

ТРПО Теория + код

Оглавление

[1. SOLID](#)

[1.1. Single Responsibility Principle \(SRP\)](#)

[1.2. Open Closed Principle \(OCP\)](#)

[1.3. Liskov Substitution Principle \(LSP\)](#)

[1.4. Interface Segregation Principle \(ISP\)](#)

[1.5. Dependency Inversion Principle \(DIP\)](#)

[2. Паттерны](#)

[2.1. Singleton](#)

[2.2. Factory Method](#)

[2.3. Abstract Factory](#)

[2.4. Builder](#)

[2.5. Prototype](#)

[2.6. Observer](#)

[2.7. Strategy](#)

[2.8. Template Method](#)

[2.9. Adapter](#)

[3. MVC](#)

[3.1. MVC \(Model-View-Controller\)](#)

[4. Отношения между классами. UML](#)

[4.1. Зависимость](#)

[4.2.1. Агрегация](#)

[4.2.2. Композиция](#)

[4.2.3. Обобщение](#)

[4.3. Реализация](#)

[5. File Monitoring](#)

[Лабораторная работа № 1. Наблюдение за файлами](#)

[6. Code Generator](#)

[Лабораторная работа № 2. Генератор кода](#)

[7. Storage Observer](#)

[Лабораторная работа № 3.1. Файловый обозреватель](#)

[Лабораторная работа № 3.2. Графическое приложение](#)

1. SOLID

1.1. Single Responsibility Principle (SRP)

Принцип Единой Ответственности - это принцип проектирования, который гласит, что каждая программная сущность должна выполнять только одну задачу

Применение принципа Единой Ответственности:

- Упрощение поддержки кода - если класс выполняет только одну задачу, его легче понять, изменить и тестировать
- Уменьшение связанности - классы меньше зависят от других классов, что делает систему более модульной
- Повышение переиспользуемости - классы, выполняющие одну задачу, проще использовать в других частях программы или проекта

Преимущества принципа Единой Ответственности:

- Читаемость - код становится проще для понимания
- Тестируемость - код легче тестируется
- Гибкость - изменения в одной части системы меньше влияют на другие части
- Упрощение отладки - проще найти источник ошибки, если класс выполняет только одну задачу

Недостатки принципа Единой Ответственности:

- Увеличение количества классов - возрастает количество классов в проекте
- Сложность управления зависимостями - может потребоваться больше усилий для управления взаимодействиями между классами

```
#include <iostream>

using namespace std;

class Report {
public:
    void generateReport() {
        cout << "Отчёт сформирован" << endl;
    }
};

class ReportSender {
public:
    void sendReport(Report* report, const string& email) {
        if (report) {
            cout << "Отчёт отправлен на почту " << email << endl;
        } else {
            cout << "Ошибка: не удалось отправить отчёт на почту " << email <<
endl;
        }
    }
};

class ReportSaver {
```

```

public:
    void saveReportToTxtFile(Report* report, const string& fileName) {
        if (report) {
            cout << "Отчёт сохранён в файл " << fileName << ".txt" << endl;
        } else {
            cout << "Ошибка: не удалось сохранить отчёт в файл " << fileName <<
            ".txt" << endl;
        }
    }

    void saveReportToDocxFile(Report* report, const string& fileName) {
        if (report) {
            cout << "Отчёт сохранён в файл " << fileName << ".docx" << endl;
        } else {
            cout << "Ошибка: не удалось созранить отчёт в файл " << fileName <<
            ".docx" << endl;
        }
    }
};

class ReportRemover {
public:
    void removeReport(Report* report) {
        if (report) {
            cout << "Отчёт удалён" << endl;
        } else {
            cout << "Ошибка: не удалось удалить отчёт" << endl;
        }
    }
};

int main() {
    setlocale(LC_ALL, "Rus");

    Report report;
    report.generateReport();

    ReportSaver reportSaver;
    reportSaver.saveReportToTxtFile(&report, "Report 2025");
    reportSaver.saveReportToDocxFile(&report, "Report 2025");

    ReportSender reportSender;
    reportSender.sendReport(&report, "Homyakov.Denis2021@yandex.ru");

    ReportRemover reportRemover;
    reportRemover.removeReport(&report);

    return 0;
}

```

```
}
```

1.2. Open Closed Principle (OCP)

Принцип Открытости / Закрытости - это принцип проектирования, который гласит, что программные сущности должны быть открыты для расширений, но закрыты для изменений

Применение принципа Открытости / Закрытости:

- Упрощение поддержки - минимизация изменений в существующем коде снижает риск внесения ошибок
- Гибкость - легко добавлять новую функциональность, не нарушая работу старой
- Масштабируемость - система становится более адаптируемой к изменениям

Преимущества принципа Открытости / Закрытости:

- Стабильность - существующий код меньше подвержен изменениям, что делает его более надёжным
- Расширяемость - новые функции добавляются без переписывания старого кода
- Тестируемость - меньше изменений в существующем коде - меньше необходимости в повторном тестировании

Недостатки принципа Открытости / Закрытости:

- Сложность проектирования - требует больше усилий для создания гибкой архитектуры
- Избыточность - приходится создавать большое количество классов или интерфейсов

```
#include <iostream>
#include <vector>

using namespace std;

class Shape {
public:
    virtual ~Shape() = default;

    virtual double area() const = 0;
};

class Circle : public Shape {
private:
    double radius;
```

```

public:
    Circle(double radius) : radius(radius) {}

    double area() const override {
        return 3.14 * radius * radius;
    }
};

class Square : public Shape {
private:
    double side;

public:
    Square(double side) : side(side) {}

    double area() const override {
        return side * side;
    }
};

class AreaCalculator {
public:
    double totalArea(const vector<Shape*>& shapes) {
        double totalArea = 0.0;

        for (const auto& shape : shapes) {
            totalArea += shape->area();
        }

        return totalArea;
    }
};

int main() {
    setlocale(LC_ALL, "Rus");

    Circle circle(5);
    Square square(6);

    vector<Shape*> shapes = { &circle, &square };
    AreaCalculator areaCalculator;
    cout << "Общая площадь: " << areaCalculator.totalArea(shapes);

    return 0;
}

```

1.3. Liskov Substitution Principle (LSP)

Принцип Подстановки Барбары Лисков - это принцип проектирования, который гласит, что производные классы должны дополнять, а не заменять поведение родительского класса

Применение принципа Подстановки Барбары Лисков:

- Обеспечение совместимости - производные классы должны работать корректно в любом контексте, где используется базовый класс
- Упрощение тестирования - тесты для базового класса будут работать и для производных классов
- Повышение надёжности - код становится более предсказуемым и устойчивым к ошибкам

Преимущества принципа Подстановки Барбары Лисков:

- Совместимость - производные классы могут использоваться вместо базовых классов без каких-либо проблем
- Модульность - код становится более модульным и легко расширяемым
- Упрощение поддержки - легче добавлять новые функции, не нарушая существующую логику

Недостатки принципа Подстановки Барбары Лисков:

- Сложность проектирования - требует тщательное проектирование иерархии классов
- Ограничения - иногда приходится жертвовать гибкостью

```
#include <iostream>

using namespace std;

class Human {
public:
    virtual void eat() const {
        cout << "Человек ест" << endl;
    }

    virtual void think() const {
        cout << "Человек думает" << endl;
    }

    virtual void work() const {
        cout << "Человек работает" << endl;
    }
};

class Engineer : public Human {
```

```

public:
    void work() const override {
        cout << "Инженер работает" << endl;
    }
};

class Teacher : public Human {
public:
    void work() const override {
        cout << "Учитель работает" << endl;
    }
};

void doWork(Human* human) {
    human->work();
}

int main() {
    setlocale(LC_ALL, "Rus");

    Engineer engineer;
    doWork(&engineer);

    Teacher teacher;
    doWork(&teacher);

    return 0;
}

```

1.4. Interface Segregation Principle (ISP)

Принцип Разделения Интерфейса - это принцип проектирования, который гласит, что программные сущности не должны зависеть от интерфейсов, которые они не используют

Применение принципа Разделение Интерфейса:

- Упрощение кода - классы реализуют только те методы, которые им действительно нужны
- Уменьшение связанности - классы не зависят от методов, которые они не используют
- Гибкость - легче изменять и расширять систему, так как изменения в одном интерфейсе затрагивают только классы, использующие его

Преимущества принципа Разделения Интерфейса:

- Упрощение поддержки - меньше кода, а значит, меньше ошибок
- Гибкость - легче добавлять новые функции, не затрагивая существующий код

- Тестируемость - узкоспециализированные интерфейсы проще тестировать

Недостатки принципа Разделения Интерфейса:

- Увеличение количества интерфейсов - может привести к увеличению числа классов и интерфейсов в проекте
- Сложность проектирования - требует тщательное проектирование интерфейсов

```
#include <iostream>

using namespace std;

class IXeroxable {
public:
    virtual void xerox() const = 0;
};

class IPrintable {
public:
    virtual void print() const = 0;
};

class IScanning {
public:
    virtual void scan() const = 0;
};

class Xerox : public IXeroxable {
public:
    void xerox() const override {
        cout << "Копирую..." << endl;
    }
};

class Printer : public IPrintable {
public:
    void print() const override {
        cout << "Печатаю..." << endl;
    }
};

class Scanner : public IScanning {
public:
    void scan() const override {
        cout << "Сканирую..." << endl;
    }
};

class MultifunctionalDevice : public IXeroxable, public IPrintable, public
```



```

IScanning {
public:
    void xerox() const override {
        cout << "Копирую..." << endl;
    }

    void print() const override {
        cout << "Печатаю..." << endl;
    }

    void scan() const override {
        cout << "Сканирую..." << endl;
    }
};

int main() {
    setlocale(LC_ALL, "Rus");

    Xerox xerox;
    xerox.xerox();

    Printer printer;
    printer.print();

    Scanner scanner;
    scanner.scan();

    MultifunctionalDevice multifunctionalDevice;
    multifunctionalDevice.xerox();
    multifunctionalDevice.print();
    multifunctionalDevice.scan();

    return 0;
}

```

1.5. Dependency Inversion Principle (DIP)

Принцип Инверсии Зависимостей - это принцип проектирования, который гласит, что модули верхних уровней не должны зависеть от модулей нижних уровней. Все они должны зависеть от абстракций. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Применение принципа Инверсии Зависимостей:

- Уменьшение связанности - код становится менее зависимым от конкретных реализаций

- Тестируемость - упрощается тестирование, так как зависимости можно заменять на mock-объекты
- Масштабируемость - система становится более адаптируемой к изменениям

Преимущества принципа Инверсии Зависимостей:

- Гибкость - легко заменять реализации, не изменяя код верхнего уровня
- Упрощение тестирования - зависимости можно подменять на заглушки (mocks) для тестов
- Упрощение поддержки - изменения в деталях реализации не затрагивают код верхнего уровня

Недостатки принципа Инверсии Зависимостей:

- Сложность проектирования - требуется больше усилий для создания абстракций и управления зависимостями
- Увеличение количества кода - может потребоваться больше интерфейсов и классов

Определение 1: mock-объекты - это специальные объекты, которые используются в тестировании для имитации поведения реальных объектов

```
#include <iostream>

using namespace std;

// Абстракция
class INotification {
public:
    virtual ~INotification() = default;

    virtual void send(const string& message) const = 0;
};

// Модуль верхнего уровня
class NotificationService {
private:
    INotification* notification;

public:
    NotificationService(INotification* notification) :
        notification(notification) {}

    ~NotificationService() {
        delete notification;
    }
}
```

```

    void sendNotification(const string& message) {
        notification->send(message);
    }
};

// Модуль нижнего уровня
class EmailNotification : public INotification {
public:
    void send(const string& message) const override {
        cout << "Отправка сообщения по почте: " << message << endl;
    }
};

// Модуль нижнего уровня
class SMSNotification : public INotification {
public:
    void send(const string& message) const override {
        cout << "Отправка сообщения по СМС: " << message << endl;
    }
};

int main() {
    setlocale(LC_ALL, "Rus");

    INotification* emailNotification = new EmailNotification();
    NotificationService emailService(emailNotification);
    emailService.sendNotification("Привет через почту");

    INotification* smsNotification = new SMSNotification();
    NotificationService smsService(smsNotification);
    smsService.sendNotification("Привет через СМС");

    return 0;
}

```

2. Паттерны

2.1. Singleton

Паттерн Одиночка - это порождающий паттерн, который гарантирует, что у класса есть только один экземпляр, и предоставляет глобальную точку доступа к этому экземпляру

Использование паттерна Одиночка:

- Единственный экземпляр - когда в система должен существовать только один экземпляр класса (например, подключение к БД, логгер)

- Глобальный доступ - когда нужно предоставить глобальную точку доступа к этому экземпляру
- Контроль создания - когда нужно контролировать процесс создания экземпляра (например, ленивая инициализация)

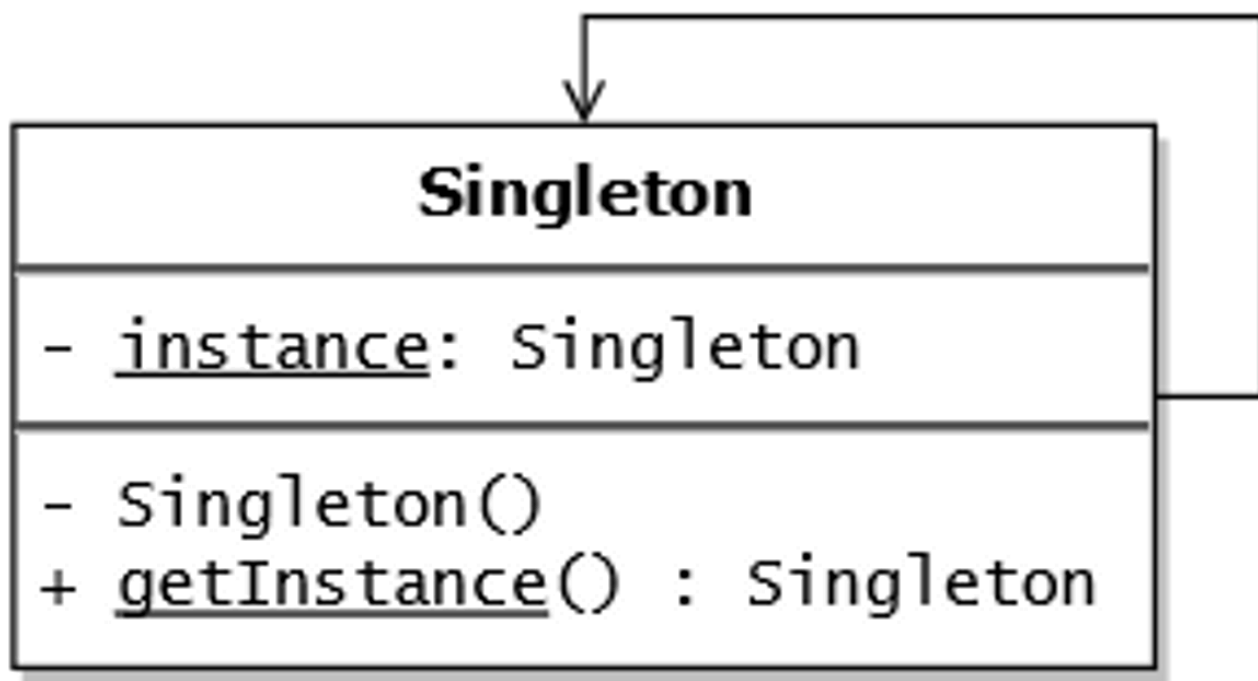
Преимущества паттерна Одиночка:

- Контроль над экземпляром - гарантирует, что у класса будет только один экземпляр
- Глобальный доступ - упрощает доступ к экземпляру из любой части программы
- Ленивая инициализация - экземпляр создаётся только при первом обращении, что может экономить ресурсы

Недостатки паттерна Одиночка:

- Глобальное состояние - может привести к нежелательным побочным эффектам, так как экземпляр доступен глобально
- Тестирование - усложняет тестирование, так как экземпляр является глобальным и может сохранять состояние между тестами
- Нарушение принципа единой ответственности - класс берёт на себя две обязанности: управление своим экземпляром и выполнение своей основной задачи

Рассмотрим UML-диаграмму паттерна Одиночка:



Пояснение к диаграмме:

- Singleton - класс, который содержит статический метод getInstance() для доступа к единственному экземпляру
- instance - статическое поле, которое хранит единственный экземпляр класса
- getInstance - статический метод, который возвращает экземпляр класса. Если экземпляра ещё не создан, он создаётся

```

#include <iostream>

using namespace std;

class Singleton {
public:
    // Удаляем конструктор копирования и оператор присваивания
    Singleton(const Singleton& other) = delete;
    Singleton& operator=(const Singleton& other) = delete;

    // Метод для получения экземпляра
    static Singleton& getInstance() {
        static Singleton instance;
        return instance;
    }

    void doSomething() {
        cout << "Какое-то действие" << endl;
    }

private:
    Singleton() {
        cout << "Одиночка создан!" << endl;
    }

    ~Singleton() {
        cout << "Одиночка удалён!" << endl;
    }
};

int main() {
    setlocale(LC_ALL, "Rus");

    // Получаем экземпляр Singleton
    Singleton& singleton = Singleton::getInstance();
    singleton.doSomething();

    // Попытка создать ещё один экземпляр приведёт к ошибке
    //Singleton anotherSingleton;

    return 0;
}

```

2.2. Factory Method

Паттерн Фабричный метод - это порождающий паттерн проектирования, который определяет интерфейс для создания объекта, но оставляет подклассам решение о

том, какой класс инстанцировать (создать экземпляр класса). Таким образом, он делегирует создание объектов подклассам

Применение паттерна Фабричный метод:

- Гибкость создания объектов - когда нужно создавать объекты, но точный тип объекта неизвестен до момента выполнения
- Изоляция кода - когда нужно отделить логику создания объекта от его использования
- Расширяемость - когда нужно легко добавлять новые типы объектов без изменения существующего кода

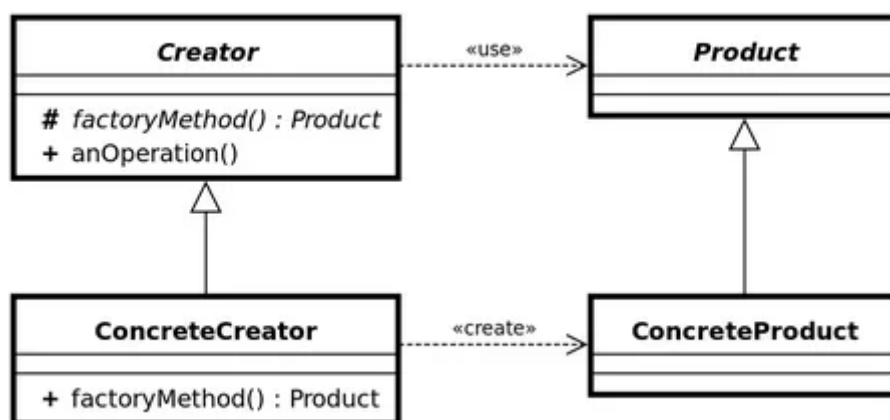
Преимущества паттерна Фабричный метод:

- Гибкость - позволяет подклассам выбирать тип создаваемого объекта
- Изоляция - логика создания объекта изолирована от клиентского кода
- Расширяемость - легко добавлять новые типы объектов, создавая новые подклассы

Недостатки паттерна Фабричный метод:

- Усложнение кода - увеличивает количество классов и объектов в системе
- Необходимость создания подклассов - для каждого нового типа объекта нужно создавать новый подкласс

Рассмотрим UML-диаграмму паттерна Фабричный метод:



Пояснение к диаграмме:

- Creator - абстрактный класс, который определяет фабричный метод factoryMethod() для создания объектов
- ConcreteCreator - конкретный класс, который реализует фабричный метод и создаёт конкретные объекты
- Product - интерфейс или абстрактный класс для создания объектов, которые создаются фабричным методом

- ConcreteProduct - конкретный класс, реализующий интерфейс Product

```
#include <iostream>

using namespace std;

// Продукт – Документ
class Document {
public:
    virtual ~Document() = default;

    virtual void open() const = 0;
    virtual void save() const = 0;
};

// Конкретный продукт – Текстовый документ
class TextDocument : public Document {
public:
    void open() const override {
        cout << "Текстовый документ открыт" << endl;
    }

    void save() const override {
        cout << "Текстовый документ сохранён" << endl;
    }
};

// Конкретный продукт – Табличный документ
class TabularDocument : public Document {
public:
    void open() const override {
        cout << "Табличный документ открыт" << endl;
    }

    void save() const override {
        cout << "Табличный документ сохранён" << endl;
    }
};

// Создатель – Создатель документов
class DocumentCreator {
public:
    virtual ~DocumentCreator() = default;

    void newDocument() const {
        Document* document = createDocument();
        document->open();
        delete document;
    }
};
```

```

    void saveDocument() const {
        Document* document = createDocument();
        document->save();
        delete document;
    }

protected:
    virtual Document* createDocument() const = 0;
};

// Конкретный создатель – Создатель текстовых документов
class TextDocumentCreator : public DocumentCreator {
protected:
    Document* createDocument() const override {
        return new TextDocument();
    }
};

// Конкретный создатель – Создатель табличных документов
class TabularDocumentCreator : public DocumentCreator {
protected:
    Document* createDocument() const override {
        return new TabularDocument();
    }
};

int main() {
    setlocale(LC_ALL, "Rus");

    DocumentCreator* documentCreator = new TextDocumentCreator();
    documentCreator->newDocument();
    documentCreator->saveDocument();
    delete documentCreator;

    documentCreator = new TabularDocumentCreator();
    documentCreator->newDocument();
    documentCreator->saveDocument();
    delete documentCreator;

    return 0;
}

```

2.3. Abstract Factory

Паттерн Абстрактная фабрика - это порождающий паттерн проектирования, который предоставляет интерфейс для создания семейств связанных или зависимых объектов

без указания их конкретных классов. Он позволяет создавать объекты, которые связаны между собой, но при этом изолирует клиентский код от конкретных классов

Применение паттерна Абстрактная фабрика:

- Создание семейств объектов - когда нужно создавать группы связанных объектов (например, элементы интерфейса - кнопки для приложения)
- Изоляция кода - когда нужно отделить логику создания объектов от их использования
- Согласованность объектов - когда важно, чтобы создаваемые объекты были совместимы друг с другом

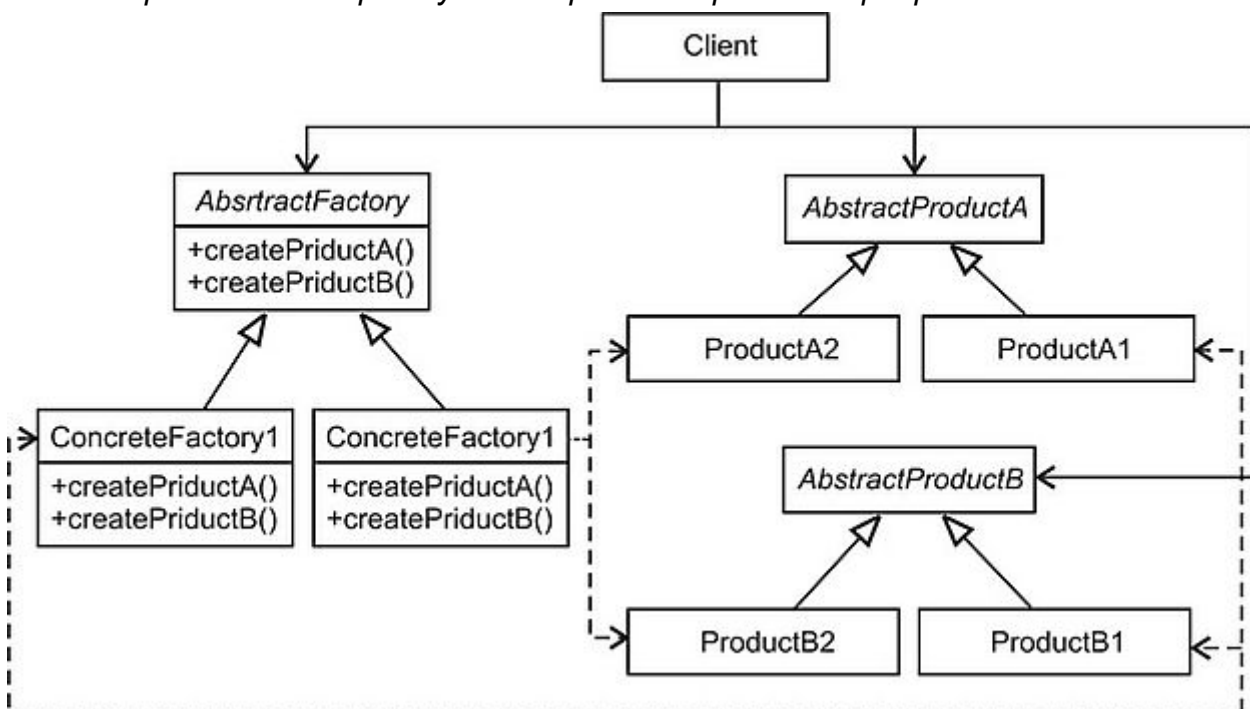
Преимущества паттерна Абстрактная фабрика:

- Гибкость - позволяет легко менять семейства создаваемых объектов
- Изоляция - логика создания объектов изолирована от клиентского кода
- Согласованность - гарантирует, что создаваемые

Недостатки паттерна Абстрактная фабрика:

- Сложность - увеличивает количество классов и объектов в системе
- Жёсткость - добавление новых типов может потребовать изменения интерфейса фабрики и всех её подклассов

Рассмотрим UML-диаграмму паттерна Абстрактная фабрика:



Пояснение к диаграмме:

- AbstractFactory - интерфейс или абстрактный класс, который определяет методы для создания продуктов

- ConcreteFactory - конкретный класс, который реализует методы для создания конкретных продуктов
- AbstractProduct - интерфейс или абстрактный класс для продуктов
- Product - конкретный класс, реализующий интерфейс AbstractProduct

```
#include <iostream>

using namespace std;

// Абстрактный продукт – Кнопка
class Button {
public:
    virtual ~Button() = default;

    virtual void render() const = 0;
};

// Конкретный продукт – Кнопка ОС Windows
class WindowsButton : public Button {
public:
    void render() const override {
        cout << "Кнопка ОС Windows" << endl;
    }
};

// Конкретный продукт – Кнопка ОС MacOS
class MacOSButton : public Button {
public:
    void render() const override {
        cout << "Кнопка ОС MacOS" << endl;
    }
};

// Абстрактный продукт – Чекбокс
class Checkbox {
public:
    virtual ~Checkbox() = default;

    virtual void render() const = 0;
};

// Конкретный продукт – Чекбокс ОС Windows
class WindowsCheckbox : public Checkbox {
public:
    void render() const override {
        cout << "Чекбокс ОС Windows" << endl;
    }
};
```

```

// Конкретный продукт – Чекбокс ОС MacOS
class MacOSCheckbox : public Checkbox {
public:
    void render() const override {
        cout << "Чекбокс ОС MacOS" << endl;
    }
};

// Абстрактная фабрика – Генератор элементов
class ElementGenerator {
public:
    virtual ~ElementGenerator() = default;

    virtual Button* createButton() const = 0;
    virtual Checkbox* createCheckbox() const = 0;
};

// Конкретная фабрика – Генератор элементов ОС Windows
class WindowsElementGenerator : public ElementGenerator {
public:
    Button* createButton() const override {
        return new WindowsButton();
    }

    Checkbox* createCheckbox() const override {
        return new WindowsCheckbox();
    }
};

// Конкретная фабрика – Генератор элементов ОС MacOS
class MacOSElementGenerator : public ElementGenerator {
public:
    Button* createButton() const override {
        return new MacOSButton();
    }

    Checkbox* createCheckbox() const override {
        return new MacOSCheckbox();
    }
};

int main() {
    setlocale(LC_ALL, "Rus");

    ElementGenerator* elementGenerator = new WindowsElementGenerator();
    Button* button = elementGenerator->createButton();
    Checkbox* checkbox = elementGenerator->createCheckbox();

    button->render();

```

```
checkbox->render();

delete elementGenerator;
delete button;
delete checkbox;

elementGenerator = new MacOSElementGenerator();
button = elementGenerator->createButton();
checkbox = elementGenerator->createCheckbox();

button->render();
checkbox->render();

delete elementGenerator;
delete button;
delete checkbox;

return 0;
}
```

2.4. Builder

Паттерн Строитель - это порождающий паттерн проектирования, который позволяет создавать сложные объекты пошагово (частями). Он отделяет процесс конструирования объекта от его представления, что позволяет использовать один и тот же процесс конструирования для создания различных представлений объекта

Определение 1: конструирование объекта - это процесс создания и настройки объекта, оно включает в себя:

- Инициализацию - создание экземпляра объекта
- Настройку - установку свойств объекта (например, через сеттеры и/или параметры конструктора)
- Сборку - пошаговое добавление частей или компонентов объекта

В паттерне Строитель конструирование выносится в отдельные класс (ConcreteBuilder), что позволяет:

- Изолировать сложную логику создания объекта
- Пошагово конфигурировать (настраивать) объект
- Создавать различные варианты объекта, используя один и тот же вариант конструирования

Определение 2: представление объекта - это то, как объект выглядит или как он используется после создания, оно включает в себя:

- Структуру объекта - какие данные и методы он содержит

- Поведение объекта - как объект взаимодействует с другими объектами
- Финальное состояние объекта - какие свойства и значения установлены после завершения конструирования

В паттерне Строитель представление объекта отделено от процесса его создания, что позволяет:

- Изменять процесс конструирования, не влияя на конечный объект
- Создавать разные представления объекта, используя один и тот же процесс конструирования

Использование паттерна Строитель:

- Создание сложных объектов - когда объект имеет много параметров и вариантов конфигураций
- Изоляция кода - когда нужно отделить логику создания объекта от представления
- Пошаговое создание - когда объект должен создаваться пошагово, возможно, с использованием разных конфигураций

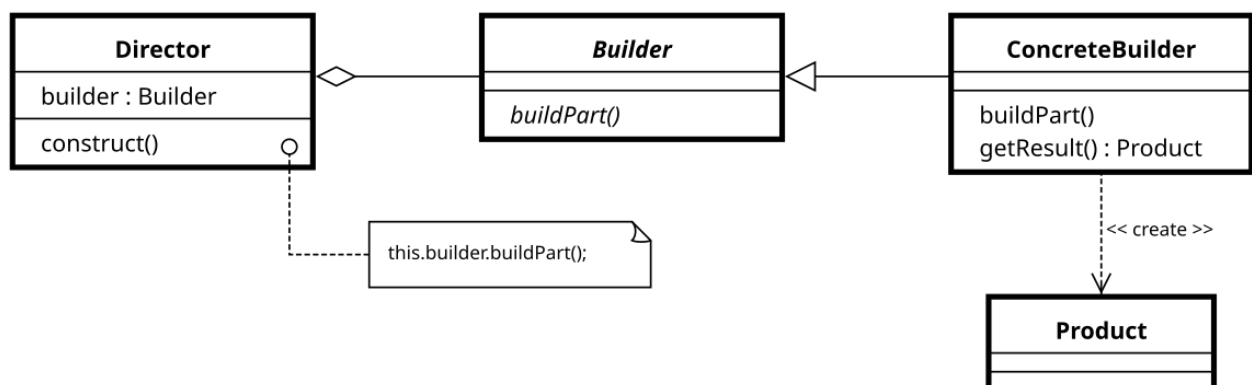
Преимущества паттерна Строитель:

- Гибкость - позволяет создавать различные конфигурации объекта, используя один и тот же процесс конструирования
- Изоляция - логика создания объекта изолирована от его представления, что упрощает поддержку и расширение
- Повторное использование - можно повторно использовать один и тот же процесс конструирования для создания различных объектов

Недостатки паттерна Строитель:

- Сложность - увеличивает сложность кода из-за введения дополнительных классов
- Избыточность - может быть избыточным для простых объектов, которые не требуют сложного процесса конструирования

Рассмотрим UML-диаграмму паттерна Строитель:



Пояснение к диаграмме:

- Director - управляет процессом конструирования, используя Builder
- Builder - интерфейс для создания частей объекта (конструирования)
- ConcreteBuilder - реализация интерфейса Builder, создаёт конкретные части объекта
- Product - конечный объект, который создаётся

```
#include <iostream>

using namespace std;

// Продукт
class House {
public:
    void setWalls(const string& walls) {
        this->walls = walls;
    }

    void setRoof(const string& roof) {
        this->roof = roof;
    }

    void setWindow(const string& window) {
        this->window = window;
    }

    void showInfo() const {
        cout << "Дом с параметрами:" << endl;
        cout << "Стены: " << walls << endl;
        cout << "Крыша: " << roof << endl;
        cout << "Окна: " << window << endl;
    }

private:
    string walls;
    string roof;
    string window;
};

// Строитель
class IHouseBuilder {
public:
    ~IHouseBuilder() = default;

    virtual void buildWalls() = 0;
    virtual void buildRoof() = 0;
    virtual void buildWindow() = 0;
```

```

    virtual House getResult() = 0;
};

// Конкретный строитель
class ConcreteHouseBuilder : public IHouseBuilder {
public:
    void buildWalls() override {
        house.setWalls("Белые стены");
    }

    void buildRoof() override {
        house.setRoof("Металлочерепица");
    }

    void buildWindow() override {
        house.setWindow("Пластиковые окна");
    }

    House getResult() override {
        return house;
    }

private:
    House house;
};

// Директор
class Director {
public:
    void setBuilder(IHouseBuilder* houseBuilder) {
        this->houseBuilder = houseBuilder;
    }

    void constructHouse() {
        houseBuilder->buildWalls();
        houseBuilder->buildRoof();
        houseBuilder->buildWindow();
    }

private:
    IHouseBuilder* houseBuilder;
};

int main() {
    setlocale(LC_ALL, "Rus");

    ConcreteHouseBuilder houseBuilder;
    Director director;

```

```
director.setBuilder(&houseBuilder);
director.constructHouse();

House house = houseBuilder.getResult();
house.showInfo();

return 0;
}
```

2.5. Prototype

Паттерн Прототип - это порождающий паттерн проектирования, который позволяет создавать новые объекты путём копирования существующих объектов (прототипов), вместо создания объектов через конструктор. Это полезно, когда создание объекта требует больших ресурсов или когда объекты имеют сложную структуру

Использование паттерна Прототип:

- Копирование объектов - когда нужно создать новый объект, который является копией существующего объекта
- Избежание сложной инициализации - когда создание объекта через конструктор требует больших ресурсов или сложной логики
- Динамическое создание объектов - когда типы создаваемых объектов определяются во время выполнения программы

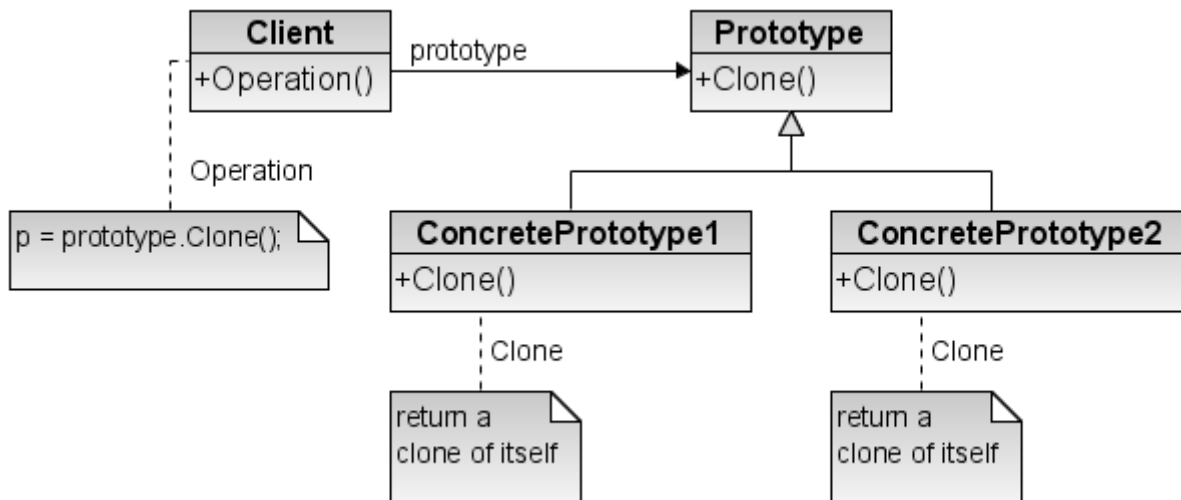
Преимущества паттерна Прототип:

- Упрощение создания объектов - позволяет создавать объекты без вызова конструктора
- Гибкость - позволяет создавать объекты с разными конфигурациями на основе прототипов
- Экономия ресурсов - избегает повторной инициализации, если объект уже существует

Недостатки паттерна Прототип:

- Сложность копирования - если объект содержит сложные структуры (например, ссылки на другие объекты), реализация копирования может быть нетривиальной
- Нарушение инкапсуляции - копирование объекта может потребовать доступа к его внутреннему состоянию

Рассмотрим UML-диаграмму паттерна Прототип:



Пояснение к диаграмме:

- **Prototype** - интерфейс или абстрактный класс, который определяет метод `clone()`
- **ConcretePrototype** - конкретный класс, реализующий метод `clone()` для создания копии объекта
- **Client** - клиентский код, который использует прототип для создания новых объектов

```
#include <iostream>

using namespace std;

class Shape {
public:
    virtual Shape* clone() const = 0;
    virtual void draw() const = 0;
    virtual ~Shape() {}
};

class Circle : public Shape {
private:
    double radius;

public:
    Circle(double rad) : radius(rad) {}

    Shape* clone() const override {
        return new Circle(*this);
    }

    void draw() const override {
        cout << "Отрисовка круга с радиусом " << radius << endl;
    }
};
```

```

class Rectangle : public Shape {
private:
    double length;
    double width;

public:
    Rectangle(double length, double width) {
        this->length = length;
        this->width = width;
    }

    Shape* clone() const override {
        return new Rectangle(*this);
    }

    void draw() const override {
        cout << "Отрисовка прямоугольника с длиной " << length << " и шириной " << width << endl;
    }
};

int main() {
    setlocale(LC_ALL, "Rus");

    Circle circlePrototype(4.5);
    Shape* shape1 = circlePrototype.clone();
    shape1->draw();

    Rectangle rectanglePrototype(4, 5);
    Shape* shape2 = rectanglePrototype.clone();
    shape2->draw();

    return 0;
}

```

2.6. Observer

Паттерн Наблюдатель - это поведенческий паттерн проектирования, который позволяет объектам (наблюдателям) подписываться на события, происходящие в другом объекте (субъекте). Когда состояние субъекта изменяется, он автоматически уведомляет всех своих наблюдателей

Использование паттерна Наблюдатель:

- Рассылка уведомлений - когда нужно уведомлять несколько объектов об изменении состояния одного объекта
- Слабая связанность - когда нужно уменьшить зависимость между объектами, чтобы изменения в одном объекте не влияли на другие

- Динамическое добавление и удаление наблюдателей - наблюдатели могут подписываться и отписываться во время выполнения программы

Преимущества паттерна Наблюдатель:

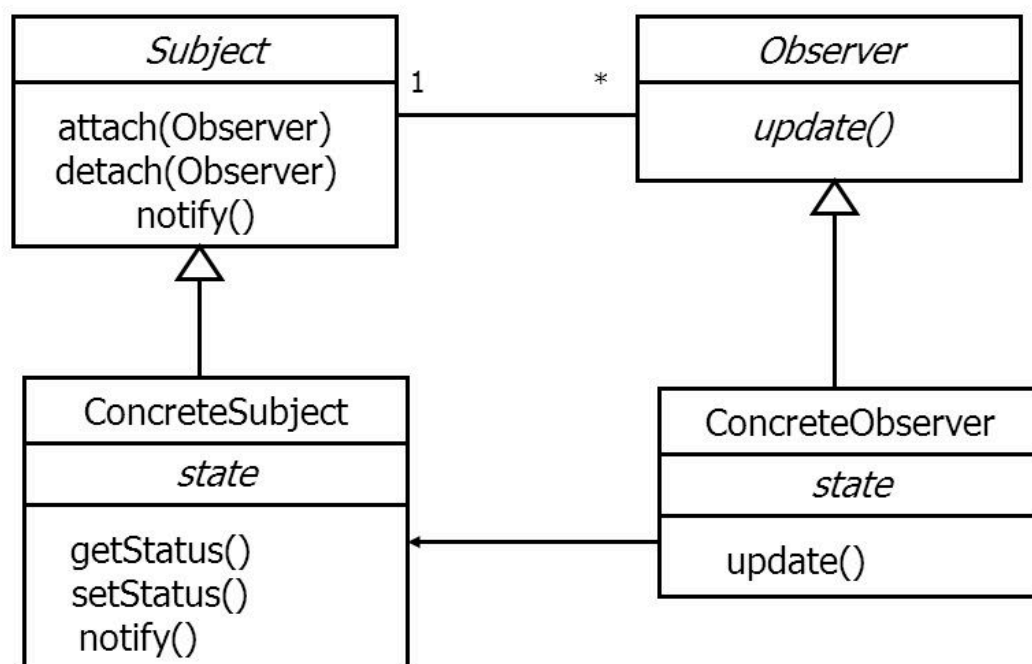
- Гибкость - позволяет динамически добавлять и удалять наблюдателей
- Слабая связанность - субъект и наблюдатели не зависят друг от друга напрямую
- Расширяемость - легко добавлять новые наблюдатели без изменения кода субъекта

Недостатки паттерна Наблюдатель:

- Неожиданные обновления - наблюдатели могут получать уведомления в неподходящее время или в неправильном порядке
- Утечки памяти - если наблюдатели не отписываются от субъекта, это может привести к утечкам памяти
- Сложность отладки - может быть сложно отследить цепочку уведомлений, особенно если наблюдателей много

Рассмотрим UML-диаграмму паттерна Наблюдатель:

Observer Pattern – Class diagram



Пояснение к диаграмме:

- Subject - интерфейс или абстрактный класс, который управляет списком наблюдателей и уведомляет их об изменениях

- Observer - интерфейс или абстрактный класс, который определяет метод update() для получения уведомлений
- ConcreteSubject - конкретный класс, который хранит состояние и уведомляет наблюдателей об изменениях
- ConcreteObserver - конкретный класс, который реализует метод update() для реакции на изменения

```
#include <iostream>
#include <vector>

using namespace std;

// Интерфейс наблюдателя
class Observer {
public:
    virtual void updateWeatherData(double temperature, double humidity,
double pressure) = 0;
};

// Субъект (метеостанция)
class Subject {
public:
    virtual void registerObserver(Observer* observer) = 0;
    virtual void deleteObserver(Observer* observer) = 0;
    virtual void notifyObservers() = 0;
    virtual void setMeasurement(double temperature, double humidity, double
pressure) = 0;
};

// Конкретный субъект (метеостанция)
class WeatherStation : public Subject {
private:
    double temperature;
    double humidity;
    double pressure;

    vector<Observer*> observers;

public:
    void registerObserver(Observer* observer) override {
        observers.push_back(observer);
    }

    void deleteObserver(Observer* observer) override {
        cout << "Наблюдатель удалён";
    }

    void notifyObservers() override {
```

```

        for (Observer* observer : observers) {
            observer->updateWeatherData(temperature, humidity, pressure);
        }
    }

    void setMeasurement(double temperature, double humidity, double
pressure) override {
        this->temperature = temperature;
        this->humidity = humidity;
        this->pressure = pressure;

        notifyObservers();
    }
};

// Конкретный наблюдатель
class Display : public Observer {
public:
    void updateWeatherData(double temperature, double humidity, double
pressure) {
        cout << "Температура: " << temperature << "°C" << endl;
        cout << "Влажность: " << humidity << "%" << endl;
        cout << "Давление: " << pressure << "гПа" << endl;
    }
};

int main() {
    setlocale(LC_ALL, "Rus");

    WeatherStation weatherStation;
    Display display;

    weatherStation.registerObserver(&display);
    weatherStation.setMeasurement(24.8, 58, 1014.5);

    return 0;
}

