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

Курсовая работа направлена на создание приложений на основе объектно-ориентированного подхода на языке C++. 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

1.1. Задание

Вариант 12

Разработать программу для представления спортивных и гоночных автомобилей.

1.2. Теоретические сведения

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

1.3. 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.

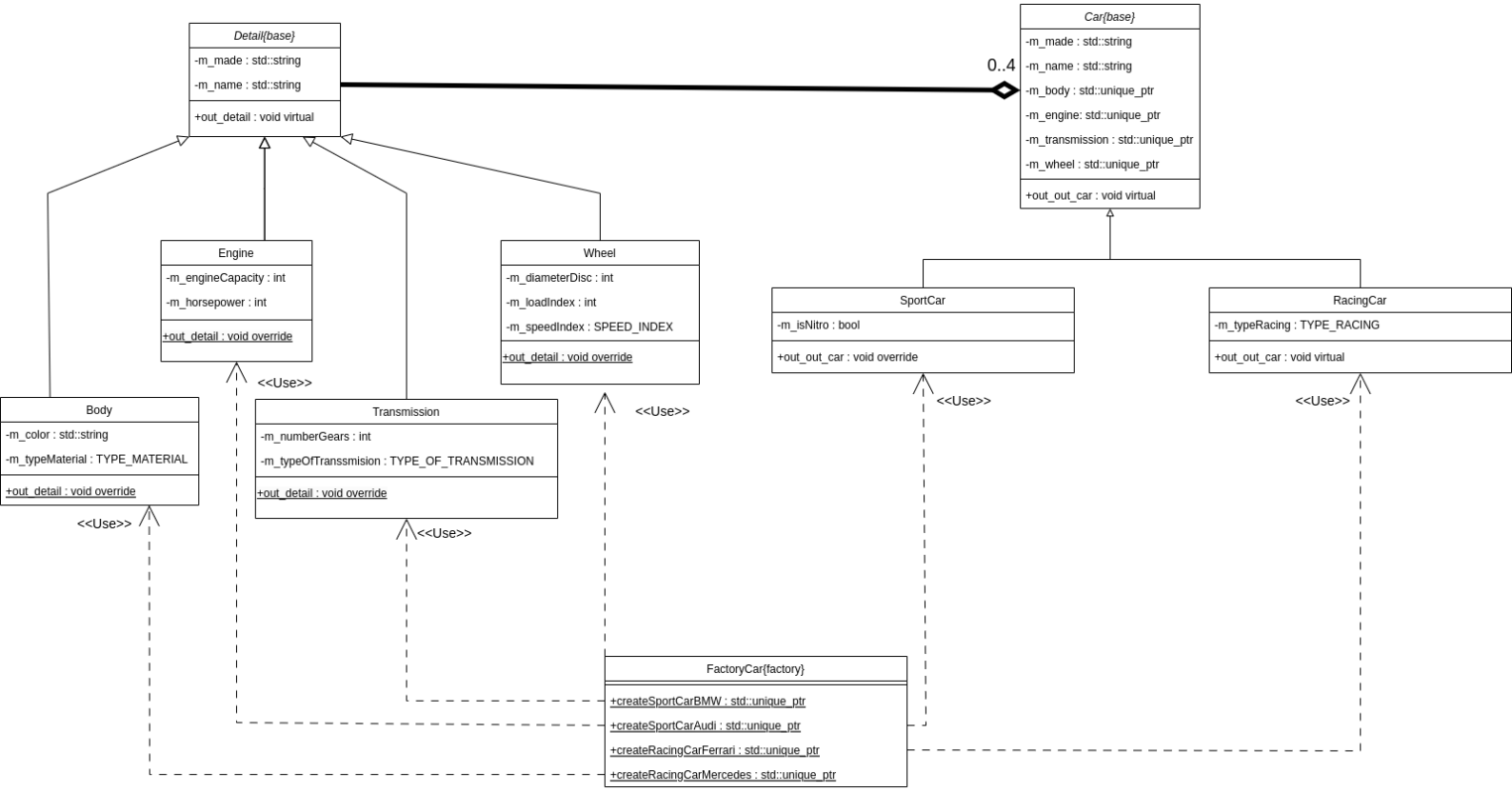

Рисунок 1.3.1. UML диаграмма классов

1.4. 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

Методы	Возвращаемый тип	Модификатор доступа	Входные параметры	Назначение
				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

1.5. Руководство пользователя

1.5.1. Назначение программы

Представление гоночных и спортивных машин.

