UPANZI
DIGITAL PUBLIC INFRASTRUCTURE
NETWORK

Pepper Robot Tour for Digital Experience Center

D3.2 Software Engineering Standards Manual

Due date: **15/03/2026**
Submission Date: **15/03/2026**
Revision Date:

Start date of project: **01/07/2026**

Duration: **12 months**

Responsible Person: **Yohannes Haile**

Revision: **1.0**

Project funded by Upanzi Network Dissemination Level		
PU	Public	**PU**
PP	Restricted to other programme participants	
RE	Restricted to a group specified by the consortium	
CO	Confidential, only for members of the consortium	

Executive Summary

Deliverable D3.2 presents the Software Engineering Standards Manual for the DEC Pepper Robot Tour project. This manual defines the software engineering standards and practices to be followed by all contributors developing software for the Pepper robot's autonomous tour guide system. It specifies: the guiding principles based on Component-Based Software Engineering (CBSE) applied to ROS2; the software development environment including tools, libraries, and version control; and the standards for specification, design, implementation, testing, and documentation of all software modules.

The manual is intended to ensure that all software developed within WP4 (Robot Sensing) and WP5 (Robot Behaviors) is consistent, interoperable, maintainable, and well-documented, enabling successful system integration as described in Deliverable D3.3.

Contents

Part I
Guiding Principles

1 Introduction

This document provides the software engineering standards manual for the Pepper Robot Tour for Upanzi's Digital Experience Center (DEC). The purpose is to establish a common set of engineering standards and practices for the development of software to be used in the DEC project, based on the Robot Operating System 2 (ROS2) Humble Hawksbill Open Robotics (2022) running on Ubuntu 22.04 LTS.

The software engineering standards are derived from established best practices in component-based software engineering Szyperski (2002) and are adapted to the specific requirements of ROS2 robotics software development. The standards are mandatory for all software submitted for integration into the DEC system.

1.1 Scope

These standards apply to all software modules developed in:

- **WP4 – Robot Sensing**: Person detection, face detection and gaze estimation, speech event detection, robot localization, conversation manager.
- **WP5 – Robot Behaviors**: ROS2 actuator control, behavior controller, Nav2 navigation, gesture animation system.

1.2 Overview

The DEC software follows a component-based architecture built on ROS2. Each software component is realized as a ROS2 *node* and communicates with other components using ROS2 topics, services, and actions. The system is organized as a *meta-package* (`dec_system`) containing individual functional packages, each implementing a specific sensing or behavior capability of the Pepper robot.

2 Component-Based Software Engineering

The DEC software is developed using Component-Based Software Engineering (CBSE) Szyperski (2002). In CBSE, a software system is composed of *components* — independently deployable units of software that interact through well-defined *interfaces*. CBSE promotes reusability, modularity, and replaceability of software units.

2.1 The Component-Port-Connector Model

The DEC software adopts the Component-Port-Connector (CPC) meta-model, in which:

- A **component** is a ROS2 node encapsulating a specific robot capability (e.g., face detection, behavior control).
- A **port** is a ROS2 topic, service, or action interface through which a component communicates.

- A **connector** is the ROS2 publish-subscribe or request-response mechanism linking producer ports to consumer ports.

Each component has:

- **Input ports** — topics subscribed to (consumed data) or services/actions provided.
- **Output ports** — topics published (produced data) or services/actions requested.
- **Configuration** — parameters loaded from YAML configuration files.

2.2 Guiding Principles

The following guiding principles apply to all DEC software development:

1. **Component Autonomy**: Each ROS2 node must be independently executable and testable using a driver-stub approach, without requiring other system components to be running.
2. **Interface Stability**: Published topic names, message types, service names, and action names constitute the public interface of a node and must not change without a version increment.
3. **Configuration-Driven Behavior**: All tunable parameters must be exposed via the ROS2 parameter server and loaded from a dedicated configuration file. Hard-coded parameter values are prohibited.
4. **Separation of Application and Implementation**: Each ROS2 package must separate the entry-point logic (application file) from the node implementation (implementation file), following the two-file pattern defined in Appendix A.
5. **Explicit Dependency Declaration**: All ROS2 and third-party library dependencies must be declared in `package.xml` and, for Python packages, in `setup.py`.
6. **Reproducible Build**: All packages must build successfully from source using `colcon build --symlink-install` in the `dec_ws` workspace without manual intervention.
7. **Naming Convention Compliance**: All file names, package names, node names, topic names, and parameter names must comply with the naming conventions defined in Appendix C.
8. **Mandatory Documentation**: Every source file must carry the standard file header. Every public function, class, and method must carry an API documentation comment as specified in Appendix B.
9. **Version Control**: All source code, configuration files, launch files, and models must be committed to the project GitHub repository at `https://github.com/yohatad/pepper4dec` with descriptive commit messages.

Part II
Software Development Environment

3 Development Platform

All DEC software is developed and deployed on **Ubuntu 22.04 LTS (Jammy Jellyfish)**. This is the officially supported platform for ROS2 Humble Hawksbill. Two programming languages (C++ and Python) will be used throughout the project.

3.1 ROS2 Framework

The Robot Operating System 2 (ROS2) **Humble Hawksbill** Open Robotics (2022) is the middleware framework for all DEC software. ROS2 provides:

- A publish-subscribe communication framework using DDS (Data Distribution Service) Object Management Group (2015).
- Client libraries: `rclpy` (Python) and `rclcpp` (C++).
- Build system: `ament_python` (Python packages) and `ament_cmake` (C++ packages).
- Build tool: `colcon` replacing ROS1's `catkin_make`.
- Launch system: Python `.launch.py` files.
- Parameter system with YAML configuration support.

3.2 Build System

Package Type	Build System	Build Files
Python node	`ament_python`	`setup.py`, `setup.cfg`, `package.xml`
C++ node	`ament_cmake`	`CMakeLists.txt`, `package.xml`
Interface package	`ament_cmake`	`CMakeLists.txt`, `package.xml`

Table 1: Build systems for different package types.

The standard build command for the entire workspace is:

```
cd ~/dec_ws
colcon build --symlink-install
source install/setup.bash
```

3.3 Version Control and Repository

All DEC software is hosted on GitHub at:

https://github.com/yohatad/pepper4dec

The repository contains the complete `dec_ws` workspace source. Contributors must fork the repository and submit pull requests for integration. Direct pushes to the `main` branch are not permitted.

Part III
Standards

4 Specification Standard

4.1 Module Specification

Every ROS2 node must be specified before implementation begins. The specification must include:

1. **Purpose**: A concise statement of what the node does.
2. **Input Ports**: All subscribed topics with message types and expected publish rates.
3. **Output Ports**: All published topics with message types and publish rates.
4. **Services**: Any ROS2 services provided or called by the node.
5. **Actions**: Any ROS2 actions provided or used by the node.
6. **Parameters**: All configuration parameters with names, types, default values, and valid ranges.
7. **Dependencies**: All ROS2 packages and third-party libraries required.

4.2 Interface Specification

Custom message types are defined in the `dec_interfaces` package. Interface definitions must follow ROS2 `.msg`, `.srv`, and `.action` file format conventions.

5 Design Standard

5.1 Package Structure Design

Each DEC node must be designed as an independent ROS2 package adhering to the standard package structure defined in Appendix A. The design must clearly distinguish:

- The **application file** — the entry point that initializes `rclpy`/`rclcpp`, instantiates the node, and handles lifecycle.
- The **implementation file** — the ROS2 node class implementing all publishers, subscribers, callbacks, and computational logic.
- The **configuration file** — YAML file containing all tunable parameters.
- The **launch file** — Python `.launch.py` file for launching the node.

5.2 Node Design Pattern

All nodes must follow the *callback-based* ROS2 design pattern:

1. Constructor declares all parameters, creates publishers, subscribers, timers, and clients.
2. Callback functions process incoming messages and trigger computation.
3. A `cleanup()` method releases all resources (models, connections, threads) before node destruction.

