МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА»
(САМАРСКИЙ УНИВЕРСИТЕТ)

С. В. Малахов

ВВЕДЕНИЕ В ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

Рекомендовано редакционно-издательским советом федерального государственного автономного образовательного учреждения высшего образования «Самарский национальный исследовательский университет имени академика С.П. Королева» в качестве учебного пособия для обучающихся по основной образовательной программе высшего образования по направлению подготовки 02.03.02 Фундаментальная информатика и информационные технологии

САМАРА
Издательство Самарского университета
2026

ISBN 978-5-7883-2305-3
© Самарский университет, 2026

УДК 004.4(075)
ББК 3973я7
М181

Рецензенты: д-р техн. наук, проф. В.Н. Тарасов;
канд. техн. наук, доц. Е.М. Мезенцева

Малахов, Сергей Валерьевич

М181 **Введение в паттерны проектирования:** учебное пособие / *С. В. Малахов*; Министерство науки и высшего образования Российской Федерации, Самарский университет. – Самара: Издательство Самарского университета, 2026. – 1 CD-ROM (2,75 Мб). – Загл. с титул. экрана. – Текст: электронный.

ISBN 978-5-7883-2305-3

В пособии рассматриваются основные принципы объектно-ориентированного программирования (ЯП Python и Java) и паттерны проектирования. Освещены ключевые концепции инкапсуляции, наследования и полиморфизма, а также практическое применение порождающих, структурных и поведенческих паттернов.

Материал предназначен для студентов бакалавриата, предполагая наличие базовых знаний в области программирования, алгоритмов, структур данных и принципов разработки программного обеспечения.

Предназначено для бакалавров и магистров, обучающихся по направлению 02.03.02 «Фундаментальная информатика и информационные технологии» (ФИИТ) и смежным IT-дисциплинам.

Подготовлено на кафедре программных систем.

УДК 004.4(075)
ББК 3973я7

Минимальные системные требования:
Процессор (CPU):
Тип: Intel Celeron, Pentium / AMD Athlon, Duron или совместимые (x86/x64);
Тактовая частота: от 1,0 ГГц и выше;
Оперативная память (RAM): 1–2 ГБ;
Операционная система: Microsoft Windows, MacOS, Linux.
Свободное место на жестком диске (HDD/SSD): от 120 МБ

© Самарский университет, 2026

Подписано для тиражирования 02.06.2026.
Объем издания 2,75 Мб.
Количество носителей 1 диск.
Тираж 11 дисков.

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С. П. КОРОЛЕВА»
(САМАРСКИЙ УНИВЕРСИТЕТ)
443086, Самара, Московское шоссе, 34.

Издательство Самарского университета.
443086, Самара, Московское шоссе, 34.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 5
1 ОСНОВНЫЕ ПРИНЦИПЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ 7
1.1 Инкапсуляция 7
1.2 Наследование 9
1.3 Полиморфизм 9
2 ВВЕДЕНИЕ В ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ 11
2.1 Паттерны проектирования 11
2.2 Проектирование с учетом будущих изменений 12
2.3 Выбор паттерна проектирования 14
2.4 Нотация схем классов 15
3 ПОРОЖДАЮЩИЕ ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ 21
3.1 Паттерн Abstract Factory (Абстрактная фабрика) 21
3.2 Паттерн Builder (Строитель) 25
3.3 Паттерн Factory Method (Фабричный метод) 28
3.4 Паттерн Singleton (Одиночка) 31
3.5 Паттерн Prototype (Прототип) 34
3.6 Контрольные вопросы для самоподготовки 37
4 СТРУКТУРНЫЕ ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ 41
4.1 Паттерн Adapter (Адаптер) 41
4.2 Паттерн Bridge (Мост) 44
4.3 Паттерн Composite (Компоновщик) 48
4.4 Паттерн Decorator (Декоратор) 51
4.5 Паттерн Facade (Фасад) 55
4.6 Паттерн Flyweight (Приспособленец) 58
4.7 Паттерн Proxy (Заместитель) 62
4.8 Контрольные вопросы для самоподготовки 65
5 ПАТТЕРНЫ ПОВЕДЕНИЯ 71
5.1 Паттерн Chain of Responsibility (Цепочка обязанностей) 71
5.2 Паттерн Command (Команда) 75
5.3 Паттерн Interpreter (Интерпретатор) 78
5.4 Паттерн Iterator (Итератор) 81
5.5 Паттерн Mediator (Посредник) 84
5.6 Паттерн Memento (Хранитель) 88
5.7 Паттерн Observer (Наблюдатель) 91
5.8 Паттерн State (Состояние) 94
5.9 Паттерн Strategy (Стратегия) 98
5.10 Паттерн Template Method (Шаблонный метод) 102
5.11 Паттерн Visitor (Посетитель) 105
5.12 Контрольные вопросы для самоподготовки 109
СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ 117

ВВЕДЕНИЕ

Современный этап развития информационных технологий характеризуется беспрецедентным ростом сложности программных систем. Переход от монолитных архитектур к распределенным микросервисным решениям, необходимость обработки колоссальных массивов данных и постоянное изменение бизнес-требований диктуют новые правила разработки программного обеспечения. В этих условиях написание кода перестает быть просто процессом перевода алгоритма на машинный язык и трансформируется в сложную инженерную дисциплину – программную инженерию и архитектурное проектирование.

Актуальность данного учебного пособия обусловлена острой необходимостью подготовки ИТ-специалистов, способных не просто писать синтаксически корректный код, но и создавать масштабируемые, легко поддерживаемые и устойчивые к изменениям программные системы. В промышленной разработке критически важным становится умение использовать проверенные временем архитектурные решения. Паттерны (шаблоны) проектирования представляют собой именно такие формализованные описания наиболее эффективных способов решения типовых задач объектно-ориентированного проектирования [1]. Владение ими является обязательным квалификационным требованием для разработчиков уровней Middle и Senior, а также базой для понимания принципов работы современных фреймворков и библиотек на языках Java и Python.

Предметом науки программной инженерии, в рамках которой изучаются паттерны проектирования, являются методы, инструменты и процессы промышленного конструирования программного обеспечения (ПО). В более узком смысле, в фокусе данной дисциплины находится архитектурный дизайн – процесс определения структурных компонентов системы, их интерфейсов и механизмов взаимодействия, обеспечивающих достижение заданных функциональных и нефункциональных требований (надежности, производительности, модифицируемости).

История становления и развития теории паттернов проектирования имеет уникальную траекторию, берущую начало за пределами информатики. Сама концепция «паттерна» была впервые предложена в 1970-х годах выдающимся архитектором-градостроителем Кристофером Александером. Он разработал язык шаблонов для решения типовых проблем архитектуры зданий и городов, утверждая, что каждый шаблон описывает проблему, постоянно возникающую в нашей среде, и суть ее решения. В конце 1980-х годов программисты Кент Бек и Уорд Каннингем адаптировали идеи Александера для разработки графических интерфейсов на языке Smalltalk. Подлинная же революция в программной инженерии произошла в 1994 году с выходом фундаментальной монографии «Паттерны проектирования: элементы повторно используемого объектно-ориентированного программного обеспечения» [2]. Ее авторы – Эрих Гамма, Ричард Хелм, Ральф Джонсон и Джон Влиссидес, вошедшие в историю ИТ как «Банда четырех» (Gang of Four, GoF) – систематизировали 23 классических паттерна. С тех пор каталог шаблонов непрерывно расширялся: появились паттерны корпоративных приложений (М. Фаулер) [3], паттерны интеграции, микросервисные шаблоны, однако классическая база GoF остается незыблемым фундаментом объектно-ориентированного дизайна.

Будучи комплексной дисциплиной, проектирование архитектуры ПО имеет глубокие связи с другими отраслями знаний. Оно опирается на методы системного анализа и теории систем (при декомпозиции сложных программных комплексов), дискретной математики и теории графов (при построении деревьев зависимостей и анализе связности классов), а также тесно

взаимодействует с эргономикой и когнитивной психологией (в части проектирования программных интерфейсов, понятных для других разработчиков). Кроме того, знание паттернов неразрывно связано с управлением проектами, так как стандартизация кода существенно снижает риски при ротации кадров в команде.

Изучение паттернов проектирования опирается на строгую преемственность использования знаний. Для успешного освоения материала данного пособия студент должен обладать компетенциями, полученными при изучении предшествующих дисциплин: «Основы программирования», «Алгоритмы и структуры данных» и «Дискретная математика». Критически важно уверенное владение базовыми парадигмами объектно-ориентированного программирования: понимание механизмов инкапсуляции, наследования и полиморфизма на примере языков Java или Python. В свою очередь, знания и практические навыки, формируемые в данном курсе, являются фундаментальной базой для освоения последующих курсов старших курсов и магистратуры, таких как «Архитектура корпоративных информационных систем», «Проектирование высоконагруженных систем», «Разработка клиент-серверных приложений», а также для успешного прохождения технологических и преддипломных практик, связанных с командной разработкой коммерческого ПО.

Методологический аппарат данного издания четко очерчен:

– объектом издания выступает процесс объектно-ориентированного проектирования и конструирования программных систем;

– предметом издания являются порождающие, структурные и поведенческие паттерны проектирования, а также методы их программной реализации и интеграции в архитектуру приложения с использованием языков Java и Python.

Цель издания – формирование у студентов глубокого теоретического понимания принципов объектно-ориентированного дизайна (в частности, принципов SOLID) и выработка устойчивых практических навыков применения паттернов проектирования для создания гибкой, расширяемой и легко поддерживаемой архитектуры программного обеспечения.

Для достижения поставленной цели в учебном пособии последовательно решаются следующие задачи:

1. Раскрыть фундаментальные принципы объектно-ориентированного дизайна, определяющие качество программной архитектуры (высокое зацепление, слабая связность, инверсия зависимостей).

2. Детально рассмотреть каталогизацию паттернов проектирования (GoF), проанализировав назначение, применимость и структуру порождающих, структурных и поведенческих шаблонов.

3. Продемонстрировать реализацию каждого паттерна на уровне исходного кода, представив сравнительный анализ подходов в статически типизированном языке (Java) и динамически типизированном языке (Python).

4. Сформировать у обучающихся навыки визуального моделирования архитектуры с применением языка UML (Unified Modeling Language) для описания структур классов и их взаимодействий.

5. Обеспечить закрепление теоретического материала посредством методически структурированных практических заданий, направленных на самостоятельное проектирование подсистем и рефакторинг неоптимального кода.

Предлагаемое учебное пособие призвано стать надежным путеводителем для начинающих инженеров-программистов, помогая им преодолеть барьер между написанием разрозненного кода и профессиональным проектированием надежных программных систем.

1 ОСНОВНЫЕ ПРИНЦИПЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

Объектно-ориентированное программирование (ООП) – это парадигма программирования, которая основывается на представлении программы как совокупности объектов, взаимодействующих между собой. На практике выполнение такой программы начинается с главной функции (например, main() в C++ или Java), которая выступает точкой входа: она создает базовые объекты системы и инициирует процесс их взаимодействия. Объекты представляют собой экземпляры классов, которые объединяют в себе свойства (данные, определяющие состояние) и поведение (которое реализуется через методы и конструкторы). Основные принципы ООП включают инкапсуляцию, наследование, полиморфизм и абстракцию [4].

1.1 Инкапсуляция

Инкапсуляция – это принцип, который заключается в сокрытии внутреннего состояния объекта (атрибутов/полей) и его поведения (методов) и предоставлении доступа к нему только через определенные методы (интерфейсы). Основная цель инкапсуляции – логически связать данные и методы их обработки в единую сущность. Совместно с механизмами управления доступом (сокрытием информации) это позволяет защитить внутреннее состояние объекта от некорректного прямого изменения извне.

