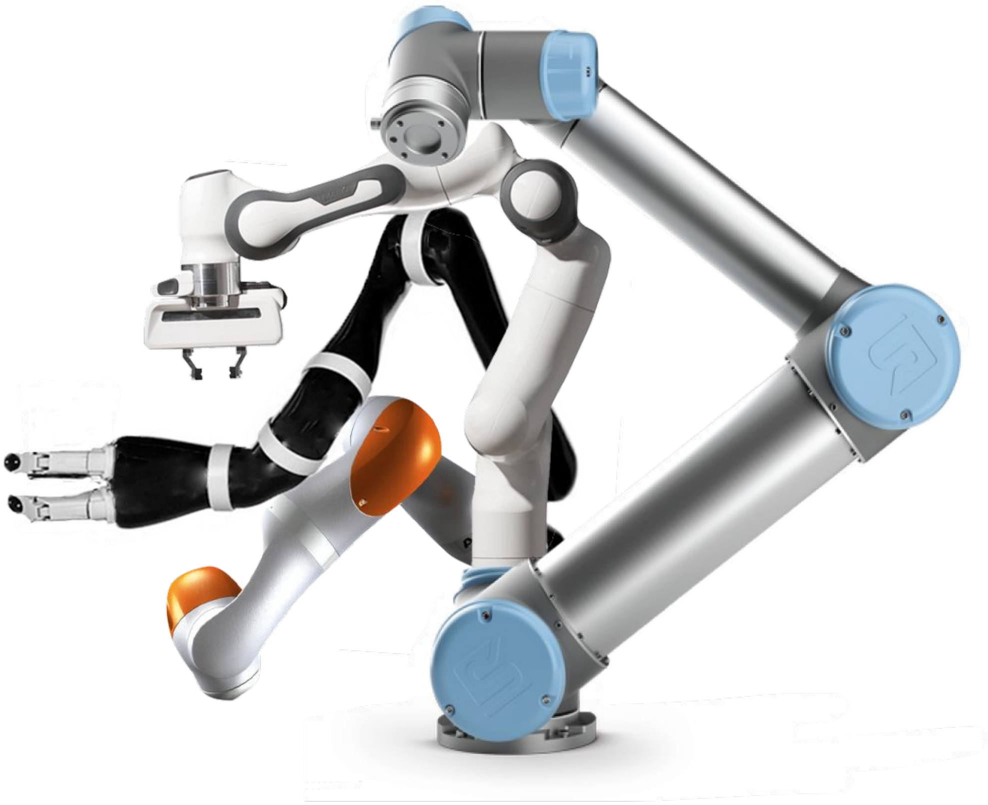
Leveraging a Functional Approach for More Testable and Maintainable ROS 2 Code

BILAL GILL

Cppcon
The C++ Conference
2023
October 01 - 06

Outline

- Introduction
- Overview of ROS 2
- ROS 2 Conventional Approach
- Introduction to Functional Programming Principles
- Refactoring using Functional Programming Principles
- Conclusion



Introduction

About Me

- Robotics Engineer on the services team at PickNik Robotics
  - Contributed to a wide variety of client projects: remotely operated underwater inspection vehicles, autonomous mobile base for agriculture applications, and more
- Have worked at General Dynamics Electric Boat, MIT Lincoln Laboratory
- Interested in robotics since high school

PICKNIK

About PickNik Robotics

- *The* **Unstructured** Robotics company
    - Unstructured: When the robot is required to perform tasks that are not predetermined or predefined in an environment that may have a variety of obstacles, objects, or events occurring

PICKNIK

About PickNik Robotics

- *The* **Unstructured** Robotics company
  - Unstructured: When the robot is required to perform tasks that are not predetermined or predefined in an environment that may have a variety of obstacles, objects, or events occurring
- Main maintainers of MoveIt
  - MoveIt is an open source robotics manipulation platform for developing commercial applications, prototyping designs, and benchmarking algorithms

PICKNIK

About PickNik Robotics

- *The* **Unstructured** Robotics company
    - Unstructured: When the robot is required to perform tasks that are not predetermined or predefined in an environment that may have a variety of obstacles, objects, or events occurring
- Main maintainers of MoveIt
    - MoveIt is an open source robotics manipulation platform for developing commercial applications, prototyping designs, and benchmarking algorithms
- Developing MoveIt Studio
    - MoveIt Studio is a developer tool and SDK that leverages MoveIt to make it easier to create robotic arm applications

PICKNIK

About PickNik Robotics

- *The* **Unstructured** Robotics company
  - Unstructured: When the robot is required to perform tasks that are not predetermined or predefined in an environment that may have a variety of obstacles, objects, or events occurring
- Main maintainers of MoveIt
  - MoveIt is an open source robotics manipulation platform for developing commercial applications, prototyping designs, and benchmarking algorithms
- Developing MoveIt Studio
  - MoveIt Studio is a developer tool and SDK that leverages MoveIt to make it easier to create robotic arm applications
- Provide consulting services to companies that range from performing feasibility studies to developing robotics software and more

PICKNIK

Why use Robot Operating System?

- Robot Operating System (ROS) is the de facto middleware of choice across robotics academia and industry

Clip taken from: https://robots.ros.org/

Why use Robot Operating System?

- Robot Operating System (ROS) is the de facto middleware of choice across robotics academia and industry
- According to the ROS 2022 Metrics Report, more than 740 companies use ROS!

Clip taken from: https://robots.ros.org/

Why use Robot Operating System?

- Robot Operating System (ROS) is the de facto middleware of choice across robotics academia and industry
- According to the ROS 2022 Metrics Report, more than 740 companies use ROS!
- Using ROS allows PickNik to leverage open source software to quickly develop code

Clip taken from: https://robots.ros.org/

PICKNIK

Why give this talk?

- Engineers at PickNik experiment with different ways to architect code that uses ROS 2 – creating code that is easy to understand, test, maintain, and extend

Why give this talk?

- Engineers at PickNik experiment with different ways to architect code that uses ROS 2 – creating code that is easy to understand, test, maintain, and extend
- At the end of a services project, code is handed over to the client

Why give this talk?

- Engineers at PickNik experiment with different ways to architect code that uses ROS 2 – creating code that is easy to understand, test, maintain, and extend
- At the end of a services project, code is handed over to the client
- How can the client expect proper operation of the software once they start developing on top of it?

Why give this talk?

- Engineers at PickNik experiment with different ways to architect code that uses ROS 2 – creating code that is easy to understand, test, maintain, and extend
- At the end of a services project, code is handed over to the client
- How can the client expect proper operation of the software once they start developing on top of it?

Answer: Tests and documentation! ***Lots and lots of documentation!***

PICKNIK

Why give this talk?

- Engineers at PickNik experiment with different ways to architect code that uses ROS 2 – creating code that is easy to understand, test, maintain, and extend
- At the end of a services project, code is handed over to the client
- How can the client expect proper operation of the software once they start developing on top of it?

Answer: Tests and documentation! ***Lots and lots of documentation!***

- ROS 2 documentation encourages an object-oriented paradigm that can lead to trouble writing code that achieves the goal

PICKNIK

Why give this talk?

- Engineers at PickNik experiment with different ways to architect code that uses ROS 2 – creating code that is easy to understand, test, maintain, and extend
- At the end of a services project, code is handed over to the client
- How can the client expect proper operation of the software once they start developing on top of it?

Answer: Tests and documentation! ***Lots and lots of documentation!***

- ROS 2 documentation encourages an object-oriented paradigm that can lead to trouble writing code that achieves the goal
- **Adopting functional programming techniques into our code has made it easier to test, maintain, and extend code!**

PICKNIK

Overview of ROS 2

PICKNIK

What is ROS 2?

- has a middleware layer that allows for message passing between different processes

What is ROS 2?

- has a middleware layer that allows for message passing between different processes
- systems are made up of nodes that can
  - publish data to topics, subscribe to topics to receive data, act as a service client, act as a service server, act as an action client, or act as an action server

What is ROS 2?

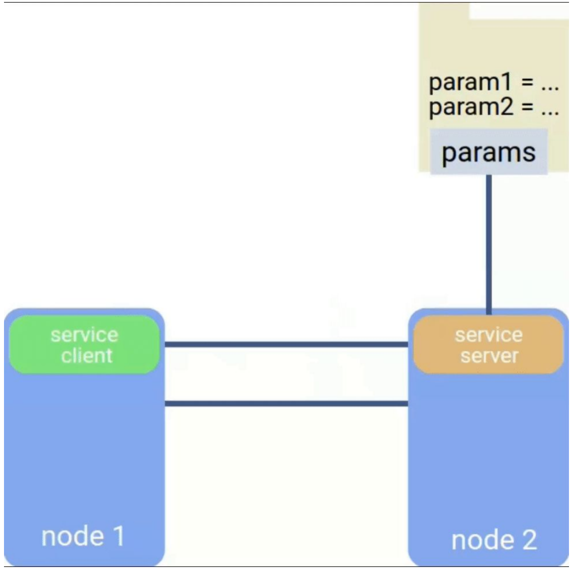
- has a middleware layer that allows for message passing between different processes
- systems are made up of nodes that can
  - publish data to topics, subscribe to topics to receive data, act as a service client, act as a service server, act as an action client, or act as an action server
  - provide configurable parameters which can be adjusted at run-time

What is ROS 2?

- has a middleware layer that allows for message passing between different processes
- systems are made up of nodes that can
  - publish data to topics, subscribe to topics to receive data, act as a service client, act as a service server, act as an action client, or act as an action server
  - provide configurable parameters which can be adjusted at run-time
  - log telemetry data that is useful for introspection

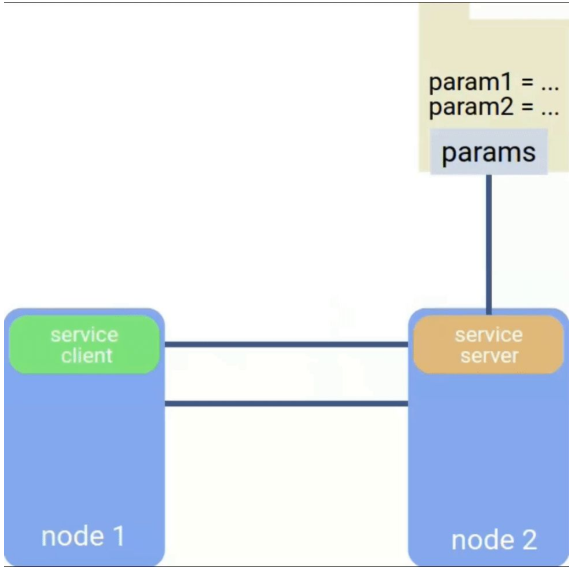
What is ROS 2?

- In this ROS 2 example, there are 2 nodes in the system



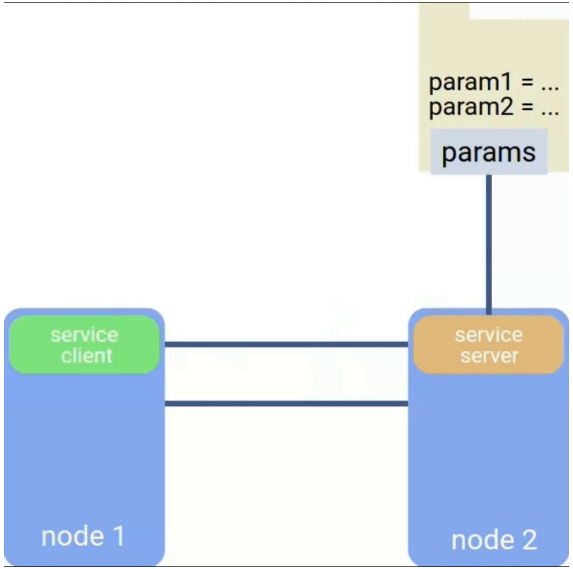
What is ROS 2?

- In this ROS 2 example, there are 2 nodes in the system
  - Node 1 acts as a service client and sends requests to Node 2



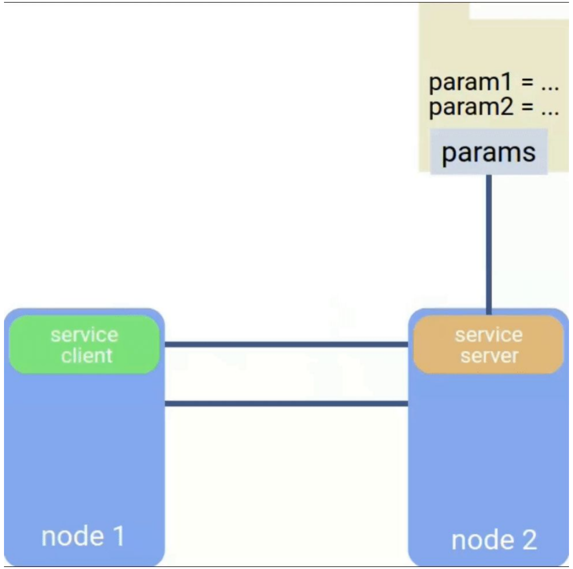
What is ROS 2?

- In this ROS 2 example, there are 2 nodes in the system
  - Node 1 acts as a service client and sends requests to Node 2
  - Node 2 acts as a service server, receives requests from Node 1, and sends back responses



What is ROS 2?

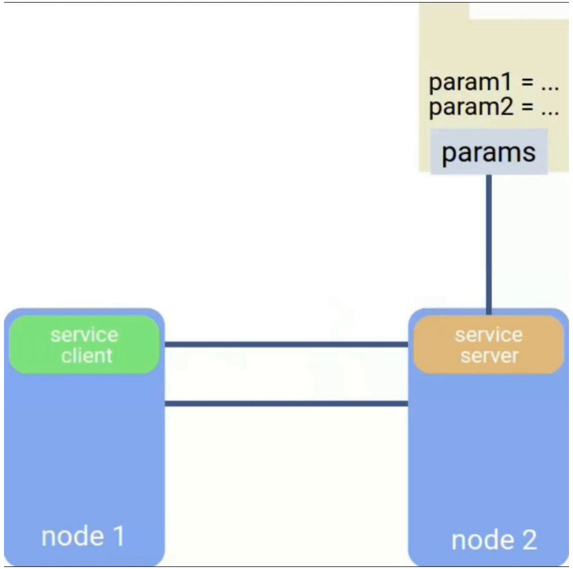
- In this ROS 2 example, there are 2 nodes in the system
  - Node 1 acts as a service client and sends requests to Node 2
  - Node 2 acts as a service server, receives requests from Node 1, and sends back responses
    - Node 2 also uses parameters at run-time to change its behavior



PICKNIK

What is ROS 2?

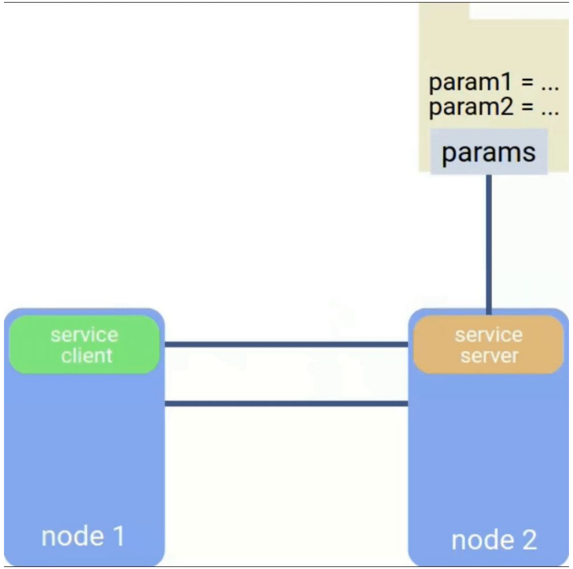
- In this ROS 2 example, there are 2 nodes in the system
  - Node 1 acts as a service client and sends requests to Node 2
  - Node 2 acts as a service server, receives requests from Node 1, and sends back responses
    - Node 2 also uses parameters at run-time to change its behavior
- Each node should be responsible for a single, modular purpose, (e.g. controlling the wheel motors or publishing the sensor data from a laser range-finder)



PICKNIK

What is ROS 2?

- In this ROS 2 example, there are 2 nodes in the system
  - Node 1 acts as a service client and sends requests to Node 2
  - Node 2 acts as a service server, receives requests from Node 1, and sends back responses
    - Node 2 also uses parameters at run-time to change its behavior
- Each node should be responsible for a single, modular purpose, (e.g. controlling the wheel motors or publishing the sensor data from a laser range-finder)
- The publishing/subscribing of data and service requests is done via the ROS 2 API



PICKNIK

Motivating Example

- Problem: A robot wants to navigate from its current location to some goal

Clip from https://www.youtube.com/watch?v=VTeY-l-Xh6c

Motivating Example

- Problem: A robot wants to navigate from its current location to some goal
- The robot needs to know where obstacles are located in its environment

Clip from https://www.youtube.com/watch?v=VTeY-l-Xh6c

Motivating Example

- Problem: A robot wants to navigate from its current location to some goal
- The robot needs to know where obstacles are located in its environment
- Enter: The occupancy map
    - A data structure used to represent the environment around a robot in terms of how “occupied” the cells in the map are

Clip from https://www.youtube.com/watch?v=VTeY-l-Xh6c

Motivating Example

- Problem: A robot wants to navigate from its current location to some goal
- The robot needs to know where obstacles are located in its environment
- Enter: The occupancy map
  - A data structure used to represent the environment around a robot in terms of how "occupied" the cells in the map are
- Assumption: The robot knows its location in the occupancy map at all times

Clip from https://www.youtube.com/watch?v=VTeY-l-Xh6c

PICKNIK

Motivating Example

- Problem: A robot wants to navigate from its current location to some goal
- The robot needs to know where obstacles are located in its environment
- Enter: The occupancy map
  - A data structure used to represent the environment around a robot in terms of how "occupied" the cells in the map are
- Assumption: The robot knows its location in the occupancy map at all times
- Solution: The robot will send a request to a ROS 2 service that generates a path from the robot's current location and goal location, given an occupancy map

Clip from https://www.youtube.com/watch?v=VTeY-I-Xh6c

PICKNIK

ROS 2 Conventional Approach

Conventional Approach

```cpp
class PathGenerator : public rclcpp::Node {
 public:
  explicit PathGenerator(
          rclcpp::NodeOptions const& options = rclcpp::NodeOptions{})
           : Node("path_generator", options);

 private:
  void set_map_service(
       const std::shared_ptr<example_srvs::srv::SetMap::Request> request,
       std::shared_ptr<example_srvs::srv::SetMap::Response> response);

  void generate_path_service(
       const std::shared_ptr<example_srvs::srv::GetPath::Request> request,
       std::shared_ptr<example_srvs::srv::GetPath::Response> response);

  bool set_costmap(const std_msgs::msg::UInt8MultiArray& costmap);

  Path generate_global_path(Position const& start,Position const& goal);

  Map<unsigned char> map_;
  int robot_size_;
  std::unique_ptr<CollisionChecker<unsigned char>> is_occupied_;
  rclcpp::Service<example_srvs::srv::SetMap>::SharedPtr map_setter_service_;
  rclcpp::Service<example_srvs::srv::GetPath>::SharedPtr path_generator_service_;
};
```

- `PathGenerator` will be used to generate the path for our robot
- This code was written using example code available from the ROS 2 documentation
- This implementation follows an object oriented approach

PICKNIK

Conventional Approach

```cpp
class PathGenerator : public rclcpp::Node {
 public:
  explicit PathGenerator(
         rclcpp::NodeOptions const& options = rclcpp::NodeOptions{})
          : Node("path_generator", options);

 private:
  void set_map_service(
      const std::shared_ptr<example_srvs::srv::SetMap::Request> request,
      std::shared_ptr<example_srvs::srv::SetMap::Response> response);

  void generate_path_service(
      const std::shared_ptr<example_srvs::srv::GetPath::Request> request,
      std::shared_ptr<example_srvs::srv::GetPath::Response> response);

  bool set_costmap(const std_msgs::msg::UInt8MultiArray& costmap);

  Path generate_global_path(Position const& start,Position const& goal);

  Map<unsigned char> map_;
  int robot_size_;
  std::unique_ptr<CollisionChecker<unsigned char>> is_occupied_;
  rclcpp::Service<example_srvs::srv::SetMap>::SharedPtr map_setter_service_;
  rclcpp::Service<example_srvs::srv::GetPath>::SharedPtr path_generator_service_;
};
```

- `PathGenerator` inherits from `rclcpp::Node` making it inextricably linked to the ROS 2 API