6 Implementation Standard

6.1 Python Implementation

Python nodes must:

- Use `rclpy.node.Node` as the base class.
- Declare all parameters using `self.declare_parameter()` in the constructor.
- Use Python type hints for all function signatures.
- Follow PEP 8 style conventions van Rossum et al. (2001) (enforced by `flake8`).
- Use the `logging` macros `self.get_logger().info()`, `.warning()`, and `.error()` for all output (no `print()` statements).

6.2 C++ Implementation

C++ nodes must:

- Use `rclcpp::Node` as the base class.
- Declare all parameters using `this->declare_parameter()` in the constructor.
- Use `RCLCPP_INFO`, `RCLCPP_WARN`, and `RCLCPP_ERROR` macros for all logging.
- Follow the Google C++ Style Guide Google LLC (2024) (enforced by `ament_cpplint`).
- Use smart pointers (`std::shared_ptr`, `std::unique_ptr`) for all heap-allocated objects.
- Use `#pragma once` header guards in all header files.

7 Documentation Standard

7.1 Required Documentation

Each software module must provide:

1. **Source file headers** — as specified in Appendix B.

2. **API documentation comments** — for all public functions and classes.
3. **Configuration file documentation** — inline comments explaining each parameter.
4. **README** — installation, build, and usage instructions specific to the package.
5. **Deliverable report section** — formal description of the module for the technical deliverable document.

7.2 Deliverable Report Standard

Each deliverable report section must include:

- A component specification table (ports, parameters, dependencies).
- A description of the algorithms and methods used.
- Experimental results or test results.
- References to relevant literature.

Appendix A File Organization

A.1 Workspace Layout

All DEC software resides in the `dec_ws` ROS2 workspace with the following structure:

```
dec_ws/
└── src/
    └── dec_system/
        ├── animate_behavior/
        ├── behaviorController/
        ├── conversation_manager/
        ├── dec_interfaces/
        ├── dec_launch/
        ├── face_detection/
        ├── gesture_execution/
        ├── overt_attention/
        ├── pepper_navmap/
        ├── person_detection/
        ├── speech_event/
        └── text_to_speech/
```

A.2 Python Package Structure

A Python ROS2 package (e.g., `face_detection`) must follow this structure:

```
face_detection/
├── face_detection/
│   ├── __init__.py
│   ├── face_detection_application.py
│   └── face_detection_implementation.py
├── launch/
│   └── face_detection.launch.py
├── config/
│   └── face_detection_configuration.yaml
├── models/
│   ├── face_detection.onnx
│   └── yolov11n.pt
├── test/
│   └── test_face_detection.py
├── resource/
│   └── face_detection
├── package.xml
├── setup.py
└── setup.cfg
```

A.3 C++ Package Structure

A C++ ROS2 package (e.g., `behavior_controller`) must follow this structure:

```
behavior_controller/
├── include/
│   └── behavior_controller/
│       └── behavior_controller_implementation.h
├── src/
│   ├── behavior_controller_application.cpp
│   └── behavior_controller_implementation.cpp
├── launch/
│   └── behavior_controller.launch.py
├── config/
│   └── behavior_controller_configuration.yaml
├── missions/
│   └── dec_tour_mission.xml
├── test/
│   └── test_behavior_controller.cpp
├── package.xml
└── CMakeLists.txt
```

A.4 Filename Conventions

File Type	Convention	Example
Python application	`<nodeName>_application.py`	`face_detection_application.py`
Python implementation	`<nodeName>_implementation.py`	`face_detection_implementation.py`
C++ application	`<nodeName>_application.cpp`	`behavior_controller_application.cpp`
C++ implementation	`<nodeName>_implementation.cpp/.h`	`behavior_controller_implementation.cpp`
YAML config	`<nodeName>_configuration.yaml`	`face_detection_configuration.yaml`
Launch file	`<nodeName>.launch.py`	`face_detection.launch.py`

Table 2: Filename conventions for DEC packages.