Атрибуты (поля) объекта скрываются с помощью модификаторов доступа**. Модификаторы доступа** – это специальные ключевые слова (или обозначения), которые используются для управления доступом к классам, переменным и методам. Они позволяют определять, кто и откуда может использовать или изменять данные внутри программы. Необходимы для того, чтобы защитить данные от случайного или неправильного использования и сделать код более структурированным.

Python основывается на принципе «соглашения, а не принуждения» ("consenting adults"), и защита атрибутов или методов от внешнего доступа осуществляется с помощью соглашений об именовании. Доступ к защищенным атрибутам осуществляется с помощью *self*.

Уровни доступа в Python:

– публичные атрибуты и методы *public*. По умолчанию все атрибуты и методы класса в Python являются публичными. Они доступны из любого места: методам самого класса, через объекты (экземпляры) класса, а также из любого внешнего кода (других модулей, функций, сторонних классов или главной функции программы);

– защищённые атрибуты и методы *protected*. Защищённые атрибуты и методы обозначаются одним подчёркиванием (_). Они считаются «доступными для использования только внутри класса и его подклассов». Однако, защита на уровне языка не реализована: их всё равно можно вызвать извне. Это соглашение служит для уведомления разработчиков о том, что доступ к таким атрибутам или методам нежелателен.

```
class Parent:
    def __init__(self):
        self._protected_attribute = "Защищённый атрибут"
    def _protected_method(self):
        print("Это защищённый метод")
```

```
class Child(Parent):
    def access_protected(self):
        print(self._protected_attribute)
        self._protected_method()
child = Child()
child.access_protected()
print(child._protected_attribute)
```

– приватные атрибуты и методы *private*. Приватные атрибуты и методы обозначаются двумя подчёркиваниями (__). Это создаёт механизм искажения имени (name mangling): приватные атрибуты и методы получают специальные имена, что затрудняет их доступ извне. Они доступны только внутри класса и недоступны напрямую из экземпляра или подкласса. Однако, даже приватные атрибуты можно обойти, используя доступ через их «искажённые» имена.

```
class Example:
    def __init__(self):
        self.__private_attribute = "Приватный атрибут"
    def __private_method(self):
        print("Это приватный метод")
    def access_private(self):
        print(self.__private_attribute)
        self.__private_method()
obj = Example()
obj.access_private()
# print(obj.__private_attribute)
print(obj._Example__private_attribute)
```

В Java существует четыре уровня доступа, определяемые модификаторами (Таблица 1.1).

Таблица 1.1. Сравнение уровней доступа

Модификатор	Класс	Пакет	Наследники (в другом пакете)	Доступ
public	✓	✓	✓	публичный
protected	✓	✓	✓	защищённый
package-private (default)	✓	✓	✗	по умолчанию
private	✓	✗	✗	закрытый

```
public class Person {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.Age = age;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        if (name == null || name.trim().isEmpty()) {
                throw new RuntimeException("Name is empty");
                }
              this.name = name;
            }
```

Доступ к защищенным данным осуществляется с помощью *getter* (геттер) и *setter* (сеттер). Геттеры и сеттеры – это методы, которые предоставляют управляемый доступ к скрытым полям

(атрибутам) объекта в объектно-ориентированном программировании (ООП). Они являются частью механизма инкапсуляции, обеспечивая контроль над чтением и изменением состояния объекта.

Геттер (*getName*()) позволяет безопасно получать значения приватных атрибутов. Сеттер (*setName*()) позволяет изменять значения атрибутов с предварительной проверкой на корректность данных.

1.2 Наследование

Наследование – это механизм, который позволяет одному классу (называемому подклассом или производным классом) наследовать свойства и методы другого класса (называемого суперклассом или базовым классом). Наследование способствует повторному использованию кода и упрощает создание новых классов, которые могут расширять или изменять поведение существующих классов. Дочерний класс наследует все публичные и защищённые атрибуты и методы родительского класса. Позволяет переопределять методы родительского класса для специфического поведения, а также помогает избежать дублирования кода и создавать иерархии классов.

Пример на Python:

```
class Animal:
    def speak(self):
        print("Это животное издаёт звук")
class Dog(Animal):
    def speak(self):
        print("Собака лает")
animal = Animal()
animal.speak()
dog = Dog()
dog.speak()
```

Пример на Java:

```
class Animal {
    void eat() {
        System.out.println("Это животное издаёт звук");    }
}
class Dog extends Animal {
    void bark() {
        System.out.println("Собака лает");     }
}
```

1.3 Полиморфизм

Полиморфизм – это принцип, который позволяет объектам разных классов обрабатывать данные через единый интерфейс. Это означает, что один и тот же метод может быть реализован по-разному в разных классах, и вызов этого метода будет автоматически выполняться в соответствии с типом объекта, на котором он был вызван. Метод с одним и тем же названием может иметь разное поведение в зависимости от класса объекта. Повышает гибкость и расширяемость программы.

Пример на Python:

```
class Cat:
    def speak(self):
        print("Кот мяукает")
class Dog:
    def speak(self):
        print("Собака лает")
def animal_sound(animal):
    animal.speak()
cat = Cat()
dog = Dog()
animal_sound(cat)
animal_sound(dog)
```

Полиморфизм позволяет использовать один интерфейс для различных типов объектов. В Java полиморфизм достигается с помощью переопределения методов (override) [5]. Переопределение применяется, когда подкласс предоставляет свою реализацию метода, который уже определён в его суперклассе. Это позволяет одному интерфейсу работать с разными реализациями.

Пример на Java:

```
class Animal {
    public void speak() {
        System.out.println("Это животное издаёт звук");
    }
}
class Cat extends Animal {
    @Override
    public void speak() {
        System.out.println("Кот мяукает");
    }
}
class Dog extends Animal {
    @Override
    public void speak() {
        System.out.println("Собака лает");
    }}
```

Здесь метод *speak* в *Cat* и *Dog* переопределяет метод из *Animal*, и благодаря полиморфизму Java вызывает правильный метод для каждого типа объекта.

2 ВВЕДЕНИЕ В ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

2.1 Паттерны проектирования

При разработке сложного программного обеспечения инженеры регулярно сталкиваются с типовыми проблемами. Чтобы не «изобретать велосипед» каждый раз заново, в инженерию ПО была перенесена идея использования паттернов – универсальных настраиваемых шаблонов проектирования [6]. Они представляют собой проверенные на практике архитектурные конструкции, адаптируемые под конкретные нужды проекта. Как отмечал основоположник этой концепции Кристофер Александер.

По словам Кристофера Александера (Christopher Alexander), «любой паттерн описывает задачу, которая снова и снова возникает в нашей работе, а также принцип ее решения, причем таким образом, что это решение можно потом использовать миллион раз, и при этом никакие две реализации не будут полностью одинаковыми».

Под паттернами проектирования понимается описание взаимодействия объектов и классов, адаптированных для решения общей задачи проектирования в конкретном контексте.

Паттерны проектирования различаются степенью детализации и уровнем абстракции. Классификация паттернов по двум критериям (Таблица 2.1).

Таблица 2.1. Пространство паттернов проектирования

Класс / Уровень	Порождающие паттерны	Структурные паттерны	Паттерны поведения
Класс	Фабричный метод	Адаптер	Интерпретатор Шаблонный метод
Объект	Абстрактная фабрика Одиночка Прототип Строитель	Адаптер Декоратор Заместитель Компоновщик Мост Приспособленец Фасад	Итератор Команда Наблюдатель Посетитель Посредник Состояние Стратегия Хранитель Цепочка обязанностей

Первый – критерий, отражает назначение паттерна. Паттерны делятся на порождающие, структурные и паттерны поведения. Первые связаны с процессом создания объектов. Вторые имеют отношение к композиции объектов и классов. Паттерны поведения характеризуют то, как классы или объекты взаимодействуют.

Второй критерий – уровень, сообщает, к чему обычно применяется паттерн: к объектам или классам. Паттерны уровня классов описывают отношения между классами и их подклассами. Такие отношения выражаются с помощью наследования, поэтому они статичны, то есть зафиксированы на этапе компиляции. Паттерны уровня объектов описывают отношения между объектами, которые могут изменяться во время выполнения и потому более динамичны. Почти все паттерны в какой-то мере используют наследование. Поэтому к категории «паттерны классов» отнесены только те, что концентрируются лишь на отношениях между классами. Большинство паттернов действует на уровне объектов.

Порождающие паттерны классов частично делегируют ответственность за создание объектов своим подклассам, тогда как порождающие паттерны объектов передают ответственность другому объекту. Структурные паттерны классов используют наследование для составления классов, в то время как структурные паттерны объектов описывают способы сборки объектов из частей. Поведенческие паттерны классов используют наследование для описания алгоритмов и потока управления, а поведенческие паттерны объектов описывают, как объекты, принадлежащие некоторой группе, совместными усилиями выполняют задачу, которая ни одному отдельному объекту не под силу.

Примеры для лучшего понимания:

1. Для порождающих паттернов: на уровне классов действует «Фабричный метод» (*Factory Method*) – базовый класс задает интерфейс, а подкласс решает, какой именно объект создать. На уровне объектов работает «Абстрактная фабрика» (*Abstract Factory*) – задача создания делегируется совершенно другому объекту-фабрике.

2. Для структурных паттернов: на уровне классов реализуется «Адаптер» (*Adapter*) через множественное наследование интерфейсов. На уровне объектов – тот же «Адаптер», но через композицию, когда один объект «оборачивает» и вызывает другой внутри себя.

3. Для поведенческих паттернов: на уровне классов примером служит «Шаблонный метод» (*Template Method*) – базовый класс фиксирует каркас алгоритма, а подклассы меняют его шаги на этапе компиляции. На уровне объектов применяется «Стратегия» (*Strategy*) – объект получает алгоритм извне и может динамически менять его прямо во время выполнения программы.

Существуют и другие способы классификации паттернов. Некоторые паттерны часто используются вместе. Например, компоновщик применяется с итератором или посетителем. Некоторыми паттернами предлагаются альтернативные решения. Так, прототип нередко можно использовать вместо абстрактной фабрики. Применение части паттернов приводит к схожему дизайну, хотя изначально их назначение различно. Например, структурные диаграммы компоновщика и декоратора похожи.

2.2 Проектирование с учетом будущих изменений

Системы должны проектироваться с учетом их дальнейшего развития. Для проектирования системы, устойчивой к таким изменениям, следует предположить, как она будет изменяться на протяжении отведенного ей времени жизни.

Если при проектировании системы не принималась во внимание возможность изменений, то есть вероятность, что в будущем ее придется полностью перепроектировать. Это может повлечь за собой переопределение и новую реализацию классов, модификацию клиентов и повторный цикл тестирования. Перепроектирование отражается на многих частях системы, поэтому непредвиденные изменения всегда оказываются дорогостоящими.

Благодаря паттернам систему всегда можно модифицировать определенным образом. Каждый паттерн позволяет изменять некоторый аспект системы независимо от всех прочих, таким образом, она менее подвержена влиянию изменений конкретного вида.

Некоторые типичные причины перепроектирования, а также паттерны, которые позволяют этого избежать:

1. При создании объекта явно указывается класс. Задание имени класса привязывает к конкретной реализации, а не к конкретному интерфейсу. Это может осложнить изменение объекта в будущем. Чтобы уйти от такой проблемы, объекты создаются косвенно.

При создании объекта (например, через оператора *new*) явно указывается класс. Задание имени класса жестко привязывает код к конкретной реализации. Это осложняет изменение и масштабирование системы в будущем.

