華中科技大學

课 程 实 验 报 告

课程名称：面向对象程序设计

实验名称：智能车控制系统的设计与实现（实验三）

院　　系 ： 计算机科学与技术

专业班级 ： _____

学　　号 ： Uxxxxxxxx

姓　　名 ： _____

指导教师 ： _____

2024 年 12 月 18 日

目 录

一、需求分析

1. 题目要求

ADAS(Advanced Driver Assistance System)是辅助驾驶员安全驾驶车辆的系统，可以增加车辆安全和道路安全，同时可以明显减轻汽车驾驶的操作负担。本实验要求设计一个 C++程序，实现智能车控制系统的执行器 Executor 组件控制指令功能。要求利结合面向对象程序设计的特性：多态性、接口/抽象类、运算符重载和函数式编程特性：Lambda 表达式实现，以提高系统的扩展性

2. 需求分析

Executor 组件可以执行如下的移动指令：

M: 前进，1 次移动 1 格

L: 左转 90 度，位置不变

R: 右转 90 度，位置不变

F: 加速指令，接收到该指令，车进入加速状态，该状态下：

M：前进 2 格（不能跳跃，只能一格一格前进）

L：先前进 1 格，然后左转 90 度

R：先前进 1 格，然后右转 90 度

再接收一次 F 指令，对应的状态取消

B: 倒车指令，接收到该指令，车进入倒车状态，该状态下：

M：在当前朝向上后退一格，朝向不变。注：比如朝向为 N 时收到 M 指令，y 坐标减 1，朝向保持 N

L：右转 90 度，位置不变

R：左转 90 度，位置不变

B 和 F 两个状态可以叠加，叠加状态下：

M：倒退 2 格（不能跳跃，只能一格一格后退）

L：先倒退一格，然后右转 90 度

R：先倒退一格，然后左转 90 度

再接收一次 B 指令，对应的状态取消。

要求可以执行上面这些指令的组合序列，如 MLMRMFMBBFF。

二、系统设计

1. 概要设计

1.1 模块划分与设计

本实验要求实现一个 Executor 组件，根据需求分析，可以明确以下三个接口。

表 2-1 Executor 接口

接口	功能	参数
初始化接口	负责将车初始化在指定位置	x, y, heading
指令执行接口	执行移动、转向指令	指令字符串
获取位置和朝向接口	获取当前的坐标位置和朝向	NA

Executor 类是一个抽象类，定义了执行指令和查询当前位置的接口,同时，定义其实例类 ExecutorImpl，具体实现父类中的抽象方法。其中包含一个私有数据成员 poseHandler 用于管理当前状态，通过封装抽离，避免了循环依赖问题。

因此，我们需要定义一个状态管理类 PoseHandler，负责车辆状态的管理，包括当前位置、方向以及加速和倒车模式的切换。提供接口更新车辆状态，并封装与位置、方向相关的操作逻辑。为此，我们定义了 Point 类和 Direction 类分别用于记录当前坐标，管理转向操作，并维护两个布尔状态变量 fast 和 reverse，分别表示加速模式和倒车模式接。此外，提供车辆的移动（前进/后退）、转向（左转/右转）、模式切换（加速/倒车）等操作方法。Query() 方法用于返回当前车辆状态（位置和方向）。

最后，定义一组指令处理类 Command，包括 MoveCommand，TurnLeftCommand 等一些列类。每个指令类实现具体的指令逻辑，并通过调用 PoseHandler 更新车辆状态。所有指令类通过重载 operator() 接口，与 PoseHandler 交互执行操作。在 ExecutorImpl 的指令执行方法中，通过表驱动，执行操作对应的指令，增强了代码的可维护性和可读性。

Point 类的职责是封装二维平面中的坐标信息，提供加减操作以支持位置更新。通过简单的整数运算更新车辆位置，确保坐标操作的高效性。接口方面：

① 提供获取 X 和 Y 坐标的方法。

② 重载运算符 += 和 -=，实现位置的加减操作。

Direction 类的职责是封装方向的逻辑，包括方向的左转、右转和对应的移动向量。使用数组或枚举表示四个方向，并通过索引计算实现方向切换。其包含三个接口：

① 提供根据当前方向返回前进向量的方法。

② 提供方向切换（左转/右转）的方法。

③ 提供当前方向字符（N, E, S, W）的获取方法。

Pose 数据结构负责封装车辆的当前位置（x, y）和朝向（heading）。提供简单的数据访问功能。作为模块间的数据传递结构，传递车辆的实时状态。

由此，我们可以总结系统包括的类和方法如下：

表 2-2 系统包含的类和接口

类/接口名称	功能描述	类/接口名称
Executor	抽象基类，定义智能车执行器组件的接口，包括执行指令 Execute 和查询当前姿态 Query 两个纯虚方法。	Executor
ExecutorImpl	Executor 的具体实现类，负责解析指令序列并通过指令映射表调用具体动作执行逻辑。	ExecutorImpl
Pose	定义车辆的位置信息（x，y）和朝向（heading），用于记录当前车辆状态。	Pose
Point	用于描述车辆在二维平面中的位置，封装了坐标的操作逻辑（如前进、后退、坐标查询等），支持运算符重载。	Point
Direction	封装车辆的方向逻辑，包括方向的左转、右转操作及获取对应的移动向量。	Direction
PoseHandler	负责车辆位置和方向的状态管理，结合指令实现对车辆状态的变更，例如前进、左转、右转、加速、倒车等逻辑。	PoseHandler
Command	定义了一组动作指令类（如 MoveCommand 、TurnLeftCommand 等），每个指令通过重载 operator() 完成对车辆状态的操作。	Command

在完成模块划分和功能描述后，根据以上描述，可以明确模块间的调用关系。系统通过

模块间的依赖关系形成层次化的调用关系：用户通过 Executor 提供的接口发送指令和查询状态。ExecutorImpl 是 Executor 的实现类，解析指令字符串并调用具体的指令逻辑。ExecutorImpl 使用 PoseHandler 管理车辆状态，所有状态变更操作都通过 PoseHandler 进行。PoseHandler 调用 Point 和 Direction 完成坐标和方向的更新。

在指令解析与执行方面：系统采用表驱动模式，将指令字符（如 M、L、R）映射到 Command 类。ExecutorImpl 遍历指令字符串，根据字符查找对应的 Command，并调用其执行逻辑。Command 类通过 PoseHandler 完成对车辆状态的操作。