A.5 Package Build Files

A.5.1 `setup.py` – Python Package

The `setup.py` file for a Python package must follow this template, shown for the `face_detection` package:

```
from setuptools import setup
from glob import glob

pkg = "face_detection"

setup(
```

```
    name=pkg,
    version="0.1.0",
    packages=[pkg],
    maintainer="Yohannes",
    maintainer_email="yohanneh@alumni.cmu.edu",
    license="BSD-3-Clause",
    entry_points={
        "console_scripts": [
            "face_detection = face_detection.face_detection_application:
   main",
        ]
    },
    data_files=[
        ("share/ament_index/resource_index/packages", [f"resource/{pkg}"
   ]),
        (f"share/{pkg}", ["package.xml"]),
        (f"share/{pkg}/launch", glob("launch/*.launch.py")),
        (f"share/{pkg}/config", glob("config/*")),
        (f"share/{pkg}/models", glob("models/*")),
        (f"share/{pkg}/data", glob("data/*")),
    ],
)
```

Listing 1: `setup.py` for the `face_detection` package.

A.5.2 `package.xml` – Python Package

The `package.xml` (format 3) for the `face_detection` package:

```
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd"?>
<package format="3">
  <name>face_detection</name>
  <version>0.1.0</version>
  <description>
    ROS2 node for face detection, head pose estimation,
    and mutual gaze detection using SixDrepNet and YOLOv11.
  </description>
  <maintainer email="yohanneh@alumni.cmu.edu">Yohannes</maintainer>
  <license>BSD-3-Clause</license>

  <buildtool_depend>setuptools</buildtool_depend>

  <exec_depend>rclpy</exec_depend>
  <exec_depend>sensor_msgs</exec_depend>
  <exec_depend>cv_bridge</exec_depend>
  <exec_depend>dec_interfaces</exec_depend>

  <export>
    <build_type>ament_python</build_type>
  </export>
</package>
```

Listing 2: `package.xml` for the `face_detection` package.

A.5.3 Python Launch File

The ROS2 launch file for the `face_detection` node:

```python
import os
from launch import LaunchDescription
from launch_ros.actions import Node
from ament_index_python.packages import get_package_share_directory

def generate_launch_description():
    config = os.path.join(
        get_package_share_directory('face_detection'),
        'config',
        'face_detection_configuration.yaml'
    )

    return LaunchDescription([
        Node(
            package='face_detection',
            executable='face_detection',
            name='face_detection',
            output='screen',
            parameters=[config]
        )
    ])
```

Listing 3: Python launch file for the `face_detection` node.

A.5.4 YAML Configuration File

The YAML configuration file for the `face_detection` node:

```yaml
face_detection:
  ros__parameters:
    # Sensor input settings
    camera_type: realsense          # Camera type: realsense or webcam
    use_compressed_image: false     # Subscribe to compressed image topic

    # Debugging
    verbose_mode: false             # Enable verbose console output

    # Detection settings
    image_timeout: 2.0              # Seconds before image timeout
    warning
    confidence_threshold: 0.90      # Minimum detection confidence [0, 1]
    head_pose_threshold: 10.0       # Gaze angle threshold in degrees
    require_person_detection: true  # Require person before face
    detection
    object_detection_timeout: 0.5   # Timeout for person detection result
```

Listing 4: YAML configuration file for the `face_detection` node.

Appendix B Documentation

B.1 Source File Header

Every source file must begin with a standard file header. The header must include: the filename, a one-line description, the author, affiliation, email, date, and version.

B.1.1 Python File Header

```python
""" face_detection_application.py

    ROS2 Node for Face and Mutual Gaze Detection and Localization.

    Author:      Yohannes Tadesse Haile
    Affiliation: Carnegie Mellon University Africa
    Email:       yohatad123@gmail.com
    Date:        April 18, 2025
    Version:     v1.0

    Copyright (C) 2025 Carnegie Mellon University Africa
    This software is provided 'as-is' for research and
    educational purposes within the DEC project.
"""
```

Listing 5: Standard Python source file header.