Правильный архитектурный подход заключается в опоре на интерфейсы (или абстрактные классы). Интерфейс проектируется самостоятельно как независимый контракт, описывающий только то, что должен делать объект. Конкретные классы реализуют этот контракт, определяя, как они это делают. Если клиентский код работает только с интерфейсом, разработчик может в любой момент подменить класс-реализацию без изменения остальной программы.

Чтобы реализовать этот принцип на практике и уйти от проблемы жесткой привязки при инстанцировании, объекты создаются косвенно с помощью порождающих паттернов.

Паттерны проектирования: абстрактная фабрика, фабричный метод, прототип.

2. Зависимость от конкретных операций. Задавая конкретную операцию, вы ограничиваете себя единственным способом выполнения запроса. Если же не включать запросы в код, то будет проще изменить способ удовлетворения запроса как на этапе компиляции, так и на этапе выполнения.

Паттерны проектирования: цепочка обязанностей, команда.

3. Зависимость от аппаратной и программной платформ. Внешние интерфейсы операционной системы и интерфейсы прикладных программ (API) различны на разных программных и аппаратных платформах. Если программа зависит от конкретной платформы, ее будет труднее перенести на другие. Возможно, даже на «родной» платформе такую программу трудно поддерживать. Поэтому при проектировании систем так важно ограничивать платформенные зависимости.

Паттерны проектирования: абстрактная фабрика, мост.

4. Зависимость от представления или реализации объекта. Если клиент располагает информацией о том, как объект представлен, хранится или реализован, то, возможно, при изменении объекта придется изменять и клиента. Сокрытие этой информации от клиентов поможет уберечься от каскадных изменений.

Паттерны проектирования: абстрактная фабрика, мост, хранитель, заместитель.

5. Зависимость от алгоритмов. Во время разработки и последующего использования алгоритмы часто расширяются, оптимизируются и заменяются. Зависящие от алгоритмов объекты придется переписывать при каждом изменении алгоритма. Поэтому алгоритмы, которые с большой вероятностью будут изменяться, следует изолировать.

Паттерны проектирования: мост, итератор, стратегия, шаблонный метод, посетитель.

6. Сильная связанность. Сильно связанные между собой классы трудно использовать порознь, так как они зависят друг от друга. Сильная связанность приводит к появлению монолитных систем, в которых нельзя ни изменить, ни удалить класс без знания деталей и модификации других классов. Такую систему трудно изучать, переносить на другие платформы и сопровождать.

Слабая связанность повышает вероятность того, что класс можно будет повторно использовать сам по себе. При этом изучение, перенос, модификация и сопровождение системы намного упрощаются. Для поддержки слабосвязанных систем в паттернах проектирования применяются такие методы, как абстрактные связи и разбиение на слои.

Паттерны проектирования: абстрактная фабрика, мост, цепочка обязанностей, команда, фасад, посредник, наблюдатель.

7. Расширение функциональности за счет порождения подклассов. Специализация объекта путем создания подкласса часто оказывается непростым делом. С каждым новым подклассом связаны фиксированные издержки реализации (инициализация, очистка и т. д.). Для определения подкласса необходимо так же ясно представлять себе устройство родительского класса. Например, замещение одной операции может потребовать замещения и других. Замещение операции может оказаться необходимым для того, чтобы можно было вызвать унаследованную операцию. Кроме того, порождение подклассов ведет к разрастанию количества классов, поскольку даже для реализации простого расширения приходится создавать новые подклассы.

Композиция объектов и делегирование – гибкие альтернативы наследованию для комбинирования поведений. Приложению можно добавить новую функциональность, меняя способ композиции объектов, а не определяя новые подклассы уже имеющихся классов. С другой стороны, интенсивное использование композиции объектов может усложнить понимание кода. С помощью многих паттернов проектирования удается построить такое решение, где специализация достигается за счет определения одного подкласса и комбинирования его экземпляров с уже существующими.

Паттерны проектирования: мост, цепочка обязанностей, компоновщик, декоратор, наблюдатель, стратегия.

8. Неудобства при изменении классов. Иногда нужно модифицировать класс, но делать это неудобно. Допустим, вам нужен исходный код, а он недоступен (так обстоит дело с коммерческими библиотеками классов). Или любое изменение тянет за собой модификации множества существующих подклассов. Благодаря паттернам проектирования можно модифицировать классы и при таких условиях.

Паттерны проектирования: адаптер, декоратор, посетитель.

2.3 Выбор паттерна проектирования

Если учесть, что каталог содержит более 20 паттернов, выбрать паттерн, лучше всего подходящий для решения конкретной задачи проектирования, будет непросто. Ниже представлены некоторые подходы к выбору подходящего паттерна:

4. Продумать, как паттерны решают проблемы проектирования. Как с помощью паттернов найти подходящие объекты, определить нужную степень их детализации, специфицировать их интерфейсы.

5. Просмотреть разделы каталога, описывающие назначение паттернов. В каких типичных ситуациях он будет полезен.

6. Изучить взаимосвязи паттернов (рисунок 2.1). Очень часто паттерны не существуют в изоляции и могут использоваться вместе для решения комплексных задач проектирования. Взаимосвязи между паттернами показывают, как один паттерн может помочь реализовать или дополнить другой.

7. Проанализировать паттерны со сходными целями. Каталог состоит из трех частей: порождающие паттерны, структурные паттерны и паттерны поведения.

8. Разобрать причины, вызывающих перепроектирование (2.2). Изучение причин перепроектирования помогает предотвратить возникновение ошибок в проекте и выбрать паттерн, который обеспечит гибкость и масштабируемость.

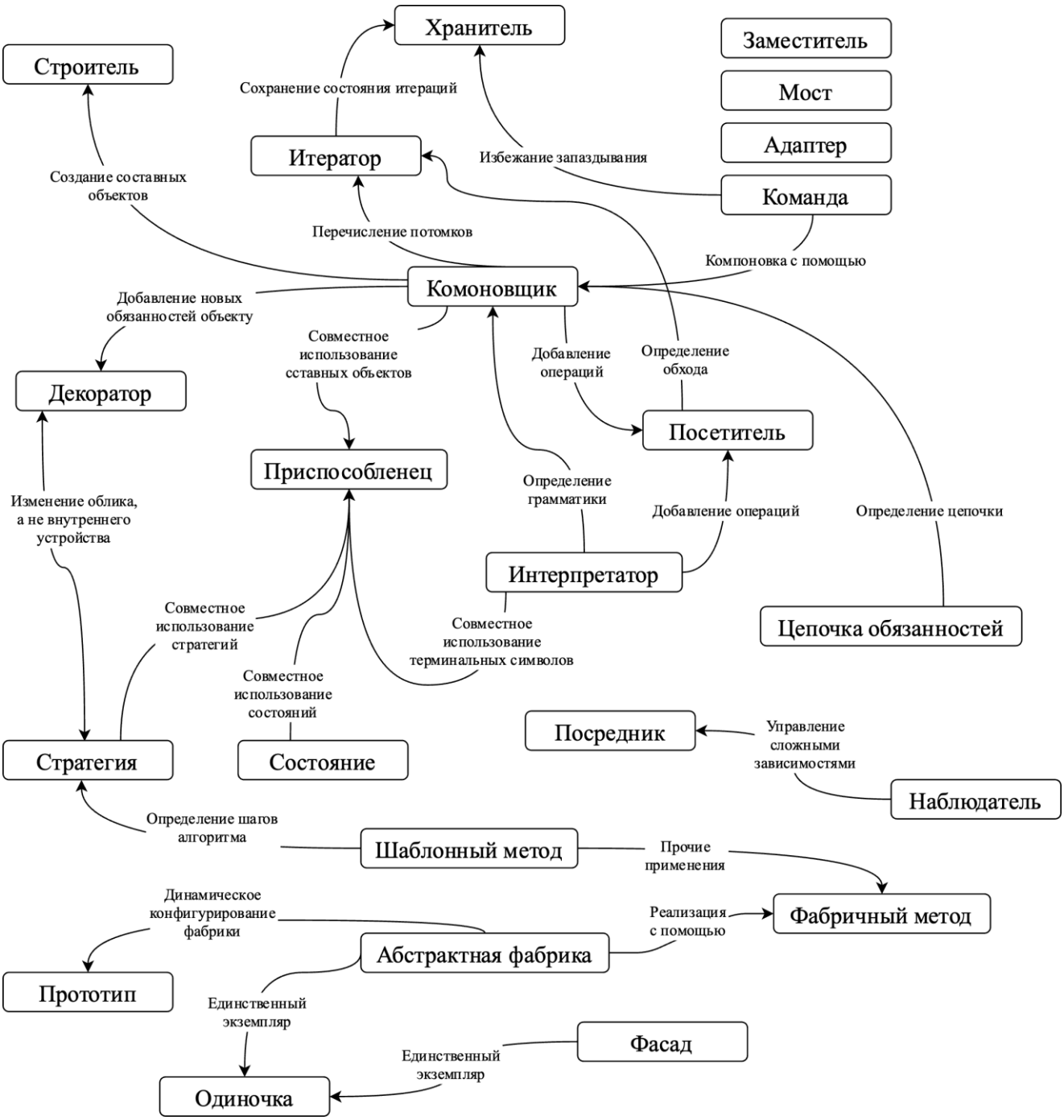

Рисунок 2.1 – Отношения между паттернами проектирования

9. Просмотреть, какие аспекты дизайна могут измениться. Те аспекты дизайна, которые разные паттерны позволяют модифицировать независимо и без перепроектирования.

2.4 Нотация схем классов

2.4.1 Диаграмма классов

UML – унифицированный язык моделирования (Unified Modeling Language) – это система обозначений, которую можно применять для объектно-ориентированного анализа и проектирования. Его можно использовать для визуализации, спецификации, конструирования и документирования программных систем. UML включает три вида строительных блоков:

диаграммы, сущности и связи. **Сущности** – это абстракции, которые являются основными элементами модели, **связи** соединяют их между собой, а **диаграммы** группируют представляющие интерес наборы сущностей [7].

Диаграмма – это графическое представление набора элементов, чаще всего изображенного в виде связного графа вершин (сущностей) и путей (связей). Язык UML включает 13 видов диаграмм, среди которых на первом месте в списке – диаграмма классов.

Все сущности реального мира, с которыми собирается работать программист, должны быть представлены объектами классов в программе [8]. При этом у каждого класса должно быть только одно назначение и уникально осмысленное имя, которое будет связано с этой целью**. Диаграммы классов** показывают набор классов, интерфейсов, а также их связи. Диаграммы этого вида чаще всего используются для моделирования объектно-ориентированных систем. Они предназначены для статического представления системы (рисунок 2.2). Пояснение к диаграмме с учетом отношений между классами представлено в 2.4.5.

