# Design Patterns trong ROS2

ROS2 Design Pattern Guide

2025-06-25

## ABSTRACT FACTORY PATTERN TRONG ROS2

### 1. Giới thiệu đơn giản

Abstract Factory Pattern là một mẫu thiết kế khởi tạo cho phép tạo các họ đối tượng liên quan mà không cần chỉ định các lớp cụ thể của chúng. Trong ROS2, pattern này đặc biệt hữu ích khi:

- Tạo các bộ controllers hoàn chỉnh (position, velocity, effort)
- Khởi tạo các hệ thống sensor đa dạng (vision, lidar, IMU)
- Tạo các bộ message handlers và transformers
- Quản lý các subsystems phức tạp

### 2. Định nghĩa chi tiết

Abstract Factory Pattern cung cấp một interface để tạo các họ đối tượng liên quan hoặc phụ thuộc mà không cần chỉ định các lớp cụ thể của chúng.

**Các thành phần chính:**

1. **Abstract Factory**:
   - Interface chung cho việc tạo products
   - Định nghĩa các factory methods
2. **Concrete Factories**:
   - Implements abstract factory
   - Tạo các concrete products
3. **Abstract Products**:
   - Interface cho một loại product
   - Định nghĩa các operations chuẩn
4. **Concrete Products**:
   - Implements abstract products
   - Các sản phẩm cụ thể

### 3. Ví dụ thực tế trong ROS2

```cpp
// 1. Abstract Products
class MotionController {
public:
    virtual ~MotionController() = default;
    virtual bool initialize(const rclcpp::Node::SharedPtr& node) = 0;
    virtual bool setTarget(const geometry_msgs::msg::Pose& target) = 0;
    virtual geometry_msgs::msg::Twist computeCommand() = 0;
    virtual std::string getControllerType() const = 0;
};

class ObstacleDetector {
public:
    virtual ~ObstacleDetector() = default;
    virtual bool initialize(const rclcpp::Node::SharedPtr& node) = 0;
    virtual bool detectObstacles() = 0;
    virtual std::vector<geometry_msgs::msg::Point> getObstaclePositions() = 0;
    virtual std::string getDetectorType() const = 0;
};

class PathPlanner {
public:
    virtual ~PathPlanner() = default;
```

```
    virtual bool initialize(const rclcpp::Node::SharedPtr& node) = 0;
    virtual bool planPath(
        const geometry_msgs::msg::Pose& start,
        const geometry_msgs::msg::Pose& goal) = 0;
    virtual std::vector<geometry_msgs::msg::PoseStamped> getPath() = 0;
    virtual std::string getPlannerType() const = 0;
};

// 2. Concrete Products - Indoor Navigation System
class IndoorMotionController : public MotionController {
public:
    bool initialize(const rclcpp::Node::SharedPtr& node) override {
        try {
            node_ = node;
            // Load indoor-specific parameters
            auto params = node_->get_parameters("indoor_motion");
            max_vel_x_ = params[0].as_double();
            max_vel_theta_ = params[1].as_double();

            initialized_ = true;
            RCLCPP_INFO(node_->get_logger(), "Indoor Motion Controller initialized");
            return true;
        } catch (const std::exception& e) {
            RCLCPP_ERROR(node_->get_logger(),
                "Failed to initialize Indoor Motion Controller: %s", e.what());
            return false;
        }
    }

    bool setTarget(const geometry_msgs::msg::Pose& target) override {
        if (!initialized_) {
            RCLCPP_ERROR(node_->get_logger(), "Controller not initialized");
            return false;
        }
        current_target_ = target;
        return true;
    }

    geometry_msgs::msg::Twist computeCommand() override {
        geometry_msgs::msg::Twist cmd;
        // Indoor-specific motion control logic
        // Consider narrow spaces, doors, etc.
        return cmd;
    }

    std::string getControllerType() const override {
        return "INDOOR_MOTION_CONTROLLER";
    }

private:
    rclcpp::Node::SharedPtr node_;
    bool initialized_ = false;
    double max_vel_x_;
    double max_vel_theta_;
    geometry_msgs::msg::Pose current_target_;
};

class IndoorObstacleDetector : public ObstacleDetector {
public:
    bool initialize(const rclcpp::Node::SharedPtr& node) override {
        try {
            node_ = node;
            // Setup indoor obstacle detection parameters
            auto params = node_->get_parameters("indoor_detection");
            min_obstacle_height_ = params[0].as_double();
            max_obstacle_distance_ = params[1].as_double();

            initialized_ = true;
            RCLCPP_INFO(node_->get_logger(), "Indoor Obstacle Detector initialized");
            return true;
        } catch (const std::exception& e) {
            RCLCPP_ERROR(node_->get_logger(),
                "Failed to initialize Indoor Obstacle Detector: %s", e.what());
            return false;
        }
    }

    bool detectObstacles() override {
        if (!initialized_) {
```

```
            return false;
        }
        // Indoor-specific obstacle detection
        // Focus on walls, furniture, doors
        return true;
    }

    std::vector<geometry_msgs::msg::Point> getObstaclePositions() override {
        std::vector<geometry_msgs::msg::Point> obstacles;
        // Return detected indoor obstacles
        return obstacles;
    }

    std::string getDetectorType() const override {
        return "INDOOR_OBSTACLE_DETECTOR";
    }

private:
    rclcpp::Node::SharedPtr node_;
    bool initialized_ = false;
    double min_obstacle_height_;
    double max_obstacle_distance_;
};

class IndoorPathPlanner : public PathPlanner {
public:
    bool initialize(const rclcpp::Node::SharedPtr& node) override {
        try {
            node_ = node;
            // Setup indoor path planning parameters
            auto params = node_->get_parameters("indoor_planning");
            corridor_width_ = params[0].as_double();
            door_width_ = params[1].as_double();

            initialized_ = true;
            RCLCPP_INFO(node_->get_logger(), "Indoor Path Planner initialized");
            return true;
        } catch (const std::exception& e) {
            RCLCPP_ERROR(node_->get_logger(),
                "Failed to initialize Indoor Path Planner: %s", e.what());
            return false;
        }
    }

    bool planPath(
        const geometry_msgs::msg::Pose& start,
        const geometry_msgs::msg::Pose& goal) override {
        if (!initialized_) {
            return false;
        }
        // Indoor-specific path planning
        // Consider room layout, corridors, doors
        return true;
    }

    std::vector<geometry_msgs::msg::PoseStamped> getPath() override {
        std::vector<geometry_msgs::msg::PoseStamped> path;
        // Return planned indoor path
        return path;
    }

    std::string getPlannerType() const override {
        return "INDOOR_PATH_PLANNER";
    }

private:
    rclcpp::Node::SharedPtr node_;
    bool initialized_ = false;
    double corridor_width_;
    double door_width_;
};

// 3. Concrete Products - Outdoor Navigation System
class OutdoorMotionController : public MotionController {
public:
    bool initialize(const rclcpp::Node::SharedPtr& node) override {
        try {
            node_ = node;
            // Load outdoor-specific parameters
```

```
            auto params = node_->get_parameters("outdoor_motion");
            max_vel_x_ = params[0].as_double();
            max_vel_theta_ = params[1].as_double();
            terrain_type_ = params[2].as_string();

            initialized_ = true;
            RCLCPP_INFO(node_->get_logger(), "Outdoor Motion Controller initialized");
            return true;
        } catch (const std::exception& e) {
            RCLCPP_ERROR(node_->get_logger(),
                "Failed to initialize Outdoor Motion Controller: %s", e.what());
            return false;
        }
    }

    bool setTarget(const geometry_msgs::msg::Pose& target) override {
        if (!initialized_) {
            RCLCPP_ERROR(node_->get_logger(), "Controller not initialized");
            return false;
        }
        current_target_ = target;
        return true;
    }

    geometry_msgs::msg::Twist computeCommand() override {
        geometry_msgs::msg::Twist cmd;
        // Outdoor-specific motion control logic
        // Consider terrain, weather conditions
        return cmd;
    }

    std::string getControllerType() const override {
        return "OUTDOOR_MOTION_CONTROLLER";
    }

private:
    rclcpp::Node::SharedPtr node_;
    bool initialized_ = false;
    double max_vel_x_;
    double max_vel_theta_;
    std::string terrain_type_;
    geometry_msgs::msg::Pose current_target_;
};

class OutdoorObstacleDetector : public ObstacleDetector {
public:
    bool initialize(const rclcpp::Node::SharedPtr& node) override {
        try {
            node_ = node;
            // Setup outdoor obstacle detection parameters
            auto params = node_->get_parameters("outdoor_detection");
            min_obstacle_size_ = params[0].as_double();
            weather_condition_ = params[1].as_string();

            initialized_ = true;
            RCLCPP_INFO(node_->get_logger(), "Outdoor Obstacle Detector initialized");
            return true;
        } catch (const std::exception& e) {
            RCLCPP_ERROR(node_->get_logger(),
                "Failed to initialize Outdoor Obstacle Detector: %s", e.what());
            return false;
        }
    }

    bool detectObstacles() override {
        if (!initialized_) {
            return false;
        }
        // Outdoor-specific obstacle detection
        // Focus on terrain features, dynamic obstacles
        return true;
    }

    std::vector<geometry_msgs::msg::Point> getObstaclePositions() override {
        std::vector<geometry_msgs::msg::Point> obstacles;
        // Return detected outdoor obstacles
        return obstacles;
    }
```

```
    std::string getDetectorType() const override {
        return "OUTDOOR_OBSTACLE_DETECTOR";
    }

private:
    rclcpp::Node::SharedPtr node_;
    bool initialized_ = false;
    double min_obstacle_size_;
    std::string weather_condition_;
};

class OutdoorPathPlanner : public PathPlanner {
public:
    bool initialize(const rclcpp::Node::SharedPtr& node) override {
        try {
            node_ = node;
            // Setup outdoor path planning parameters
            auto params = node_->get_parameters("outdoor_planning");
            terrain_resolution_ = params[0].as_double();
            gps_accuracy_ = params[1].as_double();

            initialized_ = true;
            RCLCPP_INFO(node_->get_logger(), "Outdoor Path Planner initialized");
            return true;
        } catch (const std::exception& e) {
            RCLCPP_ERROR(node_->get_logger(),
                "Failed to initialize Outdoor Path Planner: %s", e.what());
            return false;
        }
    }

    bool planPath(
        const geometry_msgs::msg::Pose& start,
        const geometry_msgs::msg::Pose& goal) override {
        if (!initialized_) {
            return false;
        }
        // Outdoor-specific path planning
        // Consider terrain, GPS waypoints
        return true;
    }

    std::vector<geometry_msgs::msg::PoseStamped> getPath() override {
        std::vector<geometry_msgs::msg::PoseStamped> path;
        // Return planned outdoor path
        return path;
    }

    std::string getPlannerType() const override {
        return "OUTDOOR_PATH_PLANNER";
    }

private:
    rclcpp::Node::SharedPtr node_;
    bool initialized_ = false;
    double terrain_resolution_;
    double gps_accuracy_;
};

// 4. Abstract Factory
class NavigationSystemFactory {
public:
    virtual ~NavigationSystemFactory() = default;
    virtual std::unique_ptr<MotionController> createMotionController() = 0;
    virtual std::unique_ptr<ObstacleDetector> createObstacleDetector() = 0;
    virtual std::unique_ptr<PathPlanner> createPathPlanner() = 0;
};

// 5. Concrete Factories
class IndoorNavigationFactory : public NavigationSystemFactory {
public:
    std::unique_ptr<MotionController> createMotionController() override {
        return std::make_unique<IndoorMotionController>();
    }

    std::unique_ptr<ObstacleDetector> createObstacleDetector() override {
        return std::make_unique<IndoorObstacleDetector>();
    }
```

```
    std::unique_ptr<PathPlanner> createPathPlanner() override {
        return std::make_unique<IndoorPathPlanner>();
    }
};

class OutdoorNavigationFactory : public NavigationSystemFactory {
public:
    std::unique_ptr<MotionController> createMotionController() override {
        return std::make_unique<OutdoorMotionController>();
    }

    std::unique_ptr<ObstacleDetector> createObstacleDetector() override {
        return std::make_unique<OutdoorObstacleDetector>();
    }

    std::unique_ptr<PathPlanner> createPathPlanner() override {
        return std::make_unique<OutdoorPathPlanner>();
    }
};

// 6. Usage Example
class NavigationSystem {
public:
    NavigationSystem(
        const rclcpp::Node::SharedPtr& node,
        std::unique_ptr<NavigationSystemFactory> factory)
        : node_(node), factory_(std::move(factory)) {

        // Initialize all components
        initializeComponents();

        // Setup ROS2 communication
        setupCommunication();
    }

    bool navigate(
        const geometry_msgs::msg::Pose& start,
        const geometry_msgs::msg::Pose& goal) {
        try {
            // 1. Detect obstacles
            if (!obstacle_detector_->detectObstacles()) {
                RCLCPP_ERROR(node_->get_logger(), "Failed to detect obstacles");
                return false;
            }

            // 2. Plan path
            if (!path_planner_->planPath(start, goal)) {
                RCLCPP_ERROR(node_->get_logger(), "Failed to plan path");
                return false;
            }

            // 3. Follow path
            auto path = path_planner_->getPath();
            for (const auto& pose : path) {
                if (!motion_controller_->setTarget(pose.pose)) {
                    RCLCPP_ERROR(node_->get_logger(), "Failed to set target");
                    return false;
                }

                // Execute motion
                auto cmd_vel = motion_controller_->computeCommand();
                cmd_vel_pub_->publish(cmd_vel);
            }

            return true;
        } catch (const std::exception& e) {
            RCLCPP_ERROR(node_->get_logger(),
                "Navigation error: %s", e.what());
            return false;
        }
    }

private:
    void initializeComponents() {
        // Create components using factory
        motion_controller_ = factory_->createMotionController();
        obstacle_detector_ = factory_->createObstacleDetector();
        path_planner_ = factory_->createPathPlanner();
```

```
        // Initialize components
        if (!motion_controller_->initialize(node_)) {
            throw std::runtime_error("Failed to initialize motion controller");
        }
        if (!obstacle_detector_->initialize(node_)) {
            throw std::runtime_error("Failed to initialize obstacle detector");
        }
        if (!path_planner_->initialize(node_)) {
            throw std::runtime_error("Failed to initialize path planner");
        }
    }

    void setupCommunication() {
        // Create publishers and subscribers
        cmd_vel_pub_ = node_->create_publisher<geometry_msgs::msg::Twist>(
            "cmd_vel", 10);

        goal_sub_ = node_->create_subscription<geometry_msgs::msg::PoseStamped>(
            "goal_pose", 1,
            std::bind(&NavigationSystem::goalCallback, this, std::placeholders::_1));
    }

    void goalCallback(
        const geometry_msgs::msg::PoseStamped::SharedPtr msg) {
        // Get current pose
        geometry_msgs::msg::Pose current_pose;
        // TODO: Get current pose from tf or odometry

        // Navigate to goal
        if (!navigate(current_pose, msg->pose)) {
            RCLCPP_ERROR(node_->get_logger(), "Navigation failed");
        }
    }

    rclcpp::Node::SharedPtr node_;
    std::unique_ptr<NavigationSystemFactory> factory_;
    std::unique_ptr<MotionController> motion_controller_;
    std::unique_ptr<ObstacleDetector> obstacle_detector_;
    std::unique_ptr<PathPlanner> path_planner_;

    rclcpp::Publisher<geometry_msgs::msg::Twist>::SharedPtr cmd_vel_pub_;
    rclcpp::Subscription<geometry_msgs::msg::PoseStamped>::SharedPtr goal_sub_;
};

// 7. Main function
int main(int argc, char** argv) {
    rclcpp::init(argc, argv);
    auto node = std::make_shared<rclcpp::Node>("navigation_node");

    // Get environment type from parameter
    std::string env_type = node->declare_parameter("environment_type", "indoor");

    // Create appropriate factory
    std::unique_ptr<NavigationSystemFactory> factory;
    if (env_type == "indoor") {
        factory = std::make_unique<IndoorNavigationFactory>();
    } else if (env_type == "outdoor") {
        factory = std::make_unique<OutdoorNavigationFactory>();
    } else {
        RCLCPP_ERROR(node->get_logger(),
            "Unknown environment type: %s", env_type.c_str());
        return 1;
    }

    // Create navigation system
    auto navigation = std::make_shared<NavigationSystem>(node, std::move(factory));

    // Spin node
    rclcpp::spin(node);
    rclcpp::shutdown();
    return 0;
}
```

### 4. Giải thích chi tiết cách hoạt động

1. **Abstract Factory**:
    - NavigationSystemFactory định nghĩa interface

- Tạo các components liên quan
- Đảm bảo tính tương thích

2. **Concrete Factories**:
   - IndoorNavigationFactory cho môi trường trong nhà
   - OutdoorNavigationFactory cho môi trường ngoài trời
   - Tạo các components phù hợp
3. **Abstract Products**:
   - MotionController, ObstacleDetector, PathPlanner
   - Định nghĩa interface chung
   - Tính đa hình thông qua virtual methods

### 5. Ưu điểm trong ROS2

1. **System Consistency**:
   - Components tương thích
   - Cấu hình nhất quán
   - Error handling đồng bộ
2. **Flexibility**:
   - Dễ thêm môi trường mới
   - Runtime configuration
   - Component swapping
3. **Maintainability**:
   - Code organization rõ ràng
   - Separation of concerns
   - Dễ test và debug

### 6. Các trường hợp sử dụng trong ROS2

1. **Navigation Systems**:
   - Indoor/Outdoor navigation
   - Multi-robot systems
   - Hybrid environments
2. **Sensor Systems**:
   - Different sensor suites
   - Environment-specific processing
   - Sensor fusion
3. **Control Systems**:
   - Different control strategies
   - Environment adaptation
   - Multi-mode operation

### 7. Best Practices trong ROS2

1. **Error Handling**:

```
try {
    auto system = createNavigationSystem(env_type);
    if (!system->initialize()) {
        RCLCPP_ERROR(logger, "System initialization failed");
        return;
    }
} catch (const std::exception& e) {
    RCLCPP_ERROR(logger, "System creation error: %s", e.what());
}
```

2. **Configuration Management**:

```
class ConfigurableFactory : public NavigationSystemFactory {
    void configure(const YAML::Node& config) {
        // Configure factory from YAML
        loadConfiguration(config);
    }
};
```

3. **Resource Management**:

```
class SafeNavigation {
    std::unique_ptr<NavigationSystem> system_;
```

```
public:
    ~SafeNavigation() {
        if (system_) {
            system_->shutdown();
        }
    }
};
```

### 8. Mở rộng và tùy chỉnh

1. **Environment Detection**:

```
class AutoNavigationFactory : public NavigationSystemFactory {
    std::unique_ptr<NavigationSystemFactory> detectEnvironment() {
        // Detect environment type and return appropriate factory
        if (isIndoorEnvironment()) {
            return std::make_unique<IndoorNavigationFactory>();
        }
        return std::make_unique<OutdoorNavigationFactory>();
    }
};
```