B.1.2 C++ File Header

```cpp
/* behaviorControllerApplication.cpp
 *
 * ROS2 node for mission-based behavior execution using
 * BehaviorTree.CPP for the DEC Pepper Tour.
 *
 * Author:      Yohannes Tadesse Haile
 * Affiliation: Carnegie Mellon University Africa
 * Email:       yohatad123@gmail.com
 * Date:        April 18, 2025
 * Version:     v1.0
 *
 * Copyright (C) 2025 Carnegie Mellon University Africa
 * This software is provided 'as-is' for research and
 * educational purposes within the DEC project.
 */
```

Listing 6: Standard C++ source file header.

B.2 Application File API Documentation

The application file (entry point) must document the node's complete interface in its module-level docstring. This includes all topics subscribed and published, with message types and topic names.

```
""" face_detection_application.py

    ROS2 Node for Face and Mutual Gaze Detection and Localization.

    The face_detection node detects faces in an RGB camera image,
    estimates head pose using SixDrepNet (ONNX), and determines
    whether a person is making mutual gaze with the camera.

    Subscribers:
        /camera/color/image_raw   (sensor_msgs/Image)
            RGB image stream from RealSense or webcam camera.

        /personDetection/data     (dec_interfaces/PersonDetection)
            Person bounding box list from the person detection node.

    Publishers:
        /faceDetection/data       (dec_interfaces/FaceDetection)
            Face bounding boxes, head pose angles, and mutual gaze flag.

    Parameters (loaded from face_detection_configuration.yaml):
        camera_type               (str,   default: 'realsense')
        use_compressed_image      (bool,  default: False)
        verbose_mode              (bool,  default: False)
        image_timeout             (float, default: 2.0)
        confidence_threshold      (float, default: 0.90)
        head_pose_threshold       (float, default: 10.0)
        require_person_detection  (bool,  default: True)
        object_detection_timeout  (float, default: 0.5)

    Author:      Yohannes Tadesse Haile
    Affiliation: Carnegie Mellon University Africa
    Email:       yohatad123@gmail.com
    Date:        April 18, 2025
    Version:     v1.0
"""

import rclpy
from .face_detection_implementation import SixDrepNet, load_configuration

SOFTWARE_VERSION = "v1.0"

def main():
    """Entry point for the face_detection ROS2 node.

    Initializes rclpy, loads configuration from the YAML parameter
    file, instantiates the SixDrepNet node, and spins until shutdown.
    Ensures cleanup() is called on exit to release GPU/model resources.
    """
    rclpy.init()
    config = load_configuration()
    node = None
    try:
        node = SixDrepNet(config)
```

```
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        if node is not None:
            node.cleanup()
            node.destroy_node()
        if rclpy.ok():
            rclpy.shutdown()

if __name__ == "__main__":
    main()
```

Listing 7: Application file docstring with full API documentation.

B.3 Implementation File Documentation

Every class and public method in the implementation file must carry a docstring (Python) or Doxygen comment (C++).

B.3.1 Python Class and Method Docstrings

```
class SixDrepNet(Node):
    """ROS2 node for face detection and head pose estimation.

    Subscribes to a camera image topic and an optional person
    detection topic. Uses YOLOv11 for face detection and SixDrepNet
    (ONNX Runtime) for head pose estimation. Publishes the detected
    faces with pose angles to /faceDetection/data.

    Attributes:
        config (dict): Configuration parameters loaded from YAML.
        session (ort.InferenceSession): ONNX Runtime inference session.
        publisher_ (Publisher): Publisher for face detection results.
        subscription_ (Subscription): Subscriber for camera images.
    """

    def __init__(self, config: dict) -> None:
        """Initialize the SixDrepNet face detection node.

        Args:
            config (dict): Configuration dictionary loaded from the
                           YAML parameter file.
        """
        super().__init__('face_detection')
        self.config = config
        # ... initialization code ...

    def image_callback(self, msg: Image) -> None:
        """Process an incoming camera image message.

        Runs face detection using YOLOv11 and head pose estimation
```

```
        using SixDrepNet ONNX model. Publishes results to the
        /faceDetection/data topic.

        Args:
            msg (sensor_msgs/Image): Incoming RGB image message.
        """
        # ... callback implementation ...

    def cleanup(self) -> None:
        """Release all resources held by the node.

        Destroys the ONNX inference session, releases camera
        connections, and cancels any pending timers.
        """
        # ... cleanup implementation ...
```

Listing 8: Python class and method documentation pattern.