Conventional Approach

```cpp
class PathGenerator : public rclcpp::Node {
 public:
  explicit PathGenerator(
          rclcpp::NodeOptions const& options = rclcpp::NodeOptions{})
           : Node("path_generator", options);

 private:
  void set_map_service(
      const std::shared_ptr<example_srvs::srv::SetMap::Request> request,
      std::shared_ptr<example_srvs::srv::SetMap::Response> response);

  void generate_path_service(
      const std::shared_ptr<example_srvs::srv::GetPath::Request> request,
      std::shared_ptr<example_srvs::srv::GetPath::Response> response);

  bool set_costmap(const std_msgs::msg::UInt8MultiArray& costmap);

  Path generate_global_path(Position const& start,Position const& goal);

  Map<unsigned char> map_;
  int robot_size_;
  std::unique_ptr<CollisionChecker<unsigned char>> is_occupied_;
  rclcpp::Service<example_srvs::srv::SetMap>::SharedPtr map_setter_service_;
  rclcpp::Service<example_srvs::srv::GetPath>::SharedPtr path_generator_service_;
};
```

- `PathGenerator` inherits from `rclcpp::Node` making it inextricably linked to the ROS 2 API
- `set_map_service` and `generate_path_service` are callback functions that run when requests are sent via the ROS 2 middleware

PICKNIK

Conventional Approach

```cpp
class PathGenerator : public rclcpp::Node {
 public:
  explicit PathGenerator(
          rclcpp::NodeOptions const& options = rclcpp::NodeOptions{})
           : Node("path_generator", options);

 private:
  void set_map_service(
       const std::shared_ptr<example_srvs::srv::SetMap::Request> request,
       std::shared_ptr<example_srvs::srv::SetMap::Response> response);

  void generate_path_service(
       const std::shared_ptr<example_srvs::srv::GetPath::Request> request,
       std::shared_ptr<example_srvs::srv::GetPath::Response> response);

  bool set_costmap(const std_msgs::msg::UInt8MultiArray& costmap);

  Path generate_global_path(Position const& start,Position const& goal);

  Map<unsigned char> map_;
  int robot_size_;
  std::unique_ptr<CollisionChecker<unsigned char>> is_occupied_;
  rclcpp::Service<example_srvs::srv::SetMap>::SharedPtr map_setter_service_;
  rclcpp::Service<example_srvs::srv::GetPath>::SharedPtr path_generator_service_;
};
```

- `PathGenerator` inherits from `rclcpp::Node` making it inextricably linked to the ROS 2 API
- `set_map_service` and `generate_path_service` are callback functions that run when requests are sent via the ROS 2 middleware
- Those functions call the `set_costmap` and `generate_global_path` functions, which are private functions that contain the actual business logic

Conventional Approach

```
class PathGenerator : public rclcpp::Node {
 public:
  explicit PathGenerator(
          rclcpp::NodeOptions const& options = rclcpp::NodeOptions{})
           : Node("path_generator", options);

 private:
  void set_map_service(
       const std::shared_ptr<example_srvs::srv::SetMap::Request> request,
       std::shared_ptr<example_srvs::srv::SetMap::Response> response);

  void generate_path_service(
       const std::shared_ptr<example_srvs::srv::GetPath::Request> request,
       std::shared_ptr<example_srvs::srv::GetPath::Response> response);

  bool set_costmap(const std_msgs::msg::UInt8MultiArray& costmap);

  Path generate_global_path(Position const& start,Position const& goal);

  Map<unsigned char> map_;
  int robot_size_;
  std::unique_ptr<CollisionChecker<unsigned char>> is_occupied_;
  rclcpp::Service<example_srvs::srv::SetMap>::SharedPtr map_setter_service_;
  rclcpp::Service<example_srvs::srv::GetPath>::SharedPtr path_generator_service_;
};
```

- `PathGenerator` inherits from `rclcpp::Node` making it inextricably linked to the ROS 2 API
- `set_map_service` and `generate_path_service` are callback functions that run when requests are sent via the ROS 2 middleware
- Those functions call the `set_costmap` and `generate_global_path` functions, which are private functions that contain the actual business logic
- Let's take a look at the `PathGenerator` constructor

Conventional Approach

```cpp
class PathGenerator : public rclcpp::Node {
 public:
  explicit PathGenerator(rclcpp::NodeOptions const& options = rclcpp::NodeOptions{})
          : Node("path_generator", options) {
    robot_size_ = this->declare_parameter<int>("robot_size", 1);
    is_occupied_ = std::make_unique<CollisionChecker<unsigned char>>(robot_size_);

    // Services for setting the map and generating the path
    map_setter_service_ =
      this->create_service<example_srvs::srv::SetMap>("set_costmap",
      std::bind(&PathGenerator::set_map_service, this,
      std::placeholders::_1, std::placeholders::_2));
    path_generator_service_ =
      this->create_service<example_srvs::srv::GetPath>("generate_global_path",
      std::bind(&PathGenerator::generate_path_service, this,
      std::placeholders::_1, std::placeholders::_2));
  }

 private:
  void set_map_service(
        const std::shared_ptr<example_srvs::srv::SetMap::Request> request,
        std::shared_ptr<example_srvs::srv::SetMap::Response> response);

  void generate_path_service(
        const std::shared_ptr<example_srvs::srv::GetPath::Request> request,
        std::shared_ptr<example_srvs::srv::GetPath::Response> response);

  /* Additional private methods and members */
};
```

- `robot_size_` is a parameter that is required to construct a `CollisionChecker` object
- It is common in ROS to fetch parameters in the constructor of the Node
- This leads to the pattern of dynamically allocating the object because it cannot be initialized in the initializer list of the class

Conventional Approach

```
class PathGenerator : public rclcpp::Node {
 public:
  explicit PathGenerator(rclcpp::NodeOptions const& options = rclcpp::NodeOptions{})
          : Node("path_generator", options) {
    robot_size_ = this->declare_parameter<int>("robot_size", 1);
    is_occupied_ = std::make_unique<CollisionChecker<unsigned char>>(robot_size_);

    // Services for setting the map and generating the path
    map_setter_service_ =
      this->create_service<example_srvs::srv::SetMap>("set_costmap",
      std::bind(&PathGenerator::set_map_service, this,
      std::placeholders::_1, std::placeholders::_2));
    path_generator_service_ =
      this->create_service<example_srvs::srv::GetPath>("generate_global_path",
      std::bind(&PathGenerator::generate_path_service, this,
      std::placeholders::_1, std::placeholders::_2));
  }

 private:
  void set_map_service(
        const std::shared_ptr<example_srvs::srv::SetMap::Request> request,
        std::shared_ptr<example_srvs::srv::SetMap::Response> response);

  void generate_path_service(
        const std::shared_ptr<example_srvs::srv::GetPath::Request> request,
        std::shared_ptr<example_srvs::srv::GetPath::Response> response);

  /* Additional private methods and members */
};
```

- `robot_size_` is a parameter that is required to construct a `CollisionChecker` object
- It is common in ROS to fetch parameters in the constructor of the Node
- This leads to the pattern of dynamically allocating the object because it cannot be initialized in the initializer list of the class
- `map_setter_service_` and `path_generator_service_` are two servers that execute the `set_map_service` and `generate_path_service` as callbacks

Conventional Approach

```
class PathGenerator : public rclcpp::Node {
 public:
  explicit PathGenerator(rclcpp::NodeOptions const& options = rclcpp::NodeOptions{})
          : Node("path_generator", options) {
    robot_size_ = this->declare_parameter<int>("robot_size", 1);
    is_occupied_ = std::make_unique<CollisionChecker<unsigned char>>(robot_size_);

    // Services for setting the map and generating the path
    map_setter_service_ =
      this->create_service<example_srvs::srv::SetMap>("set_costmap",
      std::bind(&PathGenerator::set_map_service, this,
      std::placeholders::_1, std::placeholders::_2));
    path_generator_service_ =
      this->create_service<example_srvs::srv::GetPath>("generate_global_path",
      std::bind(&PathGenerator::generate_path_service, this,
      std::placeholders::_1, std::placeholders::_2));
  }

 private:
  void set_map_service(
        const std::shared_ptr<example_srvs::srv::SetMap::Request> request,
        std::shared_ptr<example_srvs::srv::SetMap::Response> response);

  void generate_path_service(
        const std::shared_ptr<example_srvs::srv::GetPath::Request> request,
        std::shared_ptr<example_srvs::srv::GetPath::Response> response);

  /* Additional private methods and members */
};
```

- `robot_size_` is a parameter that is required to construct a `CollisionChecker` object
- It is common in ROS to fetch parameters in the constructor of the Node
- This leads to the pattern of dynamically allocating the object because it cannot be initialized in the initializer list of the class
- `map_setter_service_` and `path_generator_service_` are two servers that execute the `set_map_service` and `generate_path_service` as callbacks
- Let's take a closer look at `generate_path_service`

Conventional Approach

```
void generate_path_service(
const std::shared_ptr<example_srvs::srv::GetPath::Request> request,
      std::shared_ptr<example_srvs::srv::GetPath::Response> response) {
  if (map_.get_data().size() == 0) {
    RCLCPP_ERROR_STREAM(this->get_logger(), "MAP IS EMPTY!!");
    response->code.code = example_srvs::msg::GetPathCodes::EMPTY_OCCUPANCY_MAP;
    response->path = std_msgs::msg::UInt8MultiArray();
    return;
  }
  /* More error pre-checks */

  auto const start = Position{request->start.data[0], request->start.data[1]};
  auto const goal = Position{request->goal.data[0], request->goal.data[1]};

  // Generate the path
  auto const path = generate_global_path(start, goal);

  // Start populating the response message
  auto response_path = std_msgs::msg::UInt8MultiArray();

  /* Code about populating the message here */

  response->code.code = !path.empty() ? example_srvs::msg::GetPathCodes::SUCCESS :
example_srvs::msg::GetPathCodes::NO_VALID_PATH;
    response->path = response_path;
  }
```

PICKNIK

Conventional Approach

```
void generate_path_service(
const std::shared_ptr<example_srvs::srv::GetPath::Request> request,
      std::shared_ptr<example_srvs::srv::GetPath::Response> response) {
  if (map_.get_data().size() == 0) {
    RCLCPP_ERROR_STREAM(this->get_logger(), "MAP IS EMPTY!!");
    response->code.code = example_srvs::msg::GetPathCodes::EMPTY_OCCUPANCY_MAP;
    response->path = std_msgs::msg::UInt8MultiArray();
    return;
  }
  /* More error pre-checks */

  auto const start = Position{request->start.data[0], request->start.data[1]};
  auto const goal = Position{request->goal.data[0], request->goal.data[1]};

  // Generate the path
  auto const path = generate_global_path(start, goal);

  // Start populating the response message
  auto response_path = std_msgs::msg::UInt8MultiArray();

  /* Code about populating the message here */

  response->code.code = !path.empty() ? example_srvs::msg::GetPathCodes::SUCCESS :
example_srvs::msg::GetPathCodes::NO_VALID_PATH;
    response->path = response_path;
  }
```

- `response` is an out parameter that is set in the function

Conventional Approach

```
void generate_path_service(
const std::shared_ptr<example_srvs::srv::GetPath::Request> request,
      std::shared_ptr<example_srvs::srv::GetPath::Response> response) {
  if (map_.get_data().size() == 0) {
    RCLCPP_ERROR_STREAM(this->get_logger(), "MAP IS EMPTY!!");
    response->code.code = example_srvs::msg::GetPathCodes::EMPTY_OCCUPANCY_MAP;
    response->path = std_msgs::msg::UInt8MultiArray();
    return;
  }
  /* More error pre-checks */

  auto const start = Position{request->start.data[0], request->start.data[1]};
  auto const goal = Position{request->goal.data[0], request->goal.data[1]};

  // Generate the path
  auto const path = generate_global_path(start, goal);

  // Start populating the response message
  auto response_path = std_msgs::msg::UInt8MultiArray();

  /* Code about populating the message here */

  response->code.code = !path.empty() ? example_srvs::msg::GetPathCodes::SUCCESS :
example_srvs::msg::GetPathCodes::NO_VALID_PATH;
    response->path = response_path;
  }
```

- response is an out parameter that is set in the function
- Error handling is done both by printing to logs and returning an error code via the service response

Conventional Approach

```
void generate_path_service(
const std::shared_ptr<example_srvs::srv::GetPath::Request> request,
      std::shared_ptr<example_srvs::srv::GetPath::Response> response) {
  if (map_.get_data().size() == 0) {
    RCLCPP_ERROR_STREAM(this->get_logger(), "MAP IS EMPTY!!");
    response->code.code = example_srvs::msg::GetPathCodes::EMPTY_OCCUPANCY_MAP;
    response->path = std_msgs::msg::UInt8MultiArray();
    return;
  }
  /* More error pre-checks */

  auto const start = Position{request->start.data[0], request->start.data[1]};
  auto const goal = Position{request->goal.data[0], request->goal.data[1]};

  // Generate the path
  auto const path = generate_global_path(start, goal);

  // Start populating the response message
  auto response_path = std_msgs::msg::UInt8MultiArray();

  /* Code about populating the message here */

  response->code.code = !path.empty() ? example_srvs::msg::GetPathCodes::SUCCESS :
example_srvs::msg::GetPathCodes::NO_VALID_PATH;
    response->path = response_path;
  }
```

- `response` is an out parameter that is set in the function
- Error handling is done both by printing to logs and returning an error code via the service response
- `generate_global_path` is the function that generates the path and cannot be tested directly, since it is a private function
- The occupancy map used by generate_global_path is a private member variable, tying this algorithm to the class

PICKNIK

Limitations of the Conventional Approach

- Tight coupling between the path generating algorithm and the runtime API
  - By inheriting from rclcpp::Node, the PathGenerator is tightly coupled with the ROS 2 API
  - Testing the code is challenging without involving ROS 2 specifics

Limitations of the Conventional Approach

- Tight coupling between the path generating algorithm and the runtime API
  - By inheriting from rclcpp::Node, the PathGenerator is tightly coupled with the ROS 2 API
  - Testing the code is challenging without involving ROS 2 specifics
- The PathGenerator class is doing multiple things: managing ROS 2 communication, performing calculations, and implementing logic
  - Doesn't follow Separation of Concerns and can even be considered to violate the Single Responsibility Principle

Limitations of the Conventional Approach

- Tight coupling between the path generating algorithm and the runtime API
  - By inheriting from rclcpp::Node, the PathGenerator is tightly coupled with the ROS 2 API
  - Testing the code is challenging without involving ROS 2 specifics
- The PathGenerator class is doing multiple things: managing ROS 2 communication, performing calculations, and implementing logic
  - Doesn't follow Separation of Concerns and can even be considered to violate the Single Responsibility Principle
- Inflexibility of extensions
  - Implementing more features or handling more types of services will cause the class to grow quickly
  - Data are private variables, which causes the algorithms to be coupled with the class

PICKNIK

Testing the Conventional Approach

```
class TaskPlanningFixture : public testing::Test {
 protected:
  // Adapted from minimal_integration_test
  TaskPlanningFixture() : node_(std::make_shared<rclcpp::Node>("test_client")) {
    // Create ROS2 clients to set the map and calculate the path
    map_setter_client_ =
node_->create_client<example_srvs::srv::SetMap>("set_costmap");

    path_generator_client_ =
node_->create_client<example_srvs::srv::GetPath>("generate_global_path");
  }

  rclcpp::FutureReturnCode populateAndSetMap();

  std::pair<example_srvs::srv::GetPath::Response::SharedPtr,rclcpp::FutureReturnCode>
  sendPathRequest(const example_srvs::srv::GetPath::Request::SharedPtr request);

  // Member variables
  rclcpp::Node::SharedPtr node_;

  rclcpp::Client<example_srvs::srv::SetMap>::SharedPtr map_setter_client_;
  rclcpp::Client<example_srvs::srv::GetPath>::SharedPtr path_generator_client_;
};
```

PICKNIK

Testing the Conventional Approach

```
class TaskPlanningFixture : public testing::Test {
 protected:
  // Adapted from minimal_integration_test
  TaskPlanningFixture() : node_(std::make_shared<rclcpp::Node>("test_client")) {
    // Create ROS2 clients to set the map and calculate the path
    map_setter_client_ =
node_->create_client<example_srvs::srv::SetMap>("set_costmap");

    path_generator_client_ =
node_->create_client<example_srvs::srv::GetPath>("generate_global_path");
  }

  rclcpp::FutureReturnCode populateAndSetMap();

  std::pair<example_srvs::srv::GetPath::Response::SharedPtr,rclcpp::FutureReturnCode>
  sendPathRequest(const example_srvs::srv::GetPath::Request::SharedPtr request);

  // Member variables
  rclcpp::Node::SharedPtr node_;

  rclcpp::Client<example_srvs::srv::SetMap>::SharedPtr map_setter_client_;
  rclcpp::Client<example_srvs::srv::GetPath>::SharedPtr path_generator_client_;
};
```

- Testing the conventional approach requires creating run-time clients to send requests to the PathGenerator service

PICKNIK

Testing the Conventional Approach

```
class TaskPlanningFixture : public testing::Test {
 protected:
  // Adapted from minimal_integration_test
  TaskPlanningFixture() : node_(std::make_shared<rclcpp::Node>("test_client")) {
    // Create ROS2 clients to set the map and calculate the path
    map_setter_client_ =
node_->create_client<example_srvs::srv::SetMap>("set_costmap");

    path_generator_client_ =
node_->create_client<example_srvs::srv::GetPath>("generate_global_path");
  }

  rclcpp::FutureReturnCode populateAndSetMap();

  std::pair<example_srvs::srv::GetPath::Response::SharedPtr,rclcpp::FutureReturnCode>
  sendPathRequest(const example_srvs::srv::GetPath::Request::SharedPtr request);

  // Member variables
  rclcpp::Node::SharedPtr node_;

  rclcpp::Client<example_srvs::srv::SetMap>::SharedPtr map_setter_client_;
  rclcpp::Client<example_srvs::srv::GetPath>::SharedPtr path_generator_client_;
};
```

- Testing the conventional approach requires creating run-time clients to send requests to the PathGenerator service
- Additional utility functions are needed to keep tests concise
- This is additional code and logic also invokes the run-time environment

PICKNIK

Testing the Conventional Approach

```
class TaskPlanningFixture : public testing::Test {
 protected:
  // Adapted from minimal_integration_test
  TaskPlanningFixture() : node_(std::make_shared<rclcpp::Node>("test_client")) {
    // Create ROS2 clients to set the map and calculate the path
    map_setter_client_ =
node_->create_client<example_srvs::srv::SetMap>("set_costmap");

    path_generator_client_ =
node_->create_client<example_srvs::srv::GetPath>("generate_global_path");
  }

  rclcpp::FutureReturnCode populateAndSetMap();

  std::pair<example_srvs::srv::GetPath::Response::SharedPtr,rclcpp::FutureReturnCode>
  sendPathRequest(const example_srvs::srv::GetPath::Request::SharedPtr request);

  // Member variables
  rclcpp::Node::SharedPtr node_;

  rclcpp::Client<example_srvs::srv::SetMap>::SharedPtr map_setter_client_;
  rclcpp::Client<example_srvs::srv::GetPath>::SharedPtr path_generator_client_;
};
```

- Note the `rclcpp::Node` member variable
- This is another common pattern used to create an interface with the ROS 2 API (as opposed to inheriting from `rclcpp::Node`)

PICKNIK

Testing the Conventional Approach

```
class TaskPlanningFixture : public testing::Test {
 protected:
  // Adapted from minimal_integration_test
  TaskPlanningFixture() : node_(std::make_shared<rclcpp::Node>("test_client")) {
    // Create ROS2 clients to set the map and calculate the path
    map_setter_client_ =
node_->create_client<example_srvs::srv::SetMap>("set_costmap");

    path_generator_client_ =
node_->create_client<example_srvs::srv::GetPath>("generate_global_path");
  }

  rclcpp::FutureReturnCode populateAndSetMap();

  std::pair<example_srvs::srv::GetPath::Response::SharedPtr,rclcpp::FutureReturnCode>
  sendPathRequest(const example_srvs::srv::GetPath::Request::SharedPtr request);

  // Member variables
  rclcpp::Node::SharedPtr node_;

  rclcpp::Client<example_srvs::srv::SetMap>::SharedPtr map_setter_client_;
  rclcpp::Client<example_srvs::srv::GetPath>::SharedPtr path_generator_client_;
};
```

- Note the `rclcpp::Node` member variable
- This is another common pattern used to create an interface with the ROS 2 API (as opposed to inheriting from `rclcpp::Node`)
- Let's take a closer look at `sendPathRequest`, a utility function used in tests

PICKNIK

Testing the Conventional Approach

```
std::pair<example_srvs::srv::GetPath::Response::SharedPtr, rclcpp::FutureReturnCode>
sendPathRequest( const example_srvs::srv::GetPath::Request::SharedPtr request) {
  while (!path_generator_client_->wait_for_service(1s)) {
    if (!rclcpp::ok()) {
      RCLCPP_ERROR_STREAM(
          node_->get_logger(),
          "Interrupted while waiting for path generator service. Exiting.");
      return {std::make_shared<example_srvs::srv::GetPath::Response>(),
              rclcpp::FutureReturnCode::TIMEOUT};
    }
    RCLCPP_INFO_STREAM(
        node_->get_logger(),
        "Path generator service not available, waiting again...");
  }

  auto generate_path_result = path_generator_client_->async_send_request(request);

  return std::make_pair(generate_path_result.get(),
      rclcpp::spin_until_future_complete(node_, generate_path_result));
}
```