2. **Hybrid Systems**:

```
class HybridNavigationFactory : public NavigationSystemFactory {
    std::unique_ptr<MotionController> createMotionController() override {
        // Create controller that can handle both environments
        return std::make_unique<HybridMotionController>();
    }
};
```

3. **Plugin System**:

```
class PluginNavigationFactory : public NavigationSystemFactory {
    std::unique_ptr<MotionController> loadControllerPlugin(
        const std::string& name) {
        // Load controller plugin dynamically
        return loadPlugin<MotionController>(name);
    }
};
```

### 9. Testing

1. **Mock Objects**:

```
class MockNavigationFactory : public NavigationSystemFactory {
public:
    MOCK_METHOD(std::unique_ptr<MotionController>,
                createMotionController, (), (override));
    MOCK_METHOD(std::unique_ptr<ObstacleDetector>,
                createObstacleDetector, (), (override));
    MOCK_METHOD(std::unique_ptr<PathPlanner>,
                createPathPlanner, (), (override));
};
```

2. **Factory Tests**:

```
TEST(NavigationTest, IndoorFactoryTest) {
    auto factory = std::make_unique<IndoorNavigationFactory>();
    auto controller = factory->createMotionController();
    EXPECT_NE(controller, nullptr);
    EXPECT_EQ(controller->getControllerType(), "INDOOR_MOTION_CONTROLLER");
}
```

3. **Integration Tests**:

```
TEST(NavigationSystemTest, FullSystemTest) {
    auto node = std::make_shared<rclcpp::Node>("test_node");
    auto factory = std::make_unique<IndoorNavigationFactory>();
    auto navigation = std::make_shared<NavigationSystem>(
        node, std::move(factory));

    geometry_msgs::msg::Pose start, goal;
    EXPECT_TRUE(navigation->navigate(start, goal));
}
```

## 10. Kết luận

Abstract Factory Pattern là một mẫu thiết kế quan trọng trong ROS2, đặc biệt hữu ích cho việc tạo các hệ thống phức tạp với nhiều components liên quan. Pattern này mang lại nhiều lợi ích:

1. **System Consistency**:
   - Đảm bảo tính tương thích giữa các components
   - Cấu hình nhất quán cho từng môi trường
   - Error handling đồng bộ
2. **Flexibility và Extensibility**:
   - Dễ dàng thêm môi trường mới
   - Runtime configuration
   - Component swapping linh hoạt
3. **Code Organization**:
   - Separation of concerns rõ ràng
   - Interface standards
   - Dễ maintain và test
4. **Resource Management**:
   - Clean initialization và cleanup
   - Safe resource handling
   - Memory management hiệu quả

Trong ví dụ về navigation system, chúng ta đã thấy Abstract Factory Pattern giúp xây dựng một hệ thống navigation linh hoạt và mạnh mẽ, có thể dễ dàng chuyển đổi giữa môi trường trong nhà và ngoài trời. Pattern này là lựa chọn tốt cho các hệ thống ROS2 cần quản lý nhiều components liên quan và có thể hoạt động trong các môi trường khác nhau.

# Design Patterns trong ROS2

ROS2 Design Pattern Guide

2025-06-25

## FACTORY PATTERN TRONG ROS2

**1. Giới thiệu đơn giản** Factory Pattern là một mẫu thiết kế khởi tạo cho phép tạo các đối tượng mà không cần chỉ định chính xác lớp của chúng. Trong ROS2, pattern này đặc biệt hữu ích khi:

- Tạo các node plugins khác nhau
- Khởi tạo các sensor drivers
- Tạo các message handlers
- Quản lý các loại controllers khác nhau

**2. Định nghĩa chi tiết** Factory Pattern định nghĩa một interface để tạo đối tượng nhưng để các lớp con quyết định lớp nào sẽ được khởi tạo. Pattern này cho phép một lớp hoãn việc khởi tạo sang lớp con.

**Các thành phần chính:**

1. **Product Interface**:
   - Interface chung cho tất cả products
   - Định nghĩa các operations chuẩn
2. **Concrete Products**:
   - Các implementations cụ thể
   - Tuân theo product interface
3. **Factory Interface**:
   - Định nghĩa phương thức tạo product
   - Có thể có nhiều factory methods
4. **Concrete Factory**:
   - Implements factory interface
   - Tạo các concrete products

```cpp
// 1. Product Interface
class SensorDriver {
public:
    virtual ~SensorDriver() = default;
    virtual bool initialize(const rclcpp::Node::SharedPtr& node) = 0;
    virtual bool start() = 0;
    virtual bool stop() = 0;
    virtual std::vector<double> getData() = 0;
    virtual std::string getSensorType() const = 0;
    virtual void setParameters(const std::map<std::string, std::string>& params) = 0;
};

// 2. Concrete Products
class LidarDriver : public SensorDriver {
public:
    bool initialize(const rclcpp::Node::SharedPtr& node) override {
        try {
            node_ = node;
            // Đọc parameters
            auto params = node_->get_parameters("lidar");
            port_ = params[0].as_string();
            baud_rate_ = params[1].as_int();

            // Khởi tạo lidar connection
            return initializeLidarConnection();
        } catch (const std::exception& e) {
            RCLCPP_ERROR(node_->get_logger(),
                "Failed to initialize LidarDriver: %s", e.what());
```

```
            return false;
        }
    }

    bool start() override {
        if (!is_initialized_) {
            RCLCPP_ERROR(node_->get_logger(), "LidarDriver not initialized");
            return false;
        }

        try {
            // Bắt đầu scanning
            startScanning();
            is_running_ = true;
            return true;
        } catch (const std::exception& e) {
            RCLCPP_ERROR(node_->get_logger(),
                "Failed to start LidarDriver: %s", e.what());
            return false;
        }
    }

    bool stop() override {
        if (!is_running_) {
            return true;
        }

        try {
            // Dừng scanning
            stopScanning();
            is_running_ = false;
            return true;
        } catch (const std::exception& e) {
            RCLCPP_ERROR(node_->get_logger(),
                "Failed to stop LidarDriver: %s", e.what());
            return false;
        }
    }

    std::vector<double> getData() override {
        if (!is_running_) {
            RCLCPP_WARN(node_->get_logger(), "LidarDriver not running");
            return std::vector<double>();
        }

        try {
            // Đọc và xử lý dữ liệu lidar
            return readLidarData();
        } catch (const std::exception& e) {
            RCLCPP_ERROR(node_->get_logger(),
                "Failed to read LidarDriver data: %s", e.what());
            return std::vector<double>();
        }
    }

    std::string getSensorType() const override {
        return "LIDAR";
    }

    void setParameters(const std::map<std::string, std::string>& params) override {
        for (const auto& [key, value] : params) {
            if (key == "port") {
                port_ = value;
            } else if (key == "baud_rate") {
                baud_rate_ = std::stoi(value);
            }
            // Thêm các parameters khác
        }
    }

private:
    bool initializeLidarConnection() {
        // Khởi tạo kết nối với lidar hardware
        return true;
    }

    void startScanning() {
        // Bắt đầu quét lidar
    }
```

```
    void stopScanning() {
        // Dừng quét lidar
    }

    std::vector<double> readLidarData() {
        // Đọc và xử lý dữ liệu từ lidar
        return std::vector<double>();
    }

    rclcpp::Node::SharedPtr node_;
    std::string port_;
    int baud_rate_;
    bool is_initialized_ = false;
    bool is_running_ = false;
};

class CameraDriver : public SensorDriver {
public:
    bool initialize(const rclcpp::Node::SharedPtr& node) override {
        try {
            node_ = node;
            // Đọc camera parameters
            resolution_ = params[0].as_string();
            fps_ = params[1].as_int();

            // Khởi tạo camera
            return initializeCamera();
        } catch (const std::exception& e) {
            RCLCPP_ERROR(node_->get_logger(),
                "Failed to initialize CameraDriver: %s", e.what());
            return false;
        }
    }

    bool start() override {
        if (!is_initialized_) {
            RCLCPP_ERROR(node_->get_logger(), "CameraDriver not initialized");
            return false;
        }

        try {
            // Bắt đầu streaming
            startStreaming();
            is_running_ = true;
            return true;
        } catch (const std::exception& e) {
            RCLCPP_ERROR(node_->get_logger(),
                "Failed to start CameraDriver: %s", e.what());
            return false;
        }
    }

    bool stop() override {
        if (!is_running_) {
            return true;
        }

        try {
            // Dừng streaming
            stopStreaming();
            is_running_ = false;
            return true;
        } catch (const std::exception& e) {
            RCLCPP_ERROR(node_->get_logger(),
                "Failed to stop CameraDriver: %s", e.what());
            return false;
        }
    }

    std::vector<double> getData() override {
        if (!is_running_) {
            RCLCPP_WARN(node_->get_logger(), "CameraDriver not running");
            return std::vector<double>();
        }

        try {
            // Đọc và xử lý frame
            return processFrame();
```

```
        } catch (const std::exception& e) {
            RCLCPP_ERROR(node_->get_logger(),
                "Failed to read CameraDriver data: %s", e.what());
            return std::vector<double>();
        }
    }

    std::string getSensorType() const override {
        return "CAMERA";
    }

    void setParameters(const std::map<std::string, std::string>& params) override {
        for (const auto& [key, value] : params) {
            if (key == "resolution") {
                resolution_ = value;
            } else if (key == "fps") {
                fps_ = std::stoi(value);
            }
            // Thêm các parameters khác
        }
    }

private:
    bool initializeCamera() {
        // Khởi tạo camera
        return true;
    }

    void startStreaming() {
        // Bắt đầu streaming
    }

    void stopStreaming() {
        // Dừng streaming
    }

    std::vector<double> processFrame() {
        // Xử lý frame hiện tại
        return std::vector<double>();
    }

    rclcpp::Node::SharedPtr node_;
    std::string resolution_;
    int fps_;
    bool is_initialized_ = false;
    bool is_running_ = false;
};

// 3. Factory Interface
class SensorDriverFactory {
public:
    virtual ~SensorDriverFactory() = default;
    virtual std::unique_ptr<SensorDriver> createDriver(
        const std::string& sensor_type,
        const std::map<std::string, std::string>& params) = 0;
};

// 4. Concrete Factory
class RobotSensorFactory : public SensorDriverFactory {
public:
    std::unique_ptr<SensorDriver> createDriver(
        const std::string& sensor_type,
        const std::map<std::string, std::string>& params) override {

        std::unique_ptr<SensorDriver> driver = nullptr;

        if (sensor_type == "LIDAR") {
            driver = std::make_unique<LidarDriver>();
        } else if (sensor_type == "CAMERA") {
            driver = std::make_unique<CameraDriver>();
        } else {
            throw std::runtime_error("Unknown sensor type: " + sensor_type);
        }

        if (driver) {
            driver->setParameters(params);
        }

        return driver;
```

```
    }
};

// 5. Usage Example
class RobotSensorNode : public rclcpp::Node {
public:
    RobotSensorNode() : Node("robot_sensor") {
        // Tạo factory
        sensor_factory_ = std::make_unique<RobotSensorFactory>();

        // Khởi tạo các sensors
        initializeSensors();

        // Tạo timer để publish sensor data
        timer_ = create_wall_timer(
            std::chrono::milliseconds(100),
            std::bind(&RobotSensorNode::publishSensorData, this));
    }

    ~RobotSensorNode() {
        // Cleanup
        for (auto& sensor : sensors_) {
            sensor->stop();
        }
    }

private:
    void initializeSensors() {
        try {
            // Khởi tạo LIDAR
            std::map<std::string, std::string> lidar_params = {
                {"port", "/dev/ttyUSB0"},
                {"baud_rate", "115200"}
            };
            auto lidar = sensor_factory_->createDriver("LIDAR", lidar_params);
            if (lidar->initialize(shared_from_this())) {
                lidar->start();
                sensors_.push_back(std::move(lidar));
            }

            // Khởi tạo Camera
            std::map<std::string, std::string> camera_params = {
                {"resolution", "640x480"},
                {"fps", "30"}
            };
            auto camera = sensor_factory_->createDriver("CAMERA", camera_params);
            if (camera->initialize(shared_from_this())) {
                camera->start();
                sensors_.push_back(std::move(camera));
            }
        } catch (const std::exception& e) {
            RCLCPP_ERROR(get_logger(), "Failed to initialize sensors: %s", e.what());
        }
    }

    void publishSensorData() {
        for (auto& sensor : sensors_) {
            try {
                auto data = sensor->getData();
                // Process và publish data
                publishData(sensor->getSensorType(), data);
            } catch (const std::exception& e) {
                RCLCPP_ERROR(get_logger(),
                    "Error reading %s data: %s",
                    sensor->getSensorType().c_str(), e.what());
            }
        }
    }

    void publishData(const std::string& sensor_type,
                     const std::vector<double>& data) {
        // Publish sensor data
    }

    std::unique_ptr<SensorDriverFactory> sensor_factory_;
    std::vector<std::unique_ptr<SensorDriver>> sensors_;
    rclcpp::TimerBase::SharedPtr timer_;
};
```

```
#### 4. Giải thích chi tiết cách hoạt động
1. **Product Interface**:
   - SensorDriver định nghĩa interface chung
   - Các phương thức chuẩn cho mọi sensor
   - Tính đa hình thông qua virtual methods

2. **Concrete Products**:
   - LidarDriver và CameraDriver implements interface
   - Xử lý hardware-specific logic
   - Error handling và logging

3. **Factory Pattern**:
   - Factory method tạo sensor drivers
   - Encapsulation của object creation
   - Dynamic object creation

#### 5. Ưu điểm trong ROS2
1. **Flexibility**:
   - Dễ dàng thêm sensors mới
   - Không cần sửa code hiện có
   - Runtime configuration

2. **Maintainability**:
   - Code organization rõ ràng
   - Separation of concerns
   - Dễ test và debug

3. **Reusability**:
   - Common interface cho sensors
   - Shared functionality
   - Code reuse

#### 6. Các trường hợp sử dụng trong ROS2
1. **Sensor Systems**:
   - Different types of sensors
   - Multiple sensor configurations
   - Sensor fusion systems

2. **Plugin Management**:
   - ROS2 plugins
   - Custom node creation
   - Dynamic loading

3. **Message Handlers**:
   - Custom message types
   - Protocol adapters
   - Communication handlers

#### 7. Best Practices trong ROS2
1. **Error Handling**:
```cpp
try {
    auto driver = factory->createDriver("LIDAR", params);
    if (!driver->initialize(node)) {
        RCLCPP_ERROR(logger, "Driver initialization failed");
        return;
    }
} catch (const std::exception& e) {
    RCLCPP_ERROR(logger, "Error creating driver: %s", e.what());
}
```

### 3. Ví dụ thực tế trong ROS2

2. **Parameter Management**:

```cpp
void loadParameters() {
    auto params = node_->get_parameters("sensor");
    for (const auto& param : params) {
        config_[param.get_name()] = param.value_to_string();
    }
}
```

3. **Resource Management**:

```cpp
class SafeDriver {
    std::unique_ptr<SensorDriver> driver_;
public:
    ~SafeDriver() {
        if (driver_) {
```
```

```
            driver_->stop();
        }
    }
};
```

### 8. Mở rộng và tùy chỉnh

1. **Dynamic Loading**:

```
class PluginFactory : public SensorDriverFactory {
    std::unique_ptr<SensorDriver> createDriver(
        const std::string& type,
        const std::map<std::string, std::string>& params) override {
        // Load plugin dynamically
        return loadPlugin(type, params);
    }
};
```

2. **Configuration System**:

```
class ConfigurableFactory : public SensorDriverFactory {
    void configure(const YAML::Node& config) {
        // Configure factory from YAML
        loadConfiguration(config);
    }
};
```

3. **Validation System**:

```
class ValidatingFactory : public SensorDriverFactory {
    bool validateParams(const std::map<std::string, std::string>& params) {
        // Validate parameters
        return checkParameters(params);
    }
};
```

### 9. Testing

1. **Mock Objects**:

```
class MockSensorDriver : public SensorDriver {
public:
    MOCK_METHOD(bool, initialize, (const rclcpp::Node::SharedPtr&), (override));
    MOCK_METHOD(bool, start, (), (override));
    MOCK_METHOD(std::vector<double>, getData, (), (override));
};
```

2. **Factory Tests**:

```
TEST(FactoryTest, CreateLidarDriver) {
    auto factory = std::make_unique<RobotSensorFactory>();
    auto driver = factory->createDriver("LIDAR", {});
    EXPECT_NE(driver, nullptr);
    EXPECT_EQ(driver->getSensorType(), "LIDAR");
}
```

3. **Integration Tests**:

```
TEST(SensorSystemTest, FullSystemTest) {
    auto node = std::make_shared<rclcpp::Node>("test_node");
    auto factory = std::make_unique<RobotSensorFactory>();

    // Test LIDAR creation and initialization
    auto lidar = factory->createDriver("LIDAR", {
        {"port", "/dev/ttyUSB0"},
        {"baud_rate", "115200"}
    });
    EXPECT_TRUE(lidar->initialize(node));
    EXPECT_TRUE(lidar->start());

    // Test data acquisition
    auto data = lidar->getData();
    EXPECT_FALSE(data.empty());

    // Cleanup
    EXPECT_TRUE(lidar->stop());
}
```

**10. Kết luận** Factory Pattern là một mẫu thiết kế quan trọng trong ROS2, đặc biệt hữu ích cho việc quản lý và tạo các đối tượng phức tạp như sensor drivers, plugins, và message handlers. Pattern này mang lại nhiều lợi ích:

1. **Flexibility và Extensibility**:
    - Dễ dàng thêm sensor types mới
    - Không cần sửa đổi code hiện có
    - Runtime configuration và loading
2. **Code Organization**:
    - Separation of concerns rõ ràng
    - Interface standards
    - Dễ maintain và test
3. **Error Handling**:
    - Centralized error management
    - Graceful error recovery
    - Consistent logging
4. **Resource Management**:
    - Clean initialization và cleanup
    - Safe resource handling
    - Memory management

Trong ví dụ về sensor system, chúng ta đã thấy Factory Pattern giúp xây dựng một hệ thống sensor modular, extensible và maintainable. Pattern này là lựa chọn tốt cho các hệ thống ROS2 cần quản lý nhiều loại đối tượng khác nhau một cách linh hoạt và an toàn.

