UNIVERSITÀ
DI TRENTO
Department of
Information Engineering and Computer Science

Testing Your Code

Edoardo Lamon

Software Development for Collaborative Robots

Academic Year 2025/26

Why “pure software” testing?

In the development of a robotic application, the testing on the robot hardware is the fun part (that many AI/CV apps do not have). However, this is also a delicate process:

- Hardware is **expensive** (mistakes can cause **damage**);
- Limited **availability** (maybe it is shared with other developers);
- **Repeatability** can be difficult to achieve in real-world conditions;
- Debugging the software in case of a hardware test failure can be **time-consuming and challenging,** making it difficult to reproduce errors.

Before going to the hardware, we want to detect and remove the software bugs.

Test Driven Development (TDD)

TDD is a **software development paradigm** where you **write the tests before writing the code.**

The TDD cycle:

1. **Write a test** based on system requirements.
2. **Run it → it fails** (no code yet!).
3. **Implement** the minimal code needed to pass the test.
4. **Run all tests again;** iterate until everything passes.
5. **Refactor** and improve the code while keeping tests green.

Why It Works?

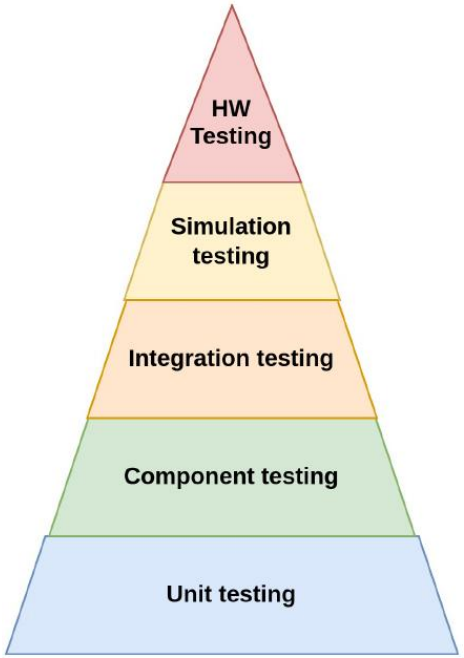
- Forces **clear requirements** before coding.
- Encourages **testable, modular design** from the start.
- Prevents “retrofitting” tests to poorly structured code.
- Makes debugging faster and safer.

When your system requirements are well-defined, **start from the tests,** not the code!

Testing in ROS2

The modularity of ROS2 helps a lot:

- ROS 2 applications are composed of many **independent building blocks** (packages, nodes, libraries).
- Each block can be **tested in isolation**, ensuring it behaves as expected before integration.
- This **reduces complexity** and increases trust in higher-level system tests.



Test Level	Scope	Goal
Unit Tests	Individual functions or libraries	Verify correctness of small, isolated pieces of code
Node Tests	Single ROS 2 nodes	Ensure correct ROS API behavior (topics, services, actions)
Integration Tests	Groups of nodes	Validate communication and interaction between components
System Tests	Full application	Verify end-to-end functionality in a realistic environment

Unit Testing

Verify that a small piece of code (function, class, or method) produces the expected output for a given input.

Advantages

- **Easier debugging** → failures point directly to the responsible function.
- **Improved readability** → tests act as **"executable documentation"**, helping understand legacy code.
- **Fine control** → allows targeted validation of low-level logic.

Disadvantages

- Very fine-grained testing can cause **maintenance overhead**: small code changes may require updating many tests.
- Find a **balance:** more detailed tests for **stable core functions**, broader ones for **frequently changing code**.

Unit Testing in ROS2

Unit test focus on isolated components, **without involving ROS communication.**

→We can use standard tools for unit testing without any adaptation;

→the ROS2 tools will simply make it easier for you to install and execute them with `colcon test`

Language	**Testing Framework**	**Integration in ROS 2**
C++	[**GTest** (Google Test)]	via `ament_add_gtest()`
Python	**Pytest**/**unittest**	via `ament_add_pytest_test()`

GTest in ROS 2

Google Test (GTest) is a popular, open-source framework for **C/C++ unit testing**.

- Common default choice for **ROS 2 C++ projects**, especially those interacting with **hardware**.
- Supports [mocking] (simulated interfaces) — perfect for testing code without touching real devices.