Point 和 Direction 独立封装了与坐标和方向相关的底层逻辑，不直接与 ExecutorImpl 或 Command 交互。

Command 模块与具体的执行逻辑解耦，只负责调用 PoseHandler。

由此，做出 UML 类图如下：

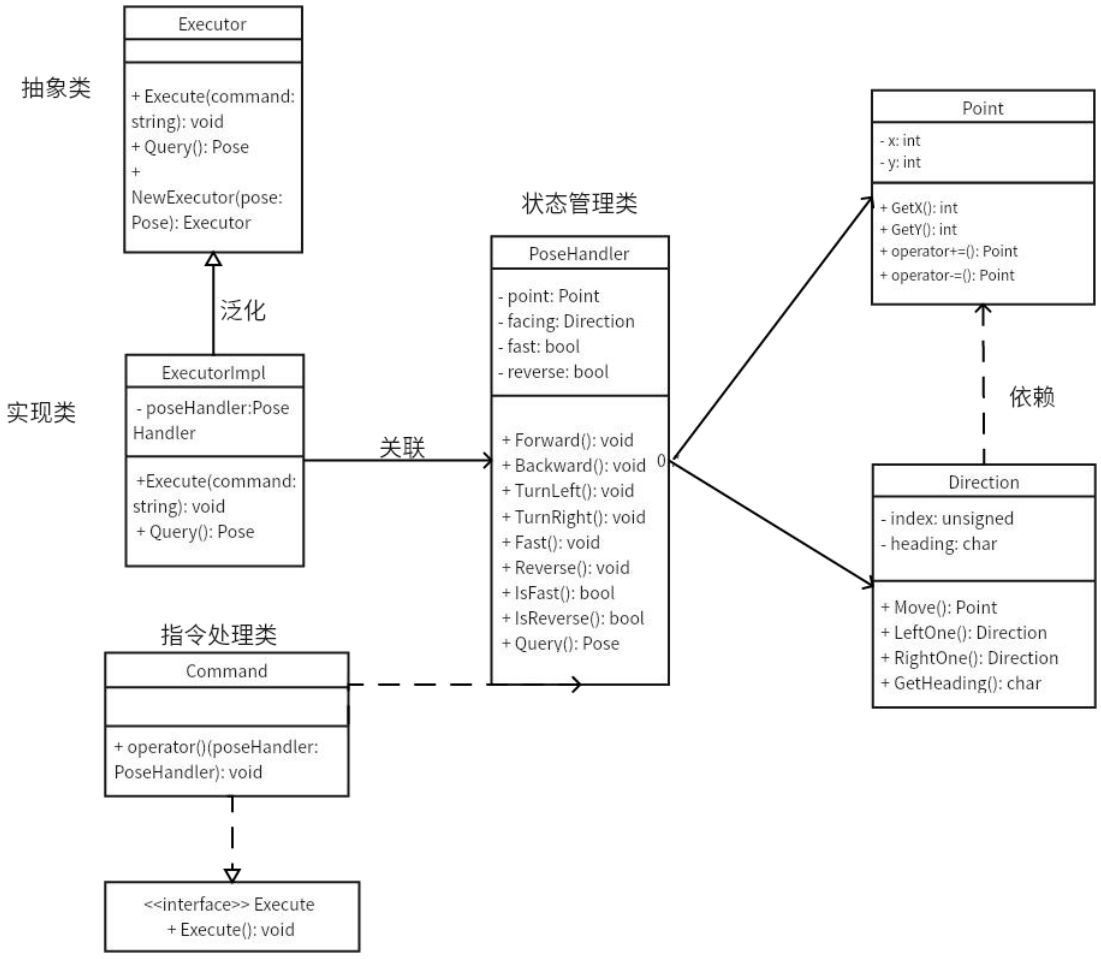


图 2-1 系统设计类图

1.2 数据字典定义

系统中的关键数据定义如下：

(1)输入数据

command：std::string

说明：用户输入的指令序列，例如 MLRFFB，其中每个字符表示一种指令。

(2)状态数据

Pose：结构体

字段：x（整数）：车辆在平面中的 X 坐标。

y（整数）：车辆在平面中的 Y 坐标。

heading（字符）：当前朝向，可能的值为 'N'（北）、'E'（东）、'S'（南）、'W'（西）。

说明：表示车辆的当前位置和朝向。

(3)状态管理数据

fast： 布尔值

说明：表示车辆是否处于加速模式。

reverse：布尔值

说明：表示车辆是否处于倒车模式。

(4)坐标与方向数据

Point：类

字段：x（整数）：X 坐标。

y（整数）：Y 坐标。

说明：封装二维平面中的点操作。

Direction：类

字段：index（整数）：方向索引（0-3）。

heading（字符）：当前方向。

说明：封装方向和方向切换逻辑。

数据流方面,系统按以下分层设计数据流：

① 输入层：用户输入的指令字符串通过 Executor 接口传入系统。

② 指令处理层：ExecutorImpl 解析指令字符串，并通过表驱动模式匹配到相应的 Command 类。

③ 状态管理层：PoseHandler 根据 Command 的执行逻辑更新车辆的状态。

④ 更新涉及坐标（通过 Point）和方向（通过 Direction）。

⑤ 输出层：系统通过 Query 方法返回车辆的当前位置和朝向。

就此，我们完成了系统的概要设计，阐述了 Executor 组件的模块划分与设计。通过明确接口功能、类职责和调用关系，我们构建了一个清晰、层次分明的系统架构。Executor 作为抽象基类，定义了执行指令和查询状态的接口，而 ExecutorImpl 作为具体实现类，负责解析指令并调用相应的操作逻辑。PoseHandler 类作为状态管理的关键，封装了车辆位置和方向的操作，而 Point 和 Direction 类则分别负责坐标和方向的逻辑处理。此外，Command 类族的引入，使得指令执行更加灵活和可维护。整个系统通过数据流的设计，实现了从用户输入到状态更新的顺畅流转。总体而言，本概要设计为后续的详细设计和实现提供了明确的指导，确保了系统的高内聚和低耦合，为智能车执行器组件的稳定运行奠定了基础。

2. 详细设计

2.1 Executor 类

Executor 类是一个抽象类，定义了执行指令和查询当前位置的接口，通过工厂方法，创建 Executor 实例，默认初始位置为 (0, 0, 'N')

表 2-3 Executor 类成员

成员/方法	类型	说明
Execute	纯虚函数	接收指令字符串并逐一执行，用于解析和调用具体实现类的指令执行逻辑。
Query	纯虚函数	返回车辆的当前位置和方向，用于查询车辆状态。
NewExecutor	静态方法	工厂方法，返回 ExecutorImpl 的实例，允许用户自定义车辆初始位置。

2.2 ExecutorImpl 类

ExecutorImpl 是 Executor 的具体实现类，负责解析指令序列，并通过 PoseHandler 管理车辆状态。它将指令字符映射到具体的 Command 类，利用表驱动模式实现高效的指令解析和执行。

表 2-4 ExecutorImpl 类成员

成员/方法	类型	说明
poseHandler	私有数据成员	PoseHandler 实例，负责车辆的状态管理。
Execute	重写方法	遍历输入的指令字符串，解析每个字符，通过映射表调用对应的 Command 类执行操作。
Query	重写方法	调用 PoseHandler 的 Query 方法，返回车辆的当前位置和方向。

以下是对关键方法的具体介绍：

- `void ExecutorImpl::Execute(const std::string &commands) noexcept`

(1)功能：解析输入指令字符串，通过映射表调用对应的指令类执行操作。

(2)实现流程：

① 初始化指令映射表，将每个指令字符（如 M、L、R）映射到具体的 Command 类。

② 遍历指令字符串，对每个字符：

③ 查找映射表中对应的 Command 对象。

④ 调用该 Command 的 operator() 方法，传入 PoseHandler，完成操作。如果输入字符不在映射表中，跳过处理。

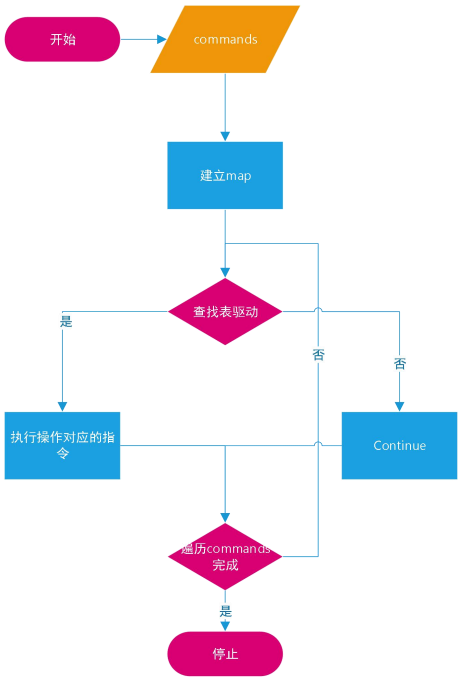


图 2-2 Execute 方法流程图

2.3 Point 类

Point 类用于表示二维空间中的一个点，它包含了两个整数坐标 x 和 y，分别代表点在水平方向和垂直方向的位置。负责描述车辆的当前位置，并提供加减操作以支持位置更新。

表 2-5 PointI 类成员

成员/方法	类型	说明
x	数据成员	整数，表示 X 坐标。
y	数据成员	整数，表示 Y 坐标。
GetX	成员方法	返回 X 坐标的值。
GetY	成员方法	返回 Y 坐标的值。
operator+=	运算符重载	对当前点执行坐标加法，更新 X 和 Y 值。
operator-=	运算符重载	对当前点执行坐标减法，更新 X 和 Y 值。