PICKNIK

Testing the Conventional Approach

```
std::pair<example_srvs::srv::GetPath::Response::SharedPtr, rclcpp::FutureReturnCode>
sendPathRequest( const example_srvs::srv::GetPath::Request::SharedPtr request) {
  while (!path_generator_client_->wait_for_service(1s)) {
    if (!rclcpp::ok()) {
      RCLCPP_ERROR_STREAM(
          node_->get_logger(),
          "Interrupted while waiting for path generator service. Exiting.");
      return {std::make_shared<example_srvs::srv::GetPath::Response>(),
              rclcpp::FutureReturnCode::TIMEOUT};
    }
    RCLCPP_INFO_STREAM(
        node_->get_logger(),
        "Path generator service not available, waiting again...");
  }

  auto generate_path_result = path_generator_client_->async_send_request(request);

  return std::make_pair(generate_path_result.get(),
      rclcpp::spin_until_future_complete(node_, generate_path_result));
}
```

- There is a loop that waits for the service to become available

PICKNIK

Testing the Conventional Approach

```
std::pair<example_srvs::srv::GetPath::Response::SharedPtr, rclcpp::FutureReturnCode>
sendPathRequest( const example_srvs::srv::GetPath::Request::SharedPtr request) {
  while (!path_generator_client_->wait_for_service(1s)) {
    if (!rclcpp::ok()) {
      RCLCPP_ERROR_STREAM(
          node_->get_logger(),
          "Interrupted while waiting for path generator service. Exiting.");
      return {std::make_shared<example_srvs::srv::GetPath::Response>(),
              rclcpp::FutureReturnCode::TIMEOUT};
    }
    RCLCPP_INFO_STREAM(
        node_->get_logger(),
        "Path generator service not available, waiting again...");
  }

  auto generate_path_result = path_generator_client_->async_send_request(request);

  return std::make_pair(generate_path_result.get(),
      rclcpp::spin_until_future_complete(node_, generate_path_result));
}
```

- There is a loop that waits for the service to become available
- `sendPathRequest` sends an asynchronous request to the `generate_global_path` service

PICKNIK

Testing the Conventional Approach

```
std::pair<example_srvs::srv::GetPath::Response::SharedPtr, rclcpp::FutureReturnCode>
sendPathRequest( const example_srvs::srv::GetPath::Request::SharedPtr request) {
  while (!path_generator_client_->wait_for_service(1s)) {
    if (!rclcpp::ok()) {
      RCLCPP_ERROR_STREAM(
          node_->get_logger(),
          "Interrupted while waiting for path generator service. Exiting.");
      return {std::make_shared<example_srvs::srv::GetPath::Response>(),
              rclcpp::FutureReturnCode::TIMEOUT};
    }
    RCLCPP_INFO_STREAM(
        node_->get_logger(),
        "Path generator service not available, waiting again...");
  }

  auto generate_path_result = path_generator_client_->async_send_request(request);

  return std::make_pair(generate_path_result.get(),
      rclcpp::spin_until_future_complete(node_, generate_path_result));
}
```

- There is a loop that waits for the service to come available
- `sendPathRequest` sends an asynchronous request to the `generate_global_path` service
- **Involving the middleware into the testing process is where flakiness can be introduced**

PICKNIK

Testing the Conventional Approach

```
std::pair<example_srvs::srv::GetPath::Response::SharedPtr, rclcpp::FutureReturnCode>
sendPathRequest( const example_srvs::srv::GetPath::Request::SharedPtr request) {
  while (!path_generator_client_->wait_for_service(1s)) {
    if (!rclcpp::ok()) {
      RCLCPP_ERROR_STREAM(
          node_->get_logger(),
          "Interrupted while waiting for path generator service. Exiting.");
      return {std::make_shared<example_srvs::srv::GetPath::Response>(),
              rclcpp::FutureReturnCode::TIMEOUT};
    }
    RCLCPP_INFO_STREAM(
        node_->get_logger(),
        "Path generator service not available, waiting again...");
  }

  auto generate_path_result = path_generator_client_->async_send_request(request);

  return std::make_pair(generate_path_result.get(),
      rclcpp::spin_until_future_complete(node_, generate_path_result));
}
```

- There is a loop that waits for the service to come available
- `sendPathRequest` sends an asynchronous request to the `generate_global_path` service
- **Involving the middleware into the testing process is where flakiness can be introduced**
- **When unit testing core logic, inter-process communication should be avoided**
- Let's look at a test

Testing the Conventional Approach

```
TEST_F(TaskPlanningFixture, no_path) {
  auto executor = std::make_shared<rclcpp::executors::SingleThreadedExecutor>();
  std::thread executor_thread;

  auto const pg = std::make_shared<PathGenerator>();

  executor->add_node(pg);
  executor_thread = std::thread([&executor]() { executor->spin(); });

  // GIVEN a populated costmap that is set without error
  auto const return_code = populateAndSetMap();

  EXPECT_EQ(return_code, rclcpp::FutureReturnCode::SUCCESS)
      << "Setting the map failed";

  // WHEN a path is requested between two positions that do not have a valid
  // path between them given the algorithm
  auto const request = std::make_shared<example_srvs::srv::GetPath::Request>();

  request->start.data = {2, 2};
  request->goal.data = {5, 5};

  auto const result = sendPathRequest(request);

  EXPECT_EQ(result.second, rclcpp::FutureReturnCode::SUCCESS) << "Generating path
failed";

  // THEN the global path produced should be empty
  std::vector<Position> const expected{};
  EXPECT_EQ(result.first->code.code,example_srvs::msg::GetPathCodes::NO_VALID_PATH);
  EXPECT_EQ(parseGeneratedPath(result.first->path), expected)
      << parseGeneratedPath(result.first->path);

  executor->cancel();
  executor_thread.join();
}
```

PICKNIK

Testing the Conventional Approach

```
TEST_F(TaskPlanningFixture, no_path) {
  auto executor = std::make_shared<rclcpp::executors::SingleThreadedExecutor>();
  std::thread executor_thread;

  auto const pg = std::make_shared<PathGenerator>();

  executor->add_node(pg);
  executor_thread = std::thread([&executor]() { executor->spin(); });

  // GIVEN a populated costmap that is set without error
  auto const return_code = populateAndSetMap();

  EXPECT_EQ(return_code, rclcpp::FutureReturnCode::SUCCESS)
      << "Setting the map failed";

  // WHEN a path is requested between two positions that do not have a valid
  // path between them given the algorithm
  auto const request = std::make_shared<example_srvs::srv::GetPath::Request>();

  request->start.data = {2, 2};
  request->goal.data = {5, 5};

  auto const result = sendPathRequest(request);

  EXPECT_EQ(result.second, rclcpp::FutureReturnCode::SUCCESS) << "Generating path
failed";

  // THEN the global path produced should be empty
  std::vector<Position> const expected{};
  EXPECT_EQ(result.first->code.code,example_srvs::msg::GetPathCodes::NO_VALID_PATH);
  EXPECT_EQ(parseGeneratedPath(result.first->path), expected)
      << parseGeneratedPath(result.first->path);

  executor->cancel();
  executor_thread.join();
}
```

This test:

- Verifies that the Path Generator will return an empty path if it cannot find a path between a start and goal position

Testing the Conventional Approach

```
TEST_F(TaskPlanningFixture, no_path) {
  auto executor = std::make_shared<rclcpp::executors::SingleThreadedExecutor>();
  std::thread executor_thread;

  auto const pg = std::make_shared<PathGenerator>();

  executor->add_node(pg);
  executor_thread = std::thread([&executor]() { executor->spin(); });

  // GIVEN a populated costmap that is set without error
  auto const return_code = populateAndSetMap();

  EXPECT_EQ(return_code, rclcpp::FutureReturnCode::SUCCESS)
      << "Setting the map failed";

  // WHEN a path is requested between two positions that do not have a valid
  // path between them given the algorithm
  auto const request = std::make_shared<example_srvs::srv::GetPath::Request>();

  request->start.data = {2, 2};
  request->goal.data = {5, 5};

  auto const result = sendPathRequest(request);

  EXPECT_EQ(result.second, rclcpp::FutureReturnCode::SUCCESS) << "Generating path
failed";

  // THEN the global path produced should be empty
  std::vector<Position> const expected{};
  EXPECT_EQ(result.first->code.code,example_srvs::msg::GetPathCodes::NO_VALID_PATH);
  EXPECT_EQ(parseGeneratedPath(result.first->path), expected)
      << parseGeneratedPath(result.first->path);

  executor->cancel();
  executor_thread.join();
}
```

This test:

- Verifies that the Path Generator will return an empty path if it cannot find a path between a start and goal position
- Requires creating a thread that executes callbacks for the PathGenerator server

PICKNIK

Testing the Conventional Approach

```
TEST_F(TaskPlanningFixture, no_path) {
  auto executor = std::make_shared<rclcpp::executors::SingleThreadedExecutor>();
  std::thread executor_thread;

  auto const pg = std::make_shared<PathGenerator>();

  executor->add_node(pg);
  executor_thread = std::thread([&executor]() { executor->spin(); });

  // GIVEN a populated costmap that is set without error
  auto const return_code = populateAndSetMap();

  EXPECT_EQ(return_code, rclcpp::FutureReturnCode::SUCCESS)
      << "Setting the map failed";

  // WHEN a path is requested between two positions that do not have a valid
  // path between them given the algorithm
  auto const request = std::make_shared<example_srvs::srv::GetPath::Request>();

  request->start.data = {2, 2};
  request->goal.data = {5, 5};

  auto const result = sendPathRequest(request);

  EXPECT_EQ(result.second, rclcpp::FutureReturnCode::SUCCESS) << "Generating path
failed";

  // THEN the global path produced should be empty
  std::vector<Position> const expected{};
  EXPECT_EQ(result.first->code.code,example_srvs::msg::GetPathCodes::NO_VALID_PATH);
  EXPECT_EQ(parseGeneratedPath(result.first->path), expected)
      << parseGeneratedPath(result.first->path);

  executor->cancel();
  executor_thread.join();
}
```

This test:

- Verifies that the Path Generator will return an empty path if it cannot find a path between a start and goal position
- Requires creating a thread that executes callbacks for the PathGenerator server
- Sends two requests: one for setting the map and one for generating the path
- Each request blocks until a response is received or the request times out

Testing the Conventional Approach

```
TEST_F(TaskPlanningFixture, no_path) {
  auto executor = std::make_shared<rclcpp::executors::SingleThreadedExecutor>();
  std::thread executor_thread;

  auto const pg = std::make_shared<PathGenerator>();

  executor->add_node(pg);
  executor_thread = std::thread([&executor]() { executor->spin(); });

  // GIVEN a populated costmap that is set without error
  auto const return_code = populateAndSetMap();

  EXPECT_EQ(return_code, rclcpp::FutureReturnCode::SUCCESS)
      << "Setting the map failed";

  // WHEN a path is requested between two positions that do not have a valid
  // path between them given the algorithm
  auto const request = std::make_shared<example_srvs::srv::GetPath::Request>();

  request->start.data = {2, 2};
  request->goal.data = {5, 5};

  auto const result = sendPathRequest(request);

  EXPECT_EQ(result.second, rclcpp::FutureReturnCode::SUCCESS) << "Generating path
failed";

  // THEN the global path produced should be empty
  std::vector<Position> const expected{};
  EXPECT_EQ(result.first->code.code,example_srvs::msg::GetPathCodes::NO_VALID_PATH);
  EXPECT_EQ(parseGeneratedPath(result.first->path), expected)
      << parseGeneratedPath(result.first->path);

  executor->cancel();
  executor_thread.join();
}
```

This test:

- Verifies that the Path Generator will return an empty path if it cannot find a path between a start and goal position
- Requires creating a thread that executes callbacks for the PathGenerator server
- Sends two requests: one for setting the map and one for generating the path
- Each request blocks until a response is received or the request times out
- Tests the response from the path generator service

Testing the Conventional Approach

```
TEST_F(TaskPlanningFixture, no_path) {
  auto executor = std::make_shared<rclcpp::executors::SingleThreadedExecutor>();
  std::thread executor_thread;

  auto const pg = std::make_shared<PathGenerator>();

  executor->add_node(pg);
  executor_thread = std::thread([&executor]() { executor->spin(); });

  // GIVEN a populated costmap that is set without error
  auto const return_code = populateAndSetMap();

  EXPECT_EQ(return_code, rclcpp::FutureReturnCode::SUCCESS)
      << "Setting the map failed";

  // WHEN a path is requested between two positions that do not have a valid
  // path between them given the algorithm
  auto const request = std::make_shared<example_srvs::srv::GetPath::Request>();

  request->start.data = {2, 2};
  request->goal.data = {5, 5};

  auto const result = sendPathRequest(request);

  EXPECT_EQ(result.second, rclcpp::FutureReturnCode::SUCCESS) << "Generating path
failed";

  // THEN the global path produced should be empty
  std::vector<Position> const expected{};
  EXPECT_EQ(result.first->code.code,example_srvs::msg::GetPathCodes::NO_VALID_PATH);
  EXPECT_EQ(parseGeneratedPath(result.first->path), expected)
      << parseGeneratedPath(result.first->path);

  executor->cancel();
  executor_thread.join();
}
```

This test:

- Verifies that the Path Generator will return an empty path if it cannot find a path between a start and goal position
- Requires creating a thread that executes callbacks for the PathGenerator server
- Sends two requests: one for setting the map and one for generating the path
- Each request blocks until a response is received or the request times out
- Tests the response from the path generator service

At the end of the test, the executor and thread must be dealt with accordingly

Testing the Conventional Approach

```
TEST_F(TaskPlanningFixture, no_path) {
  auto executor = std::make_shared<rclcpp::executors::SingleThreadedExecutor>();
  std::thread executor_thread;

  auto const pg = std::make_shared<PathGenerator>();

  executor->add_node(pg);
  executor_thread = std::thread([&executor]() { executor->spin(); });

  // GIVEN a populated costmap that is set without error
  auto const return_code = populateAndSetMap();

  EXPECT_EQ(return_code, rclcpp::FutureReturnCode::SUCCESS)
      << "Setting the map failed";

  // WHEN a path is requested between two positions that do not have a valid
  // path between them given the algorithm
  auto const request = std::make_shared<example_srvs::srv::GetPath::Request>();

  request->start.data = {2, 2};
  request->goal.data = {5, 5};

  auto const result = sendPathRequest(request);

  EXPECT_EQ(result.second, rclcpp::FutureReturnCode::SUCCESS) << "Generating path
failed";

  // THEN the global path produced should be empty
  std::vector<Position> const expected{};
  EXPECT_EQ(result.first->code.code,example_srvs::msg::GetPathCodes::NO_VALID_PATH);
  EXPECT_EQ(parseGeneratedPath(result.first->path), expected)
      << parseGeneratedPath(result.first->path);

  executor->cancel();
  executor_thread.join();
}
```

This test:

- Verifies that the Path Generator will return an empty path if it cannot find a path between a start and goal position
- Requires creating a thread that executes callbacks for the PathGenerator server
- Sends two requests: one for setting the map and one for generating the path
- Each request blocks until a response is received or the request times out
- Tests the response from the path generator service

At the end of the test, the executor and thread must be dealt with accordingly

- **Look at all the middleware invocations needed for a simple test**

PICKNIK

Testing the Conventional Approach

```
TEST_F(TaskPlanningFixture, no_path) {
  auto executor = std::make_shared<rclcpp::executors::SingleThreadedExecutor>();
  std::thread executor_thread;

  auto const pg = std::make_shared<PathGenerator>();

  executor->add_node(pg);
  executor_thread = std::thread([&executor]() { executor->spin(); });

  // GIVEN a populated costmap that is set without error
  auto const return_code = populateAndSetMap();

  EXPECT_EQ(return_code, rclcpp::FutureReturnCode::SUCCESS)
      << "Setting the map failed";

  // WHEN a path is requested between two positions that do not have a valid
  // path between them given the algorithm
  auto const request = std::make_shared<example_srvs::srv::GetPath::Request>();

  request->start.data = {2, 2};
  request->goal.data = {5, 5};

  auto const result = sendPathRequest(request);

  EXPECT_EQ(result.second, rclcpp::FutureReturnCode::SUCCESS) << "Generating path
failed";

  // THEN the global path produced should be empty
  std::vector<Position> const expected{};
  EXPECT_EQ(result.first->code.code,example_srvs::msg::GetPathCodes::NO_VALID_PATH);
  EXPECT_EQ(parseGeneratedPath(result.first->path), expected)
      << parseGeneratedPath(result.first->path);

  executor->cancel();
  executor_thread.join();
}
```

This test:

- Verifies that the Path Generator will return an empty path if it cannot find a path between a start and goal position
- Requires creating a thread that executes callbacks for the PathGenerator server
- Sends two requests: one for setting the map and one for generating the path
- Each request blocks until a response is received or the request times out
- Tests the response from the path generator service

At the end of the test, the executor and thread must be dealt with accordingly

- **Look at all the middleware invocations needed for a simple test**
- Is there a way to refactor the code so it can be tested without invoking the middleware?

PICKNIK

Introduction to Functional Programming Principles

What is Functional Programming?

- A programming paradigm characterized by the use of mathematical functions and the avoidance of side effects

PICKNIK

What is Functional Programming?

- A programming paradigm characterized by the use of mathematical functions and the avoidance of side effects
- Functional programming is identified by the use of higher order functions, pure functions, monads, declarative syntax

What is Functional Programming?

- A programming paradigm characterized by the use of mathematical functions and the avoidance of side effects
- Functional programming is identified by the use of higher order functions, pure functions, monads, declarative syntax
  - C++ has all the tools to implement functional programming, including lambda functions, std::function, std::optional, std::expected, and more

What is Functional Programming?

- A programming paradigm characterized by the use of mathematical functions and the avoidance of side effects
- Functional programming is identified by the use of higher order functions, pure functions, monads, declarative syntax
    - C++ has all the tools to implement functional programming, including lambda functions, std::function, std::optional, std::expected, and more
    - Want to maximize use of these features to write code with a minimal number of side effects

PICKNIK

What is Functional Programming?

- A programming paradigm characterized by the use of mathematical functions and the avoidance of side effects
- Functional programming is identified by the use of higher order functions, pure functions, monads, declarative syntax
  - C++ has all the tools to implement functional programming, including lambda functions, std::function, std::optional, std::expected, and more
  - Want to maximize use of these features to write code with a minimal number of side effects
- Let’s go over some principles and how they can be applied using C++

Pure Functions

- A pure function is a function that
  - is deterministic: They always return the same output for the same set of inputs

Pure Functions

- A pure function is a function that
  - is deterministic: They always return the same output for the same set of inputs
  - has no side effects: They don't alter the state of the program, global variables, or produce any observable interactions with the outside world, such as writing to files or displaying output

Pure Functions

- A pure function is a function that
  - is deterministic: They always return the same output for the same set of inputs
  - has no side effects: They don’t alter the state of the program, global variables, or produce any observable interactions with the outside world, such as writing to files or displaying output
- Why is this desirable?

PICKNIK

Pure Functions

- A pure function is a function that
  - is deterministic: They always return the same output for the same set of inputs
  - has no side effects: They don't alter the state of the program, global variables, or produce any observable interactions with the outside world, such as writing to files or displaying output
- Why is this desirable?
  - local reasoning: The code can be understood just by looking at that portion and a limited scope around it

Pure Functions

- A pure function is a function that
  - is deterministic: They always return the same output for the same set of inputs
  - has no side effects: They don't alter the state of the program, global variables, or produce any observable interactions with the outside world, such as writing to files or displaying output
- Why is this desirable?
  - local reasoning: The code can be understood just by looking at that portion and a limited scope around it
  - testability: Testing pure functions is trivial

PICKNIK

Pure Functions

- A pure function is a function that
  - is deterministic: They always return the same output for the same set of inputs
  - has no side effects: They don't alter the state of the program, global variables, or produce any observable interactions with the outside world, such as writing to files or displaying output
- Why is this desirable?
  - local reasoning: The code can be understood just by looking at that portion and a limited scope around it
  - testability: Testing pure functions is trivial
- Practically, pure functions do not:
  - contain state (static variables or class member variables)

PICKNIK

Pure Functions

- A pure function is a function that
  - is deterministic: They always return the same output for the same set of inputs
  - has no side effects: They don’t alter the state of the program, global variables, or produce any observable interactions with the outside world, such as writing to files or displaying output
- Why is this desirable?
  - local reasoning: The code can be understood just by looking at that portion and a limited scope around it
  - testability: Testing pure functions is trivial
- Practically, pure functions do not:
  - contain state (static variables or class member variables)
  - mutate input parameters

Pure Functions

- A pure function is a function that
  - is deterministic: They always return the same output for the same set of inputs
  - has no side effects: They don’t alter the state of the program, global variables, or produce any observable interactions with the outside world, such as writing to files or displaying output
- Why is this desirable?
  - local reasoning: The code can be understood just by looking at that portion and a limited scope around it
  - testability: Testing pure functions is trivial
- Practically, pure functions do not:
  - contain state (static variables or class member variables)
  - mutate input parameters
- A function is pure if and only if it could be replaced by a lookup table (potentially infinitely large!)

