**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

Курсовая работа

**по дисциплине**

**«****Объектно-ориентированное программирование»**

Тема: «Разработка приложений на основе принципов объектно-ориентированного подхода»

|  |  |  |
| --- | --- | --- |
| Студент гр. 1335 |  | Максимов Ю.Е. |

|  |  |  |
| --- | --- | --- |
| Преподаватель: |  | Новакова Н.Е. |

Санкт-Петербург

2024

**ЗАДАНИЕ**

**на курсовую работу**

|  |  |  |
| --- | --- | --- |
| Студент Максимов Ю. Е. | | |
| Группа 1335 | | |
| Тема работы: Разработка приложений на основе принципов объектно-ориентированного подхода | | |
| Исходные данные:  Среда разработки: CLion  Язык программирования: C++ | | |
| Содержание пояснительной записки:  «Содержание», «Введение», «Цель работы», «Задание», «Теоретические сведения», «Формализация задачи», «Спецификация программы», «Руководство пользователя», «Руководство программиста», «Контрольный пример», «Листинг программы», «Заключение», «Список использованных источников» | | |
| Предполагаемый объем пояснительной записки:  Не менее 40 страниц. | | |
| Дата выдачи задания: 21.06.2023 | | |
| Дата сдачи курсовой работы: 26.06.2023 | | |
| Дата защиты курсовой работы: 26.06.2023 | | |
| Студент |  | Максимов Ю.Е. |
| Преподаватель |  | Новакова Н.Е. |

**Аннотация**

Курсовая работа содержит в себе решение трех задач на основе объектно-ориентированного подхода. В первой задаче рассмотрена иерархия классов, во второй – реализация алгоритма Дейкстры на графе, в третьей – разработка имитационной модели.

На основе этих моделей были разработаны приложения, включающие в себя различные пользовательские классы. Полученные результаты приведены в работе.

**Summary**

Course work contains the solution of three problems based on an object-oriented approach. In the first task, the class hierarchy is considered, in the second, the implementation of the Dijkstra algorithm on a graph, in the third, the development of a simulation model.

Based on these models, applications have been developed that include various user interfaces. The results obtained are presented in the work.

**содержание**

|  |  |  |
| --- | --- | --- |
|  | Введение  Цель работы | 6  6 |
| 1. | Разработка объектной модели | 7 |
| 1.1. | Задание | 7 |
| 1.2.  1.3.  1.4.  1.5.  1.6.  1.7.  1.8. | Теоретические сведения  Формализация задачи  Спецификация программы  Руководство пользователя  Руководство программиста  Контрольный пример  Листинг программы | 7  7  8  9  10  10  14 |
| 2. | Работа с графами | 18 |
| 2.1. | Задание | 18 |
| 2.2.  2.3.  2.4.  2.5.  2.6.  2.7.  2.8. | Теоретические сведения  Формализация задачи  Спецификация программы  Руководство пользователя  Руководство программиста  Контрольный пример  Листинг программы | 18  19  19  20  20  21  21 |
| 3. | Имитационное моделирование | 24 |
| 3.1. | Задание | 24 |
| 3.2.  3.3.  3.4.  3.5.  3.6.  3.7.  3.8. | Теоретические сведения  Формализация задачи  Спецификация программы  Руководство пользователя  Руководство программиста  Контрольный пример  Листинг программы | 24  25  26  28  30  31  36 |
|  | Заключение | 50 |
|  | Список использованных источников | 51 |

введение

Курсовая работа направлена на создание приложений на основе объектно-ориентированного подхода на языке С++. В ней рассматриваются иерархии классов и наследования, а также имитационные модели. Для первых двух задач реализовывается консольное приложение для взаимодействия с пользователем, для третьей используются Qt.

ЦЕЛЬ РАБОТЫ

Целью курсовой работы является закрепление теоретических знаний и получение практических навыков разработки программного обеспечения на основе объектно-ориентированного подхода.

1. Разработка объектной модели
   1. Задание

Вариант 12

Разработать программу для представления спортивных и гоночных автомобилей.

* 1. Теоретические сведения

Гоночные и спортивные автомобили, могут быть различной марки, а так же различными деталями, чем их совершенствуя.

* 1. Формализация задачи

FactoryCar – фабрика для создания, спортивных или гоночных автобилий.

Car – базовый клас, имеющий метод out\_car для вывода в консоль, а так же поля где хранятся детали.

RacingCar – класс наследника Car, представляюзий собой гоночный автомобиль.

SportCar – кдасс наследника Carб представляющий собой сопртивный автомобиль.

Detail - базовый класс для детали, имеет метод out\_detail для вывода на консоль информации о детали.

Body – класс наследник Detail, представляющий из себя кузов.

Engine - класс наследник Detail, представляющий двигатель.

Transmission - класс наследник Detail, представляющий транссмисию.

Wheel - класс наследник Detail, представляющий колесо.

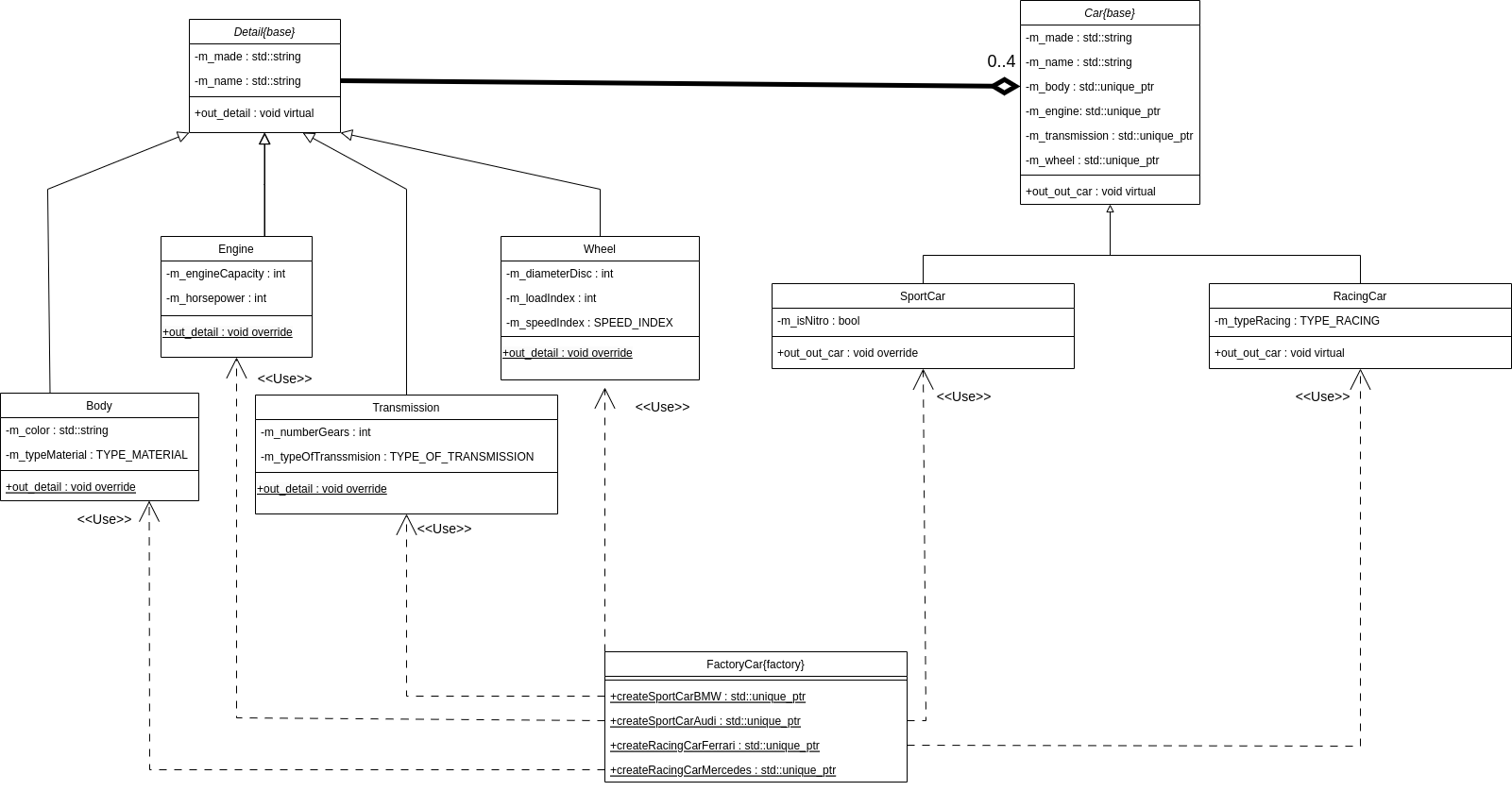


Рисунок 1.3.1. UML диаграмма классов

* 1. Спецификация программы

Таблица 1.3.1 Описание методов класса Car

| Методы | Возвращаемый тип | Модификатор доступа | Входные параметры | Назначение |
| --- | --- | --- | --- | --- |
| out\_car | void | public | - | Отображение машины |