# Design Patterns trong ROS2

ROS2 Design Pattern Guide

2025-06-25

## SINGLETON PATTERN TRONG ROS2

**1. Giới thiệu đơn giản** Tưởng tượng bạn có một robot với nhiều thành phần khác nhau (động cơ, cảm biến, bộ điều khiển) đều cần truy cập vào hardware interface. Việc có nhiều instances của hardware interface có thể gây ra:

- Xung đột khi truy cập hardware
- Lãng phí tài nguyên hệ thống
- Khó đồng bộ hóa trạng thái

Singleton Pattern đảm bảo chỉ có một instance duy nhất của hardware interface, giúp quản lý truy cập an toàn và hiệu quả.

**2. Định nghĩa chi tiết** Singleton Pattern là một mẫu thiết kế khởi tạo đảm bảo một class chỉ có một instance duy nhất và cung cấp một điểm truy cập toàn cục đến instance đó.

**Các thành phần chính:**

1. **Private Constructor**:
   - Ngăn việc tạo instance từ bên ngoài
   - Kiểm soát quá trình khởi tạo
2. **Static Instance**:
   - Lưu trữ instance duy nhất
   - Đảm bảo tính duy nhất
3. **Public Access Method**:
   - Cung cấp điểm truy cập toàn cục
   - Tạo instance nếu chưa tồn tại

```
// Hardware Interface Singleton
class RobotHardwareInterface {
public:
    static std::shared_ptr<RobotHardwareInterface> getInstance() {
        static std::mutex mutex;
        std::lock_guard<std::mutex> lock(mutex);

        if (!instance_) {
            instance_ = std::shared_ptr<RobotHardwareInterface>(new RobotHardwareInterface());
        }
        return instance_;
    }

    // Ngăn copy constructor và assignment operator
    RobotHardwareInterface(const RobotHardwareInterface&) = delete;
    RobotHardwareInterface& operator=(const RobotHardwareInterface&) = delete;

    // Hardware control methods
    bool initializeHardware() {
        if (is_initialized_) {
            RCLCPP_WARN(logger_, "Hardware already initialized");
            return true;
        }

        try {
            // Khởi tạo kết nối với hardware
            setupCommunication();
            setupMotors();
            setupSensors();
```

```
            is_initialized_ = true;
            RCLCPP_INFO(logger_, "Hardware initialized successfully");
            return true;
        } catch (const std::exception& e) {
            RCLCPP_ERROR(logger_, "Failed to initialize hardware: %s", e.what());
            return false;
        }
    }

    bool setMotorSpeed(int motor_id, double speed) {
        std::lock_guard<std::mutex> lock(motor_mutex_);
        try {
            validateMotorId(motor_id);
            validateSpeed(speed);

            // Gửi lệnh tới motor
            sendMotorCommand(motor_id, speed);
            return true;
        } catch (const std::exception& e) {
            RCLCPP_ERROR(logger_, "Motor control error: %s", e.what());
            return false;
        }
    }

    std::vector<double> getSensorData() {
        std::lock_guard<std::mutex> lock(sensor_mutex_);
        try {
            // Đọc dữ liệu từ tất cả sensors
            return readSensorValues();
        } catch (const std::exception& e) {
            RCLCPP_ERROR(logger_, "Sensor read error: %s", e.what());
            return std::vector<double>();
        }
    }

    void shutdown() {
        std::lock_guard<std::mutex> lock(motor_mutex_);
        try {
            // Dừng tất cả motors
            stopAllMotors();
            // Đóng kết nối
            closeConnection();

            is_initialized_ = false;
            RCLCPP_INFO(logger_, "Hardware shutdown complete");
        } catch (const std::exception& e) {
            RCLCPP_ERROR(logger_, "Shutdown error: %s", e.what());
        }
    }

private:
    RobotHardwareInterface()
        : logger_(rclcpp::get_logger("RobotHardwareInterface")) {
        // Private constructor
    }

    void setupCommunication() {
        // Khởi tạo giao thức truyền thông
    }

    void setupMotors() {
        // Khởi tạo các động cơ
    }

    void setupSensors() {
        // Khởi tạo các cảm biến
    }

    void validateMotorId(int motor_id) {
        if (motor_id < 0 || motor_id >= MAX_MOTORS) {
            throw std::out_of_range("Invalid motor ID");
        }
    }

    void validateSpeed(double speed) {
        if (speed < -MAX_SPEED || speed > MAX_SPEED) {
            throw std::out_of_range("Speed out of valid range");
        }
```

```
    }

    void sendMotorCommand(int motor_id, double speed) {
        // Gửi lệnh điều khiển tới motor
    }

    std::vector<double> readSensorValues() {
        // Đọc giá trị từ sensors
        return std::vector<double>();
    }

    void stopAllMotors() {
        // Dừng tất cả motors
    }

    void closeConnection() {
        // Đóng kết nối với hardware
    }

    static std::shared_ptr<RobotHardwareInterface> instance_;
    rclcpp::Logger logger_;
    std::mutex motor_mutex_;
    std::mutex sensor_mutex_;
    bool is_initialized_ = false;
    const int MAX_MOTORS = 8;
    const double MAX_SPEED = 100.0;
};

// Initialize static member
std::shared_ptr<RobotHardwareInterface> RobotHardwareInterface::instance_ = nullptr;

// Usage in ROS2 node
class RobotControlNode : public rclcpp::Node {
public:
    RobotControlNode() : Node("robot_control") {
        // Lấy instance của hardware interface
        hardware_ = RobotHardwareInterface::getInstance();

        // Khởi tạo hardware
        if (!hardware_->initializeHardware()) {
            throw std::runtime_error("Failed to initialize hardware");
        }

        // Tạo subscriber cho speed commands
        speed_sub_ = create_subscription<geometry_msgs::msg::Twist>(
            "cmd_vel", 10,
            std::bind(&RobotControlNode::speedCallback, this, std::placeholders::_1));

        // Tạo publisher cho sensor data
        sensor_pub_ = create_publisher<sensor_msgs::msg::JointState>(
            "joint_states", 10);

        // Timer để publish sensor data
        timer_ = create_wall_timer(
            std::chrono::milliseconds(100),
            std::bind(&RobotControlNode::publishSensorData, this));
    }

    ~RobotControlNode() {
        // Shutdown hardware khi node bị hủy
        hardware_->shutdown();
    }

private:
    void speedCallback(const geometry_msgs::msg::Twist::SharedPtr msg) {
        try {
            // Chuyển đổi twist message thành motor commands
            double left_speed = msg->linear.x - msg->angular.z;
            double right_speed = msg->linear.x + msg->angular.z;

            // Gửi lệnh tới motors thông qua hardware interface
            hardware_->setMotorSpeed(0, left_speed);   // Left motor
            hardware_->setMotorSpeed(1, right_speed);  // Right motor
        } catch (const std::exception& e) {
            RCLCPP_ERROR(get_logger(), "Speed control error: %s", e.what());
        }
    }

    void publishSensorData() {
```

```
        try {
            auto sensor_data = hardware_->getSensorData();

            // Tạo và publish sensor message
            auto msg = sensor_msgs::msg::JointState();
            msg.header.stamp = now();
            msg.position = sensor_data;
            sensor_pub_->publish(msg);
        } catch (const std::exception& e) {
            RCLCPP_ERROR(get_logger(), "Sensor publishing error: %s", e.what());
        }
    }

    std::shared_ptr<RobotHardwareInterface> hardware_;
    rclcpp::Subscription<geometry_msgs::msg::Twist>::SharedPtr speed_sub_;
    rclcpp::Publisher<sensor_msgs::msg::JointState>::SharedPtr sensor_pub_;
    rclcpp::TimerBase::SharedPtr timer_;
};

#### 4. Giải thích chi tiết cách hoạt động
1. **Khởi tạo an toàn**:
   - Constructor private ngăn tạo instance trực tiếp
   - Double-checked locking pattern tránh race condition
   - Lazy initialization tiết kiệm tài nguyên

2. **Quản lý truy cập**:
   - Thread-safe với mutexes
   - Kiểm tra điều kiện trước khi thực hiện operations
   - Exception handling đầy đủ

3. **Lifecycle management**:
   - Khởi tạo hardware khi cần
   - Cleanup resources khi shutdown
   - Trạng thái được theo dõi

#### 5. Ưu điểm trong ROS2
1. **Resource Management**:
   - Tránh xung đột truy cập hardware
   - Tiết kiệm tài nguyên hệ thống
   - Quản lý trạng thái tập trung

2. **Thread Safety**:
   - Đồng bộ hóa truy cập
   - Tránh race conditions
   - Xử lý concurrent requests

3. **Error Handling**:
   - Xử lý lỗi tập trung
   - Recovery mechanisms
   - Logging nhất quán

#### 6. Các trường hợp sử dụng trong ROS2
1. **Hardware Interfaces**:
   - Motor controllers
   - Sensor interfaces
   - Communication ports

2. **Global Resources**:
   - Configuration managers
   - Logging systems
   - Parameter servers

3. **Shared Services**:
   - Transform broadcasters
   - Navigation managers
   - State machines

#### 7. Best Practices trong ROS2
1. **Thread Safety**:
```cpp
class ThreadSafeSingleton {
    static std::mutex mutex_;
    static std::shared_ptr<ThreadSafeSingleton> instance_;
public:
    static std::shared_ptr<ThreadSafeSingleton> getInstance() {
        std::lock_guard<std::mutex> lock(mutex_);
        if (!instance_) {
            instance_ = std::shared_ptr<ThreadSafeSingleton>(
                new ThreadSafeSingleton());
```
```

```
        }
        return instance_;
    }
};
```

### 3. Ví dụ thực tế trong ROS2

2. **Resource Management**:

```
class ResourceManager {
    ~ResourceManager() {
        // Cleanup code
        if (is_initialized_) {
            releaseResources();
        }
    }

    void releaseResources() {
        std::lock_guard<std::mutex> lock(mutex_);
        // Release resources
        is_initialized_ = false;
    }
};
```

3. **Error Handling**:

```
class SafeSingleton {
public:
    void operation() {
        try {
            // Operation code
        } catch (const std::exception& e) {
            RCLCPP_ERROR(logger_, "Operation failed: %s", e.what());
            // Recovery code
        }
    }
};
```

### 8. Mở rộng và tùy chỉnh

1. **Configurable Singleton**:

```
class ConfigurableSingleton {
    static std::shared_ptr<ConfigurableSingleton> instance_;
    Config config_;
public:
    static void configure(const Config& config) {
        getInstance()->config_ = config;
    }
};
```

2. **Multi-instance Control**:

```
class ControlledSingleton {
    static std::map<std::string, std::shared_ptr<ControlledSingleton>> instances_;
public:
    static std::shared_ptr<ControlledSingleton> getInstance(const std::string& key) {
        // Return specific instance based on key
    }
};
```

3. **Testable Singleton**:

```
class TestableSingleton {
    static std::shared_ptr<TestableSingleton> instance_;
public:
    static void setInstance(std::shared_ptr<TestableSingleton> test_instance) {
        instance_ = test_instance;
    }
};
```

### 9. Testing

1. **Mock Objects**:

```
class MockHardwareInterface : public RobotHardwareInterface {
public:
```

```
    MOCK_METHOD(bool, initializeHardware, (), (override));
    MOCK_METHOD(bool, setMotorSpeed, (int, double), (override));
};
```

2. **Unit Tests**:

```
TEST(SingletonTest, SingleInstanceTest) {
    auto instance1 = RobotHardwareInterface::getInstance();
    auto instance2 = RobotHardwareInterface::getInstance();
    EXPECT_EQ(instance1, instance2);
}
```

3. **Integration Tests**:

```
TEST(HardwareTest, FullSystemTest) {
    auto hardware = RobotHardwareInterface::getInstance();
    EXPECT_TRUE(hardware->initializeHardware());

    // Test operations
    EXPECT_TRUE(hardware->setMotorSpeed(0, 50.0));
    auto sensor_data = hardware->getSensorData();
    EXPECT_FALSE(sensor_data.empty());

    hardware->shutdown();
}
```

**10. Kết luận** Singleton Pattern là một mẫu thiết kế quan trọng trong phát triển phần mềm robotics với ROS2, đặc biệt trong việc quản lý tài nguyên chia sẻ và hardware interfaces. Pattern này mang lại nhiều lợi ích thiết thực:

1. **Quản lý tài nguyên hiệu quả**:
   - Đảm bảo truy cập duy nhất vào hardware
   - Tránh xung đột và race conditions
   - Tối ưu hóa sử dụng tài nguyên hệ thống
2. **An toàn trong môi trường đa luồng**:
   - Thread-safe access
   - Đồng bộ hóa truy cập tài nguyên
   - Xử lý concurrent requests hiệu quả
3. **Dễ dàng bảo trì và mở rộng**:
   - Tập trung quản lý trạng thái
   - Dễ dàng thêm tính năng mới
   - Code sạch và có cấu trúc
4. **Phù hợp với robotics**:
   - Ideal cho hardware interfaces
   - Quản lý tài nguyên shared
   - Xử lý lỗi tập trung

Trong ví dụ về hardware interface, chúng ta đã thấy Singleton Pattern giúp xây dựng một hệ thống quản lý hardware an toàn và hiệu quả. Pattern này là lựa chọn tốt cho các hệ thống robotics cần quản lý tài nguyên tập trung và đảm bảo tính nhất quán của dữ liệu.

# Design Patterns trong ROS2

ROS2 Design Pattern Guide

2025-06-25

## ADAPTER PATTERN TRONG ROS2

**1. Giới thiệu đơn giản** Tưởng tượng bạn có một robot đang sử dụng cảm biến laser để đo khoảng cách, nhưng bây giờ bạn muốn thêm một cảm biến siêu âm mới. Vấn đề là:

- Cảm biến laser trả về khoảng cách theo mét
- Cảm biến siêu âm trả về khoảng cách theo inch
- Code hiện tại của robot chỉ làm việc với đơn vị mét

Adapter Pattern giống như một "bộ chuyển đổi" giúp cảm biến siêu âm có thể hoạt động với code hiện tại mà không cần thay đổi code của robot.

**2. Định nghĩa chi tiết** Adapter Pattern là một mẫu thiết kế cấu trúc cho phép các objects với interfaces không tương thích có thể làm việc cùng nhau. Pattern này hoạt động như một wrapper, chuyển đổi interface của một class thành interface khác mà client mong đợi.

**Các thành phần chính:**

1. **Target Interface**:
   - Interface mà client sử dụng
   - Trong ROS2: Interface chuẩn cho sensors
2. **Adaptee**:
   - Class cần được adapt
   - Trong ROS2: Sensor mới với interface khác
3. **Adapter**:
   - Class thực hiện việc chuyển đổi
   - Kết nối Target Interface với Adaptee

**3. Ví dụ thực tế trong ROS2** Giả sử chúng ta cần tích hợp một cảm biến siêu âm mới vào hệ thống robot hiện có:

```cpp
// 1. Target Interface - Interface chuẩn cho cảm biến khoảng cách
class DistanceSensorInterface {
public:
    virtual ~DistanceSensorInterface() = default;

    // Các phương thức chuẩn
    virtual void initialize() = 0;
    virtual double getDistanceInMeters() = 0;
    virtual std::string getSensorType() const = 0;
    virtual double getMaxRange() const = 0;
    virtual double getMinRange() const = 0;
    virtual double getFieldOfView() const = 0;  // radians
};

// 2. Existing Sensor - Cảm biến laser hiện có
class LaserSensor : public DistanceSensorInterface {
public:
    void initialize() override {
        RCLCPP_INFO(logger_, "Initializing Laser Sensor");
        // Khởi tạo hardware
    }

    double getDistanceInMeters() override {
        // Đọc dữ liệu từ laser
        return current_distance_;  // Đã ở đơn vị mét
    }
```

```
    std::string getSensorType() const override {
        return "LASER";
    }

    double getMaxRange() const override {
        return 30.0;  // 30 meters
    }

    double getMinRange() const override {
        return 0.1;  // 10 cm
    }

    double getFieldOfView() const override {
        return M_PI / 180.0 * 1.0;  // 1 degree
    }

private:
    double current_distance_ = 0.0;
    rclcpp::Logger logger_{rclcpp::get_logger("LaserSensor")};
};

// 3. Adaptee - Cảm biến siêu âm mới (Third-party/Legacy code)
class UltrasonicSensor {
public:
    // Interface khác với chuẩn
    bool connect() {
        // Kết nối với sensor
        return true;
    }

    float getRangeInInches() {
        // Đọc khoảng cách theo inch
        return current_range_;
    }

    bool isConnected() const {
        return connected_;
    }

    float getBeamWidth() const {
        return 15.0f;  // 15 degrees
    }

    float getMaxInches() const {
        return 157.48f;  // 4 meters in inches
    }

private:
    float current_range_ = 0.0f;
    bool connected_ = false;
};

// 4. Adapter - Chuyển đổi UltrasonicSensor sang DistanceSensorInterface
class UltrasonicAdapter : public DistanceSensorInterface {
public:
    explicit UltrasonicAdapter(std::shared_ptr<UltrasonicSensor> sensor)
        : ultrasonic_(sensor) {
        if (!ultrasonic_) {
            throw std::runtime_error("Null ultrasonic sensor provided");
        }
    }

    void initialize() override {
        RCLCPP_INFO(logger_, "Initializing Ultrasonic Sensor");
        if (!ultrasonic_->connect()) {
            throw std::runtime_error("Failed to connect to ultrasonic sensor");
        }
    }

    double getDistanceInMeters() override {
        if (!ultrasonic_->isConnected()) {
            throw std::runtime_error("Ultrasonic sensor not connected");
        }

        // Chuyển đổi từ inch sang mét
        return ultrasonic_->getRangeInInches() * 0.0254;
    }
```

```
    std::string getSensorType() const override {
        return "ULTRASONIC";
    }

    double getMaxRange() const override {
        // Chuyển đổi max range từ inch sang mét
        return ultrasonic_->getMaxInches() * 0.0254;
    }

    double getMinRange() const override {
        return 0.02;  // 2 cm
    }

    double getFieldOfView() const override {
        // Chuyển đổi từ độ sang radian
        return ultrasonic_->getBeamWidth() * M_PI / 180.0;
    }

private:
    std::shared_ptr<UltrasonicSensor> ultrasonic_;
    rclcpp::Logger logger_{rclcpp::get_logger("UltrasonicAdapter")};
};

// 5. Client code trong ROS2 node
class ObstacleDetectionNode : public rclcpp::Node {
public:
    ObstacleDetectionNode() : Node("obstacle_detection") {
        // Khởi tạo các sensors
        setupSensors();

        // Tạo publisher cho khoảng cách
        distance_pub_ = create_publisher<sensor_msgs::msg::Range>(
            "obstacle_distance", 10);

        // Timer để publish dữ liệu
        timer_ = create_wall_timer(
            std::chrono::milliseconds(100),
            std::bind(&ObstacleDetectionNode::publishDistances, this));
    }

private:
    void setupSensors() {
        try {
            // Khởi tạo laser sensor
            laser_sensor_ = std::make_shared<LaserSensor>();
            laser_sensor_->initialize();

            // Khởi tạo ultrasonic sensor với adapter
            auto ultrasonic = std::make_shared<UltrasonicSensor>();
            ultrasonic_sensor_ = std::make_shared<UltrasonicAdapter>(ultrasonic);
            ultrasonic_sensor_->initialize();

            RCLCPP_INFO(get_logger(), "All sensors initialized successfully");
        } catch (const std::exception& e) {
            RCLCPP_ERROR(get_logger(), "Error initializing sensors: %s", e.what());
            throw;
        }
    }

    void publishDistances() {
        try {
            // Publish dữ liệu từ cả hai sensor
            publishSensorData(laser_sensor_, "laser");
            publishSensorData(ultrasonic_sensor_, "ultrasonic");
        } catch (const std::exception& e) {
            RCLCPP_ERROR(get_logger(), "Error publishing distances: %s", e.what());
        }
    }

    void publishSensorData(
        const std::shared_ptr<DistanceSensorInterface>& sensor,
        const std::string& frame_id) {

        auto msg = sensor_msgs::msg::Range();
        msg.header.frame_id = frame_id;
        msg.header.stamp = now();
        msg.radiation_type =
            (sensor->getSensorType() == "LASER") ?
            sensor_msgs::msg::Range::INFRARED :
```

```
            sensor_msgs::msg::Range::ULTRASOUND;
        msg.field_of_view = sensor->getFieldOfView();
        msg.min_range = sensor->getMinRange();
        msg.max_range = sensor->getMaxRange();
        msg.range = sensor->getDistanceInMeters();

        distance_pub_->publish(msg);
    }

    std::shared_ptr<DistanceSensorInterface> laser_sensor_;
    std::shared_ptr<DistanceSensorInterface> ultrasonic_sensor_;
    rclcpp::Publisher<sensor_msgs::msg::Range>::SharedPtr distance_pub_;
    rclcpp::TimerBase::SharedPtr timer_;
};
```