PICKNIK

(im)Pure Functions

```
std::string get_timestamp() {
  std::time_t t = std::time(nullptr);
  std::tm tm = *std::localtime(&t);
  /* Implementation code*/
  return oss.str();
}
void log(std::string const& message)
{
  std::cout << get_timestamp() << " [INFO] " << message << '\n';
}
auto filter = [x_previous = 0.] (double xn) mutable -> double {
  /* Implementation code */
  x_previous =  xn;
  return yn;
};
void filter_timeseries(std::vector<double> const& x,
std::vector<double>& y) {
  /* Implementation code*/
  for (auto const& xn : x) {
    y.push_back(f(xn));
  }
}
int main()
{
  auto f = filter;
  f(1); // 0.7
  f(1); // 1.0
  std::vector<double> const x = {1, 1, 1};
  std::vector<double> y;
  filter_timeseries(x, y);
}
```

- Here is a simple implementation of a smoothing filter

PICKNIK

(im)Pure Functions

```
std::string get_timestamp() {
  std::time_t t = std::time(nullptr);
  std::tm tm = *std::localtime(&t);
  /* Implementation code*/
  return oss.str();
}
void log(std::string const& message)
{
  std::cout << get_timestamp() << " [INFO] " << message << '\n';
}
auto filter = [x_previous = 0.] (double xn) mutable -> double {
  /* Implementation code */
  x_previous =  xn;
  return yn;
};
void filter_timeseries(std::vector<double> const& x,
std::vector<double>& y) {
  /* Implementation code*/
  for (auto const& xn : x) {
    y.push_back(f(xn));
  }
}
int main()
{
  auto f = filter;
  f(1); // 0.7
  f(1); // 1.0
  std::vector<double> const x = {1, 1, 1};
  std::vector<double> y;
  filter_timeseries(x, y);
}
```

- Here is a simple implementation of a smoothing filter
- None of these functions are pure

PICKNIK

(im)Pure Functions

```cpp
std::string get_timestamp() {
  std::time_t t = std::time(nullptr);
  std::tm tm = *std::localtime(&t);
  /* Implementation code*/
  return oss.str();
}
void log(std::string const& message)
{
  std::cout << get_timestamp() << " [INFO] " << message << '\n';
}
auto filter = [x_previous = 0.] (double xn) mutable -> double {
  /* Implementation code */
  x_previous =  xn;
  return yn;
};
void filter_timeseries(std::vector<double> const& x,
std::vector<double>& y) {
  /* Implementation code*/
  for (auto const& xn : x) {
    y.push_back(f(xn));
  }
}
int main()
{
  auto f = filter;
  f(1); // 0.7
  f(1); // 1.0
  std::vector<double> const x = {1, 1, 1};
  std::vector<double> y;
  filter_timeseries(x, y);
}
```

- Here is a simple implementation of a smoothing filter
- None of these functions are pure
- `get_timestamp` gets the time from the local machine
    - That is an input side effect

PICKNIK

(im)Pure Functions

```
std::string get_timestamp() {
  std::time_t t = std::time(nullptr);
  std::tm tm = *std::localtime(&t);
  /* Implementation code*/
  return oss.str();
}
void log(std::string const& message)
{
  std::cout << get_timestamp() << " [INFO] " << message << '\n';
}
auto filter = [x_previous = 0.] (double xn) mutable -> double {
  /* Implementation code */
  x_previous =  xn;
  return yn;
};
void filter_timeseries(std::vector<double> const& x,
std::vector<double>& y) {
  /* Implementation code*/
  for (auto const& xn : x) {
    y.push_back(f(xn));
  }
}
int main()
{
  auto f = filter;
  f(1); // 0.7
  f(1); // 1.0
  std::vector<double> const x = {1, 1, 1};
  std::vector<double> y;
  filter_timeseries(x, y);
}
```

- Here is a simple implementation of a smoothing filter
- None of these functions are pure
- `get_timestamp` gets the time from the local machine
  - That is an input side effect
- Each invocation of `get_timestamp` will almost always return a different value

(im)Pure Functions

```
std::string get_timestamp() {
  std::time_t t = std::time(nullptr);
  std::tm tm = *std::localtime(&t);
  /* Implementation code*/
  return oss.str();
}
void log(std::string const& message)
{
  std::cout << get_timestamp() << " [INFO] " << message << '\n';
}
auto filter = [x_previous = 0.] (double xn) mutable -> double {
  /* Implementation code */
  x_previous =  xn;
  return yn;
};
void filter_timeseries(std::vector<double> const& x,
std::vector<double>& y) {
  /* Implementation code*/
  for (auto const& xn : x) {
    y.push_back(f(xn));
  }
}
int main()
{
  auto f = filter;
  f(1); // 0.7
  f(1); // 1.0
  std::vector<double> const x = {1, 1, 1};
  std::vector<double> y;
  filter_timeseries(x, y);
}
```

- Here is a simple implementation of a smoothing filter
- None of these functions are pure
- `get_timestamp` gets the time from the local machine
  - That is an input side effect
- Each invocation of `get_timestamp` will almost always return a different value
- `log` prints directly to the console

PICKNIK

(im)Pure Functions

```
std::string get_timestamp() {
  std::time_t t = std::time(nullptr);
  std::tm tm = *std::localtime(&t);
  /* Implementation code*/
  return oss.str();
}
void log(std::string const& message)
{
  std::cout << get_timestamp() << " [INFO] " << message << '\n';
}
auto filter = [x_previous = 0.] (double xn) mutable -> double {
  /* Implementation code */
  x_previous =  xn;
  return yn;
};
void filter_timeseries(std::vector<double> const& x,
std::vector<double>& y) {
  /* Implementation code*/
  for (auto const& xn : x) {
    y.push_back(f(xn));
  }
}
int main()
{
  auto f = filter;
  f(1); // 0.7
  f(1); // 1.0
  std::vector<double> const x = {1, 1, 1};
  std::vector<double> y;
  filter_timeseries(x, y);
}
```

- Here is a simple implementation of a smoothing filter
- None of these functions are pure
- `get_timestamp` gets the time from the local machine
    - That is an input side effect
- Each invocation of `get_timestamp` will almost always return a different value
- `log` prints directly to the console
    - This is an output side effect

PICKNIK

(im)Pure Functions

```
std::string get_timestamp() {
  std::time_t t = std::time(nullptr);
  std::tm tm = *std::localtime(&t);
  /* Implementation code*/
  return oss.str();
}
void log(std::string const& message)
{
  std::cout << get_timestamp() << " [INFO] " << message << '\n';
}
auto filter = [x_previous = 0.] (double xn) mutable -> double {
  /* Implementation code */
  x_previous =  xn;
  return yn;
};
void filter_timeseries(std::vector<double> const& x,
std::vector<double>& y) {
  /* Implementation code*/
  for (auto const& xn : x) {
    y.push_back(f(xn));
  }
}
int main()
{
  auto f = filter;
  f(1); // 0.7
  f(1); // 1.0
  std::vector<double> const x = {1, 1, 1};
  std::vector<double> y;
  filter_timeseries(x, y);
}
```

- Here is a simple implementation of a smoothing filter
- None of these functions are pure
- `get_timestamp` gets the time from the local machine
    - That is an input side effect
- Each invocation of `get_timestamp` will almost always return a different value
- `log` prints directly to the console
    - This is an output side effect
- `filter` contains state with the variable that is initialized in the lambda capture

(im)Pure Functions

```
std::string get_timestamp() {
  std::time_t t = std::time(nullptr);
  std::tm tm = *std::localtime(&t);
  /* Implementation code*/
  return oss.str();
}
void log(std::string const& message)
{
  std::cout << get_timestamp() << " [INFO] " << message << '\n';
}
auto filter = [x_previous = 0.] (double xn) mutable -> double {
  /* Implementation code */
  x_previous =  xn;
  return yn;
};
void filter_timeseries(std::vector<double> const& x,
std::vector<double>& y) {
  /* Implementation code*/
  for (auto const& xn : x) {
    y.push_back(f(xn));
  }
}
int main()
{
  auto f = filter;
  f(1); // 0.7
  f(1); // 1.0
  std::vector<double> const x = {1, 1, 1};
  std::vector<double> y;
  filter_timeseries(x, y);
}
```

- Here is a simple implementation of a smoothing filter
- None of these functions are pure
- `get_timestamp` gets the time from the local machine
    - That is an input side effect
- Each invocation of `get_timestamp` will almost always return a different value
- `log` prints directly to the console
    - This is an output side effect
- `filter` contains state with the variable that is initialized in the lambda capture
    - Running `filter` multiple times with the same input returns different outputs

PICKNIK

(im)Pure Functions

```
std::string get_timestamp() {
  std::time_t t = std::time(nullptr);
  std::tm tm = *std::localtime(&t);
  /* Implementation code*/
  return oss.str();
}
void log(std::string const& message)
{
  std::cout << get_timestamp() << " [INFO] " << message << '\n';
}
auto filter = [x_previous = 0.] (double xn) mutable -> double {
  /* Implementation code */
  x_previous =  xn;
  return yn;
};
void filter_timeseries(std::vector<double> const& x,
std::vector<double>& y) {
  /* Implementation code*/
  for (auto const& xn : x) {
    y.push_back(f(xn));
  }
}
int main()
{
  auto f = filter;
  f(1); // 0.7
  f(1); // 1.0
  std::vector<double> const x = {1, 1, 1};
  std::vector<double> y;
  filter_timeseries(x, y);
}
```

- Here is a simple implementation of a smoothing filter
- None of these functions are pure
- `get_timestamp` gets the time from the local machine
  - That is an input side effect
- Each invocation of `get_timestamp` will almost always return a different value
- `log` prints directly to the console
  - This is an output side effect
- `filter` contains state with the variable that is initialized in the closure
  - Running `filter` multiple times with the same input returns different outputs
- `filter_timeseries` has an out parameter that is modified

PICKNIK

Closures and Partial Applications

```cpp
int main()
{
  // Create a multiplication function
  const auto multiply = [](int a, int b){
    return a*b;
  };

  // Partially apply the 'multiply' function by fixing the
first argument to 2
  auto multiplyBy2 = [multiply](int b) {
    return multiply(2, b);
  };

  // Now 'multiplyBy2' only requires one argument
  int result = multiplyBy2(3);
  std::cout << result << std::endl;  // Output: 6
}
```

- A closure is a function object that has an environment of its own, which keeps track of the variables captured from the outer scope

Closures and Partial Applications

```
int main()
{
  // Create a multiplication function
  const auto multiply = [](int a, int b){
    return a*b;
  };

  // Partially apply the 'multiply' function by fixing the
first argument to 2
  auto multiplyBy2 = [multiply](int b) {
    return multiply(2, b);
  };

  // Now 'multiplyBy2' only requires one argument
  int result = multiplyBy2(3);
  std::cout << result << std::endl;  // Output: 6
}
```

- A closure is a function object that has an environment of its own, which keeps track of the variables captured from the outer scope
  - A closure can access and manipulate these captured variables throughout its lifetime

PICKNIK

Closures and Partial Applications

```cpp
int main()
{
  // Create a multiplication function
  const auto multiply = [](int a, int b){
    return a*b;
  };

  // Partially apply the 'multiply' function by fixing the
first argument to 2
  auto multiplyBy2 = [multiply](int b) {
    return multiply(2, b);
  };

  // Now 'multiplyBy2' only requires one argument
  int result = multiplyBy2(3);
  std::cout << result << std::endl;  // Output: 6
}
```

- A closure is a function object that has an environment of its own, which keeps track of the variables captured from the outer scope
  - A closure can access and manipulate these captured variables throughout its lifetime
  - Essentially the lambda capture!

Closures and Partial Applications

```
int main()
{
  // Create a multiplication function
  const auto multiply = [](int a, int b){
    return a*b;
  };

  // Partially apply the 'multiply' function by fixing the
first argument to 2
  auto multiplyBy2 = [multiply](int b) {
    return multiply(2, b);
  };

  // Now 'multiplyBy2' only requires one argument
  int result = multiplyBy2(3);
  std::cout << result << std::endl;  // Output: 6
}
```

- A closure is a function object that has an environment of its own, which keeps track of the variables captured from the outer scope
  - A closure can access and manipulate these captured variables throughout its lifetime
  - Essentially the lambda capture!
- Partial applications are a concept in functional programming where a function is fixed with a certain number of arguments, producing a new function with a lesser number of arguments

Closures and Partial Applications

```
Path plan(Position const& start, Position const& goal, Map const& map) {
  /* Variable setup */
  auto is_occupied = [&](auto const x, auto const y) -> bool {
    return map.at(x, y) == 255;
  };
  for (size_t i = 0; i < (std::abs(del_x)); ++i) {
    if (is_occupied(path.back().x + del_x_sign, path.back().y)) {
      return {};
    }
    path.push_back({path.back().x + del_x_sign, path.back().y});
  }
  /* More implementation code */
  return path;
}
int main() {
  // Create map
  auto map = Map(4, 4);
  map.data() = {
    0, 0, 0, 0,  //
    0, 0, 0, 0,  //
    0, 0, 0, 0,  //
    0, 0, 0, 0,  //
  };
  auto const planner = [map](auto const& start, auto const& goal){
    return plan(start, goal, map);
  };
  auto const plan = planner(Position{0, 0}, Position{2, 2});
  /* More implementation code */
}
```

- Here is more code that shows closures and partial applications

Closures and Partial Applications

```cpp
Path plan(Position const& start, Position const& goal, Map const& map) {
  /* Variable setup */
  auto is_occupied = [&](auto const x, auto const y) -> bool {
    return map.at(x, y) == 255;
  };
  for (size_t i = 0; i < (std::abs(del_x)); ++i) {
    if (is_occupied(path.back().x + del_x_sign, path.back().y)) {
      return {};
    }
    path.push_back({path.back().x + del_x_sign, path.back().y});
  }
  /* More implementation code */
  return path;
}
int main() {
  // Create map
  auto map = Map(4, 4);
  map.data() = {
    0, 0, 0, 0,  //
    0, 0, 0, 0,  //
    0, 0, 0, 0,  //
    0, 0, 0, 0,  //
  };
  auto const planner = [map](auto const& start, auto const& goal){
    return plan(start, goal, map);
  };
  auto const plan = planner(Position{0, 0}, Position{2, 2});
  /* More implementation code */
}
```

- Here is more code that shows closures and partial applications
- `is_occupied` is a lambda function in `plan` that checks if a cell in a map is occupied

PICKNIK

Closures and Partial Applications

```
Path plan(Position const& start, Position const& goal, Map const& map) {
  /* Variable setup */
  auto is_occupied = [&](auto const x, auto const y) -> bool {
    return map.at(x, y) == 255;
  };
  for (size_t i = 0; i < (std::abs(del_x)); ++i) {
    if (is_occupied(path.back().x + del_x_sign, path.back().y)) {
      return {};
    }
    path.push_back({path.back().x + del_x_sign, path.back().y});
  }
  /* More implementation code */
  return path;
}
int main() {
  // Create map
  auto map = Map(4, 4);
  map.data() = {
    0, 0, 0, 0,  //
    0, 0, 0, 0,  //
    0, 0, 0, 0,  //
    0, 0, 0, 0,  //
  };
  auto const planner = [map](auto const& start, auto const& goal){
    return plan(start, goal, map);
  };
  auto const plan = planner(Position{0, 0}, Position{2, 2});
  /* More implementation code */
}
```

- Here is more code that shows closures and partial applications
- `is_occupied` is a lambda function in `plan` that checks if a cell in a map is occupied
  - `is_occupied` captures the input parameters and all variables that come before it by reference

PICKNIK

Closures and Partial Applications

```
Path plan(Position const& start, Position const& goal, Map const& map) {
  /* Variable setup */
  auto is_occupied = [&](auto const x, auto const y) -> bool {
    return map.at(x, y) == 255;
  };
  for (size_t i = 0; i < (std::abs(del_x)); ++i) {
    if (is_occupied(path.back().x + del_x_sign, path.back().y)) {
      return {};
    }
    path.push_back({path.back().x + del_x_sign, path.back().y});
  }
  /* More implementation code */
  return path;
}
int main() {
  // Create map
  auto map = Map(4, 4);
  map.data() = {
    0, 0, 0, 0,  //
    0, 0, 0, 0,  //
    0, 0, 0, 0,  //
    0, 0, 0, 0,  //
  };
  auto const planner = [map](auto const& start, auto const& goal){
    return plan(start, goal, map);
  };
  auto const plan = planner(Position{0, 0}, Position{2, 2});
  /* More implementation code */
}
```

- Here is more code that shows closures and partial applications
- `is_occupied` is a lambda function in `plan` that checks if a cell in a map is occupied
  - `is_occupied` captures the input parameters and all variables that come before it by reference
  - That is a closure

Closures and Partial Applications

```
Path plan(Position const& start, Position const& goal, Map const& map) {
  /* Variable setup */
  auto is_occupied = [&](auto const x, auto const y) -> bool {
    return map.at(x, y) == 255;
  };
  for (size_t i = 0; i < (std::abs(del_x)); ++i) {
    if (is_occupied(path.back().x + del_x_sign, path.back().y)) {
      return {};
    }
    path.push_back({path.back().x + del_x_sign, path.back().y});
  }
  /* More implementation code */
  return path;
}
int main() {
  // Create map
  auto map = Map(4, 4);
  map.data() = {
    0, 0, 0, 0,  //
    0, 0, 0, 0,  //
    0, 0, 0, 0,  //
    0, 0, 0, 0,  //
  };
  auto const planner = [map](auto const& start, auto const& goal){
    return plan(start, goal, map);
  };
  auto const plan = planner(Position{0, 0}, Position{2, 2});
  /* More implementation code */
}
```

- Here is more code that shows closures and partial applications
- `is_occupied` is a lambda function in `plan` that checks if a cell in a map is occupied
  - `is_occupied` captures the input parameters and all variables that come before it by reference
  - That is a closure
- `planner` is a lambda function that captures map and partially applies that map to plan

PICKNIK

Closures and Partial Applications

```
Path plan(Position const& start, Position const& goal, Map const& map) {
  /* Variable setup */
  auto is_occupied = [&](auto const x, auto const y) -> bool {
    return map.at(x, y) == 255;
  };
  for (size_t i = 0; i < (std::abs(del_x)); ++i) {
    if (is_occupied(path.back().x + del_x_sign, path.back().y)) {
      return {};
    }
    path.push_back({path.back().x + del_x_sign, path.back().y});
  }
  /* More implementation code */
  return path;
}
int main() {
  // Create map
  auto map = Map(4, 4);
  map.data() = {
    0, 0, 0, 0,  //
    0, 0, 0, 0,  //
    0, 0, 0, 0,  //
    0, 0, 0, 0,  //
  };
  auto const planner = [map](auto const& start, auto const& goal){
    return plan(start, goal, map);
  };
  auto const plan = planner(Position{0, 0}, Position{2, 2});
  /* More implementation code */
}
```

- Here is more code that shows closures and partial applications
- `is_occupied` is a lambda function in `plan` that checks if a cell in a map is occupied
  - `is_occupied` captures the input parameters and all variables that come before it by reference
  - That is a closure
- `planner` is a lambda function that captures map and partially applies that map to plan
  - This reduces the number of required input arguments

PICKNIK

Higher Order Functions

- A higher order function is a function that can:
  - accept other functions as arguments

Higher Order Functions

- A higher order function is a function that can:
  - accept other functions as arguments
  - return functions as a result

Higher Order Functions

- A higher order function is a function that can:
  - accept other functions as arguments
  - return functions as a result
- Why is this desirable?

Higher Order Functions

- A higher order function is a function that can:
  - accept other functions as arguments
  - return functions as a result
- Why is this desirable?
  - Modularity and reusability: The behavior of a higher order function is configurable and they can be used in different contexts

Higher Order Functions

- A higher order function is a function that can:
  - accept other functions as arguments
  - return functions as a result
- Why is this desirable?
  - Modularity and reusability: The behavior of a higher order function is configurable and they can be used in different contexts
  - Control flow abstraction: control flows, like looping and conditional execution, can be abstracted in a readable and reusable manner

PICKNIK

Higher Order Functions

- A higher order function is a function that can:
  - accept other functions as arguments
  - return functions as a result
- Why is this desirable?
  - Modularity and reusability: The behavior of a higher order function is configurable and they can be used in different contexts
  - Control flow abstraction: control flows, like looping and conditional execution, can be abstracted in a readable and reusable manner
  - Testability: Smaller well-defined functions are easier to test

PICKNIK

Higher Order Functions

- A higher order function is a function that can:
  - accept other functions as arguments
  - return functions as a result
- Why is this desirable?
  - Modularity and reusability: The behavior of a higher order function is configurable and they can be used in different contexts
  - Control flow abstraction: control flows, like looping and conditional execution, can be abstracted in a readable and reusable manner
  - Testability: Smaller well-defined functions are easier to test
- The Standard Template Library contains many higher order functions!