Таблица 1.3.2 Описание методов класса RacingCar

| Методы | Возвращаемый тип | Модификатор доступа | Входные параметры | Назначение |
| --- | --- | --- | --- | --- |
| out\_car | void | public | - | Отображение машины |

Таблица 1.3.3 Описание методов класса SportCar

| Методы | Возвращаемый тип | Модификатор доступа | Входные параметры | Назначение |
| --- | --- | --- | --- | --- |
| out\_car | void | public | - | Отображение машины |

Таблица 1.3.4 Описание методов класса Detail

| Методы | Возвращаемый тип | Модификатор доступа | Входные параметры | Назначение |
| --- | --- | --- | --- | --- |
| out\_detail | void | public | - | Отображение детали |

Таблица 1.3.4 Описание методов класса Body

| Методы | Возвращаемый тип | Модификатор доступа | Входные параметры | Назначение |
| --- | --- | --- | --- | --- |
| out\_detail | void | public | - | Отображение корпуса |

Таблица 1.3.4 Описание методов класса Engine

| Методы | Возвращаемый тип | Модификатор доступа | Входные параметры | Назначение |
| --- | --- | --- | --- | --- |
| out\_detail | void | public | - | Отображение мотора |

Таблица 1.3.4 Описание методов класса Transmission

| Методы | Возвращаемый тип | Модификатор доступа | Входные параметры | Назначение |
| --- | --- | --- | --- | --- |
| out\_detail | void | public | - | Отображение трансмиссии |

Таблица 1.3.4 Описание методов класса Wheel

| Методы | Возвращаемый тип | Модификатор доступа | Входные параметры | Назначение |
| --- | --- | --- | --- | --- |
| out\_detail | void | public | - | Отображение колеса |

Таблица 1.3.4 Описание методов фабрики FactoryCar

| Методы | Возвращаемый тип | Модификатор доступа | Входные параметры | Назначение |
| --- | --- | --- | --- | --- |
| createSportCarBMW | std::unique\_ptr<car::Car> | public | - | Создание спортивной машины БМВ |
| createSportCarAudi | std::unique\_ptr<car::Car> | public | - | Создание спортивной машины Ауди |
| createRacingCarFerrari | std::unique\_ptr<car::Car> | public | - | Создание гоночной машины Феррари |
| out\_createRacingCarMercedes | std::unique\_ptr<car::Car> | public | - | Создание гоночной машины Мерседес |

Таблица 1.3.4 Описание полей класса Car

| Поля | Тип | Модификатор доступа | Назначение |
| --- | --- | --- | --- |
| m\_made | std::string | private | Изготовитель |
| m\_name | std::string | private | Имя |
| m\_body | std::unique\_ptr<detail::Detail> | private | Корпус |
| m\_engine | std::unique\_ptr<detail::Detail> | private | Двигатель |
| m\_transmission | std::unique\_ptr<detail::Detail> | private | Трансмиссия |
| m\_wheel | std::unique\_ptr<detail::Detail> | private | Колесо |

Таблица 1.3.4 Описание полей класса RacingCar

| Поля | Тип | Модификатор доступа | Назначение |
| --- | --- | --- | --- |
| m\_typeRacing | TYPE\_RACING | private | Для какой гонки |

Таблица 1.3.4 Описание полей класса SportCar

| Поля | Тип | Модификатор доступа | Назначение |
| --- | --- | --- | --- |
| m\_isNitro | bool | private | Есть ли нитро |

Таблица 1.3.4 Описание полей класса Detail

| Поля | Тип | Модификатор доступа | Назначение |
| --- | --- | --- | --- |
| m\_made | std::string | private | Изготовитель |
| m\_name | std::string | private | Имя |

Таблица 1.3.4 Описание полей класса Body

| Поля | Тип | Модификатор доступа | Назначение |
| --- | --- | --- | --- |
| m\_color | std::string | private | Цвет |
| m\_typeMaterial | TYPE\_MATERIAL | private | Тип матерьяла |

Таблица 1.3.4 Описание полей класса Engine

| Поля | Тип | Модификатор доступа | Назначение |
| --- | --- | --- | --- |
| m\_engineCapacity | int | private | Обьем двигателя |
| m\_horsepower | int | private | Лошадиные силы |

Таблица 1.3.4 Описание полей класса Transmission

| Поля | Тип | Модификатор доступа | Назначение |
| --- | --- | --- | --- |
| m\_numberGears | int | private | Сколько ступеней |
| m\_typeOfTransmission | TYPE\_OF\_TRANSMISSION | private | Тип трансмиссии |

Таблица 1.3.4 Описание полей класса Wheel

| Поля | Тип | Модификатор доступа | Назначение |
| --- | --- | --- | --- |
| m\_diameterDisc | int | private | Диаметр Дисков |
| m\_loadIndex | int | private | Максимум нагрузки |
| m\_speedIndex | SPEED\_INDEX | private | Максимум скорости |

* 1. Руководство пользователя
     1. Назначение программы

Представление гоночных и спортивных машин.

* + 1. Условие выполнения программы:

Для запуска программы необходим ggc/clang, cmake, Ninja.

* + 1. Выполнение программы:

1. Пользователь запускает программу.
2. В консоли выводится разный типаж гоночных и спортивных автомобилей.
   1. Руководство программиста
      1. Запуск программы:

Для запуска программы необходим ggc/clang, cmake, Ninja.

* + 1. Характеристика программы

Программа имеет список команд для функционирования, позволяя продемонстрировать гоночные и спортивные автомобили.

* + 1. Входные и выходные данные

Программа не предполагает взаимодействие с пользователем напрямую. Новые данные об объектах вносит программист. Результатом служит вывод на консоль.

* 1. Контрольный пример

На рисунке 1.7.1 было произведено создания 2 гоночных и 2 спортивных автомобилей с разными комплектующими :

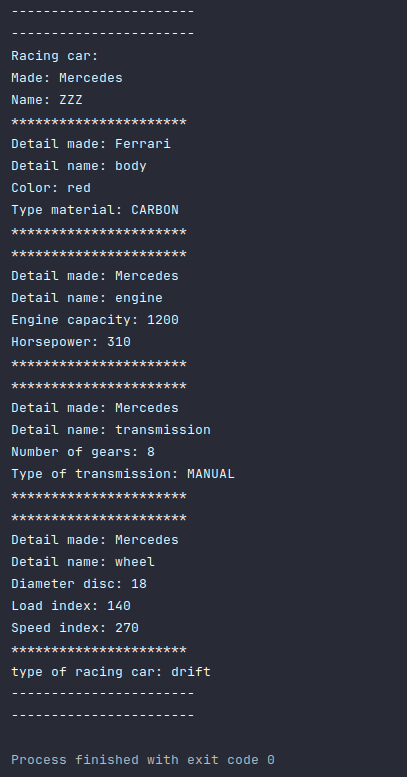
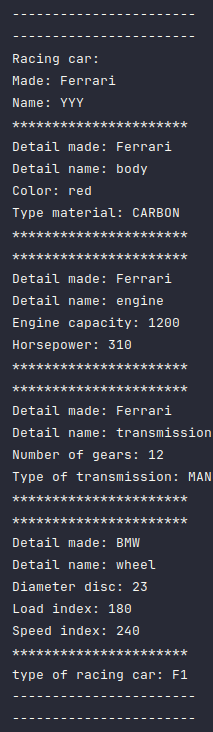
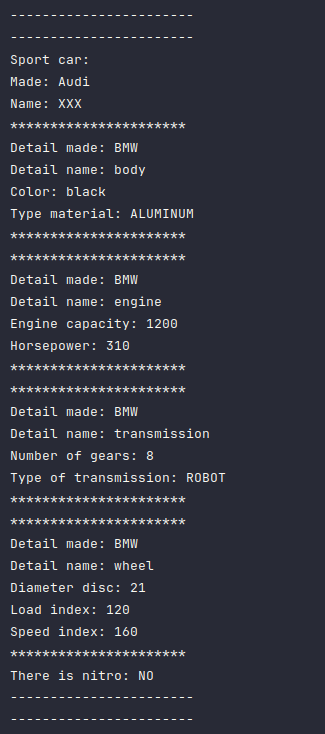
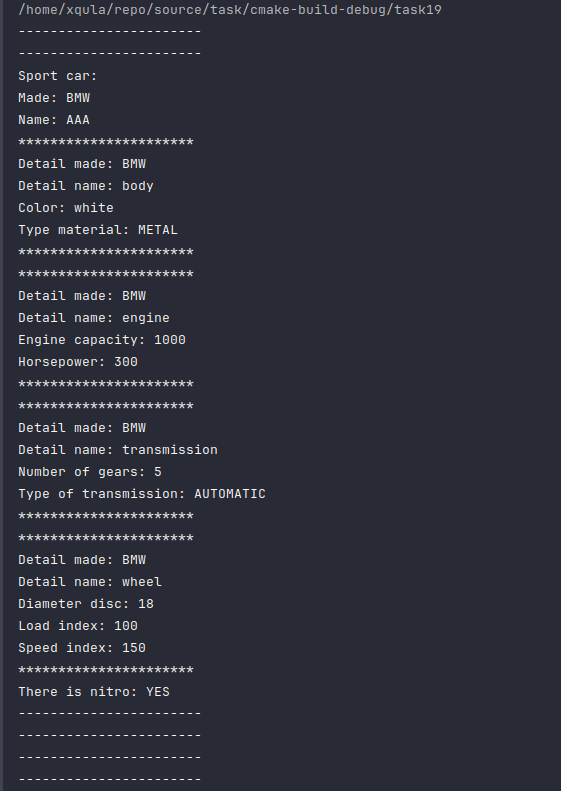