1.5.2. Условие выполнения программы:

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

1.5.3. Выполнение программы:

1) Пользователь запускает программу.
2) В консоли выводится разный типаж гоночных и спортивных автомобилей.
3)

1.6. Руководство программиста

1.6.1. Запуск программы:

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

1.6.2. Характеристика программы

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

1.6.3. Входные и выходные данные

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

1.7. Контрольный пример

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

```
/home/xqula/repo/source/task/cmake-build-debug/task19
----------------------
----------------------
Sport car:
Made: BMW
Name: AAA
**********************
Detail made: BMW
Detail name: body
Color: white
Type material: METAL
**********************
**********************
Detail made: BMW
Detail name: engine
Engine capacity: 1000
Horsepower: 300
**********************
**********************
Detail made: BMW
Detail name: transmission
Number of gears: 5
Type of transmission: AUTOMATIC
**********************
**********************
Detail made: BMW
Detail name: wheel
Diameter disc: 18
Load index: 100
Speed index: 150
**********************
There is nitro: YES
----------------------
----------------------
----------------------
----------------------
```

```
----------------------
----------------------
Sport car:
Made: Audi
Name: XXX
**********************
Detail made: BMW
Detail name: body
Color: black
Type material: ALUMINUM
**********************
**********************
Detail made: BMW
Detail name: engine
Engine capacity: 1200
Horsepower: 310
**********************
**********************
Detail made: BMW
Detail name: transmission
Number of gears: 8
Type of transmission: ROBOT
**********************
**********************
Detail made: BMW
Detail name: wheel
Diameter disc: 21
Load index: 120
Speed index: 160
**********************
There is nitro: NO
----------------------
----------------------
```

```
----------------------
----------------------
Racing car:
Made: Ferrari
Name: YYY
**********************
Detail made: Ferrari
Detail name: body
Color: red
Type material: CARBON
**********************
**********************
Detail made: Ferrari
Detail name: engine
Engine capacity: 1200
Horsepower: 310
**********************
**********************
Detail made: Ferrari
Detail name: transmission
Number of gears: 12
Type of transmission: MAN
**********************
**********************
Detail made: BMW
Detail name: wheel
Diameter disc: 23
Load index: 180
Speed index: 240
**********************
type of racing car: F1
----------------------
----------------------
```