### 4. Giải thích chi tiết cách hoạt động

1. **Khởi tạo hệ thống**:
   - Tạo interface chuẩn cho cảm biến khoảng cách
   - Laser sensor implement interface này trực tiếp
   - Ultrasonic sensor được wrap bởi adapter
2. **Chuyển đổi dữ liệu**:
   - Adapter chuyển đổi đơn vị từ inch sang mét
   - Chuyển đổi các thông số khác (FOV, ranges)
   - Xử lý lỗi và exceptions
3. **Publishing dữ liệu**:
   - Node sử dụng interface chung cho cả hai sensor
   - Publish dữ liệu theo định dạng ROS2 standard
   - Xử lý lỗi gracefully

### 5. Ưu điểm trong ROS2

1. **Tích hợp linh hoạt**:
   - Dễ dàng thêm sensors mới
   - Không cần sửa code hiện có
   - Tái sử dụng code tối đa
2. **Bảo trì dễ dàng**:
   - Tách biệt logic chuyển đổi
   - Dễ thay đổi/cập nhật adapter
   - Code sạch và có tổ chức
3. **Xử lý lỗi tốt**:
   - Kiểm tra null pointers
   - Xử lý connection errors
   - Exception handling

### 6. Các trường hợp sử dụng trong ROS2

1. **Hardware Integration**:
   - Tích hợp sensors mới
   - Sử dụng drivers cũ
   - Chuyển đổi data formats
2. **Message Adaptation**:
   - Chuyển đổi message types
   - Custom message bridges
   - Protocol adaptation
3. **Legacy System Integration**:
   - Tích hợp code cũ
   - Cập nhật interfaces
   - Backwards compatibility

### 7. Best Practices trong ROS2

1. **Error Handling**:

```
try {
    double distance = sensor->getDistanceInMeters();
```

```
    if (distance < 0) {
        throw std::runtime_error("Invalid distance reading");
    }
} catch (const std::exception& e) {
    RCLCPP_ERROR(logger, "Sensor error: %s", e.what());
    // Fallback behavior
}
```

2. **Resource Management**:

```
class SensorAdapter {
    std::unique_ptr<SensorDriver> driver_;
public:
    ~SensorAdapter() {
        if (driver_) {
            driver_->disconnect();
        }
    }
};
```

3. **Thread Safety**:

```
std::mutex sensor_mutex_;
double getDistance() {
    std::lock_guard<std::mutex> lock(sensor_mutex_);
    return sensor_->getDistanceInMeters();
}
```

### 8. Mở rộng và tùy chỉnh

1. **Two-Way Adapter**:

```
class BidirectionalAdapter : public NewInterface, public OldInterface {
    // Implement both interfaces
    // Convert in both directions
};
```

2. **Multiple Adaptation**:

```
class MultiSensorAdapter : public DistanceSensorInterface {
    std::vector<std::unique_ptr<BaseSensor>> sensors_;
public:
    double getDistanceInMeters() override {
        // Combine data from multiple sensors
    }
};
```

3. **Dynamic Adaptation**:

```
class ConfigurableAdapter : public DistanceSensorInterface {
    std::map<std::string, double> conversion_factors_;
public:
    void updateConversion(const std::string& unit, double factor) {
        conversion_factors_[unit] = factor;
    }
};
```

### 9. Testing

1. **Mock Objects**:

```
class MockUltrasonicSensor : public UltrasonicSensor {
public:
    MOCK_METHOD(float, getRangeInInches, (), (override));
    MOCK_METHOD(bool, connect, (), (override));
};
```

2. **Unit Tests**:

```
TEST(UltrasonicAdapterTest, ConversionTest) {
    auto mock_sensor = std::make_shared<MockUltrasonicSensor>();
    EXPECT_CALL(*mock_sensor, getRangeInInches())
        .WillOnce(Return(39.37f));  // 1 meter

    UltrasonicAdapter adapter(mock_sensor);
    EXPECT_NEAR(adapter.getDistanceInMeters(), 1.0, 0.001);
}
```

3. **Integration Tests**:

```
TEST(SensorSystemTest, FullSystemTest) {
    ObstacleDetectionNode node;

    // Test sensor initialization
    EXPECT_NO_THROW(node.setupSensors());

    // Test data publishing
    auto messages = std::make_shared<MessageCollector>();
    node.addSubscriber(messages);

    // Verify message contents
    EXPECT_TRUE(messages->waitForMessage(1s));
    auto msg = messages->getLastMessage();
    EXPECT_TRUE(msg.range > 0);
}
```

**10. Kết luận** Adapter Pattern là một mẫu thiết kế cấu trúc quan trọng trong phát triển phần mềm robotics với ROS2, đặc biệt trong việc tích hợp các thành phần không tương thích. Pattern này mang lại nhiều giá trị thực tiễn:

1. **Tích hợp linh hoạt**:
   - Cho phép sử dụng các sensors và thiết bị mới mà không cần sửa đổi code hiện có
   - Hỗ trợ việc chuyển đổi giữa các định dạng dữ liệu khác nhau
   - Tạo cầu nối giữa các hệ thống legacy và modern
2. **Tối ưu cho robotics**:
   - Dễ dàng thêm các sensors mới vào hệ thống
   - Chuyển đổi mượt mà giữa các đơn vị đo lường
   - Tương thích với nhiều loại hardware interfaces
3. **Bảo trì hiệu quả**:
   - Tách biệt logic chuyển đổi khỏi business logic
   - Dễ dàng cập nhật và thay đổi adapters
   - Code sạch và có cấu trúc rõ ràng
4. **Giá trị thực tế**:
   - Tiết kiệm thời gian và chi phí phát triển
   - Giảm thiểu rủi ro khi tích hợp components mới
   - Tăng khả năng tái sử dụng code

Trong ví dụ về việc tích hợp cảm biến siêu âm mới vào hệ thống robot, chúng ta đã thấy Adapter Pattern giúp giải quyết vấn đề không tương thích một cách thanh lịch và hiệu quả. Pattern này là công cụ thiết yếu cho các nhà phát triển ROS2 trong việc xây dựng hệ thống robotics linh hoạt và dễ mở rộng.

# Design Patterns trong ROS2

ROS2 Design Pattern Guide

2025-06-25

## BRIDGE PATTERN TRONG ROS2

**1. Giới thiệu đơn giản** Tưởng tượng bạn đang xây dựng một hệ thống điều khiển robot với nhiều loại robot khác nhau (mobile robot, robot arm, drone) và nhiều loại điều khiển khác nhau (position control, velocity control, force control). Thay vì phải viết code riêng cho mỗi kết hợp robot-controller, Bridge Pattern cho phép bạn:

- Tách riêng robot interface và controller interface
- Kết hợp linh hoạt bất kỳ robot nào với bất kỳ controller nào
- Thêm robot mới hoặc controller mới mà không ảnh hưởng đến code hiện có

**2. Định nghĩa chi tiết** Bridge Pattern là một mẫu thiết kế cấu trúc cho phép bạn tách một class lớn hoặc một tập các class có liên quan thành hai hệ thống phân cấp riêng biệt - abstraction và implementation - có thể phát triển độc lập với nhau.

**Các thành phần chính:**

1. **Abstraction**:
   - Interface cao cấp cho client sử dụng
   - Trong ROS2: Robot interface
2. **Implementation**:
   - Interface cho các concrete implementations
   - Trong ROS2: Controller interface
3. **Refined Abstraction**:
   - Mở rộng của abstraction
   - Trong ROS2: Specific robot types
4. **Concrete Implementation**:
   - Implementation cụ thể
   - Trong ROS2: Specific controllers

**3. Ví dụ thực tế trong ROS2** Giả sử chúng ta đang xây dựng một hệ thống điều khiển robot đa năng:

```cpp
// 1. Implementation Interface (Bridge)
class RobotController {
public:
    virtual ~RobotController() = default;

    // Các phương thức điều khiển cơ bản
    virtual void initialize() = 0;
    virtual bool setTarget(const geometry_msgs::msg::Pose& target) = 0;
    virtual bool executeMotion() = 0;
    virtual void stop() = 0;
    virtual std::string getControllerType() const = 0;

    // Getters cho trạng thái
    virtual geometry_msgs::msg::Pose getCurrentPose() const = 0;
    virtual bool isMoving() const = 0;
    virtual double getControlFrequency() const = 0;
};

// 2. Concrete Implementations

// 2.1 Position Controller
class PositionController : public RobotController {
public:
    void initialize() override {
        RCLCPP_INFO(logger_, "Initializing Position Controller");
        // Khởi tạo PID controllers
```

```
        setupPIDControllers();
    }

    bool setTarget(const geometry_msgs::msg::Pose& target) override {
        target_pose_ = target;
        return validateTarget(target);
    }

    bool executeMotion() override {
        if (!isTargetValid()) {
            RCLCPP_ERROR(logger_, "Invalid target pose");
            return false;
        }

        // Thực hiện position control
        return runPositionControl();
    }

    void stop() override {
        RCLCPP_INFO(logger_, "Stopping position controller");
        stopMotion();
    }

    std::string getControllerType() const override {
        return "POSITION_CONTROLLER";
    }

    geometry_msgs::msg::Pose getCurrentPose() const override {
        return current_pose_;
    }

    bool isMoving() const override {
        return is_moving_;
    }

    double getControlFrequency() const override {
        return 100.0;  // 100Hz
    }

private:
    void setupPIDControllers() {
        // Setup PID gains
    }

    bool validateTarget(const geometry_msgs::msg::Pose& target) {
        // Kiểm tra target có hợp lệ
        return true;
    }

    bool runPositionControl() {
        // Implement position control loop
        return true;
    }

    void stopMotion() {
        is_moving_ = false;
        // Dừng động cơ
    }

    geometry_msgs::msg::Pose target_pose_;
    geometry_msgs::msg::Pose current_pose_;
    bool is_moving_ = false;
    rclcpp::Logger logger_{rclcpp::get_logger("PositionController")};
};

// 2.2 Velocity Controller
class VelocityController : public RobotController {
public:
    void initialize() override {
        RCLCPP_INFO(logger_, "Initializing Velocity Controller");
        // Khởi tạo velocity control parameters
    }

    bool setTarget(const geometry_msgs::msg::Pose& target) override {
        // Chuyển đổi target pose thành velocity commands
        return calculateVelocityProfile(target);
    }

    bool executeMotion() override {
```

```
        // Thực hiện velocity control
        return runVelocityControl();
    }

    void stop() override {
        RCLCPP_INFO(logger_, "Stopping velocity controller");
        setZeroVelocity();
    }

    std::string getControllerType() const override {
        return "VELOCITY_CONTROLLER";
    }

    geometry_msgs::msg::Pose getCurrentPose() const override {
        return current_pose_;
    }

    bool isMoving() const override {
        return current_velocity_ > 0.001;
    }

    double getControlFrequency() const override {
        return 200.0;  // 200Hz
    }

private:
    bool calculateVelocityProfile(const geometry_msgs::msg::Pose& target) {
        // Tính toán velocity profile
        return true;
    }

    bool runVelocityControl() {
        // Implement velocity control loop
        return true;
    }

    void setZeroVelocity() {
        current_velocity_ = 0.0;
        // Dừng động cơ
    }

    geometry_msgs::msg::Pose current_pose_;
    double current_velocity_ = 0.0;
    rclcpp::Logger logger_{rclcpp::get_logger("VelocityController")};
};

// 3. Abstraction
class Robot {
public:
    explicit Robot(std::shared_ptr<RobotController> controller)
        : controller_(controller) {
        if (!controller_) {
            throw std::runtime_error("Null controller provided");
        }
    }

    virtual ~Robot() = default;

    // Common interface for all robots
    virtual void initialize() = 0;
    virtual bool moveToTarget(const geometry_msgs::msg::Pose& target) = 0;
    virtual void emergencyStop() = 0;
    virtual std::string getRobotType() const = 0;

    // Getters
    geometry_msgs::msg::Pose getCurrentPose() const {
        return controller_->getCurrentPose();
    }

    bool isMoving() const {
        return controller_->isMoving();
    }

    std::string getControllerType() const {
        return controller_->getControllerType();
    }

protected:
    std::shared_ptr<RobotController> controller_;
```

```
};

// 4. Refined Abstractions

// 4.1 Mobile Robot
class MobileRobot : public Robot {
public:
    explicit MobileRobot(std::shared_ptr<RobotController> controller)
        : Robot(controller) {}

    void initialize() override {
        RCLCPP_INFO(logger_, "Initializing Mobile Robot");
        controller_->initialize();
        setupSafetyLimits();
    }

    bool moveToTarget(const geometry_msgs::msg::Pose& target) override {
        if (!isSafeTarget(target)) {
            RCLCPP_ERROR(logger_, "Target pose outside safety limits");
            return false;
        }

        if (!controller_->setTarget(target)) {
            RCLCPP_ERROR(logger_, "Failed to set target");
            return false;
        }

        return controller_->executeMotion();
    }

    void emergencyStop() override {
        RCLCPP_WARN(logger_, "Emergency stop triggered for mobile robot");
        controller_->stop();
    }

    std::string getRobotType() const override {
        return "MOBILE_ROBOT";
    }

private:
    void setupSafetyLimits() {
        // Setup safety boundaries
    }

    bool isSafeTarget(const geometry_msgs::msg::Pose& target) {
        // Check if target is within safety limits
        return true;
    }

    rclcpp::Logger logger_{rclcpp::get_logger("MobileRobot")};
};

// 4.2 Robot Arm
class RobotArm : public Robot {
public:
    explicit RobotArm(std::shared_ptr<RobotController> controller)
        : Robot(controller) {}

    void initialize() override {
        RCLCPP_INFO(logger_, "Initializing Robot Arm");
        controller_->initialize();
        loadKinematicsModel();
    }

    bool moveToTarget(const geometry_msgs::msg::Pose& target) override {
        if (!isReachable(target)) {
            RCLCPP_ERROR(logger_, "Target pose not reachable");
            return false;
        }

        if (!controller_->setTarget(target)) {
            RCLCPP_ERROR(logger_, "Failed to set target");
            return false;
        }

        return controller_->executeMotion();
    }

    void emergencyStop() override {
```

```
        RCLCPP_WARN(logger_, "Emergency stop triggered for robot arm");
        controller_->stop();
        engageBrakes();
    }

    std::string getRobotType() const override {
        return "ROBOT_ARM";
    }

private:
    void loadKinematicsModel() {
        // Load robot arm kinematics
    }

    bool isReachable(const geometry_msgs::msg::Pose& target) {
        // Check if target is within workspace
        return true;
    }

    void engageBrakes() {
        // Engage mechanical brakes
    }

    rclcpp::Logger logger_{rclcpp::get_logger("RobotArm")};
};

// 5. Client code trong ROS2 node
class RobotControlNode : public rclcpp::Node {
public:
    RobotControlNode() : Node("robot_control") {
        // Khởi tạo robot và controller
        setupRobotSystem();

        // Tạo subscribers và services
        target_sub_ = create_subscription<geometry_msgs::msg::PoseStamped>(
            "target_pose", 10,
            std::bind(&RobotControlNode::targetCallback, this, std::placeholders::_1));

        stop_service_ = create_service<std_srvs::srv::Trigger>(
            "emergency_stop",
            std::bind(&RobotControlNode::stopCallback, this,
                      std::placeholders::_1, std::placeholders::_2));
    }

private:
    void setupRobotSystem() {
        try {
            // Đọc parameters
            std::string robot_type = get_parameter("robot_type").as_string();
            std::string controller_type = get_parameter("controller_type").as_string();

            // Tạo controller phù hợp
            std::shared_ptr<RobotController> controller;
            if (controller_type == "position") {
                controller = std::make_shared<PositionController>();
            } else if (controller_type == "velocity") {
                controller = std::make_shared<VelocityController>();
            } else {
                throw std::runtime_error("Unknown controller type");
            }

            // Tạo robot phù hợp
            if (robot_type == "mobile") {
                robot_ = std::make_unique<MobileRobot>(controller);
            } else if (robot_type == "arm") {
                robot_ = std::make_unique<RobotArm>(controller);
            } else {
                throw std::runtime_error("Unknown robot type");
            }

            // Khởi tạo hệ thống
            robot_->initialize();
            RCLCPP_INFO(get_logger(),
                "Initialized %s with %s",
                robot_->getRobotType().c_str(),
                robot_->getControllerType().c_str());

        } catch (const std::exception& e) {
            RCLCPP_ERROR(get_logger(), "Failed to setup robot: %s", e.what());
```

```
            throw;
        }
    }

    void targetCallback(const geometry_msgs::msg::PoseStamped::SharedPtr msg) {
        try {
            if (robot_->isMoving()) {
                RCLCPP_WARN(get_logger(), "Robot is already moving, ignoring target");
                return;
            }

            if (robot_->moveToTarget(msg->pose)) {
                RCLCPP_INFO(get_logger(), "Robot moving to target");
            } else {
                RCLCPP_ERROR(get_logger(), "Failed to move to target");
            }
        } catch (const std::exception& e) {
            RCLCPP_ERROR(get_logger(), "Error in target callback: %s", e.what());
        }
    }

    void stopCallback(
        const std::shared_ptr<std_srvs::srv::Trigger::Request>,
        std::shared_ptr<std_srvs::srv::Trigger::Response> response) {

        try {
            robot_->emergencyStop();
            response->success = true;
            response->message = "Emergency stop executed";
        } catch (const std::exception& e) {
            response->success = false;
            response->message = std::string("Failed to stop: ") + e.what();
        }
    }

    std::unique_ptr<Robot> robot_;
    rclcpp::Subscription<geometry_msgs::msg::PoseStamped>::SharedPtr target_sub_;
    rclcpp::Service<std_srvs::srv::Trigger>::SharedPtr stop_service_;
};
```