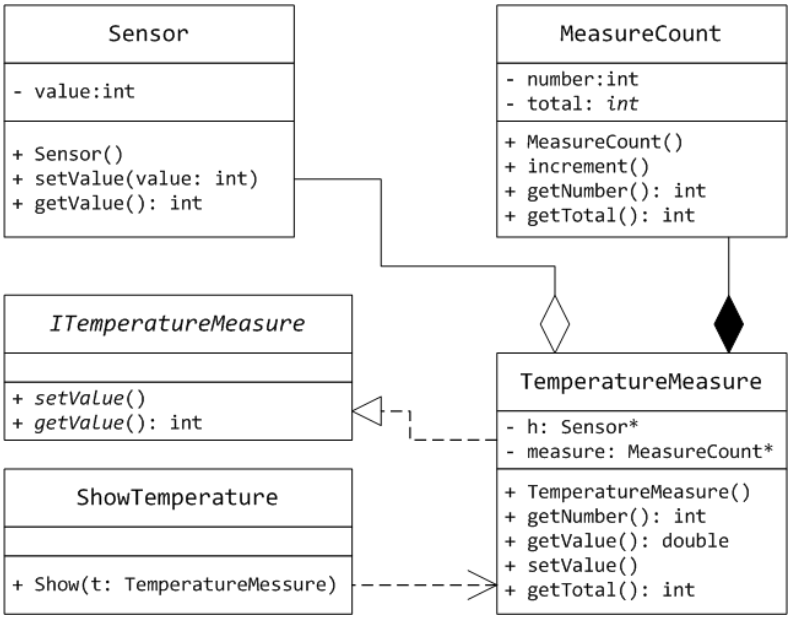

Рисунок 2.2 – Диаграмма классов

2.4.2 Элемент диаграммы – Класс

Класс – элемент диаграммы, обозначающий множество объектов, обладающих одинаковой внутренней структурой, поведением и отношениями с объектами других классов. Изображается класс на диаграмме в виде прямоугольника, разделённого на три секции (рисунок 2.3):

1) имя класса,
2) список атрибутов (полей) класса,
3) список методов (операций) класса.

Класс, в котором есть только статические поля и методы и на основе которого не создаются объекты, называется **статическим классом**. Чтобы показать на диаграмме, что наш класс статический, нужно добавить к имени модификатор «*utility*». Название модификатора «*utility*» связано с тем, статический класс предоставляет набор утилит, которые могут быть использованы любыми классами, которые в них нуждаются [9].

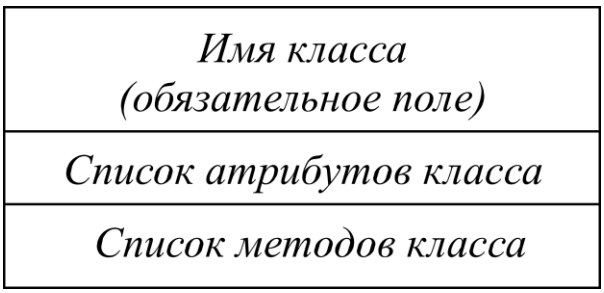

Рисунок 2.3 – Изображение класса на диаграмме

Класс, который является базовым для других классов и объекты которого мы не собираемся создавать, называют **абстрактным.** Абстрактный класс на диаграмме изображается так же, как и обычной класс, однако имя такого класса должно быть записано курсивом.

2.4.3 Список полей класса

Это именованный атрибут (поле) класса, описывающее диапазон значений, которые может принимать экземпляр атрибута. Класс может иметь любое число атрибутов или не иметь ни одного. В последнем случае блок атрибутов оставляют пустым. Атрибут представляет некоторое свойство моделируемой сущности, которым обладают все объекты данного класса. Имя атрибута, как и имя класса, может представлять собой текст. Статические атрибуты класса обозначаются подчеркиванием.

Каждое поле класса должно описываться следующим образом, как на рисунке 2.4

<уровень видимости><идентификатор>[кратность]:<тип поля>=<начальное значение>{свойство}
<visibility><identificator>(multiplicity):<type>=<initial value>{property}

Рисунок 2.4 – Общий вид поля класса

Уровень видимости (от англ. "*visibility*") – свойство поля, которое показывает, из какой части программы можно обратиться к данному полю: "+" – открытое поле (аналог *public*), "-" – закрытое поле (аналог *private*) и "#" – защищённое поле (аналог *protected*).

Идентификатор (от англ. "*identificator*") – название поля. Является обязательным элементом для описания переменной на диаграмме классов, поскольку однозначно её определяет (все идентификаторы на диаграммах уникальны).

Кратность (от англ. "*multiplicity*") – интервал, определяющий диапазон количества элементов в массиве. Если для поля указана кратность, то его следует считать массивом. Количество элементов в таком массиве и будет определяться указанным интервалом.

Тип поля (англ. "*type of field*") показывает, какой тип имеет данное поле в нашей программе. На ранней стадии проектирования можно и не уточнять, какой тип имеет то или иное поле.

2.4.4 Список методов класса

Операция (метод) – это реализация метода класса. Класс может иметь любое число операций либо не иметь ни одной. Часто вызов операции объекта изменяет его атрибуты.

Допускается указание только имен операций. Имя операции, как и имя класса, должно представлять собой текст. На практике для именования операции используются короткие глагольные конструкции, описывающие некое поведение класса, которому принадлежит операция. Обычно каждое слово в имени операции пишется с заглавной буквы, за исключением первого, например, *isEmpty* [10].

Можно специфицировать операцию, устанавливая ее сигнатуру, включающую имя, тип и значение по умолчанию всех параметров, а применительно к функциям – тип возвращаемого значения.

На рисунке 2.5 представлен общий вид описания метода класса и описание аргументов метода.

<уровень видимости><идентификатор>(список аргументов):<тип возвращающего значения>
<visibility><identificator>(list of arguments):<return type>
<имя аргумента>:<тип аргумента>
<name of argument>:<type>

Рисунок 2.5 – Общий вид поля методы

2.4.5 Виды отношений

Отношения (связи) представляют собой базовые строительные блоки для описания отношений (рисунок 2.6) в UML, используемые для разработки хорошо согласованных моделей.

Отношение ассоциации (рисунок 2.7) используют, чтобы показать, что между классами (например, между двумя классами) существует некоторая связь. Обычно с помощью него на диаграмме классов показывают, что один класс пользуется функционалом другого класса. Кратность ассоциации обозначает количество объектов, которые участвуют во взаимодействии.

Отношение зависимости используют, чтобы показать, что изменение одного класса требует изменение другого класса. Стрелка отношения зависимости направлена от зависимого класса к независимому.

Рисунок 2.6 – Виды отношений

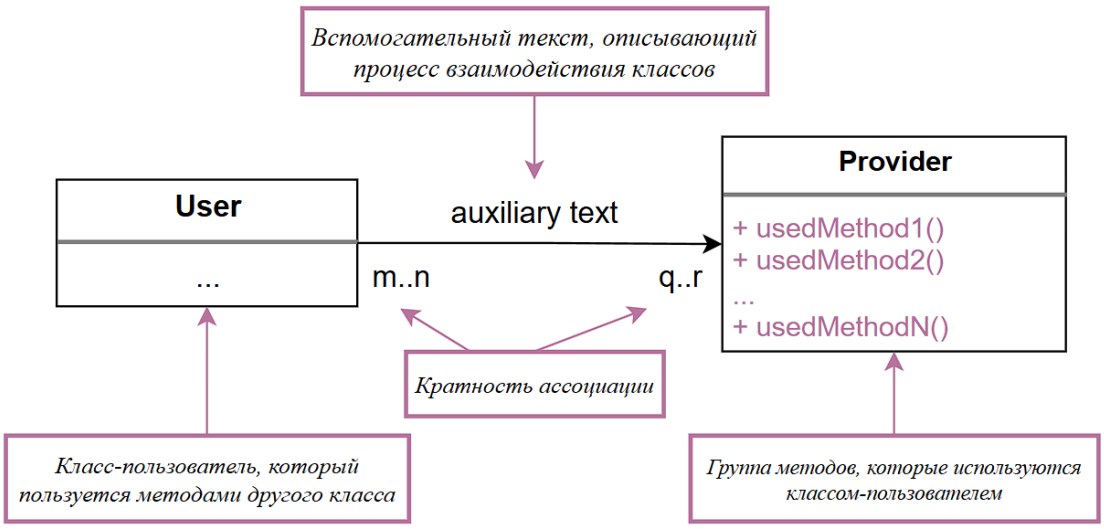

Рисунок 2.7 – Отношение ассоциации

Отношение наследования (рисунок 2.8) используется, чтобы показать, что один класс является родителем (базовым классом или суперклассом) для другого класса (потомка, производного класса).

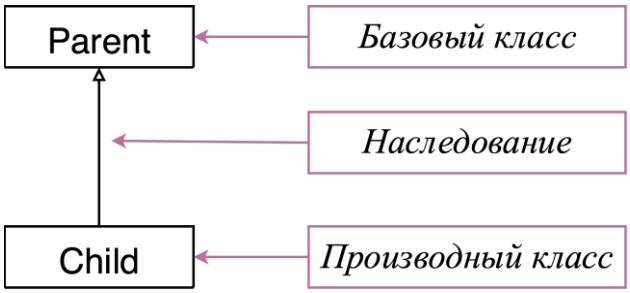

Рисунок 2.8 – Отношение наследования

Отношение агрегации (рисунок 2.8) между двумя классами показывает, что один из них включает в себя другой класс в качестве составной части. При этом класс-часть может существовать обособленно от класса-целого.

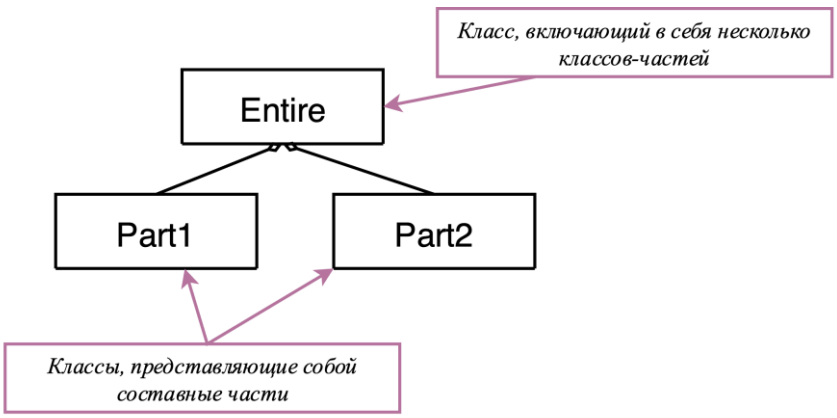

Рисунок 2.9 – Отношение агрегации

Отношение композиции является частным случаем отношения агрегации. Однако у него есть одно отличие – классы-части, которые он соединяет с классом-целым, не могут существовать обособленно.

Реализация – это семантическая связь между классами, когда один из них (поставщик) определяет соглашение, которого второй (клиент) обязан придерживаться. Это связи между интерфейсами и классами, которые реализуют эти интерфейсы. Это, своего рода, отношение «целое-часть». Поставщик, как правило, представлен абстрактным классом. В графическом исполнении связь реализации – это гибрид связей обобщения и зависимости: треугольник указывает на поставщика, а второй конец пунктирной линии – на клиента.

На диаграмме классов (рисунок 2.2) основным классом является класс *TemperatureMeasure*, который и является измерителем температуры. В качестве измеренного значения формируется среднее арифметическое всех измерений – сумма всех измерений, деленная на их количество.

Для получения измерений и их суммирования используется класс *Sensor* (в качестве датчика температуры). В консольной задаче сами измерения передаются в этот класс для суммирования. Класс состоит в отношении агрегации с основным классом *TemperatureMeasure*: сначала создается объект класса *Sensor*, а потом передаем его в качестве параметра конструктора классу *TemperatureMeasure*, чтобы использовать его в качестве части класса.

Количество измерений формируется классом *MeasureCount*, который содержит статическое поле *total* для подсчета общего измерений, а также поле *count* для подсчета количества измерителей конкретного объекта *TemperatureMeasure*. Класс *MeasureCount* находится в отношении композиции с классом *TemperatureMeasure*: объект *MeasureCount* создается непосредственно при создании объекта *TemperatureMeasure* (в его конструкторе).

Класс *ITemperatureMeasure* представляет собой интерфейс класса *TemperatureMeasure* и является своего рода поставщиком в отношении реализации.

Наконец, класс *ShowTemperature* находится в отношении зависимости с классом *TemperatureMeasure*, поскольку реализация единственного метода *Show* класса *ShowTemperature* зависит от структуры класса *TemperatureMeasure*.