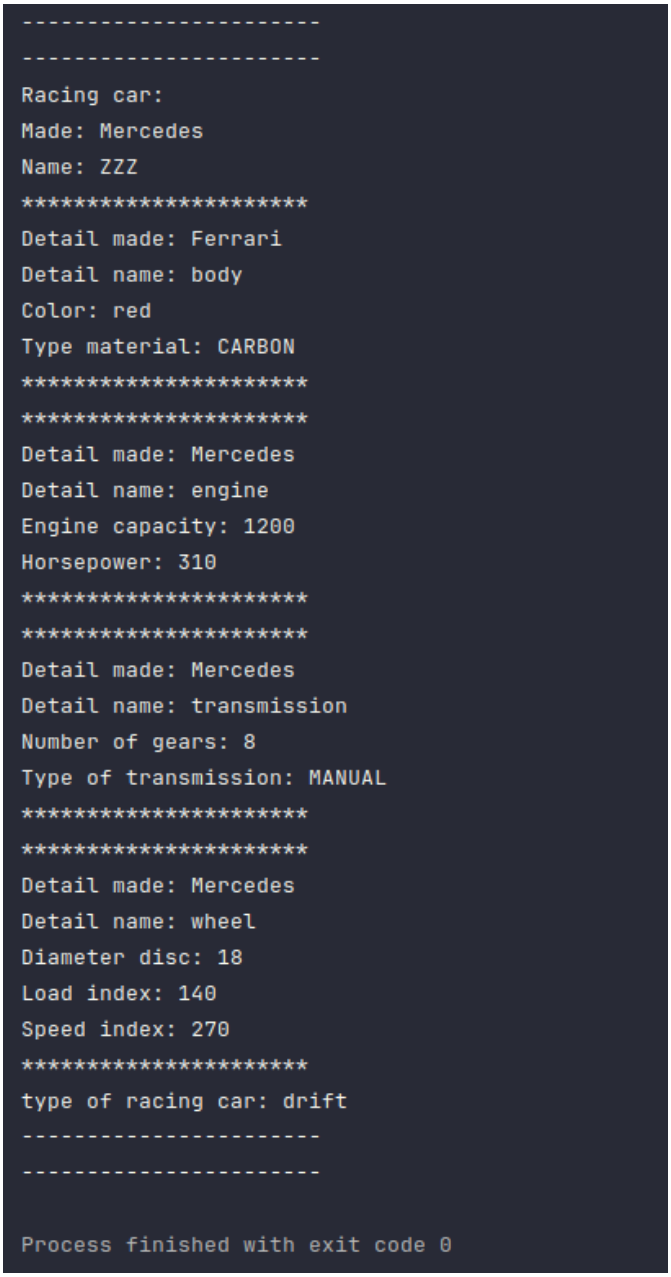

Рисунок 1.7.1 Работа программы

1.8. 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"
```

```cpp
#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

```cpp
//
// 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

```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:

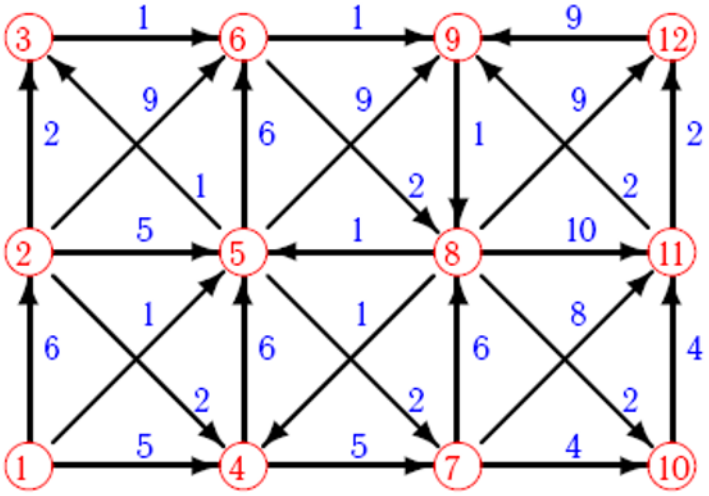

Рисунок 2.1.1 Граф для контрольного примера

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.

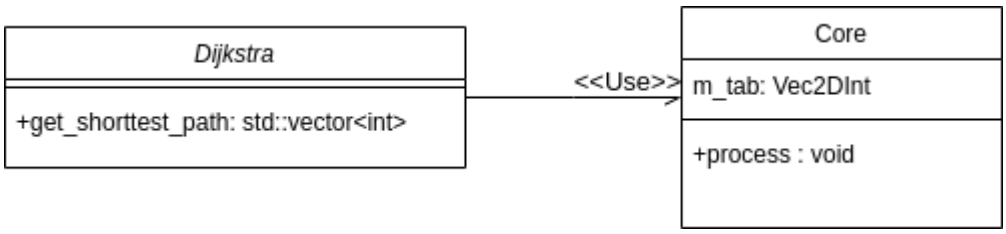

Рисунок 2.3.1 UML Диаграмма классов

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

2.5.1 Назначение программы

Данная программа позволяет найти кратчайщий путь в графе.

2.5.2 Условие выполнения программы:

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

2.5.3 Выполнение программы:

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

2.6 Руководство программиста

2.6.1 Запуск программы:

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

2.6.2 Характеристика **программы**

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

2.6.3 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.

2.7 Контрольный пример

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

```
/home/xqula/Desktop/learn/oop/src/curs/task2/cmake-build-debug/curs2
GRAPH:
----------------------
INF 6   INF 5   1   INF INF INF INF INF INF INF
INF INF 2   2   5   9   INF INF INF INF INF INF
INF INF INF INF INF 1   INF INF INF INF INF INF
INF INF INF INF 6   INF 5   INF INF INF INF INF
INF INF 1   INF INF 6   2   INF 9   INF INF INF
INF INF INF INF INF INF INF 2   1   INF INF INF
INF INF INF INF INF INF INF 6   INF 4   8   INF
INF INF INF 1   1   INF INF INF INF 2   10   9
INF INF INF INF INF INF INF 1   INF INF INF INF
INF INF INF INF INF INF INF INF INF INF 4   INF
INF INF INF INF INF INF INF INF 9   INF INF 2
INF INF INF INF INF INF INF INF 9   INF INF INF
----------------------
Enter start: 1
Enter end: 12
GRAPH:
----------------------
INF 6   INF 5   1   INF INF INF INF INF INF INF
INF INF 2   2   5   9   INF INF INF INF INF INF
INF INF INF INF INF 1   INF INF INF INF INF INF
INF INF INF INF 6   INF 5   INF INF INF INF INF
INF INF 1   INF INF 6   2   INF 9   INF INF INF
INF INF INF INF INF INF INF 2   1   INF INF INF
INF INF INF INF INF INF INF 6   INF 4   8   INF
INF INF INF 1   1   INF INF INF INF 2   10   9
INF INF INF INF INF INF INF 1   INF INF INF INF
INF INF INF INF INF INF INF INF INF INF 4   INF
INF INF INF INF INF INF INF INF 9   INF INF 2
INF INF INF INF INF INF INF INF 9   INF INF INF
----------------------
PATH:
----------------------
1 5 7 11 12
----------------------
----------------------
Sum path: 13
----------------------

Process finished with exit code 0
```

