Технологии программирования

Лекция 6 | Паттерны проектирования

Лектор: Киселев Станислав Дмитриевич

Создаем качественный код

Качественный код – инвестиция в настоящее и будущее

Когда мы создаём сложные и многозадачные приложения, важно уделять особенное внимание упорядоченной, надёжной и понятной структуре кода

В этом нам помогут паттерны проектирования – они как готовые рецепты, которые помогают находить решения для распространённых задач

Создаем качественный код

Паттерны программирования

И основные типы

Мы здесь

Креационные паттерны

Абстрактная фабрика, Строитель, Фабричный метод, Прототип, Одиночка

Структурные паттерны

Адаптер, Мост, Компоновщик, Декоратор, Фасад, Легковес, Заместитель

Поведенческие паттерны

Цепочка обязанностей, Команда, Итератор, Посредник, Снимок, Наблюдатель, Состояние, Стратегия, Шаблонный метод, Посетитель, Интерпретатор

Test-Driven Design (TDD)

И связь с паттернами программирования

Заключение

И что еще можно изучить по теме паттернов

Паттерны программирования

Паттерн – решение часто встречающейся задачи в программировании, оформленное в виде повторяемой структуры

Пример: «Одиночка» обеспечивает, чтобы объект создавался только один раз

Антипаттерн – ошибка проектирования, которая приводит к усложнению поддержки и развития кода

Пример: «Божественный объект» — класс, который знает всё и делает всё

Антипаттерны | Божественный Объект

Он содержит слишком много функциональности, а его изменения влияют на большую часть системы

```
class GodObject {
public:
    void saveUserToDatabase(const
std::string& user) { /*...*/ }

    void sendEmail(const std::string&
email) { /*...*/ }

    void calculateStatistics() { /*...*/ }

    void generateReport() { /*...*/ }
    // и еще 100 методов...
};
```



```
class UserManager {
public:
    void saveUserToDatabase(const std::string& user) { /*...*/ }
};

class EmailService {
public:
    void sendEmail(const std::string& email) { /*...*/ }
};

class StatisticsCalculator {
public:
    void calculateStatistics() { /*...*/ }
};

class ReportGenerator {
public:
    void generateReport() { /*...*/ }
};
```

Паттерны программирования

Паттерн – решение часто встречающейся задачи в программировании, оформленное в виде повторяемой структуры

Пример: «Одиночка» обеспечивает, чтобы объект создавался только один раз

Применимость

- **Улучшение архитектуры кода:**
  Делает систему модульной и устойчивой к изменениям
- **Улучшение читаемости:**
  Чётко обозначает роль и поведение компонентов
- **Упрощение тестирования:**
  Повышает тестируемость кода за счёт строгой структуры
- **Снижение дублирования:**
  Паттерны способствуют переиспользованию кода

Паттерны программирования

Паттерны сложны, но

Паттерны – база создания рабочего продукта

А также они помогут в прохождении технических собеседований

Принципы паттернов программирования

1. **Разделение обязанностей** (Separation of Concerns)
2. **Ослабление связей** (Loose Coupling)
3. **Повторное использование кода** (Reusability)
4. **Инкапсуляция изменения** (Encapsulation of Change)
5. **Открытость для расширения, закрытость для изменений** (Open/Closed Principle)
6. **Снижение дублирования кода** (DRY — Don’t Repeat Yourself)

1. Разделение обязанностей (Separation of Concerns)

! Каждый компонент системы должен заниматься своей задачей

Паттерны помогают изолировать разные аспекты логики:

- Логика создания объектов (порождающие паттерны)
- Организация структуры системы (структурные паттерны)
- Поведение компонентов (поведенческие паттерны)

Применение в программировании:
Разделение кода на модули или слои с четкими обязанностями
(например, бизнес-логика, работа с данными, пользовательский интерфейс)

Паттерны: Адаптер, Фасад, Наблюдатель, Стратегия и другие

2. Ослабление связей (Loose Coupling)

! Компоненты должны быть минимально зависимыми друг от друга, чтобы изменения в одной части системы не затрагивали другие

Например, электронные приборы для работы подключаются к сети в розетку:

- Розетка может принимать устройства с разной мощностью, напряжением или потребляемым током
- Устройство может подключаться к различным источникам питания (розеткам)

Розетки и прибор взаимодействуют через стандартный интерфейс (форма вилки, отверстия розетки), но **не зависят** от конкретных характеристик друг друга

3. Повторное использование кода (Reusability)

! Код должен быть хорошо структурирован, с четкими интерфейсами и минимальными зависимостями универсальных компонентов – которые могут быть переиспользованы в разных частях проекта или в других проектах

В конструкторе LEGO один и тот же кубик или детальку можно использовать для построения замка, машины или самолета

Кубик универсален, и вы можете использовать его в разных конструкциях, минимизируя затраты на создание новых деталей

4. Инкапсуляция изменения (Encapsulation of Change)

! Часто меняющиеся аспекты системы должны быть изолированы, чтобы изменения не касались остальных частей

Предположим, у вас есть автомобиль, который обслуживаете в одном и том же салоне

При смене директора функции салона останутся прежними, и при отсутствии других изменений для вас, как для клиента, ничего не поменяется

5. Открытость для расширения, закрытость для изменений

! Структура и работа компонентов кода должны давать возможность добавлять новые функции без изменения существующего кода

Например, как меню в кафе:

Если добавить новое блюдо (например, десерт), можно просто включить его в меню, не убирая и не изменяя остальное

Основные категории меню (закуски, горячие блюда, напитки) остаются неизменными, так что система не ломается

Типы паттернов программирования

Паттерны отличаются по уровню сложности, детализации и охвата проектируемой системы

Низкоуровневые и простые паттерны **– идиомы**
Они не универсальны, поскольку применимы только в рамках одного языка программирования

Самые универсальные – архитектурные паттерны,
которые можно реализовать практически на любом языке

Они нужны для проектирования всей программы, а не отдельных её элементов

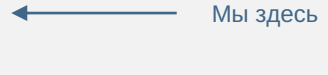
- **Креационные паттерны**
  Беспокоятся о гибком создании объектов без внесения в программу лишних зависимостей
- **Структурные паттерны**
  Показывают различные способы построения связей между объектами
- **Поведенческие паттерны**
  Заботятся об эффективной коммуникации между объектами

Паттерны программирования

И основные типы

Креационные паттерны

Абстрактная фабрика, Строитель, Фабричный метод, Прототип, Одиночка



Структурные паттерны

Адаптер, Мост, Компоновщик, Декоратор, Фасад, Легковес, Заместитель

Поведенческие паттерны

Цепочка обязанностей, Команда, Итератор, Посредник, Снимок, Наблюдатель, Состояние, Стратегия, Шаблонный метод, Посетитель, Интерпретатор

Test-Driven Design (TDD)

И связь с паттернами программирования

Заключение

И что еще можно изучить по теме паттернов

Фабричный метод (Factory Method)

! Создавайте интерфейс создания связанных объектов

Есть некая фабрика по производству автомобилей

На ней выпускают разные модели машин (седан, внедорожник, кабриолет)
Каждая модель автомобиля собирается по-разному

Фабричный метод (Factory Method)

Применяется для динамического выбора класса создаваемого объекта
Например, при загрузке изображений в разных форматах (JPEG, PNG)

```
class Product {
public:
    virtual void show() = 0;
};
```

```
class Creator {
public:
    virtual Product* factoryMethod() = 0;
};
```

```
int main() {
    Creator* creator = new ConcreteCreator();
    Product* product = creator->factoryMethod();
    product->show();
    delete product;
    delete creator;
    return 0;
}
```

```
class ConcreteProduct : public Product {
public:
    void show() override {
        std::cout << "Concrete Product" << std::endl;
    }
};
```

```
class ConcreteCreator : public Creator {
public:
    Product* factoryMethod() override {
        return new ConcreteProduct();
    }
};
```

Фабричный метод | Применимость

- Когда нужно отделить создание объектов от их использования
- Когда тип создаваемого объекта определяется во время выполнения программы
- Когда нужно создать семейство связанных объектов
- Когда нужно скрыть сложность создания объекта

Фабричный метод как пример принципа Разделения обязанностей (1)

Фабричный метод разделяет создание объектов и их использование

ДО

```
class Report:
    def generate(self):
        print("Generating report...")

    def save_to_file(self, filename):
        print(f"Saving to {filename}")
```

ПОСЛЕ

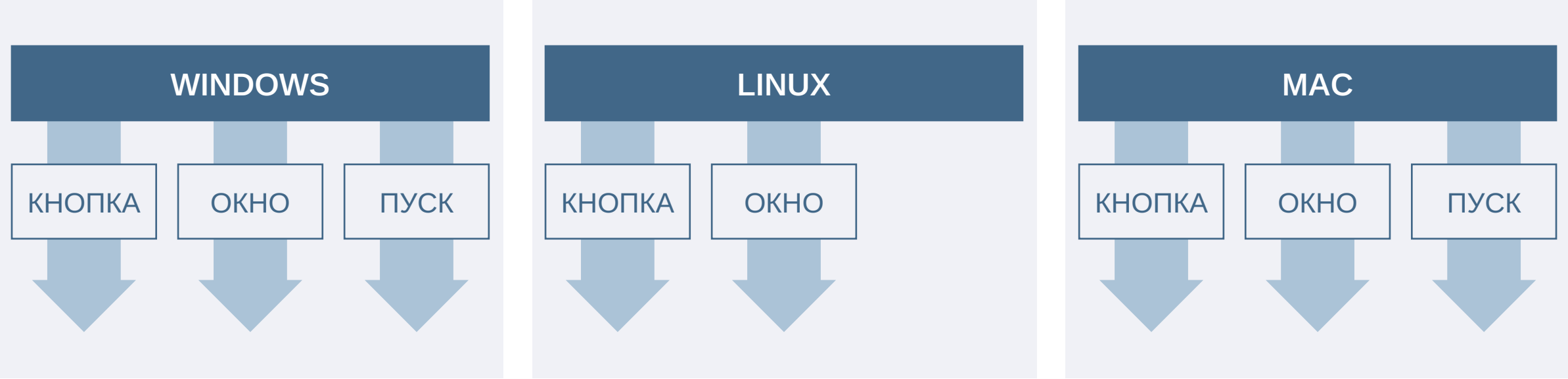
```
class ReportGenerator:
    def generate(self):
        return "Report Content"

class FileSaver:
    def save(self, content, filename):
        with open(filename, 'w') as file:
            file.write(content)

report = ReportGenerator().generate()
FileSaver().save(report, "report.txt")
```

Абстрактная Фабрика (Abstract Factory)