Рисунок 1.7.1 Работа программы

* 1. Листинг программы

car.cpp

//  
// Created by xqula on 21.06.24.  
//  
#pragma once  
#include "memory"  
#include "car/car.h"  
namespace factory {  
 class FactoryCar {  
 public:  
 FactoryCar() = default;  
  
 virtual ~FactoryCar() = default;  
  
 static auto createSportCarBMW() -> std::unique\_ptr<car::Car>;  
  
 static auto createSportCarAudi() -> std::unique\_ptr<car::Car>;  
  
 static auto createRacingCarFerrari() -> std::unique\_ptr<car::Car>;  
  
 static auto createRacingCarMercedes() -> std::unique\_ptr<car::Car>;  
 };  
  
}

**car.h**

//  
// Created by xqula on 21.06.24.  
//  
#pragma once  
  
#include "string"  
#include "detail/detail.h"  
#include "memory"  
namespace car {  
 class Car {  
 public:  
 explicit Car(  
 std::string *made*,  
 std::string *name*,  
 std::unique\_ptr<detail::Detail> *body*,  
 std::unique\_ptr<detail::Detail> *engine*,  
 std::unique\_ptr<detail::Detail> *transmission*,  
 std::unique\_ptr<detail::Detail> *wheel*);  
 virtual ~Car() = default;  
 virtual auto out\_car() -> void;  
 private:  
 std::string m\_made;  
 std::string m\_name;  
 std::unique\_ptr<detail::Detail> m\_body;  
 std::unique\_ptr<detail::Detail> m\_engine;  
 std::unique\_ptr<detail::Detail> m\_transmission;  
 std::unique\_ptr<detail::Detail> m\_wheel;  
 };  
  
}

**racingcar.h**

//  
// Created by xqula on 21.06.24.  
//  
#pragma once  
#include "car.h"  
namespace car{  
 enum TYPE\_RACING{  
 **F1** = 0,  
 **DRIFT** = 1,  
 **RALLY** = 2  
 };  
 class RacingCar: public Car{  
 public:  
 explicit RacingCar(  
 std::string *made*,  
 std::string *name*,  
 std::unique\_ptr<detail::Detail> *body*,  
 std::unique\_ptr<detail::Detail> *engine*,  
 std::unique\_ptr<detail::Detail> *transmission*,  
 std::unique\_ptr<detail::Detail> *wheel*,  
 TYPE\_RACING *typeRacing* );  
 ~RacingCar() override = default;  
 auto out\_car() -> void override;  
 private:  
 TYPE\_RACING m\_typeRacing;  
 };  
  
}

**racingcar.cpp**

//  
// Created by xqula on 21.06.24.  
//  
#include "racingcar.h"  
#include "iostream"  
car::RacingCar::RacingCar(std::string *made*,  
 std::string *name*,  
 std::unique\_ptr<detail::Detail> *body*,  
 std::unique\_ptr<detail::Detail> *engine*,  
 std::unique\_ptr<detail::Detail> *transmission*,  
 std::unique\_ptr<detail::Detail> *wheel*,  
 car::TYPE\_RACING *typeRacing*):  
 Car(std::move(*made*), std::move(*name*), std::move(*body*), std::move(*engine*), std::move(*transmission*), std::move(*wheel*)),  
 m\_typeRacing(*typeRacing*)  
{  
}  
auto car::RacingCar::out\_car() -> void {  
 std::**cout** << "-----------------------" << std::endl;  
 std::**cout** << "-----------------------" << std::endl;  
 std::**cout** << "Racing car:" << std::endl;  
 Car::out\_car();  
 std::**cout** << "type of racing car: ";  
 switch (static\_cast<int>(m\_typeRacing)) {  
 case 0:  
 std::**cout** << "F1" << std::endl;  
 break;  
 case 1:  
 std::**cout** << "drift" << std::endl;  
 break;  
 case 2:  
 std::**cout** << "rally" << std::endl;  
 break;  
 default:  
 std::**cout** << "unknown" << std::endl;  
 break;  
 }  
 std::**cout** << "-----------------------" << std::endl;  
 std::**cout** << "-----------------------" << std::endl;  
}

**sportcar.h**

//  
// Created by xqula on 21.06.24.  
//  
#pragma once  
  
#include "car.h"  
namespace car {  
 class SportCar: public Car {  
 public:  
 explicit SportCar(  
 std::string *made*,  
 std::string *name*,  
 std::unique\_ptr<detail::Detail> *body*,  
 std::unique\_ptr<detail::Detail> *engine*,  
 std::unique\_ptr<detail::Detail> *transmission*,  
 std::unique\_ptr<detail::Detail> *wheel*,  
 bool *isNitro* );  
 ~SportCar() override = default;  
 auto out\_car() -> void override;  
 private:  
 bool m\_isNitro;  
 };  
  
}

**sportcar.cpp**

//  
// Created by xqula on 21.06.24.  
//  
#include "sportcar.h"  
#include "iostream"  
car::SportCar::SportCar(std::string *made*,  
 std::string *name*,  
 std::unique\_ptr<detail::Detail> *body*,  
 std::unique\_ptr<detail::Detail> *engine*,  
 std::unique\_ptr<detail::Detail> *transmission*,  
 std::unique\_ptr<detail::Detail> *wheel*,  
 bool *isNitro*):  
 Car(std::move(*made*), std::move(*name*), std::move(*body*), std::move(*engine*), std::move(*transmission*), std::move(*wheel*)),  
 m\_isNitro(*isNitro*)  
{  
}  
auto car::SportCar::out\_car() -> void {  
 std::**cout** << "-----------------------" << std::endl;  
 std::**cout** << "-----------------------" << std::endl;  
 std::**cout** << "Sport car:" << std::endl;  
 Car::out\_car();  
 std::**cout** << "There is nitro: ";  
 if(m\_isNitro){  
 std::**cout** << "YES" << std::endl;  
 } else {  
 std::**cout** << "NO" << std::endl;  
 }  
 std::**cout** << "-----------------------" << std::endl;  
 std::**cout** << "-----------------------" << std::endl;  
}

**body.cpp**

//  
// Created by xqula on 21.06.24.  
//  
#include "body.h"  
#include "iostream"  
detail::Body::Body(std::string *made*, std::string *name*, std::string *color*, detail::TYPE\_MATERIAL *typeMaterial*)  
 : Detail(std::move(*made*), std::move(*name*)),  
 m\_color(std::move(*color*)),  
 m\_typeMaterial(*typeMaterial*)  
{  
}  
auto detail::Body::out\_detail() -> void {  
 std::**cout** << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << std::endl;  
 Detail::out\_detail();  
 std::**cout** << "Color: " << m\_color << std::endl;  
 std::**cout** << "Type material: ";  
 switch (static\_cast<int>(m\_typeMaterial)) {  
 case 0:  
 std::**cout** << "METAL" << std::endl;  
 break;  
 case 1:  
 std::**cout** << "ALUMINUM" << std::endl;  
 break;  
 case 2:  
 std::**cout** << "CARBON" << std::endl;  
 break;  
 }  
 std::**cout** << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << std::endl;  
}

**body.h**

//  
// Created by xqula on 21.06.24.  
//  
#include "detail.h"  
#include "string"  
namespace detail {  
 enum class TYPE\_MATERIAL {  
 **METAL** = 0,  
 **ALUMINUM** = 1,  
 **CARBON** = 2  
 };  
 class Body: public Detail {  
 public:  
 explicit Body(std::string *made*, std::string *name*, std::string *color*, TYPE\_MATERIAL *typeMaterial*);  
 auto out\_detail() -> void override;  
 private:  
 std::string m\_color;  
 TYPE\_MATERIAL m\_typeMaterial;  
 };  
}