PICKNIK

Higher Order Functions

- A higher order function is a function that can:
  - accept other functions as arguments
  - return functions as a result
- Why is this desirable?
  - Modularity and reusability: The behavior of a higher order function is configurable and they can be used in different contexts
  - Control flow abstraction: control flows, like looping and conditional execution, can be abstracted in a readable and reusable manner
  - Testability: Smaller well-defined functions are easier to test
- The Standard Template Library contains many higher order functions!
  - std::transform, std::find_if, std::copy, and more

Higher Order Functions

- A higher order function is a function that can:
  - accept other functions as arguments
  - return functions as a result
- Why is this desirable?
  - Modularity and reusability: The behavior of a higher order function is configurable and they can be used in different contexts
  - Control flow abstraction: control flows, like looping and conditional execution, can be abstracted in a readable and reusable manner
  - Testability: Smaller well-defined functions are easier to test
- The Standard Template Library contains many higher order functions!
  - std::transform, std::find_if, std::copy, and more
  - Let's look at some STL higher order functions

PICKNIK

Higher Order Functions in the STL

```
std::vector<int> input {0, 1, 2, 3, 4};

const auto triple = [](const auto num) -> int {return num*3;};

// input = {0, 1, 2, 3, 4}
std::transform(input.cbegin(), input.cend(), input.begin(), triple);
// input = {0, 3, 6, 9, 12}

const auto less_than_5 = [](const auto num) -> int {return num < 5;};

const auto new_end = std::remove_if(input.begin(), input.end() ,
less_than_5);
// input = {6, 9, 12, 9, 12}
input.resize(std::distance(input.begin(), new_end));
// input = {6, 9, 12}

const auto result = std::accumulate(input.cbegin(), input.cend(), 0);
// result = 27
```

- The code:
  - Multiplies each element of a vector by 3

PICKNIK

Higher Order Functions in the STL

```
std::vector<int> input {0, 1, 2, 3, 4};

const auto triple = [](const auto num) -> int {return num*3;};

// input = {0, 1, 2, 3, 4}
std::transform(input.cbegin(), input.cend(), input.begin(), triple);
// input = {0, 3, 6, 9, 12}

const auto less_than_5 = [](const auto num) -> int {return num < 5;};

const auto new_end = std::remove_if(input.begin(), input.end() ,
less_than_5);
// input = {6, 9, 12, 9, 12}
input.resize(std::distance(input.begin(), new_end));
// input = {6, 9, 12}

const auto result = std::accumulate(input.cbegin(), input.cend(), 0);
// result = 27
```

- The code:
  - Multiplies each element of a vector by 3
  - Removes any element less than 5

PICKNIK

Higher Order Functions in the STL

```
std::vector<int> input {0, 1, 2, 3, 4};

const auto triple = [](const auto num) -> int {return num*3;};

// input = {0, 1, 2, 3, 4}
std::transform(input.cbegin(), input.cend(), input.begin(), triple);
// input = {0, 3, 6, 9, 12}

const auto less_than_5 = [](const auto num) -> int {return num < 5;};

const auto new_end = std::remove_if(input.begin(), input.end() ,
less_than_5);
// input = {6, 9, 12, 9, 12}
input.resize(std::distance(input.begin(), new_end));
// input = {6, 9, 12}

const auto result = std::accumulate(input.cbegin(), input.cend(), 0);
// result = 27
```

- The code:
  - Multiplies each element of a vector by 3
  - Removes any element less than 5
  - Sums up all of the elements

PICKNIK

Higher Order Functions in the STL

```
std::vector<int> input {0, 1, 2, 3, 4};

const auto triple = [](const auto num) -> int {return num*3;};

// input = {0, 1, 2, 3, 4}
std::transform(input.cbegin(), input.cend(), input.begin(), triple);
// input = {0, 3, 6, 9, 12}

const auto less_than_5 = [](const auto num) -> int {return num < 5;};

const auto new_end = std::remove_if(input.begin(), input.end() ,
less_than_5);
// input = {6, 9, 12, 9, 12}
input.resize(std::distance(input.begin(), new_end));
// input = {6, 9, 12}

const auto result = std::accumulate(input.cbegin(), input.cend(), 0);
// result = 27
```

- The code:
  - Multiplies each element of a vector by 3
  - Removes any element less than 5
  - Sums up all of the elements
- The three algorithms used, `transform`, `reduce_if`, and `accumulate` are very common in the functional paradigm

PICKNIK

Higher Order Functions in the STL

```
std::vector<int> input {0, 1, 2, 3, 4};

const auto triple = [](const auto num) -> int {return num*3;};

// input = {0, 1, 2, 3, 4}
std::transform(input.cbegin(), input.cend(), input.begin(), triple);
// input = {0, 3, 6, 9, 12}

const auto less_than_5 = [](const auto num) -> int {return num < 5;};

const auto new_end = std::remove_if(input.begin(), input.end() ,
less_than_5);
// input = {6, 9, 12, 9, 12}
input.resize(std::distance(input.begin(), new_end));
// input = {6, 9, 12}

const auto result = std::accumulate(input.cbegin(), input.cend(), 0);
// result = 27
```

- The code:
  - Multiplies each element of a vector by 3
  - Removes any element less than 5
  - Sums up all of the elements
- The three algorithms used, `transform`, `reduce_if`, and `accumulate` are very common in the functional paradigm
  - They are canonically known as map, filter, reduce

PICKNIK

Higher Order Functions in the STL

```
std::vector<int> input {0, 1, 2, 3, 4};

const auto triple = [](const auto num) -> int {return num*3;};

// input = {0, 1, 2, 3, 4}
std::transform(input.cbegin(), input.cend(), input.begin(), triple);
// input = {0, 3, 6, 9, 12}

const auto less_than_5 = [](const auto num) -> int {return num < 5;};

const auto new_end = std::remove_if(input.begin(), input.end() ,
less_than_5);
// input = {6, 9, 12, 9, 12}
input.resize(std::distance(input.begin(), new_end));
// input = {6, 9, 12}

const auto result = std::accumulate(input.cbegin(), input.cend(), 0);
// result = 27
```

- The code:
  - Multiplies each element of a vector by 3
  - Removes any element less than 5
  - Sums up all of the elements
- The three algorithms used, `transform`, `reduce_if`, and `accumulate` are very common in the functional paradigm
  - They are canonically known as map, filter, reduce
- The declarative syntax of the code makes it easy to understand

Higher Order Functions in the STL

```
std::vector<int> input {0, 1, 2, 3, 4};

const auto triple = [](const auto num) -> int {return num*3;};

// input = {0, 1, 2, 3, 4}
std::transform(input.cbegin(), input.cend(), input.begin(), triple);
// input = {0, 3, 6, 9, 12}

const auto less_than_5 = [](const auto num) -> int {return num < 5;};

const auto new_end = std::remove_if(input.begin(), input.end() ,
less_than_5);
// input = {6, 9, 12, 9, 12}
input.resize(std::distance(input.begin(), new_end));
// input = {6, 9, 12}

const auto result = std::accumulate(input.cbegin(), input.cend(), 0);
// result = 27
```

- The code:
  - Multiplies each element of a vector by 3
  - Removes any element less than 5
  - Sums up all of the elements
- The three algorithms used, `transform`, `reduce_if`, and `accumulate` are very common in the functional paradigm
  - They are canonically known as map, filter, reduce
- The declarative syntax of the code makes it easy to understand
- Higher order functions are extremely flexible

PICKNIK

Higher Order Functions

```
using WaypointGenerator = std::function<Waypoint(Position const&,
Waypoint const&, Speed)>;
Waypoint interpolate(Position const& p, Waypoint const& w, Speed
s) {
    auto const distance = norm(p, w.p);
    auto const duration = distance / s;
    return {p, duration + w.stamp};
}
Trajectory generate_trajectory(Path const& path, Speed speed,
WaypointGenerator next_waypoint) {
    Trajectory traj = {{path.front(), 0.}};
    std::transform(std::next(path.begin()), path.end(),
std::back_inserter(traj),
        [&](auto const& point) {
            return next_waypoint(point, traj.back(), speed);
        }
    );
    return traj;
}
int main() {
    Path p = { {1, 2}, {2, 3}, {3, 4} };
    auto const t = generate_trajectory(p, 1, interpolate);
    /* More Implementation Code */
}
```

- Here is a simplified implementation of a trajectory generator

Higher Order Functions

```
using WaypointGenerator = std::function<Waypoint(Position const&,
Waypoint const&, Speed)>;
Waypoint interpolate(Position const& p, Waypoint const& w, Speed
s) {
    auto const distance = norm(p, w.p);
    auto const duration = distance / s;
    return {p, duration + w.stamp};
}
Trajectory generate_trajectory(Path const& path, Speed speed,
WaypointGenerator next_waypoint) {
    Trajectory traj = {{path.front(), 0.}};
    std::transform(std::next(path.begin()), path.end(),
std::back_inserter(traj),
        [&](auto const& point) {
            return next_waypoint(point, traj.back(), speed);
        }
    );
    return traj;
}
int main() {
    Path p = { {1, 2}, {2, 3}, {3, 4} };
    auto const t = generate_trajectory(p, 1, interpolate);
    /* More Implementation Code */
}
```

- Here is a simplified implementation of a trajectory generator
- `generate_trajectory` is a higher order function

PICKNIK

Higher Order Functions

```
using WaypointGenerator = std::function<Waypoint(Position const&,
Waypoint const&, Speed)>;
Waypoint interpolate(Position const& p, Waypoint const& w, Speed
s) {
    auto const distance = norm(p, w.p);
    auto const duration = distance / s;
    return {p, duration + w.stamp};
}
Trajectory generate_trajectory(Path const& path, Speed speed,
WaypointGenerator next_waypoint) {
    Trajectory traj = {{path.front(), 0.}};
    std::transform(std::next(path.begin()), path.end(),
std::back_inserter(traj),
        [&](auto const& point) {
            return next_waypoint(point, traj.back(), speed);
        }
    );
    return traj;
}
int main() {
    Path p = { {1, 2}, {2, 3}, {3, 4} };
    auto const t = generate_trajectory(p, 1, interpolate);
    /* More Implementation Code */
}
```

- Here is a simplified implementation of a trajectory generator
- `generate_trajectory` is a higher order function
    - It can take in a function of type `WaypointGenerator`

PICKNIK

Higher Order Functions

```
using WaypointGenerator = std::function<Waypoint(Position const&,
Waypoint const&, Speed)>;
Waypoint interpolate(Position const& p, Waypoint const& w, Speed
s) {
    auto const distance = norm(p, w.p);
    auto const duration = distance / s;
    return {p, duration + w.stamp};
}
Trajectory generate_trajectory(Path const& path, Speed speed,
WaypointGenerator next_waypoint) {
    Trajectory traj = {{path.front(), 0.}};
    std::transform(std::next(path.begin()), path.end(),
std::back_inserter(traj),
        [&](auto const& point) {
            return next_waypoint(point, traj.back(), speed);
        }
    );
    return traj;
}
int main() {
    Path p = { {1, 2}, {2, 3}, {3, 4} };
    auto const t = generate_trajectory(p, 1, interpolate);
    /* More Implementation Code */
}
```

- Here is a simplified implementation of a trajectory generator
- `generate_trajectory` is a higher order function
    - It can take in a function of type `WaypointGenerator`
- In main, `interpolate` is passed into `generate_trajectory`

PICKNIK

Higher Order Functions

```
using WaypointGenerator = std::function<Waypoint(Position const&,
Waypoint const&, Speed)>;
Waypoint interpolate(Position const& p, Waypoint const& w, Speed
s) {
    auto const distance = norm(p, w.p);
    auto const duration = distance / s;
    return {p, duration + w.stamp};
}
Trajectory generate_trajectory(Path const& path, Speed speed,
WaypointGenerator next_waypoint) {
    Trajectory traj = {{path.front(), 0.}};
    std::transform(std::next(path.begin()), path.end(),
std::back_inserter(traj),
        [&](auto const& point) {
            return next_waypoint(point, traj.back(), speed);
        }
    );
    return traj;
}
int main() {
    Path p = { {1, 2}, {2, 3}, {3, 4} };
    auto const t = generate_trajectory(p, 1, interpolate);
    /* More Implementation Code */
}
```

- Here is a simplified implementation of a trajectory generator
- `generate_trajectory` is a higher order function
    - It can take in a function of type `WaypointGenerator`
- In `main`, `interpolate` is passed into `generate_trajectory`
- The type of trajectory that can be generated is customizable by passing in a different `WaypointGenerator` function

Monadic Error Handling

- Monadic error handling is a functional programming technique for dealing with errors in a clean, compositional, and type-safe way

Monadic Error Handling

- Monadic error handling is a functional programming technique for dealing with errors in a clean, compositional, and type-safe way
- This approach encapsulates error handling into types and operations on these types

Monadic Error Handling

- Monadic error handling is a functional programming technique for dealing with errors in a clean, compositional, and type-safe way
- This approach encapsulates error handling into types and operations on these types
- Why is this desirable?

Monadic Error Handling

- Monadic error handling is a functional programming technique for dealing with errors in a clean, compositional, and type-safe way
- This approach encapsulates error handling into types and operations on these types
- Why is this desirable?
    - Type encapsulation: Monadic error handling encapsulates the result of computations along with possible errors within a single type

Monadic Error Handling

- Monadic error handling is a functional programming technique for dealing with errors in a clean, compositional, and type-safe way
- This approach encapsulates error handling into types and operations on these types
- Why is this desirable?
    - Type encapsulation: Monadic error handling encapsulates the result of computations along with possible errors within a single type
    - Compositional error handling: Monadic error handling allows composition of operations that might fail, in a way that if any operation fails, the whole computation fails

PICKNIK

Monadic Error Handling

- Monadic error handling is a functional programming technique for dealing with errors in a clean, compositional, and type-safe way
- This approach encapsulates error handling into types and operations on these types
- Why is this desirable?
  - Type encapsulation: Monadic error handling encapsulates the result of computations along with possible errors within a single type
  - Compositional error handling: Monadic error handling allows composition of operations that might fail, in a way that if any operation fails, the whole computation fails
  - Error Propagation: Errors can be automatically propagated through a sequence of computations until they are explicitly handled

Monadic Error Handling

- Monadic error handling is a functional programming technique for dealing with errors in a clean, compositional, and type-safe way
- This approach encapsulates error handling into types and operations on these types
- Why is this desirable?
  - Type encapsulation: Monadic error handling encapsulates the result of computations along with possible errors within a single type
  - Compositional error handling: Monadic error handling allows composition of operations that might fail, in a way that if any operation fails, the whole computation fails
  - Error Propagation: Errors can be automatically propagated through a sequence of computations until they are explicitly handled
- Let’s look at some ways to handle errors before looking at monadic error handling

Error Handling

```
int divide(int a, int b) { return a / b; }
int divide_try(int a, int b) {
    if (b == 0) throw std::domain_error("Denominator is zero");
    return a / b;
}
std::error_code divide_errc(int a, int b, int& out) {
    if (b == 0) return std::make_error_code(std::errc::invalid_argument);
    out = a / b;
    return {};
}
int main() {
  // No error handling
  std::cout << divide(4, 2) << "\n";
  // std::cout << divide(1, 0); // Program terminated with signal: SIGFPE
  // Exceptions
  try {
    std::cout << divide_try(1, 0);
  } catch (std::exception const& e) {
    std::cerr << e.what() << "\n";
  }

  // Error codes
  {
    int result;
    auto const error = divide_errc(1, 0, result);
    if (error) {
        std::cerr << error.message() << "\n";
    } else {
        std::cout << result;
    }
  }
}
```

- Here is code that shows some ways to handle errors

PICKNIK

Error Handling

```
int divide(int a, int b) { return a / b; }
int divide_try(int a, int b) {
    if (b == 0) throw std::domain_error("Denominator is zero");
    return a / b;
}
std::error_code divide_errc(int a, int b, int& out) {
    if (b == 0) return std::make_error_code(std::errc::invalid_argument);
    out = a / b;
    return {};
}
int main() {
  // No error handling
  std::cout << divide(4, 2) << "\n";
  // std::cout << divide(1, 0); // Program terminated with signal: SIGFPE
  // Exceptions
  try {
    std::cout << divide_try(1, 0);
  } catch (std::exception const& e) {
    std::cerr << e.what() << "\n";
  }

  // Error codes
  {
    int result;
    auto const error = divide_errc(1, 0, result);
    if (error) {
        std::cerr << error.message() << "\n";
    } else {
        std::cout << result;
    }
  }
}
```

- Here is code that shows some ways to handle errors
- The first way to “handle” an error is to do nothing, just let the program terminate

Error Handling

```
int divide(int a, int b) { return a / b; }
int divide_try(int a, int b) {
    if (b == 0) throw std::domain_error("Denominator is zero");
    return a / b;
}
std::error_code divide_errc(int a, int b, int& out) {
    if (b == 0) return std::make_error_code(std::errc::invalid_argument);
    out = a / b;
    return {};
}
int main() {
  // No error handling
  std::cout << divide(4, 2) << "\n";
  // std::cout << divide(1, 0); // Program terminated with signal: SIGFPE
  // Exceptions
  try {
    std::cout << divide_try(1, 0);
  } catch (std::exception const& e) {
    std::cerr << e.what() << "\n";
  }

  // Error codes
  {
    int result;
    auto const error = divide_errc(1, 0, result);
    if (error) {
        std::cerr << error.message() << "\n";
    } else {
        std::cout << result;
    }
  }
}
```

- Here is code that shows some ways to handle errors
- The first way to “handle” an error is to do nothing, just let the program terminate
- The method most people are familiar with is to throw an exception

PICKNIK

Error Handling

```cpp
int divide(int a, int b) { return a / b; }
int divide_try(int a, int b) {
    if (b == 0) throw std::domain_error("Denominator is zero");
    return a / b;
}
std::error_code divide_errc(int a, int b, int& out) {
    if (b == 0) return std::make_error_code(std::errc::invalid_argument);
    out = a / b;
    return {};
}
int main() {
  // No error handling
  std::cout << divide(4, 2) << "\n";
  // std::cout << divide(1, 0); // Program terminated with signal: SIGFPE
  // Exceptions
  try {
    std::cout << divide_try(1, 0);
  } catch (std::exception const& e) {
    std::cerr << e.what() << "\n";
  }

  // Error codes
  {
    int result;
    auto const error = divide_errc(1, 0, result);
    if (error) {
        std::cerr << error.message() << "\n";
    } else {
        std::cout << result;
    }
  }
}
```

- Here is code that shows some ways to handle errors
- The first way to “handle” an error is to do nothing, just let the program terminate
- The method most people are familiar with is to throw an exception
    - To make sure the program doesn’t terminate, the code that might throw an exception needs to be wrapped in a try catch block

PICKNIK

Error Handling

```cpp
int divide(int a, int b) { return a / b; }
int divide_try(int a, int b) {
    if (b == 0) throw std::domain_error("Denominator is zero");
    return a / b;
}
std::error_code divide_errc(int a, int b, int& out) {
    if (b == 0) return std::make_error_code(std::errc::invalid_argument);
    out = a / b;
    return {};
}
int main() {
  // No error handling
  std::cout << divide(4, 2) << "\n";
  // std::cout << divide(1, 0); // Program terminated with signal: SIGFPE
  // Exceptions
  try {
    std::cout << divide_try(1, 0);
  } catch (std::exception const& e) {
    std::cerr << e.what() << "\n";
  }
  // Error codes
  {
    int result;
    auto const error = divide_errc(1, 0, result);
    if (error) {
        std::cerr << error.message() << "\n";
    } else {
        std::cout << result;
    }
  }
}
```

- Here is code that shows some ways to handle errors
- The first way to "handle" an error is to do nothing, just let the program terminate
- The method most people are familiar with is to throw an exception
    - To make sure the program doesn't terminate, the code that might throw an exception needs to be wrapped in a try catch block
- Error codes are a concept where the function returns an error code and sets the result via an out parameter

PICKNIK

Error Handling

```cpp
std::tuple<std::error_code, int> divide_product(int a, int b) {
    if (b == 0) {
        return std::make_tuple(
            std::make_error_code(std::errc::invalid_argument), 0);
    }
    auto const result = a / b;
    return std::make_tuple(std::error_code{}, result);
}
std::optional<int> divide_maybe(int a, int b) {
    if (b == 0) return {};
    auto const result = a / b;
    return result;
}
int main() {
  {
    auto const [error, result] = divide_product(1, 0);
    if (error) {
      std::cerr << error.message() << "\n";
    } else {
      std::cout << result;
    }
    // Note that this is possible and the error can just be ignored
    // auto const [_, result] = divide_product(1, 0);
  }
  {
    auto const result = divide_maybe(1, 0);
    if (!result.has_value()) {
      std::cerr << "No result" << "\n";
    } else {
      std::cout << result.value();
    }
  }
}
```