! Создавайте единую точку создания связанных объектов

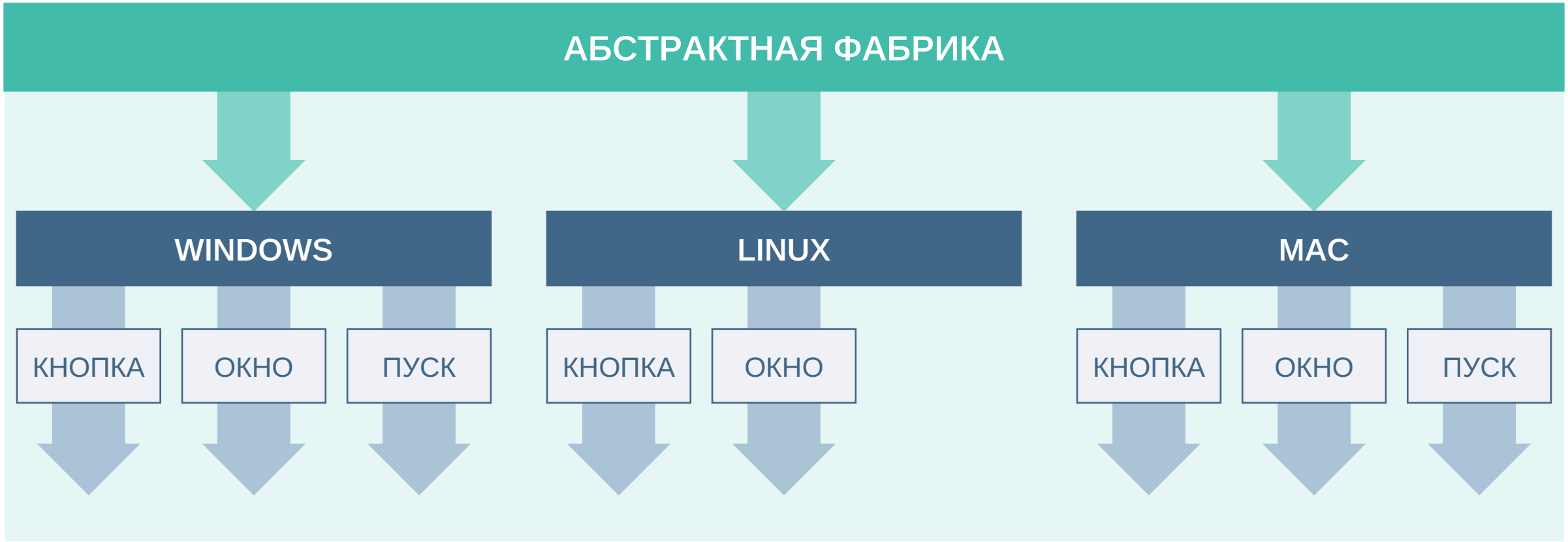
Представьте, что мы создаем интерфейс

У нас есть различные системы (Windows, Mac, Linux), которые создают интерфейс
Независимо от платформы, у нас всегда определенный набор элементов: кнопка, диалоговое окно, пуск



Абстрактная Фабрика (Abstract Factory)

! Используйте единую точку создания связанных объектов



Абстрактная Фабрика (Abstract Factory)

Пример:
GUI-библиотека, где нужно создать кнопки и текстовые поля для разных платформ (Windows, macOS)

```
class Button {
public:
    virtual void render() = 0;
};
```

```
class GUIFactory {
public:
    virtual Button* createButton() = 0;
};
```

```
class WindowsButton : public Button {
public:
    void render() override {
        std::cout << "Rendering Windows Button" <<
std::endl;
    }
};
```

```
class WindowsFactory : public GUIFactory {
public:
    Button* createButton() override {
        return new WindowsButton();
    }
};
```

```
class MacButton : public Button {
public:
    void render() override {
        std::cout << "Rendering Mac Button" << std::endl;
    }
};
```

```
class MacFactory : public GUIFactory {
public:
    Button* createButton() override {
        return new MacButton();
    }
};
```

```
int main() {
    GUIFactory* factory = new WindowsFactory();
    Button* button = factory->createButton();
    button->render();
    delete button;
    delete factory;
    return 0;
}
```

Абстрактная Фабрика | Применимость

- **Создание нескольких взаимосвязанных объектов**
  Абстрактная фабрика позволяет централизовать этот процесс и обеспечить согласованность (например, элементы пользовательского интерфейса для разных платформ)

- **Конфигурирование системы на этапе выполнения**
  Можно динамически изменять поведение системы, переключаясь между различными фабриками

- **Изоляция от конкретных реализаций**
  Абстрагирует клиента от конкретных классов продуктов, делая код более гибким и легко изменяемым

Строитель (Builder)

! Пошагово создавайте сложные объекты, отделяя процесс конструирования от его представления

Предположим, что необходимо построить 3 домика примерно одного стиля и вида

Домик с гаражом

Домик с бассейном

Домик с садом

Строитель (Builder)

Лучшим вариантом было бы описать все нужные для постройки элементы (фундамент, материал, высоту стен, варианты пристроек и тд.), задокументировать, и в процессе менять только желаемые:

Строитель (Builder)

Так, мы можем создать класс «Строитель», в котором соберем все нужные методы, используемые при создании объектов

```
class Product {
private:
    std::string part1, part2;
public:
    void setPart1(const std::string& p) { part1 = p; }
    void setPart2(const std::string& p) { part2 = p; }
    void show() { std::cout << part1 << " " << part2 << std::endl; }
};
```

```
class Builder {
public:
    virtual void buildPart1() = 0;
    virtual void buildPart2() = 0;
    virtual void buildPart3() = 0;
    virtual Product* getResult() = 0;
};
```

```
class ConcreteBuilder : public Builder {
private:
    Product* product;
public:
    ConcreteBuilder() { product = new Product(); }
    void buildPart1() override { product->setPart1("Part 1"); }
    void buildPart2() override { product->setPart2("Part 2"); }
    Product* getResult() override { return product; }
};
```

```
int main() {
    Builder* builder = new ConcreteBuilder();
    builder->buildPart1();
    builder->buildPart2();
    Product* product = builder->getResult();
    product->show();
    delete product;
    delete builder;
    return 0;
}
```

Строитель | Применимость

- **Сложные объекты**
  Когда объект имеет много опциональных параметров или когда процесс создания объекта требует определенной последовательности шагов

- **Различные представления одного объекта**
  Когда нужно создавать разные вариации одного и того же объекта

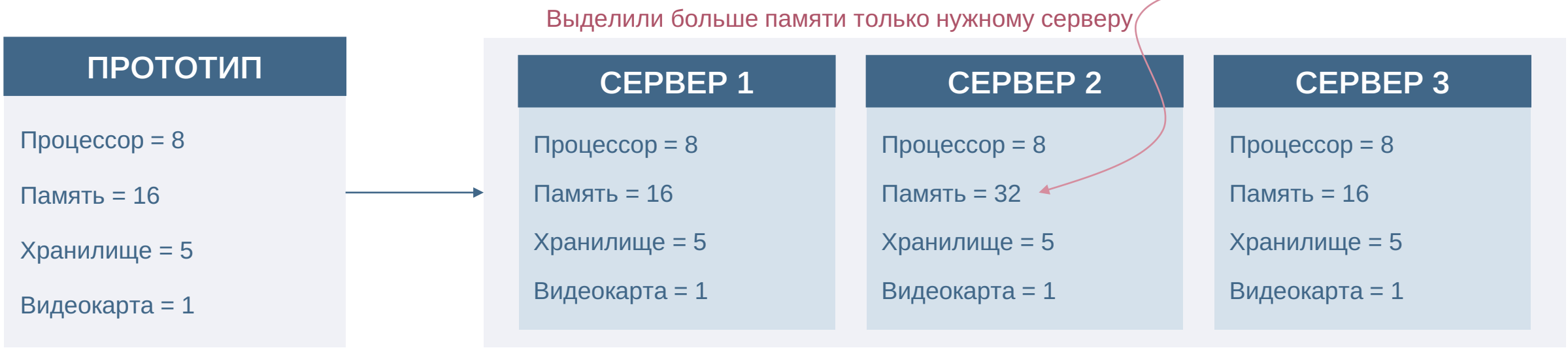
- **Отделение конструкции от представления**
  Когда нужно отделить логику создания объекта от его представления

Прототип (Prototype)

! Если нужно много похожих объектов – создавайте новые путем копирования прототипа

Например, в компании купили 3 новых сервера
Вместо того, чтобы вручную создавать новую конфигурацию с нуля, стоит создать «шаблонную» конфигурацию (прототип) и клонировать ее для каждого нового сервера

Причем можно изменять отдельные параметры в клонированной конфигурации, не затрагивая оригинал



Прототип (Prototype)

Такая схема может быть при копировании документов в текстовом редакторе вместе с их стилями

```
class Prototype {
public:
    virtual Prototype* clone() = 0;
    virtual void show() = 0;
};
```

```
    class ConcretePrototype : public
Prototype {
    public:
        Prototype* clone() override {
            return new ConcretePrototype();
        }

        void show() override {
            std::cout << "Concrete Prototype"
<< std::endl;
        }
    };
```

```
    int main() {
        ConcretePrototype* prototype = new
ConcretePrototype();
        Prototype* clone = prototype->clone();
        clone->show();
        delete prototype;
        delete clone;
        return 0;
    }
```

Прототип | Применимость

- **Сложные объекты**
  Когда создание объекта с нуля требует большого количества ресурсов или сложных вычислений

- **Множественные похожие объекты**
  Когда нужно создать множество объектов, которые отличаются друг от друга лишь небольшими деталями

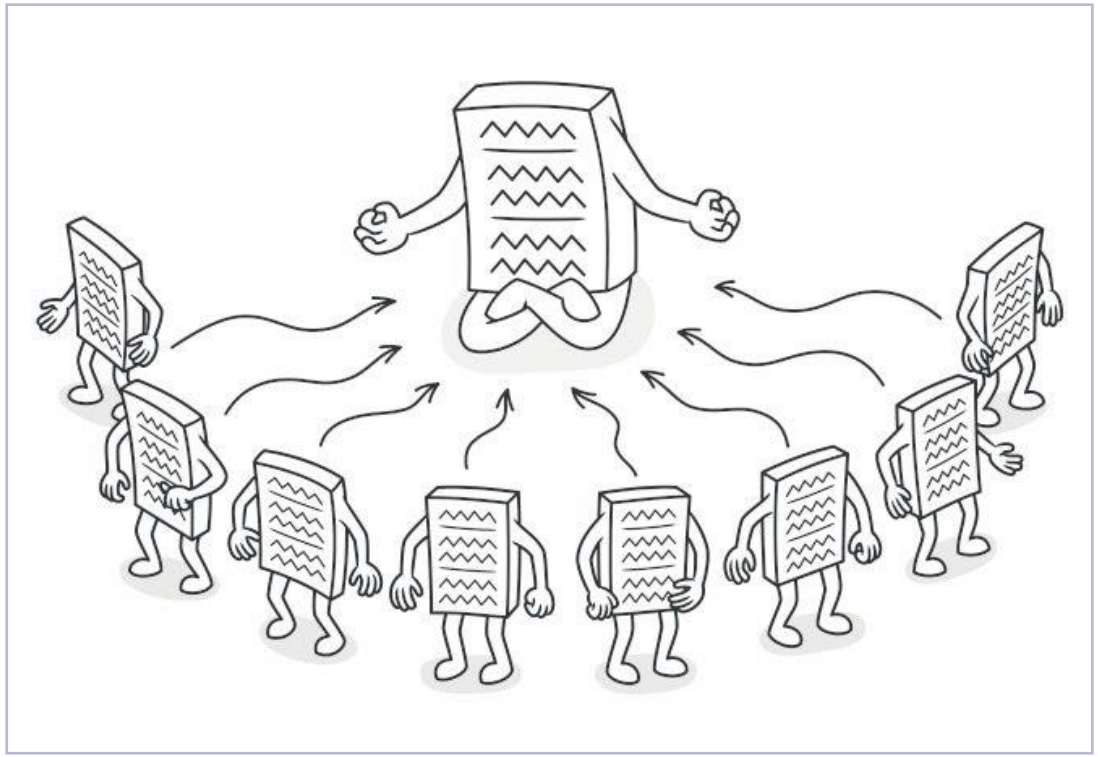
- **Гибкая конфигурация**
  Когда требуется возможность динамически изменять структуру объекта

Одиночка (Singleton)

! Если необходим глобальный доступ к классу, то пусть у этого класса будет только один экземпляр, который и предоставляет точку доступа к нему

Почти в любом приложении есть настройки, которые:
а) должны быть доступны из любой части приложения
б) они должны быть едиными для всей системы

Вместо того, чтобы создавать множество объектов настроек, создаем один объект-одиночку, который будет хранить все настройки и предоставлять к ним доступ



Одиночка (Singleton)