**detail.cpp**

//  
// Created by xqula on 21.06.24.  
//  
#include "detail.h"  
#include <utility>  
#include "iostream"  
detail::Detail::Detail(std::string *made*, std::string *name*) : m\_made(std::move(*made*)), m\_name(std::move(*name*)) {}  
auto detail::Detail::out\_detail() -> void {  
 std::**cout** << "Detail made: " << m\_made << "\nDetail name: " << m\_name << std::endl;  
}

**detail.h**

//  
// Created by xqula on 21.06.24.  
//  
#pragma once  
  
#include "string"  
namespace detail {  
 class Detail {  
 public:  
 explicit Detail(std::string *made*, std::string *name*);  
 virtual ~Detail() = default;  
 virtual auto out\_detail() -> void;  
  
 private:  
 std::string m\_made;  
 std::string m\_name;  
 };  
}

**engine.cpp**

//  
// Created by xqula on 21.06.24.  
//  
#include "engine.h"  
#include <iostream>  
detail::Engine::Engine(std::string *made*, std::string *name*, const int *engineCapacity*, const int *m\_horsepower*):  
 Detail(std::move(*made*), std::move(*name*)),  
 m\_engineCapacity(*engineCapacity*),  
 m\_horsepower(*m\_horsepower*)  
{  
}  
auto detail::Engine::out\_detail() -> void {  
 std::**cout** << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << std::endl;  
 Detail::out\_detail();  
 std::**cout** << "Engine capacity: " << m\_engineCapacity << std::endl;  
 std::**cout** << "Horsepower: " << m\_horsepower << std::endl;  
 std::**cout** << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << std::endl;  
}

**engine.h**

//  
// Created by xqula on 21.06.24.  
//  
#pragma once  
  
#include "detail.h"  
namespace detail {  
 class Engine : public Detail {  
 public:  
 explicit Engine(std::string *made*, std::string *name*, int *engineCapacity*, int *m\_horsepower*);  
 auto out\_detail() -> void override;  
 private:  
 int m\_engineCapacity;  
 int m\_horsepower;  
 };  
  
}

**transmission.h**

//  
// Created by xqula on 21.06.24.  
//  
#pragma once  
  
#include "detail.h"  
namespace detail {  
 enum class TYPE\_OF\_TRANSMISSION {  
 **MANUAL** = 0,  
 **AUTOMATIC** = 1,  
 **VARIATION** = 2,  
 **ROBOT** = 3  
 };  
  
 class Transmission : public Detail {  
 public:  
 explicit Transmission(  
 std::string *made*,  
 std::string *name*,  
 int *numberGears*,  
 TYPE\_OF\_TRANSMISSION *typeOfTransmission*);  
 auto out\_detail() -> void override;  
 private:  
 int m\_numberGears;  
 TYPE\_OF\_TRANSMISSION m\_typeOfTransmission;  
 };  
}

**transmission.cpp**

//  
// Created by xqula on 21.06.24.  
//  
#include "transmission.h"  
#include "iostream"  
detail::Transmission::Transmission(std::string *made*,  
 std::string *name*,  
 const int *numberGears*,  
 const detail::TYPE\_OF\_TRANSMISSION *typeOfTransmission*)  
 : Detail(std::move(*made*), std::move(*name*)),  
 m\_numberGears(*numberGears*),  
 m\_typeOfTransmission(*typeOfTransmission*)  
{  
}  
auto detail::Transmission::out\_detail() -> void {  
 std::**cout** << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << std::endl;  
 Detail::out\_detail();  
 std::**cout** << "Number of gears: " << m\_numberGears << std::endl;  
 std::**cout** << "Type of transmission: ";  
 switch (static\_cast<int>(m\_typeOfTransmission)) {  
 case 0:  
 std::**cout** << "MANUAL" << std::endl;  
 break;  
 case 1:  
 std::**cout** << "AUTOMATIC" << std::endl;  
 break;  
 case 2:  
 std::**cout** << "VARIATION" << std::endl;  
 break;  
 case 3:  
 std::**cout** << "ROBOT" << std::endl;  
 break;  
 }  
 std::**cout** << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << std::endl;  
}

**Wheel.cpp**

//  
// Created by xqula on 21.06.24.  
//  
#include "wheel.h"  
#include <utility>  
#include "iostream"  
detail::Wheel::Wheel(  
 std::string *made*,  
 std::string *name*,  
 const int *diameterDisc*,  
 const int *loadIndex*,  
 const detail::SPEED\_INDEX *speedIndex*) :  
 m\_diameterDisc(*diameterDisc*),  
 m\_loadIndex(*loadIndex*),  
 m\_speedIndex(*speedIndex*),  
 Detail(std::move(*made*),std::move(*name*))  
{  
}  
/\*  
 \* Outputs the details of a car to the console.  
 \*  
 \* @param car The car to output the details of.  
 \*/  
auto detail::Wheel::out\_detail() -> void {  
 std::**cout** << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << std::endl;  
 Detail::out\_detail();  
 std::**cout** << "Diameter disc: " << m\_diameterDisc << std::endl;  
 std::**cout** << "Load index: " << m\_loadIndex << std::endl;  
 std::**cout** << "Speed index: " << static\_cast<int>(m\_speedIndex) << std::endl;  
 std::**cout** << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << std::endl;  
}

**Wheel.h**

//  
// Created by xqula on 21.06.24.  
//  
#pragma once  
  
#include "detail.h"  
namespace detail {  
 enum class SPEED\_INDEX {  
 **P** = 150,  
 **Q** = 160,  
 **R** = 170,  
 **S** = 180,  
 **T** = 190,  
 **U** = 200,  
 **H** = 210,  
 **V** = 240,  
 **W** = 270,  
 **Y** = 300  
 };  
  
 class Wheel: public Detail {  
 public:  
 explicit Wheel( std::string *made*,  
 std::string *name*,  
 int *diameterDisc*,  
 int *loadIndex*,  
 detail::SPEED\_INDEX *speedIndex*);  
 auto out\_detail() -> void override;  
 private:  
 int m\_diameterDisc{};  
 int m\_loadIndex{};  
 SPEED\_INDEX m\_speedIndex;  
 };  
  
}

**factorycar.cpp**

//  
// Created by xqula on 21.06.24.  
//  
#include "factorycar.h"  
#include "detail/body.h"  
#include "detail/engine.h"  
#include "detail/transmission.h"  
#include "detail/wheel.h"  
#include "car/sportcar.h"  
#include "car/racingcar.h"  
using namespace factory;  
  
/\*  
 \* Creates a new BMW sport car.  
 \*  
 \* @return A unique pointer to a new BMW sport car.  
 \*/  
auto FactoryCar::createSportCarBMW() -> std::unique\_ptr<car::Car> {  
 return std::make\_unique<car::SportCar>( "BMW",  
 "AAA",  
 std::make\_unique<detail::Body>( "BMW",  
 "body",  
 "white",  
 detail::TYPE\_MATERIAL::**METAL**),  
 std::make\_unique<detail::Engine>("BMW",  
 "engine",  
 1000,  
 300 ),  
 std::make\_unique<detail::Transmission>("BMW",  
 "transmission",\  
 5,  
 detail::TYPE\_OF\_TRANSMISSION::**AUTOMATIC**),  
 std::make\_unique<detail::Wheel>("BMW",  
 "wheel",  
 18,  
 100,  
 detail::SPEED\_INDEX::**P**),  
 true  
 );  
}  
/\*  
 \* Creates a new Audi sport car.  
 \*  
 \* @return A unique pointer to a new Audi sport car.  
 \*/  
auto FactoryCar::createSportCarAudi() -> std::unique\_ptr<car::Car> {  
 return std::make\_unique<car::SportCar>( "Audi",  
 "XXX",  
 std::make\_unique<detail::Body>( "BMW",  
 "body",  
 "black",  
 detail::TYPE\_MATERIAL::**ALUMINUM**),  
 std::make\_unique<detail::Engine>("BMW",  
 "engine",  
 1200,  
 310 ),  
 std::make\_unique<detail::Transmission>("BMW",  
 "transmission",\  
 8,  
 detail::TYPE\_OF\_TRANSMISSION::**ROBOT**),  
 std::make\_unique<detail::Wheel>("BMW",  
 "wheel",  
 21,  
 120,  
 detail::SPEED\_INDEX::**Q**),  
 false  
 );  
}  
/\*  
 \* Creates a new Ferrari racing car.  
 \*  
 \* @return A unique pointer to a new Ferrari racing car.  
 \*/  
auto FactoryCar::createRacingCarFerrari() -> std::unique\_ptr<car::Car> {  
 return std::make\_unique<car::RacingCar>( "Ferrari", "YYY", std::make\_unique<detail::Body>( "Ferrari",  
 "body",  
 "red",  
 detail::TYPE\_MATERIAL::**CARBON**),  
 std::make\_unique<detail::Engine>("Ferrari",  
 "engine",  
 1200,  
 310 ),  
 std::make\_unique<detail::Transmission>("Ferrari",  
 "transmission",\  
 12,  
 detail::TYPE\_OF\_TRANSMISSION::**MANUAL**),  
 std::make\_unique<detail::Wheel>("BMW",  
 "wheel",  
 23,  
 180,  
 detail::SPEED\_INDEX::**V**),  
 car::TYPE\_RACING::**F1** );  
}  
/\*  
 \* Creates a new Mercedes racing car.  
 \*  
 \* @return A unique pointer to a new Mercedes racing car.  
 \*/  
auto FactoryCar::createRacingCarMercedes() -> std::unique\_ptr<car::Car> {  
 return std::make\_unique<car::RacingCar>( "Mercedes", "ZZZ", std::make\_unique<detail::Body>( "Ferrari",  
 "body",  
 "red",  
 detail::TYPE\_MATERIAL::**CARBON**),  
 std::make\_unique<detail::Engine>("Mercedes",  
 "engine",  
 1200,  
 310 ),  
 std::make\_unique<detail::Transmission>("Mercedes",  
 "transmission",\  
 8,  
 detail::TYPE\_OF\_TRANSMISSION::**MANUAL**),  
 std::make\_unique<detail::Wheel>("Mercedes",  
 "wheel",  
 18,  
 140,  
 detail::SPEED\_INDEX::**W**),  
 car::TYPE\_RACING::**DRIFT** );  
}