3 ПОРОЖДАЮЩИЕ ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

Порождающие паттерны проектирования абстрагируют процесс создания экземпляров. Они позволяют сделать систему независимой от способа создания, композиции и представления объектов. Паттерн, порождающий классы, использует наследование, чтобы варьировать класс создаваемого экземпляра, а паттерн, порождающий объекты, делегирует создание экземпляров другому объекту. Эти паттерны начинают играть более важную роль, когда система эволюционирует и начинает в большей степени зависеть от композиции объектов, чем от наследования классов. При этом основной акцент смещается с жесткого кодирования фиксированного набора поведений на определение небольшого набора фундаментальных поведений, посредством композиции которых можно получить любое число более сложных. Таким образом, для создания объектов с конкретным поведением требуется нечто большее, чем простое создание экземпляра класса [11].

Для порождающих паттернов характерны два аспекта. Во-первых, эти паттерны инкапсулируют знания о конкретных классах, которые применяются в системе. Во-вторых, они скрывают подробности создания и компоновки экземпляров этих классов. Единственная информация об объектах, известная системе, – это их интерфейсы, определенные с помощью абстрактных классов. Следовательно, порождающие паттерны обеспечивают большую гибкость в отношении того, что создается, кто это создает, как и когда. Это позволяет настроить систему «готовыми» объектами с самой различной структурой и функциональностью статически (на этапе компиляции) или динамически (во время выполнения).

3.1 Паттерн Abstract Factory (Абстрактная фабрика)

3.1.1 Описание паттерна Abstract Factory (Абстрактная фабрика)

Это порождающий паттерн (шаблон) проектирования, который позволяет создавать семейства связанных объектов без указания их конкретных классов. Этот паттерн используется, когда системе нужно создавать различные продукты, относящиеся к одному семейству, но фактический тип этих продуктов определяется во время выполнения.

Основные условия для применения паттерна абстрактная фабрика:

1) система не должна зависеть от того, как создаются, компонуются и представляются входящие в нее объекты;

2) система должна настраиваться одним из семейств объектов;

3) входящие в семейство взаимосвязанные объекты спроектированы для совместной работы, и вы должны обеспечить выполнение этого ограничения;

4) необходимо предоставить библиотеку объектов, раскрывая только их интерфейсы, но не реализацию.

Основные элементы паттерна:

1. *Abstract Factory* (Абстрактная фабрика) – это интерфейс или абстрактный класс, который определяет набор методов для создания связанных объектов.

2. *Concrete Factories* (Конкретные фабрики) – реализуют методы абстрактной фабрики для создания конкретных объектов.

3. *Abstract Products* (Абстрактные продукты) – определяют интерфейсы для продуктов, которые могут быть созданы фабриками.

4. *Concrete Products* (Конкретные продукты) – это реальные классы, которые создаются конкретными фабриками.

5. *Client* (Клиент) – использует абстрактную фабрику для создания продуктов, не завися от их конкретных классов.

6. Основные отношения между сущностями в UML-диаграмме классов для данного паттерна, следующие: ассоциация – фабрика создает семейства связанных объектов (продуктов); реализация – конкретные фабрики реализуют интерфейс абстрактной фабрики.

Результаты паттерна:

1. Изолирует конкретные классы. Паттерн помогает контролировать классы объектов, создаваемых приложением. Поскольку фабрика инкапсулирует ответственность за создание классов и сам процесс их создания, то она изолирует клиента от подробностей реализации классов. Клиенты манипулируют экземплярами через их абстрактные интерфейсы. Имена изготавливаемых классов известны только конкретной фабрике, в коде клиента они не упоминаются.

2. Упрощает замену семейств продуктов. Класс конкретной фабрики появляется в приложении только один раз: при создании экземпляра. Это облегчает замену используемой приложением конкретной фабрики. Приложение может изменить конфигурацию продуктов, просто подставив новую конкретную фабрику. Поскольку абстрактная фабрика создает все семейство продуктов, то и заменяется сразу все семейство.

3. Гарантирует сочетаемость продуктов. Если продукты некоторого семейства спроектированы для совместного использования, то важно, чтобы приложение в каждый момент времени работало только с продуктами единственного семейства. Класс *AbstractFactory* позволяет легко соблюсти это ограничение.

4. Не упрощает задачу поддержки нового вида продуктов. Расширение абстрактной фабрики для изготовления новых видов продуктов – непростая задача. Дело в том, что интерфейс *AbstractFactory* фиксирует набор продуктов, которые можно создать. Для поддержки новых продуктов необходимо расширить интерфейс фабрики, то есть изменить класс *AbstractFactory* и все его подклассы.

Недостатки паттерна, это сложность добавления новых типов продуктов, так как это требует изменения всех конкретных фабрик. Много дополнительных классов, увеличивающих сложность системы. Трудно поддерживать, если продукты имеют сложные зависимости.

Пример из реальной жизни: строительство дома. Нужны двери, окна и крыша. Если строится деревянный дом, понадобятся деревянные двери, деревянные окна и деревянная крыша. Если строится каменный дом, то нужны каменные двери, каменные окна и каменная крыша. Логика выбора материала должна быть скрыта от строителей – они просто будут использовать фабрику, которая создаст все необходимые компоненты для дома.

3.1.2 Пример на Python

Есть приложение, которое может работать с разными интерфейсами: например, с Windows и macOS. Каждая из этих операционных систем имеет свои специфические элементы управления, например, кнопки и окна. В приложении могут быть кнопки и окна, но их реализация должна зависеть от операционной системы. Абстрактная фабрика – позволяет создавать соответствующие объекты (например, кнопку для Windows или кнопку для macOS) в зависимости от окружения.

```
class Button:
    def click(self):
        pass
class Window:
    def open(self):
        pass
class WindowsButton(Button):
    def click(self):
        return "Нажата кнопка Windows"
class WindowsWindow(Window):
    def open(self):
        return "Открыто окно Windows"
class MacOSButton(Button):
    def click(self):
        return "Нажата кнопка MacOS"
class MacOSWindow(Window):
    def open(self):
        return "Открыто окно MacOS"
class GUIFactory:
    def create_button(self) -> Button:
        pass
    def create_window(self) -> Window:
        pass
class WindowsFactory(GUIFactory):
    def create_button(self) -> Button:
        return WindowsButton()
    def create_window(self) -> Window:
        return WindowsWindow()
class MacOSFactory(GUIFactory):
    def create_button(self) -> Button:
        return MacOSButton()
    def create_window(self) -> Window:
        return MacOSWindow()
class Application:
    def __init__(self, factory: GUIFactory):
        self.button = factory.create_button()
        self.window = factory.create_window()
    def run(self):
        print(self.button.click())
        print(self.window.open())
windows_app = Application(WindowsFactory())
windows_app.run()
mac_app = Application(MacOSFactory())
mac_app.run()
```

Пояснение:

– *GUIFactory* – это абстрактная фабрика, которая описывает методы для создания кнопок и окон;

– *WindowsFactory* и *MacOSFactory* – это конкретные фабрики, которые реализуют методы создания кнопок и окон для каждой платформы;

– клиентский код использует абстрактную фабрику для создания объектов, не зная, какая конкретная фабрика будет использована. Он просто работает с объектами, предоставленными фабрикой, через их интерфейсы.

3.1.3 Пример на Java

Аналогичный пример на Java с использованием паттерна Абстрактная фабрика. В этом примере создаются интерфейсы для кнопок и окон, а затем их реализация для Windows и macOS. Затем создается абстрактная фабрика и её конкретные реализации для каждой операционной системы.

```
public interface Button {
    void click();
}
public interface Window {
    void open();
}
public class WindowsButton implements Button {
    @Override
    public void click() {
        System.out.println("Нажата кнопка Windows");
    }
}
public class WindowsWindow implements Window {
    @Override
    public void open() {
        System.out.println("Открыто окно Windows");
    }
}
public class MacOSButton implements Button {
    @Override
    public void click() {
        System.out.println("Нажата кнопка MacOS");
    }
}
public class MacOSWindow implements Window {
    @Override
    public void open() {
        System.out.println("Открыто окно MacOS");
    }
}
public interface GUIFactory {
    Button createButton();
    Window createWindow();
}
public class WindowsFactory implements GUIFactory {
    @Override
    public Button createButton() {
        return new WindowsButton();
    }
    @Override
    public Window createWindow() {
        return new WindowsWindow();
    }
}
public class MacOSFactory implements GUIFactory {
    @Override
    public Button createButton() {
        return new MacOSButton();
    }
```

```
    @Override
    public Window createWindow() {
        return new MacOSWindow();
    }
}
public class Application {
    private Button button;
    private Window window;
    public Application(GUIFactory factory) {
        button = factory.createButton();
        window = factory.createWindow();
    }
    public void run() {
        button.click();
        window.open();
    }
    public static void main(String[] args) {
        GUIFactory windowsFactory = new WindowsFactory();
        Application windowsApp = new Application(windowsFactory);
        windowsApp.run();
        GUIFactory macFactory = new MacOSFactory();
        Application macApp = new Application(macFactory);
        macApp.run();
    }
}
```

3.2 Паттерн Builder (Строитель)

3.2.1 Описание паттерна Builder (Строитель)

Это порождающий паттерн проектирования, который используется для поэтапного создания сложных объектов. Он позволяет создавать объект пошагово, раздельно управляясь с его составными частями, при этом итоговый объект может быть изменен без изменения клиентского кода. Этот паттерн особенно полезен, когда процесс создания объекта состоит из множества шагов или, когда требуется гибкость в процессе сборки различных представлений одного и того же объекта.

Основные условия для применения паттерна абстрактная фабрика:

1) алгоритм создания сложного объекта не должен зависеть от того, из каких частей состоит объект и как они стыкуются между собой;

2) процесс конструирования должен обеспечивать различные представления конструируемого объекта.

Основные элементы паттерна:

1. *Builder* (Строитель) – интерфейс или абстрактный класс, который определяет шаги создания объекта. Обычно каждый метод возвращает самого себя, что позволяет выстраивать цепочку вызовов.

2. *ConcreteBuilder* (Конкретный строитель) – конкретные реализации строителей, которые выполняют конкретные шаги по сборке объекта.

3. *Product* (Продукт) – сам объект, который строится. Это конечный сложный объект, который должен быть построен.

4. *Director* (Руководитель) – объект, который управляет процессом сборки. Он определяет, в каком порядке и какие шаги использовать для создания объекта. Не является обязательной частью паттерна.

Основные отношения между сущностями в UML-диаграмме классов для данного паттерна, следующие: ассоциация – директор использует строителя для создания объекта; агрегация – строитель имеет ссылку на объект, который он строит.

Результаты паттерна:

1. Позволяет изменять внутреннее представление продукта. Объект *Builder* предоставляет распорядителю абстрактный интерфейс для конструирования продукта, за которым он может скрыть представление и внутреннюю структуру продукта, а также процесс его сборки. Поскольку продукт конструируется через абстрактный интерфейс, то для изменения внутреннего представления достаточно всего лишь определить новый вид строителя.

2. Изолирует код, реализующий конструирование и представление. Паттерн Строитель улучшает модульность, инкапсулируя способ конструирования и представления сложного объекта. Клиентам ничего не надо знать о классах, определяющих внутреннюю структуру продукта; эти классы не входят в интерфейс строителя.

3. Предоставляет более точный контроль над процессом конструирования. В отличие от порождающих паттернов, которые сразу конструируют весь объект целиком, строитель делает это шаг за шагом под управлением распорядителя. И лишь когда продукт завершен, распорядитель забирает его у строителя. Поэтому интерфейс строителя в большей степени отражает процесс конструирования продукта, нежели другие порождающие паттерны. Это позволяет обеспечить более тонкий контроль над процессом конструирования, а значит, и над внутренней структурой готового продукта.