Подходит для управления доступом к общему ресурсу, например, логгеру или пулу соединений с базой данных

```
class Singleton {
public:
    Singleton(const Singleton&) = delete;
    Singleton& operator=(const Singleton&) =
delete;

    static Singleton& getInstance() {
        std::call_once(initFlag, []() {
instance.reset(new Singleton()); });
        return *instance;
    }

    // Пример метода
    void doSomething() const {
        std::cout << "Singleton instance doing
something!" << std::endl;
    }
```

```
private:
    Singleton() = default;
    ~Singleton() = default;

    static std::unique_ptr<Singleton> instance;
    static std::once_flag initFlag;
};




// Инициализация статических
переменных
std::unique_ptr<Singleton> Singleton::instance
= nullptr;
std::once_flag Singleton::initFlag;
```

```
int main() {
    // Получаем единственный экземпляр
класса Singleton
    Singleton& singleton = Singleton::getInstance();
    singleton.doSomething();

    // Попытка создать второй экземпляр
(не получится)
    // Singleton secondSingleton = singleton; //
Ошибка компиляции

    return 0;
}
```

Одиночка | Применимость

- **Глобальные ресурсы**
  Когда нужно контролировать доступ к единственному экземпляру ресурса (например, база данных, логгер)

- **Конфигурации**
  Когда нужны глобальные настройки, доступные из любой части приложения

- **Счетчики и статистики**
  Когда нужно отслеживать общее состояние приложения

Паттерны программирования

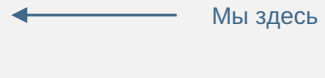
И основные типы

Креационные паттерны

Абстрактная фабрика, Строитель, Фабричный метод, Прототип, Одиночка

Структурные паттерны

Адаптер, Мост, Компоновщик, Декоратор, Фасад, Легковес, Заместитель



Поведенческие паттерны

Цепочка обязанностей, Команда, Итератор, Посредник, Снимок, Наблюдатель, Состояние, Стратегия, Шаблонный метод, Посетитель, Интерпретатор

Test-Driven Design (TDD)

И связь с паттернами программирования

Заключение

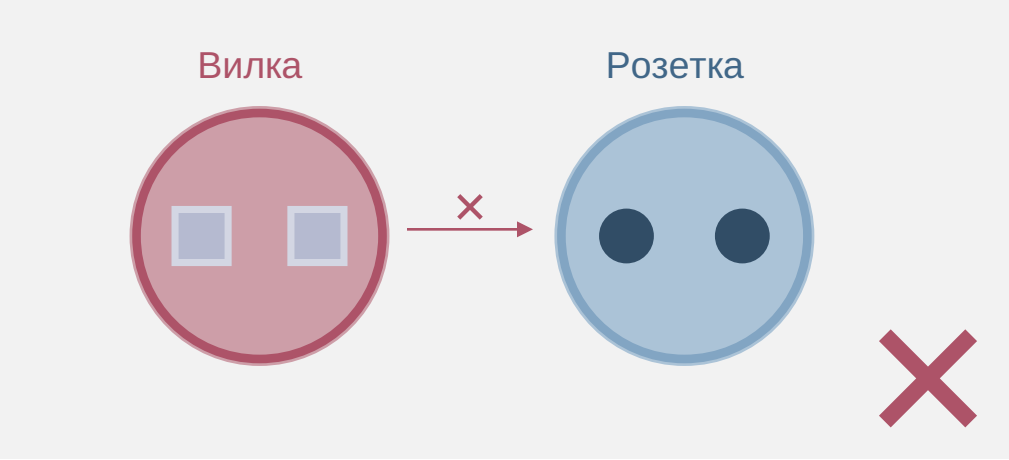
И что еще можно изучить по теме паттернов

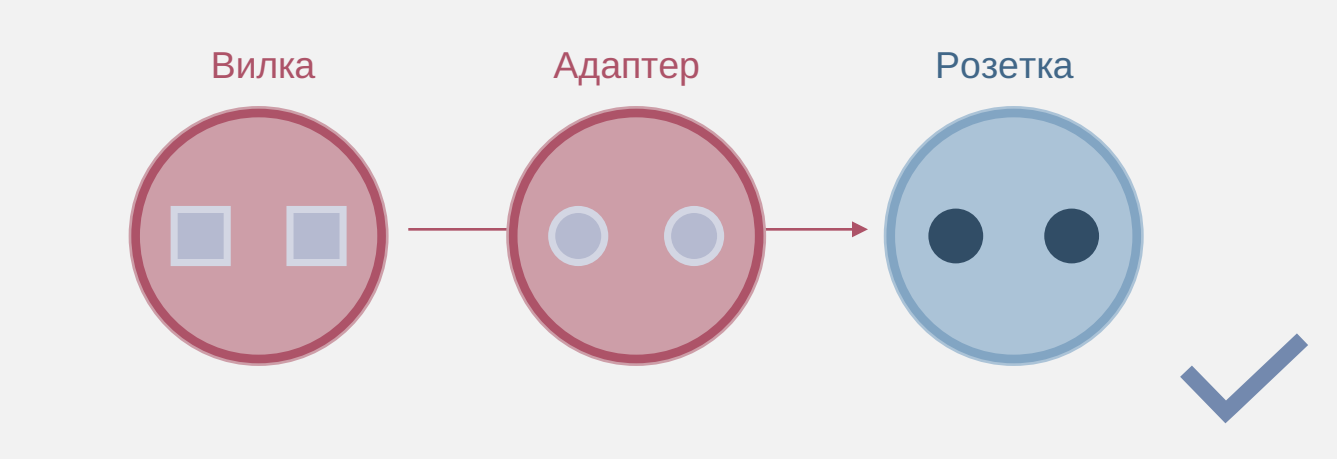
Адаптер (Adapter)

! Чтобы не переделывать существующий сложный класс под новые требования к интерфейсу, напишите другой объект, переопределяющий необходимые методы

Как в ситуации, когда при покупке нового китайского устройства тот пришел с неподходящей, китайской вилкой

Для того, чтобы использовать блок питания в РФ необходимо адаптировать его для нашей розетки через переходник (адаптер)





Адаптер (Adapter)

Используется для интеграции старого API с новым кодом, например, при переходе от устаревшей библиотеки к современной

```
class Target {
public:
    virtual void request() = 0;
};
```

```
class Adaptee {
public:
    void specificRequest() {
        std::cout << "Specific request" <<
std::endl;
    }
};
```

```
class Adapter : public Target {
private:
    Adaptee* adaptee;
public:
    Adapter(Adaptee* a) : adaptee(a) {}
    void request() override {
        adaptee->specificRequest();
    }
};
```

```
int main() {
    Adaptee* adaptee = new Adaptee();
    Target* target = new
Adapter(adaptee);
    target->request();
    delete target;
    delete adaptee;
    return 0;
}
```

Адаптер | Применимость

- **Несовместимые интерфейсы**
  Когда нужно использовать существующий класс, но его интерфейс не соответствует потребностям

- **Изменение интерфейса существующего класса**
  Когда необходимо изменить интерфейс существующего класса, но нет возможности изменить сам класс

- **Использование сторонних библиотек**
  Когда нужно интегрировать сторонние библиотеки с существующим кодом

Декоратор (Decorator)

! Для добавления нового функционала, не изменяя текущий код, создайте обертку для этого кода

Представим, что у нас есть робот-доставщик, который выполняет простую задачу: доставить посылку из точки А в точку Б

БАЗОВАЯ ФУНКЦИОНАЛЬНОСТЬ		ДЕКОРАТОРЫ ДЛЯ УЛУЧШЕНИЙ
Алгоритм планирования маршрута с учётом расстояния между точками Робот всегда выбирает кратчайший путь	+	• Добавляем учёт пробок • Добавляем учёт состояния батареи • Добавляем учёт погоды • Добавляем учёт приоритетов • …

Декоратор (Decorator)

Используется для добавления функциональности объекту, например, при добавлении шифрования к существующему потоку данных

```
class Component {
public:
    virtual void operation() = 0;
};
```

```
class Decorator : public Component {
protected:
    Component* component;
public:
    Decorator(Component* comp) : component(comp) {}
    void operation() override {
        component->operation();
    }
};
```

```
int main() {
    Component* component = new
ConcreteComponent();
    Component* decorated = new
ConcreteDecorator(component);
    decorated->operation();
    delete component;
    delete decorated;
    return 0;
}
```

```
class ConcreteComponent : public Component {
public:
    void operation() override {
        std::cout << "Concrete Component" << std::endl;
    }
};
```

```
class ConcreteDecorator : public Decorator {
public:
    ConcreteDecorator(Component* comp) : Decorator(comp) {}
    void operation() override {
        Decorator::operation();
        std::cout << "Additional functionality" << std::endl;
    }
};
```

Декоратор | Применимость

- **Динамическое добавление функциональности**
  Когда нужно добавлять новые возможности объекту во время выполнения программы

- **Гибкая конфигурация**
  Когда требуется гибко настраивать объекты, комбинируя различные аспекты их поведения

- **Избежание множественного наследования**
  Когда использование множественного наследования усложняет структуру программы

Декоратор как пример принципа Инкапсуляции изменения (4)

Декоратор позволяет добавлять функциональность объекту без изменения его кода

```
trait Component {
    fn operation(&self) -> String;
}

struct ConcreteComponent;
impl Component for ConcreteComponent {
    fn operation(&self) -> String {
        "Base Component".to_string()
    }
}

struct Decorator {
    component: Box<dyn Component>,
}
```

```
impl Component for Decorator {
    fn operation(&self) -> String {
        format!("{} + Decorator",
self.component.operation())
    }
}

fn main() {
    let component = ConcreteComponent;
    let decorated = Decorator { component:
Box::new(component) };
    println!("{}", decorated.operation());
}
```

Мост (Bridge)

! Для расширяемости делите код на две части: «что делать» (абстракция) **и «как работает»** (реализация)

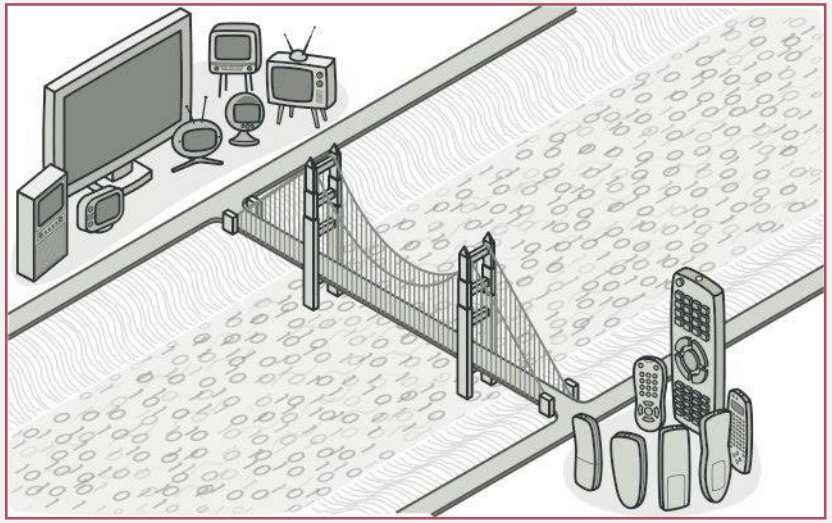
Можно представить на примере пульта управления и устройств

Пульт – абстракция, которая имеет кнопки для включения, выключения или регулировки,
а устройства (например, телевизор, кондиционер или музыкальная колонка) – это реализация

Если это умный пульт, который подключается к разным устройствам, то легко можно заменить телевизор на кондиционер, не меняя сам пульт

Или же всегда можно усовершенствовать пульт, например, добавив голосовое управление, при этом не трогая рабочие устройства

Это позволяет создавать универсальные решения,
где абстракция и реализация развиваются независимо



Мост (Bridge)