### 4. Giải thích chi tiết cách hoạt động

1. **Phân tách interface**:
   - Robot interface (Abstraction)
   - Controller interface (Implementation)
   - Hai hệ thống phát triển độc lập
2. **Kết hợp linh hoạt**:
   - Bất kỳ robot nào với bất kỳ controller nào
   - Runtime configuration
   - Dễ dàng mở rộng
3. **Xử lý lỗi**:
   - Kiểm tra null pointers
   - Validation input
   - Exception handling

### 5. Ưu điểm trong ROS2

1. **Tính mở rộng**:
   - Thêm robot mới dễ dàng
   - Thêm controller mới độc lập
   - Không ảnh hưởng code hiện có
2. **Tái sử dụng code**:
   - Controllers dùng chung cho nhiều robots
   - Common functionality trong base classes
   - Reduced code duplication
3. **Dễ bảo trì**:
   - Các thành phần độc lập
   - Dễ test riêng từng phần
   - Clear responsibilities

## 6. Các trường hợp sử dụng trong ROS2

1. **Robot Control**:
   - Different robot types
   - Multiple control strategies
   - Hardware abstraction
2. **Sensor Systems**:
   - Different sensor types
   - Processing algorithms
   - Data fusion
3. **Navigation**:
   - Path planning algorithms
   - Local planners
   - Global planners

## 7. Best Practices trong ROS2

1. **Error Handling**:

```
try {
    if (!robot->initialize()) {
        throw std::runtime_error("Robot initialization failed");
    }
} catch (const std::exception& e) {
    RCLCPP_ERROR(logger, "Error: %s", e.what());
    // Fallback behavior
}
```

2. **Resource Management**:

```
class SafeRobot {
    std::unique_ptr<Robot> robot_;
public:
    ~SafeRobot() {
        if (robot_) {
            robot_->emergencyStop();
        }
    }
};
```

3. **Parameter Management**:

```
void loadParameters() {
    auto params = node_->get_parameters({
        "robot_type",
        "controller_type",
        "safety_limits"
    });
    // Process parameters
}
```

## 8. Mở rộng và tùy chỉnh

1. **New Robot Type**:

```
class DroneRobot : public Robot {
public:
    void initialize() override {
        // Drone specific initialization
    }

    bool moveToTarget(const geometry_msgs::msg::Pose& target) override {
        // Drone specific movement
    }
};
```

2. **New Controller Type**:

```
class ForceController : public RobotController {
public:
    bool executeMotion() override {
        // Force control implementation
    }
};
```

3. **Combined Controllers**:

```
class HybridController : public RobotController {
    std::vector<std::shared_ptr<RobotController>> controllers_;
public:
    bool executeMotion() override {
        // Choose best controller based on situation
    }
};
```

## 9. Testing

1. **Mock Objects**:

```
class MockController : public RobotController {
public:
    MOCK_METHOD(void, initialize, (), (override));
    MOCK_METHOD(bool, executeMotion, (), (override));
};
```

2. **Unit Tests**:

```
TEST(RobotTest, InitializationTest) {
    auto mock_controller = std::make_shared<MockController>();
    EXPECT_CALL(*mock_controller, initialize())
        .WillOnce(Return(true));

    MobileRobot robot(mock_controller);
    EXPECT_NO_THROW(robot.initialize());
}
```

3. **Integration Tests**:

```
TEST(RobotSystemTest, FullSystemTest) {
    auto node = std::make_shared<rclcpp::Node>("test_node");
    RobotControlNode control_node;

    // Test robot movement
    geometry_msgs::msg::Pose target;
    EXPECT_TRUE(control_node.moveToTarget(target));

    // Test emergency stop
    EXPECT_NO_THROW(control_node.emergencyStop());
}
```

**10. Kết luận** Bridge Pattern là một mẫu thiết kế cấu trúc mạnh mẽ trong phát triển phần mềm robotics với ROS2, đặc biệt trong việc tách biệt abstraction và implementation. Pattern này mang lại nhiều lợi thế quan trọng:

1. **Thiết kế linh hoạt**:
   - Tách biệt rõ ràng giữa interface và implementation
   - Cho phép thay đổi độc lập giữa robot types và controllers
   - Dễ dàng mở rộng cả về chiều rộng (thêm robots) và chiều sâu (thêm controllers)
2. **Tối ưu cho robotics**:
   - Phù hợp với các hệ thống điều khiển robot phức tạp
   - Hỗ trợ đa dạng các loại robot và chiến lược điều khiển
   - Dễ dàng chuyển đổi giữa các modes điều khiển
3. **Quản lý code hiệu quả**:
   - Cấu trúc code rõ ràng và có tổ chức
   - Giảm thiểu code trùng lặp
   - Dễ dàng maintain và update từng phần độc lập
4. **Giá trị thực tiễn**:
   - Tăng tính tái sử dụng của code
   - Giảm thời gian phát triển cho các robot mới
   - Tăng độ tin cậy của hệ thống

Trong ví dụ về hệ thống điều khiển robot, chúng ta đã thấy Bridge Pattern giúp xây dựng một kiến trúc linh hoạt, cho phép kết hợp bất kỳ loại robot nào với bất kỳ loại controller nào. Pattern này là lựa chọn xuất sắc cho các dự án robotics cần sự linh hoạt cao và khả năng mở rộng dễ dàng.

# Design Patterns trong ROS2

ROS2 Design Pattern Guide

2025-06-25

## OBSERVER PATTERN TRONG ROS2

**1. Giới thiệu đơn giản** Tưởng tượng bạn có một robot với nhiều cảm biến nhiệt độ ở các bộ phận khác nhau (động cơ, pin, bo mạch). Khi nhiệt độ của bất kỳ bộ phận nào vượt ngưỡng:

- Hệ thống cần gửi cảnh báo
- Giảm tốc độ robot
- Ghi log để phân tích sau

Observer Pattern giống như một "hệ thống theo dõi" tự động thông báo cho tất cả các thành phần quan tâm khi có thay đổi, thay vì phải liên tục kiểm tra.

**2. Định nghĩa chi tiết** Observer Pattern là một mẫu thiết kế hành vi cho phép bạn định nghĩa cơ chế đăng ký (subscription) để thông báo cho nhiều đối tượng về bất kỳ sự kiện nào xảy ra với đối tượng mà họ đang quan sát.

**Các thành phần chính:**

1. **Subject (Observable)**:
   - Đối tượng chứa trạng thái cần theo dõi
   - Trong ROS2: Temperature monitor
2. **Observer**:
   - Interface cho các observers
   - Trong ROS2: Temperature handlers
3. **Concrete Observers**:
   - Các observers cụ thể
   - Trong ROS2: Alert, Speed Control, Logger

**3. Ví dụ thực tế trong ROS2** Giả sử chúng ta cần xây dựng một hệ thống giám sát nhiệt độ cho robot:

```cpp
// 1. Observer Interface
class TemperatureObserver {
public:
    virtual ~TemperatureObserver() = default;

    virtual void update(const std::string& sensor_id,
                        double temperature,
                        double threshold) = 0;

    virtual std::string getObserverType() const = 0;
};

// 2. Subject Interface
class TemperatureSubject {
public:
    virtual ~TemperatureSubject() = default;

    virtual void attach(std::shared_ptr<TemperatureObserver> observer) = 0;
    virtual void detach(std::shared_ptr<TemperatureObserver> observer) = 0;
    virtual void notify(const std::string& sensor_id,
                        double temperature,
                        double threshold) = 0;
};

// 3. Concrete Subject
class TemperatureMonitor : public TemperatureSubject,
                           public rclcpp::Node {
public:
    explicit TemperatureMonitor()
```

```
        : Node("temperature_monitor") {
        // Khởi tạo parameters
        initializeParameters();

        // Tạo subscribers cho các sensors
        createSensorSubscribers();

        // Timer để kiểm tra nhiệt độ định kỳ
        timer_ = create_wall_timer(
            std::chrono::milliseconds(100),
            std::bind(&TemperatureMonitor::checkTemperatures, this));

        RCLCPP_INFO(get_logger(), "Temperature Monitor initialized");
    }

    void attach(std::shared_ptr<TemperatureObserver> observer) override {
        std::lock_guard<std::mutex> lock(observers_mutex_);
        observers_.push_back(observer);
        RCLCPP_INFO(get_logger(),
            "Attached observer: %s",
            observer->getObserverType().c_str());
    }

    void detach(std::shared_ptr<TemperatureObserver> observer) override {
        std::lock_guard<std::mutex> lock(observers_mutex_);
        observers_.erase(
            std::remove_if(
                observers_.begin(),
                observers_.end(),
                [&](const auto& obs) {
                    return obs->getObserverType() == observer->getObserverType();
                }),
            observers_.end());
    }

    void notify(const std::string& sensor_id,
                double temperature,
                double threshold) override {
        std::lock_guard<std::mutex> lock(observers_mutex_);
        for (const auto& observer : observers_) {
            try {
                observer->update(sensor_id, temperature, threshold);
            } catch (const std::exception& e) {
                RCLCPP_ERROR(get_logger(),
                    "Error notifying observer %s: %s",
                    observer->getObserverType().c_str(),
                    e.what());
            }
        }
    }

private:
    void initializeParameters() {
        // Khai báo parameters
        declare_parameter("motor_temp_threshold", 80.0);
        declare_parameter("battery_temp_threshold", 60.0);
        declare_parameter("electronics_temp_threshold", 70.0);

        // Load thresholds
        thresholds_["motor"] =
            get_parameter("motor_temp_threshold").as_double();
        thresholds_["battery"] =
            get_parameter("battery_temp_threshold").as_double();
        thresholds_["electronics"] =
            get_parameter("electronics_temp_threshold").as_double();
    }

    void createSensorSubscribers() {
        // Tạo subscribers cho từng sensor
        motor_temp_sub_ = create_subscription<sensor_msgs::msg::Temperature>(
            "motor_temperature", 10,
            [this](const sensor_msgs::msg::Temperature::SharedPtr msg) {
                temperatures_["motor"] = msg->temperature;
            });

        battery_temp_sub_ = create_subscription<sensor_msgs::msg::Temperature>(
            "battery_temperature", 10,
            [this](const sensor_msgs::msg::Temperature::SharedPtr msg) {
                temperatures_["battery"] = msg->temperature;
```

```
            });

        electronics_temp_sub_ = create_subscription<sensor_msgs::msg::Temperature>(
            "electronics_temperature", 10,
            [this](const sensor_msgs::msg::Temperature::SharedPtr msg) {
                temperatures_["electronics"] = msg->temperature;
            });
    }

    void checkTemperatures() {
        for (const auto& [sensor_id, temperature] : temperatures_) {
            auto threshold = thresholds_[sensor_id];
            if (temperature > threshold) {
                notify(sensor_id, temperature, threshold);
            }
        }
    }

    std::vector<std::shared_ptr<TemperatureObserver>> observers_;
    std::mutex observers_mutex_;
    std::map<std::string, double> temperatures_;
    std::map<std::string, double> thresholds_;
    rclcpp::TimerBase::SharedPtr timer_;
    rclcpp::Subscription<sensor_msgs::msg::Temperature>::SharedPtr
        motor_temp_sub_;
    rclcpp::Subscription<sensor_msgs::msg::Temperature>::SharedPtr
        battery_temp_sub_;
    rclcpp::Subscription<sensor_msgs::msg::Temperature>::SharedPtr
        electronics_temp_sub_;
};

// 4. Concrete Observers

// 4.1 Alert Observer
class AlertObserver : public TemperatureObserver,
                      public rclcpp::Node {
public:
    AlertObserver()
        : Node("temperature_alert") {
        // Publisher cho alerts
        alert_pub_ = create_publisher<std_msgs::msg::String>(
            "temperature_alerts", 10);
    }

    void update(const std::string& sensor_id,
                double temperature,
                double threshold) override {
        auto msg = std_msgs::msg::String();
        msg.data = fmt::format(
            "WARNING: {} temperature ({:.1f}°C) exceeded threshold ({:.1f}°C)",
            sensor_id, temperature, threshold);

        alert_pub_->publish(msg);
        RCLCPP_WARN(get_logger(), "%s", msg.data.c_str());
    }

    std::string getObserverType() const override {
        return "ALERT_OBSERVER";
    }

private:
    rclcpp::Publisher<std_msgs::msg::String>::SharedPtr alert_pub_;
};

// 4.2 Speed Control Observer
class SpeedControlObserver : public TemperatureObserver,
                             public rclcpp::Node {
public:
    SpeedControlObserver()
        : Node("temperature_speed_control") {
        // Publisher cho speed commands
        speed_pub_ = create_publisher<geometry_msgs::msg::Twist>(
            "cmd_vel", 10);
    }

    void update(const std::string& sensor_id,
                double temperature,
                double threshold) override {
        // Tính toán speed reduction factor
```

```
        double temp_diff = temperature - threshold;
        double reduction_factor = std::max(0.0,
            1.0 - (temp_diff / threshold) * 0.5);

        // Publish reduced speed command
        auto cmd = geometry_msgs::msg::Twist();
        cmd.linear.x = max_speed_ * reduction_factor;
        speed_pub_->publish(cmd);

        RCLCPP_INFO(get_logger(),
            "Reducing speed to %.1f%% due to high %s temperature",
            reduction_factor * 100.0, sensor_id.c_str());
    }

    std::string getObserverType() const override {
        return "SPEED_CONTROL_OBSERVER";
    }

private:
    rclcpp::Publisher<geometry_msgs::msg::Twist>::SharedPtr speed_pub_;
    const double max_speed_ = 1.0;  // m/s
};

// 4.3 Logger Observer
class LoggerObserver : public TemperatureObserver,
                       public rclcpp::Node {
public:
    LoggerObserver()
        : Node("temperature_logger") {
        // Tạo log file
        std::string log_dir = "temperature_logs";
        if (!std::filesystem::exists(log_dir)) {
            std::filesystem::create_directory(log_dir);
        }

        auto timestamp =
            std::chrono::system_clock::now().time_since_epoch().count();
        log_file_.open(
            fmt::format("{}/temp_log_{}.csv", log_dir, timestamp));

        // Write header
        log_file_ << "timestamp,sensor_id,temperature,threshold\n";
    }

    ~LoggerObserver() {
        if (log_file_.is_open()) {
            log_file_.close();
        }
    }

    void update(const std::string& sensor_id,
                double temperature,
                double threshold) override {
        if (!log_file_.is_open()) {
            RCLCPP_ERROR(get_logger(), "Log file not open");
            return;
        }

        auto now = std::chrono::system_clock::now();
        auto timestamp = std::chrono::duration_cast<std::chrono::milliseconds>(
            now.time_since_epoch()).count();

        log_file_ << fmt::format("{},{},{:.1f},{:.1f}\n",
            timestamp, sensor_id, temperature, threshold);
        log_file_.flush();

        RCLCPP_DEBUG(get_logger(),
            "Logged temperature data for %s", sensor_id.c_str());
    }

    std::string getObserverType() const override {
        return "LOGGER_OBSERVER";
    }

private:
    std::ofstream log_file_;
};

// 5. Main node để khởi tạo và kết nối các components
```

```cpp
class TemperatureMonitoringNode : public rclcpp::Node {
public:
    TemperatureMonitoringNode()
        : Node("temperature_monitoring") {
        // Khởi tạo monitor
        monitor_ = std::make_shared<TemperatureMonitor>();

        // Khởi tạo và đăng ký các observers
        auto alert_observer = std::make_shared<AlertObserver>();
        auto speed_observer = std::make_shared<SpeedControlObserver>();
        auto logger_observer = std::make_shared<LoggerObserver>();

        monitor_->attach(alert_observer);
        monitor_->attach(speed_observer);
        monitor_->attach(logger_observer);

        RCLCPP_INFO(get_logger(), "Temperature monitoring system initialized");
    }

private:
    std::shared_ptr<TemperatureMonitor> monitor_;
};

// 6. Main function
int main(int argc, char* argv[]) {
    rclcpp::init(argc, argv);

    // Tạo executor để chạy nhiều nodes
    rclcpp::executors::MultiThreadedExecutor executor;

    // Khởi tạo node
    auto node = std::make_shared<TemperatureMonitoringNode>();

    // Thêm node vào executor
    executor.add_node(node);

    // Spin
    executor.spin();

    rclcpp::shutdown();
    return 0;
}
```

### 4. Giải thích chi tiết cách hoạt động

1. **Khởi tạo hệ thống**:
   - Tạo temperature monitor
   - Đăng ký các observers
   - Khởi tạo ROS2 subscribers
2. **Monitoring loop**:
   - Đọc nhiệt độ từ các sensors
   - So sánh với ngưỡng
   - Notify observers khi vượt ngưỡng
3. **Observer actions**:
   - Alert: Publish cảnh báo
   - Speed Control: Giảm tốc độ
   - Logger: Ghi log file

### 5. Ưu điểm trong ROS2

1. **Loose Coupling**:
   - Monitor không biết về observers
   - Dễ thêm/xóa observers
   - Không ảnh hưởng code hiện có
2. **Real-time Response**:
   - Phản ứng ngay khi có thay đổi
   - Không cần polling
   - Giảm latency
3. **Extensibility**:
   - Dễ thêm sensors mới
   - Dễ thêm observers mới

- Tái sử dụng code tốt

### 6. Các trường hợp sử dụng trong ROS2

1. **Sensor Monitoring**:
   - Temperature monitoring
   - Battery monitoring
   - Error detection
2. **Robot State**:
   - Position tracking
   - Velocity monitoring
   - Load monitoring
3. **System Events**:
   - Emergency stops
   - Collision detection
   - Safety violations

### 7. Best Practices trong ROS2

1. **Thread Safety**:

```
class SafeObserver : public TemperatureObserver {
    std::mutex update_mutex_;
public:
    void update(const std::string& sensor_id,
                double temperature,
                double threshold) override {
        std::lock_guard<std::mutex> lock(update_mutex_);
        // Handle update
    }
};
```

2. **Error Handling**:

```
void notify(const std::string& sensor_id,
            double temperature,
            double threshold) override {
    try {
        for (const auto& observer : observers_) {
            observer->update(sensor_id, temperature, threshold);
        }
    } catch (const std::exception& e) {
        RCLCPP_ERROR(logger_, "Notification error: %s", e.what());
    }
}
```