Недостатки паттерна, это увеличение числа классов, если требуется создавать разные типы объектов с различными конфигурациями. Не подходит, если объект должен быть собран целиком сразу, без поэтапной настройки. Могут возникнуть сложности, если объект имеет зависимости, которые трудно инкапсулировать в строителе.

Пример из реальной жизни: заказ пиццы в ресторане. Пицца – это сложный объект, который может включать множество параметров: размер, вид теста, соусы, начинки (сыр, мясо, овощи и т. д.). Однако, тебе не обязательно сразу сообщать все параметры. Ты можешь постепенно добавлять ингредиенты или менять параметры, пока не получишь пиццу, которую хочешь.

3.2.2 Пример на Python

Необходимо создать пиццу с различными опциями: сыром, ветчиной, оливками и т.д. С помощью паттерна *Builder* можно легко управлять процессом создания пиццы, добавляя или не добавляя нужные ингредиенты.

```
class Pizza:
    def __init__(self, dough=None, sauce=None, topping=None):
        self.dough = dough
        self.sauce = sauce
        self.topping = topping
    def __str__(self):
        return f'Pizza with dough: {self.dough}, sauce:
{self.sauce}, topping: {self.topping}'
class PizzaBuilder:
    def __init__(self):
```

```
        self.dough = None
        self.sauce = None
        self.topping = None
    def set_dough(self, dough):
        self.dough = dough
        return self
    def set_sauce(self, sauce):
        self.sauce = sauce
        return self
    def set_topping(self, topping):
        self.topping = topping
        return self
    def build(self):
        return Pizza(self.dough, self.sauce, self.topping)
    builder = PizzaBuilder()
    pizza = builder.set_dough("Thin
    crust").set_sauce("Tomato").set_topping("Cheese +
Pepperoni").build()
    print(pizza)
```

Пояснение:

– в классе *Pizza* каждый атрибут (*dough*, *sauce*, *topping*) описывает составляющие пиццы. Метод *__str__*() служит для удобного вывода информации о созданной пицце;

– в классе *PizzaBuilder* методы *set_dough*(), *set_sauce*() и *set_topping*() позволяют пошагово задавать параметры пиццы. Все методы возвращают самого себя (*self*), что позволяет вызывать их в цепочке (цепочка вызовов). Метод *build*() создает и возвращает объект *Pizza*, передавая текущие значения полей;

– создание объекта *PizzaBuilder* и настройка пиццы с помощью цепочки методов. В конце вызывается *build*(), который создает и возвращает готовую пиццу.

3.2.3 Пример на Java

Аналогичный пример на Java с использованием паттерна *Builder* (Строитель).

```
public class Pizza {
    private String dough;   // Тесто
    private String sauce;   // Соус
    private String topping; // Начинка
    private Pizza(PizzaBuilder builder) {
        this.dough = builder.dough;
        this.sauce = builder.sauce;
        this.topping = builder.topping;
    }
    @Override
    public String toString() {
        return "Pizza with dough: " + dough + ", sauce: " + sauce + ",
topping: " + topping;
    }
    public static class PizzaBuilder {
        private String dough;
        private String sauce;
        private String topping;
        public PizzaBuilder setDough(String dough) {
            this.dough = dough;
            return this;
        }
```

```
        public PizzaBuilder setSauce(String sauce) {
            this.sauce = sauce;
            return this;
        }
        public PizzaBuilder setTopping(String topping) {
            this.topping = topping;
            return this;
        }
        public Pizza build() {
            return new Pizza(this);
        }
    }
}
public class Main {
    public static void main(String[] args) {
        Pizza pizza = new Pizza.PizzaBuilder()
            .setDough("Thin crust")
            .setSauce("Tomato")
            .setTopping("Cheese + Pepperoni")
            .build();
        System.out.println(pizza);
    }
}
```

3.3 Паттерн Factory Method (Фабричный метод)

3.3.1 Описание паттерна Factory Method (Фабричный метод)

Это порождающий паттерн проектирования, который предоставляет интерфейс для создания объектов, но позволяет подклассам изменять тип создаваемого объекта. Это полезно, когда нужно делегировать процесс создания объекта на более низкий уровень, обеспечивая при этом возможность изменять конкретный класс создаваемого объекта.

Основные условия для применения паттерна фабричный метод:

1) классу заранее неизвестно, объекты каких классов ему нужно создавать;

2) класс спроектирован так, чтобы объекты, которые он создает, определялись подклассами;

3) класс делегирует свои обязанности одному из нескольких вспомогательных подклассов, и вам нужно локализовать информацию о том, какой класс принимает эти обязанности на себя.

Основные элементы паттерна:

1. Продукт (*Product*) – интерфейс или абстрактный класс создаваемых объектов.

2. Конкретный продукт (*Concrete Product*) – классы, которые реализуют интерфейс продукта.

3. Создатель (*Creator*) – абстрактный класс или интерфейс, который объявляет фабричный метод, возвращающий объект продукта.

4. Конкретный создатель (*Concrete Creator*) – классы, которые переопределяют фабричный метод для создания определённых объектов.

Основные отношения между сущностями в UML-диаграмме классов для данного паттерна, следующие: наследование – базовый класс определяет интерфейс фабричного метода, а подклассы реализуют его; ассоциация – фабричный метод возвращает экземпляры продуктов.

Результаты паттерна:

1. Фабричные методы избавляют проектировщика от необходимости встраивать в код зависящие от приложения классы. Код имеет дело только с интерфейсом класса *Product*, поэтому он может работать с любыми определенными пользователями классами конкретных продуктов.

2. Потенциальный недостаток фабричного метода состоит в том, что клиентам, возможно, придется создавать подкласс класса *Creator* для создания лишь одного объекта *ConcreteProduct*. Порождение подклассов оправданно, если клиенту так или иначе приходится создавать подклассы *Creator*, в противном случае клиенту придется иметь дело с дополнительным уровнем подклассов.

3. Подклассам предоставляются операции-зацепки (*hooks*). Создание объектов внутри класса с помощью фабричного метода всегда оказывается более гибким решением, чем непосредственное создание. Фабричный метод создает в подклассах операции-зацепки (операции, которые выполняются в определенной последовательности и зависят друг от друга) для предоставления расширенной версии объекта.

4. Соединение параллельных иерархий. Параллельные иерархии возникают в случае, когда класс делегирует часть своих обязанностей другому классу, который не является производным от него. Например, графические фигуры, которыми можно манипулировать интерактивно: растягивать, двигать или вращать с помощью мыши. Реализовать такие взаимодействия бывает непросто, часто приходится сохранять и обновлять информацию о текущем состоянии манипуляций. Но это состояние используется только во время самой манипуляции, поэтому помещать его в объект, представляющий фигуру, не следует. К тому же фигуры по-разному ведут себя, когда пользователь манипулирует ими. Например, растягивание отрезка может сводиться к изменению положения концевой точки, а растягивание текста – к изменению междустрочных интервалов.

Недостатки паттерна, это увеличение числа классов, так как для каждой новой реализации требуется создавать отдельный подкласс. Сложность поддержки кода из-за множества подклассов и необходимости их правильного взаимодействия. Может быть избыточным для простых задач, где создание объектов можно обойтись без наследования.

Пример из реальной жизни: заказ такси через мобильное приложение. В зависимости от выбора пользователя, можно заказать разные типы автомобилей: эконом, комфорт, бизнес-класс или минивэн. Каждому типу заказа соответствует свой класс, а конкретный тип машины определяется фабричным методом, исходя из предпочтений пользователя.

3.3.2 Пример на Python

Есть транспортные средства, и каждое из них может двигаться по-разному. Нужно создать фабрику, которая будет создавать разные транспортные средства (машины, самолеты, корабли), но не изменять основной код при добавлении новых видов транспорта.

```
class Transport(ABC):
    @abstractmethod
    def move(self) -> None:
        pass
class Car(Transport):
    def move(self) -> None:
        print("Машина едет по дороге")
class Airplane(Transport):
    def move(self) -> None:
        print("Самолет летит по воздуху")
class Ship(Transport):
```

```
    def move(self) -> None:
        print("Корабль плывет по воде")
class TransportFactory(ABC):
    @abstractmethod
    def create_transport(self) -> Transport:
        pass
    def use_transport(self) -> None:
        transport = self.create_transport()
        transport.move()
class CarFactory(TransportFactory):
    def create_transport(self) -> Transport:
        return Car()
class AirplaneFactory(TransportFactory):
    def create_transport(self) -> Transport:
        return Airplane()
class ShipFactory(TransportFactory):
    def create_transport(self) -> Transport:
        return Ship()
if __name__ == "__main__":
    car_factory = CarFactory()
    car_factory.use_transport()
    airplane_factory = AirplaneFactory()
    airplane_factory.use_transport()
    ship_factory = ShipFactory()
    ship_factory.use_transport()
```

Пояснение:

– *Transport* – это абстрактный класс или интерфейс, который определяет метод *move*. Все конкретные виды транспорта должны его реализовать;

– *Car*, *Airplane*, *Ship* – это конкретные продукты, которые реализуют интерфейс *Transport* и предоставляют свою реализацию метода *move*;

– *TransportFactory* – абстрактный класс, который объявляет фабричный метод *create_transport*. Этот метод должен возвращать объект типа *Transport*. В этом классе также есть метод *use_transport*, который использует продукт (транспортное средство);

– *CarFactory*, *AirplaneFactory*, *ShipFactory* – конкретные создатели, которые реализуют фабричный метод и возвращают конкретные объекты (машину, самолет или корабль).

3.3.3 Пример на Java

Аналогичный пример на Java с использованием паттерна *Factory Method* (Фабричный метод).

```
public interface Transport {
    void move();
}
public class Car implements Transport {
    @Override
    public void move() {
        System.out.println("Машина едет по дороге");
    }
}
public class Airplane implements Transport {
    @Override
    public void move() {
        System.out.println("Самолет летит по воздуху");
```

```
    }
}
public class Ship implements Transport {
    @Override
    public void move() {
        System.out.println("Корабль плывет по воде");
    }
}
public abstract class TransportFactory {
    public abstract Transport createTransport();
    public void useTransport() {
        Transport transport = createTransport();
        transport.move();
    }
}
public class CarFactory extends TransportFactory {
    @Override
    public Transport createTransport() {
        return new Car();
    }
}
public class AirplaneFactory extends TransportFactory {
    @Override
    public Transport createTransport() {
        return new Airplane();
    }
}
public class ShipFactory extends TransportFactory {
    @Override
    public Transport createTransport() {
        return new Ship();
    }}
public class Main {
    public static void main(String[] args) {
        TransportFactory carFactory = new CarFactory();
        carFactory.useTransport();
        TransportFactory airplaneFactory = new AirplaneFactory();
        airplaneFactory.useTransport();
        TransportFactory shipFactory = new ShipFactory();
        shipFactory.useTransport();
    }}
```

3.4 Паттерн Singleton (Одиночка)

3.4.1 Описание паттерна Singleton (Одиночка)

Порождающий шаблон проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет глобальную точку доступа к этому экземпляру. Этот паттерн часто используется, когда необходимо ограничить создание объектов определенным количеством экземпляров (чаще всего – одним) для управления доступом к общему ресурсу или состоянию.

Основные условия для применения паттерна одиночка:

1) должен существовать ровно один экземпляр некоторого класса, к которому может обратиться любой клиент через известную точку доступа;

2) единственный экземпляр должен расширяться путем порождения подклассов, а клиенты должны иметь возможность работать с расширенным экземпляром без модификации своего кода.