**factorycar.h**

//  
// Created by xqula on 21.06.24.  
//  
#pragma once  
#include "memory"  
#include "car/car.h"  
namespace factory {  
 class FactoryCar {  
 public:  
 FactoryCar() = default;  
  
 virtual ~FactoryCar() = default;  
  
 static auto createSportCarBMW() -> std::unique\_ptr<car::Car>;  
  
 static auto createSportCarAudi() -> std::unique\_ptr<car::Car>;  
  
 static auto createRacingCarFerrari() -> std::unique\_ptr<car::Car>;  
  
 static auto createRacingCarMercedes() -> std::unique\_ptr<car::Car>;  
 };  
  
}

2. Работа с графами

2.1. Задание

Вариант задания Г-41-1

Для заданного орграфа найти кратчайший путь между вершинами 1 и 12. Задачу решить в общем виде. На рисунке рядом с вершинами указан ее вес.

В качестве контрольного примера используются следующий граф:

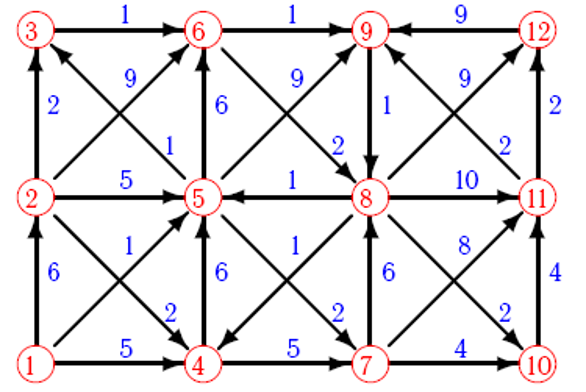


Рисунок 2.1.1 Граф для контрольного примера

2.2. Теоретические сведения

Граф (G(V,E)) – совокупность двух множеств — множества объектов V, называемого множеством вершин, и множества их парных связей E, называемого множеством рёбер. Элемент множества рёбер есть пара элементов множества вершин.

Взвешенный граф – граф, в котором каждому ребру поставлено в соответствие некоторое число, называемое весом ребра.

Матрица смежности – таблица, где как столбцы, так и строки соответствуют вершинам графа. В каждой ячейке этой матрицы записывается число, определяющее наличие связи от вершины-строки к вершине-столбцу (либо наоборот).

В теории графов транспортная сеть — ориентированный граф G = (V, E), в котором каждое ребро (u, v) ∈ E имеет неотрицательную пропускную способность c (u, v) ⩾ 0 и поток f(u,v). Выделяются две вершины: источник s и сток t такие, что любая другая вершина сети лежит на пути из s в t, при этом s ≠ t. Транспортная сеть может быть использована для моделирования, например, дорожного трафика.

Алгори́тм Де́йкстры (англ. Dijkstra’s algorithm) — алгоритм на графах, изобретённый нидерландским учёным Эдсгером Дейкстрой в 1959 году. Находит кратчайшие пути от одной из вершин графа до всех остальных. Алгоритм работает только для графов без рёбер отрицательного веса. Алгоритм широко применяется в программировании, например, его используют протоколы маршрутизации OSPF и IS-IS.

2.3. Формализация задачи

Core – класс содержащий в себе матрицу смежности, и выводящий информацию по кратчайщему пути в консоль

Dijlstra – класс реализующий метод поиска кратчайщего пути.

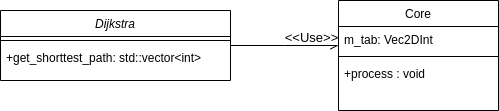


Рисунок 2.3.1 UML Диаграмма классов

2.4. Спецификация программы

Таблица 2.4.1 Описание методов класса Core

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Методы | Возвращаемый тип | Модификатор доступа | Входные параметры | Назначение |
| procces | void | public | void | Создание матрицы смежноости и вывода информации в консоль |

Таблица 2.4.2 Описание методов класса Deijkstra

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Методы | Возвращаемый тип | Модификатор доступа | Входные параметры | Назначение |
| get\_shorttest\_path | int | public | Std::vector<std::vector<int> | Нахождение кратчайщего пути |

2.5. Руководство пользователя

* + 1. Назначение программы

Данная программа позволяет найти кратчайщий путь в графе.

* + 1. Условие выполнения программы:

Для запуска программы необходим gcc/clang, cmake, ninja.

* + 1. Выполнение программы:

Программа разработана в виде консольного приложения. Она считывает матрицу смежности из метода Main и выводит кратчайщий путь.

* 1. **Руководство программиста**
     1. Запуск программы:

Для запуска программы необходим gcc/clang, cmake, ninja.

* + 1. Характеристика программы

Программа решает задачу в общем виде, находит кратчайщий путь с помощью алгоритма Дейкстры по матрице смежности, заданной в методе Main.

* + 1. Входные и выходные данные

Программа предполагает взаимодействие с пользователем напрямую. Новые данные вносит программист (изменяет матрицу смежности и ). Результатом служит вывод кратчайщего пути в консоль.

* 1. **Контрольный пример**

На рисунке 2.7.1 – работа программы для графа, данного в качестве контрольного примера:

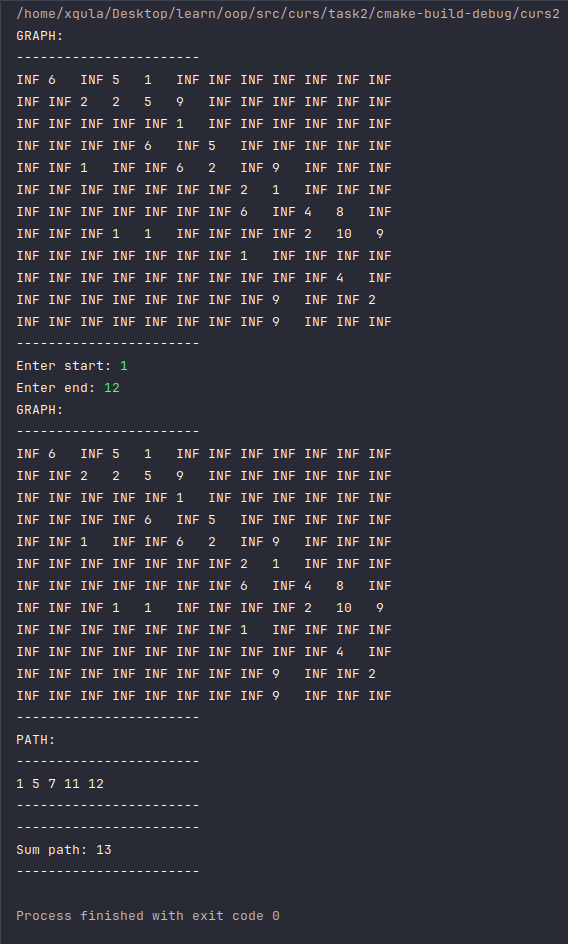


Рисунок 2.7.1 Работа программы

* 1. **Листинг программы**

**dijkstra.h**

//  
// Created by xqula on 23.06.24.  
//  
#pragma once  
#include "vector"  
#include "set"  
#include "limits"  
#include "ranges"  
#include "algorithm"  
namespace algo {  
 using VecInt = std::vector<int>;  
 using Vec2DInt = std::vector<VecInt>;  
 class Dijkstra {  
 public:  
 Dijkstra() = delete;  
 Dijkstra(const Dijkstra &) = delete;  
 Dijkstra operator=(const Dijkstra &) = delete;  
 static auto **get\_shortest\_path**(const Vec2DInt &*graph*)  
 -> VecInt;  
 };  
  