Подходит для разделения интерфейса и реализации, например, для абстрактной системы управления устройствами (мониторы, проекторы) с разными протоколами

```
class Implementor {
public:
    virtual void operationImpl() = 0;
};
```

```
class RefinedAbstraction : public Abstraction {
public:
    RefinedAbstraction(Implementor* imp) :
Abstraction(imp) {}
    void operation() override {
        implementor->operationImpl();
    }
};
```

```
int main() {
    Implementor* implementor = new
ConcreteImplementorA();
    Abstraction* abstraction = new
RefinedAbstraction(implementor);
    abstraction->operation();
    delete abstraction;
    delete implementor;
    return 0;
}
```

```
class ConcreteImplementorA : public
Implementor {
public:
    void operationImpl() override {
        std::cout << "Concrete Implementor A" <<
std::endl;
    }
};
```

```
class Abstraction {
protected:
    Implementor* implementor;
public:
    Abstraction(Implementor* imp) :
implementor(imp) {}
    virtual void operation() = 0;
};
```

Мост | Применимость

- Когда у вас есть две независимые иерархии классов, которые нужно комбинировать
- Когда вы хотите отделить абстракцию от реализации
- Когда вы ожидаете, что обе иерархии будут часто изменяться

Легковес (Flyweight)

! Если система предполагает большое количество объектов, то оберни их в небольшой класс и напиши к элементам Адаптер

Представьте себе магазин с книгами

Про каждую книгу мы знаем ее название, автора, жанр и год издания

Если в магазине несколько десятков экземпляров одной и той же книги, и у каждого будут эти одинаковые данные, то оптимальнее будет сохранить их только один раз, а в каждом экземпляре отмечать только уникальную информацию (например, номер экземпляра или состояние)

КНИГА N	ЭКЗЕМПЛЯР	ЭКЗЕМПЛЯР	ЭКЗЕМПЛЯР	ЭКЗЕМПЛЯР
Автор	Номер экземпляра	Номер экземпляра	Номер экземпляра	Номер экземпляра
Название	Состояние	Состояние	Состояние	Состояние
Жанр				
Год издания				

Легковес (Flyweight)

Применяется для оптимизации использования памяти, например, при отображении множества объектов в игре (деревья, камни) с общими данными

```
class Flyweight {
private:
    char intrinsicState;

public:
    Flyweight(char state) : intrinsicState(state)
{}
    void display(int extrinsicState) const {
        std::cout << "Character: " <<
intrinsicState << ", Font size: " <<
extrinsicState << "\n";
    }
};
```

```
auto main() {
    // Usage:
    FlyweightFactory* factory = new
FlyweightFactory();
    Flyweight* a = factory->getFlyweight('a');
    Flyweight* b = factory->getFlyweight('b');
    a->display(12);
    b->display(14);
    delete factory;
}
```

```
class FlyweightFactory {
private:
    std::unordered_map<char, Flyweight*> flyweights;

public:
    Flyweight* getFlyweight(char key) {
        if (flyweights.find(key) == flyweights.end()) {
            flyweights[key] = new Flyweight(key);
        }
        return flyweights[key];
    }

    ~FlyweightFactory() {
        for (auto& pair : flyweights) {
            delete pair.second;
        }
    }
};
```

Легковес | Применимость

- **Большое количество мелких объектов**
  Когда приложение создает большое количество объектов с похожими свойствами

- **Ограниченная память**
  Когда нужно оптимизировать использование памяти

- **Частое создание и удаление объектов**
  Когда объекты создаются и удаляются динамически

Фасад (Facade)

! Оборачивайте сложную систему классов простой интерфейс, скрывая ее внутреннюю структуру

Представьте момент, когда вы включаете компьютер

Внутри компьютера множество сложных компонентов: процессор, оперативная память, жесткий диск…

Каждый из этих компонентов имеет свой собственный интерфейс и настройки. Но для обычного пользователя достаточно нажать кнопку «Пуск», которая и есть фасад. Она скрывает всю сложность запуска компьютера и предоставляет простой интерфейс для пользователя

Фасад (Facade)

Подходит для упрощения сложного интерфейса, например, предоставления единой точки доступа к нескольким системным модулям

```
class SubsystemA {
public:
    void operationA() const { std::cout << "Subsystem A: Operation A\n"; }
};

class SubsystemB {
public:
    void operationB() const { std::cout << "Subsystem B: Operation B\n"; }
};

// Usage:
Facade* facade = new Facade();
facade->operation();
delete facade;
```

```
class Facade {
private:
    SubsystemA* subsystemA;
    SubsystemB* subsystemB;

public:
    Facade() {
        subsystemA = new SubsystemA();
        subsystemB = new SubsystemB();
    }

    ~Facade() {
        delete subsystemA;
        delete subsystemB;
    }

    void operation() const {
        std::cout << "Facade coordinates subsystems:\n";
        subsystemA->operationA();
        subsystemB->operationB();
    }
};
```

Фасад | Применимость

- **Сложные подсистемы**
  Когда система состоит из большого количества взаимосвязанных классов

- **Упрощение интерфейса**
  Когда нужно скрыть сложность системы от клиента

- **Поэтапная модернизация**
  Когда нужно постепенно модернизировать систему, не изменяя существующий клиентский код

Фасад как пример принципа
Открытости для расширения, закрытости для изменений (5)

Объединяет сложные подсистемы, упрощая их использование, но не изменяя их внутреннюю структуру

```
class LegacyPrinter {
public:
    void printDocument() { /* Старый сложный
способ */ }
};

class NewPrinter {
public:
    void printDocument() { /* Новый способ */ }
};
```

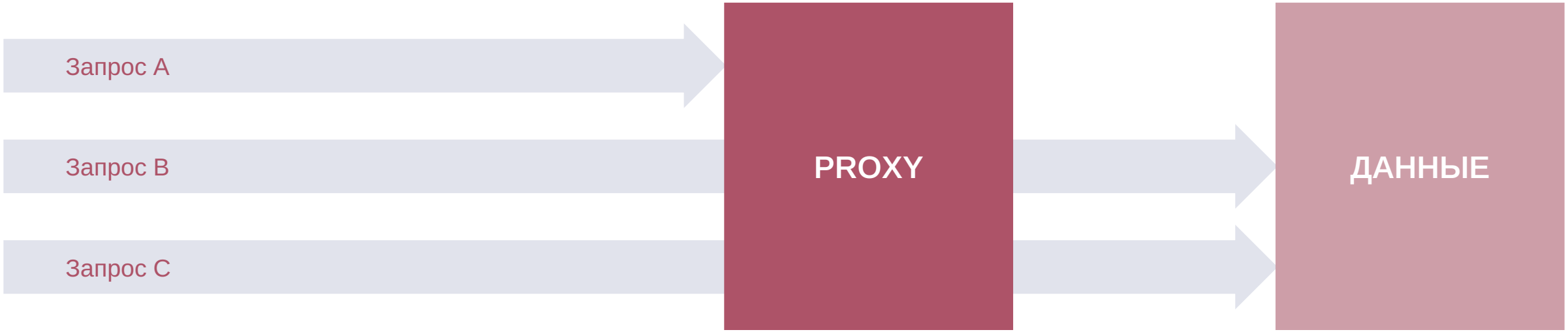
```
class PrinterAdapter {
    NewPrinter* printer;
public:
    PrinterAdapter(NewPrinter* p) : printer(p) {}
    void print() { printer->printDocument(); }
};
```

Заместитель (Proxy)

! Всегда создавайте отдельный объект для управления доступом к приватному элементу

Например, при распределении доступов к какой-то секретной информации

Вместо того, чтобы прописывать всем прямой доступ, лучше создать посредника (прокси), который уже будет проверять, имеет ли конкретный запрашивающий эти права



Заместитель (Proxy)

Применяется для последовательной обработки запросов, например, в системе обработки событий (фильтрация, логирование)

```
class Subject {
public:
    virtual void request() = 0;
};
```

```
class RealSubject : public Subject {
public:
    void request() override {
        std::cout << "RealSubject request\n";
    }
};
```

```
class Proxy : public Subject {
private:
    RealSubject* realSubject;
public:
    Proxy() : realSubject(nullptr) {}

    void request() override {
        if (!realSubject) {
            realSubject = new RealSubject();
        }
        realSubject->request();
    }
    ~Proxy() {
        delete realSubject;
    }
};
```

```
int main() {
    Proxy* proxy = new Proxy();
    proxy->request();


    delete proxy;
    return 0;
}
```

Заместитель | Применимость

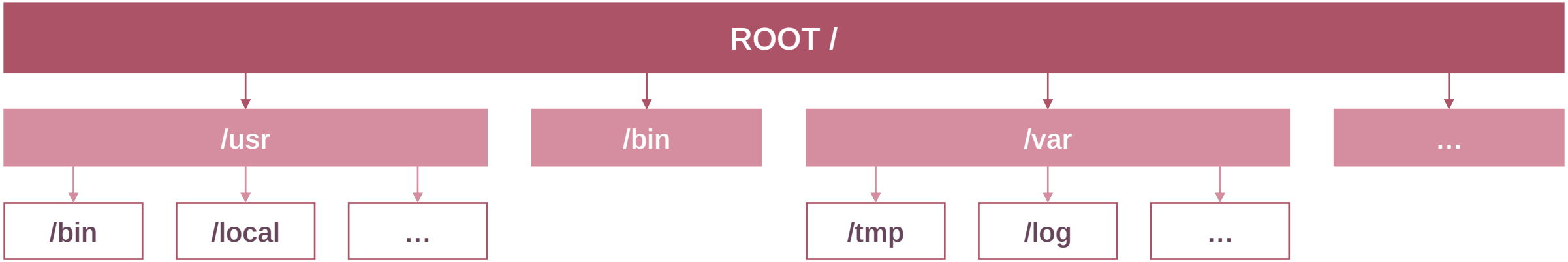
- **Когда нужно управлять доступом к объекту**
  (например, разрешать или запрещать действия)
- **Когда объект слишком дорогой в создании или требует медленных операций, и его создание нужно отложить**
- **Когда необходимо оптимизировать производительность**
  (например, с помощью кеширования)
- **Когда нужно добавить дополнительное поведение к объекту, не изменяя сам объект**

Компоновщик (Composite)

! При работе с древовидной структурой обеспечьте единый интерфейс для элементов не привязанный к уровню вложенности

Если рассмотрим файловую систему с папками и файлами, то папка – это сложный объект, который может содержать другие папки и файлы, а файл – простой объект

Операции вроде «открыть», «удалить» или «показать свойства» можно выполнить как для файла, так и для папки, хотя принцип будет разным. Но, благодаря компоновщику, неважно, работаете ли вы с отдельным файлом или с группой файлов и папок, – интерфейс останется единым



Компоновщик (Composite)

Применяется для работы с деревьями объектов, например, в файловых системах, где директории и файлы имеют общий интерфейс

```
class Component {
public:
    virtual void operation() = 0;
};
```

```
class Leaf : public Component {
public:
    void operation() override {
        std::cout << "Leaf operation" <<
std::endl;
    }
};
```

```
class Composite : public Component {
private:
    std::vector<Component*> children;
public:
    void add(Component* c) {
        children.push_back(c);
    }

    void operation() override {
        for (auto child : children) {
            child->operation();
        }
    }
};
```

```
int main() {
    Leaf* leaf1 = new Leaf();
    Leaf* leaf2 = new Leaf();
    Composite* composite = new
Composite();
    composite->add(leaf1);
    composite->add(leaf2);

    composite->operation();
    delete leaf1;
    delete leaf2;
    delete composite;
    return 0;
}
```

Компоновщик | Применимость

- **Иерархические структуры данных**
  Когда нужно представить данные в виде дерева или графа

- **Единый интерфейс для элементов**
  Когда клиент должен одинаково работать с отдельными элементами и группами элементов

- **Динамическое добавление и удаление элементов**
  Когда структура данных может изменяться во время выполнения программы

Паттерны программирования

И основные типы

Креационные паттерны

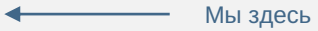
Абстрактная фабрика, Строитель, Фабричный метод, Прототип, Одиночка

