo que ya está hecho.
- Porque trabajar con **C++ y Python** (y en parte con **ROS 2**) requiere coordinación: cada uno programa en su ordenador, pero al final todo se junta en el rover.

## ¿A quién va dirigido?

A todos los miembros del **departamento de software**, pero también a los de otros equipos (mecánica, electrónica, teléco) que tengan que escribir scripts, módulos o pruebas.

En resumen: este documento es nuestra **guía común** para que el desarrollo sea claro, organizado y escalable, independientemente de quién lo escriba o en qué lenguaje lo haga.

# Principios de diseño y estilo comunes

Cuando programamos en equipo no basta con que el código **funcione**, también debe ser **legible, ordenado y coherente**. Si cada persona escribe a su manera, el resultado es un caos difícil de mantener. Por eso, adoptamos unas **reglas comunes** que nos permiten:

- Que cualquier miembro pueda leer y entender el trabajo de otro.
- Que el código sea más fácil de depurar y ampliar.
- Que se eviten errores al integrar diferentes módulos.

A continuación se presentan las normas básicas de estilo y diseño que todos debemos seguir.

## Nombres de variables, funciones y clases

Usaremos **nombres claros y descriptivos**, evitando abreviaciones innecesarias como **tmp**, **var**, **coso**, etc. El nombre debe dejar claro **qué representa** o **qué hace**.

### Variables

- Se escriben en **snake\_case** (minúsculas con guiones bajos).
- Deben ser **descriptivas**, evitando abreviaturas como **tmp**, **var**, **coso**.

```
# Python
temperatura_media = 23.5
contador_errores = 0
```

```
// C++
int velocidad_maxima = 10;
float tiempo_respuesta = 0.25;
```

### Funciones y métodos

- Se escriben en **snake\_case**.
- Deben usar **verbos** o frases que indiquen qué hacen.
- Si devuelve un valor, su nombre debe reflejar qué calculan/devuelven.

```
# Python
def calcular_distancia(x1, y1, x2, y2):
    """
    Calcula la distancia euclidiana entre dos puntos en 2D.
    """
    return ((x2 - x1)**2 + (y2 - y1)**2) ** 0.5
```

```
// C++
float calcular_distancia(float x1, float y1, float x2, float y2) {
    return std::sqrt(std::pow(x2 - x1, 2) + std::pow(y2 - y1, 2));
}
```

## Clases

- Se escriben en **PascalCase** (cada palabra empieza en mayúscula, sin guiones bajos).
- El nombre debe indicar claramente la **entidad** que representa.

```
class ControlTemperatura:  
    def __init__(self, limite_maximo):  
        self.limite_maximo = limite_maximo
```

```
class SensorDistancia {  
public:  
    SensorDistancia(float rango) : rango_maximo(rango) {}  
private:  
    float rango_maximo;  
};
```

## Constantes

- Se escriben en **MAYÚSCULAS\_CON\_GUIONES**.
- Se utilizan para valores fijos que no deben cambiar durante la ejecución.

```
MAX_VELOCIDAD = 120  
TIEMPO_ESPERA = 0.5
```

```
const int MAX_VELOCIDAD = 120;  
const float TIEMPO_ESPERA = 0.5f;
```

## Booleanos

- Deben usar prefijos que indiquen claramente su propósito:
  - **is\_...** → estados.
  - **has\_...** → posesión.
  - **can\_...** → capacidad.

```
is_active = True  
has_permission = False  
can_execute = True
```

```
bool is_active = true;
bool has_permission = false;
bool can_execute = true;
```

## Comentarios y documentación

Los comentarios deben explicar **qué hace el código y por qué**, nunca repetir lo obvio.

- Cada función debe incluir una breve explicación de su propósito.
- Los comentarios deben ser breves y claros, pero suficientes para entender decisiones importantes.
- En Python se usan **docstrings**.
- En C++ se recomienda el estilo **Doxygen** (`///`).

```
def calcular_media(valores):
    """
    Calcula la media aritmética de una lista de valores.
    Se utiliza en el módulo de sensores para estimar lecturas.
    """
    return sum(valores) / len(valores)
```

```
/// @brief Calcula la media aritmética de un vector de enteros.
/// @param valores vector con los números a promediar.
/// @return valor medio como float.
float calcular_media(const std::vector<int>& valores) {
    int suma = 0;
    for (int v : valores) suma += v;
    return static_cast<float>(suma) / valores.size();
}
```

## Modularidad y organización

El código debe dividirse en **funciones y módulos pequeños** que sean fáciles de entender, probar y reutilizar.