Рисунок 2.7.1 Работа программы

2.8 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

3.1 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.

3.2 Теоретические сведения

Имитационное моделирование заключается в создании процессов близких к реальным.

3.3 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.

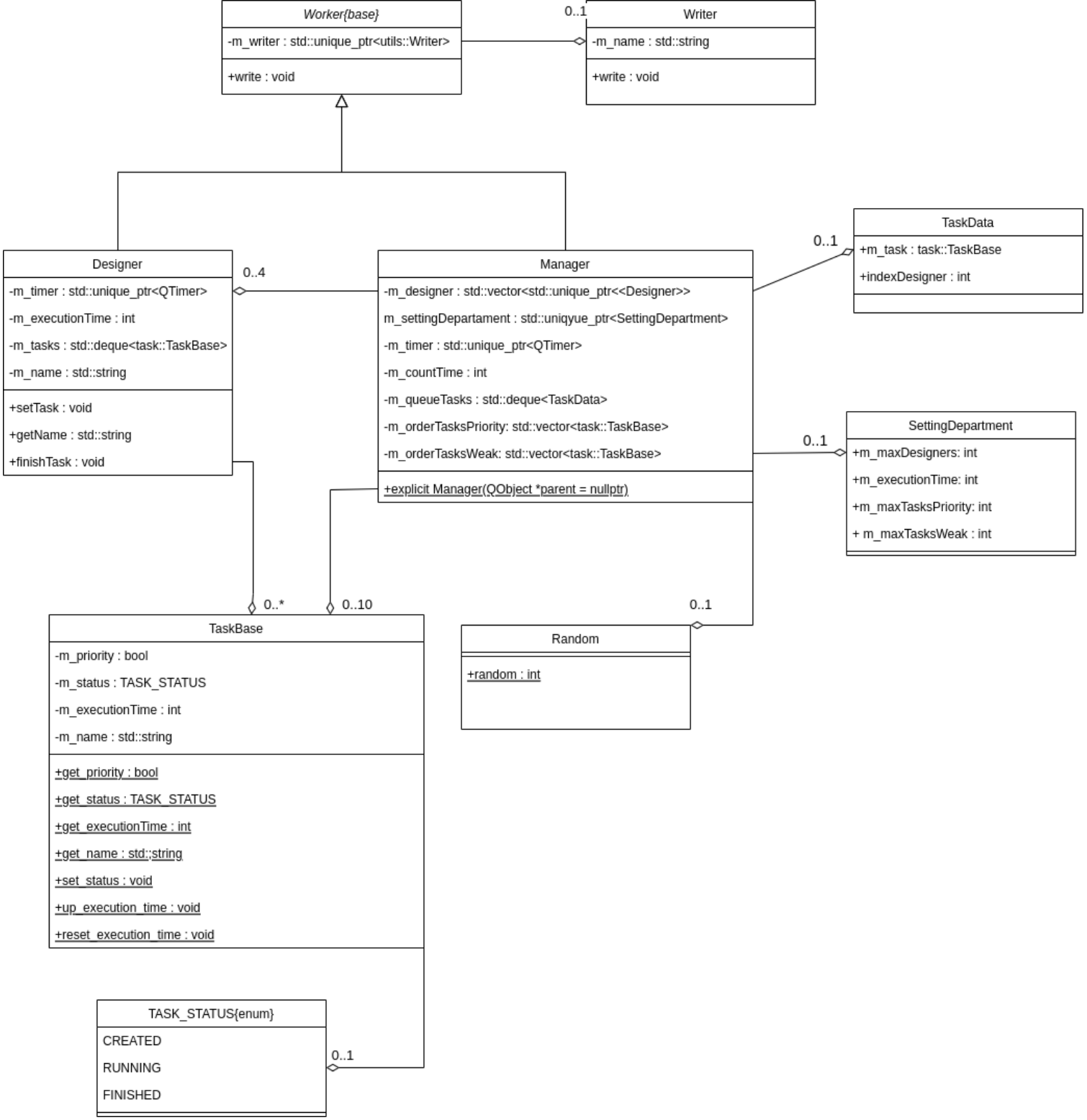

Рисунок 3.3.1 UML Диаграмма классов

3.4 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

3.5 Алгоритм

3.5.1 Создаются с помощью вихря Мейсена рандомно количество задач обычных, а срочных 2, в рандомно порядке.

3.5.2 Задачи выдаются менеджеру.

3.5.3 По определённому истечению времени выдается задача первому разработчику

3.5.4 Первый разработчик начинает работу над задачей

3.5.5 Если прилетает срочная задача, обычную откладывает и начинает со срочной.

3.5.6 Если прилетает еще одна задача, она становится второй.

3.5.7 Как только он заканчивает с задачей он отдает менеджеру и говорит, что закончил и записывается в файл.

3.5.8 Менеджер отдает следующему работнику и так по всем работникам(4).

3.6 Руководство пользователя

3.6.1 Назначение программы

Данная программа позволяет моделировать работу проектного отдела.

3.6.2 Условие выполнения программы:

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

3.6.3 Выполнение программы:

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

3.7 Руководство программиста

3.7.1 Запуск программы:

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

3.7.2 Характеристика программы

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

3.7.3 Входные и выходные данные

Выходные данные записываются в лог файл log.txt.

3.8 Контрольный пример

manager.h random.cpp random.h log.txt

```
Manager: Start new task designer 1 WEAK 7
Designer 1: Start new task WEAK 7
Designer 1: Finish task WEAK 7
Manager: Start new task designer 2 WEAK 7
Designer 2: Start new task WEAK 7
Designer 2: Finish task WEAK 7
Manager: Start new task designer 1 PRIORITY 1