To define one or more tests, Gtests provides a [set of macros] that are automatically discovered at initialization time. *Tests* use assertions to verify the tested code's behavior. If a test crashes or has a failed assertion, then it *fails*; otherwise, it *succeeds*.

```
TEST(TestSuiteName, TestName) {
  ... Run some stuff
  ... Make some asserions
}
```

The [assertions] are another set of macros that allow you to verify that the outputs of the algorithm you are testing are within the expected ones. An assertion's result can be *success*, *nonfatal failure*, or *fatal failure*. If a fatal failure occurs, it aborts the current function; otherwise, the program continues normally.

GTest in ROS 2

```cpp
#include <gtest/gtest.h>
#include "rclcpp/rclcpp.hpp"

TEST(MyNodeTest, simple_check)
{
  auto node = std::make_shared<rclcpp::Node>("test_node");
  ASSERT_TRUE(node->count_publishers("/chatter") == 1);
}

int main(int argc, char **argv)
{
  testing::InitGoogleTest(&argc, argv);
  rclcpp::init(argc, argv);
  int result = RUN_ALL_TESTS();
  rclcpp::shutdown();
  return result;
}
```

CMake Integration:

```cmake
ament_add_gtest(my_test test/my_test.cpp)
ament_target_dependencies(my_test rclcpp)
```

Then simply run:

```
colcon test
```

GTest in ROS 2 – Text Fixtures

- When several tests share **setup or common data**, use a **fixture class** to avoid code repetition.
- The class derives from testing::Test.
- Each test **TEST_F** runs as a **method** of that class, with access to its members.
- The fixture *TestFixturesName* is created before each test (calling *SetUp()*) and destroyed after (calling *TearDown()*), ensuring isolation and clean state.

```

#include <rclcpp/rclcpp.hpp>
#include <gtest/gtest.h>

#include "a_class_using_ros.hpp"

class TestFixtureName : public testing::Test {
 public:
  TestFixtureName()
      : node_(std::make_shared<rclcpp::Node>("
    test_with_node")){}

  void SetUp() override {
    // Any code that should execute before the test starts
    node_->declare_parameter("param_name", std::string{"
    param_default_value"});
  }

  void TearDown() override {
    // Any code that should execute after the test run
  }

 protected:
  rclcpp::Node::SharedPtr node_;
};

TEST_F(TestFixtureName, TestMethodTrue) {
  AClassUsingRos class = AClassUsingRos(node_);
  EXPECT_TRUE(class.aClassMethodThatShouldReturnTrue());
}

TEST_F(TestFixtureName, AnotherTestName) {
  EXPECT_EQ(aFunctionThatShouldReturnOne(node_), 1);
}
```

GTest in ROS 2 – Parametrized Text

- Use when the **same logic** must run with **different inputs**.
- TEST_P macro allows you to define a vector with different parameters against which your test will be executed.
- Define test parameters with INSTANTIATE_TEST_CASE_P macro.
- Access them via GetParam() inside the test.
- Multiple parameters can be passed with [Tuples](#).

```cpp

class ParamTestFixtureName : public ::testing::
    TestWithParam<std::tuple<int, int, bool>> { ... }

TEST_P(ParamTestFixtureName, TestSmaller) {
    auto first_element = std::get<0>(GetParam());
    auto second_element = std::get<1>(GetParam());
    auto expected_result = std::get<2>(GetParam());
    EXPECT_EQ(IsSmaller(first_element, second_element),
    expected_result);
}

INSTANTIATE_TEST_CASE_P(
        BoringSmallerTests,
        ParamTestFixtureName,
        ::testing::Values(
                std::make_tuple(0, 1, true),
                std::make_tuple(1, 0, false),
                std::make_tuple(-5, -4, true));
```

GTest in ROS 2 – Mocks

- Robot software often needs to **talk to hardware** (motors, sensors, I/O boards).
- Plugging real devices in every time you test is **inconvenient** and **risky**.

→**Abstract** the hardware layer from your logic: **Dependency Injection Pattern** (code depends on **interfaces**, not concrete hardware classes.)