- Cada función debe hacer **una sola cosa** (Principio de Responsabilidad Única).
- El proyecto debe organizarse en **carpetas** según el propósito (ej: `/navegacion`, `/vision`, `/control`).
- En ROS 2, cada módulo independiente debe ser un **paquete** separado.

- Se recomienda tener archivos de configuración centralizados (`config.py`, `constantes.h`).

```
def leer_sensor():  
    # Devuelve el valor leído de un sensor  
    return valor  
  
def procesar_dato(valor):  
    # Procesa el dato para eliminar ruido  
    return resultado  
  
def actuar(resultado):  
    # Envía una orden al motor  
    pass
```

## Espaciado y estilo de bloques

El espaciado y la indentación deben ser **consistentes** para mejorar la legibilidad.

✗ Incorrecto:

```
if(x==10){doSomething();}
```

✓ Correcto:

```
if (x == 10) {  
    doSomething();  
}
```

En Python, la indentación obligatoria será siempre de **4 espacios**.

## Principios de diseño y estilo comunes

Aunque compartimos unas normas comunes de estilo (nombres de variables, funciones, clases, etc.), cada lenguaje tiene características propias que debemos respetar. A continuación se detallan las guías específicas para **C++** y **Python**, que complementan lo visto en el punto anterior.

## C++

El código en C++ debe escribirse siguiendo los principios de C++ moderno (C++17 o superior). Esto significa evitar prácticas antiguas y aprovechar las herramientas del lenguaje actual.

Usar **smart pointers** (`std::unique_ptr`, `std::shared_ptr`) en lugar de punteros crudos siempre que sea posible.

```
#include <memory>

std::unique_ptr<int> numero = std::make_unique<int>(42);
```

**Const-correctness:** declarar variables, parámetros y métodos como `const` siempre que no vayan a modificarse.

```
float calcular_area(const float radio) {
    return 3.14159f * radio * radio;
}
```

Usar referencias en lugar de punteros cuando no haya necesidad de punteros.

```
void imprimir_valor(const std::string& texto) {
    std::cout << texto << std::endl;
}
```

**Estructura del proyecto en archivos .h y .cpp** (como vimos en el punto 2.3).

- `.h` → declaración de clases, funciones y constantes.
- `.cpp` → implementación.

**Control de errores** → usar `try / catch` solo cuando sea necesario y con excepciones específicas.

```
try {
    // Código que puede fallar
} catch (const std::runtime_error& e) {
    std::cerr << "Error: " << e.what() << std::endl;
}
```

**Compilación** con `-Wall` `-Wextra` siempre activados para detectar advertencias.



## Python

En Python seguiremos las recomendaciones de **PEP 8** (estilo) y **PEP 257** (docstrings).

Usar **type hints** para mejorar la legibilidad y permitir análisis estático con **mypy**.

```
def calcular_area(radio: float) -> float:
    return 3.14159 * radio * radio
```

Usar **docstrings** en formato **Google** para describir funciones y clases.

```
def calcular_distancia(x1: float, y1: float, x2: float, y2: float) ->
float:
    """
    Calcula la distancia euclidiana entre dos puntos en 2D.

    Args:
        x1: Coordenada x del primer punto.
        y1: Coordenada y del primer punto.
        x2: Coordenada x del segundo punto.
        y2: Coordenada y del segundo punto.

    Returns:
        Distancia en float.
    """
    return ((x2 - x1)**2 + (y2 - y1)**2) ** 0.5
```

Gestión de errores con **try / except**.

```
try:
    valor = int("abc")
except ValueError as e:
    print(f"Error: {e}")
```

**Importaciones ordenadas** en tres bloques separados:

- Librerías estándar.
- Librerías externas.
- Módulos internos del proyecto.

```
import os
import sys

import numpy as np

from rover.vision import procesador_imagen
```

**Evitar duplicación de código:** crear funciones auxiliares o módulos cuando algo se repite.

**Entornos virtuales:** siempre trabajar con un **venv** o **poetry** para gestionar dependencias.

## Estructura del repositorio

Un repositorio bien organizado permite que cualquier persona del equipo entienda rápidamente dónde está cada parte del proyecto y cómo debe trabajar. La estructura de carpetas debe ser **clara, homogénea y modular**, de manera que:

- Cada subequipo (mecánica, software, electrónica, teléco) sepa dónde aportar.
- Sea fácil encontrar código, configuraciones y documentación.
- Se facilite la integración en entornos como **ROS 2**.