值得注意的是，Point 类重载了+=和-=运算符，便于后续 PoseHandler 类中方法对状态进行改变。

2.4 Direction 类

Direction 封装方向的逻辑，它封装了一个方向索引和一个方向字符，用于表示东（East）、南（South）、西（West）、北（North）四个基本方向。

表 2-6 Direction 类成员

成员/方法	类型	说明
index	数据成员	整数，方向索引（0 表示东，1 表示南，2 表示西，3 表示北）。
heading	数据成员	字符，表示当前方向（'N', 'E', 'S', 'W'）。
Move	成员方法	返回当前方向对应的前进向量（Point）。
LeftOne	成员方法	返回当前方向左转 90 度后的新方向。
RightOne	成员方法	返回当前方向右转 90 度后的新方向。

在设计方面，Diretion 类通过数组和索引值，将复杂的方向切换问题转化为简单的数学操作（索引加减和取模运算），从而避免了大量的条件分支判断，使得代码更加简洁、高效、易扩展。一下对相关函数成员进行具体介绍。

- `Direction::GetDirection(const char heading) noexcept`

(1)作用：根据方向字符（'N', 'E', 'S', 'W'）获取对应的方向实例。

(2)实现：遍历静态数组 directions，查找与字符匹配的方向对象。如果未找到，返回默认方向（北，'N'）。

(3)代码：

```
const Direction &Direction::GetDirection(const char heading) noexcept
    {
        // 遍历方向数组，查找与指定方向字符匹配的方向
        for (const auto &dir : directions)
        {
            if (dir.heading == heading)
            {
```

```
                return dir; // 找到匹配方向，返回引用
            }
        }
        // 如果未找到匹配项，默认返回东（E）方向
        return directions[3];
    }
```

- `const Direction& LeftOne(void) const noexcept`

  (1)作用：返回当前方向左转 90 度后的方向对象。

  (2)实现：

  ① 计算左转后的索引：(index + 3) % 4。

  ② 在数组中，左转等价于逆时针移动一格，相当于将索引减 1。

  ③ 为避免出现负数，实际操作是加 3 再取模。

  ④ 使用计算后的索引返回 directions 数组中的方向对象。

  (3) 代码：

```
const Direction &Direction::LeftOne() const noexcept
    {
        // 左转将索引逆时针移动1，相当于 (index + 3) % 4
        return directions[(index + 3) % 4];
    }
```

2.5 PoseHandler 类

PoseHandler 类用于管理和更新车辆的状态，包括位置（Point）、方向（Direction）、是否处于快速模式以及是否处于倒车模式，并提供一系列接口用于更新或查询车辆状态。PoseHandler 将车辆的状态管理封装在一个单独的类中，遵循单一职责原则，使得状态管理逻辑与其它系统组件分离。

表 2-7 PoseHandler 类成员

成员/方法	类型	说明
point	数 据 成员	Point 实例，表示车辆在二维平面中的位置（x, y 坐标）。
facing	数 据 成员	Direction 指针，指向当前车辆朝向的方向对象。
fast	数 据 成员	布尔值，标识车辆是否处于加速模式。
reverse	数 据 成员	布尔值，标识车辆是否处于倒车模式。
Forward	成 员 方法	车辆向当前方向前进一格（或两格，取决于模式）。
Backward	成 员 方法	车辆向当前方向后退一格（或两格，取决于模式）。
TurnLeft	成 员 方法	车辆左转 90 度。
TurnRight	成 员 方法	车辆右转 90 度。
Fast	成 员 方法	切换加速模式，如果当前处于加速模式，则取消加速模式；否则开启加速模式。
Reverse	成 员 方法	切换倒车模式，如果当前处于倒车模式，则取消倒车模式；否则开启倒车模式。
IsFast	成 员 方法	返回车辆是否处于加速模式。
IsReverse	成 员 方法	返回车辆是否处于倒车模式。
Query	成 员 方法	返回 Pose 对象，包含当前位置和方向。

在实现了 Point 类和 Direction 类的基础上基础上，这些方法的实现都比较简单，以 Forward 方法为例，只需要调用 Direction 中定义的 Move 方法即可。

```
void PoseHandler::Forward() noexcept
    {
        // 将当前坐标加上方向向量
        point += facing->Move();
    }
```

2.6 Command 模块

Command.hpp 模块定义了一系列命令类，每个类都代表一个特定的操作，这些操作可以作用于 PoseHandler 对象，通过重载 operator()，每个指令类调用 PoseHandler 的方法更新车辆状态。

通过将每个操作封装为一个对象，命令模式解耦了发出命令的责任和执行命令的责任。这使得添加新命令变得容易，并且可以灵活地组合命令。每个命令类都重载了 operator()，使其成为一个函数对象，可以直接调用并作用于 PoseHandler

表 2-8 Command 指令类

指令类	功能说明
MoveCommand	控制车辆前进，调用 PoseHandler 的 Forward 方法，根据模式执行正常前进或加速前进。
TurnLeftCommand	控制车辆左转，调用 PoseHandler 的 TurnLeft 方法。
TurnRightCommand	控制车辆右转，调用 PoseHandler 的 TurnRight 方法。
FastCommand	切换加速模式，调用 PoseHandler 的 Fast 方法。
ReverseCommand	切换倒车模式，调用 PoseHandler 的 Reverse 方法。

其中每个指令类根据需求描述调用 PoseHandler 类对应的方法实现目标功能。

三、软件开发

1. 开发环境

本次实验中使用的环境配置如下：

(1) 操作系统版本：Microsoft Windows 11 家庭中文版 X64

(2) 集成开发环境：Visual Studio Code

(3) 编译器：GCC 8.1.0

(4) 构建工具：CMake 3.0

(5) 调试工具：GDB（来自 MinGW-w64 工具链，已集成在 VSCode 的调试配置中）

项目配置方面，本实验使用了华为提供的模板工程，包含了以下配置和脚本文件：

(1) launch.json

配置 VSCode 的调试功能，使用 GDB 作为调试器。

设置调试目标为 bin/ 目录下生成的可执行文件。

配置命令包括启用 GDB 的格式化输出和 Intel 指令格式。

(2) settings.json

配置文件关联，用于识别项目中的 C++ 文件。

启用格式化选项（保存时自动格式化、粘贴时格式化）。

配置终端为 PowerShell，确保编码兼容性（UTF-8）。

(3) tasks.json

定义两项主要任务：cmake configure：运行 CMake 以生成构建系统文件。

make：使用 MinGW 工具链编译项目，生成目标文件。

(4) build.bat

用于一键执行项目的构建操作。创建 build/ 目录（如果不存在）。使用 CMake 生成 MinGW 的 Makefile。使用 mingw32-make 编译生成二进制文件。

2. 编译，构建流程

整个项目的构建过程分为 Cmake 配置、源文件编译和链接三个阶段。

在配置阶段，通过运行 cmake 命令读取根目录和子目录下的 CMakeLists.txt 文件，CMake 检测系统的编译器（GCC 8.1.0），并根据源文件和头文件目录生成适用于 MinGW 的 Makefile 构建文件，输出到 build/ 目录中。

在编译阶段，CMake 调用 mingw32-make 编译工具，根据 Makefile 中的规则，将每个源文件（如 Direction.cpp、PoseHandler.cpp）编译成目标文件（.obj）。src/ 目录下的目标文件会进一步被链接为静态库 libADAS.a，存储于 bin/ 目录中。同时，Google Test 的源文件也会被编译并生成测试相关的静态库（如 libgtest.a）,用于后续测试。

在链接阶段，测试代码和生成的静态库通过链接器合并为可执行文件 ADAS-main.exe，存储在 bin/ 目录中。最终的可执行文件可运行整个测试套件，验证项目功能。整个过程通过 build.bat 或 CMake 的自动化机制实现模块化、高效化管理。