Approach	**Description**	**Pros / Cons**
Emulation	Build fake devices that simulate real behavior (using simulators!).	+ Realistic, - Costly to create/maintain
Mocking (via GMock)	Create mock classes that expect specific function calls.	+ Lightweight, + Quick to set up

GTest in ROS 2 – Mocks example

1. Start from the interface class:

Define an integrated umbrella used by the robot to protect itself from the rain.

```
class UmbrellaInterface {
  virtual ~UmbrellaInterface() {};
  virtual void openUmbrella() = 0;
  virtual void closeUmbrella() = 0;
  virtual bool isOpen() const = 0;
};
```

2. Pick a function that uses the interface:

It will open the umbrella when it's raining and close it when it's not.

```
class Umbrella : public UmbrellaInterface {
  public:
  void openUmbrella() override {
    ... Advanced science stuff that interacts with real
    umbrellas
  }
  void closeUmbrella() override { ... }
  bool isOpen() const oeverride { ... }
};

void manageUmbrella(bool its_raining, std::shared_ptr<
    UmbrellaInterface> umbrella) {
  if(its_raining && !umbrella->isOpen())
    umbrella->openUmbrella();
  if(!its_raining && umbrella->isOpen())
    umbrella->closeUmbrella();
}
```

GTest in ROS 2 – Mocks example

3. Create a mock of the Umbrella class

Fill in the macro MOCK_METHOD for each function using the signature "MOCK_METHOD(ReturnType, MethodName,(Args))"

```

#include "gmock/gmock.h"

class MockUmbrella : public UmbrellaInterface {
 public:
  MOCK_METHOD(void, openUmbrella, (), (override));
  MOCK_METHOD(void, closeUmbrella, (), (override));
  MOCK_METHOD(bool, isOpen, (), (const, override));
};
```

4. Write a test for the function:

The test verifies that, if it rains and the umbrella is not open, the function openUmbrella is called.

```

#include "mock_umbrella.hpp"
#include "headers/with/functions/we/want/to/test.hpp"
#include "gmock/gmock.h"
#include "gtest/gtest.h"

using ::testing::AtLeast;
using ::testing::Return;

TEST(UmbrellaTest, willOpenUmbrella) {
  auto umbrella = std::make_shared<MockUmbrella>();
  EXPECT_CALL(*umbrella, isOpen())
     .WillOnce(Return(false));
  EXPECT_CALL(*umbrella, openUmbrella())
      .Times(AtLeast(1));
  const bool its_raining{true};
  manageUmbrella(its_raining, umbrella);
}
```

Compile GTest with ROS 2

- Tests are usually added to a *test* folder within the package;
- Tests can be compiled by adding to the *CMakeLists.txt*:

```
if(BUILD_TESTING)
  find_package(ament_cmake_gtest REQUIRED)

  ament_add_gtest(${PROJECT_NAME}_tutorial_test test/tutorial_test.cpp)
  target_include_directories(${PROJECT_NAME}_tutorial_test PUBLIC
    $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
    $<INSTALL_INTERFACE:include>
  )
  target_link_libraries(${PROJECT_NAME}_tutorial_test name_of_local_library)
endif()
```

- If you are using GMocks in your test:

```
find_package(ament_gmock REQUIRED)
ament_add_gmock(${PROJECT_NAME}_mock_test test/mock_test.cpp)
```

- And to the *package.xml:*

```
<test_depend>ament_cmake_gtest</test_depend>
```

Compile GTest with ROS 2

CODE STYLE

- ROS 2 uses the Google C++ Style Guide, with some modifications (check the guidelines!)

- If you want to enforce the code to be formatted following the ROS 2 guidelines:

  CMakeLists.txt:

  ```
  find_package(ament_lint_auto REQUIRED)
  ament_lint_auto_find_test_dependencies()
  ```

  package.xml:

  ```
  <test_depend>ament_lint_auto</test_depend>
  <test_depend>ament_lint_common</test_depend>
  ```

Run Test in ROS 2

- Tests need to be run before committing chenges to the repo!
- To compile and run the tests:
  colcon test --ctest-args tests [package_selection_args]
- To examine the results:
  colcon test-result --all
  colcon test-result --all --verbose (to see test cases which fail)

- If a C++ test is failing, gdb can be used directly on the test executable in the build directory. First, clean the cache and rebuild the code in debug mode:
  colcon build --cmake-clean-cache --mixin debug
- Run the test directly through gdb (C++ debugger):
  gdb -ex run ./build/rcl/test/test_logging
- More info about backtraces and GDB with ROS2 are available in this guide.