- More intricate error handling is also possible using `std::tuple` and `std::optional`

Error Handling

```cpp
std::tuple<std::error_code, int> divide_product(int a, int b) {
    if (b == 0) {
        return std::make_tuple(
            std::make_error_code(std::errc::invalid_argument), 0);
    }
    auto const result = a / b;
    return std::make_tuple(std::error_code{}, result);
}
std::optional<int> divide_maybe(int a, int b) {
    if (b == 0) return {};
    auto const result = a / b;
    return result;
}
int main() {
  {
    auto const [error, result] = divide_product(1, 0);
    if (error) {
      std::cerr << error.message() << "\n";
    } else {
      std::cout << result;
    }
    // Note that this is possible and the error can just be ignored
    // auto const [_, result] = divide_product(1, 0);
  }
  {
    auto const result = divide_maybe(1, 0);
    if (!result.has_value()) {
      std::cerr << "No result" << "\n";
    } else {
      std::cout << result.value();
    }
  }
}
```

- More intricate error handling is also possible using `std::tuple` and `std::optional`
- With `std::tuple`, both an error code and result can be returned

Error Handling

```
std::tuple<std::error_code, int> divide_product(int a, int b) {
    if (b == 0) {
        return std::make_tuple(
            std::make_error_code(std::errc::invalid_argument), 0);
    }
    auto const result = a / b;
    return std::make_tuple(std::error_code{}, result);
}
std::optional<int> divide_maybe(int a, int b) {
    if (b == 0) return {};
    auto const result = a / b;
    return result;
}
int main() {
  {
    auto const [error, result] = divide_product(1, 0);
    if (error) {
      std::cerr << error.message() << "\n";
    } else {
      std::cout << result;
    }
    // Note that this is possible and the error can just be ignored
    // auto const [_, result] = divide_product(1, 0);
  }
  {
    auto const result = divide_maybe(1, 0);
    if (!result.has_value()) {
      std::cerr << "No result" << "\n";
    } else {
      std::cout << result.value();
    }
  }
}
```

- More intricate error handling is also possible using `std::tuple` and `std::optional`
- With `std::tuple`, both an error code and result can be returned
- With std::optional, the result either contains a value or is empty

Error Handling

```cpp
std::tuple<std::error_code, int> divide_product(int a, int b) {
    if (b == 0) {
        return std::make_tuple(
            std::make_error_code(std::errc::invalid_argument), 0);
    }
    auto const result = a / b;
    return std::make_tuple(std::error_code{}, result);
}
std::optional<int> divide_maybe(int a, int b) {
    if (b == 0) return {};
    auto const result = a / b;
    return result;
}
int main() {
  {
    auto const [error, result] = divide_product(1, 0);
    if (error) {
      std::cerr << error.message() << "\n";
    } else {
      std::cout << result;
    }
    // Note that this is possible and the error can just be ignored
    // auto const [_, result] = divide_product(1, 0);
  }
  {
    auto const result = divide_maybe(1, 0);
    if (!result.has_value()) {
      std::cerr << "No result" << "\n";
    } else {
      std::cout << result.value();
    }
  }
}
```

- More intricate error handling is also possible using `std::tuple` and `std::optional`
- With `std::tuple`, both an error code and result can be returned
- With std::optional, the result either contains a value or is empty
  - Is there a way to either return a value or an error code, and tailor future operations depending on the return?

PICKNIK

Monadic Error Handling

```
std::expected<int, std::error_code> divide(int a, int b) {
  if (b == 0) {
    return std::make_unexpected(
      std::make_error_code(std::errc::invalid_argument));
  }
  auto const result = a / b;
  return result;
}
std::expected<void, std::error_code> print_value(int const&
value) {
  std::cout << value;
  return {};
}
void print_error(std::error_code const& error) {
  std::cerr << error.message() << "\n";
}
int main() {
  // Conditional handling
  auto const result = divide(1, 0);
  if (!result.has_value()) {
    std::cerr << result.error().message() << "\n";
  } else {
    std::cout << result.value();
  }
  // Monadic functions
  divide(1, 0).and_then(print_value).or_else(print_error);
  return 0;
}
```

- Yes!

PICKNIK

Monadic Error Handling

```cpp
std::expected<int, std::error_code> divide(int a, int b) {
  if (b == 0) {
    return std::make_unexpected(
      std::make_error_code(std::errc::invalid_argument));
  }
  auto const result = a / b;
  return result;
}
std::expected<void, std::error_code> print_value(int const&
value) {
  std::cout << value;
  return {};
}
void print_error(std::error_code const& error) {
  std::cerr << error.message() << "\n";
}
int main() {
  // Conditional handling
  auto const result = divide(1, 0);
  if (!result.has_value()) {
    std::cerr << result.error().message() << "\n";
  } else {
    std::cout << result.value();
  }
  // Monadic functions
  divide(1, 0).and_then(print_value).or_else(print_error);
  return 0;
}
```

- Yes!
- Monadic error handling is the functional programming concept of returning a type that can either contain an expected value or an unexpected value and composing operations depending on that value

PICKNIK

Monadic Error Handling

```cpp
std::expected<int, std::error_code> divide(int a, int b) {
  if (b == 0) {
    return std::make_unexpected(
      std::make_error_code(std::errc::invalid_argument));
  }
  auto const result = a / b;
  return result;
}
std::expected<void, std::error_code> print_value(int const&
value) {
  std::cout << value;
  return {};
}
void print_error(std::error_code const& error) {
  std::cerr << error.message() << "\n";
}
int main() {
  // Conditional handling
  auto const result = divide(1, 0);
  if (!result.has_value()) {
    std::cerr << result.error().message() << "\n";
  } else {
    std::cout << result.value();
  }
  // Monadic functions
  divide(1, 0).and_then(print_value).or_else(print_error);
  return 0;
}
```

- Yes!
- Monadic error handling is the functional programming concept of returning a type that can either contain an expected value or an unexpected value and composing operations depending on that value
- `divide` returns a monadic type, which either contains an `int` or an `error_code`

PICKNIK

Monadic Error Handling

```
std::expected<int, std::error_code> divide(int a, int b) {
  if (b == 0) {
    return std::make_unexpected(
      std::make_error_code(std::errc::invalid_argument));
  }
  auto const result = a / b;
  return result;
}
std::expected<void, std::error_code> print_value(int const&
value) {
  std::cout << value;
  return {};
}
void print_error(std::error_code const& error) {
  std::cerr << error.message() << "\n";
}
int main() {
  // Conditional handling
  auto const result = divide(1, 0);
  if (!result.has_value()) {
    std::cerr << result.error().message() << "\n";
  } else {
    std::cout << result.value();
  }
  // Monadic functions
  divide(1, 0).and_then(print_value).or_else(print_error);
  return 0;
}
```

- Yes!
- Monadic error handling is the functional programming concept of returning a type that can either contain an expected value or an unexpected value and composing operations depending on that value
- `divide` returns a monadic type, which either contains an `int` or an `error_code`
- If the `divide` return contains an `int`, `and_then` takes the expected value and passes it to `print_value`

PICKNIK

Monadic Error Handling

```
std::expected<int, std::error_code> divide(int a, int b) {
  if (b == 0) {
    return std::make_unexpected(
      std::make_error_code(std::errc::invalid_argument));
  }
  auto const result = a / b;
  return result;
}
std::expected<void, std::error_code> print_value(int const&
value) {
  std::cout << value;
  return {};
}
void print_error(std::error_code const& error) {
  std::cerr << error.message() << "\n";
}
int main() {
  // Conditional handling
  auto const result = divide(1, 0);
  if (!result.has_value()) {
    std::cerr << result.error().message() << "\n";
  } else {
    std::cout << result.value();
  }
  // Monadic functions
  divide(1, 0).and_then(print_value).or_else(print_error);
  return 0;
}
```

- Yes!
- Monadic error handling is the functional programming concept of returning a type that can either contain an expected value or an unexpected value and composing operations depending on that value
- `divide` returns a monadic type, which either contains an `int` or an `error_code`
- If the `divide` return contains an `int`, `and_then` takes the expected value and passes it to `print_value`
- If the `divide` return contains an `error_code`, `or_else` takes the unexpected value and passes it to `print_error`

PICKNIK

Monadic Error Handling

```cpp
std::expected<int, std::error_code> divide(int a, int b) {
  if (b == 0) {
    return std::make_unexpected(
      std::make_error_code(std::errc::invalid_argument));
  }
  auto const result = a / b;
  return result;
}
std::expected<void, std::error_code> print_value(int const&
value) {
  std::cout << value;
  return {};
}
void print_error(std::error_code const& error) {
  std::cerr << error.message() << "\n";
}
int main() {
  // Conditional handling
  auto const result = divide(1, 0);
  if (!result.has_value()) {
    std::cerr << result.error().message() << "\n";
  } else {
    std::cout << result.value();
  }
  // Monadic functions
  divide(1, 0).and_then(print_value).or_else(print_error);
  return 0;
}
```

- Yes!
- Monadic error handling is the functional programming concept of returning a type that can either contain an expected value or an unexpected value and composing operations depending on that value
- `divide` returns a monadic type, which either contains an `int` or an `error_code`
- If the `divide` return contains an `int`, `and_then` takes the expected value and passes it to `print_value`
- If the `divide` return contains an `error_code`, `or_else` takes the unexpected value and passes it to `print_error`
- This method of chaining operations is fundamental to functional programming

PICKNIK

How does functional programming help?

- Functional programming lends itself to the minimization of mutable state

How does functional programming help?

- Functional programming lends itself to the minimization of mutable state
- Testing is easier with pure functions because only the return value of the function needs to be evaluated

PICKNIK

How does functional programming help?

- Functional programming lends itself to the minimization of mutable state
- Testing is easier with pure functions because only the return value of the function needs to be evaluated
- Different functions can be passed as arguments to higher order functions, lending itself to modularity

PICKNIK

How does functional programming help?

- Functional programming lends itself to the minimization of mutable state
- Testing is easier with pure functions because only the return value of the function needs to be evaluated
- Different functions can be passed as arguments to higher order functions, lending itself to modularity
- Monadic error handling simplifies error checking

PICKNIK

How does functional programming help?

- Functional programming lends itself to the minimization of mutable state
- Testing is easier with pure functions because only the return value of the function needs to be evaluated
- Different functions can be passed as arguments to higher order functions, lending itself to modularity
- Monadic error handling simplifies error checking
- Let’s try and refactor `PathGenerator`

PICKNIK

How does functional programming help?

- Functional programming lends itself to the minimization of mutable state
- Testing is easier with pure functions because only the return value of the function needs to be evaluated
- Different functions can be passed as arguments to higher order functions, lending itself to modularity
- Monadic error handling simplifies error checking
- Let's try and refactor `PathGenerator`
    - **Claim: that the refactored `PathGenerator` has 100% coverage**

PICKNIK

Refactoring using Functional Programming Principles

Refactoring PathGenerator

```
class PathGenerator : public rclcpp::Node {
 public:
  explicit PathGenerator(rclcpp::NodeOptions const&
                         options = rclcpp::NodeOptions{})
          : Node("path_generator", options);

 private:
  void set_map_service(
     const std::shared_ptr<example_srvs::srv::SetMap::Request> request,
     std::shared_ptr<example_srvs::srv::SetMap::Response> response);

  void generate_path_service(
     const std::shared_ptr<example_srvs::srv::GetPath::Request> request,
     std::shared_ptr<example_srvs::srv::GetPath::Response> response);

  bool set_costmap(const std_msgs::msg::UInt8MultiArray& costmap);

  Path generate_global_path(Position const& start,Position const& goal);

  /* Additional private members*/
};
```

- How the current `PathGenerator` looks

Refactoring PathGenerator

```
class PathGenerator : public rclcpp::Node {
 public:
  explicit PathGenerator(rclcpp::NodeOptions const&
                         options = rclcpp::NodeOptions{})
          : Node("path_generator", options);

 private:
  void set_map_service(
     const std::shared_ptr<example_srvs::srv::SetMap::Request> request,
     std::shared_ptr<example_srvs::srv::SetMap::Response> response);

  void generate_path_service(
     const std::shared_ptr<example_srvs::srv::GetPath::Request> request,
     std::shared_ptr<example_srvs::srv::GetPath::Response> response);

  bool set_costmap(const std_msgs::msg::UInt8MultiArray& costmap);

  Path generate_global_path(Position const& start,Position const& goal);

  /* Additional private members*/
};
```

- A `rclcpp::Node` object can be constructed in `main` and services can be assigned
- No need to have a `PathGenerator` object that inherits from `rclcpp::Node`

PICKNIK

Refactoring PathGenerator

```
class PathGenerator : public rclcpp::Node {
 public:
  explicit PathGenerator(rclcpp::NodeOptions const&
                         options = rclcpp::NodeOptions{})
          : Node("path_generator", options);

 private:
  void set_map_service(
     const std::shared_ptr<example_srvs::srv::SetMap::Request> request,
     std::shared_ptr<example_srvs::srv::SetMap::Response> response);

  void generate_path_service(
     const std::shared_ptr<example_srvs::srv::GetPath::Request> request,
     std::shared_ptr<example_srvs::srv::GetPath::Response> response);

  bool set_costmap(const std_msgs::msg::UInt8MultiArray& costmap);

  Path generate_global_path(Position const& start,Position const& goal);

  /* Additional private members*/
};
```

- A `rclcpp::Node` object can be constructed in `main` and services can be assigned
- No need to have a `PathGenerator` object that inherits from `rclcpp::Node`
- The `create_service` method accepts class methods, free functions, and lambdas as the callback function
- The private functions of `PathGenerator` can be turned into free functions and lambda functions

Refactoring PathGenerator

```
class PathGenerator : public rclcpp::Node {
 public:
  explicit PathGenerator(rclcpp::NodeOptions const&
                         options = rclcpp::NodeOptions{})
          : Node("path_generator", options);

 private:
  void set_map_service(
     const std::shared_ptr<example_srvs::srv::SetMap::Request> request,
     std::shared_ptr<example_srvs::srv::SetMap::Response> response);

  void generate_path_service(
     const std::shared_ptr<example_srvs::srv::GetPath::Request> request,
     std::shared_ptr<example_srvs::srv::GetPath::Response> response);

  bool set_costmap(const std_msgs::msg::UInt8MultiArray& costmap);

  Path generate_global_path(Position const& start,Position const& goal);

  /* Additional private members*/
};
```

- A `rclcpp::Node` object can be constructed in `main` and services can be assigned
- No need to have a `PathGenerator` object that inherits from `rclcpp::Node`
- The `create_service` method accepts class methods, free functions, and lambdas as the callback function
- The private functions of `PathGenerator` can be turned into free functions and lambda functions
- Let's refactor the callback function for the generate path service

Refactoring PathGenerator

```
void generate_path_service(
const std::shared_ptr<example_srvs::srv::GetPath::Request> request,
      std::shared_ptr<example_srvs::srv::GetPath::Response> response) {
  if (map_.get_data().size() == 0) {
    RCLCPP_ERROR_STREAM(this->get_logger(), "MAP IS EMPTY!!");
    response->code.code = example_srvs::msg::GetPathCodes::EMPTY_OCCUPANCY_MAP;
    response->path = std_msgs::msg::UInt8MultiArray();
    return;
  }
  /* More error pre-checks */

  auto const start = Position{request->start.data[0], request->start.data[1]};
  auto const goal = Position{request->goal.data[0], request->goal.data[1]};

  // Generate the path
  auto const path = generate_global_path(start, goal);

  // Start populating the response message
  auto response_path = std_msgs::msg::UInt8MultiArray();

  /* Code about populating the message here */

  response->code.code = !path.empty() ? example_srvs::msg::GetPathCodes::SUCCESS :
example_srvs::msg::GetPathCodes::NO_VALID_PATH;
    response->path = response_path;
  }
```

Refactoring PathGenerator

```cpp
void generate_path_service(
const std::shared_ptr<example_srvs::srv::GetPath::Request> request,
      std::shared_ptr<example_srvs::srv::GetPath::Response> response) {
  if (map_.get_data().size() == 0) {
    RCLCPP_ERROR_STREAM(this->get_logger(), "MAP IS EMPTY!!");
    response->code.code = example_srvs::msg::GetPathCodes::EMPTY_OCCUPANCY_MAP;
    response->path = std_msgs::msg::UInt8MultiArray();
    return;
  }
  /* More error pre-checks */

  auto const start = Position{request->start.data[0], request->start.data[1]};
  auto const goal = Position{request->goal.data[0], request->goal.data[1]};

  // Generate the path
  auto const path = generate_global_path(start, goal);

  // Start populating the response message
  auto response_path = std_msgs::msg::UInt8MultiArray();

  /* Code about populating the message here */

  response->code.code = !path.empty() ? example_srvs::msg::GetPathCodes::SUCCESS :
example_srvs::msg::GetPathCodes::NO_VALID_PATH;
    response->path = response_path;
  }
```

- generate_path_service is:
  - printing errors

Refactoring PathGenerator

```
void generate_path_service(
const std::shared_ptr<example_srvs::srv::GetPath::Request> request,
      std::shared_ptr<example_srvs::srv::GetPath::Response> response) {
  if (map_.get_data().size() == 0) {
    RCLCPP_ERROR_STREAM(this->get_logger(), "MAP IS EMPTY!!");
    response->code.code = example_srvs::msg::GetPathCodes::EMPTY_OCCUPANCY_MAP;
    response->path = std_msgs::msg::UInt8MultiArray();
    return;
  }
  /* More error pre-checks */

  auto const start = Position{request->start.data[0], request->start.data[1]};
  auto const goal = Position{request->goal.data[0], request->goal.data[1]};

  // Generate the path
  auto const path = generate_global_path(start, goal);

  // Start populating the response message
  auto response_path = std_msgs::msg::UInt8MultiArray();

  /* Code about populating the message here */

  response->code.code = !path.empty() ? example_srvs::msg::GetPathCodes::SUCCESS :
example_srvs::msg::GetPathCodes::NO_VALID_PATH;
    response->path = response_path;
  }
```

- `generate_path_service` is:
    - printing errors
    - generating the path

Refactoring PathGenerator

```
void generate_path_service(
const std::shared_ptr<example_srvs::srv::GetPath::Request> request,
      std::shared_ptr<example_srvs::srv::GetPath::Response> response) {
  if (map_.get_data().size() == 0) {
    RCLCPP_ERROR_STREAM(this->get_logger(), "MAP IS EMPTY!!");
    response->code.code = example_srvs::msg::GetPathCodes::EMPTY_OCCUPANCY_MAP;
    response->path = std_msgs::msg::UInt8MultiArray();
    return;
  }
  /* More error pre-checks */

  auto const start = Position{request->start.data[0], request->start.data[1]};
  auto const goal = Position{request->goal.data[0], request->goal.data[1]};

  // Generate the path
  auto const path = generate_global_path(start, goal);

  // Start populating the response message
  auto response_path = std_msgs::msg::UInt8MultiArray();

  /* Code about populating the message here */

  response->code.code = !path.empty() ? example_srvs::msg::GetPathCodes::SUCCESS :
example_srvs::msg::GetPathCodes::NO_VALID_PATH;
    response->path = response_path;
  }
```

- `generate_path_service` is:
  - printing errors
  - generating the path
  - setting an out parameter

Refactoring PathGenerator

```
void generate_path_service(
const std::shared_ptr<example_srvs::srv::GetPath::Request> request,
      std::shared_ptr<example_srvs::srv::GetPath::Response> response) {
  if (map_.get_data().size() == 0) {
    RCLCPP_ERROR_STREAM(this->get_logger(), "MAP IS EMPTY!!");
    response->code.code = example_srvs::msg::GetPathCodes::EMPTY_OCCUPANCY_MAP;
    response->path = std_msgs::msg::UInt8MultiArray();
    return;
  }
  /* More error pre-checks */

  auto const start = Position{request->start.data[0], request->start.data[1]};
  auto const goal = Position{request->goal.data[0], request->goal.data[1]};

  // Generate the path
  auto const path = generate_global_path(start, goal);

  // Start populating the response message
  auto response_path = std_msgs::msg::UInt8MultiArray();

  /* Code about populating the message here */

  response->code.code = !path.empty() ? example_srvs::msg::GetPathCodes::SUCCESS :
example_srvs::msg::GetPathCodes::NO_VALID_PATH;
    response->path = response_path;
  }
```

- `generate_path_service` is:
  - printing errors
  - generating the path
  - setting an out parameter
- Let's isolate the error printing functionality to another function
  - The error printing function needs to be passed an error type