四、软件测试

要求基于 gtest 进行功能测试。要求以表格的形式给出所有测试用例，包括：测试用例名称、测试用例功能。并给出测试用例运行结果的截图。

本项目使用 Google Test（GTest）框架进行功能测试，使用 ASSERT_EQ 断言测试 Pose 对象是否与预期一致。

1. 基础功能测试组

ExecutorTest 测试组用于普通状态下测试，前进，左转，右转三个基础功能，利用正交分解法构建测试防护网，包含十四个测试用例。

测试防护网：正交分解法

方向 / 指令	E	W	N	S
M	当前朝向E 执行M X+1	当前朝向W 执行M X−1	当前朝向N 执行M Y+1	当前朝向S 执行M Y−1
L	当前朝向E 执行L 朝向N	当前朝向W 执行L 朝向S	当前朝向N 执行L 朝向W	当前朝向S 执行L 朝向E
R	当前朝向E 执行R 朝向S	当前朝向W 执行R 朝向N	当前朝向N 执行R 朝向E	当前朝向s 执行R 朝向W

图 4-1 正交分解法构建测试组 1 用例

表 4-1 ExecutorTest 测试组测试用例

测试用例名称	功能描述
should_return_init_pose_when_without_command	测试当无指令时返回初始化的 Pose
should_return_default_pose_when_without_init_and_command	测试无初始位置和指令时返回默认的 Pose
should_return_x_plus_1_given_command_is_M_and_facing_is_E	测试向东时执行 M 指令后，x 坐标增加 1
should_return_x_minus_1_given_command_is_M_and_facing_is_W	测试向西时执行 M 指令后，x 坐标减少 1
should_return_y_plus_1_given_command_is_M_and_facing_is_N	测试向北时执行 M 指令后，y 坐标增加 1
should_return_y_minus_1_given_command_is_M_and_facing_is_S	测试向南时执行 M 指令后，y 坐标减少 1
should_return_facing_N_given_command_is_L_and_facing_is_E	测试向东时执行 L 指令后，朝向变为北
should_return_facing_S_given_command_is_L_and_facing_is_W	测试向西时执行 L 指令后，朝向变为南
should_return_facing_W_given_command_is_L_and_facing_is_N	测试向北时执行 L 指令

	后，朝向变为西
should_return_facing_E_given_command_is_L_and_facing_is_S	测试向南时执行 L 指令后，朝向变为东
should_return_facing_S_given_command_is_R_and_facing_is_E	测试向东时执行 R 指令后，朝向变为南
should_return_facing_W_given_command_is_R_and_facing_is_S	测试向南时执行 R 指令后，朝向变为西
should_return_facing_N_given_command_is_R_and_facing_is_W	测试向西时执行 R 指令后，朝向变为北
should_return_facing_E_given_command_is_R_and_facing_is_N	测试向北时执行 R 指令后，朝向变为东

所有测试用例均成功通过，以下为测试运行结果的截图

```
[----------] 14 tests from ExecutorTest
[ RUN      ] ExecutorTest.should_return_init_pose_when_without_command
[       OK ] ExecutorTest.should_return_init_pose_when_without_command (0 ms)
[ RUN      ] ExecutorTest.should_return_default_pose_when_without_init_and_command
[       OK ] ExecutorTest.should_return_default_pose_when_without_init_and_command (0 ms)
[ RUN      ] ExecutorTest.should_return_x_plus_1_given_command_is_M_and_facing_is_E
[       OK ] ExecutorTest.should_return_x_plus_1_given_command_is_M_and_facing_is_E (0 ms)
[ RUN      ] ExecutorTest.should_return_x_minus_1_given_command_is_M_and_facing_is_W
[       OK ] ExecutorTest.should_return_x_minus_1_given_command_is_M_and_facing_is_W (0 ms)
[ RUN      ] ExecutorTest.should_return_y_plus_1_given_command_is_M_and_facing_is_N
[       OK ] ExecutorTest.should_return_y_plus_1_given_command_is_M_and_facing_is_N (0 ms)
[ RUN      ] ExecutorTest.should_return_y_minus_1_given_command_is_M_and_facing_is_S
[       OK ] ExecutorTest.should_return_y_minus_1_given_command_is_M_and_facing_is_S (0 ms)
[ RUN      ] ExecutorTest.should_return_facing_N_given_command_is_L_and_facing_is_E
[       OK ] ExecutorTest.should_return_facing_N_given_command_is_L_and_facing_is_E (0 ms)
[ RUN      ] ExecutorTest.should_return_facing_S_given_command_is_L_and_facing_is_W
[       OK ] ExecutorTest.should_return_facing_S_given_command_is_L_and_facing_is_W (0 ms)
[ RUN      ] ExecutorTest.should_return_facing_W_given_command_is_L_and_facing_is_N
[       OK ] ExecutorTest.should_return_facing_W_given_command_is_L_and_facing_is_N (0 ms)
[ RUN      ] ExecutorTest.should_return_facing_E_given_command_is_L_and_facing_is_S
[       OK ] ExecutorTest.should_return_facing_E_given_command_is_L_and_facing_is_S (0 ms)
[ RUN      ] ExecutorTest.should_return_facing_S_given_command_is_R_and_facing_is_E
[       OK ] ExecutorTest.should_return_facing_S_given_command_is_R_and_facing_is_E (0 ms)
[ RUN      ] ExecutorTest.should_return_facing_W_given_command_is_R_and_facing_is_S
[       OK ] ExecutorTest.should_return_facing_W_given_command_is_R_and_facing_is_S (0 ms)
[ RUN      ] ExecutorTest.should_return_facing_N_given_command_is_R_and_facing_is_W
[       OK ] ExecutorTest.should_return_facing_N_given_command_is_R_and_facing_is_W (0 ms)
[ RUN      ] ExecutorTest.should_return_facing_E_given_command_is_R_and_facing_is_N
[       OK ] ExecutorTest.should_return_facing_E_given_command_is_R_and_facing_is_N (0 ms)
[----------] 14 tests from ExecutorTest (54 ms total)
```

图 4-2 ExecutorTest 测试组测试结果图

2. 加速功能测试组

本测试组(ExecutorFastTest)主要测试加速指令功能，加速状态下的前进，左转，右转指令能否按预期执行，测试用例的构建上，同样采用正交分解法，设计了四个测试用例。

状态 指令	F	FF
M	当前朝向E 执行FM X+1, X+1	当前朝向 N 执行FFM Y+1
L	当前朝向E 执行FL X+1,朝向N	NA
R	当前朝向E 执行FR X+1,朝向S	NA

图 4-3 正交分解法构建测试组 2 用例

表 4-2 ExecutorFastTest 测试组测试用例

测试用例名称	功能描述
should_return_x_plus_2_given_status_is_fast_command_is_M_and_facing_is_E	测试加速模式下，向东时执行 FM 后 x 坐标加 2
should_return_N_and_x_plus_1_given_status_is_fast_command_is_L_and_facing_is_E	测试加速模式下，向东执行 FL 后朝向变为北，x 坐标加 1
should_return_S_and_x_plus_1_given_status_is_fast_given_command_is_R_and_facing_is_E	测试加速模式下，向东执行 FR 后朝向变为南，x 坐标加 1
should_return_y_plus_1_given_command_is_FFM_and_facing_is_N	测试加速模式下向北执行 FFM 后 y 坐标加 1

所有测试用例均成功通过，以下为测试运行结果的截图

```
[----------] 4 tests from ExecutorFastTest
[ RUN      ] ExecutorFastTest.should_return_x_plus_2_given_status_is_fast_command_is_M_and_facing_is_E
[       OK ] ExecutorFastTest.should_return_x_plus_2_given_status_is_fast_command_is_M_and_facing_is_E (0 ms)
[ RUN      ] ExecutorFastTest.should_return_N_and_x_plus_1_given_status_is_fast_command_is_L_and_facing_is_E
[       OK ] ExecutorFastTest.should_return_N_and_x_plus_1_given_status_is_fast_command_is_L_and_facing_is_E (0 ms)
[ RUN      ] ExecutorFastTest.should_return_S_and_x_plus_1_given_status_is_fast_given_command_is_R_and_facing_is_E
[       OK ] ExecutorFastTest.should_return_S_and_x_plus_1_given_status_is_fast_given_command_is_R_and_facing_is_E (0 ms)
[ RUN      ] ExecutorFastTest.should_return_y_plus_1_given_command_is_FFM_and_facing_is_N
[       OK ] ExecutorFastTest.should_return_y_plus_1_given_command_is_FFM_and_facing_is_N (0 ms)
[----------] 4 tests from ExecutorFastTest (18 ms total)
```