```

2.7. Strategy

Паттерн Стратегия - это поведенческий паттерн проектирования, который позволяет определять семейство алгоритмов, инкапсулировать каждый из них и делать их взаимозаменяемыми. Это позволяет изменять алгоритмы независимо от клиента, который их использует

Применение паттерна Стратегия:

- Выбор алгоритма - когда нужно выбрать алгоритм из нескольких вариантов во время выполнения программы
- Изоляция логики - когда нужно отделить логику алгоритма от клиентского кода
- Расширяемость - когда нужно легко добавлять новые алгоритмы без изменения существующего кода

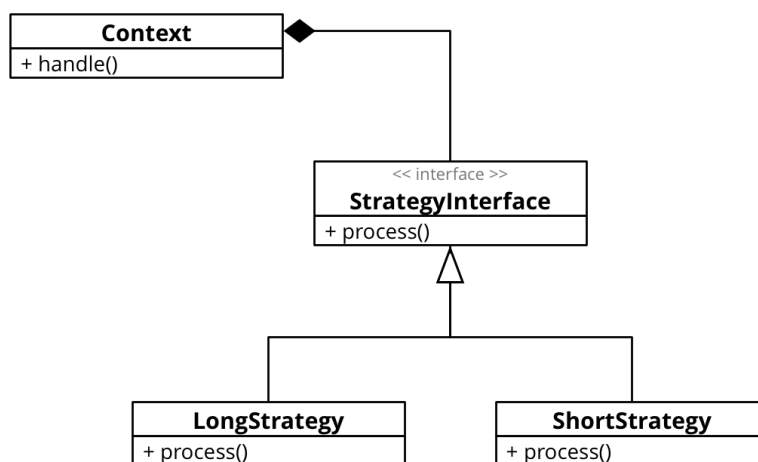
Преимущества паттерна Стратегия:

- Гибкость - позволяет легко менять алгоритмы во время выполнения программы
- Изоляция - логика алгоритмов изолирована от клиентского кода
- Расширяемость - легко добавлять новые стратегии без изменения существующего кода

Недостатки паттерна Стратегия:

- Усложнение кода - увеличивает количество классов в системе
- Необходимость выбора стратегии - клиент должен знать о существовании различных стратегий и выбирать подходящую

Рассмотрим UML-диаграмму паттерна Стратегия:



Пояснение к диаграмме:

- Context - класс, который использует стратегию. Он содержит ссылку на объект стратегии и может заменить её во время выполнения
- Strategy - интерфейс или абстрактный класс, который определяет метод execute() для выполнения алгоритма
- ConcreteStrategyA / ConcreteStrategyB - конкретные классы, реализующие интерфейс Strategy и предоставляющие конкретные алгоритмы

```
#include <iostream>
```

```

using namespace std;

// Стратегия оплаты
class PaymentStrategy {
public:
    virtual ~PaymentStrategy() = default;

    virtual void pay(int amount) const = 0;
};

// Конкретная стратегия – оплата картой
class CardPayment : public PaymentStrategy {
public:
    void pay(int amount) const override {
        cout << "Выбрана оплата картой: " << amount << " ₽" << endl;
    }
};

// Конкретная стратегия – оплата наличными
class CashPayment : public PaymentStrategy {
public:
    void pay(int amount) const override {
        cout << "Выбрана оплата наличными: " << amount << " ₽" << endl;
    }
};

// Контекст
class PaymentContext {
private:
    PaymentStrategy* paymentStrategy;

public:
    void setPaymentMethod(PaymentStrategy* paymentStrategy) {
        this->paymentStrategy = paymentStrategy;
    }

    void executePayment(int amount) const {
        if (paymentStrategy) {
            paymentStrategy->pay(amount);
        } else {
            cout << "Не выбрана стратегия оплаты" << endl;
        }
    }
};

int main() {
    setlocale(LC_ALL, "Rus");

    CardPayment cardPayment;
    CashPayment cashPayment;

```

```
PaymentContext paymentContext;  
  
paymentContext.setPaymentMethod(&cardPayment);  
paymentContext.executePayment(100);  
  
paymentContext.setPaymentMethod(&cashPayment);  
paymentContext.executePayment(200);  
  
return 0;  
}
```

2.8. Template Method

Паттерн Шаблонный метод - это поведенческий паттерн проектирования, который определяет скелет алгоритма, оставляя некоторые шаги алгоритма подклассам. Это позволяет подклассам переопределять некоторые шаги алгоритма, не меняя его структуру

Применение паттерна Шаблонный метод:

- Повторное использование кода - когда нужно избежать дублирования кода в нескольких классах, реализующих схожие алгоритмы
- Контроль структуры алгоритма - когда нужно задать общую структуру алгоритма, но позволить подклассам изменять отдельные шаги
- Упрощение расширения - когда нужно упростить добавление новых вариантов алгоритма

Преимущества паттерна Шаблонный метод:

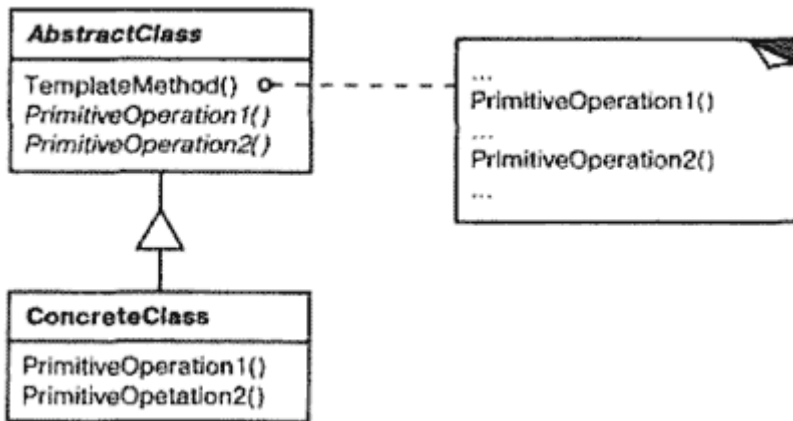
- Повторное использование - общий код алгоритма находится в одном месте
- Гибкость - подклассы могут изменять отдельные шаги алгоритма
- Контроль - базовый класс контролирует структуру алгоритма

Недостатки паттерна Шаблонный метод:

- Ограниченная гибкость - если нужно изменить структуру алгоритма, это может потребовать изменения базового класса
- Сложность отладки - может быть сложно отследить выполнение алгоритма, особенно если шаги переопределены в нескольких подклассах

Ключевая особенность паттерна Шаблонный метод заключается в том, что он имеет хуки (hooks) - это методы которые предоставляют подклассам возможность влиять на выполнение алгоритма, но не являются обязательными. Они могут быть переопределены в подклассах, чтобы добавить дополнительное поведение или изменить логику алгоритма

Рассмотрим UML-диаграмму паттерна Шаблонный метод:



Пояснение к диаграмме:

- AbstractClass - абстрактный класс, который определяет шаблонный метод и абстрактные шаги
- ConcreteClass - конкретный класс, который реализует абстрактные шаги

```
#include <iostream>

using namespace std;

// Абстрактный класс – Напиток
class Beverage {
public:
    virtual ~Beverage() = default;

    void prepareBeverage() const {
        boilWater();
        brew();
        pourInCup();

        if (customerWantsCondiments()) {
            addCondiments();
        }
    }

protected:
    virtual void brew() const = 0;
    virtual void addCondiments() const = 0;

    void boilWater() const {
        cout << "Кипятим воду" << endl;
    }

    void pourInCup() const {
        cout << "Наливаем в чашку" << endl;
    }
}
```

```

// Хук (необязательный шаг)
virtual bool customerWantsCondiments() const {
    return true;
}
};

// Конкретный класс – Чай
class Tea : public Beverage {
protected:
    void brew() const override {
        cout << "Завариваем чай" << endl;
    }

    void addCondiments() const override {
        cout << "Добавляем лимон с мёдом" << endl;
    }

    bool customerWantsCondiments() const override {
        cout << "Вы хотите добавить в чай лимон с мёдом? (y / n): ";
        char answer;
        cin >> answer;

        return answer == 'y' || answer == 'Y';
    }
};

// Конкретный класс – Кофе
class Coffee : public Beverage {
protected:
    void brew() const override {
        cout << "Засыпаем кофе в чашку" << endl;
    }

    void addCondiments() const override {
        cout << "Добавляем молоко или сливки" << endl;
    }
};

int main() {
    setlocale(LC_ALL, "Rus");

    Tea tea;
    tea.prepareBeverage();

    Coffee coffee;
    coffee.prepareBeverage();

    return 0;
}

```

2.9. Adapter

Паттерн Адаптер - это структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе. Он действует как мост между двумя несовместимыми интерфейсами, преобразуя интерфейс одного класса в интерфейс, ожидаемый клиентом

Применение паттерна Адаптер:

- Интеграция старых и новых систем - когда нужно использовать старый класс в новой системе, но его интерфейс не совместим с новым
- Работа со сторонними библиотеками - когда нужно использовать стороннюю библиотеку, но её интерфейс не соответствует ожиданиям кода
- Упрощение взаимодействия - когда нужно упростить взаимодействие между классами, которые не могут работать вместе напрямую

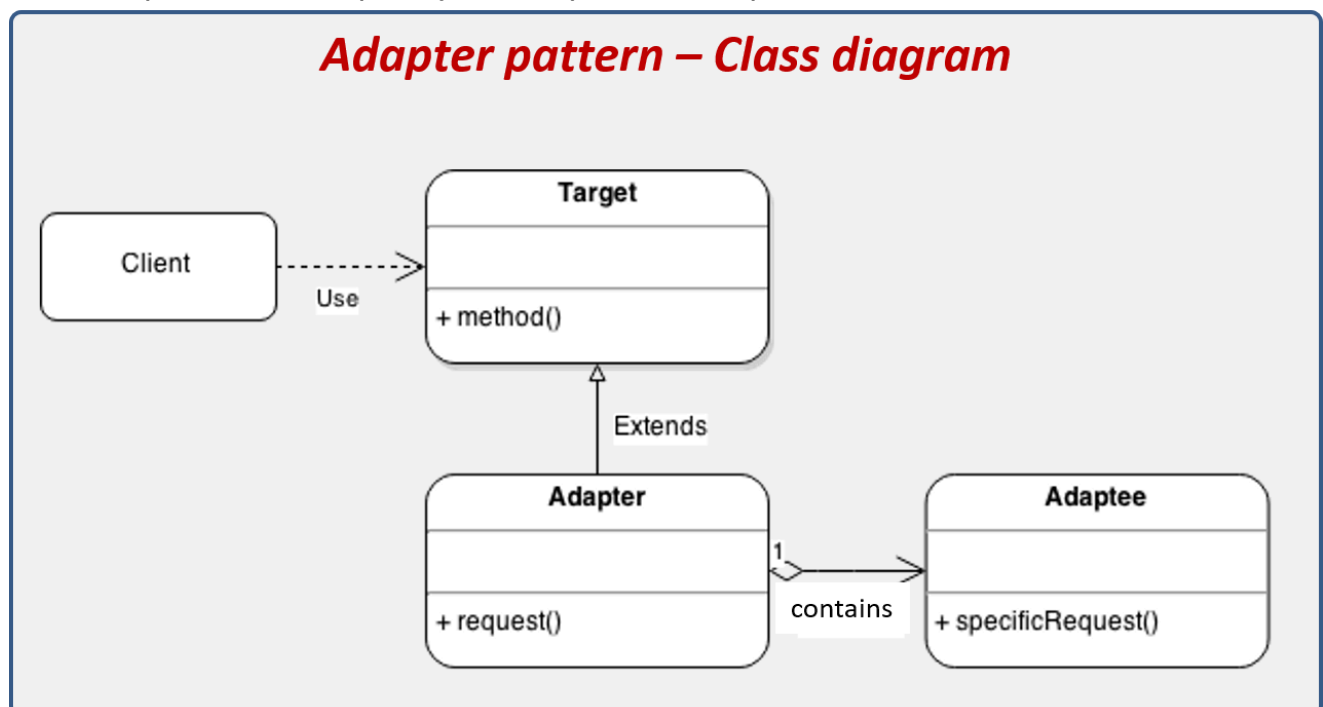
Преимущества паттерна Адаптер:

- Гибкость - позволяет использовать классы с несовместимыми интерфейсами
- Изоляция - изолирует клиентский код от деталей реализации адаптируемого класса
- Повторное использование - позволяет повторно использовать существующие классы без их изменения

Недостатки паттерна Адаптер:

- Сложность - увеличивает количество классов и объектов в системе
- Непрямой доступ - клиентский код работает с адаптером, а не напрямую с адаптируемым классом, что может усложнить отладку

Рассмотрим UML-диаграмму паттерна Адаптер:



Пояснение к диаграмме:

- Target - интерфейс, который ожидает клиент
- Adaptee - класс, который нужно адаптировать
- Adapter - класс, который адаптирует интерфейс Adaptee к интерфейсу Target
- Client - клиентский код, который использует Target

```
#include <iostream>

using namespace std;

// Адаптируемый класс – устаревший печатник
class LegacyPrinter {
public:
    void printInUppercase(const string& text) {
        cout << "Напечатано: " << text << endl;
    }
};

// Клиент – современный печатник
class ModernPrinter {
public:
    void sendCommand(const string& command) {
        cout << "Отправка команды: " << command << endl;
    }
};

// Адаптер, чтобы совместить классы LegacyPrinter и ModernPrinter
class PrinterAdapter {
private:
    LegacyPrinter* legacyPrinter;
public:
    void sendCommand(const string& command) {
        // Преобразуем команду в_верхний_регистр и передадим её LegacyPrinter
        string uppercaseCommand = command;

        for (char& symbol : uppercaseCommand) {
            symbol = toupper(symbol);
        }

        legacyPrinter->printInUppercase(uppercaseCommand);
    }
};

int main() {
    setlocale(LC_ALL, "Rus");

    ModernPrinter modernPrinter;
```

```
PrinterAdapter printerAdapter;

modernPrinter.sendCommand("печать в нижнем регистре");
printerAdapter.sendCommand("печать в верхнем регистре");

return 0;
}
```

3.MVC

3.1. MVC (Model-View-Controller)

Паттерн MVC - это архитектурный паттерн, который разделяет приложение на три основные компоненты:

- Model (Модель) - отвечает за данные и бизнес-логику приложения. Модель не зависит от представления и контроллер
- View (Представление) - отвечает за отображение данных пользователю. Оно получает данные от модели и отображает их
- Controller (Контроллер) - обрабатывает пользовательский ввод, взаимодействует с моделью и обновляет представление