Структурные паттерны

Адаптер, Мост, Компоновщик, Декоратор, Фасад, Легковес, Заместитель

Поведенческие паттерны

Цепочка обязанностей, Команда, Итератор, Посредник, Снимок, Наблюдатель, Состояние, Стратегия, Шаблонный метод, Посетитель, Интерпретатор



Test-Driven Design (TDD)

И связь с паттернами программирования

Заключение

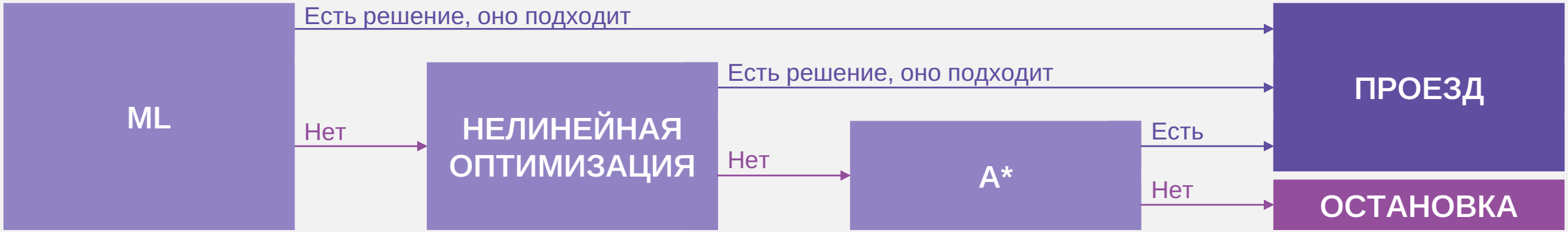
И что еще можно изучить по теме паттернов

Цепочка обязанностей (Chain of Responsibility)

! Если хотите повысить отказоустойчивость системы с помощью альтернативных алгоритмов, создайте для них последовательную обработку

Пример – создание траектории для робота-курьера, где нет 100% надежного алгоритма

Поэтому часто отрабатывается несколько алгоритмов от лучшего к самому надежному,
а каждый результат проверяется на соответствие определенным критериям выполнения задачи



Если алгоритм прошел, то цепочка заканчивается, а если провалился – запускается следующий

Если ни один из вариантов не сработал, то мы резко останавливаемся

Цепочка обязанностей (Chain of Responsibility)

Применяется для последовательной обработки запросов – таких как в системе обработки событий (фильтрация, логирование)

```
class ConcreteHandlerA : public Handler {
public:
    void handleRequest(int request) override {
        if (request == 1) {
            std::cout << "Handler A handled request" << std::endl;
        } else {
            Handler::handleRequest(request);
        }
    }
};
```

```
class ConcreteHandlerB : public Handler {
public:
    void handleRequest(int request) override {
        if (request == 2) {
            std::cout << "Handler B handled request" << std::endl;
        } else {
            Handler::handleRequest(request);
        }
    }
};
```

```
class Handler {
protected:
    Handler* next;
public:
    Handler() : next(nullptr) {}
    void setNext(Handler* handler) {
        next = handler;
    }
    virtual void handleRequest(int request) {
        if (next) {
            next->handleRequest(request);
        }
    }
};
```

```
int main() {
    Handler* handlerA = new ConcreteHandlerA();
    Handler* handlerB = new ConcreteHandlerB();
    handlerA->setNext(handlerB);

    handlerA->handleRequest(1);  // Output:
Handler A handled request
    handlerA->handleRequest(2);  // Output:
Handler B handled request

    delete handlerA;
    delete handlerB;
    return 0;
}
```

Цепочка обязанностей | Применимость

- **Несколько обработчиков**
  Когда несколько объектов могут обрабатывать один и тот же запрос

- **Неизвестный получатель**
  Когда заранее неизвестно, какой именно объект должен обработать запрос

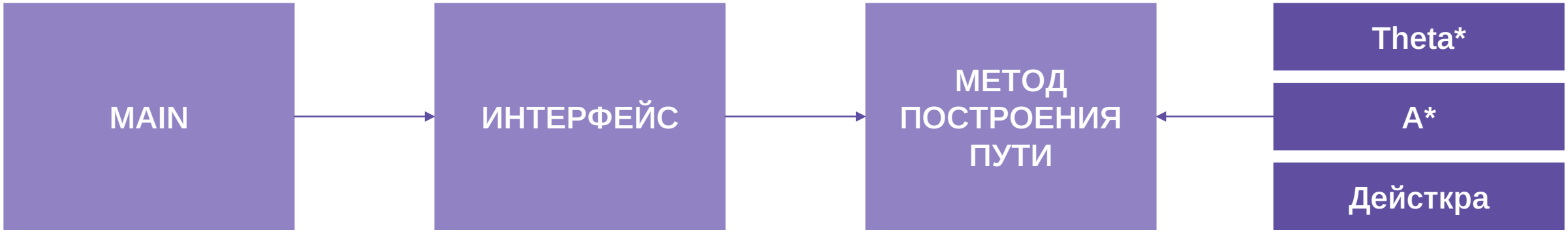
- **Динамическая конфигурация**
  Когда нужно динамически добавлять или удалять обработчики из цепочки

Стратегия (Strategy)

! Если требуется применять разные алгоритмы из семейства в разных ситуациях, то их нужно привести к единому интерфейсу для удобства исполнения

Предположим, мы ищем оптимальный путь в двунаправленном полносвязном графе с тремя и более измерениями – для подбора алгоритма для разных условий запуска (скорость/эффективность/ресурсозатратность…) мы можем пробовать разные алгоритмы

Чтобы удобно переключаться между ними можно определить одинаковый, независимый от алгоритма интерфейс для его вызова – и менять алгоритм уже в отдельном методе

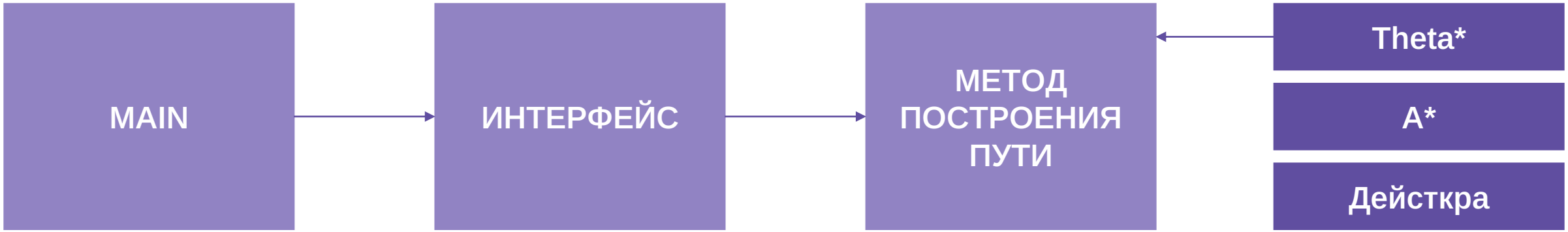


Стратегия (Strategy)

! Если требуется применять разные алгоритмы из семейства в разных ситуациях, то их нужно привести к единому интерфейсу для удобства исполнения

Предположим, мы ищем оптимальный путь в двунаправленном полносвязном графе с тремя и более измерениями – для подбора алгоритма для разных условий запуска (скорость/эффективность/ресурсозатратность…) мы можем пробовать разные алгоритмы

Чтобы удобно переключаться между ними можно определить одинаковый, независимый от алгоритма интерфейс для его вызова – и менять алгоритм уже в отдельном методе

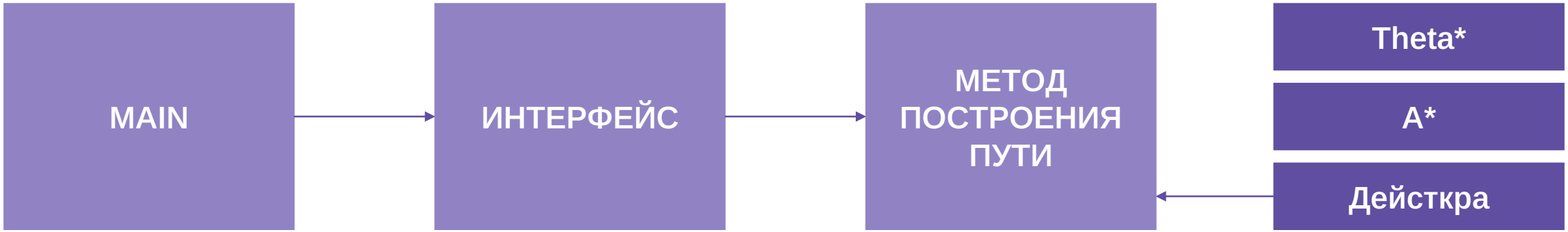


Стратегия (Strategy)

! Если требуется применять разные алгоритмы из семейства в разных ситуациях, то их нужно привести к единому интерфейсу для удобства исполнения

Предположим, мы ищем оптимальный путь в двунаправленном полносвязном графе с тремя и более измерениями – для подбора алгоритма для разных условий запуска (скорость/эффективность/ресурсозатратность…) мы можем пробовать разные алгоритмы

Чтобы удобно переключаться между ними можно определить одинаковый, независимый от алгоритма интерфейс для его вызова – и менять алгоритм уже в отдельном методе



Стратегия (Strategy)

Используется для выбора алгоритма выполнения задачи, например, для расчёта стоимости доставки разными методами (курьер, почта)

```
class Strategy {
public:
    virtual void execute() = 0;
};

class ConcreteStrategyA : public Strategy {
public:
    void execute() override {
        std::cout << "Strategy A\n";
    }
};

class ConcreteStrategyB : public Strategy {
public:
    void execute() override {
        std::cout << "Strategy B\n";
    }
};
```

```
class Context {
private:
    Strategy* strategy;
public:
    Context(Strategy* s) : strategy(s) {}
    void setStrategy(Strategy* s) {
        strategy = s;
    }
    void executeStrategy() {
        strategy->execute();
    }
};
```

```
int main() {
    ConcreteStrategyA* strategyA = new ConcreteStrategyA();
    ConcreteStrategyB* strategyB = new ConcreteStrategyB();

    Context* context = new Context(strategyA);
    context->executeStrategy();

    context->setStrategy(strategyB);
    context->executeStrategy();

    delete strategyA;
    delete strategyB;
    delete context;
    return 0;
}
```

Стратегия | Применимость

- **Алгоритмы, которые могут меняться во время выполнения**
- **Семейство похожих алгоритмов**
  Когда существует несколько вариантов решения одной задачи
- **Изменение алгоритма без изменения клиента**
  Когда нужно заменить алгоритм, не изменяя код, который его использует

Стратегия как пример принципа
Повторное использование кода (3)

Позволяет легко переключать алгоритмы в зависимости от ситуации

```python
from abc import ABC, abstractmethod

# Базовый интерфейс для стратегий
class DiscountStrategy(ABC):
    @abstractmethod
    def apply_discount(self, price: float) -> float:
        pass

# Конкретные стратегии
class FixedDiscount(DiscountStrategy):
    def __init__(self, discount: float):
        self.discount = discount

    def apply_discount(self, price: float) -> float:
        return price - self.discount

class PercentageDiscount(DiscountStrategy):
    def __init__(self, percentage: float):
        self.percentage = percentage

    def apply_discount(self, price: float) -> float:
        return price * (1 - self.percentage / 100)
```