Designer 1: Start new task PRIORITY 1
Manager: Start new task designer 3 WEAK 7
Designer 3: Start new task WEAK 7
Designer 1: Finish task PRIORITY 1
Designer 3: Finish task WEAK 7
Manager: Start new task designer 2 PRIORITY 1
Designer 2: Start new task PRIORITY 1
Manager: Start new task designer 4 WEAK 7
Designer 4: Start new task WEAK 7
Designer 2: Finish task PRIORITY 1
Manager: Start new task designer 1 WEAK 6
Designer 1: Start new task WEAK 6
Manager: Start new task designer 3 PRIORITY 1
Designer 3: Start new task PRIORITY 1
Designer 4: Finish task WEAK 7
Designer 1: Finish task WEAK 6
Designer 3: Finish task PRIORITY 1
Manager: Start new task designer 4 PRIORITY 1
Designer 4: Start new task PRIORITY 1
Manager: Start new task designer 2 WEAK 6
Designer 2: Start new task WEAK 6
Designer 4: Finish task PRIORITY 1
Manager: Start new task designer 1 WEAK 5
Designer 1: Start new task WEAK 5
Designer 2: Finish task WEAK 6
Manager: Start new task designer 3 WEAK 6
Designer 3: Start new task WEAK 6
Designer 1: Finish task WEAK 5
Manager: Start new task designer 2 WEAK 5
Designer 2: Start new task WEAK 5
Designer 3: Finish task WEAK 6
Manager: Start new task designer 4 WEAK 6
Designer 4: Start new task WEAK 6
Designer 2: Finish task WEAK 5
Manager: Start new task designer 1 PRIORITY 0
Designer 1: Start new task PRIORITY 0
Manager: Start new task designer 3 WEAK 5
Designer 3: Start new task WEAK 5
Designer 4: Finish task WEAK 6
Designer 1: Finish task PRIORITY 0
Designer 3: Finish task WEAK 5
Manager: Start new task designer 2 PRIORITY 0
Designer 2: Start new task PRIORITY 0
Manager: Start new task designer 4 WEAK 5
Designer 4: Start new task WEAK 5
Designer 2: Finish task PRIORITY 0
Designer 1: Start new task WEAK 4
Designer 4: Finish task WEAK 5
Manager: Start new task designer 3 PRIORITY 0
```

Рисунок 3.7.1 Работа программы

3.9 Обработка информации вывода

3.9.1 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.

3.10 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/