B.3.2 C++ Doxygen Comments

```
/**
 * @class BehaviorController
 * @brief ROS2 node for executing mission behavior trees.
 *
 * Loads a BehaviorTree.CPP XML mission file and executes the
 * behavior tree in response to operator commands. Communicates
 * with actuator control and navigation nodes via ROS2 topics
 * and action servers.
 */
class BehaviorController : public rclcpp::Node {
public:
    /**
     * @brief Construct a new BehaviorController node.
     *
     * Declares all parameters, loads the mission XML file,
     * registers BehaviorTree.CPP nodes, and creates ROS2
     * publishers and subscribers.
     *
     * @param options ROS2 node options (default: NodeOptions())
     */
    explicit BehaviorController(
        const rclcpp::NodeOptions& options = rclcpp::NodeOptions());

    /**
     * @brief Release all resources held by the node.
     *
     * Halts any running behavior tree, cancels pending action
     * goals, and destroys the BehaviorTree.CPP factory.
     */
    void cleanup();

private:
```

```
    /**
     * @brief Callback triggered when a mission command is received.
     *
     * @param msg The mission command message containing the
     *            mission name and execution parameters.
     */
    void missionCommandCallback(
        const dec_interfaces::msg::MissionCommand::SharedPtr msg);
};
```

Listing 9: C++ Doxygen-style documentation pattern.

Appendix C Programming Style

C.1 Naming Conventions

Entity	Python	C++	Example
Package name	`snake_case`	`snake_case`	`face_detection`
Node name	`camelCase`	`camelCase`	`faceDetection`
Class name	`PascalCase`	`PascalCase`	`SixDrepNet`
Function/method	`snake_case`	`camelCase`	`image_callback` / `imageCallback`
Variable	`snake_case`	`camelCase`	`confidence_threshold`
Constant	`UPPER_SNAKE`	`UPPER_SNAKE`	`SOFTWARE_VERSION`
Private member	`_name`	`name_`	`_session` / `session_`
Topic name	`camelCase/data`	`camelCase/data`	`/faceDetection/data`
Parameter name	`snake_case`	`snake_case`	`confidence_threshold`
File name	`snake_case.py`	`camelCase.cpp/.h`	`face_detection_application.py`

Table 3: Naming conventions for DEC software.

C.2 Import and Include Ordering

C.2.1 Python Import Order

Python imports must be grouped in the following order, with a blank line between each group:

1. Standard library modules (`os`, `sys`, `time`, etc.)
2. Third-party libraries (`numpy`, `cv2`, `onnxruntime`, etc.)
3. ROS2 libraries (`rclpy`, `sensor_msgs`, `dec_interfaces`, etc.)
4. Local package imports (relative imports using `.`)

```
# Standard library
import os
import time
from typing import Optional, List

```

```python
# Third-party
import numpy as np
import cv2
import onnxruntime as ort

# ROS2
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import Image
from cv_bridge import CvBridge
from dec_interfaces.msg import FaceDetection

# Local (relative)
from .face_detection_implementation import SixDrepNet
```

Listing 10: Python import ordering example.

C.2.2 C++ Include Order

C++ includes must be grouped in the following order:

1. Corresponding header file (for `.cpp` files)
2. ROS2 headers (`rclcpp`, message types)
3. Third-party library headers (`behaviortree_cpp_v3`)
4. Standard library headers (`string`, `memory`, etc.)

```cpp
// Corresponding header
#include "behavior_controller/behavior_controller_implementation.h"

// ROS2
#include "rclcpp/rclcpp.hpp"
#include "dec_interfaces/msg/mission_command.hpp"
#include "dec_interfaces/msg/mission_status.hpp"

// Third-party
#include "behaviortree_cpp_v3/bt_factory.h"
#include "behaviortree_cpp_v3/behavior_tree.h"

// Standard library
#include <string>
#include <memory>
#include <fstream>
```

Listing 11: C++ include ordering example.