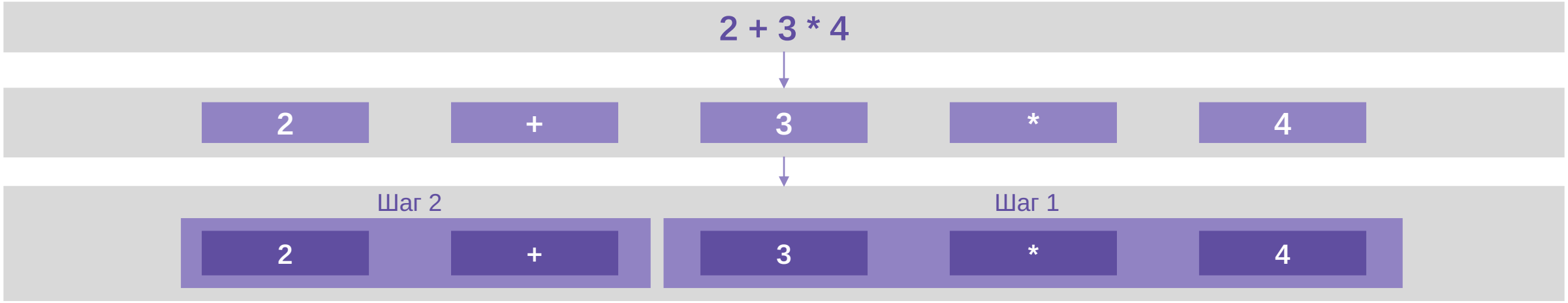
```python
# Контекст
class Product:
    def __init__(self, price: float, discount_strategy: DiscountStrategy):
        self.price = price
        self.discount_strategy = discount_strategy

    def set_discount_strategy(self, strategy: DiscountStrategy):
        self.discount_strategy = strategy

    def get_price(self) -> float:
        return self.discount_strategy.apply_discount(self.price)

# Пример использования
product = Product(100, FixedDiscount(10))
print("Fixed discount:", product.get_price())  # $90

product.set_discount_strategy(PercentageDiscount(15))
print("Percentage discount:", product.get_price())  # $85
```

Интерпретатор (Interpreter)

! Декомпозируйте сложные задачи на отдельные написанные вами логические блоки

Представьте себе калькулятор, который может вычислять простые арифметические выражения, записанные в виде строки

Полученную строку он сначала последовательно разберет на базовые операции и элементы, потом добавит значения в операции и отсортирует их для запуска в необходимой очереди:



Интерпретатор (Interpreter)

Подходит для обработки текстовых команд, например, при реализации языка запросов для базы данных

```
class Expression {
public:
    virtual bool interpret(const std::string&
context) = 0;
};
```

```
class TerminalExpression : public Expression
{
private:
    std::string data;
public:
    TerminalExpression(const std::string& s) :
data(s) {}
    bool interpret(const std::string& context)
override {
        return context.find(data) !=
std::string::npos;
    }
};
```

```
class OrExpression : public Expression {
private:
    Expression* expr1;
    Expression* expr2;
public:
    OrExpression(Expression* e1, Expression*
e2) : expr1(e1), expr2(e2) {}
    bool interpret(const std::string& context)
override {
        return expr1->interpret(context) || expr2-
>interpret(context);
    }
};
```

```
int main() {
    Expression* isMale = new TerminalExpression("Male");
    Expression* isMarried = new TerminalExpression("Married");
    Expression* isMaleOrMarried = new OrExpression(isMale,
isMarried);


    std::cout << "Is Male or Married? " << isMaleOrMarried-
>interpret("Male") << std::endl;
    std::cout << "Is Male or Married? " << isMaleOrMarried-
>interpret("Single") << std::endl;


    delete isMale;
    delete isMarried;
    delete isMaleOrMarried;
    return 0;
}
```

Интерпретатор | Применимость

- **Когда нужно создать собственный язык для решения конкретной задачи**
  (например, язык запросов к базе данных, конфигурационный язык)

- **Вычисление выражений**
  Для интерпретации математических выражений, логических условий и других типов выражений

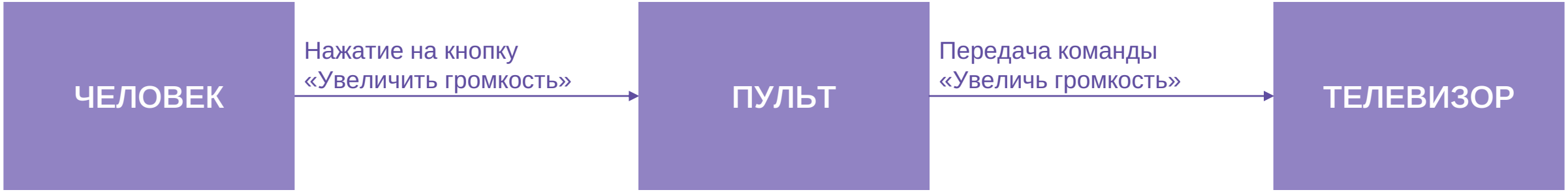
- **Простые языки программирования**
  Для создания небольших языков программирования, которые могут быть легко расширены

Команда (Command)

! Если необходимо тонкое управление запросами, то стоит выделить команду в отдельный объект с возможностью любых операций (вместо, например, прямой функции обработки)

Можно представить на примере пульта дистанционного управления телевизором:

- Каждая кнопка на пульте – команда (например, включение, выключение, увеличение громкости)
- Нажатие кнопки – создание объекта команды
- Этот объект содержит информацию о том, какую операцию должно быть выполнено (например, увеличить громкость)
- Пульт (invoker) принимает эту команду и передает ее телевизору (receiver) для выполнения операции



Команда (Command)

Используется для инкапсуляции запросов, например, в системе действий «Отменить / Повторить» в текстовом редакторе

```
class Command {
public:
    virtual void execute() = 0;
};

class Light {
public:
    void turnOn() { std::cout << "Light is ON" << std::endl; }
    void turnOff() { std::cout << "Light is OFF" << std::endl; }
};
```

```
class LightOnCommand : public Command {
private:
    Light* light;
public:
    LightOnCommand(Light* l) : light(l) {}
    void execute() override {
        light->turnOn();
    }
};
```

```
class LightOffCommand : public Command {
private:
    Light* light;
public:
    LightOffCommand(Light* l) : light(l) {}
    void execute() override {
        light->turnOff();
    }
};
```

```
class RemoteControl {
private:
    Command* command;
public:
    void setCommand(Command* cmd) {
        command = cmd;
    }
    void pressButton() {
        command->execute();
    }
};
```

```
int main() {
    // ...
    RemoteControl* remote = new RemoteControl();

    remote->setCommand(lightOn);
    remote->pressButton();

    remote->setCommand(lightOff);
    remote->pressButton();
}
```

Команда | Применимость

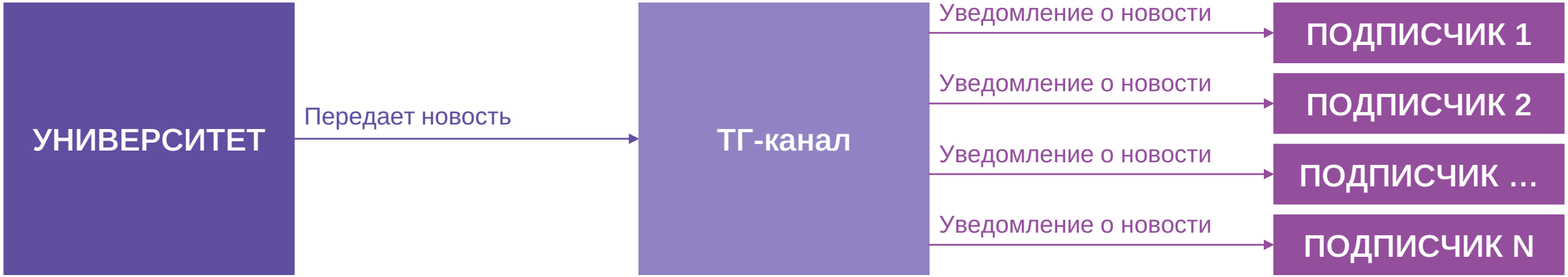
- Отложенное выполнение
- Многоуровневая и обычная отмена действий
- Журналирование действий
- Многопоточность

Наблюдатель (Observer)

! Для автоматического оповещения об изменениях объекта всех подписанных систем следует использовать отдельную сущность/-и, отвечающую за это

Чтобы узнать новости своего университета студенты могут подписаться на официальный ТГ-канал

Таким образом они становятся подписчиками («наблюдателями»). а ТГ-канал – «издателем» изменений университета. Когда выходит новый пост, все подписчики автоматически уведомляются об этом и могут прочитать свежие новости



Наблюдатель (Observer)

Применяется для подписки на события
Например, обновление интерфейса при изменении данных в программе

```
class Observer {
public:
    virtual void update(int value) = 0;
};

class ConcreteObserver : public Observer {
private:
    std::string name;
public:
    ConcreteObserver(const std::string& n) : name(n) {}
    void update(int value) override {
        std::cout << name << " received update: " << value
<< std::endl;
    }
};
```

```
class Subject {
private:
    std::vector<Observer*> observers;
public:
    void addObserver(Observer* observer) {
        observers.push_back(observer);
    }
    void removeObserver(Observer* observer) {
        observers.erase(std::remove(observers.begin(),
observers.end(), observer), observers.end());
    }
    void notify(int value) {

        for (Observer* observer : observers) {
            observer->update(value);
        }
    }
};
```

```
int main() {
    Subject* subject = new Subject();

    ConcreteObserver* observer1 = new
ConcreteObserver("Observer 1");
    ConcreteObserver* observer2 = new
ConcreteObserver("Observer 2");

    subject->addObserver(observer1);
    subject->addObserver(observer2);

    subject->notify(5);

    delete observer1;
    delete observer2;
    delete subject;
    return 0;
}
```

Наблюдатель | Применимость

- **Графические пользовательские интерфейсы**
  Когда элементы интерфейса должны обновляться в ответ на изменения в данных модели

- **Для реализации систем оповещений**
  Например, о новых сообщениях, изменениях в данных и т.д

- **Для обработки событий в различных приложениях**

- **Многопользовательские игры**
  Для синхронизации состояния игры между клиентами

Наблюдатель как пример принципа
Ослабление связей (2)

Позволяет одному объекту подписываться на изменения другого без сильной привязки

```python
class Subject:
    def __init__(self):
        self.observers = []

    def attach(self, observer):
        self.observers.append(observer)

    def notify(self, data):
        for observer in self.observers:
            observer.update(data)
```

```python
class Observer:
    def update(self, data):
        print(f"Received data: {data}")

subject = Subject()
observer = Observer()
subject.attach(observer)
subject.notify("Hello, World!")
```

Снимок (Memento)

! Если необходимо сохранить состояние объекта с последующим восстановлением – используйте «снимок» объекта (сохранение состояния)

Для реализации паттерна следует реализовать запоминание ключевых элементов, через которые можно без потерь восстановить изначальный объект

Примером может послужить привычный нам робот пылесос – при выполнении уборки, во время остановки на паузу, он запомнит поставленную задачу и состояние своей системы

При помощи этой информации он сможет без происшествий продолжить и завершить уборку при последующем возобновлении задачи

Снимок (Memento)

Используется для сохранения состояния объекта, например, в играх для реализации функции «Сохранить / Загрузить»

```
class Memento {
private:
    std::string state;
public:
    Memento(const std::string& st) :
state(st) {}
    std::string getState() { return state; }
};
```

```
class Originator {
private:
    std::string state;
public:
    void setState(const std::string& st) { state = st; }
    Memento* saveStateToMemento() {
        return new Memento(state);
    }
    void getStateFromMemento(Memento* memento) {
        state = memento->getState();
    }
    void showState() { std::cout << "State: " << state <<
std::endl; }
};
```

```
int main() {
    Originator* originator = new Originator();
    originator->setState("State 1");
    originator->showState();

    Memento* memento = originator-
>saveStateToMemento();

    originator->setState("State 2");
    originator->showState();

    originator->getStateFromMemento(memento);
    originator->showState();

    delete memento;
    delete originator;
    return 0;
}
```