Применение паттерна MVC: используется для разделения ответственностей в приложении, что делает код более модульным, поддерживаемым и тестируемым. Это особенно полезно в веб-приложениях и графических интерфейсах пользователя (GUI)

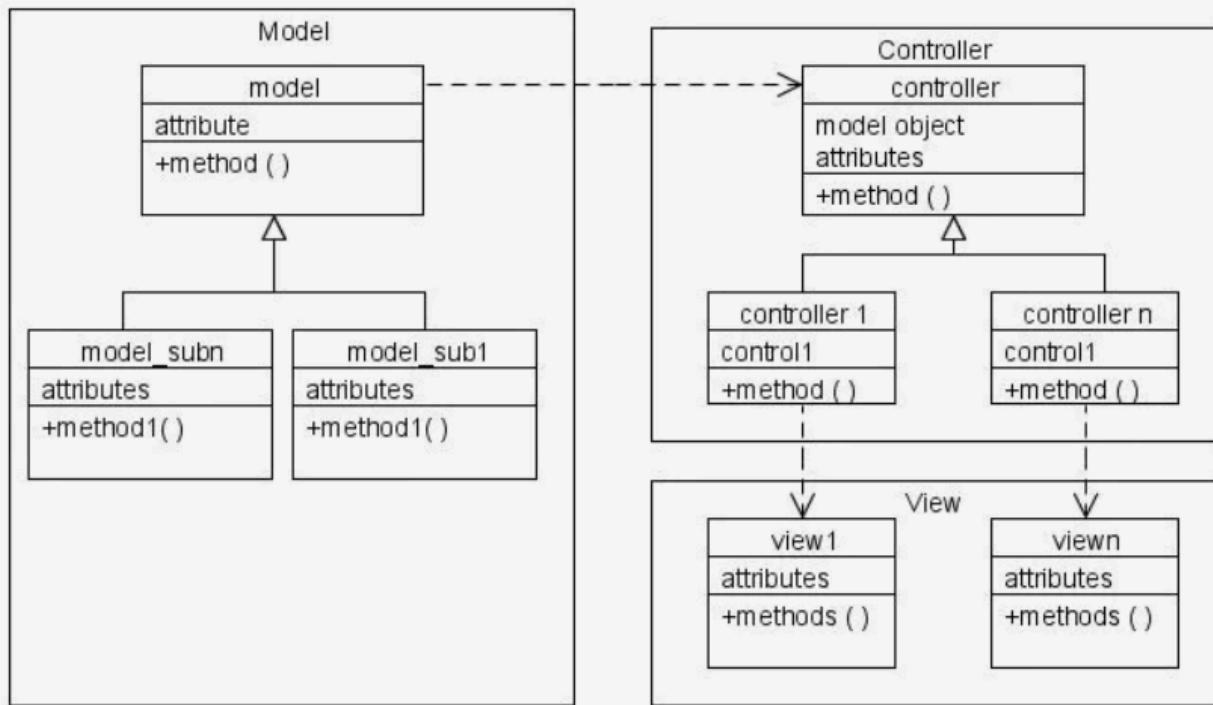
Преимущества паттерна MVC:

- Разделение ответственностей - каждый компонент выполняет свою задачу, что упрощает разработку и поддержку
- Повторное использование кода - модель и контроллер могут быть использованы в разных представлениях
- Упрощение тестирования - компоненты можно тестировать независимо друг от друга
- Гибкость - легко вносить изменения в один компонент, не затрагивая другие

Недостатки паттерна MVC:

- Сложность - для небольших приложений может быть избыточным
- Количество кода - требуется больше кода для реализации всех компонентов

Рассмотрим UML-диаграмму паттерна MVC:



Пояснение к диаграмме:

- View - взаимодействует с Controller для обработки пользовательского ввода
- Controller - обновляет Model и уведомляет View об изменениях
- Model - уведомляет View об изменениях данных, и View обновляет отображение

```
#include <iostream>

using namespace std;

// Model (Модель) – Пользователь
class UserModel {
private:
    string name;
    int age{};

public:
    string getName() const {
        return this->name;
    }

    void setName(const string& name) {
        this->name = name;
    }

    int getAge() const {
        return this->age;
    }
}
```

```

    void setAge(int age) {
        this->age = age;
    }
};

// View (Представление) – Отображение данные о пользователе
class UserView {
public:
    void showDetails(const string& name, int age) const {
        cout << "Имя пользователя: " << name << endl;
        cout << "Возраст пользователя: " << age << endl;
    }
};

// Controller (Контроллер) – Ввод данных о пользователе
class UserController {
private:
    UserModel model;
    UserView view;

public:
    void setDetails(const string& name, int age) {
        model.setName(name);
        model.setAge(age);
    }

    void updateView() {
        cout << "Данные обновлены" << endl;
        view.showDetails(model.getName(), model.getAge());
    }
};

int main() {
    setlocale(LC_ALL, "Rus");

    UserController userController;
    userController.setDetails("Денис", 21);
    userController.updateView();

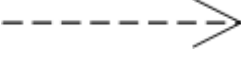
    userController.setDetails("Юлия", 22);
    userController.updateView();

    return 0;
}

```

4. Отношения между классами. UML

4.1. Зависимость

Зависимость(зависимый  независимый) - это тип отношения между классами, который показывает, что один класс зависит от другого, чтобы выполнить свою функциональность. Это означает, что при изменении независимого класса, зависимый от него может поменять своё поведение. Зависимость является самой слабой связью между классами. Наиболее распространённый пример такого типа отношения - класс X использует в своём методе метод класса Y, то есть X зависит от Y

Объяснение: в этом примере класс Car зависит от класса Engine, так как использует его метод start() в своём методе start(Engine& engine) Или, другими словами, для запуска автомобиля требуется запустить двигатель

```
#include <iostream>

using namespace std;

class Engine {
public:
    void start() const {
        cout << "Двигатель запущен" << endl;
    }
};

class Car {
public:
    void start(Engine& engine) const {
        engine.start();

        cout << "Машина готова к поездке" << endl;
    }
};

int main() {
    setlocale(LC_ALL, "Rus");

    Engine engine;


    Car car;
    car.start(engine);

    return 0;
}
```

4.2.1. Агрегация

Ассоциация - это тип отношения между классами, который показывает, что объекты одного класса связаны с объектами другого класса

Существует три частных случая ассоциации - *Агрегация*, *Композиция*, *Обобщение*

Агрегация(часть  целое) - это отношение "часть-целое", где объекты составляют часть объекта-контейнера, но не зависят от него. То есть объект-контейнер не управляет их временем жизни (если контейнер будет уничтожен, то его содержимое - нет)

Объяснение: в этом примере класс University содержит объекты Student, но Student не зависят от University. Если University будет уничтожен, то Student - нет

```
#include <iostream>
#include <vector>

using namespace std;

class Student {
private:
    string name;

public:
    Student(const string& name) : name(name) {}

    string getName() const {
        return name;
    }
};

class University {
private:
    vector<Student*> students;

public:
    void addStudent(Student* student) {
        students.push_back(student);
    }

    void displayStudents() const {
        for (const auto& student : students) {
            cout << "Студент " << student->getName() << endl;
        }
    }
};

int main() {
    setlocale(LC_ALL, "Rus");

    Student student1("Денис");
```

```

Student student2("Юлия");

University university;
university.addStudent(&student1);
university.addStudent(&student2);
university.displayStudents();


return 0;
}

```

4.2.2. Композиция

Ассоциация - это тип отношения между классами, который показывает, что объекты одного класса связаны с объектами другого класса

Существует три частных случая ассоциации - *Агрегация*, *Композиция*, *Обобщение*

Композиция (часть  целое) - это отношение "часть-целое", где объекты создаются внутри класса-контейнера и управляются им. На содержащийся объект может ссылаться только содержащий его объект-контейнер, и первый должен быть удалён при удалении объекта-контейнера. То есть объект-контейнер управляет временем жизни содержащихся объектов

Объяснение: в этом примере класс Car содержит объект Engine, который создаётся внутри Car и удаляется вместе с ним

```

#include <iostream>

using namespace std;

class Engine {
public:
    void start() const {
        cout << "Двигатель запущен" << endl;
    }
};

class Car {
private:
    Engine* engine;

public:
    Car() {
        engine = new Engine();
    }

    ~Car() {
        delete engine;
    }
}

```

```

    cout << "Поездка окончена, двигатель выключен" << endl;
}

void start() const {
    engine->start();

    cout << "Машина к поездке готова" << endl;
}
};

int main() {
    setlocale(LC_ALL, "Rus");

    Car car;
    car.start();


    return 0;
}

```

4.2.3. Обобщение

Ассоциация - это тип отношения между классами, который показывает, что объекты одного класса связаны с объектами другого класса

Существует три частных случая ассоциации - *Агрегация*, *Композиция*, *Обобщение*

Обобщение (дочерний класс/подкласс  родительский класс/суперклассом/базовый класс) - это тип отношений, который выражается через наследование, где дочерний класс является более конкретным (уточнённым / расширенным) по отношению к родительскому классу

Объяснение: в этом примере класс Dog наследуется от класса Animal и переопределяет метод makeSound(). Это показывает, что Dog является более конкретным типом Animal

```

#include <iostream>

using namespace std;

class Animal {
public:
    virtual void makeSound() const {
        cout << "*Животное издаёт звук*" << endl;
    }
};

class Dog : public Animal {

```

```

public:
    void makeSound() const override {
        cout << "Гав-гав!" << endl;
    }
};

int main() {
    setlocale(LC_ALL, "Rus");


    Animal* dog = new Dog();
    dog->makeSound();

    delete dog;

    return 0;
}

```

4.3. Реализация

Реализация (реализующий класс  интерфейс/абстрактный класс) - это тип отношений между классами, который выражается через интерфейсы или абстрактные классы, где класс реализует методы интерфейса или абстрактного класса

Объяснение: в этом примере класс Xiaomi реализует интерфейс ISmartphone, предоставляя свои версии методов call(const string& number), useBrowser(), useCamera()

```

#include <iostream>

using namespace std;

class ISmartphone {
public:
    virtual void call(const string& number) const = 0;
    virtual void useBrowser() const = 0;
    virtual void useCamera() const = 0;
};

class Xiaomi : public ISmartphone {
public:
    void call(const string& number) const override {
        cout << "Вызов на номер " << number << endl;
    }

    void useBrowser() const override {
        cout << "Загрузка Google Chrome..." << endl;
    }
}

```

```
void useCamera() const override {  
    cout << "Открытие камеры..." << endl;  
}  
};  
  
int main() {  
    setlocale(LC_ALL, "Rus");  
  
    Xiaomi xiaomi;  
    xiaomi.call("+7-999-000-11-22");  
    xiaomi.useBrowser();  
    xiaomi.useCamera();  
  
    return 0;  
}
```

5.File Monitoring

Лабораторная работа № 1. Наблюдение за файлами

Постановка задачи

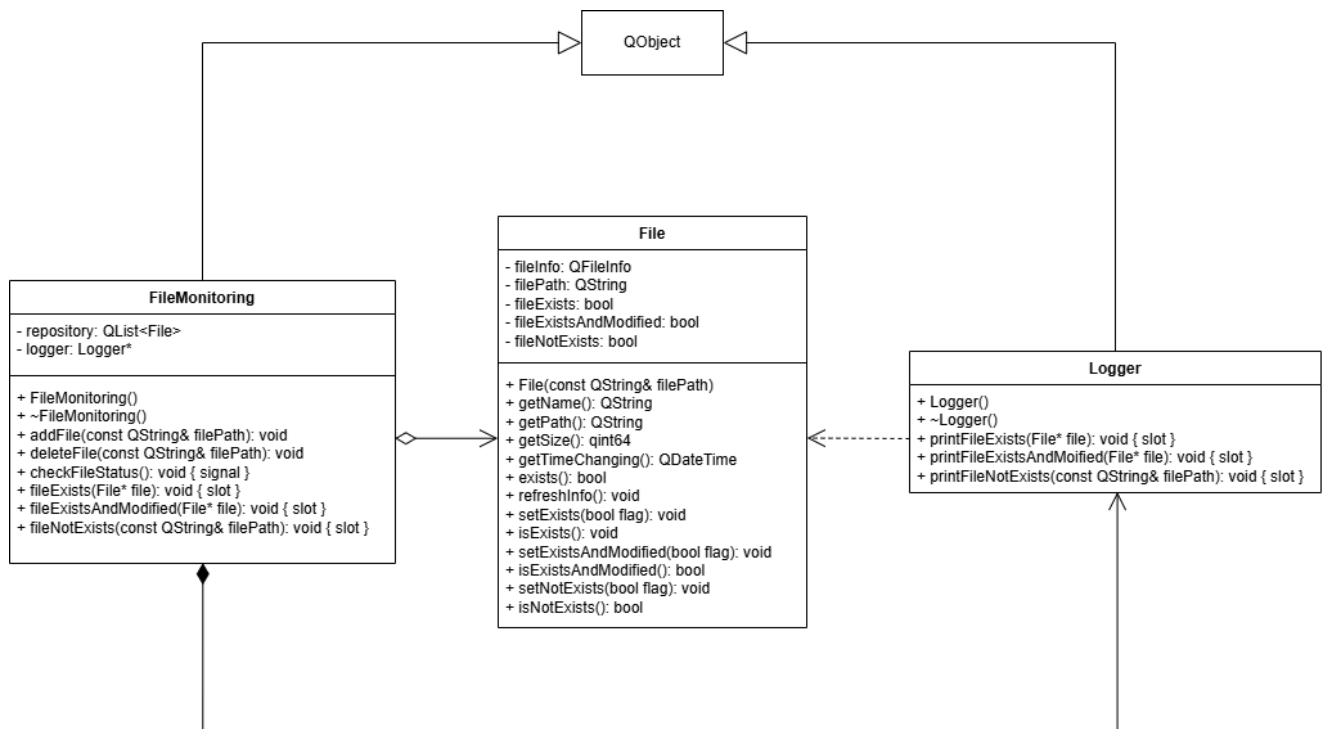
Реализовать консольное приложение, которое выполняет слежение за выбранными файлами. При возникновении изменения состояния наблюдаемого файла, на экран будет выведено соответствующее сообщение

Характеристика файла

1. Существование файла
2. Размер файла

Возможные ситуации для наблюдаемого файла

1. Файл существует, файл не был изменён - на экран выводится факт существования файла и его размер
2. Файл существует, файл был изменён - на экран выводится факт существования файла, а также сообщение с информацией о том, что файл был изменён, и текущий размер файла
3. Файл не существует - на экран выводится информация о том, что файл не существует!