Refactoring PathGenerator

```
void generate_path_service(
const std::shared_ptr<example_srvs::srv::GetPath::Request> request,
    std::shared_ptr<example_srvs::srv::GetPath::Response> response) {
  if (map_.get_data().size() == 0) {
    RCLCPP_ERROR_STREAM(this->get_logger(), "MAP IS EMPTY!!");
    response->code.code = example_srvs::msg::GetPathCodes::EMPTY_OCCUPANCY_MAP;
    response->path = std_msgs::msg::UInt8MultiArray();
    return;
  }
  /* More error pre-checks */

  auto const start = Position{request->start.data[0], request->start.data[1]};
  auto const goal = Position{request->goal.data[0], request->goal.data[1]};

  // Generate the path
  auto const path = generate_global_path(start, goal);

  // Start populating the response message
  auto response_path = std_msgs::msg::UInt8MultiArray();

  /* Code about populating the message here */

  response->code.code = !path.empty() ? example_srvs::msg::GetPathCodes::SUCCESS :
example_srvs::msg::GetPathCodes::NO_VALID_PATH;
    response->path = response_path;
  }
```

- `generate_path_service` is:
  - printing errors
  - generating the path
  - setting an out parameter
- Let's isolate the error printing functionality to another function
  - The error printing function needs to be passed an error type
- The object held by the shared pointer can be assigned by another function

Refactoring PathGenerator

```
void generate_path_service(
const std::shared_ptr<example_srvs::srv::GetPath::Request> request,
      std::shared_ptr<example_srvs::srv::GetPath::Response> response) {
  if (map_.get_data().size() == 0) {
    RCLCPP_ERROR_STREAM(this->get_logger(), "MAP IS EMPTY!!");
    response->code.code = example_srvs::msg::GetPathCodes::EMPTY_OCCUPANCY_MAP;
    response->path = std_msgs::msg::UInt8MultiArray();
    return;
  }
  /* More error pre-checks */

  auto const start = Position{request->start.data[0], request->start.data[1]};
  auto const goal = Position{request->goal.data[0], request->goal.data[1]};

  // Generate the path
  auto const path = generate_global_path(start, goal);

  // Start populating the response message
  auto response_path = std_msgs::msg::UInt8MultiArray();

  /* Code about populating the message here */

  response->code.code = !path.empty() ? example_srvs::msg::GetPathCodes::SUCCESS :
example_srvs::msg::GetPathCodes::NO_VALID_PATH;
    response->path = response_path;
  }
```

- `generate_path_service` is:
  - printing errors
  - generating the path
  - setting an out parameter
- Let's isolate the error printing functionality to another function
  - The error printing function needs to be passed an error type
- The object held by the shared pointer can be assigned by another function
- The `generate_global_path` function and associated pre-checks can be extracted to another function

Refactoring PathGenerator

```
std::expected<example_srvs::srv::GetPath::Response, error> generate_path(
  std::shared_ptr<example_srvs::srv::GetPath::Request> const request,
  Map<unsigned char> const& occupancy_map, PathingGenerator path_generator) {
  if (occupancy_map.get_data().size() == 0) {
    return std::unexpected(error::EMPTY_OCCUPANCY_MAP);
  }
  /* More error pre-checks */

  auto const start = Position{request->start.data[0], request->start.data[1]};
  auto const goal = Position{request->goal.data[0], request->goal.data[1]};

  // Generate the path using the path generator function that was input
  auto const path = path_generator(start, goal, occupancy_map);
  if (!path.has_value()) {
    return std::unexpected(error::NO_VALID_PATH);
  }

  auto response = example_srvs::srv::GetPath::Response{};
  /* More implementation code */
  return response;
}
```

- Here is the refactored core functionality of the generate path callback

Refactoring PathGenerator

```
std::expected<example_srvs::srv::GetPath::Response, error> generate_path(
  std::shared_ptr<example_srvs::srv::GetPath::Request> const request,
  Map<unsigned char> const& occupancy_map, PathingGenerator path_generator) {
  if (occupancy_map.get_data().size() == 0) {
    return std::unexpected(error::EMPTY_OCCUPANCY_MAP);
  }
  /* More error pre-checks */

  auto const start = Position{request->start.data[0], request->start.data[1]};
  auto const goal = Position{request->goal.data[0], request->goal.data[1]};

  // Generate the path using the path generator function that was input
  auto const path = path_generator(start, goal, occupancy_map);
  if (!path.has_value()) {
    return std::unexpected(error::NO_VALID_PATH);
  }

  auto response = example_srvs::srv::GetPath::Response{};
  /* More implementation code */
  return response;
}
```

- Here is the refactored core functionality of the generate path callback
- This function returns a type which can be used for **monadic error handling**

PICKNIK

Refactoring PathGenerator

```
std::expected<example_srvs::srv::GetPath::Response, error> generate_path(
  std::shared_ptr<example_srvs::srv::GetPath::Request> const request,
  Map<unsigned char> const& occupancy_map, PathingGenerator path_generator) {
  if (occupancy_map.get_data().size() == 0) {
    return std::unexpected(error::EMPTY_OCCUPANCY_MAP);
  }
  /* More error pre-checks */

  auto const start = Position{request->start.data[0], request->start.data[1]};
  auto const goal = Position{request->goal.data[0], request->goal.data[1]};

  // Generate the path using the path generator function that was input
  auto const path = path_generator(start, goal, occupancy_map);
  if (!path.has_value()) {
    return std::unexpected(error::NO_VALID_PATH);
  }

  auto response = example_srvs::srv::GetPath::Response{};
  /* More implementation code */
  return response;
}
```

- Here is the refactored core functionality of the generate path callback
- This function returns a type which can be used for **monadic error handling**
- If there is an error, the function can handle the error in a compile time checkable way

Refactoring PathGenerator

```
using PathingGenerator = std::function<std::optional<Path>(
    Position const&, Position const&, Map<unsigned char> const&)>;

std::expected<example_srvs::srv::GetPath::Response, error> generate_path(
  std::shared_ptr<example_srvs::srv::GetPath::Request> const request,
  Map<unsigned char> const& occupancy_map, PathingGenerator path_generator) {
  if (occupancy_map.get_data().size() == 0) {
    return std::unexpected(error::EMPTY_OCCUPANCY_MAP);
  }
  /* More error pre-checks */

  auto const start = Position{request->start.data[0], request->start.data[1]};
  auto const goal = Position{request->goal.data[0], request->goal.data[1]};

  // Generate the path using the path generator function that was input
  auto const path = path_generator(start, goal, occupancy_map);
  if (!path.has_value()) {
    return std::unexpected(error::NO_VALID_PATH);
  }

  auto response = example_srvs::srv::GetPath::Response{};
  /* More implementation code */
  return response;
}
```

- Here is the refactored core functionality of the generate path callback
- This function returns a type which can be used for **monadic error handling**
- If there is an error, the function can handle the error in a compile time checkable way
- The function that generates the path can now be passed in, making this function a **higher order function**

PICKNIK

Refactoring PathGenerator

```
using PathingGenerator = std::function<std::optional<Path>(
    Position const&, Position const&, Map<unsigned char> const&)>;

std::expected<example_srvs::srv::GetPath::Response, error> generate_path(
  std::shared_ptr<example_srvs::srv::GetPath::Request> const request,
  Map<unsigned char> const& occupancy_map, PathingGenerator path_generator) {
  if (occupancy_map.get_data().size() == 0) {
    return std::unexpected(error::EMPTY_OCCUPANCY_MAP);
  }
  /* More error pre-checks */

  auto const start = Position{request->start.data[0], request->start.data[1]};
  auto const goal = Position{request->goal.data[0], request->goal.data[1]};

  // Generate the path using the path generator function that was input
  auto const path = path_generator(start, goal, occupancy_map);
  if (!path.has_value()) {
    return std::unexpected(error::NO_VALID_PATH);
  }

  auto response = example_srvs::srv::GetPath::Response{};
  /* More implementation code */
  return response;
}
```

- Here is the refactored core functionality of the generate path callback
- This function returns a type which can be used for **monadic error handling**
- If there is an error, the function can handle the error in a compile time checkable way
- The function that generates the path can now be passed in, making this function a **higher order function**
- This function is deterministic and has no side effects, so it is a **pure function**

PICKNIK

Refactoring PathGenerator

```
using PathingGenerator = std::function<std::optional<Path>(
    Position const&, Position const&, Map<unsigned char> const&)>;

std::expected<example_srvs::srv::GetPath::Response, error> generate_path(
  std::shared_ptr<example_srvs::srv::GetPath::Request> const request,
  Map<unsigned char> const& occupancy_map, PathingGenerator path_generator) {
  if (occupancy_map.get_data().size() == 0) {
    return std::unexpected(error::EMPTY_OCCUPANCY_MAP);
  }
  /* More error pre-checks */

  auto const start = Position{request->start.data[0], request->start.data[1]};
  auto const goal = Position{request->goal.data[0], request->goal.data[1]};

  // Generate the path using the path generator function that was input
  auto const path = path_generator(start, goal, occupancy_map);
  if (!path.has_value()) {
    return std::unexpected(error::NO_VALID_PATH);
  }

  auto response = example_srvs::srv::GetPath::Response{};
  /* More implementation code */
  return response;
}
```

- Here is the refactored core functionality of the generate path callback
- This function returns a type which can be used for **monadic error handling**
- If there is an error, the function can handle the error in a compile time checkable way
- The function that generates the path can now be passed in, making this function a **higher order function**
- This function is deterministic and has no side effects, so it is a **pure function**
- Let's test this function

PICKNIK

Testing the Refactored PathGenerator

```
TEST(GeneratePath, NoValidPath) {
  // GIVEN a GetPath request and an occupancy map
  auto const sample_occupancy_map = get_test_occupancy_map();

  auto const request = std::make_shared<example_srvs::srv::GetPath::Request>();

  request->start.data = {2, 2};
  request->goal.data = {5, 5};

  // WHEN the path is requested
  auto const response = pathing::generate_path::generate_path(
      request, sample_occupancy_map, pathing::generate_global_path);

  // THEN there should be an error with the error::NO_VALID_PATH type
  EXPECT_EQ(response.error(), pathing::generate_path::error::NO_VALID_PATH);
}

TEST(GeneratePath, PathGenerated) {
  // GIVEN a GetPath request and an occupancy map
  auto const sample_occupancy_map = get_test_occupancy_map();

  auto const request = std::make_shared<example_srvs::srv::GetPath::Request>();

  request->start.data = {0, 0};
  request->goal.data = {7, 7};

  // WHEN the path is requested
  auto const response = pathing::generate_path::generate_path(
      request, sample_occupancy_map, pathing::generate_global_path);

  // THEN there should no errors
  EXPECT_TRUE(response.has_value());
}
```

PICKNIK

Testing the Refactored PathGenerator

```cpp
TEST(GeneratePath, NoValidPath) {
  // GIVEN a GetPath request and an occupancy map
  auto const sample_occupancy_map = get_test_occupancy_map();

  auto const request = std::make_shared<example_srvs::srv::GetPath::Request>();

  request->start.data = {2, 2};
  request->goal.data = {5, 5};

  // WHEN the path is requested
  auto const response = pathing::generate_path::generate_path(
      request, sample_occupancy_map, pathing::generate_global_path);

  // THEN there should be an error with the error::NO_VALID_PATH type
  EXPECT_EQ(response.error(), pathing::generate_path::error::NO_VALID_PATH);
}

TEST(GeneratePath, PathGenerated) {
  // GIVEN a GetPath request and an occupancy map
  auto const sample_occupancy_map = get_test_occupancy_map();

  auto const request = std::make_shared<example_srvs::srv::GetPath::Request>();

  request->start.data = {0, 0};
  request->goal.data = {7, 7};

  // WHEN the path is requested
  auto const response = pathing::generate_path::generate_path(
      request, sample_occupancy_map, pathing::generate_global_path);

  // THEN there should no errors
  EXPECT_TRUE(response.has_value());
}
```

- Testing the refactored functionality is trivial

PICKNIK

Testing the Refactored PathGenerator

```cpp
TEST(GeneratePath, NoValidPath) {
  // GIVEN a GetPath request and an occupancy map
  auto const sample_occupancy_map = get_test_occupancy_map();

  auto const request = std::make_shared<example_srvs::srv::GetPath::Request>();

  request->start.data = {2, 2};
  request->goal.data = {5, 5};

  // WHEN the path is requested
  auto const response = pathing::generate_path::generate_path(
      request, sample_occupancy_map, pathing::generate_global_path);

  // THEN there should be an error with the error::NO_VALID_PATH type
  EXPECT_EQ(response.error(), pathing::generate_path::error::NO_VALID_PATH);
}

TEST(GeneratePath, PathGenerated) {
  // GIVEN a GetPath request and an occupancy map
  auto const sample_occupancy_map = get_test_occupancy_map();

  auto const request = std::make_shared<example_srvs::srv::GetPath::Request>();

  request->start.data = {0, 0};
  request->goal.data = {7, 7};

  // WHEN the path is requested
  auto const response = pathing::generate_path::generate_path(
      request, sample_occupancy_map, pathing::generate_global_path);

  // THEN there should no errors
  EXPECT_TRUE(response.has_value());
}
```

- Testing the refactored functionality is trivial
  - Create required parameters

PICKNIK

Testing the Refactored PathGenerator

```
TEST(GeneratePath, NoValidPath) {
  // GIVEN a GetPath request and an occupancy map
  auto const sample_occupancy_map = get_test_occupancy_map();

  auto const request = std::make_shared<example_srvs::srv::GetPath::Request>();

  request->start.data = {2, 2};
  request->goal.data = {5, 5};

  // WHEN the path is requested
  auto const response = pathing::generate_path::generate_path(
      request, sample_occupancy_map, pathing::generate_global_path);

  // THEN there should be an error with the error::NO_VALID_PATH type
  EXPECT_EQ(response.error(), pathing::generate_path::error::NO_VALID_PATH);
}

TEST(GeneratePath, PathGenerated) {
  // GIVEN a GetPath request and an occupancy map
  auto const sample_occupancy_map = get_test_occupancy_map();

  auto const request = std::make_shared<example_srvs::srv::GetPath::Request>();

  request->start.data = {0, 0};
  request->goal.data = {7, 7};

  // WHEN the path is requested
  auto const response = pathing::generate_path::generate_path(
      request, sample_occupancy_map, pathing::generate_global_path);

  // THEN there should no errors
  EXPECT_TRUE(response.has_value());
}
```

- Testing the refactored functionality is trivial
  - Create required parameters
  - Pass the parameters into the function under test

PICKNIK

Testing the Refactored PathGenerator

```
TEST(GeneratePath, NoValidPath) {
  // GIVEN a GetPath request and an occupancy map
  auto const sample_occupancy_map = get_test_occupancy_map();

  auto const request = std::make_shared<example_srvs::srv::GetPath::Request>();

  request->start.data = {2, 2};
  request->goal.data = {5, 5};

  // WHEN the path is requested
  auto const response = pathing::generate_path::generate_path(
      request, sample_occupancy_map, pathing::generate_global_path);

  // THEN there should be an error with the error::NO_VALID_PATH type
  EXPECT_EQ(response.error(), pathing::generate_path::error::NO_VALID_PATH);
}

TEST(GeneratePath, PathGenerated) {
  // GIVEN a GetPath request and an occupancy map
  auto const sample_occupancy_map = get_test_occupancy_map();

  auto const request = std::make_shared<example_srvs::srv::GetPath::Request>();

  request->start.data = {0, 0};
  request->goal.data = {7, 7};

  // WHEN the path is requested
  auto const response = pathing::generate_path::generate_path(
      request, sample_occupancy_map, pathing::generate_global_path);

  // THEN there should no errors
  EXPECT_TRUE(response.has_value());
}
```

- Testing the refactored functionality is trivial
  - Create required parameters
  - Pass the parameters into the function under test
  - Check the return

PICKNIK

Testing the Refactored PathGenerator

```cpp
TEST(GeneratePath, NoValidPath) {
  // GIVEN a GetPath request and an occupancy map
  auto const sample_occupancy_map = get_test_occupancy_map();

  auto const request = std::make_shared<example_srvs::srv::GetPath::Request>();

  request->start.data = {2, 2};
  request->goal.data = {5, 5};

  // WHEN the path is requested
  auto const response = pathing::generate_path::generate_path(
      request, sample_occupancy_map, pathing::generate_global_path);

  // THEN there should be an error with the error::NO_VALID_PATH type
  EXPECT_EQ(response.error(), pathing::generate_path::error::NO_VALID_PATH);
}

TEST(GeneratePath, PathGenerated) {
  // GIVEN a GetPath request and an occupancy map
  auto const sample_occupancy_map = get_test_occupancy_map();

  auto const request = std::make_shared<example_srvs::srv::GetPath::Request>();

  request->start.data = {0, 0};
  request->goal.data = {7, 7};

  // WHEN the path is requested
  auto const response = pathing::generate_path::generate_path(
      request, sample_occupancy_map, pathing::generate_global_path);

  // THEN there should no errors
  EXPECT_TRUE(response.has_value());
}
```

- Testing the refactored functionality is trivial
    - Create required parameters
    - Pass the parameters into the function under test
    - Check the return
- All the functions that have been refactored so far can be tested this way

PICKNIK

Testing the Refactored PathGenerator

```
TEST(GeneratePath, NoValidPath) {
  // GIVEN a GetPath request and an occupancy map
  auto const sample_occupancy_map = get_test_occupancy_map();

  auto const request = std::make_shared<example_srvs::srv::GetPath::Request>();

  request->start.data = {2, 2};
  request->goal.data = {5, 5};

  // WHEN the path is requested
  auto const response = pathing::generate_path::generate_path(
      request, sample_occupancy_map, pathing::generate_global_path);

  // THEN there should be an error with the error::NO_VALID_PATH type
  EXPECT_EQ(response.error(), pathing::generate_path::error::NO_VALID_PATH);
}

TEST(GeneratePath, PathGenerated) {
  // GIVEN a GetPath request and an occupancy map
  auto const sample_occupancy_map = get_test_occupancy_map();

  auto const request = std::make_shared<example_srvs::srv::GetPath::Request>();

  request->start.data = {0, 0};
  request->goal.data = {7, 7};

  // WHEN the path is requested
  auto const response = pathing::generate_path::generate_path(
      request, sample_occupancy_map, pathing::generate_global_path);

  // THEN there should no errors
  EXPECT_TRUE(response.has_value());
}
```

- Testing the refactored functionality is trivial
  - Create required parameters
  - Pass the parameters into the function under test
  - Check the return
- All the functions that have been refactored so far can be tested this way
- Everything can now be put together for the callback being executed by the generate path service

PICKNIK

Testing the Refactored PathGenerator

```
TEST(GeneratePath, NoValidPath) {
  // GIVEN a GetPath request and an occupancy map
  auto const sample_occupancy_map = get_test_occupancy_map();

  auto const request = std::make_shared<example_srvs::srv::GetPath::Request>();

  request->start.data = {2, 2};
  request->goal.data = {5, 5};

  // WHEN the path is requested
  auto const response = pathing::generate_path::generate_path(
      request, sample_occupancy_map, pathing::generate_global_path);

  // THEN there should be an error with the error::NO_VALID_PATH type
  EXPECT_EQ(response.error(), pathing::generate_path::error::NO_VALID_PATH);
}

TEST(GeneratePath, PathGenerated) {
  // GIVEN a GetPath request and an occupancy map
  auto const sample_occupancy_map = get_test_occupancy_map();

  auto const request = std::make_shared<example_srvs::srv::GetPath::Request>();

  request->start.data = {0, 0};
  request->goal.data = {7, 7};

  // WHEN the path is requested
  auto const response = pathing::generate_path::generate_path(
      request, sample_occupancy_map, pathing::generate_global_path);

  // THEN there should no errors
  EXPECT_TRUE(response.has_value());
}
```

- Testing the refactored functionality is trivial
  - Create required parameters
  - Pass the parameters into the function under test
  - Check the return
- All the functions that have been refactored so far can be tested this way
- Everything can now be put together for the callback being executed by the generate path service
- **All of this has been done without invoking the ROS 2 API!**

PICKNIK

Putting it all together

```
[this](auto const request, auto response) {
    auto const print_error = [this](std::string_view error)
        -> std::expected<GetPath::Response, std::string> {...};

    auto const return_empty_response = []([[maybe_unused]] auto const)
        -> std::expected<GetPath::Response, std::string> {...};

    auto const stringify_error = [](auto const error) {...};

    *response = generate_path::generate_path(request, this->map_,
generate_global_path)
            .map_error(stringify_error)
            .or_else(print_error)
            .or_else(return_empty_response)
            .value();
  }
```

- The generate path callback function has been replaced by a lambda function

Putting it all together

```
[this](auto const request, auto response) {
    auto const print_error = [this](std::string_view error)
        -> std::expected<GetPath::Response, std::string> {...};

    auto const return_empty_response = []([[maybe_unused]] auto const)
        -> std::expected<GetPath::Response, std::string> {...};

    auto const stringify_error = [](auto const error) {...};

    *response = generate_path::generate_path(request, this->map_,
generate_global_path)
            .map_error(stringify_error)
            .or_else(print_error)
            .or_else(return_empty_response)
            .value();
  }
```

- The generate path callback function has been replaced by a lambda function
- If `generate_path` returns the expected value, it is directly assigned to `response`

PICKNIK

Putting it all together

```
[this](auto const request, auto response) {
    auto const print_error = [this](std::string_view error)
        -> std::expected<GetPath::Response, std::string> {...};

    auto const return_empty_response = []([[maybe_unused]] auto const)
        -> std::expected<GetPath::Response, std::string> {...};

    auto const stringify_error = [](auto const error) {...};

    *response = generate_path::generate_path(request, this->map_,
generate_global_path)
            .map_error(stringify_error)
            .or_else(print_error)
            .or_else(return_empty_response)
            .value();
  }
```

- The generate path callback function has been replaced by a lambda function
- If `generate_path` returns the expected value, it is directly assigned to `response`
- If `generate_path` returns an error, the error is handled by chaining functions together
    - This is the result of returning a monadic type and performing monadic error handling

PICKNIK

Putting it all together

```
[this](auto const request, auto response) {
    auto const print_error = [this](std::string_view error)
        -> std::expected<GetPath::Response, std::string> {...};

    auto const return_empty_response = []([[maybe_unused]] auto const)
        -> std::expected<GetPath::Response, std::string> {...};

    auto const stringify_error = [](auto const error) {...};

    *response = generate_path::generate_path(request, this->map_,
generate_global_path)
            .map_error(stringify_error)
            .or_else(print_error)
            .or_else(return_empty_response)
            .value();
  }
```

- The generate path callback function has been replaced by a lambda function
- If `generate_path` returns the expected value, it is directly assigned to `response`
- If `generate_path` returns an error, the error is handled by chaining functions together
    - This is the result of returning a monadic type and performing monadic error handling
- If needed, more functions can be added to manipulate the expected type or error type, increasing modularity

PICKNIK

Putting it all together

```
[this](auto const request, auto response) {
    auto const print_error = [this](std::string_view error)
        -> std::expected<GetPath::Response, std::string> {...};

    auto const return_empty_response = []([[maybe_unused]] auto const)
        -> std::expected<GetPath::Response, std::string> {...};

    auto const stringify_error = [](auto const error) {...};

    *response = generate_path::generate_path(request, this->map_,
generate_global_path)
            .map_error(stringify_error)
            .or_else(print_error)
            .or_else(return_empty_response)
            .value();
  }
```

- The generate path callback function has been replaced by a lambda function
- If `generate_path` returns the expected value, it is directly assigned to `response`
- If `generate_path` returns an error, the error is handled by chaining functions together
    - This is the result of returning a monadic type and performing monadic error handling
- If needed, more functions can be added to manipulate the expected type or error type, increasing modularity
- How can this lambda be tested?