Appendix D Programming Practice

D.1 Python Best Practices

D.1.1 Type Hints

All function signatures must use Python type hints. Return types must be annotated even for `None`-returning functions:

```
# Correct: full type annotation
def preprocess_image(self, image: np.ndarray,
                     size: tuple) -> np.ndarray:
    ...

def image_callback(self, msg: Image) -> None:
    ...

# Incorrect: missing type hints
def preprocess_image(self, image, size):
    ...
```

D.1.2 ROS2 Parameter Declaration

Parameters must be declared in the constructor before access. Use `get_parameter().value` to retrieve values:

```
def __init__(self, config: dict) -> None:
    super().__init__('face_detection')

    # Declare parameters with defaults
    self.declare_parameter('confidence_threshold', 0.90)
    self.declare_parameter('head_pose_threshold', 10.0)
    self.declare_parameter('verbose_mode', False)

    # Retrieve parameter values
    self.confidence_threshold = \
        self.get_parameter('confidence_threshold').value
    self.head_pose_threshold = \
        self.get_parameter('head_pose_threshold').value
    self.verbose_mode = \
        self.get_parameter('verbose_mode').value
```

D.1.3 ROS2 Logging

Use ROS2 logger macros exclusively. The `print()` function is prohibited in production code:

```
# Correct: use ROS2 logger
self.get_logger().info(f"Node initialized. Version: {SOFTWARE_VERSION}")
self.get_logger().warning("Image timeout -- no image received.")
self.get_logger().error(f"Failed to load ONNX model: {e}")

# Incorrect: do not use print()
print("Node initialized.")
```

D.2 C++ Best Practices

D.2.1 Header Guards

All C++ header files must use `#pragma once`:

```
#pragma once

#include "rclcpp/rclcpp.hpp"
#include "dec_interfaces/msg/mission_command.hpp"

class BehaviorController : public rclcpp::Node {
    // ...
};
```

D.2.2 Smart Pointers

Heap-allocated objects must use smart pointers. Raw `new`/`delete` is prohibited:

```
// Correct: use smart pointers
auto node = std::make_shared<BehaviorController>();
std::unique_ptr<BT::BehaviorTreeFactory> factory_ =
    std::make_unique<BT::BehaviorTreeFactory>();

// Incorrect: raw new/delete
BehaviorController* node = new BehaviorController();
```

D.2.3 ROS2 Logging Macros

Use the appropriate `RCLCPP_*` macro for all logging:

```
// Info-level logging
RCLCPP_INFO(this->get_logger(),
    "BehaviorController initialized. Version: %s",
    SOFTWARE_VERSION);

// Warning-level logging
RCLCPP_WARN(this->get_logger(),
    "Mission XML file not found: %s", mission_file_.c_str());

// Error-level logging
RCLCPP_ERROR(this->get_logger(),
    "Failed to execute behavior tree: %s", e.what());
```

D.2.4 ROS2 Parameter Declaration in C++

Parameters must be declared with types and descriptions using `ParameterDescriptor`:

```
// Declare a string parameter with description
rcl_interfaces::msg::ParameterDescriptor mission_desc;
mission_desc.description = "Path to the BehaviorTree XML mission file";
this->declare_parameter<std::string>(
    "mission_file", "dec_tour_mission.xml", mission_desc);
```

```
// Retrieve parameter value
std::string mission_file_ =
    this->get_parameter("mission_file").as_string();
```

References

Google LLC (2024). Google C++ style guide. https://google.github.io/styleguide/cppguide.html. Accessed: April 2025.

Object Management Group (2015). Data Distribution Service (DDS) specification, version 1.4. Technical Report formal/2015-04-10, Object Management Group.

Open Robotics (2022). ROS 2 Humble Hawksbill: Robot operating system 2. https://docs.ros.org/en/humble/index.html. Accessed: April 2025.

Szyperski, C. (2002). *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, 2nd edition.

van Rossum, G., Warsaw, B., and Coghlan, N. (2001). PEP 8 – style guide for Python code. https://peps.python.org/pep-0008/. Accessed: April 2025.

Version History

Version 1.0

Initial release.
Yohannes Haile.
15 March 2026.