3. **Resource Management**:

```
class AutoCleanupObserver : public TemperatureObserver {
    std::unique_ptr<std::ofstream> log_file_;
public:
    ~AutoCleanupObserver() {
        if (log_file_ && log_file_->is_open()) {
            log_file_->close();
        }
    }
};
```

### 8. Mở rộng và tùy chỉnh

1. **Priority Observers**:

```
class PriorityObserver : public TemperatureObserver {
    int priority_;
public:
    explicit PriorityObserver(int priority) : priority_(priority) {}
    int getPriority() const { return priority_; }
};
```

2. **Filtered Notifications**:

```
class FilteredObserver : public TemperatureObserver {
    double min_threshold_;
public:
```

```
    void update(const std::string& sensor_id,
                double temperature,
                double threshold) override {
        if (temperature > min_threshold_) {
            // Process update
        }
    }
};
```

3. **Composite Observers**:

```
class CompositeObserver : public TemperatureObserver {
    std::vector<std::shared_ptr<TemperatureObserver>> children_;
public:
    void update(const std::string& sensor_id,
                double temperature,
                double threshold) override {
        for (const auto& child : children_) {
            child->update(sensor_id, temperature, threshold);
        }
    }
};
```

## 9. Testing

1. **Mock Objects**:

```
class MockObserver : public TemperatureObserver {
public:
    MOCK_METHOD(void, update,
        (const std::string&, double, double), (override));
    MOCK_METHOD(std::string, getObserverType, (), (const, override));
};
```

2. **Unit Tests**:

```
TEST(TemperatureMonitorTest, NotificationTest) {
    auto monitor = std::make_shared<TemperatureMonitor>();
    auto mock_observer = std::make_shared<MockObserver>();

    EXPECT_CALL(*mock_observer, update("motor", 90.0, 80.0))
        .Times(1);

    monitor->attach(mock_observer);
    monitor->notify("motor", 90.0, 80.0);
}
```

3. **Integration Tests**:

```
TEST(TemperatureSystemTest, FullSystemTest) {
    rclcpp::init(0, nullptr);
    auto node = std::make_shared<TemperatureMonitoringNode>();

    // Publish test temperature data
    auto temp_pub = node->create_publisher<sensor_msgs::msg::Temperature>(
        "motor_temperature", 10);

    auto msg = sensor_msgs::msg::Temperature();
    msg.temperature = 85.0;
    temp_pub->publish(msg);

    // Verify alerts and speed changes
    // ...

    rclcpp::shutdown();
}
```

**10. Kết luận** Observer Pattern là một mẫu thiết kế quan trọng trong ROS2 và robotics, đặc biệt trong các hệ thống theo dõi và phản ứng thời gian thực. Pattern này mang lại nhiều lợi ích thiết thực:

1. **Tính linh hoạt cao**:
   - Dễ dàng thêm/xóa các observers mới mà không ảnh hưởng đến code hiện có
   - Có thể tái sử dụng observers cho nhiều subjects khác nhau
   - Hỗ trợ tốt việc mở rộng hệ thống
2. **Phù hợp với robotics**:

    - Xử lý tốt các tình huống real-time monitoring
    - Tích hợp tự nhiên với hệ thống publish/subscribe của ROS2
    - Đáp ứng nhanh với các thay đổi trạng thái của robot
3. **Dễ bảo trì và test**:
    - Code được tổ chức rõ ràng, dễ hiểu
    - Có thể test riêng từng observer
    - Dễ dàng debug và xử lý lỗi
4. **Hiệu quả trong thực tế**:
    - Giảm thiểu việc polling không cần thiết
    - Tối ưu hóa tài nguyên hệ thống
    - Xử lý đồng thời nhiều observers một cách hiệu quả

Trong ví dụ về hệ thống giám sát nhiệt độ, chúng ta đã thấy Observer Pattern giúp xây dựng một hệ thống mạnh mẽ, có khả năng mở rộng và dễ bảo trì. Pattern này là lựa chọn tốt cho các hệ thống robotics cần theo dõi và phản ứng với các thay đổi trạng thái một cách real-time.

# Design Patterns trong ROS2

ROS2 Design Pattern Guide

2025-06-25

## STRATEGY PATTERN TRONG ROS2

**1. Giới thiệu đơn giản** Strategy Pattern cho phép định nghĩa một nhóm các thuật toán, đóng gói từng thuật toán và làm cho chúng có thể hoán đổi cho nhau. Trong ROS2, pattern này đặc biệt hữu ích cho:

- Các thuật toán planning khác nhau
- Các chiến lược điều khiển robot
- Các phương pháp xử lý sensor data
- Các chiến lược navigation

**2. Định nghĩa chi tiết** Strategy Pattern cho phép thay đổi thuật toán độc lập với code sử dụng thuật toán đó. Pattern này bao gồm:

**Các thành phần chính:**

1. **Strategy Interface**:
   - Định nghĩa interface chung
   - Các phương thức chuẩn
2. **Concrete Strategies**:
   - Implements các thuật toán cụ thể
   - Tuân theo strategy interface
3. **Context**:
   - Sử dụng strategy
   - Có thể thay đổi strategy runtime

```cpp
// 1. Strategy Interface
class NavigationStrategy {
public:
    virtual ~NavigationStrategy() = default;

    virtual bool initialize(const rclcpp::Node::SharedPtr& node) = 0;
    virtual geometry_msgs::msg::Twist computeVelocity(
        const geometry_msgs::msg::PoseStamped& current_pose,
        const geometry_msgs::msg::PoseStamped& goal_pose,
        const nav_msgs::msg::OccupancyGrid& map) = 0;
    virtual bool isGoalReached(
        const geometry_msgs::msg::PoseStamped& current_pose,
        const geometry_msgs::msg::PoseStamped& goal_pose) = 0;
    virtual std::string getName() const = 0;
};

// 2. Concrete Strategies
class DWAPlanner : public NavigationStrategy {
public:
    bool initialize(const rclcpp::Node::SharedPtr& node) override {
        try {
            node_ = node;

            // Load DWA parameters
            auto params = node_->get_parameters("dwa");
            max_vel_x_ = params[0].as_double();
            max_vel_theta_ = params[1].as_double();
            acc_lim_x_ = params[2].as_double();
            acc_lim_theta_ = params[3].as_double();

            initialized_ = true;
            RCLCPP_INFO(node_->get_logger(), "DWA Planner initialized");
```

```
            return true;
        } catch (const std::exception& e) {
            RCLCPP_ERROR(node_->get_logger(),
                "Failed to initialize DWA Planner: %s", e.what());
            return false;
        }
    }

    geometry_msgs::msg::Twist computeVelocity(
        const geometry_msgs::msg::PoseStamped& current_pose,
        const geometry_msgs::msg::PoseStamped& goal_pose,
        const nav_msgs::msg::OccupancyGrid& map) override {

        if (!initialized_) {
            RCLCPP_ERROR(node_->get_logger(), "DWA Planner not initialized");
            return geometry_msgs::msg::Twist();
        }

        try {
            // 1. Generate velocity samples
            auto samples = generateVelocitySamples();

            // 2. Score each trajectory
            auto scored_trajectories = scoreTrajectories(
                samples, current_pose, goal_pose, map);

            // 3. Select best trajectory
            auto best_trajectory = selectBestTrajectory(scored_trajectories);

            // 4. Return velocity command
            return trajectoryToTwist(best_trajectory);
        } catch (const std::exception& e) {
            RCLCPP_ERROR(node_->get_logger(),
                "Error computing velocity: %s", e.what());
            return geometry_msgs::msg::Twist();
        }
    }

    bool isGoalReached(
        const geometry_msgs::msg::PoseStamped& current_pose,
        const geometry_msgs::msg::PoseStamped& goal_pose) override {

        // Calculate distance to goal
        double dx = goal_pose.pose.position.x - current_pose.pose.position.x;
        double dy = goal_pose.pose.position.y - current_pose.pose.position.y;
        double distance = std::sqrt(dx*dx + dy*dy);

        // Calculate angle difference
        double angle_diff = calculateAngleDifference(
            current_pose.pose.orientation,
            goal_pose.pose.orientation);

        return distance < position_tolerance_ &&
               std::abs(angle_diff) < angle_tolerance_;
    }

    std::string getName() const override {
        return "DWA_PLANNER";
    }

private:
    std::vector<Trajectory> generateVelocitySamples() {
        std::vector<Trajectory> samples;

        // Generate velocity samples within constraints
        for (double vx = -max_vel_x_; vx <= max_vel_x_; vx += vel_x_step_) {
            for (double vth = -max_vel_theta_; vth <= max_vel_theta_;
                 vth += vel_theta_step_) {
                // Create trajectory from velocity pair
                samples.push_back(createTrajectory(vx, vth));
            }
        }

        return samples;
    }

    std::vector<ScoredTrajectory> scoreTrajectories(
        const std::vector<Trajectory>& trajectories,
        const geometry_msgs::msg::PoseStamped& current_pose,
```

```
        const geometry_msgs::msg::PoseStamped& goal_pose,
        const nav_msgs::msg::OccupancyGrid& map) {

        std::vector<ScoredTrajectory> scored_trajectories;

        for (const auto& trajectory : trajectories) {
            double obstacle_cost = calculateObstacleCost(trajectory, map);
            double goal_cost = calculateGoalCost(trajectory, goal_pose);
            double path_cost = calculatePathCost(trajectory);

            double total_cost = obstacle_weight_ * obstacle_cost +
                                goal_weight_ * goal_cost +
                                path_weight_ * path_cost;

            scored_trajectories.push_back({trajectory, total_cost});
        }

        return scored_trajectories;
    }

    Trajectory selectBestTrajectory(
        const std::vector<ScoredTrajectory>& scored_trajectories) {

        auto best = std::min_element(
            scored_trajectories.begin(),
            scored_trajectories.end(),
            [](const auto& a, const auto& b) {
                return a.cost < b.cost;
            });

        return best->trajectory;
    }

    geometry_msgs::msg::Twist trajectoryToTwist(
        const Trajectory& trajectory) {

        geometry_msgs::msg::Twist cmd_vel;
        cmd_vel.linear.x = trajectory.vx;
        cmd_vel.angular.z = trajectory.vth;
        return cmd_vel;
    }

    rclcpp::Node::SharedPtr node_;
    bool initialized_ = false;

    // DWA parameters
    double max_vel_x_;
    double max_vel_theta_;
    double acc_lim_x_;
    double acc_lim_theta_;
    double vel_x_step_ = 0.1;
    double vel_theta_step_ = 0.1;

    // Cost weights
    double obstacle_weight_ = 0.8;
    double goal_weight_ = 0.6;
    double path_weight_ = 0.4;

    // Goal tolerances
    double position_tolerance_ = 0.1;  // meters
    double angle_tolerance_ = 0.1;     // radians
};

class TebPlanner : public NavigationStrategy {
public:
    bool initialize(const rclcpp::Node::SharedPtr& node) override {
        try {
            node_ = node;

            // Load TEB parameters
            auto params = node_->get_parameters("teb");
            max_vel_x_ = params[0].as_double();
            max_vel_theta_ = params[1].as_double();
            optimization_steps_ = params[2].as_int();

            initialized_ = true;
            RCLCPP_INFO(node_->get_logger(), "TEB Planner initialized");
            return true;
        } catch (const std::exception& e) {
```

```
            RCLCPP_ERROR(node_->get_logger(),
                "Failed to initialize TEB Planner: %s", e.what());
            return false;
        }
    }

    geometry_msgs::msg::Twist computeVelocity(
        const geometry_msgs::msg::PoseStamped& current_pose,
        const geometry_msgs::msg::PoseStamped& goal_pose,
        const nav_msgs::msg::OccupancyGrid& map) override {

        if (!initialized_) {
            RCLCPP_ERROR(node_->get_logger(), "TEB Planner not initialized");
            return geometry_msgs::msg::Twist();
        }

        try {
            // 1. Generate initial trajectory
            auto initial_trajectory = generateInitialTrajectory(
                current_pose, goal_pose);

            // 2. Optimize trajectory
            auto optimized_trajectory = optimizeTrajectory(
                initial_trajectory, map);

            // 3. Extract velocity command
            return extractVelocity(optimized_trajectory);
        } catch (const std::exception& e) {
            RCLCPP_ERROR(node_->get_logger(),
                "Error computing velocity: %s", e.what());
            return geometry_msgs::msg::Twist();
        }
    }

    bool isGoalReached(
        const geometry_msgs::msg::PoseStamped& current_pose,
        const geometry_msgs::msg::PoseStamped& goal_pose) override {

        // Calculate distance to goal
        double dx = goal_pose.pose.position.x - current_pose.pose.position.x;
        double dy = goal_pose.pose.position.y - current_pose.pose.position.y;
        double distance = std::sqrt(dx*dx + dy*dy);

        // Calculate angle difference
        double angle_diff = calculateAngleDifference(
            current_pose.pose.orientation,
            goal_pose.pose.orientation);

        return distance < position_tolerance_ &&
               std::abs(angle_diff) < angle_tolerance_;
    }

    std::string getName() const override {
        return "TEB_PLANNER";
    }

private:
    Trajectory generateInitialTrajectory(
        const geometry_msgs::msg::PoseStamped& start,
        const geometry_msgs::msg::PoseStamped& goal) {
        // Generate initial trajectory using simple interpolation
        return createInitialGuess(start, goal);
    }

    Trajectory optimizeTrajectory(
        const Trajectory& initial_trajectory,
        const nav_msgs::msg::OccupancyGrid& map) {

        Trajectory current_trajectory = initial_trajectory;

        // Optimize trajectory using TEB approach
        for (int i = 0; i < optimization_steps_; ++i) {
            // 1. Calculate cost terms
            auto costs = calculateCosts(current_trajectory, map);

            // 2. Build optimization problem
            auto problem = buildOptimizationProblem(
                current_trajectory, costs);
```

```
            // 3. Solve one iteration
            current_trajectory = solveIteration(problem);
        }

        return current_trajectory;
    }

    geometry_msgs::msg::Twist extractVelocity(
        const Trajectory& trajectory) {

        geometry_msgs::msg::Twist cmd_vel;

        // Extract velocity from first trajectory segment
        cmd_vel.linear.x = trajectory.segments[0].velocity.linear;
        cmd_vel.angular.z = trajectory.segments[0].velocity.angular;

        return cmd_vel;
    }

    rclcpp::Node::SharedPtr node_;
    bool initialized_ = false;

    // TEB parameters
    double max_vel_x_;
    double max_vel_theta_;
    int optimization_steps_;

    // Goal tolerances
    double position_tolerance_ = 0.1;  // meters
    double angle_tolerance_ = 0.1;     // radians
};

// 3. Context
class NavigationManager {
public:
    NavigationManager(const rclcpp::Node::SharedPtr& node)
        : node_(node) {
        // Initialize available strategies
        strategies_["DWA"] = std::make_unique<DWAPlanner>();
        strategies_["TEB"] = std::make_unique<TebPlanner>();

        // Get default strategy from parameter
        std::string default_strategy =
            node_->get_parameter("default_strategy").as_string();

        // Set default strategy
        setStrategy(default_strategy);

        // Initialize subscribers
        pose_sub_ = node_->create_subscription<geometry_msgs::msg::PoseStamped>(
            "current_pose", 10,
            std::bind(&NavigationManager::poseCallback, this, std::placeholders::_1));

        map_sub_ = node_->create_subscription<nav_msgs::msg::OccupancyGrid>(
            "map", 1,
            std::bind(&NavigationManager::mapCallback, this, std::placeholders::_1));

        goal_sub_ = node_->create_subscription<geometry_msgs::msg::PoseStamped>(
            "goal_pose", 1,
            std::bind(&NavigationManager::goalCallback, this, std::placeholders::_1));

        // Initialize publisher
        cmd_vel_pub_ = node_->create_publisher<geometry_msgs::msg::Twist>(
            "cmd_vel", 10);

        // Create timer for control loop
        timer_ = node_->create_wall_timer(
            std::chrono::milliseconds(100),
            std::bind(&NavigationManager::controlLoop, this));
    }

    bool setStrategy(const std::string& strategy_name) {
        auto it = strategies_.find(strategy_name);
        if (it == strategies_.end()) {
            RCLCPP_ERROR(node_->get_logger(),
                "Unknown strategy: %s", strategy_name.c_str());
            return false;
        }
```

```
        current_strategy_ = it->second.get();
        return current_strategy_->initialize(node_);
    }

    void controlLoop() {
        if (!current_strategy_ || !current_pose_ || !goal_pose_ || !map_) {
            return;
        }

        try {
            // Check if goal is reached
            if (current_strategy_->isGoalReached(*current_pose_, *goal_pose_)) {
                RCLCPP_INFO(node_->get_logger(), "Goal reached!");
                publishZeroVelocity();
                return;
            }

            // Compute velocity command
            auto cmd_vel = current_strategy_->computeVelocity(
                *current_pose_, *goal_pose_, *map_);

            // Publish command
            cmd_vel_pub_->publish(cmd_vel);
        } catch (const std::exception& e) {
            RCLCPP_ERROR(node_->get_logger(),
                "Error in control loop: %s", e.what());
            publishZeroVelocity();
        }
    }

private:
    void poseCallback(
        const geometry_msgs::msg::PoseStamped::SharedPtr msg) {
        current_pose_ = msg;
    }

    void mapCallback(
        const nav_msgs::msg::OccupancyGrid::SharedPtr msg) {
        map_ = msg;
    }

    void goalCallback(
        const geometry_msgs::msg::PoseStamped::SharedPtr msg) {
        goal_pose_ = msg;
    }

    void publishZeroVelocity() {
        geometry_msgs::msg::Twist zero_vel;
        cmd_vel_pub_->publish(zero_vel);
    }

    rclcpp::Node::SharedPtr node_;
    NavigationStrategy* current_strategy_ = nullptr;
    std::map<std::string, std::unique_ptr<NavigationStrategy>> strategies_;

    // Subscribers
    rclcpp::Subscription<geometry_msgs::msg::PoseStamped>::SharedPtr pose_sub_;
    rclcpp::Subscription<nav_msgs::msg::OccupancyGrid>::SharedPtr map_sub_;
    rclcpp::Subscription<geometry_msgs::msg::PoseStamped>::SharedPtr goal_sub_;

    // Publisher
    rclcpp::Publisher<geometry_msgs::msg::Twist>::SharedPtr cmd_vel_pub_;

    // Timer
    rclcpp::TimerBase::SharedPtr timer_;

    // Latest data
    geometry_msgs::msg::PoseStamped::SharedPtr current_pose_;
    geometry_msgs::msg::PoseStamped::SharedPtr goal_pose_;
    nav_msgs::msg::OccupancyGrid::SharedPtr map_;
};

// 4. Usage Example
int main(int argc, char** argv) {
    rclcpp::init(argc, argv);
    auto node = std::make_shared<rclcpp::Node>("navigation_node");

    // Declare parameters
    node->declare_parameter("default_strategy", "DWA");
```

```
    node->declare_parameter("dwa.max_vel_x", 0.5);
    node->declare_parameter("dwa.max_vel_theta", 1.0);
    node->declare_parameter("dwa.acc_lim_x", 2.5);
    node->declare_parameter("dwa.acc_lim_theta", 3.2);

    node->declare_parameter("teb.max_vel_x", 0.4);
    node->declare_parameter("teb.max_vel_theta", 0.9);
    node->declare_parameter("teb.optimization_steps", 3);

    // Create navigation manager
    auto navigation = std::make_shared<NavigationManager>(node);

    // Spin node
    rclcpp::spin(node);
    rclcpp::shutdown();
    return 0;
}
```