PICKNIK

DI and Functional Programming

```
template <typename ServiceType>
using ServiceCallback =  std::function<void(
        std::shared_ptr<typename ServiceType::Request>const ,
        std::shared_ptr<typename ServiceType::Response>)>;

struct Manager {
  struct MiddlewareHandle {
    // Define map service callback type
    using SetMapCallback = ServiceCallback<example_srvs::srv::SetMap>;

    // Define path generation service callback type
    using GeneratePathCallback = ServiceCallback<example_srvs::srv::GetPath>;

    virtual ~MiddlewareHandle() = default;

    virtual void register_set_map_service(SetMapCallback callback) = 0;

    virtual void register_generate_path_service(GeneratePathCallback callback) = 0;

    virtual void log_error(std::string const& msg) = 0;

    virtual void log_info(std::string const& msg) = 0;
  };

  Manager(std::unique_ptr<MiddlewareHandle> mw);

 private:
  std::unique_ptr<MiddlewareHandle> mw_;

  Map<unsigned char> map_;
};
```

- With Dependency Injection (DI)!

PICKNIK

DI and Functional Programming

```
template <typename ServiceType>
using ServiceCallback =  std::function<void(
        std::shared_ptr<typename ServiceType::Request>const ,
        std::shared_ptr<typename ServiceType::Response>)>;

struct Manager {
  struct MiddlewareHandle {
    // Define map service callback type
    using SetMapCallback = ServiceCallback<example_srvs::srv::SetMap>;

    // Define path generation service callback type
    using GeneratePathCallback = ServiceCallback<example_srvs::srv::GetPath>;

    virtual ~MiddlewareHandle() = default;

    virtual void register_set_map_service(SetMapCallback callback) = 0;

    virtual void register_generate_path_service(GeneratePathCallback callback) = 0;

    virtual void log_error(std::string const& msg) = 0;

    virtual void log_info(std::string const& msg) = 0;
  };

  Manager(std::unique_ptr<MiddlewareHandle> mw);

 private:
  std::unique_ptr<MiddlewareHandle> mw_;

  Map<unsigned char> map_;
};
```

- With Dependency Injection (DI)!
  - DI is used to move or “inject” objects into another object

DI and Functional Programming

```
template <typename ServiceType>
using ServiceCallback =  std::function<void(
        std::shared_ptr<typename ServiceType::Request>const ,
        std::shared_ptr<typename ServiceType::Response>)>;

struct Manager {
  struct MiddlewareHandle {
    // Define map service callback type
    using SetMapCallback = ServiceCallback<example_srvs::srv::SetMap>;

    // Define path generation service callback type
    using GeneratePathCallback = ServiceCallback<example_srvs::srv::GetPath>;

    virtual ~MiddlewareHandle() = default;

    virtual void register_set_map_service(SetMapCallback callback) = 0;

    virtual void register_generate_path_service(GeneratePathCallback callback) = 0;

    virtual void log_error(std::string const& msg) = 0;

    virtual void log_info(std::string const& msg) = 0;
  };

  Manager(std::unique_ptr<MiddlewareHandle> mw);

 private:
  std::unique_ptr<MiddlewareHandle> mw_;

  Map<unsigned char> map_;
};
```

- With Dependency Injection (DI)!
  - DI is used to move or “inject” objects into another object
- There still needs to be mutable state, to keep track of the occupancy map between service calls, thus the `map_` member variable

PICKNIK

DI and Functional Programming

```cpp
template <typename ServiceType>
using ServiceCallback =  std::function<void(
        std::shared_ptr<typename ServiceType::Request>const ,
        std::shared_ptr<typename ServiceType::Response>)>;

struct Manager {
  struct MiddlewareHandle {
    // Define map service callback type
    using SetMapCallback = ServiceCallback<example_srvs::srv::SetMap>;

    // Define path generation service callback type
    using GeneratePathCallback = ServiceCallback<example_srvs::srv::GetPath>;

    virtual ~MiddlewareHandle() = default;

    virtual void register_set_map_service(SetMapCallback callback) = 0;

    virtual void register_generate_path_service(GeneratePathCallback callback) = 0;

    virtual void log_error(std::string const& msg) = 0;

    virtual void log_info(std::string const& msg) = 0;
  };

  Manager(std::unique_ptr<MiddlewareHandle> mw);

 private:
  std::unique_ptr<MiddlewareHandle> mw_;

  Map<unsigned char> map_;
};
```

- With Dependency Injection (DI)!
  - DI is used to move or “inject” objects into another object
- There still needs to be mutable state, to keep track of the occupancy map between service calls, thus the `map_` member variable
- For the `Manager` object, a `MiddlewareHandle` struct is defined that is the interface for the injected dependency
- This abstract interface can be used to implement each function using the ROS API

PICKNIK

DI and Functional Programming

```cpp
template <typename ServiceType>
using ServiceCallback =  std::function<void(
        std::shared_ptr<typename ServiceType::Request>const ,
        std::shared_ptr<typename ServiceType::Response>)>;

struct Manager {
  struct MiddlewareHandle {
    // Define map service callback type
    using SetMapCallback = ServiceCallback<example_srvs::srv::SetMap>;

    // Define path generation service callback type
    using GeneratePathCallback = ServiceCallback<example_srvs::srv::GetPath>;

    virtual ~MiddlewareHandle() = default;

    virtual void register_set_map_service(SetMapCallback callback) = 0;

    virtual void register_generate_path_service(GeneratePathCallback callback) = 0;

    virtual void log_error(std::string const& msg) = 0;

    virtual void log_info(std::string const& msg) = 0;
  };

  Manager(std::unique_ptr<MiddlewareHandle> mw);

 private:
  std::unique_ptr<MiddlewareHandle> mw_;

  Map<unsigned char> map_;
};
```

- With Dependency Injection (DI)!
  - DI is used to move or "inject" objects into another object
- There still needs to be mutable state, to keep track of the occupancy map between service calls, thus the `map_` member variable
- For the `Manager` object, a `MiddlewareHandle` struct is defined that is the interface for the injected dependency
- This abstract interface can be used to implement each function using the ROS API
- The lambda function that is used for the generate path service can be captured via mocking and tested

PICKNIK

Testing with DI

```
struct PathManagerFixture : public testing::Test {
  PathManagerFixture() : mw_{std::make_unique<MockMiddlewareHandle>()} {
    // When the map callback is called, set the costmap
    ON_CALL(*mw_, register_set_map_service(testing::_))
        .WillByDefault([&](auto const& map_callback) {
          auto const map_request = make_occupancy_map();
          auto map_response = std::make_shared<SetMap::Response>();
          map_callback(map_request, map_response);
        });
    // Capture the path callback so it can be called later
    ON_CALL(*mw_, register_generate_path_service(testing::_))
        .WillByDefault(testing::SaveArg<0>(&path_callback_));
  }
  std::unique_ptr<MockMiddlewareHandle> mw_;
  pathing::Manager::MiddlewareHandle::GeneratePathCallback path_callback_;
};
TEST_F(PathManagerFixture, NoPath) {
  // GIVEN a path generator with a costmap
  auto const path_generator = pathing::Manager{std::move(mw_)};
  // WHEN the generate path service is called with an unreachable goal
  auto path_request = std::make_shared<GetPath::Request>();
  path_request->start.data = {2, 2};
  path_request->goal.data = {5, 5};
  auto path_response = std::make_shared<GetPath::Response>();
  path_callback_(path_request, path_response);
  // THEN the path generator should succeed
  EXPECT_EQ(path_response->code.code, example_srvs::msg::GetPathCodes::NO_VALID_PATH);
  auto const expected = pathing::Path{};
  // AND the path should be empty
  EXPECT_EQ(pathing::utilities::parseGeneratedPath(path_response->path), expected);
}
```

- Here is the code testing the generate path lambda function

Testing with DI

```cpp
struct PathManagerFixture : public testing::Test {
  PathManagerFixture() : mw_{std::make_unique<MockMiddlewareHandle>()} {
    // When the map callback is called, set the costmap
    ON_CALL(*mw_, register_set_map_service(testing::_))
        .WillByDefault([&](auto const& map_callback) {
          auto const map_request = make_occupancy_map();
          auto map_response = std::make_shared<SetMap::Response>();
          map_callback(map_request, map_response);
        });
    // Capture the path callback so it can be called later
    ON_CALL(*mw_, register_generate_path_service(testing::_))
        .WillByDefault(testing::SaveArg<0>(&path_callback_));
  }
  std::unique_ptr<MockMiddlewareHandle> mw_;
  pathing::Manager::MiddlewareHandle::GeneratePathCallback path_callback_;
};
TEST_F(PathManagerFixture, NoPath) {
  // GIVEN a path generator with a costmap
  auto const path_generator = pathing::Manager{std::move(mw_)};
  // WHEN the generate path service is called with an unreachable goal
  auto path_request = std::make_shared<GetPath::Request>();
  path_request->start.data = {2, 2};
  path_request->goal.data = {5, 5};
  auto path_response = std::make_shared<GetPath::Response>();
  path_callback_(path_request, path_response);
  // THEN the path generator should succeed
  EXPECT_EQ(path_response->code.code, example_srvs::msg::GetPathCodes::NO_VALID_PATH);
  auto const expected = pathing::Path{};
  // AND the path should be empty
  EXPECT_EQ(pathing::utilities::parseGeneratedPath(path_response->path), expected);
}
```

- Here is the code testing the generate path lambda function
- This test fixture calls the callback function for the set occupancy map service when a mock function is executed

Testing with DI

```cpp
struct PathManagerFixture : public testing::Test {
  PathManagerFixture() : mw_{std::make_unique<MockMiddlewareHandle>()} {
    // When the map callback is called, set the costmap
    ON_CALL(*mw_, register_set_map_service(testing::_))
        .WillByDefault([&](auto const& map_callback) {
          auto const map_request = make_occupancy_map();
          auto map_response = std::make_shared<SetMap::Response>();
          map_callback(map_request, map_response);
        });
    // Capture the path callback so it can be called later
    ON_CALL(*mw_, register_generate_path_service(testing::_))
        .WillByDefault(testing::SaveArg<0>(&path_callback_));
  }
  std::unique_ptr<MockMiddlewareHandle> mw_;
  pathing::Manager::MiddlewareHandle::GeneratePathCallback path_callback_;
};
TEST_F(PathManagerFixture, NoPath) {
  // GIVEN a path generator with a costmap
  auto const path_generator = pathing::Manager{std::move(mw_)};
  // WHEN the generate path service is called with an unreachable goal
  auto path_request = std::make_shared<GetPath::Request>();
  path_request->start.data = {2, 2};
  path_request->goal.data = {5, 5};
  auto path_response = std::make_shared<GetPath::Response>();
  path_callback_(path_request, path_response);
  // THEN the path generator should succeed
  EXPECT_EQ(path_response->code.code, example_srvs::msg::GetPathCodes::NO_VALID_PATH);
  auto const expected = pathing::Path{};
  // AND the path should be empty
  EXPECT_EQ(pathing::utilities::parseGeneratedPath(path_response->path), expected);
}
```

- Here is the code testing the generate path lambda function
- This test fixture calls the callback function for the set occupancy map service when a mock function is executed
- This text fixture also captures the callback function for the generate path service so it can be executed later

Testing with DI

```cpp
struct PathManagerFixture : public testing::Test {
  PathManagerFixture() : mw_{std::make_unique<MockMiddlewareHandle>()} {
    // When the map callback is called, set the costmap
    ON_CALL(*mw_, register_set_map_service(testing::_))
        .WillByDefault([&](auto const& map_callback) {
          auto const map_request = make_occupancy_map();
          auto map_response = std::make_shared<SetMap::Response>();
          map_callback(map_request, map_response);
        });
    // Capture the path callback so it can be called later
    ON_CALL(*mw_, register_generate_path_service(testing::_))
        .WillByDefault(testing::SaveArg<0>(&path_callback_));
  }
  std::unique_ptr<MockMiddlewareHandle> mw_;
  pathing::Manager::MiddlewareHandle::GeneratePathCallback path_callback_;
};
TEST_F(PathManagerFixture, NoPath) {
  // GIVEN a path generator with a costmap
  auto const path_generator = pathing::Manager{std::move(mw_)};
  // WHEN the generate path service is called with an unreachable goal
  auto path_request = std::make_shared<GetPath::Request>();
  path_request->start.data = {2, 2};
  path_request->goal.data = {5, 5};
  auto path_response = std::make_shared<GetPath::Response>();
  path_callback_(path_request, path_response);
  // THEN the path generator should succeed
  EXPECT_EQ(path_response->code.code, example_srvs::msg::GetPathCodes::NO_VALID_PATH);
  auto const expected = pathing::Path{};
  // AND the path should be empty
  EXPECT_EQ(pathing::utilities::parseGeneratedPath(path_response->path), expected);
}
```

- Here is the code testing the generate path lambda function
- This test fixture calls the callback function for the set occupancy map service when a mock function is executed
- This text fixture also captures the callback function for the generate path service so it can be executed later
- For this test the occupancy map has already been set via the test fixture

Testing with DI

```
struct PathManagerFixture : public testing::Test {
  PathManagerFixture() : mw_{std::make_unique<MockMiddlewareHandle>()} {
    // When the map callback is called, set the costmap
    ON_CALL(*mw_, register_set_map_service(testing::_))
        .WillByDefault([&](auto const& map_callback) {
          auto const map_request = make_occupancy_map();
          auto map_response = std::make_shared<SetMap::Response>();
          map_callback(map_request, map_response);
        });
    // Capture the path callback so it can be called later
    ON_CALL(*mw_, register_generate_path_service(testing::_))
        .WillByDefault(testing::SaveArg<0>(&path_callback_));
  }
  std::unique_ptr<MockMiddlewareHandle> mw_;
  pathing::Manager::MiddlewareHandle::GeneratePathCallback path_callback_;
};
TEST_F(PathManagerFixture, NoPath) {
  // GIVEN a path generator with a costmap
  auto const path_generator = pathing::Manager{std::move(mw_)};
  // WHEN the generate path service is called with an unreachable goal
  auto path_request = std::make_shared<GetPath::Request>();
  path_request->start.data = {2, 2};
  path_request->goal.data = {5, 5};
  auto path_response = std::make_shared<GetPath::Response>();
  path_callback_(path_request, path_response);
  // THEN the path generator should succeed
  EXPECT_EQ(path_response->code.code, example_srvs::msg::GetPathCodes::NO_VALID_PATH);
  auto const expected = pathing::Path{};
  // AND the path should be empty
  EXPECT_EQ(pathing::utilities::parseGeneratedPath(path_response->path), expected);
}
```

- Here is the code testing the generate path lambda function
- This test fixture calls the callback function for the set occupancy map service when a mock function is executed
- This text fixture also captures the callback function for the generate path service so it can be executed later
- For this test the occupancy map has already been set via the test fixture
- The generate path callback can now be tested by executing the callback function directly

PICKNIK

Testing with DI

```cpp
struct PathManagerFixture : public testing::Test {
  PathManagerFixture() : mw_{std::make_unique<MockMiddlewareHandle>()} {
    // When the map callback is called, set the costmap
    ON_CALL(*mw_, register_set_map_service(testing::_))
        .WillByDefault([&](auto const& map_callback) {
          auto const map_request = make_occupancy_map();
          auto map_response = std::make_shared<SetMap::Response>();
          map_callback(map_request, map_response);
        });
    // Capture the path callback so it can be called later
    ON_CALL(*mw_, register_generate_path_service(testing::_))
        .WillByDefault(testing::SaveArg<0>(&path_callback_));
  }
  std::unique_ptr<MockMiddlewareHandle> mw_;
  pathing::Manager::MiddlewareHandle::GeneratePathCallback path_callback_;
};
TEST_F(PathManagerFixture, NoPath) {
  // GIVEN a path generator with a costmap
  auto const path_generator = pathing::Manager{std::move(mw_)};
  // WHEN the generate path service is called with an unreachable goal
  auto path_request = std::make_shared<GetPath::Request>();
  path_request->start.data = {2, 2};
  path_request->goal.data = {5, 5};
  auto path_response = std::make_shared<GetPath::Response>();
  path_callback_(path_request, path_response);
  // THEN the path generator should succeed
  EXPECT_EQ(path_response->code.code, example_srvs::msg::GetPathCodes::NO_VALID_PATH);
  auto const expected = pathing::Path{};
  // AND the path should be empty
  EXPECT_EQ(pathing::utilities::parseGeneratedPath(path_response->path), expected);
}
```

- Here is the code testing the generate path lambda function
- This test fixture calls the callback function for the set occupancy map service when a mock function is executed
- This text fixture also captures the callback function for the generate path service so it can be executed later
- For this test the occupancy map has already been set via the test fixture
- The generate path callback can now be tested by executing the callback function directly
- **There was no invocation of the middleware using DI and all code is testable without invoking the ROS 2 API!**

PICKNIK

Conclusion

PICKNIK

Conclusion

- The refactored tests are deterministic - they cannot be flaky

Conclusion

- The refactored tests are deterministic - they cannot be flaky
- Break down code into discrete components that can be tested

PICKNIK

Conclusion

- The refactored tests are deterministic - they cannot be flaky
- Break down code into discrete components that can be tested
- Prioritize using pure functions - easier to test and reason about

PICKNIK

Conclusion

- The refactored tests are deterministic - they cannot be flaky
- Break down code into discrete components that can be tested
- Prioritize using pure functions - easier to test and reason about
- Using higher order functions increased the modularity of the code, in this case allowing for different path generating algorithms to be used

PICKNIK

Conclusion

- The refactored tests are deterministic - they cannot be flaky
- Break down code into discrete components that can be tested
- Prioritize using pure functions - easier to test and reason about
- Using higher order functions increased the modularity of the code, in this case allowing for different path generating algorithms to be used
- Monadic error handling led to easier error checking

PICKNIK

Conclusion

- The refactored tests are deterministic - they cannot be flaky
- Break down code into discrete components that can be tested
- Prioritize using pure functions - easier to test and reason about
- Using higher order functions increased the modularity of the code, in this case allowing for different path generating algorithms to be used
- Monadic error handling led to easier error checking
- Refactoring `PathGenerator` using DI in conjunction with the functional programming paradigm led to code that has 100% coverage

Bilal Gill

Leveraging a Functional Approach for More Testable and Maintainable ROS 2 Code

Thank you!

All code and the presentation are available at
https://github.com/PickNikRobotics/ros_testing_templates

PICKNIK