图 4-4 ExecutorFastTest 测试组测试结果图

3. 倒车功能测试组

同加速功能一样，运用正交分解法构建类似的四个测试用例

表 4-3 ExecutorReverseTest 测试组测试用例

测试用例名称	功能描述
should_return_x_minus_1_given_status_is_back_command_is_M_and_facing_is_E	测试倒车模式下，向东执行 BM 后 x 坐标减 1
should_return_S_given_status_is_reverse_command_is_L_and_facing_is_E	测试倒车模式下，向东执行 BL 后朝向变为南
should_return_N_given_status_is_reverse_command_is_R_and_facing_is_E	测试倒车模式下，向东执行 BR 后朝向变为北
should_return_y_plus_1_given_command_is_BBM_and_facing_is_N	测试倒车模式下向北执行 BBM 后 y 坐标加 1

所有测试用例均成功通过，以下为测试运行结果的截图

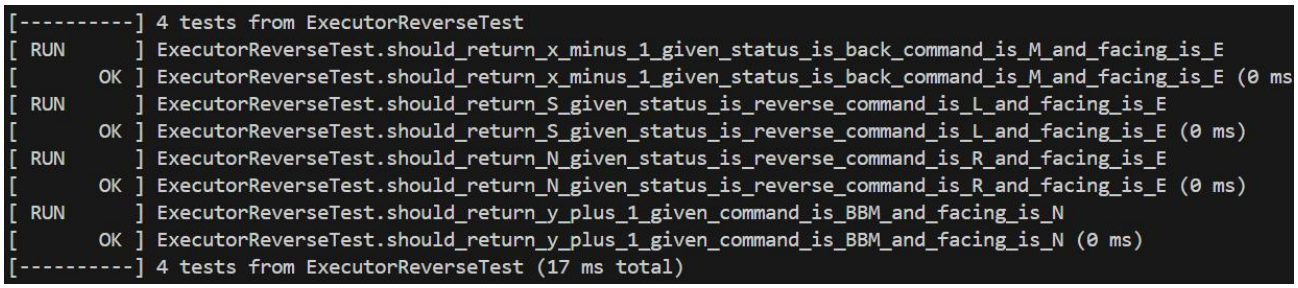


图 4-5 ExecutorReverseTest 测试组测试结果图

4. 加速倒车叠加状态测试

根据需求，B 和 F 两个状态可以叠加，叠加状态下：

M：倒退 2 格（不能跳跃，只能一格一格后退）

L：先倒退一格，然后右转 90 度

R：先倒退一格，然后左转 90 度

同样根据正交分解法构建测试用例,由于其它用例已在前面写过，本测试组实际只有三个测试用例。

状态 指令	B	F	BF	BB	FF
M	当前朝向E 执行BM X−1	当前朝向E 执行FM X+2	当前朝向E 执行BFM X−2	当前朝向N 执行BBM Y+1	当前朝向N 执行FFM Y+1
L	当前朝向E 执行BL 朝向S	当前朝向E 执行FL X+1，朝向N	当前朝向E 执行BFL X−1，朝向S	NA	NA
R	当前朝向E 执行BR 朝向N	当前朝向E 执行FR X+1，朝向S	当前朝向E 执行BFR X−1，朝向N	NA	NA

图 4-6 正交分解法构建测试组 4 用例

表 4-4 ExecutorReverseFastTest 测试组测试用例

测试用例名称	功能描述
ExecutorReverseFastTest.should_return_x_minus_2_given_status_is_fast_and_reverse_command_is_M_and_facing_is_E	测试在快速和倒车模式下，面向东方执行 FM 后 x 坐标减 2
ExecutorReverseFastTest.should_return_S_and_x_minus_1_given_status_is_fast_and_reverse_command_is_L_and_facing_is_E	测试在快速和倒车模式下，面向东方执行 FL 后 x 坐标减 1 且朝向变为南
ExecutorReverseFastTest.should_return_N_and_x_minus_1_given_status_is_fast_and_reverse_command_is_R_and_facing_is_E	测试在快速和倒车模式下，面向东方执行 FR 后 x 坐标减 1 且朝向变为北

所有测试用例均成功通过，以下为测试运行结果的截图

```
[----------] 3 tests from ExecutorReverseFastTest
[ RUN      ] ExecutorReverseFastTest.should_return_x_minus_2_given_status_is_fast_and_reverse_command_is_M_and_facing_is_E
[       OK ] ExecutorReverseFastTest.should_return_x_minus_2_given_status_is_fast_and_reverse_command_is_M_and_facing_is_E (0 ms)
[ RUN      ] ExecutorReverseFastTest.should_return_S_and_x_minus_1_given_status_is_fast_and_reverse_command_is_L_and_facing_is_E
[       OK ] ExecutorReverseFastTest.should_return_S_and_x_minus_1_given_status_is_fast_and_reverse_command_is_L_and_facing_is_E (0 ms)
[ RUN      ] ExecutorReverseFastTest.should_return_N_and_x_minus_1_given_status_is_fast_and_reverse_command_is_R_and_facing_is_E
[       OK ] ExecutorReverseFastTest.should_return_N_and_x_minus_1_given_status_is_fast_and_reverse_command_is_R_and_facing_is_E (0 ms)
[----------] 3 tests from ExecutorReverseFastTest (16 ms total)
```

图 4-7 ExecutorReverseFastTest 测试组测试结果图

总结：本次实验针对车辆控制系统在不同工作模式下的核心功能进行了全面的测试验证。测试覆盖了正常模式、加速模式以及倒车模式，共计 25 个精心设计的测试用例，评估了各模

块在标准输入条件下的表现。所有测试用例均成功通过，验证了系统在正常工作状态下的稳定性和可靠性。通过采用 GTest 框架进行测试用例的管理和结果输出，本实验确保了测试过程的透明度和结果的准确性，从而为系统功能的可靠性提供了科学的保障。测试结果表明，车辆控制系统在不同条件下能够准确地更新位置和方向信息，所有功能模块均表现出高度的稳定性和可靠性，确保了车辆控制系统的整体性能。

五、特点与不足

1. 技术特点

1.1 表驱动

在 C++中，可以使用 std::map 来存储键值对，其中键是条件，值是对应的处理函数或者对象。

表驱动模式通过使用 std::unordered_map 来实现命令和对应处理方法的映射。表驱动模式的核心思想是使用查找表（通常是字典、映射等数据结构）来代替条件判断（如 if/switch 语句），通过查表来执行相应的操作。这种方式可以极大提高代码的可扩展性、可维护性和简洁性。

在 Executor 方法中，若像下面代码一样使用传统的 if-else 结构，随着命令类型的增加，if-else 语句会变得越来越冗长，维护起来不方便。每当需要增加新的命令时，需要修改指令处理的主干的逻辑，容易导致错误。

```
if (cmd == 'M')
            {
                // 智能指针指向MoveCommand实例，不用担心delete了
                cmder = std::make_unique<MoveCommand>();
            }
            else if (cmd == 'L')
            {
                // 智能指针指向TurnLeftCommand实例
                cmder = std::make_unique<TurnLeftCommand>();
            }
            else if (cmd == 'R')
            {
                // 智能指针指向TurnRightCommand实例
                cmder = std::make_unique<TurnRightCommand>();
            }
```