Снимок | Применимость

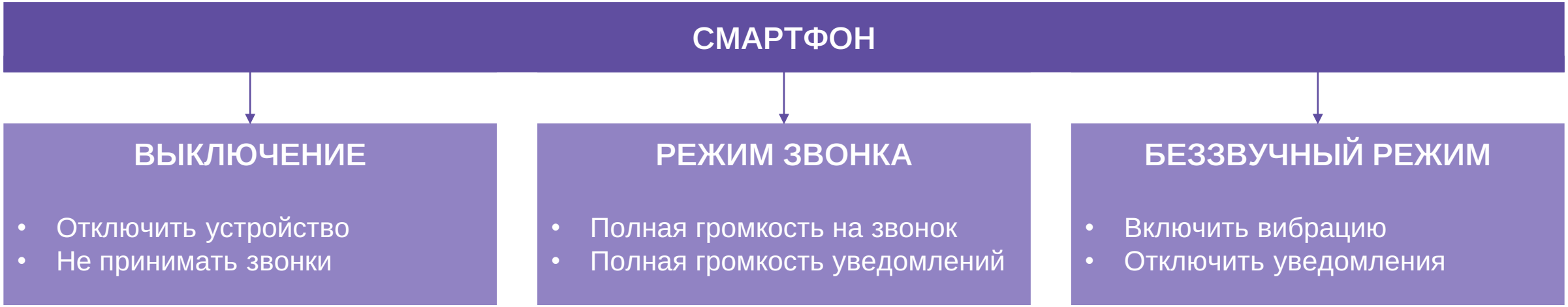
- **Когда нужно реализовать механизм отмены действий**
  (например, в текстовых редакторах, графических редакторах)
- **Когда необходимо хранить историю изменений объекта**
  для последующего восстановления
- **Сохранение состояний в играх**
  Для реализации функций сохранения и загрузки игры
- **Создание точек восстановления**
  В системах, где важна возможность вернуться к стабильному состоянию

Состояние (State)

! Для изменения поведения объекта в зависимости от его внутреннего состояния лучше использовать отдельные сущности, описывающие конкретную логику при конкретном состоянии

Такой принцип существует в смартфонах – их настройки меняются в зависимости от выбранного нами состояния: выключен, в режиме звонка, в беззвучном режиме и т.д.

Каждый режим (состояние) определяет набор возможных действий: сбросить звонок, включить звук уведомлений, включить вибрацию и т.д.



Состояние (State)

Подходит для изменения поведения объекта в зависимости от его состояния, например, для реализации работы банкомата

```
class State {
public:
    virtual void handle() = 0;
};

class ConcreteStateA : public State {
public:
    void handle() override {
        std::cout << "Handling state A\n";
    }
};

class ConcreteStateB : public State {
public:
    void handle() override {
        std::cout << "Handling state B\n";
    }
};
```

```
class Context {
private:
    State* state;
public:
    Context(State* s) : state(s) {}

    void setState(State* s) {
        state = s;
    }

    void request() {
        state->handle();
    }
};
```

```
int main() {
    ConcreteStateA* stateA = new ConcreteStateA();
    ConcreteStateB* stateB = new ConcreteStateB();

    Context* context = new Context(stateA);
    context->request();

    context->setState(stateB);
    context->request();

    delete stateA;
    delete stateB;
    delete context;
    return 0;
}
```

Состояние | Применимость

- **Системы с конечным числом состояний**
  Когда объект может находиться в ограниченном количестве состояний, и его поведение зависит от текущего состояния

- **Управление состоянием объектов**
  Когда необходимо явно управлять переходами между состояниями объекта

- **Реализация сложных состояний**
  Когда требуется моделировать сложные состояния с множеством переходов

Итератор (Iterator)

! Чтобы последовательно перебрать элементы коллекции без просмотра внутренней структуры – используй итератор

Для примера возьмем таблицу со статистикой

Данные могут храниться сложным образом и состоять из множества структур и объектов, к которым прямого доступ у нас часто нет. В таком случае требуется реализовать методы для обхода всей таблицы, предоставляя данные согласно определенной логике

Например, прямая ил обратная последовательность, каждый второй элемент и т. д.

!! При реализации паттерна важно помнить, что итераторы могут быть в роли альтернативы для прямого доступа к данным

Пример – std в C++!!

Итератор (Iterator)

Подходит для обхода коллекций
Например, в базе данных для последовательного доступа к записям

```
template <typename T>
class Iterator {
private:
    std::vector<T>& collection;
    size_t index;
public:
    Iterator(std::vector<T>& col) : collection(col), index(0) {}
    bool hasNext() {
        return index < collection.size();
    }
    T next() {
        return collection[index++];
    }
};
```

```
int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5};
    Iterator<int> iterator(numbers);


    while (iterator.hasNext()) {
        std::cout << iterator.next() << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

Итератор | Применимость

- **Абстрагирование доступа к элементам коллекции**
  Когда нужно скрыть внутреннюю реализацию коллекции от клиента

- **Разные способы обхода**
  Когда требуется несколько способов обхода одной и той же коллекции (например, прямой и обратный)

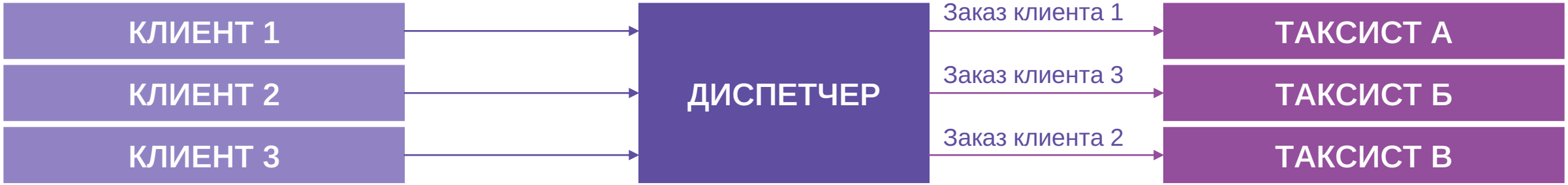
- **Поддержка множественных итераторов**
  Когда несколько клиентов должны одновременно перебирать элементы коллекции

Посредник (Mediator)

! Если необходима связь между множеством объектов, то следует заменить прямое общение на коммуникацию через класс-посредник

По такому принципу работает служба такси – водители и клиенты не связываются друг с другом напрямую, а взаимодействуют через диспетчера

1. Клиент звонит диспетчеру, чтобы заказать такси
2. Диспетчер (посредник) связывается с ближайшим свободным водителем и передает ему информацию о заказе



Диспетчер (посредник) координирует взаимодействие между клиентами и водителями, снижая сложность системы и уменьшая количество прямых связей между ними

Посредник (Mediator)

Применяется для управления взаимодействием между объектами, например, в чат-приложении для передачи сообщений между пользователями через сервер

```
class Colleague;


class Mediator {
public:
    virtual void sendMessage(const std::string&
message, Colleague* colleague) = 0;
};
```

```
class Colleague {
protected:
    Mediator* mediator;
public:
    Colleague(Mediator* med) : mediator(med) {}
    virtual void receiveMessage(const std::string&
message) = 0;
};
```

```
class ConcreteMediator : public Mediator {
private:
    Colleague* colleague1;
    Colleague* colleague2;
public:
    void setColleague1(Colleague* c1) { colleague1 = c1; }
    void setColleague2(Colleague* c2) { colleague2 = c2; }


    void sendMessage(const std::string& message,
Colleague* colleague) override {
        if (colleague == colleague1) {
            colleague2->receiveMessage(message);
        } else {
            colleague1->receiveMessage(message);
        }
    }
};
```

```
class ConcreteColleague1 : public Colleague {
public:
    ConcreteColleague1(Mediator* med) : Colleague(med) {}
    void receiveMessage(const std::string& message) override {
        std::cout << "Colleague 1 received: " << message <<
std::endl;
    }
};
```

```
class ConcreteColleague2 : public Colleague {
public:
    ConcreteColleague2(Mediator* med) : Colleague(med) {}
    void receiveMessage(const std::string& message) override {
        std::cout << "Colleague 2 received: " << message <<
std::endl;
    }
};
```

Посредник (Mediator)

Применяется для управления взаимодействием между объектами, например, в чат-приложении для передачи сообщений между пользователями через сервер

```
int main() {
    ConcreteMediator* mediator = new ConcreteMediator();

    ConcreteColleague1* colleague1 = new ConcreteColleague1(mediator);
    ConcreteColleague2* colleague2 = new ConcreteColleague2(mediator);

    mediator->setColleague1(colleague1);
    mediator->setColleague2(colleague2);

    colleague1->receiveMessage("Hello, from Colleague 1!");
    colleague2->receiveMessage("Hello, from Colleague 2!");

    delete colleague1;
    delete colleague2;
    delete mediator;
    return 0;
}
```

Посредник | Применимость

- **Когда множество объектов сильно связаны между собой,**
  что затрудняет понимание и изменение системы
- **Если необходимо централизованно управлять взаимодействием между объектами**
- **Когда взаимодействие между объектами становится сложным и запутанным**
- **При создании графических интерфейсов,**
  где множество элементов взаимодействуют друг с другом

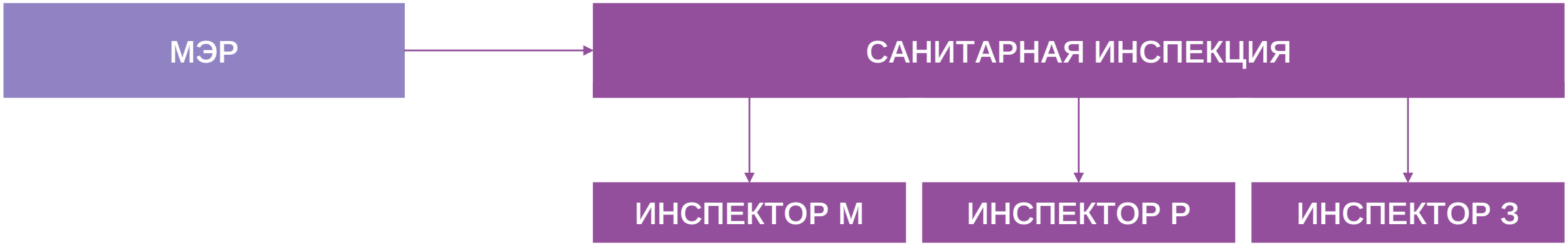
Посетитель (Visitor)

! Чтобы не изменять код при добавлении новых операций к существующей структуре объектов, выделите эти операции в отдельную сущность

Представим, что мы мэр города, котором есть Магазин, Ресторан и Завод

Нам необходимо проводить над ними санитарный надзор, но каждый тип предприятия имеет свои особенности, и для проверки каждого требуется особый набор операций – и ни одним из них мы не владеем

Так что лучший вариант – создать отдельную санитарную инспекцию, которая и поможет с проверкой:



Посетитель (Visitor)

Используется для добавления операций над объектами, например, выполнения анализа дерева выражений в компиляторе

```
class Element;

class Visitor {
public:
    virtual void visit(Element* element) = 0;
};

class Element {
public:
    virtual void accept(Visitor* visitor) = 0;
};
```

```
class ConcreteElementA : public Element {
public:
    void accept(Visitor* visitor) override {
        visitor->visit(this);
    }
};

class ConcreteElementB : public Element {
public:
    void accept(Visitor* visitor) override {
        visitor->visit(this);
    }
};

class ConcreteVisitor : public Visitor {
public:
    void visit(Element* element) override {
        std::cout << "Visiting element\n";
    }
};
```