 inline auto Dijkstra::**get\_shortest\_path**(const Vec2DInt &*graph*)  
 -> VecInt  
 {  
 using Vec2DIntSize\_t = Vec2DInt::size\_type;  
 constexpr double INF = std::numeric\_limits<int>::**max**( );  
 const auto n = *graph*.size();  
 int s = 0; // стартовая вершина  
 std::vector<double> d(n, INF);  
 std::vector<int> p (n);  
 d[s] = 0;  
 std::set<std::pair<double,int>> q;  
 q.insert (std::make\_pair(d[s], s));  
 while (!q.empty()) {  
 const auto v = q.begin()->second;  
 q.erase (q.begin());  
 for (Vec2DIntSize\_t j=0; j<*graph*[v].size(); ++j) {  
 const auto to = j;  
 const auto len = *graph*[v][j];  
 if (d[v] + *graph*[v][j] < d[to]) {  
 q.erase (std::make\_pair(d[to], static\_cast<int>(to)));  
 d[to] = d[v] + len;  
 p[to] = v;  
 q.insert (std::make\_pair(d[to], static\_cast<int>(to)));  
 }  
 }  
 }  
 const auto t = *graph*.size()-1;  
 std::vector<int> path;  
 for (auto v=t; v!=s; v=p[v])  
 path.push\_back(static\_cast<int>(v));  
 path.push\_back (s);  
 std::ranges::reverse(path);  
 return path;  
 }  
}

**Core.cpp**

//  
// Created by xqula on 23.06.24.  
//  
#pragma once  
#include "vector"  
#include "set"  
#include "limits"  
#include "ranges"  
#include "algorithm"  
namespace algo {  
 using VecInt = std::vector<int>;  
 using Vec2DInt = std::vector<VecInt>;  
 class Dijkstra {  
 public:  
 Dijkstra() = delete;  
 Dijkstra(const Dijkstra &) = delete;  
 Dijkstra operator=(const Dijkstra &) = delete;  
 static auto **get\_shortest\_path**(const Vec2DInt &*graph*)  
 -> VecInt;  
 };  
  
 inline auto Dijkstra::**get\_shortest\_path**(const Vec2DInt &*graph*)  
 -> VecInt  
 {  
 using Vec2DIntSize\_t = Vec2DInt::size\_type;  
 constexpr double INF = std::numeric\_limits<int>::**max**( );  
 const auto n = *graph*.size();  
 int s = 0; // стартовая вершина  
 std::vector<double> d(n, INF);  
 std::vector<int> p (n);  
 d[s] = 0;  
 std::set<std::pair<double,int>> q;  
 q.insert (std::make\_pair(d[s], s));  
 while (!q.empty()) {  
 const auto v = q.begin()->second;  
 q.erase (q.begin());  
 for (Vec2DIntSize\_t j=0; j<*graph*[v].size(); ++j) {  
 const auto to = j;  
 const auto len = *graph*[v][j];  
 if (d[v] + *graph*[v][j] < d[to]) {  
 q.erase (std::make\_pair(d[to], static\_cast<int>(to)));  
 d[to] = d[v] + len;  
 p[to] = v;  
 q.insert (std::make\_pair(d[to], static\_cast<int>(to)));  
 }  
 }  
 }  
 const auto t = *graph*.size()-1;  
 std::vector<int> path;  
 for (auto v=t; v!=s; v=p[v])  
 path.push\_back(static\_cast<int>(v));  
 path.push\_back (s);  
 std::ranges::reverse(path);  
 return path;  
 }  
}

**Core.h**

//  
// Created by xqula on 23.06.24.  
//  
#pragma once  
#include "vector"  
namespace core {  
 using VecInt = std::vector<int>;  
 using Vec2DInt = std::vector<VecInt>;  
  
 class Core {  
 public:  
 explicit Core();  
 auto process() -> void;  
  
 private:  
 Vec2DInt m\_tab;  
 };  
  
}

**3. Имитационное моделирование**

* 1. **Задание**

Вариант 7

Модель проектного отдела

В проектном отделе разрабатывают новые изделия. Этим занимаются 4 проектировщика, каждый из которых специализируется на своём этапе проектирования. Весь процесс проектирования можно разбить на 4 этапа, за каждый из которых отвечает отдельный проектировщик. Задания на проектирование поступают через каждые A±B дней. Проектирование на каждом этапе занимает Ck±Dk дней, где k — номер проектировщика (номер этапа). Обычно проектирование протекает от 1-го этапа ко 2-му и т. д. Но может прийти срочный заказ и тогда необходимо выполнить в первую очередь его. 1-й проектировщик откладывает выполнявшийся ранее проект и начинает заниматься новым, а остальные пока продолжают заниматься прежними проектами. Когда материалы срочного проекта доходят до очередного проектировщика, он начинает заниматься им, откладывая предыдущую работу. После окончания срочной работы каждый проектировщик возвращается к своей прежней работе и заканчивает её. Проанализировать работу над 10 проектами, из которых 2 оказываются срочными (выбор — случайным образом). Два последовательных срочных проекта выстраиваются в очередь.

Результат сбора статистики должен быть выведен в текстовый файл.

* 1. **Теоретические сведения**

Имитационное моделирование заключается в создании процессов близких к реальным.

* 1. **Формализация задачи**

Worker – базовый клас для всех работников отдела

Manager – класс наследник Worker, релизует мост между сборщиками, и распределяет задачи.

Designer – класс наследник Worker, реализует сборщиков, которые выполняют задачи.

TaskBase – класс представляющий задачи.

Random – класс реализующий вихрь мейсона, для генирации псевдослучайных чисел.

Writer – класс руализующий запись в файл.

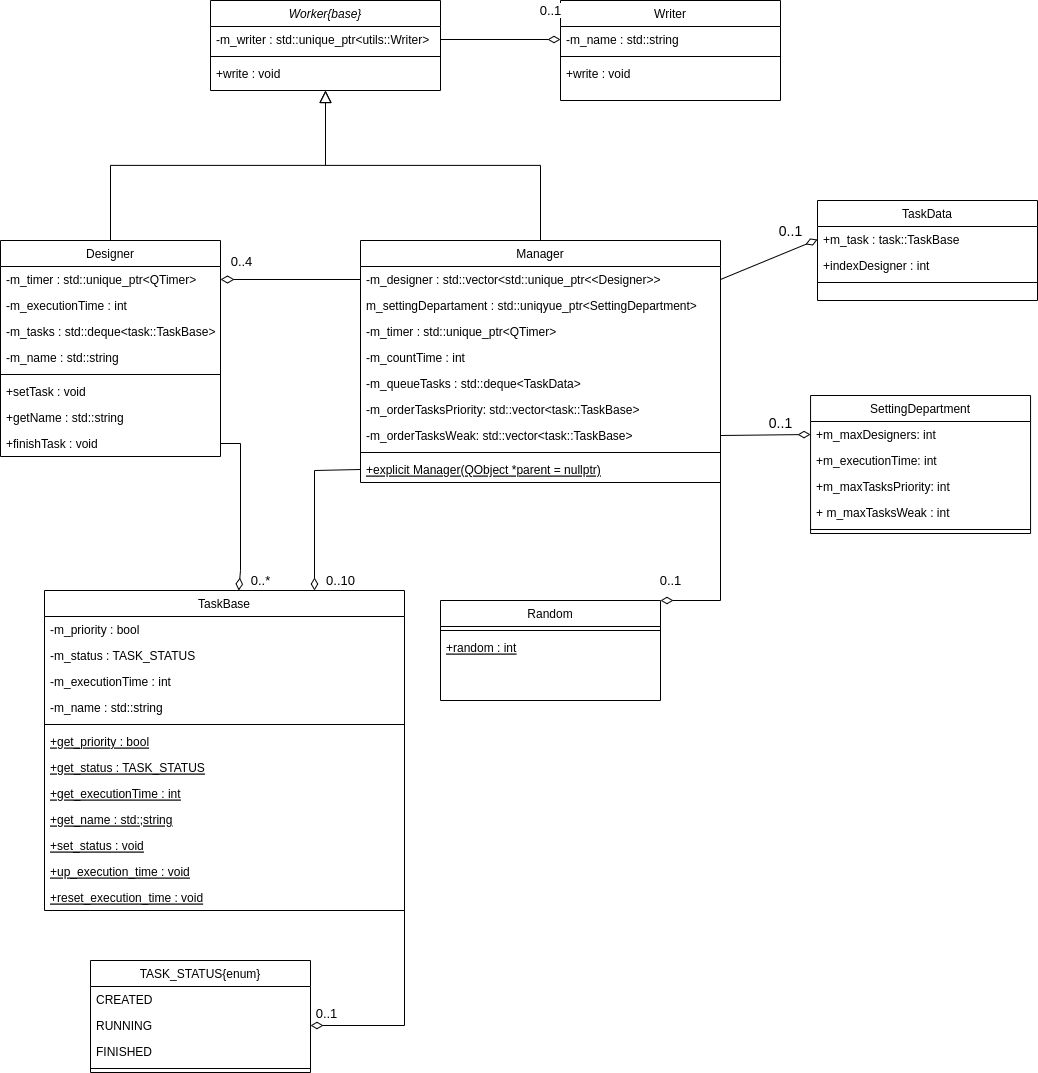


Рисунок 3.3.1 UML Диаграмма классов

* 1. **Спецификация программы**

Таблица 3.4.1 Описание методов класса Worker

| Методы | Возвращаемый тип | Модификатор доступа | Входные параметры | Назначение |
| --- | --- | --- | --- | --- |
| write | void | public | std::string\_view txt | Запись данных в файл |