在本系统实现的 Executor 方法中，采用了表驱动模式，将不同的命令字符（如 M、L、R）映射到相应的函数对象（如 MoveCommand()、TurnLeftCommand()等）。在 Execute 函数中，根据传入的 commands 字符串中的每个字符，查找表中对应的命令并执行。

```
// 表驱动
        std::unordered_map<char, std::function<void(PoseHandler &)>> cmde
rMap{
            {'M', MoveCommand()},       // 前进
            {'L', TurnLeftCommand()},  // 左转
```

```
            {'R', TurnRightCommand()}, // 右转
            {'F', FastCommand()},      // 快速
            {'B', ReverseCommand()},
        };
        for (const auto cmd : commands) // const auto cmd : commands 是
C++11 的特性，用于遍历字符串
        {
            // 根据操作查找表驱动
            const auto it = cmderMap.find(cmd);
            // 如果找到表驱动，执行操作对应的指令
            if (it != cmderMap.end())
            {
                it->second(poseHandler);
            }
        }
```

表驱动的优势：

(1)简化代码逻辑：避免了大量的条件语句，使代码更易读。而传统的条件判断随着命令数量的增加，可能变得冗长且难以维护。

(2)易于扩展：添加新的条件或和处理函数时，只需要更新表格，而不需要修改现有的逻辑例如，如果需要新增一个命令 T(掉头)，只需要在 cmderMap 中加入相应的映射，如：

```
{'T', TurnAroundCommand()}  //掉头指令
```

(3)提高可维护性：逻辑和数据分离，便于维护和调试。如果有需要修改命令的执行方式，只需修改相应命令的处理类（如 MoveCommand 或 TurnLeftCommand），不需要去修改 Execute 函数的核心逻辑。这种松耦合的设计使得修改和扩展变得更加简单。例如，想要修改 MoveCommand 的行为，只需要修改 MoveCommand 类，而 Execute 函数不需要做任何修改。

总之，表驱动模式简化命令解析的过程，并将命令字符与相应操作进行映射。相比传统的 if-else 或 switch-case 方式，表驱动模式具有更高的可扩展性、可维护性和清晰的结构，能够方便地增加新命令并减少代码重复。此外，表驱动还提高了代码的可读性，因为命令与操作的对应关系一目了然

1.2 状态抽象和计算属性

状态抽象是通过将不同的状态和与之相关的行为封装到对象中，形成一个清晰的状态模型，对外提供统一的接口。每个状态对象负责管理与该状态相关的行为和属性，状态间的流转通常通过方法调用来实现。这种方式将状态及其行为分离，使得系统更易于管理和扩展。

在本项目中，Direction 类充当了方向状态的抽象。每个 Direction 对象包含一个方向索引和一个方向字符（如 E、S、W、N），并提供了一系列方法（如 Move()、LeftOne()、RightOne()）来描述与当前方向相关的行为。每个方向都通过一个 Direction 对象表示，封装了与该方向相关的操作（如前进、左转、右转等）。方向之间的流转通过 LeftOne() 和 RightOne() 方法实现，用户无需关心底层的索引变换，直接通过方法调用来改变方向。

```
class Direction final
    {
    public:
        // 根据方向字符获取方向
        static const Direction &GetDirection(const char heading) noexcept
;
    public:
        Direction(const unsigned index, const char heading) noexcept;
    public:
        const Point &Move(void) const noexcept;          // 前进
        const Direction &LeftOne(void) const noexcept;   // 左转
        const Direction &RightOne(void) const noexcept; // 右转
        const char GetHeading(void) const noexcept; // 获取方向字符
    private:
        unsigned index; // 方向索引 0 1 2 3
        char heading;   // 方向字符 E S W N
    };
```

通过状态抽象，增强了程序的可扩展性，当需要添加新的方向或新的行为时，可以通过扩展 Direction 类来实现，而不需要改变现有的控制流程或修改大量条件判断，使得系统更加灵活，便于扩展，也更容易管理,所有与方向相关的行为都集中在 Direction 类中，管理起来更加方便。如果需要更改流转规则（例如，改变转向逻辑），只需修改 Direction 类中的方法，而不需要在多个地方修改。

此外，本系统巧妙地运用了用计算机的原始计算属性解决复杂状态流转问题。

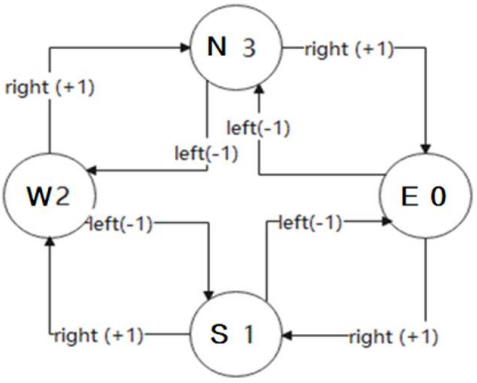
计算属性是指那些根据对象当前状态动态计算得出的属性，而不是直接存储的属性。这种属性的值通常依赖于对象的其他状态信息，并在每次访问时根据这些信息实时计算。计算属性避免了冗余存储，使得对象更加轻量化，并且确保属性值始终与对象的状态保持一致。

本项目通过挖掘次态和现态间的数学关系(如图 5-1)，简化了状态转换逻辑。

我们定义的 directions 方向数组建立了 0-E、1-S、2-W、3-N 的映射关系，这样就可以通过当前的方向索引计算出左转或右转后的方向索引。Points 坐标数组给出了每个方向对应的移动坐标，(1,0)|(0, -1)|(-1,0)|(0,1)正好对应了 E|S|W|N 对应的移动坐标。例如 directions[0]对应的方向数据为{0,'E'}，代表索引 0 对应了方向"E"；points[0]对应的移动坐标为{1,0}，代表向东移动一步坐标需要加上(1,0)；

根据右转状态编码公式：(0 + 1) % 4 = 1，direciont s[1]对应的方向数据为{1,'S'}，代表朝向为"E"时右转后的朝向为"S"；

同理左转状态编码公式：(0 + 3) % 4 = 3， direciont s[3]对应的方向数据为{3,'N'}，代表朝向为"E"时左转后的朝向为"N"



右转后状态编码 =（当前状态编码 + 1）% 4

左转后状态编码 =（当前状态编码 – 1 + 4）% 4 =（当前状态 + 3）% 4

。

图 5-1 转向操作优化示意图

例如 LeftOne() 和 RightOne() 方法根据当前方向索引的变化动态计算左转或右转后的方向：

```
const Direction &Direction::LeftOne() const noexcept
{
    return directions[(index + 3) % 4];
}
const Direction &Direction::RightOne() const noexcept
{
    return directions[(index + 1) % 4];
}
```

这些方法基于当前的 index 状态，动态计算对应的前进坐标或方向，无需额外存储结果，节省了内存并确保状态和行为的统一。也无需通过传统的 if-else 语句进行判断，避免了冗长

的代码，提高了灵活性和可维护性。传统方法往往需要额外的逻辑来更新和管理属性值，而计算属性让这部分工作透明化，减少了潜在的错误。

1.3 函数式编程

函数式编程（Functional Programming，简称 FP）是一种编程范式，它强调对表达式的运算而非执行命令（声明式）。涉及纯函数、高阶函数、延迟计算等概念。

在本实验实现的 Execute 方法中，cmderMap 的类型为

```
std::unordered_map<char, std::function<void(PoseHandler &)>>
```

它将字符命令与对应的操作函数（如 MoveCommand、TurnLeftCommand 等）关联起来。这里使用了 std::function，这本质上是一个高阶函数，即通过 std::function 将函数作为值存储在容器中。

代码中所有的命令（如 MoveCommand, TurnLeftCommand 等）都通过函数对象来执行，这些函数对象都是“纯函数”，即它们的行为只依赖于传入的 PoseHandler 参数，并不改变外部的状态或有副作用。比如，MoveCommand 的调用只是通过 poseHandler 执行前进操作，并不会改变外部状态。结合之前表驱动中展示的代码可见，函数式编程具有以下优势：