```
int main() {
    ConcreteElementA* elementA = new
ConcreteElementA();
    ConcreteElementB* elementB = new
ConcreteElementB();
    ConcreteVisitor* visitor = new ConcreteVisitor();

    elementA->accept(visitor);
    elementB->accept(visitor);

    delete elementA;
    delete elementB;
    delete visitor;
    return 0;
}
```

Посетитель | Применимость

- Добавление новых операций к существующим классам
- Когда алгоритмы обработки данных должны быть отделены от самих данных
- Когда требуется генерировать различные типы отчетов на основе одной и той же структуры данных
- Когда необходимо выполнять различные виды анализа над сложной структурой данных

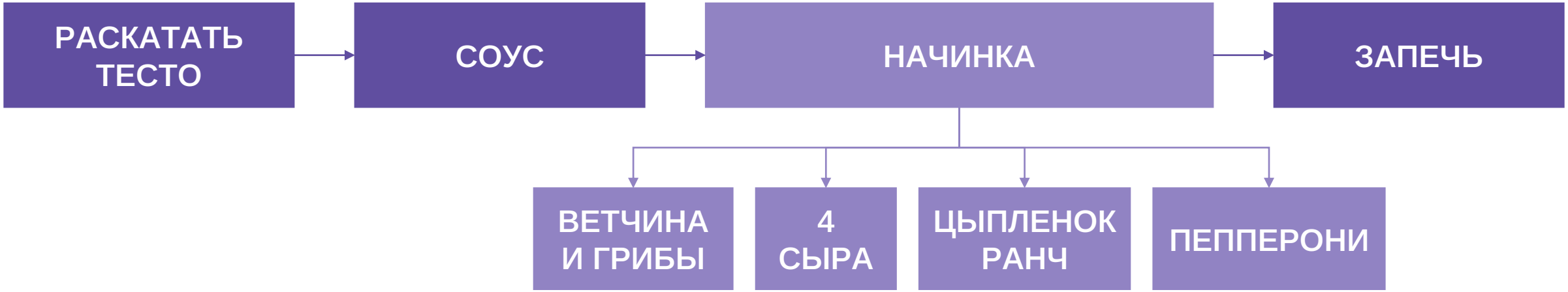
Шаблонный метод (Template Method)

! Если необходимо модернизировать шаги алгоритма, то следует, не меняя скелета программы, переопределить шаги через подклассы

Возьмем в пример пиццерию

Основные шаги одинаковы для всех видов, однако начинка может быть разной

Так что можно определить общий скелет рецепта (шаги приготовления), а дополнения («подклассы») и конкретные виды пиццы могут переопределять только шаг добавления начинки



Шаблонный метод (Template Method)

Применяется для создания базового алгоритма с возможностью изменения отдельных шагов, например, в обработке файлов разного типа

```
class AbstractClass {
public:
    virtual void stepOne() = 0;
    virtual void stepTwo() = 0;
    void templateMethod() {
        stepOne();
        stepTwo();
    }
};
```

```
class ConcreteClass : public
AbstractClass {
public:
    void stepOne() override {
        std::cout << "Step One\n";
    }

    void stepTwo() override {
        std::cout << "Step Two\n";
    }
};
```

```
int main() {
    ConcreteClass* concreteClass = new
ConcreteClass();
    concreteClass->templateMethod();

    delete concreteClass;
    return 0;
}
```

Шаблонный метод | Применимость

- Когда нужно определить общий алгоритм для семейства классов, позволяя подклассам изменять некоторые его части
- Когда часть алгоритма одинакова для всех подклассов, а другая часть может изменяться
- Когда нужно контролировать расширение алгоритма подклассами

Паттерны программирования

И основные типы

Креационные паттерны

Абстрактная фабрика, Строитель, Фабричный метод, Прототип, Одиночка

Структурные паттерны

Адаптер, Мост, Компоновщик, Декоратор, Фасад, Легковес, Заместитель

Поведенческие паттерны

Цепочка обязанностей, Команда, Итератор, Посредник, Снимок, Наблюдатель, Состояние, Стратегия, Шаблонный метод, Посетитель, Интерпретатор

Test-Driven Design (TDD)

И связь с паттернами программирования

Мы здесь

Заключение

И что еще можно изучить по теме паттернов

Test-Driven Design (TDD)

> **Test-Driven Design (TDD) –** это методология разработки программного обеспечения, при которой процесс создания кода начинается с написания тестов, а не с самого кода

Основной принцип TDD заключается в тестировании функционала программы на каждом шаге разработки – это дает нам:

- Нахождение ошибки на ранней стадии
- Упрощение рефакторинга
- Упрощение поддержания программы

Методология Test-Driven Design (TDD) делает процесс разработки более предсказуемым и надежным, обеспечивая быстрые итерации и облегчая поддержку и расширение программных решений

Цикл TDD

1

Написание теста

Прежде чем писать код, разработчик пишет тест, который определяет поведение или функциональность, которую должен реализовать будущий код

2

Написание кода

Разработчик реализует минимальный код – исключительно для того, чтобы пройти тест, что помогает избежать ненужных сложностей и излишней функциональности

3

Запуск теста

Если тест прошел успешно, можно двигаться дальше, если нет – исправляем код до тех пор, пока тест не будет пройден

Пример цикла TDD

1

Написание теста

Мы пишем тест, который проверяет правильность суммы двух чисел

2

Написание кода

Написание кода для функции `add`, которая будет проходить тесты, написанные в шаге 1

3

Запуск теста

Запуск теста для проверки правильности работы функции

```
#include <cassert>

2 int add(int a, int b) {
    return a + b;
  }

1 void test_add() {
    assert(add(2, 3) == 5);  // Ожидаем, что 2 + 3 = 5
    assert(add(-1, 1) == 0); // Ожидаем, что -1 + 1 = 0
    assert(add(0, 0) == 0);  // Ожидаем, что 0 + 0 = 0
  }

  int main() {
3   test_add();  // Запуск теста
    return 0;
  }
```

Преимущества TDD

Улучшение качества кода
Поскольку тесты пишутся до написания кода, это помогает гарантировать, что каждый фрагмент кода имеет конкретную цель и работает правильно с самого начала – это минимизирует вероятность появления ошибок

Сокращение количества ошибок
Тестирование на каждом шаге разработки позволяет находить и устранять ошибки на ранних этапах, что уменьшает стоимость исправления багов и ускоряет процесс разработки

Легкость в адаптации к изменениям
Благодаря наличию тестов, можно безопасно вносить изменения и рефакторинг в код, так как тесты обеспечивают, что код продолжает работать корректно

Документация кода
Написанные тесты служат документацией, которая показывает, как код должен себя вести, что упрощает понимание функциональности для других разработчиков, которые будут работать с этим кодом в будущем

Покрытие всех аспектов функциональности
Тесты часто затрагивают все возможные сценарии работы кода, что помогает обнаружить и устранить недочеты, которые могли бы быть упущены при обычной разработке

Связь Паттернов Программирования с TDD

Тесты в TDD помогают проверять и убедиться, что паттерны реализованы правильно

Например, справа логика подхода
при написании паттерна «Фабрика»

Если тесты прошли, значит, паттерн реализован правильно

1

Написание теста на создание объектов

«Мы хотим, чтобы фабрика создавала объект нужного типа в зависимости от условий»

2

Написание кода

«Здесь создается фабрика, которая по условию создает объекты определенных классов»

3

Запуск теста

«Проверка, что фабрика создает правильные объекты в зависимости от типа»

Связь Паттернов Программирования с TDD

Тесты в TDD помогают проверять и убедиться, что паттерны реализованы правильно

Как паттерн Фабрика (Factory) можно тестировать с помощью TDD:

1. **Написание теста на создание объектов**

   «Мы хотим, чтобы фабрика создавала объект нужного типа в зависимости от условий»

2. **Написание кода**

   «Здесь создается фабрика, которая по условию создает объекты определенных классов»

3. **Запуск теста**

   «Проверка, что фабрика создает правильные объекты в зависимости от типа»

Как паттерн Фабрика (Factory) можно тестировать с помощью TDD:

```
class Product {
public:
  virtual void doSomething() = 0;
};


// Concrete Products с тестовой реализацией


class Creator {
public:
  Product* createProduct(int type) {
     if (type == 1)
        return new ConcreteProductA();
     else
        return new ConcreteProductB();
  }
};
```

```
void testFactory() {
  Creator creator;

  Product* productA = creator.createProduct(1);
  productA->doSomething();  // Ожидаем "Product A"
  assert(dynamic_cast<ConcreteProductA*>(productA) !=
nullptr);

  Product* productB = creator.createProduct(2);
  productB->doSomething();  // Ожидаем "Product B"
  assert(dynamic_cast<ConcreteProductB*>(productB) !=
nullptr);
}

int main() {
  testFactory();
  return 0;
}
```

Паттерны программирования

И основные типы

Креационные паттерны

Абстрактная фабрика, Строитель, Фабричный метод, Прототип, Одиночка

Структурные паттерны

Адаптер, Мост, Компоновщик, Декоратор, Фасад, Легковес, Заместитель

Поведенческие паттерны

Цепочка обязанностей, Команда, Итератор, Посредник, Снимок, Наблюдатель, Состояние, Стратегия, Шаблонный метод, Посетитель, Интерпретатор

Test-Driven Design (TDD)

И связь с паттернами программирования

Заключение

И что еще можно изучить по теме паттернов

Паттерны – проверенные временем решения

Они работают как универсальный язык между разработчиками, помогая быстро понимать архитектурные идеи

Антипаттерны же, напротив, служат напоминанием об ошибках, которых лучше избегать

Понимание паттернов – не цель, а инструмент

Важно не пытаться «запихнуть» паттерны в проект ради их использования, а применять их только тогда, когда это улучшает качество кода

Что еще можно изучить? (1/2)

1

Архитектурные паттерны (такие как MVC, CQRS, Event Sourcing)

Они решают задачи на уровне построения приложений и систем

2

Шаблоны из других областей

Например, паттерны для работы с «облаками» (Serverless, Microservices)
или «реактивными системами» (Reactive Streams)

3

Антипаттерны (от плохой организации кода Big Ball of Mud до ошибок в управлении проектом Death March)

Понять, как они проявляются и как их избегать

Что еще можно изучить? (1/2)

4

Современные библиотеки и фреймворки

Многие из них используют паттерны под капотом. Например:

- **В C++:**
  Boost, Qt
- **В Python:**
  Django (ORM как "Активный рекорд") или FastAPI
- **В Rust:**
  Tokio, actix-web

Что принято в компаниях?

- **Паттерны в реальных проектах – зависит от контекста**

В стартапах часто применяют гибкие подходы (например, Dependency Injection в REST API), в то время как в крупных компаниях больше внимания уделяется сложным архитектурным решениям

- **Выбор инструментов**

Многие компании используют фреймворки или библиотеки, которые скрывают детали реализации паттернов

Например, ORM в базах данных (паттерн «Репозиторий»), или Pub/Sub в обработке событий

А если команда решила применить антипаттерн?

1. Поговорите с командой

Если вам предложили явный антипаттерн, попробуйте объяснить, почему он может привести к проблемам (например, сложности в поддержке или низкая производительность)

2. Предложите альтернативы

Выскажите решение, которое лучше соответствует задаче, указав на его преимущества

3. Принимайте во внимание исключения

Иногда использование антипаттернов оправдано (например, ради быстрого прототипирования). В таких случаях важно зафиксировать это как **технический долг**

Общие советы по разработке

Знайте проблему, а не только решение
Перед применением паттерна убедитесь, что он действительно решает вашу задачу

Паттерны не всегда нужны
Если простой код можно реализовать без паттерна и он останется понятным, это может быть лучшим решением

Делайте ревью и учитесь
Архитектурные решения проходят через обсуждения с командой – используйте код-ревью как возможность улучшать архитектуру

Учитывайте контекст
Решение, которое работает в одном проекте, может не подойти другому из-за разных требований, ограничений или масштаба

Будьте готовы к изменению
Даже правильно выбранные паттерны могут устареть или стать менее подходящими с развитием проекта – будьте гибкими

Спасибо за внимание!
Ваши вопросы?