Коллоквиум 3

Михайлова Александра 11 группа

1 вариант

# 1. Поведенческие шаблоны проектирования

Поведенческие шаблоны позволяют инкапсулировать способы взаимодействия объектов. Их цель — разделение обязанностей и независимость алгоритмов от объектов, над которыми они работают.

Анализ шаблонов:

## 🔹1.1. Команда (Command)

Проблема:

Вам нужно выполнить действия, которые:

-можно отложить;

-можно отменить/повторить;

-нужно логировать или складывать в очередь;

-зависят от типа получателя, но вызываются единообразно.

Решение (паттерн):

Инкапсулировать запрос как объект: каждое действие превращается в объект-команду, реализующий единый интерфейс execute(). Передаётся получателю для исполнения.

class Command {

public:

virtual void execute() = 0;

};

class PrintCommand : public Command {

Printer\* printer;

public:

PrintCommand(Printer\* p) : printer(p) {}

void execute() override { printer->print(); }

};

Принцип:

-Инкапсуляция: объект знает только команду, не знает, как она реализована.

-Разделяй и властвуй: отправитель и получатель разорваны логически.

-Ортогональность: логика UI (меню, кнопка) и логика действия (печать) разделены.

Многопоточность:

-Команды можно складывать в потокобезопасные очереди (например, из рабочих потоков в очередь задач).

-Используется в пуле потоков, GUI, обработке событий, логировании.

## 🔹 1.2. Стратегия (Strategy)

Проблема:

Нужно реализовать взаимозаменяемые алгоритмы (например, сортировки, фильтрации, проверки), чтобы можно было подставлять их на лету без изменения кода клиента.

Решение:

Создать семейство алгоритмов, инкапсулировать каждый в свой класс и сделать их взаимозаменяемыми через общий интерфейс.

class SortingStrategy {

public:

virtual void sort(std::vector<int>& data) = 0;

};

class QuickSort : public SortingStrategy {

void sort(std::vector<int>& data) override { /\* ... \*/ }

};

class BubbleSort : public SortingStrategy {

void sort(std::vector<int>& data) override { /\* ... \*/ }

};

Принцип:

-Инкапсуляция алгоритма отдельно от его использования.

-Ортогональность: UI не зависит от типа сортировки.

-Разделяй и властвуй: добавление новой стратегии не требует изменения клиентов.

Многопоточность:

-Каждый поток может использовать свою стратегию, без конфликтов.

-Можно применять разные стратегии в параллельных потоках: напр., быстрая сортировка на больших данных и пузырьковая — на малых.

## 🔹 1.3. Шаблонный метод (Template Method)

Проблема:

Много однотипных алгоритмов, имеющих общую структуру, но отличающихся в деталях. Нужно избежать дублирования кода и обеспечить гибкую реализацию шагов.

Решение:

Определить скелет алгоритма в базовом классе, а конкретные шаги реализовать в подклассах.

class DataProcessor {

public:

void process() {

load();

transform();

save();

}

virtual void load() = 0;

virtual void transform() = 0;

virtual void save() = 0;

};

Принцип:

-Инкапсуляция общей логики, с возможностью варьировать детали.

-Разделение ответственности между этапами алгоритма.

-Ортогональность: добавление новой реализации не меняет структуру process().

Многопоточность:

-Каждый поток может выполнять process() независимо.

-Важно защитить ресурсы (например, файл, БД) в load() / save() через мьютексы, чтобы избежать гонок.

2. Что такое архитектура ПО + влияние многопоточности

Архитектура программного обеспечения — это высокоуровневая структура системы, включающая:

-основные компоненты,

-связи между ними,

-принципы организации и взаимодействия,

-решения о распределении обязанностей и потоков данных.

Это скелет программы, определяющий её:

-модульность,

-расширяемость,

-тестируемость,

-производительность.

## 1. Разделение компонентов и ответственность

-Компоненты должны быть изолированы (Single Responsibility Principle).

-Потоки должны владеть своими данными или использовать потокобезопасные структуры.

2. Синхронизация и гонки

-Использование мьютексов, семафоров, барьеров.

-Нужно архитектурно заложить защиту от состояний гонки: через акторы, очереди сообщений и т.д.

## 3. Проектные шаблоны

-Command → используется в пулах потоков.

-Observer → для обработки событий.

-Producer-Consumer → для очередей заданий.

## 4. Распределение нагрузки Архитектура должна учитывать масштабируемость:

-независимые компоненты → независимые потоки;

-параллелизм → на уровне данных, операций, задач.

5. Отладка и тестирование

Архитектура должна упрощать логирование, трассировку и диагностику многопоточных операций.