Таблица 3.4.2 Описание методов классов Designer

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Методы | Возвращаемый тип | Модификатор доступа | Входные параметры | Назначение |
| setTask | void | public | const task::TaskBase &task | Принимает задачу |
| finishTask | мщшв | public | const task::TaskBase &task | Отправляет сигнал по завершению задачи |

Таблица 3.4.3 Описание методов класса Writer

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Методы | Возвращаемый тип | Модификатор доступа | Входные параметры | Назначение |
| write | void | public | std::string\_view txt | Запись данных в файл |

Таблица 3.4.4 Описание методов класса Random

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Методы | Возвращаемый тип | Модификатор доступа | Входные параметры | Назначение |
| random | void | public | int min, int max | Генерация пвседвослучайных числел |

Таблица 3.4.4 Описание методов класса TaskBase

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Методы | Возвращаемый тип | Модификатор доступа | Входные параметры | Назначение |
| reset\_execution\_time | Void | Public | Void | Обнуляет время |
| up\_execution\_time | void | Public | void | Добавляет время |
| set\_status | Void | Public | TASK\_STATUS | Меняет статус задачи |
| get\_name | std::string | Public | Void | Возвращает имя задачи |
| get\_executionTime | Int | Public | Void | Получение времени выполнение задачи |
| get\_status | TASK\_STATUS | public | void | Получение поля статуса |
| get\_priority | bool | public | void | Получение поля приоритета |

* 1. **Алгоритм**
     1. **Создаются с помощью вихря Мейсена рандомно количество задач обычных, а срочных 2, в рандомно порядке.**
     2. **Задачи выдаются менеджеру.**
     3. **По определённому истечению времени выдается задача первому разработчику**
     4. **Первый разработчик начинает работу над задачей**
     5. **Если прилетает срочная задача, обычную откладывает и начинает со срочной.**
     6. **Если прилетает еще одна задача, она становится второй.**
     7. **Как только он заканчивает с задачей он отдает менеджеру и говорит, что закончил и записывается в файл.**
     8. **Менеджер отдает следующему работнику и так по всем работникам(4).**
  2. **Руководство пользователя**
     1. **Назначение программы**

Данная программа позволяет моделировать работу проектного отдела.

* + 1. **Условие выполнения программы:**

Для запуска программы необходим gcc/clang, cmake, ninja, qt5.15

* + 1. **Выполнение программы:**

Программа разработана с использованием Qt5.15. Пользователь запускает программу и види работу проектного отдела в лог файле log.txt.

* 1. **Руководство программиста**
     1. **Запуск программы:**

Для запуска программы необходим gcc/clang, cmake, ninja, qt5.15

* + 1. **Характеристика программы**

Программа выполняется с использованием Qt5.15. Программа имеет интуитивно понятный интерфейс, проста и доступна в использовании.

* + 1. **Входные и выходные данные**

Выходные данные записываются в лог файл log.txt.

* 1. **Контрольный пример**



Рисунок 3.7.1 Работа программы

* 1. **Обработка информации вывода**
     1. **Мы видим, что срочные задачи забивают время работников, так как они каждый раз при запуске программы они рандомно располагаются, выполнение обычных задач затягивается. И становится очень большая очередь у работников из задач. А так же пока у одного большой поток задач, другие могут сидеть без дела, так как до низ задача не дошла.**
  2. **Листинг программы**

designer.cpp

//  
// Created by xqula on 24.06.24.  
//  
#include "designer.h"  
department::Designer::Designer(std::string *name*, QObject \*parent) :  
 Worker( "../log.txt", nullptr),  
 m\_timer(std::make\_unique<QTimer>(nullptr)),  
 m\_executionTime(5),  
 m\_name(std::move(*name*))  
{  
 m\_timer->start(200);  
 **connect**(m\_timer.get(), &QTimer::timeout, this, [this](){  
 if(!this->m\_tasks.empty()) {  
 m\_tasks.front().set\_status(task::TASK\_STATUS::RUNNING);  
 m\_tasks.front().up\_execution\_time();  
 if(m\_tasks.front().get\_executionTime() == m\_executionTime) {  
 m\_tasks.front().set\_status(task::TASK\_STATUS::FINISHED);  
 write("Designer " + m\_name + ": Finish task " + m\_tasks.front().get\_name());  
 emit finishTask(m\_tasks.front());  
 m\_tasks.pop\_front();  
 }  
 }  
 });  
  
}  
auto department::Designer::setTask(const task::TaskBase &*task*) -> void {  
 write("Designer " + m\_name + ": Start new task " + *task*.get\_name());  
 if(*task*.get\_priority()) {  
 if(not m\_tasks.empty()) {  
 if(m\_tasks.front().get\_priority()) {  
 const auto tmpTask = m\_tasks.front();  
 m\_tasks.pop\_front();  
 m\_tasks.push\_front(*task*);  
 m\_tasks.push\_front(tmpTask);  
 return;  
 }  
 }  
 m\_tasks.push\_front(*task*);  
 } else {  
 m\_tasks.push\_back(*task*);  
 }  
}  
auto department::Designer::getName() const -> std::string {  
 return m\_name;  
}

designer.h

//  
// Created by xqula on 24.06.24.  
//  
#pragma once  
#include "worker.h"  
#include "task/taskbase.h"  
#include "deque"  
#include "memory"  
#include "string"  
#include "QTimer"  
#include "QObject"  
namespace department {  
 class Designer: public Worker {  
 Q\_OBJECT  
 public:  
 explicit Designer(std::string *name*, QObject \**parent* = nullptr);  
 auto setTask(const task::TaskBase &*task*) -> void;  
 [[nodiscard]] auto getName() const -> std::string;  
 ~Designer() override = default;  
 signals:  
 void finishTask(const task::TaskBase &*task*);  
 private:  
 std::unique\_ptr<QTimer> m\_timer;  
 int m\_executionTime;  
 std::deque<task::TaskBase> m\_tasks;  
 std::string m\_name;  
 };  
}