Основные элементы паттерна:

Singleton (Одиночка) – основной класс, который управляет созданием и доступом к своему единственному экземпляру.

Основные отношения между сущностями в UML-диаграмме классов для данного паттерна, следующие: ассоциация – клиенты зависят от экземпляра одиночки. Класс одиночки предоставляет глобальный доступ через статический метод или свойство.

Результаты паттерна:

1. Контролируемый доступ к единственному экземпляру. Поскольку класс *Singleton* инкапсулирует свой единственный экземпляр, он полностью контролирует то, как и когда клиенты получают доступ к нему.

2. Сокращение пространства имен. Паттерн Одиночка – шаг вперед по сравнению с глобальными переменными. Он позволяет избежать засорения пространства имен глобальными переменными, в которых хранятся уникальные экземпляры.

3. Возможность уточнения операций и представления. От класса *Singleton* можно порождать подклассы, а приложение легко настраивается экземпляром расширенного класса. Приложение можно настроить экземпляром нужного класса во время выполнения.

4. Возможность использования переменного числа экземпляров. Паттерн позволяет легко изменить решение и разрешить появление более одного экземпляра класса *Singleton*. Более того, тот же подход может использоваться для управления числом экземпляров, используемых в приложении. Изменить нужно будет лишь операцию, дающую доступ к экземпляру класса *Singleton*.

5. Большая гибкость, чем у операций класса. Другой способ реализации функциональности одиночки – использование операций класса, то есть статических функций класса и методов класса.

Недостатки паттерна, это сложное тестирование, так как контролировать состояние одиночки в тестах сложно. Нарушает принцип единственной ответственности (SRP), так как класс отвечает и за создание экземпляра, и за свою основную логику. Может привести к созданию скрытых зависимостей между классами.

Пример из реальной жизни: человек хочет получить паспорт. В любой стране этот человек обращается в соответствующее государственное учреждение, которое принадлежит к единой государственной системе. Он не может создать новое учреждение для получения паспорта, ему приходится пользоваться существующим, которое управляет всеми соответствующими процессами. Все граждане страны обращаются к одному и тому же учреждению, тем самым показывая идею единственного глобального доступа к центральному органу власти.

3.4.2 Пример на Python

Вместо того, чтобы каждый раз при вызове конструктора создавать новый объект, необходимо делать так, чтобы создавался только один объект, а все последующие запросы возвращали ссылку на уже существующий объект. Для того, чтобы другие части программы не могли создать объект через new, конструктор класса делается приватным. Предоставляется метод, который либо создаёт объект (если его ещё нет), либо возвращает уже существующий объект.

```
class Singleton:
    _instance = None
    def __new__(cls):
        if cls._instance is None:
            cls._instance = super(Singleton, cls).__new__(cls)
            cls._instance.init_data()
        return cls._instance
    def init_data(self):
        self.value = 42
if __name__ == "__main__":
    s1 = Singleton()
    s2 = Singleton()
    print(f"Одинаковый объект? {'Да' if s1 is s2 else 'Нет'}")
    print(f"s1.value = {s1.value}")
    print(f"s2.value = {s2.value}")
    s1.value = 100
    print(f"Значение в s2 после изменения s1: {s2.value}")
```

Пояснение:

– приватная переменная *_instance* хранит ссылку на единственный экземпляр класса. Она инициализируется значением *None*, что означает, что экземпляр ещё не создан;

– метод *__new__* отвечает за создание нового экземпляра класса. Если экземпляр уже существует, возвращается старый экземпляр, и создание нового объекта не происходит;

– метод *init_data* вспомогательный метод для инициализации каких-либо данных, которые нужны экземпляру. Он вызывается только один раз при создании объекта.

3.4.3 Пример на Java

В Java паттерн Одиночка реализуется по тем же принципам, что и в Python, но с учётом особенностей языка. В Java нужно использовать приватный конструктор, а также статический метод для глобальной точки доступа к экземпляру.

```
public class Singleton {
    private static Singleton instance;
    private Singleton() {
        this.value = 42;
    }
    private int value;
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
    public int getValue() {
        return value;
    }
    public void setValue(int value) {
        this.value = value;
    }
    public static void main(String[] args) {
        Singleton s1 = Singleton.getInstance();
        Singleton s2 = Singleton.getInstance();
        System.out.println("Одинаковый объект? " + (s1 == s2));
        System.out.println("s1.value = " + s1.getValue());
        System.out.println("s2.value = " + s2.getValue());
```

```
        s1.setValue(100);
        System.out.println("Значение в s2 после изменения s1: " +
s2.getValue());
    }
}
```

3.5 Паттерн Prototype (Прототип)

3.5.1 Описание паттерна Prototype (Прототип)

Используется для создания объектов на основе уже существующих экземпляров путем их клонирования. Это позволяет создавать новые объекты без необходимости привязываться к конкретным классам. Часто применяется, когда создание объекта с помощью конструктора является сложным или затратным процессом.

Использование паттерна прототип, когда система не должна зависеть от того, как в ней создаются, компонуются и представляются продукты, кроме того:

– классы для создания экземпляров определяются во время выполнения, например с помощью динамической загрузки,

или

– для того чтобы избежать построения иерархий классов или фабрик, параллельных иерархии классов продуктов,

или

– экземпляры класса могут находиться в одном из не очень большого числа различных состояний. Может быть удобнее установить соответствующее число прототипов и клонировать их, а не создавать экземпляр каждый раз вручную в подходящем состоянии.

Основные элементы паттерна:

1. *Prototype* (Прототип) – интерфейс или абстрактный класс, который определяет метод для клонирования объектов (например, clone()).

2. *ConcretePrototype* (Конкретный прототип) – классы, реализующие интерфейс *Prototype*, которые способны создавать копии самих себя. Эти классы реализуют метод клонирования.

3. *Client* (Клиент) – объект, который использует прототип для создания новых экземпляров. Клиент не знает, какой именно тип объекта он клонирует – он просто работает с объектами через их интерфейс.

Основные отношения между сущностями в UML-диаграмме классов для данного паттерна, следующие: реализация – конкретные прототипы реализуют метод клонирования; ассоциация – клиент обращается к прототипу для создания нового объекта.

Результаты паттерна:

У прототипа те же самые результаты, что у абстрактной фабрики и строителя: он скрывает от клиента конкретные классы продуктов, уменьшая тем самым число известных клиенту имен. Кроме того, все эти паттерны позволяют клиентам работать с классами, специфичными для приложения, без модификаций.

Ниже перечислены дополнительные преимущества паттерна прототип:

1. Добавление и удаление продуктов во время выполнения. Прототип позволяет включать новый конкретный класс продуктов в систему, просто зарегистрировав новый экземпляр-прототип на стороне клиента. Это несколько более гибкое решение по сравнению с тем, что удастся

сделать с помощью других порождающих паттернов, ибо клиент может устанавливать и удалять прототипы во время выполнения.

2. Определение новых объектов путем изменения значений. Динамичные системы позволяют определять поведение посредством композиции объектов – например, путем задания значений переменных объекта, – а не посредством определения новых классов. Фактически определяется новые виды объектов, создавая экземпляры уже существующих классов и регистрируя их экземпляры как прототипы клиентских объектов. Клиент может изменить поведение, делегируя свои обязанности прототипу.

3. Такой дизайн позволяет пользователям определять новые «классы» без программирования. Фактически клонирование объекта аналогично созданию экземпляра. Паттерн Прототип может резко уменьшить число необходимых системе классов.

4. Определение новых объектов путем изменения структуры. Многие приложения строят объекты из крупных и мелких составляющих. Например, редакторы для проектирования печатных плат создают электрические схемы из подсхем. Такие приложения часто позволяют создавать экземпляры сложных, определенных пользователем структур, для многократного использования некоторой подсхемы.

5. Уменьшение числа подклассов. Паттерн Фабричный метод часто порождает иерархию классов *Creator*, параллельную иерархии классов продуктов. Прототип позволяет клонировать прототип, а не запрашивать фабричный метод создать новый объект. Поэтому иерархия класса *Creator* становится вообще ненужной.

6. Динамическая настройка конфигурации приложения классами. Некоторые среды позволяют динамически загружать классы в приложение во время его выполнения.

Основной недостаток паттерна прототип заключается в том, что каждый подкласс класса *Prototype* должен реализовывать операцию *Clone*, а это далеко не всегда просто. Например, сложно добавить операцию *Clone*, если рассматриваемые классы уже существуют. Проблемы возникают и в случае, если во внутреннем представлении объекта присутствуют другие объекты, не поддерживающие копирования, или наличествуют циклические ссылки.

Пример из реальной жизни: изготовление дубликатов ключей. Есть оригинальный ключ, и нужно сделать несколько его копий. Мастер не создает каждый ключ с нуля, он берет оригинал (прототип) и просто копирует его. Этот процесс быстрее и дешевле, чем изготавливать каждый ключ с чистого листа.

3.5.2 Пример на Python

Имеется приложение для рисования, где пользователи могут создавать различные формы (круги, квадраты и т.д.). Каждый раз, когда пользователь хочет создать новую форму, он должен выбирать ее тип и задавать параметры. Вместо того чтобы создавать новый объект с нуля, можно использовать прототипы для клонирования уже существующих форм.

Создание базового класса *Shape*, который будет представлять форму, и два подкласса: *Circle* и *Square*.

```
class Shape:
    def clone(self):
        return copy.deepcopy(self)
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius
```

```
    def __str__(self):
        return f"Circle with radius: {self.radius}"
class Square(Shape):
    def __init__(self, side_length):
        self.side_length = side_length
    def __str__(self):
        return f"Square with side length: {self.side_length}"
def main():
    original_circle = Circle(radius=5)
    original_square = Square(side_length=4)
    cloned_circle = original_circle.clone()
    cloned_square = original_square.clone()
    cloned_circle.radius = 10
    cloned_square.side_length = 8
    print("Original Circle:", original_circle)
    print("Cloned Circle:", cloned_circle)
    print("Original Square:", original_square)
    print("Cloned Square:", cloned_square)
if __name__ == "__main__":
    main()
```

Пояснение:

– класс *Shape* базовый класс для всех форм. Метод *clone* создает и возвращает глубокую копию объекта, что позволяет создавать новые экземпляры без изменения оригинала;

– класс *Circle* наследует от *Shape*. Конструктор принимает радиус и сохраняет его как атрибут. Метод *__str__* определяет строковое представление объекта, что позволяет удобно выводить информацию о нем;

– класс *Square* аналогичен классу *Circle*, но представляет квадрат. Имеет свойство *side_length* и переопределяет метод *__str__* для представления информации о квадрате;

– функция *main* создает экземпляры *Circle* и *Square*, задавая им начальные параметры. Клонирует эти объекты с помощью метода *clone*. Клонированные объекты независимы от оригиналов. Изменяет параметры клонированных объектов, чтобы продемонстрировать, что изменения не влияют на оригиналы. Печатает информацию об оригинальных и клонированных объектах, показывая их состояние.