(1) 代码简洁性：函数式编程通常使用更少的代码来实现相同的功能，使代码更加简洁和易读。传统的方法可能会依赖大量的 if-else 或 switch-case 来判断指令类型并执行相应的操作，而函数式编程通过将操作封装为高阶函数，使用表驱动模式来查找和执行操作，减少了冗余的条件判断，代码更加简洁和清晰。

(2) 可维护性：由于函数是纯函数且数据是不可变的，代码更容易理解和测试。这种特性使得代码的维护和调试更加方便。每个命令类（如 MoveCommand、TurnLeftCommand）作为独立的函数对象，具有明确的输入和输出，并且没有副作用。这使得它们非常容易进行单元测试。我们可以独立地测试每个命令，确保它们对 PoseHandler 对象的操作是正确的。

(3) 可预测性：纯函数的行为是确定的，对于相同的输入总是产生相同的输出，因此我们可以更准确地预测代码的行为。函数式编程强调纯函数和不可变性，这在本程序中体现得尤为明显。所有命令操作都是无副作用的，它们只依赖输入（PoseHandler）并返回一个新的状态，而不会修改外部状态或变量，这使得代码更加健壮，并易于理解。

(4) 并发编程：由于函数式编程强调不可变性和无副作用，减少了并发编程中的数据竞争问题，使得多线程编程更加安全和高效。

(5) 代码复用性：函数式编程中的高阶函数和柯里化等概念使得代码更具复用性和组合性，可

以更方便地构建复杂的功能。

这些优势使得函数式编程在提高代码质量、减少错误和提高开发效率方面具有显著的收益。

2. 不足和改进的建议

2.1 耦合度较高

在目前的表驱动设计中，每次调用时都会生成一个 cmderMap，这不仅增加了系统的开销，还影响了执行效率。此外，新增指令时需要修改 Executor、command 两个地方，这种设计分散了逻辑，增加了维护成本，容易引入错误。

```
std::unordered_map<char, std::function<void(PoseHandler &)>> cmderMap{
            {'M', MoveCommand()},       // 前进
            {'L', TurnLeftCommand()},   // 左转
            {'R', TurnRightCommand()}, // 右转
            {'F', FastCommand()},    // 快速
}
```

解决方案：

(1) 指令对象生成与执行分开：先生成指令列表，再执行指令

(2) 抽取指令对象处理为 command 层：在 command 层中处理所有与指令对象相关的操作，在 Executor 层只负责指令的调度，而不直接处理具体的逻辑。从而实现逻辑与数据分离，提升代码的模块化程度， 进一步降低 Executor 和 command 之间的耦合性。

(3) 使用单体对象：在 command 层中只生成一个 cmderMap，并将其作为全局单例对象保存，避免重复生成，从而避免了每次调用都创建 cmderMap，减少不必要的资源消耗。

2.2 指令实现复杂

当需要新添加的指令需要多个操作（如前进、左转、后退等）完成，同时其执行受车速、倒车状态等上下文状态的影响式，不同条件下会产生不同的处理逻辑，导致条件分支复杂。需要大量的 if-else 语句和相应的处理操作以保证掉头逻辑的正确性和一致性。随着需求变化，掉头逻辑的扩展和维护变得困难且代码可读性较差。

解决方案：任务编排

(1) 原子操作的组合：掉头指令由多个操作组成。在执行过程中，这些操作按照预定义的顺序

依次执行，完成复杂的业务任务

(2) 失败处理：企业软件开发中，可能会有某些操作失败。为确保系统能够有效地处理这些失败情况，通过重试、回滚或补偿机制，保证业务流程的完整性和一致性

(3) 实现步骤：

① 原子指令抽象。定义原子指令，明确每个原子操作的输入、输出和执行逻辑。例如，左转 90 度、向前进 1 格等；定义抽象接口，为每个原子指令定义统一的接口，使得不同的原子指令可以通过相同的方式调用

② Command 指令编排：将多个原子指令按照业务需求进行组合，形成一个完整的任务。例如，TR 指令在加速状态下的顺序是：向前进 1 格->左转 90 度->向前进 1 格->左转 90 度

③ 原子指令批量操作调用：高效地执行编排好的原子指令列表，将编排好的原子指令列表传递给执行器，执行器按顺序调用每个原子指令

六、过程和体会

1. 遇到的主要问题和解决方法

（1）编译失败：在平常的编程的练习中，我们大多遇到的都是些语法错误，但在本次实验中，由于第一次使用 Vscode 进行项目构建。文件，变量名繁多，头文件包含混乱，我遇到了多次编译链接错误。有些是函数名声明和定义不符，有些是编译 lib 库和引用 lib 库选项不一致，有些是头文件的重复引用……造成了大量“无法解析的外部命令”、“已经在.obj 中定义”等报错。经过反复 debug，相关解决方案的学习，我积累了许多开发、组织项目的经验。此外，在实验初期也经常因为 Cmake，编译器选择版本问题导致无法正常运行，通过检查环境配置，重新安装并选择正确版本后解决了问题。

（2）测试失败：在功能测试中，经常存在某些测试用例未能通过。尤其是存在加速，倒车等复杂状态时，由于没有理清任务需求，导致状态切换未正常处理。通过 GDB 调试工具，或在测试用例中增加新的断言，设置断点等操作，我们可以逐步分析方法逻辑，发现问题，确保不同状态组合下的行为符合预期。

（3）反复修改优化不当导致错误：实验过程中，我们根据实验指南一步步完成需求，同时又要不断分析不足，进行优化。在修改过程中，由于未完全理解指南中的设计思想和原因，只会盲目操作，导致在某些地方没有进行对应的更改，引起未知错误。通过请教老师和 AI，我逐步理解了设计原则，代码组织和结构，对 C++的各种技术和特性有了更深的体会。

2. 实验的体会

在本次实验中，我深入理解了 C++编程语言的强大功能，并对软件开发的工程实践有了更为深刻的理解。这不仅是对语言特性的学习，也是一次全面提升综合能力的实践。从基础的编程习惯到复杂的软件架构设计，每一步都让我体会到高质量代码的重要性。

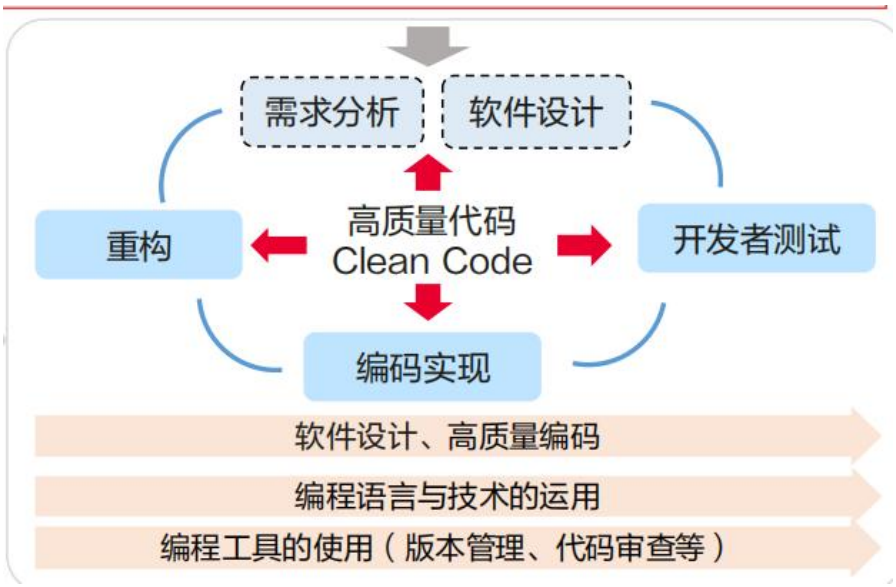


图 6-1 软件开发流程和技术

良好的编程习惯是高效开发的基础。在项目实践过程中，我深刻体会到代码的清晰性和规范性对整个开发流程的重大影响。合理的命名、清晰的逻辑和规范的格式，不仅提升了代码的可读性，也显著降低了后续修改和调试的成本。通过实验，我学习了驼峰命名法的规范，了解如何减少符号依赖，比如尽量避免使用”extern”关键字，除非引用某些汇编或者编译器生成符号，不要在头文件中使用匿名 namespace 或 static 定义非外部可见符号，尽可能多使用 private 和匿名命名空间隐藏符号，使用 P-IMPL 手法，将头文件中无需暴露的细节转移到实现文件中。