Unit Test in ROS 2

- Examples (both in the yasmin and yasmin_ros folders): https://github.com/uleroboticsgroup/yasmin/tree/main

Component Testing in ROS 2

- Unit tests → check individual functions or classes.
- Component tests → check the correct behavior of a ROS 2 node as a whole:
  - Verify that a **node behaves as expected** when integrated into its ROS environment.
  - Focus on **external behavior** (topics, services, actions), not internal implementation details.

How It Works?

1. Launch the **node under test** with its parameters.
2. Replace dependent nodes with **mocks or stubs**.
3. Interact via **ROS interfaces** (publishers, services, actions).
4. Optionally, feed **recorded sensor data** (rosbags) to simulate the robot environment.

Recording and playing back data

A rosbag is a file that stores **ROS messages over time**.

- It allows you to **record**, **inspect**, and **replay** the data exchanged between ROS nodes.
- Essential for testing, debugging, and simulation.

Action	**Command**	**Purpose**

Record	`ros2 bag record --topics <topic_name>`
	`ros2 bag record -a`
	`ros2 bag record -a -o <bag_name>`
		Capture specific topics in the system
		Capture all topics in the system
		Save bag in specific folder

Play	`ros2 bag play <bag_name>` `ros2 bag play -l <bag_name>`	Replay recorded messages in real time Replay in a loop
Info	`ros2 bag info <bag_name>`	Inspect bag contents (topics, message types, duration)

Component Testing example

- Write a **launch** file that starts the node together with its environment:
- Alongside the node to be tested, we start another launchfile to publish the robot description, and a bagfile containing data previously recorded from the real robot.

```python
#! /usr/bin/env python3

import os
import sys
from ament_index_python import get_package_share_directory
from launch import LaunchDescription
from launch import LaunchService
from launch.actions import ExecuteProcess,
    IncludeLaunchDescription
from launch.launch_description_sources import
    AnyLaunchDescriptionSource
from launch_ros.actions import Node
from launch_testing.legacy import LaunchTestService


def generate_launch_description():
    # Assuming the bagfile to be installed in the shared
    folder
    bagfile = get_package_share_directory('a_package') + "/
    bags/" + "my_bag"
    return LaunchDescription([
        IncludeLaunchDescription(
            AnyLaunchDescriptionSource(
                os.path.join(
                    robot_description_pkg,
                    'launch/robot_description.launch.py'))
        ),
        Node(
            package='a_package',
            executable='name_of_the_exe_to_be_tested',
            name='test_node_name',
            output='screen'
        ),
        ExecuteProcess(
            cmd=['ros2', 'bag', 'play', bagfile,
                '-l', '--clock'
        )
    ])


def main(argv=sys.argv[1:]):
    ld = generate_launch_description()

    testExecutable = os.getenv('TEST_EXECUTABLE')

    first_test_action = ExecuteProcess(
        cmd=[testExecutable],
        name='your_test_name',
        output='screen')

    lts = LaunchTestService()
    lts.add_test_action(ld, first_test_action)
    ls = LaunchService(argv=argv)
    ls.include_launch_description(ld)
    return lts.run(ls)

if __name__ == '__main__':
    sys.exit(main())
```

Component Testing example

The content of the interchanged data can be compared to the expected one using the usual assertions provided by **Gtests**;