### 3. Ví dụ thực tế trong ROS2

### 4. Giải thích chi tiết cách hoạt động

1. **Strategy Interface**:
   - NavigationStrategy định nghĩa interface chung
   - Các phương thức chuẩn cho mọi planner
   - Tính đa hình thông qua virtual methods
2. **Concrete Strategies**:
   - DWAPlanner và TebPlanner implements interface
   - Mỗi planner có thuật toán riêng
   - Error handling và logging
3. **Context**:
   - NavigationManager quản lý strategies
   - Thay đổi strategy runtime
   - Xử lý ROS2 communication

### 5. Ưu điểm trong ROS2

1. **Flexibility**:
   - Dễ dàng thêm planners mới
   - Runtime strategy switching
   - Parameter configuration
2. **Maintainability**:
   - Code organization rõ ràng
   - Separation of concerns
   - Dễ test và debug
3. **Reusability**:
   - Common interface cho planners
   - Shared functionality
   - Code reuse

### 6. Các trường hợp sử dụng trong ROS2

1. **Navigation**:
   - Path planning
   - Obstacle avoidance
   - Local planning
2. **Control**:
   - Motion controllers
   - Joint controllers
   - Force controllers
3. **Perception**:
   - Object detection
   - SLAM algorithms
   - Sensor fusion

### 7. Best Practices trong ROS2

1. **Error Handling**:

```
try {
    auto cmd_vel = strategy->computeVelocity(current_pose, goal_pose, map);
    if (!validateCommand(cmd_vel)) {
        RCLCPP_WARN(logger, "Invalid velocity command");
        return;
    }
    publishCommand(cmd_vel);
} catch (const std::exception& e) {
    RCLCPP_ERROR(logger, "Computation error: %s", e.what());
}
```

2. **Parameter Management**:

```
void loadParameters() {
    auto params = node_->get_parameters("planner");
    for (const auto& param : params) {
        config_[param.get_name()] = param.value_to_string();
    }
}
```

3. **Resource Management**:

```
class SafeStrategy {
    std::unique_ptr<NavigationStrategy> strategy_;
public:
    ~SafeStrategy() {
        if (strategy_) {
            strategy_->cleanup();
        }
    }
};
```

### 8. Mở rộng và tùy chỉnh

1. **Dynamic Loading**:

```
class PluginStrategy : public NavigationStrategy {
    void loadPlugin(const std::string& name) {
        // Load navigation plugin dynamically
        plugin_ = loadNavigationPlugin(name);
    }
};
```

2. **Configuration System**:

```
class ConfigurableStrategy : public NavigationStrategy {
    void configure(const YAML::Node& config) {
        // Configure strategy from YAML
        loadConfiguration(config);
    }
};
```

3. **Hybrid System**:

```
class HybridStrategy : public NavigationStrategy {
    void selectStrategy(const State& state) {
        // Select appropriate strategy based on state
        current_ = chooseStrategy(state);
    }
};
```

### 9. Testing

1. **Mock Objects**:

```
class MockNavigationStrategy : public NavigationStrategy {
public:
    MOCK_METHOD(bool, initialize, (const rclcpp::Node::SharedPtr&), (override));
    MOCK_METHOD(geometry_msgs::msg::Twist, computeVelocity,
        (const geometry_msgs::msg::PoseStamped&,
         const geometry_msgs::msg::PoseStamped&,
         const nav_msgs::msg::OccupancyGrid&), (override));
};
```

2. **Strategy Tests**:

```cpp
TEST(NavigationTest, DWAPlannerTest) {
    auto node = std::make_shared<rclcpp::Node>("test_node");
    auto planner = std::make_unique<DWAPlanner>();

    EXPECT_TRUE(planner->initialize(node));

    geometry_msgs::msg::PoseStamped current_pose;
    geometry_msgs::msg::PoseStamped goal_pose;
    nav_msgs::msg::OccupancyGrid map;

    auto cmd_vel = planner->computeVelocity(current_pose, goal_pose, map);
    EXPECT_NE(cmd_vel.linear.x, 0.0);
}
```

3. **Integration Tests**:

```cpp
TEST(NavigationSystemTest, FullSystemTest) {
    auto node = std::make_shared<rclcpp::Node>("test_node");
    auto navigation = std::make_shared<NavigationManager>(node);

    // Test strategy switching
    EXPECT_TRUE(navigation->setStrategy("DWA"));
    EXPECT_TRUE(navigation->setStrategy("TEB"));

    // Test navigation to goal
    geometry_msgs::msg::PoseStamped goal;
    goal.pose.position.x = 1.0;
    goal.pose.position.y = 1.0;

    navigation->setGoal(goal);

    // Wait for navigation
    rclcpp::spin_some(node);

    // Verify goal reached
    EXPECT_TRUE(navigation->isGoalReached());
}
```

**10. Kết luận** Strategy Pattern là một mẫu thiết kế quan trọng trong ROS2, đặc biệt hữu ích cho việc quản lý các thuật toán và chiến lược khác nhau trong robotics. Pattern này mang lại nhiều lợi ích:

1. **Flexibility và Adaptability**:
   - Dễ dàng thêm thuật toán mới
   - Runtime strategy switching
   - Dynamic configuration
2. **Code Organization**:
   - Clean separation of concerns
   - Interface standards
   - Dễ maintain và test
3. **Error Handling**:
   - Robust error management
   - Graceful degradation
   - Safe fallbacks
4. **Performance Optimization**:
   - Strategy selection based on conditions
   - Resource efficient
   - Runtime optimization

Trong ví dụ về navigation system, chúng ta đã thấy Strategy Pattern giúp xây dựng một hệ thống navigation linh hoạt và mạnh mẽ. Pattern này là lựa chọn tốt cho các hệ thống ROS2 cần quản lý nhiều thuật toán khác nhau và có thể thay đổi chiến lược runtime.

# Design Patterns trong ROS2

ROS2 Design Pattern Guide

2025-06-25

## BUILDER PATTERN TRONG ROS2

**1. Giới thiệu đơn giản** Builder Pattern cho phép xây dựng các đối tượng phức tạp từng bước một cách có cấu trúc. Trong ROS2, pattern này đặc biệt hữu ích cho:

- Tạo robot configurations phức tạp
- Xây dựng launch files động
- Tạo sensor fusion pipelines
- Cấu hình navigation stacks
- Khởi tạo multi-robot systems

**2. Định nghĩa chi tiết** Builder Pattern tách biệt việc xây dựng đối tượng phức tạp khỏi cách biểu diễn của nó, cho phép cùng một quá trình xây dựng có thể tạo ra các biểu diễn khác nhau.

**Các thành phần chính:**

1. **Product**:
   - Đối tượng phức tạp cần tạo
   - Chứa nhiều components
2. **Builder Interface**:
   - Định nghĩa các bước xây dựng
   - Abstract interface cho builders
3. **Concrete Builders**:
   - Implements các bước cụ thể
   - Tạo product variants khác nhau
4. **Director**:
   - Điều khiển quá trình xây dựng
   - Sử dụng builder interface

```
// 1. Product - Robot Configuration
class RobotConfiguration {
public:
    struct SensorConfig {
        std::string type;
        std::string topic;
        std::string frame_id;
        double frequency;
        std::map<std::string, double> parameters;
    };

    struct NavigationConfig {
        std::string global_planner;
        std::string local_planner;
        std::string costmap_type;
        std::map<std::string, double> parameters;
    };

    struct HardwareConfig {
        std::string driver_type;
        std::string device_path;
        std::map<std::string, std::string> hardware_params;
    };

    void addSensor(const SensorConfig& sensor) {
        sensors_.push_back(sensor);
    }
```

```
    void setNavigation(const NavigationConfig& nav) {
        navigation_ = nav;
    }

    void setHardware(const HardwareConfig& hw) {
        hardware_ = hw;
    }

    void setRobotDescription(const std::string& urdf_path) {
        robot_description_path_ = urdf_path;
    }

    void addLaunchFile(const std::string& package, const std::string& file) {
        launch_files_.push_back({package, file});
    }

    // Getters
    const std::vector<SensorConfig>& getSensors() const { return sensors_; }
    const NavigationConfig& getNavigation() const { return navigation_; }
    const HardwareConfig& getHardware() const { return hardware_; }
    const std::string& getRobotDescription() const { return robot_description_path_; }
    const std::vector<std::pair<std::string, std::string>>& getLaunchFiles() const {
        return launch_files_;
    }

    // Validation
    bool isValid() const {
        return !sensors_.empty() &&
               !robot_description_path_.empty() &&
               !hardware_.driver_type.empty();
    }

    // Generate launch content
    std::string generateLaunchContent() const {
        std::stringstream ss;
        ss << "<?xml version=\"1.0\"?>\n";
        ss << "<launch>\n";

        // Robot description
        ss << "  <param name=\"robot_description\" textfile=\""
           << robot_description_path_ << "\" />\n";

        // Hardware driver
        ss << "  <node pkg=\"" << hardware_.driver_type << "\" type=\"driver_node\" name=\"hardware_driver\">\n";
        for (const auto& param : hardware_.hardware_params) {
            ss << "    <param name=\"" << param.first << "\" value=\"" << param.second << "\" />\n";
        }
        ss << "  </node>\n";

        // Sensors
        for (const auto& sensor : sensors_) {
            ss << "  <node pkg=\"" << sensor.type << "\" type=\"sensor_node\" name=\"" << sensor.type << "_node\"
            ss << "    <param name=\"topic\" value=\"" << sensor.topic << "\" />\n";
            ss << "    <param name=\"frame_id\" value=\"" << sensor.frame_id << "\" />\n";
            ss << "    <param name=\"frequency\" value=\"" << sensor.frequency << "\" />\n";
            for (const auto& param : sensor.parameters) {
                ss << "    <param name=\"" << param.first << "\" value=\"" << param.second << "\" />\n";
            }
            ss << "  </node>\n";
        }

        // Navigation
        if (!navigation_.global_planner.empty()) {
            ss << "  <include file=\"$(find nav2_bringup)/launch/navigation_launch.py\">\n";
            ss << "    <arg name=\"global_planner\" value=\"" << navigation_.global_planner << "\" />\n";
            ss << "    <arg name=\"local_planner\" value=\"" << navigation_.local_planner << "\" />\n";
            ss << "  </include>\n";
        }

        ss << "</launch>\n";
        return ss.str();
    }

private:
    std::vector<SensorConfig> sensors_;
    NavigationConfig navigation_;
    HardwareConfig hardware_;
    std::string robot_description_path_;
```

```
    std::vector<std::pair<std::string, std::string>> launch_files_;
};

// 2. Builder Interface
class RobotConfigurationBuilder {
public:
    virtual ~RobotConfigurationBuilder() = default;

    virtual RobotConfigurationBuilder& setSensorSuite() = 0;
    virtual RobotConfigurationBuilder& setNavigationStack() = 0;
    virtual RobotConfigurationBuilder& setHardwareInterface() = 0;
    virtual RobotConfigurationBuilder& setRobotModel() = 0;
    virtual RobotConfigurationBuilder& addCustomComponents() = 0;

    virtual std::unique_ptr<RobotConfiguration> build() = 0;

protected:
    std::unique_ptr<RobotConfiguration> configuration_;

    void reset() {
        configuration_ = std::make_unique<RobotConfiguration>();
    }
};

// 3. Concrete Builders
class TurtleBotBuilder : public RobotConfigurationBuilder {
public:
    TurtleBotBuilder() {
        reset();
    }

    RobotConfigurationBuilder& setSensorSuite() override {
        // Add LiDAR
        RobotConfiguration::SensorConfig lidar;
        lidar.type = "rplidar_ros";
        lidar.topic = "/scan";
        lidar.frame_id = "laser_frame";
        lidar.frequency = 10.0;
        lidar.parameters["angle_compensate"] = 1.0;
        lidar.parameters["serial_port"] = 0;  // /dev/ttyUSB0
        configuration_->addSensor(lidar);

        // Add Camera
        RobotConfiguration::SensorConfig camera;
        camera.type = "usb_cam";
        camera.topic = "/camera/image_raw";
        camera.frame_id = "camera_frame";
        camera.frequency = 30.0;
        camera.parameters["video_device"] = 0;  // /dev/video0
        camera.parameters["image_width"] = 640.0;
        camera.parameters["image_height"] = 480.0;
        configuration_->addSensor(camera);

        // Add IMU
        RobotConfiguration::SensorConfig imu;
        imu.type = "imu_node";
        imu.topic = "/imu/data";
        imu.frame_id = "imu_frame";
        imu.frequency = 100.0;
        imu.parameters["calibration_time"] = 10.0;
        configuration_->addSensor(imu);

        return *this;
    }

    RobotConfigurationBuilder& setNavigationStack() override {
        RobotConfiguration::NavigationConfig nav;
        nav.global_planner = "NavfnPlanner";
        nav.local_planner = "DWBLocalPlanner";
        nav.costmap_type = "costmap_2d";
        nav.parameters["controller_frequency"] = 20.0;
        nav.parameters["planner_patience"] = 5.0;
        nav.parameters["oscillation_timeout"] = 10.0;
        nav.parameters["oscillation_distance"] = 0.5;

        configuration_->setNavigation(nav);
        return *this;
    }
```

```
    RobotConfigurationBuilder& setHardwareInterface() override {
        RobotConfiguration::HardwareConfig hw;
        hw.driver_type = "turtlebot3_bringup";
        hw.device_path = "/dev/ttyACM0";
        hw.hardware_params["baud"] = "115200";
        hw.hardware_params["timeout"] = "1000";
        hw.hardware_params["model"] = "burger";

        configuration_->setHardware(hw);
        return *this;
    }

    RobotConfigurationBuilder& setRobotModel() override {
        configuration_->setRobotDescription(
            "$(find turtlebot3_description)/urdf/turtlebot3_burger.urdf");
        return *this;
    }

    RobotConfigurationBuilder& addCustomComponents() override {
        // Add teleop
        configuration_->addLaunchFile("turtlebot3_teleop", "turtlebot3_teleop_key.launch");

        // Add visualization
        configuration_->addLaunchFile("turtlebot3_bringup", "turtlebot3_rviz.launch");

        return *this;
    }

    std::unique_ptr<RobotConfiguration> build() override {
        if (!configuration_->isValid()) {
            throw std::runtime_error("Invalid TurtleBot configuration");
        }

        auto result = std::move(configuration_);
        reset();  // Reset for next build
        return result;
    }
};

class IndustrialRobotBuilder : public RobotConfigurationBuilder {
public:
    IndustrialRobotBuilder() {
        reset();
    }

    RobotConfigurationBuilder& setSensorSuite() override {
        // Add Force/Torque Sensor
        RobotConfiguration::SensorConfig ft_sensor;
        ft_sensor.type = "robotiq_ft_sensor";
        ft_sensor.topic = "/ft_sensor/wrench";
        ft_sensor.frame_id = "ft_sensor_frame";
        ft_sensor.frequency = 125.0;
        ft_sensor.parameters["calibration_matrix"] = 1.0;
        ft_sensor.parameters["bias_compensation"] = 1.0;
        configuration_->addSensor(ft_sensor);

        // Add Vision System
        RobotConfiguration::SensorConfig vision;
        vision.type = "industrial_camera";
        vision.topic = "/camera/image_raw";
        vision.frame_id = "camera_frame";
        vision.frequency = 60.0;
        vision.parameters["exposure_time"] = 5000.0;
        vision.parameters["gain"] = 1.0;
        vision.parameters["trigger_mode"] = 1.0;
        configuration_->addSensor(vision);

        // Add Joint State Sensors
        RobotConfiguration::SensorConfig joint_states;
        joint_states.type = "joint_state_publisher";
        joint_states.topic = "/joint_states";
        joint_states.frame_id = "base_link";
        joint_states.frequency = 50.0;
        configuration_->addSensor(joint_states);

        return *this;
    }

    RobotConfigurationBuilder& setNavigationStack() override {
```

```
        // Industrial robots typically don't use mobile navigation
        // Instead, they use motion planning
        RobotConfiguration::NavigationConfig nav;
        nav.global_planner = "OMPL";
        nav.local_planner = "Pilz";
        nav.costmap_type = "collision_detection";
        nav.parameters["planning_time"] = 5.0;
        nav.parameters["max_velocity_scaling"] = 0.5;
        nav.parameters["max_acceleration_scaling"] = 0.3;

        configuration_->setNavigation(nav);
        return *this;
    }

    RobotConfigurationBuilder& setHardwareInterface() override {
        RobotConfiguration::HardwareConfig hw;
        hw.driver_type = "ur_robot_driver";
        hw.device_path = "192.168.1.100";  // IP address for industrial robots
        hw.hardware_params["robot_ip"] = "192.168.1.100";
        hw.hardware_params["kinematics_config"] = "/etc/ur_cal/ur5e_calibration.yaml";
        hw.hardware_params["use_tool_communication"] = "true";
        hw.hardware_params["tool_voltage"] = "24";

        configuration_->setHardware(hw);
        return *this;
    }

    RobotConfigurationBuilder& setRobotModel() override {
        configuration_->setRobotDescription(
            "$(find ur_description)/urdf/ur5e_robot.urdf.xacro");
        return *this;
    }

    RobotConfigurationBuilder& addCustomComponents() override {
        // Add MoveIt planning
        configuration_->addLaunchFile("ur5e_moveit_config", "ur5e_moveit_planning_execution.launch");

        // Add safety monitoring
        configuration_->addLaunchFile("ur_robot_driver", "ur5e_safety_monitor.launch");

        // Add tool control
        configuration_->addLaunchFile("robotiq_2f_gripper_control", "robotiq_action_server.launch");

        return *this;
    }

    std::unique_ptr<RobotConfiguration> build() override {
        if (!configuration_->isValid()) {
            throw std::runtime_error("Invalid Industrial Robot configuration");
        }

        auto result = std::move(configuration_);
        reset();
        return result;
    }
};

class DroneBuilder : public RobotConfigurationBuilder {
public:
    DroneBuilder() {
        reset();
    }

    RobotConfigurationBuilder& setSensorSuite() override {
        // Add GPS
        RobotConfiguration::SensorConfig gps;
        gps.type = "nmea_navsat_driver";
        gps.topic = "/gps/fix";
        gps.frame_id = "gps_frame";
        gps.frequency = 10.0;
        gps.parameters["port"] = 0;  // /dev/ttyUSB0
        gps.parameters["baud"] = 4800.0;
        configuration_->addSensor(gps);

        // Add IMU
        RobotConfiguration::SensorConfig imu;
        imu.type = "imu_node";
        imu.topic = "/imu/data";
        imu.frame_id = "imu_frame";
```

```
        imu.frequency = 200.0;
        imu.parameters["calibration_samples"] = 1000.0;
        configuration_->addSensor(imu);

        // Add Camera for visual odometry
        RobotConfiguration::SensorConfig camera;
        camera.type = "usb_cam";
        camera.topic = "/camera/image_raw";
        camera.frame_id = "camera_frame";
        camera.frequency = 30.0;
        camera.parameters["video_device"] = 0;
        camera.parameters["image_width"] = 1280.0;
        camera.parameters["image_height"] = 720.0;
        configuration_->addSensor(camera);

        // Add Barometer
        RobotConfiguration::SensorConfig baro;
        baro.type = "baro_node";
        baro.topic = "/barometer/pressure";
        baro.frame_id = "base_link";
        baro.frequency = 50.0;
        configuration_->addSensor(baro);

        return *this;
    }

    RobotConfigurationBuilder& setNavigationStack() override {
        RobotConfiguration::NavigationConfig nav;
        nav.global_planner = "PX4_mission_planner";
        nav.local_planner = "position_controller";
        nav.costmap_type = "3d_costmap";
        nav.parameters["max_velocity_horizontal"] = 5.0;
        nav.parameters["max_velocity_vertical"] = 3.0;
        nav.parameters["position_tolerance"] = 0.5;
        nav.parameters["yaw_tolerance"] = 0.1;

        configuration_->setNavigation(nav);
        return *this;
    }

    RobotConfigurationBuilder& setHardwareInterface() override {
        RobotConfiguration::HardwareConfig hw;
        hw.driver_type = "mavros";
        hw.device_path = "/dev/ttyACM0";
        hw.hardware_params["fcu_url"] = "/dev/ttyACM0:57600";
        hw.hardware_params["gcs_url"] = "udp://:14550@127.0.0.1:14557";
        hw.hardware_params["target_system_id"] = "1";
        hw.hardware_params["target_component_id"] = "1";

        configuration_->setHardware(hw);
        return *this;
    }

    RobotConfigurationBuilder& setRobotModel() override {
        configuration_->setRobotDescription(
            "$(find drone_description)/urdf/quadrotor.urdf.xacro");
        return *this;
    }

    RobotConfigurationBuilder& addCustomComponents() override {
        // Add MAVROS
        configuration_->addLaunchFile("mavros", "px4.launch");

        // Add visual odometry
        configuration_->addLaunchFile("vio_estimator", "vio_estimator.launch");

        // Add mission control
        configuration_->addLaunchFile("drone_control", "mission_control.launch");

        return *this;
    }

    std::unique_ptr<RobotConfiguration> build() override {
        if (!configuration_->isValid()) {
            throw std::runtime_error("Invalid Drone configuration");
        }

        auto result = std::move(configuration_);
        reset();
```

```
        return result;
    }
};

// 4. Director
class RobotConfigurationDirector {
public:
    void setBuilder(std::unique_ptr<RobotConfigurationBuilder> builder) {
        builder_ = std::move(builder);
    }

    std::unique_ptr<RobotConfiguration> buildBasicRobot() {
        if (!builder_) {
            throw std::runtime_error("No builder set");
        }

        return builder_->setSensorSuite()
                       .setHardwareInterface()
                       .setRobotModel()
                       .build();
    }

    std::unique_ptr<RobotConfiguration> buildFullyEquippedRobot() {
        if (!builder_) {
            throw std::runtime_error("No builder set");
        }

        return builder_->setSensorSuite()
                       .setNavigationStack()
                       .setHardwareInterface()
                       .setRobotModel()
                       .addCustomComponents()
                       .build();
    }

    std::unique_ptr<RobotConfiguration> buildCustomRobot(
        const std::function<RobotConfigurationBuilder&(RobotConfigurationBuilder&)>& customizer) {

        if (!builder_) {
            throw std::runtime_error("No builder set");
        }

        return customizer(*builder_).build();
    }

private:
    std::unique_ptr<RobotConfigurationBuilder> builder_;
};

// 5. ROS2 Integration
class RobotLauncher {
public:
    RobotLauncher(const rclcpp::Node::SharedPtr& node) : node_(node) {}

    bool launchRobot(const RobotConfiguration& config) {
        try {
            // Validate configuration
            if (!config.isValid()) {
                RCLCPP_ERROR(node_->get_logger(), "Invalid robot configuration");
                return false;
            }

            // Generate launch file
            std::string launch_content = config.generateLaunchContent();
            std::string launch_file_path = "/tmp/robot_config.launch";

            // Write launch file
            std::ofstream launch_file(launch_file_path);
            if (!launch_file.is_open()) {
                RCLCPP_ERROR(node_->get_logger(), "Failed to create launch file");
                return false;
            }
            launch_file << launch_content;
            launch_file.close();

            // Launch robot
            RCLCPP_INFO(node_->get_logger(), "Launching robot with configuration:");
            RCLCPP_INFO(node_->get_logger(), "  Sensors: %zu", config.getSensors().size());
            RCLCPP_INFO(node_->get_logger(), "  Navigation: %s",
```

```
                config.getNavigation().global_planner.c_str());
            RCLCPP_INFO(node_->get_logger(), "  Hardware: %s",
                config.getHardware().driver_type.c_str());

            // Start launch process
            return startLaunchProcess(launch_file_path);

        } catch (const std::exception& e) {
            RCLCPP_ERROR(node_->get_logger(), "Failed to launch robot: %s", e.what());
            return false;
        }
    }

private:
    bool startLaunchProcess(const std::string& launch_file) {
        // In real implementation, this would start the ROS2 launch process
        std::string command = "ros2 launch " + launch_file;
        RCLCPP_INFO(node_->get_logger(), "Executing: %s", command.c_str());

        // For demo purposes, just return true
        return true;
    }

    rclcpp::Node::SharedPtr node_;
};

// 6. Usage Example
int main(int argc, char** argv) {
    rclcpp::init(argc, argv);
    auto node = std::make_shared<rclcpp::Node>("robot_configuration_node");

    try {
        // Create director
        RobotConfigurationDirector director;

        // Example 1: Build TurtleBot
        RCLCPP_INFO(node->get_logger(), "Building TurtleBot configuration...");
        director.setBuilder(std::make_unique<TurtleBotBuilder>());
        auto turtlebot_config = director.buildFullyEquippedRobot();

        // Launch TurtleBot
        RobotLauncher launcher(node);
        launcher.launchRobot(*turtlebot_config);

        // Example 2: Build Industrial Robot
        RCLCPP_INFO(node->get_logger(), "Building Industrial Robot configuration...");
        director.setBuilder(std::make_unique<IndustrialRobotBuilder>());
        auto industrial_config = director.buildFullyEquippedRobot();

        // Example 3: Build Custom Drone
        RCLCPP_INFO(node->get_logger(), "Building Custom Drone configuration...");
        director.setBuilder(std::make_unique<DroneBuilder>());
        auto drone_config = director.buildCustomRobot([](RobotConfigurationBuilder& builder) -> RobotConfiguratio
            return builder.setSensorSuite()
                          .setHardwareInterface()
                          .setRobotModel();
            // Skip navigation and custom components for basic drone
        });

        // Example 4: Runtime configuration based on parameters
        std::string robot_type = node->declare_parameter<std::string>("robot_type", "turtlebot");

        if (robot_type == "turtlebot") {
            director.setBuilder(std::make_unique<TurtleBotBuilder>());
        } else if (robot_type == "industrial") {
            director.setBuilder(std::make_unique<IndustrialRobotBuilder>());
        } else if (robot_type == "drone") {
            director.setBuilder(std::make_unique<DroneBuilder>());
        } else {
            RCLCPP_ERROR(node->get_logger(), "Unknown robot type: %s", robot_type.c_str());
            return 1;
        }

        auto runtime_config = director.buildFullyEquippedRobot();
        launcher.launchRobot(*runtime_config);

        rclcpp::spin(node);

    } catch (const std::exception& e) {
```

```
        RCLCPP_ERROR(node->get_logger(), "Application error: %s", e.what());
        return 1;
    }

    rclcpp::shutdown();
    return 0;
}
```