manager.cpp

//  
// Created by xqula on 24.06.24.  
//  
#include "manager.h"  
#include "task/taskbase.h"  
#include "utils/random.h"  
department::Manager::Manager(QObject \**parent*):  
 Worker("../log.txt", *parent*),  
 m\_settingDepartment(std::make\_unique<SettingDepartment>()),  
 m\_timer(std::make\_unique<QTimer>(nullptr)),  
 m\_countTime(0)  
{  
 m\_settingDepartment->m\_maxDesigners = 4;  
 m\_settingDepartment->m\_executionTime = 10;  
 m\_settingDepartment->m\_maxTasksPriority = 2;  
 m\_settingDepartment->m\_maxTasksWeak = 8;  
 for(int i = 0; i < m\_settingDepartment->m\_maxTasksPriority; ++i) {  
 m\_orderTasksPriority.emplace\_back(true, "PRIORITY " +std::to\_string(i));  
 }  
 for(int i = 0; i < m\_settingDepartment->m\_maxTasksWeak; ++i) {  
 m\_orderTasksWeak.emplace\_back(false, "WEAK " +std::to\_string(i));  
 }  
 for(int i = 0; i < m\_settingDepartment->m\_maxDesigners; ++i) {  
 m\_designers.push\_back(std::make\_unique<Designer>(std::to\_string(i + 1), nullptr));  
 **connect**(m\_designers.back().get(), &Designer::finishTask, this, [this, i](const task::TaskBase &*task*){  
 if(i == m\_settingDepartment->m\_maxDesigners - 1) {  
 return;  
 }  
 if(*task*.get\_priority()) {  
 m\_queueTasks.push\_front({*task*, i});  
 }else {  
 m\_queueTasks.push\_back({*task*, i});  
 }  
 });  
 }  
 m\_timer->start(200);  
 **connect**(m\_timer.get(), &QTimer::timeout, this, [this](){  
 this->m\_countTime += 1;  
 if(this->m\_countTime == this->m\_settingDepartment->m\_executionTime) {  
 m\_countTime = 0;  
 if(m\_orderTasksPriority.empty() && m\_orderTasksWeak.empty()) {  
 write("Manager: No tasks");  
 return;  
 }  
 if(m\_orderTasksPriority.empty()) {  
 this->m\_designers.front()->setTask(m\_orderTasksWeak.back());  
 m\_orderTasksWeak.pop\_back();  
 return;  
 }  
 if(m\_orderTasksWeak.empty()) {  
 this->m\_designers.front()->setTask(m\_orderTasksPriority.back());  
 m\_orderTasksPriority.pop\_back();  
 return;  
 }  
 if(utils::Random::**random**(0,1)) {  
 write("Manager: Start new task designer 1 " + m\_orderTasksPriority.back().get\_name());  
 this->m\_designers.front()->setTask(m\_orderTasksPriority.back());  
 m\_orderTasksPriority.pop\_back();  
 }else {  
 write("Manager: Start new task designer 1 " + m\_orderTasksWeak.back().get\_name());  
 this->m\_designers.front()->setTask(m\_orderTasksWeak.back());  
 m\_orderTasksWeak.pop\_back();  
 }  
 }  
 if(not m\_queueTasks.empty()) {  
 write("Manager: Start new task designer " + std::to\_string(m\_queueTasks.front().indexDesigner + 2) + " " + m\_queueTasks.front().m\_task.get\_name());  
 m\_queueTasks.front().m\_task.reset\_execution\_time();  
 m\_queueTasks.front().m\_task.set\_status(task::TASK\_STATUS::CREATED);  
 m\_designers[m\_queueTasks.front().indexDesigner + 1]->setTask(m\_queueTasks.front().m\_task);  
 m\_queueTasks.pop\_front();  
 }  
 });  
}

manager.h

//  
// Created by xqula on 24.06.24.  
//  
#pragma once  
#include "worker.h"  
#include "department/designer.h"  
#include "deque"  
#include "vector"  
#include "memory"  
#include "QTimer"  
namespace department {  
 struct SettingDepartment {  
 int m\_maxDesigners;  
 int m\_executionTime;  
 int m\_maxTasksPriority;  
 int m\_maxTasksWeak;  
 };  
  
 struct TaskData {  
 task::TaskBase m\_task;  
 int indexDesigner;  
 };  
  
 class Manager : public Worker{  
 Q\_OBJECT  
 public:  
 explicit Manager(QObject \**parent* = nullptr);  
 ~Manager() override = default;  
 private:  
 std::vector<std::unique\_ptr<Designer>> m\_designers;  
 std::unique\_ptr<SettingDepartment> m\_settingDepartment;  
 std::unique\_ptr<QTimer> m\_timer;  
 int m\_countTime;  
 std::deque<TaskData> m\_queueTasks;  
 std::vector<task::TaskBase> m\_orderTasksPriority;  
 std::vector<task::TaskBase> m\_orderTasksWeak;  
 };  
}

worker.cpp

//  
// Created by xqula on 24.06.24.  
//  
#include "worker.h"  
Worker::Worker(std::string *name\_file*, QObject \**parent*):  
 QObject(*parent*),  
 m\_writer(std::make\_unique<utils::Writer>(std::move(*name\_file*)))  
{  
}  
auto Worker::write(const std::string\_view *txt*) const -> void {  
 m\_writer->write(*txt*);  
}

worker.h

//  
// Created by xqula on 24.06.24.  
//  
#ifndef WORKER\_H  
#define WORKER\_H  
#include "utils/writer.h"  
#include "memory"  
#include "string"  
#include "QObject"  
class Worker : public QObject {  
 Q\_OBJECT  
public:  
 explicit Worker(std::string *name\_file* ,QObject \**parent* = nullptr);  
 ~Worker() override = default;  
 auto write(std::string\_view *txt*) const -> void;  
private:  
 std::unique\_ptr<utils::Writer> m\_writer;  
};  
  
  
  
#endif //WORKER\_H

taskbase.cpp

//  
// Created by xqula on 24.06.24.  
//  
#include "taskbase.h"  
task::TaskBase::TaskBase(const bool *priority*, std::string *name*) :  
 m\_priority(*priority*),  
 m\_status(TASK\_STATUS::CREATED),  
 m\_executionTime(0),  
 m\_name(std::move(*name*))  
{  
}  
task::TaskBase::~TaskBase() = default;  
  
auto task::TaskBase::get\_priority() const -> bool {  
 return m\_priority;  
}  
auto task::TaskBase::get\_status() const -> TASK\_STATUS {  
 return m\_status;  
}  
auto task::TaskBase::get\_executionTime() const -> int {  
 return m\_executionTime;  
}  
auto task::TaskBase::get\_name() const -> std::string {  
 return m\_name;  
}  
auto task::TaskBase::set\_status(const TASK\_STATUS *status*) -> void {  
 m\_status = *status*;  
}  
auto task::TaskBase::up\_execution\_time() -> void {  
 ++m\_executionTime;  
}  
auto task::TaskBase::reset\_execution\_time() -> void {  
 m\_executionTime = 0;  
}

taskbase.h

//  
// Created by xqula on 24.06.24.  
//  
#pragma once  
#include "string"  
namespace task {  
 enum class TASK\_STATUS {  
 CREATED,  
 RUNNING,  
 FINISHED  
 };  
 class TaskBase {  
 public:  
 explicit TaskBase(bool *priority*, std::string *name*);  
 virtual ~TaskBase();  
 [[nodiscard]] auto get\_priority() const -> bool;  
 [[nodiscard]] auto get\_status() const -> TASK\_STATUS;  
 [[nodiscard]] auto get\_executionTime() const -> int;  
 [[nodiscard]] auto get\_name() const -> std::string;  
 auto set\_status(TASK\_STATUS *status*) -> void;  
 auto up\_execution\_time() -> void;  
 auto reset\_execution\_time() -> void;  
 private:  
 bool m\_priority;  
 TASK\_STATUS m\_status;  
 int m\_executionTime;  
 std::string m\_name;  
 };  
}

random.cpp

//  
// Created by xqula on 24.06.24.  
//  
#include "random.h"  
#include "random"  
auto utils::Random::**random**(const int *min*, const int *max*) -> int {  
 std::random\_device rd; // non-deterministic generator  
 std::mt19937 gen(rd()); // to seed mersenne twister.  
 std::uniform\_int\_distribution<> dist(*min*,*max*); // define the range  
 return dist(gen); // generate numbers  
}

random.h

//  
// Created by xqula on 24.06.24.  
//  
#pragma once  
namespace utils {  
 class Random {  
 public:  
 static auto **random**(int *min*, int *max*) -> int;  
 };  
}

writer.cpp

//  
// Created by xqula on 24.06.24.  
//  
#include "writer.h"  
#include <fstream>  
#include <utility>  
utils::Writer::Writer(std::string *name*) : m\_name(std::move(*name*)) {}  
auto utils::Writer::write(const std::string\_view *txt*) const -> void {  
 std::ofstream out;  
 out.open(m\_name, std::ios\_base::**app**);  
 if (out.is\_open())  
 {  
 out << *txt* << std::endl;  
 }else {  
 throw std::runtime\_error("Can't open file" + m\_name);  
 }  
 out.close();  
}

writer.h

//  
// Created by xqula on 24.06.24.  
//  
#pragma once  
#include "string"  
namespace utils {  
 class Writer {  
 public:  
 explicit Writer(std::string *name*);  
 ~Writer() = default;  
 auto write(std::string\_view *txt*) const -> void;  
  
 private:  
 std::string m\_name;  
 };  
}

**заключение**

В результате выполнения курсовой работы были разработаны 3 программы, основанные на принципах объектно-ориентированного подхода. Были закреплены навыки работы с файлами, Qt, интерфейсами, абстрактными классами. Использовались принципы инкапсуляции, наследования, полиморфизма, агрегации. Получен опыт в разработке приложений, моделирующих непостоянные процессы.

**список использованных источников**

1. Поздняков, Рыбин – Дискретная математика (дата обращения: 24.06.2024)
2. Moodle – источник заданий URL: vec.etu.ru/moodle (дата обращения 24.06.2024)
3. Документация по языку C++ URL: https://learn.microsoft.com/ru-ru/dotnet/csharp/ (дата обращения: 24.05.2024)
4. Статья с информацией о UML-диаграммах классах URL: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-aggregation-vs-composition/
5. Документация по UML-диаграммам URL: https://www.uml-diagrams.org/