An action server residing on the tested node is called by an action client created in the test.

```cpp
#include <gtest/gtest.h>
#include <chrono>
#include <memory>
#include "my_msgs/action/my_action.hpp"
#include "rclcpp/rclcpp.hpp"
#include "rclcpp_action/rclcpp_action.hpp"

using namespace testing;
class MyTestFixture : public ::testing::Test {
 public:
  static void SetUpTestCase() { rclcpp::init(0, nullptr); }

  void SetUp() override {
    node_ = rclcpp::Node::make_shared("test_node");
    client_ptr_ = rclcpp_action::create_client<MyAction>(
    test_node_, "my_action");
  }

  void sendGoalToNode(const my_msgs::action::MyAction::Goal
    & goal, rclcpp_action::ClientGoalHandle<MyTestFixture::
    MyAction>::WrappedResult& result) {
    auto fut_response = client_ptr_->async_send_goal(
    goal_msg);
    ASSERT_TRUE(rclcpp::spin_until_future_complete(
    test_node_, fut_response) ==
                rclcpp::FutureReturnCode::SUCCESS);
    auto goal_handle = fut_response.get();
    auto fut_result = client_ptr_->async_get_result(
    goal_handle);
    ASSERT_TRUE(rclcpp::spin_until_future_complete(
    test_node_, fut_result) ==
                rclcpp::FutureReturnCode::SUCCESS);
    result = fut_result.get();
  }

  std::shared_ptr<rclcpp::Node> test_node_;
  rclcpp_action::Client<my_msgs::action::MyAction>::
    SharedPtr client_ptr_;
};

TEST_F(MyTestFixture, TestMyAction) {
  ASSERT_TRUE(client_ptr_->wait_for_action_server(5s));
  auto goal = my_msgs::action::MyAction::Goal();
  rclcpp_action::ClientGoalHandle<MyTestFixture::MyAction
    >::WrappedResult goal_result;
  sendGoalToNode(goal_msg, goal_result);
  EXPECT_TRUE(result->success);
}

int main(int argc, char** argv) {
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

Component Testing example

The same fixture class can of course be used to define multiple tests. These tests will then behave as unit tests, with the only difference that a node, executed by the previously defined launchfile, will be running in parallel to them. To achieve this, we need to add a couple of entries to our CMakeLists.txt

```
find_package(ament_cmake_gtest REQUIRED)
include(GoogleTest)
SET (TEST_NAME "name_of_your_test")

ament_add_gtest_executable(${TEST_NAME}
  test_file.cpp
)

target_link_libraries(${TEST_NAME}
  gtest_main
)

ament_target_dependencies(${TEST_NAME}
  rclcpp
)

ament_add_test(${TEST_NAME}
  GENERATE_RESULT_FOR_RETURN_CODE_ZERO
  COMMAND "${CMAKE_CURRENT_SOURCE_DIR}/launchfile_name.py"
  WORKING_DIRECTORY "${CMAKE_CURRENT_BINARY_DIR}"
  ENV
    TEST_DIR=${TEST_DIR}
    TEST_LAUNCH_DIR=${TEST_LAUNCH_DIR}
    TEST_EXECUTABLE=$<TARGET_FILE:${TEST_NAME}>
)
```

In case you need the gmock features, you might have to additionally add the following before ament_add_test:

```
find_package(ament_cmake_gmock REQUIRED)
_ament_cmake_gmock_find_gmock()

target_include_directories(${TEST_NAME} PUBLIC ${
    GMOCK_INCLUDE_DIRS})
target_link_libraries(${TEST_NAME}
  gtest_main
  gmock
)
```

Integration test

- Ensure that **multiple ROS 2 nodes** (a subsystem) can **work together correctly** to achieve a shared goal.
- Goes beyond component tests: checks **inter-node cooperation**, not single-node logic.

1. **Create a launch file**
   Start all nodes under test and their environment (parameters, rosbag, etc.).
2. **Write integration tests**
   Interact with one or more nodes via their ROS interfaces (topics, services, actions).
3. **Verify expected results**
   Check that joint behavior matches the desired system outcome.

Simulation-Based Test

- As systems grow larger, **integration tests** become complex and fragile. Simulators offer a way to test **whole robot systems** — without physical hardware or complex setup.
- The simulator models the **robot**, its **sensors**, and the **environment**.
- Nodes interact as if connected to real hardware.
- Predict robot behavior **accurately and safely.**
- Test in **diverse, repeatable environments** (even randomized).
- Integrate smoothly into **CI/CD pipelines.**

Testing on Hardware

Even with perfect simulations, **new issues often emerge** in the physical world.

Challenge	Description
Hardware can fail	Broken circuits, loose joints, scratched sensors: always verify the hardware first.
Communication latency	Real buses introduce unpredictable delays, critical for control loops.
Hardware degrades	Wear, drift, dirt, and calibration errors accumulate over time.
Sensor noise & artefacts	Shadows, reflections, sunlight, or interference can affect readings.
Unpredictable environments	Surfaces, objects, and lighting change, impacting robot performance.

Best Practices

- Test **hardware functionality** before software.
- Calibrate and log **latency and drift** regularly.
- Keep a **hardware checklist** for systematic verification.
- Use simulations and rosbags first, then move to hardware tests gradually.