## Estructura recomendada del proyecto

```
/ERC
|
|— docs/                # Documentación (manuales, diagramas,
informes)
|— config/              # Archivos de configuración globales
|— scripts/             # Scripts auxiliares (Python, bash)
|— src/                 # Código fuente principal
|   |— c++/              # Código en C++
|   |   |— control/      # Control de motores, cinemática
|   |   |— vision/       # Procesamiento de visión en C++
|   |   |— ...
|   |— python/           # Código en Python
|   |   |— vision/       # Procesamiento de imágenes
|   |   |— nav/          # Navegación y localización
|   |   |— ...
|— ros2_ws/             # Workspace de ROS 2
|   |— src/              # Paquetes de ROS 2
```

```

├── ┌── nav_pkg/      # Ejemplo: paquete de navegación
│   ├── vision_pkg/  # Ejemplo: paquete de visión
│   └── ...
├── launch/          # Archivos de lanzamiento .launch.py
├── tests/           # Pruebas unitarias y de integración
│   ├── c++/         # Tests en C++
│   └── python/       # Tests en Python
├── .gitignore        # Archivos a ignorar por Git
├── README.md         # Descripción general del proyecto
├── LICENSE           # Licencia del proyecto
└── requirements.txt  # Dependencias Python (si aplica)


```

## Convenciones para nombres de archivos y carpetas

Todo en **minúsculas** y en **snake\_case**.

Evitar nombres genéricos como **cosas**, **varios** o **prueba1**.

Los archivos **.cpp** y **.py** deben reflejar su contenido.

-  **control\_motores.cpp**, **procesar\_imagen.py**
-  **codigo.cpp**, **nuevo.py**

## Archivos base del repositorio

**README.md** → descripción del proyecto y flujo de trabajo (ya definido).

**LICENSE** → licencia del proyecto (académica/interna).

**.gitignore** → lista de archivos a ignorar (builds, venv, etc.).

**requirements.txt** → dependencias de Python.

**CMakeLists.txt** y **package.xml** → para proyectos ROS 2 en C++.

**setup.py** / **pyproject.toml** → para proyectos ROS 2 en Python.

**CODEOWNERS** → define quién debe revisar qué partes del código.

# Control de versiones y flujo de trabajo con Git

Git es la herramienta que usamos para **trabajar en equipo sin pisarnos el código**. Seguir un flujo común evita problemas y nos permite mantener un historial claro y ordenado.

## Ramas principales

Trabajaremos con dos ramas principales:

- **develop** → rama de trabajo.  
Aquí se suben las nuevas funcionalidades, pruebas y desarrollos.
- **main** → rama estable y protegida.  
Solo recibe cambios a través de **Pull Requests (PR)** desde **develop**, y requiere la aprobación del coordinador.

⚠ **Nunca se hace push directo a **main**.**

## Branching para features, bugs y hotfixes

Cada nueva tarea debe realizarse en una rama independiente, creada a partir de **develop**.

**Features** (nuevas funcionalidades):

```
git checkout develop
git pull
git checkout -b feature/navegacion_autonoma
```

**Bugs** (correcciones de errores):

```
git checkout develop
git pull
git checkout -b bugfix/correccion_sensor
```

**Hotfixes** (errores críticos en **main**):

```
git checkout main
git pull
git checkout -b hotfix/bug_critico
```

Una vez completada la tarea, la rama se sube a GitHub y se abre un **Pull Request (PR)** hacia `develop`.

## Mensajes de commit

Los commits deben ser **pequeños, frecuentes y descriptivos**.

Formato recomendado (estilo *Conventional Commits*):

- `feat: añade cálculo de trayectoria`
- `fix: corrige error en lectura de sensor`
- `docs: actualiza README con instrucciones de instalación`
- `refactor: reorganiza funciones de control de motores`
- `test: añade pruebas unitarias a módulo de visión`

Ejemplo correcto:

```
git commit -m "feat: implementa función de cálculo de distancia euclidiana"
```

✗ Ejemplo incorrecto:

```
git commit -m "cambio cosas"
```

## Política de Pull Requests y revisiones

Todos los cambios hacia `develop` o `main` se realizan mediante **Pull Requests (PRs)**.

Cada PR debe:

- Explicar qué cambios incluye.
- Referenciar la Issue asociada (si existe).
- Pasar las pruebas automáticas (cuando estén implementadas).

Los PR hacia `main` requieren aprobación del coordinador (CODEOWNER).

Todas las conversaciones abiertas en un PR deben resolverse antes de hacer merge.