file.h

```

#ifndef FILE_H
#define FILE_H

#include <QFileInfo>
#include <QDateTime>

class File {
public:
    File(const QString& filePath);

    QString getName() const;
    QString getPath() const;
    qint64 getSize() const;
    QDateTime getTimeChanging() const;

    bool exists() const;
    void refreshInfo();

    void setExists(bool flag);
    bool isExists() const;

    void setExistsAndModified(bool flag);
    bool isExistsAndModified() const;

    void setNotExists(bool flag);
    bool isNotExists() const;

private:

```

```

    QFileInfo fileInfo;
    QString filePath;

    bool fileExists;
    bool fileExistsAndModified;
    bool fileNotExists;
};

#endif // FILE_H

```

file.cpp

```

#include "file.h"

File::File(const QString& filePath) {
    fileInfo = QFileInfo(filePath);
    this->filePath = filePath;

    fileExists = false;
    fileExistsAndModified = false;
    fileNotExists = false;
}

QString File::getName() const {
    return fileInfo.fileName();
}

QString File::getPath() const {
    return fileInfo.filePath();
}

qint64 File::getSize() const {
    return fileInfo.size();
}

QDateTime File::getTimeChanging() const {
    return fileInfo.lastModified();
}

bool File::exists() const {
    return fileInfo.exists();
}

void File::refreshInfo() {
    fileInfo.refresh();
}

void File::setExists(bool flag) {
    fileExists = flag;
}

```

```

}

bool File::isExists() const {
    return fileExists;
}

void File::setExistsAndModified(bool flag) {
    fileExistsAndModified = flag;
}

bool File::isExistsAndModified() const {
    return fileExistsAndModified;
}

void File::setNotExists(bool flag) {
    fileNotExists = flag;
}

bool File::isNotExists() const {
    return fileNotExists;
}

```

filemonitoring.h

```

#ifndef FILEMONITORING_H
#define FILEMONITORING_H

#include <QObject>
#include <QList>
#include "file.h"

class Logger;

class FileMonitoring : public QObject {
    Q_OBJECT

public:
    FileMonitoring();
    ~FileMonitoring();

    void addFile(const QString& filePath);
    void deleteFile(const QString& filePath);

public slots:
    void checkFileStatus();

signals:
    void fileExists(File* file);
    void fileExistsAndModified(File* file);

```



```

        void fileNotExists(const QString& filePath);

private:
    QList<File> repository;
    Logger* logger;
};

#endif // FILEMONITORING_H

```

filemonitoring.cpp

```

#include "filemonitoring.h"
#include "logger.h"

FileMonitoring::FileMonitoring() {
    logger = new Logger();

    connect(this, &FileMonitoring::fileExists, logger,
    &Logger::printFileExists);
    connect(this, &FileMonitoring::fileExistsAndModified, logger,
    &Logger::printFileExistsAndModified);
    connect(this, &FileMonitoring::fileNotExists, logger,
    &Logger::printFileNotExists);
}

FileMonitoring::~FileMonitoring() {
    delete logger;
}

void FileMonitoring::addFile(const QString& filePath) {
    File file(filePath);
    file.refreshInfo();

    repository.push_back(file);
}

void FileMonitoring::deleteFile(const QString& filePath) {
    int index = -1;

    for (int i = 0; i < repository.size(); ++i) {
        if (repository[i].getPath() == filePath) {
            index = i;

            break;
        }
    }

    if (index != -1) {
        repository.removeAt(index);
    }
}

```

```

    }
}

void FileMonitoring::checkFileStatus() {
    for (auto& currentFile : repository) {
        QFileInfo updatedFileInfo(currentFile.getPath());

        if (!updatedFileInfo.exists()) {
            if (!currentFile.isNotExists()) {
                emit fileNotExists(currentFile.getPath());

                currentFile.setNotExists(true);
                currentFile.setExists(false);
                currentFile.setExistsAndModified(false);
            }
        } else {
            if (updatedFileInfo.lastModified() ==
currentFile.getTimeChanging()) {
                if (!currentFile.isExists()) {
                    emit fileExists(&currentFile);

                    currentFile.setExists(true);
                    currentFile.setNotExists(false);
                }
            } else {
                currentFile.refreshInfo();

                emit fileExistsAndModified(&currentFile);

                currentFile.setExistsAndModified(true);
                currentFile.setExists(true);
                currentFile.setNotExists(false);
            }
        }
    }
}
}

```

logger.h

```

#ifndef LOGGER_H
#define LOGGER_H

#include <QObject>

class File;

class Logger : public QObject {
    Q_OBJECT

```

```

public:
    Logger() {}
    ~Logger() {}

public slots:
    void printFileExists(File* file);
    void printFileExistsAndModified(File* file);
    void printFileNotExists(const QString& filePath);
};

#endif // LOGGER_H

```

logger.cpp

```

#include "logger.h"
#include "file.h"
#include <QDebug>

void Logger::printFileExists(File* file) {
    qDebug() << "File " << file->getName() << " exists";
    qDebug() << "File size: " << file->getSize();
    qDebug() << "";
}

void Logger::printFileExistsAndModified(File* file) {
    qDebug() << "File " << file->getName() << " exists and modified";
    qDebug() << "File size: " << file->getSize();
    qDebug() << "";
}

void Logger::printFileNotExists(const QString& filePath) {
    qDebug() << "File on path " << filePath << " doesn't exists";
    qDebug() << "";
}

```

main.cpp

```

#include <QCoreApplication>
#include <QDebug>
#include <QTimer>
#include "filemonitoring.h"

int main(int argc, char *argv[]) {
    QCoreApplication a(argc, argv);

    qDebug() << "\tFile Monitoring Program\n";

    FileMonitoring fileMonitoring1;
}

```

```

fileMonitoring1.addFile("../file1.txt");

FileMonitoring fileMonitoring2;
fileMonitoring2.addFile("../file2.txt");

QTimer timer;

QObject::connect(&timer, &QTimer::timeout, &fileMonitoring1,
&FileMonitoring::checkFileStatus);
QObject::connect(&timer, &QTimer::timeout, &fileMonitoring2,
&FileMonitoring::checkFileStatus);

timer.start(1000);

return a.exec();
}

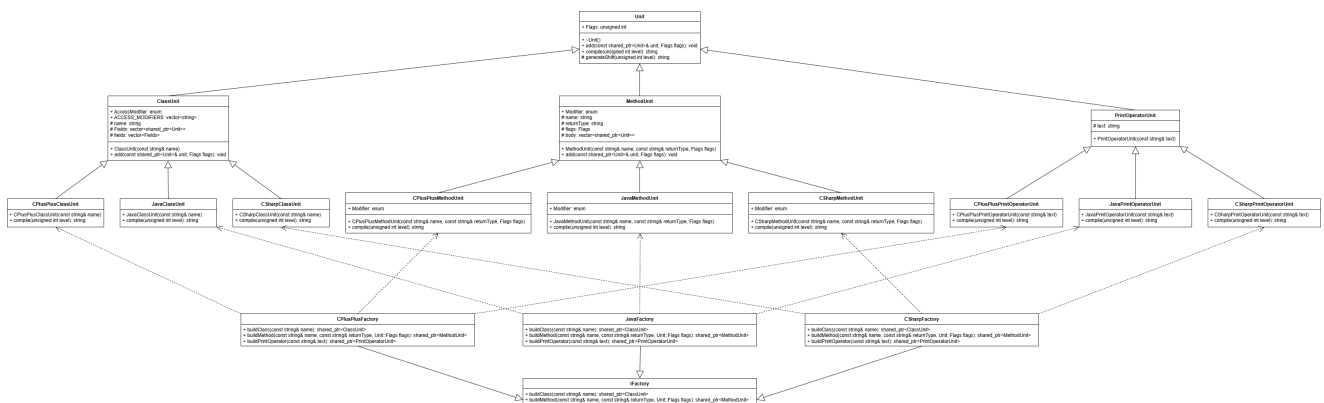
```

6. Code Generator

Лабораторная работа № 2. Генератор кода

Постановка задачи

Используя паттерн "Абстрактная фабрика", реализовать консольное приложение, способное генерировать код для таких языков программирования, как C++, C#, Java



unit.h

```

#ifndef UNIT_H
#define UNIT_H

#include <iostream>
#include <memory>
#include <vector>

using std::string;
using std::vector;

```

```

using std::shared_ptr;

class Unit {
public:
    using Flags = unsigned int;

public:
    virtual ~Unit() = default;

    virtual void add(const shared_ptr<Unit>&, Flags) {
        throw std::runtime_error("Not supported");
    }

    virtual string compile(unsigned int level = 0) const = 0;

protected:
    virtual string generateShift(unsigned int level) const {
        static const auto defaultShift = " ";
        string result;

        for (unsigned int i = 0; i < level; ++i) {
            result += defaultShift;
        }

        return result;
    }
};

class ClassUnit : public Unit {
public:
    enum AccessModifier {
        PUBLIC,
        PROTECTED,
        PRIVATE
    };

    static const vector<string> ACCESS_MODIFIERS;

public:
    explicit ClassUnit(const string& name) : name(name) {
        fields.resize(ACCESS_MODIFIERS.size());
    }

    void add(const shared_ptr<Unit>& unit, Flags flags) override {
        int accessModifier = PRIVATE;

        if (flags < ACCESS_MODIFIERS.size()) {
            accessModifier = flags;
        }
    }
}

```

```

        fields[accessModifier].push_back(unit);
    }

protected:
    string name;
    using Fields = vector<shared_ptr<Unit>>;
    vector<Fields> fields;
};

class MethodUnit : public Unit {
public:
    enum Modifier {
        STATIC = 1,
        CONST = 1 << 1,
        VIRTUAL = 1 << 2,
        FINAL = 1 << 3,
        ABSTRACT = 1 << 4,
        PUBLIC = 1 << 5,
        PROTECTED = 1 << 6,
        PRIVATE = 1 << 7,
        INTERNAL = 1 << 8
    };

public:
    MethodUnit(const string& name, const string& returnType, Flags flags)
    : name(name), returnType(returnType), flags(flags) {}

    void add(const shared_ptr<Unit>& unit, Flags /* flags */ = 0) override
    {
        body.push_back(unit);
    }

protected:
    string name;
    string returnType;
    Flags flags;
    vector<shared_ptr<Unit>> body;
};

class PrintOperatorUnit : public Unit {
public:
    explicit PrintOperatorUnit(const string& text) : text(text) {}

protected:
    string text;
};

#endif // UNIT_H

```

unit.cpp

```
#include "unit.h"

const vector<string> ClassUnit::ACCESS_MODIFIERS = { "public",
"protected", "private" };
```

factory.h

```
#ifndef FACTORY_H
#define FACTORY_H

#include "unit.h"

class IFactory {
public:
    virtual shared_ptr<ClassUnit> buildClass(const string& name) = 0;
    virtual shared_ptr<MethodUnit> buildMethod(const string& name, const
string& returnType, Unit::Flags flags) = 0;
    virtual shared_ptr<PrintOperatorUnit> buildPrintOperator(const string&
text) = 0;
};

class CPlusPlusFactory : public IFactory {
public:
    shared_ptr<ClassUnit> buildClass(const string& name) override;
    shared_ptr<MethodUnit> buildMethod(const string& name, const string&
returnType, Unit::Flags flags) override;
    shared_ptr<PrintOperatorUnit> buildPrintOperator(const string& text)
override;
};

class JavaFactory : public IFactory {
public:
    shared_ptr<ClassUnit> buildClass(const string& name) override;
    shared_ptr<MethodUnit> buildMethod(const string& name, const string&
returnType, Unit::Flags flags) override;
    shared_ptr<PrintOperatorUnit> buildPrintOperator(const string& text)
override;
};

class CSharpFactory : public IFactory {
public:
    shared_ptr<ClassUnit> buildClass(const string& name) override;
    shared_ptr<MethodUnit> buildMethod(const string& name, const string&
returnType, Unit::Flags flags) override;
    shared_ptr<PrintOperatorUnit> buildPrintOperator(const string& text)
override;
};
```

```
};

#endif // FACTORY_H
```

factory.cpp

```
#include "factory.h"
#include "cplusplusunit.h"
#include <javaunit.h>
#include <csharpunit.h>

using std::make_shared;

shared_ptr<ClassUnit> CPlusPlusFactory::buildClass(const string& name) {
    return make_shared<CPlusPlusClassUnit>(name);
}

shared_ptr<MethodUnit> CPlusPlusFactory::buildMethod(const string& name,
const string& returnType, Unit::Flags flags) {
    return make_shared<CPlusPlusMethodUnit>(name, returnType, flags);
}

shared_ptr<PrintOperatorUnit> CPlusPlusFactory::buildPrintOperator(const
string& text) {
    return make_shared<CPlusPlusPrintOperatorUnit>(text);
}

shared_ptr<ClassUnit> JavaFactory::buildClass(const string& name) {
    return make_shared<JavaClassUnit>(name);
}

shared_ptr<MethodUnit> JavaFactory::buildMethod(const string& name, const
string& returnType, Unit::Flags flags) {
    return make_shared<JavaMethodUnit>(name, returnType, flags);
}

shared_ptr<PrintOperatorUnit> JavaFactory::buildPrintOperator(const
string& text) {
    return make_shared<JavaPrintOperatorUnit>(text);
}

shared_ptr<ClassUnit> CSharpFactory::buildClass(const string& name) {
    return make_shared<CSharpClassUnit>(name);
}

shared_ptr<MethodUnit> CSharpFactory::buildMethod(const string& name,
const string& returnType, Unit::Flags flags) {
    return make_shared<CSharpMethodUnit>(name, returnType, flags);
}
```



```

shared_ptr<PrintOperatorUnit> CSharpFactory::buildPrintOperator(const
string& text) {
    return make_shared<CSharpPrintOperatorUnit>(text);
}

```

cplusplusunit.h

```

#ifndef CPLUSPLUSUNIT_H
#define CPLUSPLUSUNIT_H

#include "unit.h"

class CPlusPlusClassUnit : public ClassUnit {
public:
    CPlusPlusClassUnit(const string& name) : ClassUnit(name) {}

    string compile(unsigned int level = 0) const override;
};

class CPlusPlusMethodUnit : public MethodUnit {
public:
    enum Modifier {
        STATIC = 1,
        CONST = 1 << 1,
        VIRTUAL = 1 << 2
    };

public:
    CPlusPlusMethodUnit(const string& name, const string& returnType,
Flags flags) : MethodUnit(name, returnType, flags) {}

    string compile(unsigned int level = 0) const override;
};

class CPlusPlusPrintOperatorUnit : public PrintOperatorUnit {
public:
    CPlusPlusPrintOperatorUnit(const string& text) :
PrintOperatorUnit(text) {}

    string compile(unsigned int level = 0) const override;
};

#endif // CPLUSPLUSUNIT_H