### 3. Ví dụ thực tế trong ROS2

### 4. Giải thích chi tiết cách hoạt động

1. **Product (RobotConfiguration)**:
   - Chứa tất cả thông tin cấu hình robot
   - Sensors, navigation, hardware, model
   - Validation và generation logic
2. **Builder Interface**:
   - Định nghĩa các bước xây dựng chuẩn
   - Fluent interface pattern
   - Method chaining
3. **Concrete Builders**:
   - TurtleBotBuilder: Mobile robot với LiDAR, camera
   - IndustrialRobotBuilder: Manipulator với force sensor
   - DroneBuilder: UAV với GPS, IMU, camera
4. **Director**:
   - Điều khiển quá trình xây dựng
   - Predefined build sequences
   - Custom build support

### 5. Ưu điểm trong ROS2

1. **Flexibility**:
   - Dễ dàng tạo robot configurations khác nhau
   - Runtime configuration switching
   - Modular component assembly
2. **Maintainability**:
   - Clear separation of concerns
   - Reusable components
   - Standardized interfaces
3. **Extensibility**:
   - Dễ thêm robot types mới
   - Plugin architecture support
   - Configuration validation

### 6. Các trường hợp sử dụng trong ROS2

1. **Robot Fleet Management**:

```
class FleetConfigurationBuilder : public RobotConfigurationBuilder {
    std::vector<std::unique_ptr<RobotConfiguration>> fleet_;

public:
    FleetConfigurationBuilder& addRobot(const std::string& type) {
        // Add robot to fleet
        return *this;
    }

    FleetConfigurationBuilder& setFleetCommunication() {
        // Configure inter-robot communication
        return *this;
    }

    std::unique_ptr<FleetConfiguration> buildFleet() {
        // Return complete fleet configuration
        return std::make_unique<FleetConfiguration>(std::move(fleet_));
    }
};
```

2. **Dynamic Reconfiguration**:

```
class ReconfigurableRobotBuilder : public RobotConfigurationBuilder {
public:
    RobotConfigurationBuilder& addSensorConditionally(
        const std::string& condition) {

        if (evaluateCondition(condition)) {
            // Add sensor based on runtime condition
        }
        return *this;
    }

    RobotConfigurationBuilder& setAdaptiveNavigation() {
        // Configure navigation based on environment
        return *this;
    }
};
```

3. **Multi-Environment Configurations**:

```
class EnvironmentAwareBuilder : public RobotConfigurationBuilder {
    enum Environment { INDOOR, OUTDOOR, UNDERWATER, SPACE };
    Environment env_;

public:
    EnvironmentAwareBuilder(Environment env) : env_(env) {}

    RobotConfigurationBuilder& setSensorSuite() override {
        switch (env_) {
            case INDOOR:
                addIndoorSensors();
                break;
            case OUTDOOR:
                addOutdoorSensors();
                break;
            case UNDERWATER:
                addUnderwaterSensors();
                break;
            case SPACE:
                addSpaceSensors();
                break;
        }
        return *this;
    }
};
```

### 7. Best Practices trong ROS2

1. **Configuration Validation**:

```
class ValidatedBuilder : public RobotConfigurationBuilder {
    bool validateSensorCompatibility() {
        // Check sensor compatibility
        return true;
    }

    bool validateHardwareRequirements() {
        // Check hardware requirements
        return true;
    }

public:
    std::unique_ptr<RobotConfiguration> build() override {
        if (!validateSensorCompatibility()) {
            throw std::runtime_error("Sensor compatibility check failed");
        }

        if (!validateHardwareRequirements()) {
            throw std::runtime_error("Hardware requirements not met");
        }

        return std::move(configuration_);
    }
};
```

2. **Resource Management**:

```
class ResourceAwareBuilder : public RobotConfigurationBuilder {
    struct ResourceLimits {
```

```
        double max_cpu_usage;
        double max_memory_usage;
        double max_network_bandwidth;
    };

    ResourceLimits limits_;

public:
    RobotConfigurationBuilder& setResourceLimits(const ResourceLimits& limits) {
        limits_ = limits;
        return *this;
    }

    RobotConfigurationBuilder& optimizeForResources() {
        // Adjust configuration based on resource limits
        return *this;
    }
};
```

3. **Error Handling**:

```
class RobustBuilder : public RobotConfigurationBuilder {
public:
    RobotConfigurationBuilder& setSensorSuite() override {
        try {
            // Try to configure primary sensors
            configurePrimarySensors();
        } catch (const std::exception& e) {
            RCLCPP_WARN(logger_, "Primary sensors failed, using fallback: %s", e.what());
            configureFallbackSensors();
        }
        return *this;
    }

private:
    void configurePrimarySensors() {
        // Configure preferred sensors
    }

    void configureFallbackSensors() {
        // Configure fallback sensors
    }

    rclcpp::Logger logger_ = rclcpp::get_logger("RobustBuilder");
};
```

## 8. Mở rộng và tùy chỉnh

1. **Plugin System**:

```
class PluginBuilder : public RobotConfigurationBuilder {
    std::map<std::string, std::shared_ptr<ComponentPlugin>> plugins_;

public:
    RobotConfigurationBuilder& loadPlugin(const std::string& name) {
        auto plugin = loadComponentPlugin(name);
        plugins_[name] = plugin;
        return *this;
    }

    RobotConfigurationBuilder& configurePlugin(
        const std::string& name, const YAML::Node& config) {

        if (auto it = plugins_.find(name); it != plugins_.end()) {
            it->second->configure(config);
        }
        return *this;
    }
};
```

2. **Template-based Builders**:

```
template<typename RobotType>
class TemplateRobotBuilder : public RobotConfigurationBuilder {
public:
    RobotConfigurationBuilder& setSpecializedComponents() {
        // Use template specialization for robot-specific components
        RobotType::configureSensors(*this);
```

```
        RobotType::configureNavigation(*this);
        return *this;
    }
};

// Specializations
struct TurtleBotType {
    static void configureSensors(RobotConfigurationBuilder& builder) {
        // TurtleBot-specific sensor configuration
    }

    static void configureNavigation(RobotConfigurationBuilder& builder) {
        // TurtleBot-specific navigation configuration
    }
};
```

3. **Configuration from Files**:

```
class FileBasedBuilder : public RobotConfigurationBuilder {
public:
    RobotConfigurationBuilder& loadFromYAML(const std::string& file_path) {
        try {
            YAML::Node config = YAML::LoadFile(file_path);

            // Parse sensors
            if (config["sensors"]) {
                for (const auto& sensor_config : config["sensors"]) {
                    addSensorFromYAML(sensor_config);
                }
            }

            // Parse navigation
            if (config["navigation"]) {
                setNavigationFromYAML(config["navigation"]);
            }

            // Parse hardware
            if (config["hardware"]) {
                setHardwareFromYAML(config["hardware"]);
            }

        } catch (const YAML::Exception& e) {
            throw std::runtime_error("Failed to load configuration: " + std::string(e.what()));
        }

        return *this;
    }

private:
    void addSensorFromYAML(const YAML::Node& sensor_node) {
        RobotConfiguration::SensorConfig sensor;
        sensor.type = sensor_node["type"].as<std::string>();
        sensor.topic = sensor_node["topic"].as<std::string>();
        sensor.frame_id = sensor_node["frame_id"].as<std::string>();
        sensor.frequency = sensor_node["frequency"].as<double>();

        if (sensor_node["parameters"]) {
            for (const auto& param : sensor_node["parameters"]) {
                sensor.parameters[param.first.as<std::string>()] =
                    param.second.as<double>();
            }
        }

        configuration_->addSensor(sensor);
    }
};
```

## 9. Testing

1. **Builder Tests**:

```
TEST(BuilderTest, TurtleBotBuilderTest) {
    TurtleBotBuilder builder;

    auto config = builder.setSensorSuite()
                         .setNavigationStack()
                         .setHardwareInterface()
                         .setRobotModel()
```

```
                        .build();

    ASSERT_TRUE(config->isValid());
    EXPECT_EQ(config->getSensors().size(), 3);  // LiDAR, Camera, IMU
    EXPECT_EQ(config->getNavigation().global_planner, "NavfnPlanner");
    EXPECT_EQ(config->getHardware().driver_type, "turtlebot3_bringup");
}
```

2. **Director Tests**:

```
TEST(DirectorTest, BuilderSelectionTest) {
    RobotConfigurationDirector director;

    // Test TurtleBot
    director.setBuilder(std::make_unique<TurtleBotBuilder>());
    auto turtlebot = director.buildFullyEquippedRobot();
    EXPECT_EQ(turtlebot->getHardware().driver_type, "turtlebot3_bringup");

    // Test Industrial Robot
    director.setBuilder(std::make_unique<IndustrialRobotBuilder>());
    auto industrial = director.buildFullyEquippedRobot();
    EXPECT_EQ(industrial->getHardware().driver_type, "ur_robot_driver");
}
```

3. **Integration Tests**:

```
TEST(IntegrationTest, LaunchConfigurationTest) {
    auto node = std::make_shared<rclcpp::Node>("test_node");
    RobotLauncher launcher(node);

    TurtleBotBuilder builder;
    auto config = builder.setSensorSuite()
                        .setHardwareInterface()
                        .setRobotModel()
                        .build();

    EXPECT_TRUE(launcher.launchRobot(*config));
}
```

**10. Kết luận** Builder Pattern là một mẫu thiết kế cực kỳ hữu ích trong ROS2 robotics, đặc biệt cho việc tạo các cấu hình robot phức tạp. Pattern này mang lại:

1. **Flexibility trong Configuration**:
   - Dễ dàng tạo nhiều robot types khác nhau
   - Runtime configuration switching
   - Modular component assembly
2. **Code Organization**:
   - Clear separation giữa construction logic
   - Reusable components
   - Standardized interfaces
3. **Maintainability**:
   - Dễ thêm robot types mới
   - Configuration validation
   - Error handling
4. **Integration với ROS2**:
   - Launch file generation
   - Parameter management
   - Node lifecycle management

Builder Pattern đặc biệt phù hợp cho các hệ thống robotics phức tạp cần quản lý nhiều components, sensors, và configurations khác nhau. Trong ví dụ trên, chúng ta đã thấy cách pattern này giúp xây dựng một hệ thống configuration linh hoạt và mạnh mẽ cho nhiều loại robot khác nhau.