# Anexo – Ejemplos

## Ejemplo de modularidad en C++

Imagina que tenemos un **sensor de distancia**. En lugar de meter todo en un único archivo, lo separamos en:

- `sensor_distancia.h` → declaración de la clase.
- `sensor_distancia.cpp` → implementación de los métodos.
- `main.cpp` → programa principal que utiliza la clase.

### `sensor_distancia.h`

```
#ifndef SENSOR_DISTANCIA_H
#define SENSOR_DISTANCIA_H

#include <iostream>

/// @brief Clase que representa un sensor de distancia ultrasónico.
class SensorDistancia {
public:
    /// @brief Constructor con el rango máximo del sensor.
    SensorDistancia(float rango_maximo);

    /// @brief Lee el valor del sensor (simulado en este ejemplo).
    /// @return distancia medida en cm.
    float leer();

private:
    float rango_maximo;
};


#endif // SENSOR_DISTANCIA_H
```

### `sensor_distancia.cpp`

```
#include "sensor_distancia.h"
#include <cstdlib> // rand()

SensorDistancia::SensorDistancia(float rango_maximo)
    : rango_maximo(rango_maximo) {}
```

```
float SensorDistancia::leer() {  
    // Simulación: devuelve un valor aleatorio entre 0 y rango_maximo  
    return static_cast<float>(rand() % static_cast<int>(rango_maximo));  
}
```

 main.cpp

```
#include "sensor_distancia.h"  
#include <iostream>  
  
int main() {  
    SensorDistancia sensor(100.0f); // Sensor con rango máximo de 100  
    cm  
  
    float valor = sensor.leer();  
    std::cout << "Distancia medida: " << valor << " cm" << std::endl;  
  
    return 0;  
}
```

## Compilación

Para compilarlo desde consola:

```
g++ main.cpp sensor_distancia.cpp -o programa  
./programa
```

# Anexo – Convenciones específicas para ROS 2

## Paquetes

- Los nombres de los paquetes se escriben en **snake\_case**.
- El nombre debe reflejar su propósito claramente.

✓ nav\_pkg, vision\_pkg, control\_motores

✗ Nav, VisionThing, Paquete1

## Nodos

- Los nodos también usan **snake\_case**.
- El nombre debe indicar **qué hace el nodo**.

✓ path\_planner, object\_detector, motor\_controller

✗ nodo1, prueba, main\_node

## Tópicos

- Los nombres de tópicos van en **minúsculas** y separados por **/**.
- Deben indicar claramente el **tipo de dato** o **fuentes**.
- Si son de un paquete concreto, se pueden agrupar por prefijo.

✓ /odom, /cmd\_vel, /camera/image\_raw, /vision/objects

✗ /datos, /topic1, /cosas

## Parámetros

- Los parámetros se nombran en **snake\_case**.
- Usar nombres descriptivos que indiquen su función.

✓ max\_velocidad, umbral\_error, tiempo\_espera

✗ vmax, par1, dato



Ejemplo en un `launch.py`:

```
from launch_ros.actions import Node

Node(
    package="nav_pkg",
    executable="path_planner",
    name="path_planner",
    parameters=[{"max_velocidad": 1.0, "tiempo_espera": 0.5}],
)
```

## Servicios y acciones

- **Servicios** → nombre en **snake\_case**, empezando con el verbo que describe la acción.  
✓ `/reset_odometria`, `/iniciar_mision`
- **Acciones** → deben indicar el objetivo de la acción.  
✓ `/mover_brazo`, `/navegar_a_punto`

## QoS y comunicación

- Para datos críticos (ej. control de motores) usar **QoS Reliability = RELIABLE**.
- Para sensores con alta frecuencia (ej. cámaras, LIDAR) usar **QoS Reliability = BEST\_EFFORT**.
- Documentar el QoS elegido en el paquete para evitar inconsistencias.

## Frames y TF

- Los nombres de los **frames** deben ser claros y consistentes con el estándar ROS.
- Frame raíz → `map`
- Base del robot → `base_link`
- Odometría → `odom`
- Ejemplos de sensores: `camera_link`, `lidar_link`, `imu_link`.

## Archivos de lanzamiento

- Todos los nodos deben poder ejecutarse desde un `.launch.py`.
- Los archivos se guardan en la carpeta `launch/` de cada paquete.
- El nombre del archivo debe indicar su propósito.

✓ `nav_sim.launch.py`, `vision_test.launch.py`

✗ `prueba.launch.py`, `launch1.py`