```

cplusplusunit.cpp

```

#include "cplusplusunit.h"

string CPlusPlusClassUnit::compile(unsigned int level) const {
    string result = generateShift(level) + "class " + name + "{\n";

    for (size_t i = 0; i < ACCESS_MODIFIERS.size(); ++i) {
        if (fields[i].empty()) {
            continue;
        }

        result += ACCESS_MODIFIERS[i] + ":\n";

        for (const auto& f : fields[i]) {
            result += f->compile(level + 1);
            result += "\n";
        }
    }

    result += generateShift(level) + "};\n";
    return result;
}

string CPlusPlusMethodUnit::compile(unsigned int level) const {
    string result = generateShift(level);

    if (flags & STATIC) {
        result += "static ";
    } else if (flags & VIRTUAL) {
        result += "virtual ";
    }

    result += returnType + " ";
    result += name + "()";

    if (flags & CONST) {
        result += " const";
    }

    result += "{\n";

    for (const auto& b : body) {
        result += b->compile(level + 1);
    }

    result += generateShift(level) + ";\n";
    return result;
}

string CPlusPlusPrintOperatorUnit::compile(unsigned int level) const {

```

```
    return generateShift(level) + "printf(\"" + text + "\");\n";  
}
```

csharpunit.h

```
#ifndef CSHARPUNIT_H  
#define CSHARPUNIT_H  
  
#include "unit.h"  
  
class CSharpClassUnit : public ClassUnit {  
public:  
    CSharpClassUnit(const string& name) : ClassUnit(name) {}  
  
    string compile(unsigned int level = 0) const override;  
};  
  
class CSharpMethodUnit : public MethodUnit {  
public:  
    enum Modifier {  
        STATIC = 1,  
        CONST = 1 << 1,  
        VIRTUAL = 1 << 2,  
        PUBLIC = 1 << 5,  
        PROTECTED = 1 << 6,  
        PRIVATE = 1 << 7,  
        INTERNAL = 1 << 8  
    };  
  
public:  
    CSharpMethodUnit(const string& name, const string& returnType, Flags  
flags) : MethodUnit(name, returnType, flags) {}  
  
    string compile(unsigned int level = 0) const override;  
};  
  
class CSharpPrintOperatorUnit : public PrintOperatorUnit {  
public:  
    CSharpPrintOperatorUnit(const string& text) : PrintOperatorUnit(text)  
{}  
  
    string compile(unsigned int level = 0) const override;  
};  
  
#endif // CSHARPUNIT_H
```

csharpunit.cpp

```
#include "csharpunit.h"
```

```
string CSharpClassUnit::compile(unsigned int level) const {
    string result = generateShift(level) + "class " + name + "{\n";

    for (size_t i = 0; i < ACCESS_MODIFIERS.size(); ++i) {
        if (fields[i].empty()) {
            continue;
        }

        for (const auto& f : fields[i]) {
            result += f->compile(level + 1);
            result += "\n";
        }
    }

    result += generateShift(level) + "}\n";
    return result;
}
```

```
string CSharpMethodUnit::compile(unsigned int level) const {
    string result = generateShift(level);

    if (flags & PUBLIC) {
        result += "public ";
    } else if (flags & PROTECTED) {
        result += "protected ";
    } else if (flags & PRIVATE) {
        result += "private ";
    } else if (flags & INTERNAL) {
        result += "internal ";
    }

    if (flags & STATIC) {
        result += "static ";
    } else if (flags & CONST) {
        result += "const ";
    } else if (flags & VIRTUAL) {
        result += "virtual ";
    }

    result += returnType + " ";
    result += name + "()";
    result += "{\n";

    for (const auto& b : body) {
        result += b->compile(level + 1);
    }
}
```

```

        result += generateShift(level) + "}\n";
        return result;
    }

    string CSharpPrintOperatorUnit::compile(unsigned int level) const {
        return generateShift(level) + "Console.WriteLine(\"" + text +
"\");\n";
    }

```

javaunit.h

```

#ifndef JAVAUNIT_H
#define JAVAUNIT_H

#include "unit.h"

class JavaClassUnit : public ClassUnit {
public:
    JavaClassUnit(const string& name) : ClassUnit(name) {}

    string compile(unsigned int level = 0) const override;
};

class JavaMethodUnit : public MethodUnit {
public:
    enum Modifier {
        STAATIC = 1,
        FINAL = 1 << 3,
        ABSTRACT = 1 << 4,
        PUBLIC = 1 << 5,
        PROTECTED = 1 << 6,
        PRIVATE = 1 << 7
    };

public:
    JavaMethodUnit(const string& name, const string& returnType, Flags
flags) : MethodUnit(name, returnType, flags) {}

    string compile(unsigned int level = 0) const override;
};

class JavaPrintOperatorUnit : public PrintOperatorUnit {
public:
    JavaPrintOperatorUnit(const string& text) : PrintOperatorUnit(text) {}

    string compile(unsigned int level = 0) const override;
};

```

```
#endif // JAVAUNIT_H
```

javaunit.cpp

```
#include "javaunit.h"

string JavaClassUnit::compile(unsigned int level) const {
    string result = generateShift(level) + "class " + name + "{\n";

    for (size_t i = 0; i < ACCESS_MODIFIERS.size(); ++i) {
        if (fields[i].empty()) {
            continue;
        }

        for (const auto& f : fields[i]) {
            result += f->compile(level + 1);
            result += "\n";
        }
    }

    result += generateShift(level) + "};\n";
    return result;
}

string JavaMethodUnit::compile(unsigned int level) const {
    string result = generateShift(level);

    if (flags & PUBLIC) {
        result += "public ";
    } else if (flags & PROTECTED) {
        result += "protected ";
    } else if (flags & PRIVATE) {
        result += "private ";
    }

    if (flags & STATIC) {
        result += "static ";
    } else if (flags & FINAL) {
        result += "final ";
    }

    result += returnType + " ";
    result += name + "()";
    result += "{\n";

    for (const auto& b : body) {
        result += b->compile(level + 1);
    }
}
```

```

        result += generateShift(level) + "}\n";
        return result;
    }

    string JavaPrintOperatorUnit::compile(unsigned int level) const {
        return generateShift(level) + "System.out.println(\"" + text +
"\");\n";
    }

```

main.cpp

```

#include <QCoreApplication>
#include "unit.h"
#include "factory.h"

using std::cout;
using std::endl;

string generateProgram(IFactory* factory, const string& className, const
string& text) {
    shared_ptr<ClassUnit> myClass = factory->buildClass(className);

    myClass->add(
        factory->buildMethod("testFunc1", "void", MethodUnit::PUBLIC),
        ClassUnit::PUBLIC
    );

    myClass->add(
        factory->buildMethod("testFunc2", "void", MethodUnit::STATIC |
MethodUnit::PRIVATE),
        ClassUnit::PRIVATE
    );

    myClass->add(
        factory->buildMethod("testFunc3", "void", MethodUnit::CONST |
MethodUnit::PROTECTED | MethodUnit::PRIVATE),
        ClassUnit::PROTECTED
    );

    shared_ptr<MethodUnit> method = factory->buildMethod("testFunc4",
"void", MethodUnit::STATIC | MethodUnit::PROTECTED);
    method->add(factory->buildPrintOperator(text));
    myClass->add(method, ClassUnit::PROTECTED);

    return myClass->compile();
}

int main(int argc, char *argv[]) {

```

```

QCoreApplication a(argc, argv);

cout << "C++ program:" << endl;
CPlusPlusFactory factory1;
cout << generateProgram(&factory1, "CPlusPlusClass", "Hello C++") << endl;

cout << "Java program:" << endl;
JavaFactory factory2;
cout << generateProgram(&factory2, "JavaClass", "Hello Java") << endl;

cout << "C# program:" << endl;
CSharpFactory factory3;
cout << generateProgram(&factory3, "CSharpClass", "Hello C#") << endl;

return a.exec();
}

```

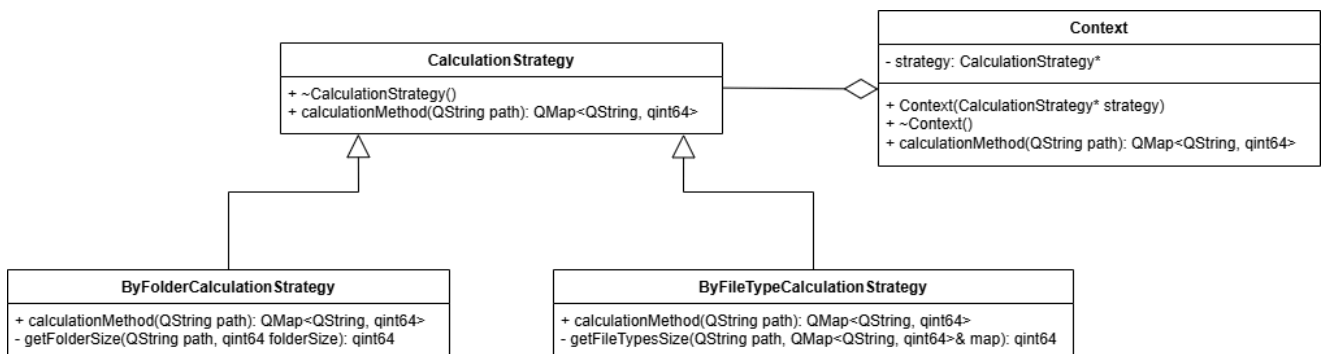
7. Storage Observer

Лабораторная работа № 3.1. Файловый обозреватель

Постановка задачи

Используя паттерн "Стратегия", необходимо реализовать консольное приложение, принцип работы которого следующий: пользователь указывает директорию, для которой требуется вычислить размер содержимого.

Функция, в зависимости от выбранной стратегии, проводит вычисление, где результатом является набор соответствующих данных вида списка файлов и папок (только верхнего уровня) / списка типов файлов, содержащихся в директории, а также занимаемый ими объём в процентах (точность два знака после запятой; если точности не хватает, а размер элемента не равен нулю, требуется это показать, например, в виде "< 0.01%")



calculationstrategy.h


```

#ifndef CALCULATIONSTRATEGY_H
#define CALCULATIONSTRATEGY_H

#include <iostream>
#include <QDir>
#include <QFileInfo>
#include <QMap>
#include <QString>

class CalculationStrategy {
public:
    virtual ~CalculationStrategy() {}

    virtual QMap<QString, qint64> calculationMethod(QString path) = 0;
};

#endif // CALCULATIONSTRATEGY_H

```

byfoldercalculationstrategy.h

```

#ifndef BYFOLDERCALCULATIONSTRATEGY_H
#define BYFOLDERCALCULATIONSTRATEGY_H

#include "calculationstrategy.h"

class ByFolderCalculationStrategy : public CalculationStrategy {
public:
    QMap<QString, qint64> calculationMethod(QString path);

private:
    qint64 getFolderSize(QString path, qint64 folderSize);
};

#endif // BYFOLDERCALCULATIONSTRATEGY_H

```

byfoldercalculationstrategy.cpp

```

#include "byfoldercalculationstrategy.h"

QMap<QString, qint64>
ByFolderCalculationStrategy::calculationMethod(QString path) {
    QDir directory(path);

    if (!directory.exists()) {
        throw std::runtime_error("The specified directory doesn't exist");
    }
}

```

```

    QMap<QString, qint64> directoryMap;

    directory.setFilter(QDir::Files | QDir::Dirs | QDir::NoDotAndDotDot |
QDir::Hidden | QDir::NoSymLinks);
    QFileInfoList list = directory.entryInfoList();

    for (int i = 0; i < list.size(); ++i) {
        QFileInfo fileInfo = list.at(i);

        if (fileInfo.isDir()) {
            qint64 currentFolderSize = 0;
            directoryMap[fileInfo.fileName()] =
getFolderSize(fileInfo.absoluteFilePath(), currentFolderSize);
        } else {
            directoryMap["(Current folder)"] += fileInfo.size();
        }
    }

    return directoryMap;
}

qint64 ByFolderCalculationStrategy::getFolderSize(QString path, qint64
folderSize) {
    QDir directory(path);

    foreach (QFileInfo file, directory.entryInfoList(QDir::Files |
QDir::NoDotAndDotDot | QDir::Hidden | QDir::NoSymLinks)) {
        folderSize += file.size();
    }

    foreach (QFileInfo folder, directory.entryInfoList(QDir::Dirs |
QDir::NoDotAndDotDot | QDir::Hidden | QDir::NoSymLinks)) {
        qint64 currentFolderSize = 0;
        folderSize += getFolderSize(folder.absoluteFilePath(),
currentFolderSize);
    }

    return folderSize;
}

```

byfiletypecalculationstrategy.h

```

#ifndef BYFILETYPECALCULATIONSTRATEGY_H
#define BYFILETYPECALCULATIONSTRATEGY_H

#include "calculationstrategy.h"

class ByFileTypeCalculationStrategy : public CalculationStrategy {
public:

```

```

    QMap<QString, qint64> calculationMethod(QString path);

private:
    QMap<QString, qint64> getFileTypesSize(QString path, QMap<QString,
qint64>& map);
};

#endif // BYFILETYPECALCULATIONSTRATEGY_H

```

byfiletypecalculationstrategy.cpp

```

#include "byfiletypecalculationstrategy.h"

QMap<QString, qint64>
ByFileTypeCalculationStrategy::calculationMethod(QString path) {
    QDir directory(path);

    if (!directory.exists()) {
        throw std::runtime_error("The specified directory doesn't exist");
    }

    QMap<QString, qint64> directoryMap;
    directoryMap = getFileTypesSize(path, directoryMap);

    return directoryMap;
}

QMap<QString, qint64>
ByFileTypeCalculationStrategy::getFileTypesSize(QString path,
QMap<QString, qint64>& map) {
    QDir directory = QDir(path);

    foreach (QFileInfo file, directory.entryInfoList(QDir::Files |
QDir::NoDotAndDotDot | QDir::Hidden | QDir::NoSymLinks)) {
        if (map.contains(file.suffix())) {
            map[file.suffix()] += file.size();
        } else {
            map[file.suffix()] = file.size();
        }
    }

    foreach (QFileInfo folder, directory.entryInfoList(QDir::Dirs |
QDir::NoDotAndDotDot | QDir::Hidden | QDir::NoSymLinks)) {
        getFileTypesSize(folder.absoluteFilePath(), map);
    }

    return map;
}