类别	命名风格	形式
命名空间，类类型，结构体类型，联合体类型，枚举类型，typedef定义的类型，类型别名	大驼峰	AbbBbb
函数（包括成员函数，作用域内函数，全局函数）	大驼峰	AbbBbb
全局变量（包括全局、文件、namespace域下的变量以及相应作用域下的静态变量）	带’g_’前缀的小驼峰	g_aaaBbb
类成员变量	小驼峰	aaaBbb
局部变量，函数参数，宏参数，结构体和联合体中的成员变量	小驼峰	aaaBbb
枚举值，常量	全大写，下划线分割	AAA_BBB

图 6-2 驼峰命名法风格

除了编程习惯，利用工程化工具让我获得了新的视野。Git 的版本控制让我能够更直观地掌握项目的发展轨迹。在功能开发和调试过程中，分支管理帮助我在不同版本之间灵活切换，而清晰的提交记录则让我能够快速定位错误或回滚到上一个稳定状态。同时，CMake 的使用简化了项目构建流程，尤其在处理复杂依赖和多模块协作时，它展现出了极大的便利性。通过配置自动化编译脚本，我逐渐掌握了有效组织项目结构的方法，从而提升了开发效率。

命令	作用	remote repo	local repo	working tree	index
pull	从远程仓库获取最新版本并合并到本地分支		更新并合并	更新并合并	更新并合并
fetch	从远程仓库获取最新版本，但不合并到本地分支		更新		
clone	克隆远程仓库到本地		新建并初始化	新建并初始化	新建并初始化
push	将本地分支的提交推送到远程仓库	更新			
checkout	切换分支或恢复工作区文件			更新	
add	将工作区的修改添加到暂存区				更新
commit	将暂存区的修改提交到本地仓库		更新		清空

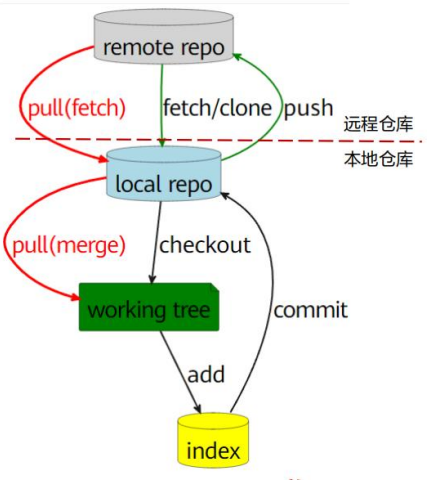


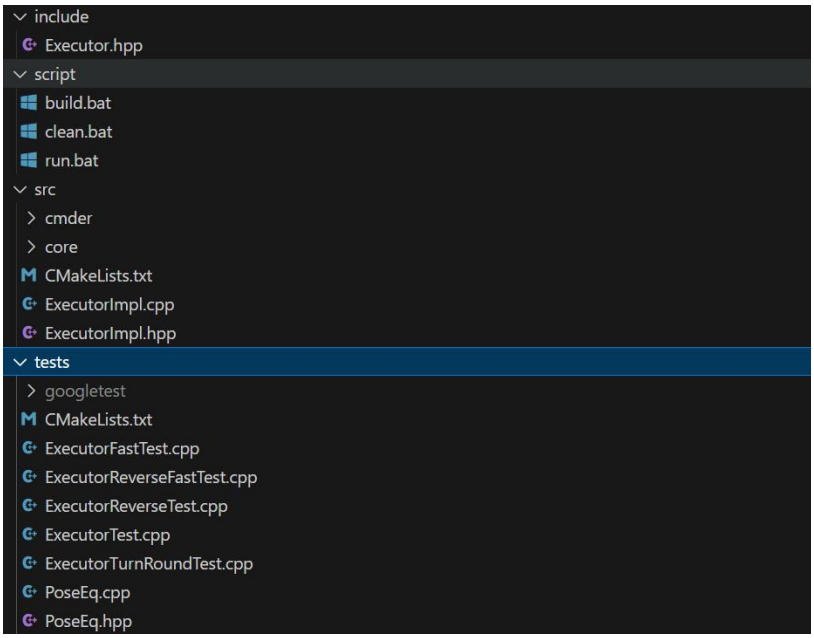
图 6-3 git 代码管理

在解决具体问题的过程中，C++语言特性的强大表现让我深刻感受到编程的魅力。面向对象编程的核心思想贯穿了整个项目设计。通过抽象类与继承机制的结合，我实现了系统的灵活扩展。多态性的应用让指令解析和处理更为优雅，而封装特性则有效降低了不同模块之间的耦合。对 Lambda 表达式的应用让我体会到了它在简化代码方面的价值，无论是高效定义匿名函数，还是在测试框架中的灵活运用，都让我对现代 C++特性有了更深入的理解。设计模式的引入则为项目的实现注入了更多的灵活性。在车辆控制系统中，表驱动模式的应用让我摆脱了冗长的条件判断逻辑，将命令字符和操作函数的映射清晰地分离开来。这种设计让代码更加简洁，扩展性也大幅提升。例如，当新增指令时，只需在映射表中添加对应的条目，无需修改主逻辑，这种优雅的解决方式让我认识到设计模式对复杂系统管理的重要性。命令模式的使用同样为系统注入了灵活与可维护性，每个指令对应独立的类，实现了逻辑的高度解耦，为代码的扩展和重构提供了充分的空间。

软件测试的实践使我深刻理解了测试驱动开发的重要性。以 Google Test 框架为工具，我掌握了如何通过单元测试验证各模块的功能。如何构建测试防护网和相关原则。比如 FIRST 原则，正交分解法，测试命名规则。理解并应用了“测试即文档”的概念，使测试用例名称清晰明了，便于理解和维护在对车辆状态切换逻辑进行测试时，我设计了多种组合场景，并通过断言验证输出是否符合预期。这一从需求到实现再到测试的完整开发流程，使我逐渐培育了严谨的编程思维。在进行加速与倒车状态叠加测试时，复杂的状态管理逻辑需要细致的验证，每次测试结果的通过都增强了我对系统可靠性的信心，也让我意识到测试的重要性不仅在于功能验证，更在于提升开发者对代码的理解与掌控能力。

在整个实验过程中所面临的挑战，亦成为了宝贵的学习契机。初期由于对开发环境的不熟悉，项目构建中频繁遭遇链接错误和头文件包含冲突，这促使我不断查询资料，学习 CMake

和编译器的相关知识。经过多次调试与优化，我逐渐理解了如何合理组织代码文件、如何高效利用工具以及如何规范使用项目结构。复杂逻辑的设计让我在解决问题的过程中感受到数学与编程结合的力量，例如通过数学关系优化转向操作，使得代码逻辑简洁之余，运行效率亦得到了提升。这些经验让我深刻领悟到，编程不仅仅是代码的编写，更是逻辑推理与系统设计的有机结合



。

图 6-4 目录结构设计

通过本次实验，我对企业级软件开发有了初步的认识。大型软件的开发需要未来导向的设计思维，每一行代码均需考虑其可扩展性与可维护性。在实验中，我尝试通过模块化设计和面向对象的抽象来实现这一目标，并在实践中深刻体会到了代码质量与开发效率之间的密切相关性。同时，我也认识到，企业软件开发不仅需要过硬的技术，更需要良好的合作能力、工具的熟练应用以及对规范的严格遵循。

此次实验不仅为我提供了技术锻炼的机会，更使我的思维得到了提升。我逐渐体会到编程的乐趣不仅在于实现功能的满足感，更在于解决问题的创造力与系统设计的成就感。从理论到实践的转变，让我看到了自身能力的提升，这次实验所积累的经验将为未来的开发垫下基础。