```

context.h

```
#ifndef CONTEXT_H
#define CONTEXT_H

#include "byfoldercalculationstrategy.h"
#include "byfiletypecalculationstrategy.h"

class Context {
public:
    Context(CalculationStrategy* strategy) {
        this->strategy = strategy;
    }

    ~Context() {
        delete strategy;
    }

    QMap<QString, qint64> calculationMethod(QString path) {
        return strategy->calculationMethod(path);
    }

private:
    CalculationStrategy* strategy;
};

#endif // CONTEXT_H
```

main.cpp

```
#include <QCoreApplication>
#include <iostream>
#include "context.h"

QString getPercent(qint64 directorySize, qint64 currentSize) {
    double percent = static_cast<double>(currentSize) /
static_cast<double>(directorySize) * 100;

    if (percent < 0.01) {
        return "< 0.01";
    } else {
        return QString::number(percent, 'f', 2);
    }
}

qint64 getDirectorySize(QMap<QString, qint64> map) {
    qint64 totalSize = 0;
```

```

    for (auto i = map.cbegin(); i != map.cend(); ++i) {
        totalSize += i.value();
    }

    return totalSize;
}

void printInfo(QMap<QString, qint64> map) {
    qint64 mapSize = getDirectorySize(map);

    for (auto i = map.cbegin(); i != map.cend(); ++i) {
        std::cout << qPrintable(i.key()) << ": " <<
qPrintable(getPersent(mapSize, i.value())) << " %" << std::endl;
    }
}

int main(int argc, char *argv[]) {
    QString path = "../Test";
    Context* context = new Context(new ByFileTypeCalculationStrategy);
    std::cout << "By file types calculation:" << std::endl;
    printInfo(context->calculationMethod(path));
    delete context;

    std::cout << std::endl;

    context = new Context(new ByFolderCalculationStrategy);
    std::cout << "By folder calculation:" << std::endl;
    printInfo(context->calculationMethod(path));
    delete context;

    context = nullptr;

    return 0;
}

```

Лабораторная работа № 3.2. Графическое приложение

Постановка задачи

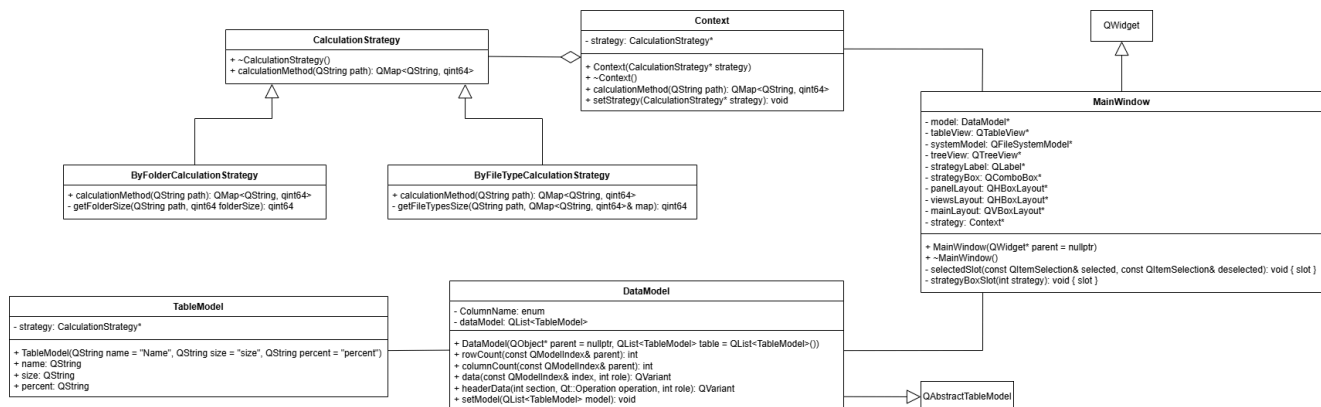
Используя концепцию MVC, требуется разработать модель, данные в которой будут заполняться с помощью реализованных ранее стратегий обхода содержимого папки.

Модель должна содержать, название и занимаемый объём в процентах. Использовать модель файловой системы (QFileSystemModel), отображаемая в двух представлениях: QTreeView (слева) и QTableView (справа).

После реализации модели, достаточно просто сменить тип отображаемой модели у QTableView.

Таким образом, пользователь, выбрав директорию в левой части окна, запускает процесс её сканирования и может увидеть содержимое директории (относительно заданного типа группировки) в правой части окна.

Соответственно, также требуется доработать графический интерфейс, позволив пользователю выбирать способ группировки содержимого.



calculation.h

```

#ifndef CALCULATION_H
#define CALCULATION_H

#include <QDir>
#include <QFileInfo>
#include <QMap>
#include <QString>

class CalculationStrategy {
public:
    virtual ~CalculationStrategy() {}

    virtual QMap<QString, qint64> calculationMethod(QString path) = 0;
};

class ByFolderCalculationStrategy : public CalculationStrategy {
public:
    QMap<QString, qint64> calculationMethod(QString path);

    qint64 getFolderSize(QString path, qint64 folderSize);
};

class ByFileTypeCalculationStrategy : public CalculationStrategy {
public:
    QMap<QString, qint64> calculationMethod(QString path);

    QMap<QString, qint64> getFileTypesSize(QString path, QMap<QString,
qint64>& map);
};

```

```

class Context {
public:
    Context(CalculationStrategy* strategy) {
        this->strategy = strategy;
    }

    ~Context() {
        delete strategy;
    }

    QMap<QString, qint64> calculationMethod(QString path) {
        return strategy->calculationMethod(path);
    }

    void setStrategy(CalculationStrategy* strategy) {
        this->strategy = strategy;
    }

private:
    CalculationStrategy* strategy;
};

#endif // CALCULATION_H

```

calculation.cpp

```

#include "calculation.h"

QMap<QString, qint64>
ByFolderCalculationStrategy::calculationMethod(QString path) {
    QDir directory(path);

    if (!directory.exists()) {
        throw std::runtime_error("The specified directory doesn't exist");
    }

    QMap<QString, qint64> directoryMap;

    directory.setFilter(QDir::Files | QDir::Dirs | QDir::NoDotAndDotDot |
QDir::Hidden | QDir::NoSymLinks);
    QFileInfoList list = directory.entryInfoList();

    for (int i = 0; i < list.size(); ++i) {
        QFileInfo fileInfo = list.at(i);

        if (fileInfo.isDir()) {
            qint64 currentFolderSize = 0;
            directoryMap[fileInfo.fileName()] =

```

```

getFolderSize(fileInfo.absoluteFilePath(), currentFolderSize);
    } else {
        directoryMap["(Current folder)"] += fileInfo.size();
    }
}

return directoryMap;
}

qint64 ByFolderCalculationStrategy::getFolderSize(QString path, qint64
folderSize) {
    QDir directory = QDir(path);

    foreach (QFileInfo file, directory.entryInfoList(QDir::Files |
QDir::NoDotAndDotDot | QDir::Hidden | QDir::NoSymLinks)) {
        folderSize += file.size();
    }

    foreach (QFileInfo folder, directory.entryInfoList(QDir::Dirs |
QDir::NoDotAndDotDot | QDir::Hidden | QDir::NoSymLinks)) {
        qint64 currentFolderSize = 0;
        folderSize += getFolderSize(folder.absoluteFilePath(),
currentFolderSize);
    }

    return folderSize;
}

 QMap<QString, qint64>
ByFileTypeCalculationStrategy::calculationMethod(QString path) {
    QDir directory(path);

    if (!directory.exists()) {
        throw std::runtime_error("The specified directory doesn't exist");
    }

    QMap<QString, qint64> directoryMap;
    directoryMap = getFileTypesSize(path, directoryMap);

    return directoryMap;
}

 QMap<QString, qint64>
ByFileTypeCalculationStrategy::getFileTypesSize(QString path,
 QMap<QString, qint64>& map) {
    QDir directory = QDir(path);

    foreach (QFileInfo file, directory.entryInfoList(QDir::Files |
QDir::NoDotAndDotDot | QDir::Hidden | QDir::NoSymLinks)) {
        if (map.contains(file.suffix())) {

```



```

        map[file.suffix()] += file.size();
    } else {
        map[file.suffix()] = file.size();
    }
}

foreach (QFileInfo folder, directory.entryInfoList(QDir::Dirs |
QDir::NoDotAndDotDot | QDir::Hidden | QDir::NoSymLinks)) {
    getFileTypesSize(folder.absoluteFilePath(), map);
}

return map;
}

```

datamodel.h

```

#ifndef DATAMODEL_H
#define DATAMODEL_H

#include <QObject>
#include <QList>
#include <QString>
#include <QAbstractTableModel>

class TableModel {
public:
    TableModel(QString name = "Name", QString size = "Size", QString
percent = "Percent") : name(name), size(size), percent(percent) {}

    QString name;
    QString size;
    QString percent;
};

class DataModel : public QAbstractTableModel {
    Q_OBJECT

public:
    DataModel(QObject* parent = nullptr, QList<TableModel> table =
QList<TableModel>());

    int rowCount(const QModelIndex& parent) const;
    int columnCount(const QModelIndex& parent) const;

    QVariant data(const QModelIndex& index, int role) const;
    QVariant headerData(int section, Qt::Orientation orientation, int
role) const;

    void setModel(QList<TableModel> model);

```

```

private:
    enum ColumnName {
        NAME = 0,
        SIZE,
        PERCENT
    };

    QList<TableModel> dataModel;
};

#endif // DATAMODEL_H

```

datamodel.cpp

```

#include "datamodel.h"

DataModel::DataModel(QObject* parent, QList<TableModel> table) :
    QAbstractTableModel(parent) {
    dataModel = table;
}

int DataModel::rowCount(const QModelIndex& parent) const {
    Q_UNUSED(parent);

    return dataModel.count();
}

int DataModel::columnCount(const QModelIndex& parent) const {
    Q_UNUSED(parent);

    return PERCENT + 1;
}

QVariant DataModel::data(const QModelIndex& index, int role) const {
    if (!index.isValid() || dataModel.count() <= index.row() || (role !=
Qt::DisplayRole && role != Qt::EditRole)) {
        return QVariant();
    }

    switch (index.column()) {
    case NAME:
        return dataModel[index.row()].name;
        break;

    case SIZE:
        return dataModel[index.row()].size;
        break;
    }
}

```

```

        case PERCENT:
            return dataModel[index.row()].percent;
            break;

        default:
            break;
    }

    return QVariant();
}

QVariant DataModel::headerData(int section, Qt::Orientation orientation,
int role) const {
    if (role == Qt::DisplayRole && orientation == Qt::Horizontal) {
        switch (section) {
            case NAME:
                return "Name";

            case SIZE:
                return "Size";

            case PERCENT:
                return "Percent";
        }
    }

    return QVariant();
}

void DataModel::setModel(QList<TableModel> model) {
    dataModel = model;

    emit layoutChanged();
}

```

mainwindow.h

```

#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QWidget>
#include <QTableView>
#include <QTreeView>
#include <QHBoxLayout>
#include <QVBoxLayout>
#include <QFileSystemModel>
#include <QLabel>
#include <QComboBox>

```

```

#include "calculation.h"
#include "datamodel.h"

class MainWindow : public QWidget {
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();

private:
    DataModel* model;
    QTableView* tableView;
    QFileSystemModel* systemModel;
    QTreeView* treeView;

    QLabel* strategyLabel;
    QComboBox* strategyBox;

    QHBoxLayout* panelLayout;
    QHBoxLayout* viewsLayout;
    QVBoxLayout* mainLayout;

    Context* strategy;

private slots:
    void selectedSlot(const QItemSelection& selected, const
QItemSelection& deselected);
    void strategyBoxSlot(int strategy);
};

#endif // MAINWINDOW_H

```

mainwindow.cpp

```

#include <QDebug>
#include "mainwindow.h"

MainWindow::MainWindow(QWidget* parent) : QWidget(parent) {
    setWindowTitle("Storage Observer");
    this->strategy = new Context(new ByFolderCalculationStrategy);
    model = new DataModel(this);
    setWindowIcon(QIcon("Storage Observer.png"));

    systemModel = new QFileSystemModel(this);
    systemModel->setFilter(QDir::NoDotAndDotDot | QDir::AllDirs);
    systemModel->setRootPath(QDir::homePath());

    treeView = new QTreeView(this);

```

```

treeView->setModel(systemModel);

tableView = new QTableView(this);
tableView->setModel(model);

strategyLabel = new QLabel("Select the typy of the review", this);
strategyBox = new QComboBox(this);
strategyBox->addItem({"By folder", "By file type"});

panelLayout = new QHBoxLayout();
if (panelLayout != nullptr) {
    panelLayout->addStretch(1);
    panelLayout->addWidget(strategyLabel);
    panelLayout->addWidget(strategyBox);
}

viewsLayout = new QHBoxLayout();
if (viewsLayout != nullptr) {
    viewsLayout->addWidget(treeView);
    viewsLayout->addWidget(tableView);
    viewsLayout->setStretchFactor(tableView, 1);
}

mainLayout = new QVBoxLayout(this);
if (mainLayout != nullptr) {
    mainLayout->addLayout(panelLayout);
    mainLayout->addLayout(viewsLayout);
}

connect(treeView->selectionModel(),
&QItemSelectionModel::selectionChanged, this, &MainWindow::selectedSlot);
connect(strategyBox, qOverload<int>(&QComboBox::currentIndexChanged),
this, &MainWindow::strategyBoxSlot);
}

MainWindow::~MainWindow() {
    delete strategy;
}

QString getPercent(qint64 currentSize, qint64 directorySize) {
    double percent = static_cast<double>(currentSize) /
static_cast<double>(directorySize) * 100;

    if (percent < 0.01) {
        return "< 0.01";
    } else {
        return QString::number(percent, 'f', 2) + " %";
    }
}

```

```

QString convertKiloBytes(qint64 size) {
    double kiloBytes = static_cast<double>(size) / 1024;

    return QString::number(kiloBytes, 'f', 2) + " Kb";
}

qint64 getDirectorySize(QMap<QString, qint64> map) {
    qint64 totalSize = 0;

    for (auto i = map.cbegin(); i != map.cend(); ++i) {
        totalSize += i.value();
    }

    return totalSize;
}

QList<TableModel> fillTable(QMap<QString, qint64> map) {
    qint64 totalSize = getDirectorySize(map);
    QList<TableModel> result;

    for (auto i = map.cbegin(); i != map.cend(); ++i) {
        result.append(TableModel(i.key(), convertKiloBytes(i.value()),
getPercent(i.value(), totalSize)));
    }

    return result;
}

void MainWindow::selectedSlot(const QItemSelection& selected, const
QItemSelection& deselected) {
    Q_UNUSED(selected);
    Q_UNUSED(deselected);

    QModelIndex index = treeView->selectionModel()->currentIndex();

    if (index.isValid()) {
        model->setModel(fillTable(strategy->calculationMethod(systemModel-
>filePath(index))));
    }
}

void MainWindow::strategyBoxSlot(int strategy) {
    switch (strategy) {
    case 0:
        this->strategy->setStrategy(new ByFolderCalculationStrategy);
        break;

    case 1:
        this->strategy->setStrategy(new ByFileTypeCalculationStrategy);
        break;
    }
}

```

```
}

QModelIndex index = treeView->selectionModel()->currentIndex();
model->setModel(fillTable(this->strategy-
>calculationMethod(systemModel->filePath(index))));
}
```

main.cpp

```
#include "mainwindow.h"

#include <QApplication>

int main(int argc, char *argv[]) {
    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    return a.exec();
}
```