3.5.3 Пример на Java

Аналогичный пример использования паттерна Прототип на языке Java.

```
interface Shape {
    Shape clone();
}
class Circle implements Shape {
    private int radius;
    public Circle(int radius) {
        this.radius = radius;
    }
    @Override
    public Shape clone() {
        return new Circle(this.radius);
    }
    @Override
    public String toString() {
        return "Circle with radius: " + radius;
```

```
    }
}
class Square implements Shape {
    private int sideLength;
    public Square(int sideLength) {
        this.sideLength = sideLength;
    }
    @Override
    public Shape clone() {
        return new Square(this.sideLength);
    }
    @Override
    public String toString() {
        return "Square with side length: " + sideLength;
    }
}
public class PrototypeExample {
    public static void main(String[] args) {
        Shape originalCircle = new Circle(5);
        Shape originalSquare = new Square(4);
        Shape clonedCircle = originalCircle.clone();
        Shape clonedSquare = originalSquare.clone();
        clonedCircle = new Circle(10);
        clonedSquare = new Square(8);
        System.out.println("Original Circle: " + originalCircle);
        System.out.println("Cloned Circle: " + clonedCircle);
        System.out.println("Original Square: " + originalSquare);
        System.out.println("Cloned Square: " + clonedSquare);
    }
}
```

3.6 Контрольные вопросы для самоподготовки

Тема 1. Проектирование и реализация паттерна «Абстрактная фабрика» (Abstract Factory)

Цель самоподготовки: закрепление навыков проектирования систем, требующих создания семейств взаимосвязанных объектов, с обеспечением независимости клиентского кода от конкретных классов продуктов.

1. Проектирование иерархии продуктов. Каким образом следует проводить декомпозицию предметной области для выделения семейств взаимосвязанных продуктов? Опишите процесс проектирования базовых интерфейсов (абстрактных классов) для различных типов продуктов внутри одного семейства.

2. Формирование контракта фабрики. Какие требования предъявляются к интерфейсу *AbstractFactory*? Обоснуйте состав методов, необходимых для декларации создания каждого типа абстрактного продукта, и объясните, почему возвращаемые типы должны быть абстрактными.

3. Реализация конкретных семейств. Опишите механизм взаимодействия конкретной фабрики с соответствующими ей классами продуктов. Как гарантировать на уровне архитектуры, что конкретная фабрика будет инстанцировать объекты только одного, строго согласованного семейства?

4. Изоляция клиентского кода. Спроектируйте структуру класса клиента, взаимодействующего с фабрикой. Каким образом реализуется принцип инверсии зависимостей, чтобы клиент не использовал прямое создание (оператор *new*) конкретных классов продуктов?

5. Архитектурная визуализация и анализ. Какие ключевые зависимости должны быть отображены на UML-диаграмме классов для данного паттерна? Объясните, в чем заключается разница в отображении отношений между клиентом и абстракциями по сравнению с отношениями между конкретными фабриками и продуктами (реализация vs. ассоциация).

Тема 2. Проектирование и реализация паттерна «Строитель» (Builder)

Цель самоподготовки: освоение концепции пошагового конструирования сложных объектов и разделения алгоритма сборки от его технической реализации для получения различных представлений продукта.

1. Анализ структуры сложного продукта. В каких случаях прямое использование конструкторов приводит к антипаттерну «телескопический конструктор»? Спроектируйте структуру объекта, содержащего большое количество опциональных полей, и обоснуйте необходимость применения паттерна «Строитель» для управления его состоянием.

2. Проектирование интерфейса *Builder*. Опишите принципы формирования набора независимых методов в базовом интерфейсе *Builder*. Каким образом проектируется «текучий интерфейс» (*fluent interface*) и какие преимущества он дает при конфигурировании продукта?

3. Специфика конкретных строителей. Как реализовать инкапсуляцию логики сборки в классах *ConcreteBuilder* для различных конфигураций одного и того же продукта? Опишите назначение и механизм работы терминального метода (например, *getResult()*).

4. Роль директора в процессе конструирования. Спроектируйте класс *Director*. Каким образом он должен инкапсулировать типовые алгоритмы (строгие последовательности шагов) сборки? Объясните, почему директор должен работать с абстрактным интерфейсом строителя, а не с его конкретными реализациями.

5. Взаимодействие компонентов в клиентском коде. Опишите жизненный цикл объектов в процессе создания продукта: от инстанцирования конкретного строителя до получения итогового результата. Почему клиент должен запрашивать готовый продукт у строителя, а не у директора?

6. Анализ объектных связей на UML-диаграмме. Обоснуйте выбор отношений между участниками паттерна на диаграмме классов. В чем проявляется отношение агрегации между директором и базовым строителем, и как графически отображается зависимость конкретного строителя от создаваемого им продукта?

Тема 3. Проектирование и реализация паттерна «Одиночка» (Singleton)

Цель самоподготовки: изучение архитектурного механизма строгого ограничения инстанцирования класса единственным объектом и организации глобальной унифицированной точки доступа к нему.

1. Механизмы ограничения инстанцирования. Назовите основные причины, по которым в системе может потребоваться ограничение количества экземпляров класса до одного (приведите примеры из области системного программирования). Каким образом модификаторы видимости конструктора позволяют технически запретить создание объекта через оператор *new* во внешнем коде?

2. Инкапсуляция статического состояния. Обоснуйте необходимость использования статического поля ссылочного типа внутри класса *Singleton*. Какова роль этого поля в обеспечении жизненного цикла единственного экземпляра?

3. Логика глобальной точки доступа. Спроектируйте алгоритм работы статического метода *getInstance()*. В чем заключаются преимущества реализации «ленивой инициализации» (*lazy initialization*) и в какой момент жизненного цикла программы должен быть создан объект?

4. Проектирование в условиях многопоточности. Проанализируйте риск возникновения «состояния гонки» (*race condition*) при одновременном вызове метода доступа несколькими потоками. Опишите механизм «блокировки с двойной проверкой» (*double-checked locking*): как он обеспечивает потокобезопасность и почему однократной проверки внутри блока синхронизации недостаточно?

5. Специфика визуализации статических членов в UML. Каким образом на диаграмме классов обозначаются статические поля и методы согласно нотации UML? Опишите, как графически выделить приватный конструктор и подчеркнуть уникальную природу паттерна *Singleton*.

Тема 4. Проектирование и реализация паттерна «Фабричный метод» (Factory Method)

Цель самоподготовки: изучение механизма делегирования процесса создания объектов подклассам для обеспечения независимости базовой логики системы от конкретных реализаций продуктов.

1. Проектирование общей абстракции продукта. Каким образом выделение интерфейса или абстрактного класса *Product* способствует ослаблению связанности (*loose coupling*) в системе? Опишите критерии выбора методов, которые должны быть объявлены в этой базе.

2. Реализация вариативности продуктов. Опишите процесс создания конкретных классов *ConcreteProduct*. Как инкапсуляция специфической логики в подклассах позволяет основной системе работать с объектами единообразно?

3. Роль абстрактного создателя (*Creator*). Объясните архитектурный смысл объявления абстрактного фабричного метода внутри класса *Creator*. Каким образом основная бизнес-логика создателя может использовать продукт, если его конкретный тип неизвестен на этапе компиляции?

4. Специализация конкретных создателей. Опишите механизм переопределения фабричного метода в подклассах. Каким образом достигается связь между конкретным типом создателя и соответствующим ему типом продукта?

5. Соблюдение принципа открытости/закрытости (OCP). Докажите, что применение «Фабричного метода» позволяет расширять систему новыми типами продуктов без модификации существующего кода создателей и клиентов. Какие действия необходимо предпринять программисту для добавления нового продукта в такую архитектуру?

6. Анализ параллельных иерархий в UML. Как на UML-диаграмме классов визуально отображаются параллельные иерархии создателей и продуктов? Опишите связи зависимости между ними и объясните, почему этот паттерн часто называют «виртуальным конструктором».

Тема 5. Проектирование и реализация паттерна «Прототип» (Prototype)

Цель самоподготовки: освоение механизмов динамического создания объектов путем клонирования существующих эталонных экземпляров для минимизации зависимости клиентского кода от конкретных классов.

1. Интерфейс клонирования как абстракция. В чем заключается преимущество использования единого контракта *Prototype* (с методом *clone()*) перед обычным инстанцированием через конструкторы? Опишите, как этот интерфейс изолирует клиентский код от деталей реализации конкретных классов.

2. Инкапсуляция состояния в прототипах. Спроектируйте структуру нескольких конкретных классов-прототипов. Какие внутренние данные и бизнес-логику целесообразно инкапсулировать в таких классах, чтобы их клонирование было эффективнее создания «с нуля»?

3. Реализация механизмов глубокого копирования (*Deep Copy*). Проанализируйте разницу между поверхностным (*Shallow*) и глубоким (*Deep*) копированием. Опишите алгоритм рекурсивного дублирования вложенных сущностей (коллекций, массивов, ссылок). Как гарантировать отсутствие побочных эффектов при изменении состояния клона относительно оригинала?

4. Организация реестра прототипов (*Prototype Registry*). Спроектируйте класс-менеджер (реестр) для централизованного управления оригиналами. Какую структуру данных (например, «ключ-значение») эффективнее использовать для хранения эталонов? Опишите протокол взаимодействия клиента с реестром при запросе объекта по идентификатору.

5. Архитектурное моделирование подсистемы. Обоснуйте выбор связей на UML-диаграмме для паттерна «Прототип». Как графически отобразить отношение агрегации между реестром и интерфейсом *Prototype*? Объясните, почему реестр должен зависеть от интерфейса, а не от конкретных реализаций прототипов.

4 СТРУКТУРНЫЕ ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

Это один из трех основных типов паттернов проектирования, цель которых заключается в упрощении создания и управления сложными структурами объектов и классов. Они помогают в организации и компоновке объектов и классов таким образом, чтобы упростить взаимодействие между ними, улучшить поддержку кода и повысить гибкость архитектуры.

Структурные паттерны уровня класса используют наследование для составления композиций из интерфейсов и реализаций. Простой пример – использование множественного наследования для объединения нескольких классов в один. В результате получается класс, обладающий свойствами всех своих родителей. Этот паттерн особенно полезен для организации совместной работы нескольких независимо разработанных библиотек.

Структурные паттерны обеспечивают способы комбинирования объектов и классов, чтобы упростить сложные системы. Это делает архитектуру более понятной и легкой для сопровождения. Паттерны позволяют скрыть сложные реализации и предоставить клиентам удобные и простые интерфейсы для работы с объектами или структурами. Это повышает уровень абстракции и отделяет детали реализации от использования. Благодаря структурным паттернам можно изменить или расширить поведение системы, не изменяя существующий код. Это способствует повторному использованию компонентов и упрощает внесение изменений. Некоторые паттерны помогают организовать совместное использование объектов, чтобы избежать излишнего потребления памяти или других ресурсов.

Преимущества структурных паттернов:

– структуры становятся более модульными, что упрощает их поддержку и расширение;

– использование структурных паттернов делает код более понятным и структурированным;

– одна и та же структура может быть использована в разных контекстах или приложениях, снижая дублирование кода.

Таким образом, структурные паттерны проектирования помогают строить гибкую и легко модифицируемую архитектуру приложений, обеспечивая хорошую организацию и снижение сложности взаимодействия между объектами и классами.

4.1 Паттерн Adapter (Адаптер)

4.1.1 Описание паттерна Adapter (Адаптер)

Структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе. Он используется для того, чтобы согласовать два класса, у которых интерфейсы не совпадают, но они должны совместно выполнять задачи. Т.е. предоставить объекту интерфейс, совместимый с другим объектом, с которым нужно взаимодействовать, без изменения их исходного кода.

Основные условия для применения паттерна адаптер:

1) если нужно использовать существующий класс, но его интерфейс не соответствует вашим потребностям;

2) требуется создать повторно используемый класс, который должен взаимодействовать с заранее неизвестными или не связанными с ним классами, имеющими несовместимые интерфейсы;

3) (только для адаптера объектов!) Нужно использовать несколько существующих подклассов, но непрактично адаптировать их интерфейсы путем порождения новых подклассов от каждого. В этом случае адаптер объектов может приспосабливать интерфейс их общего родительского класса.

Основные элементы паттерна:

1. Цель (*Target*) – интерфейс, который ожидает клиент.

2. Адаптируемый (*Adaptee*) – класс с несовместимым интерфейсом, который нужно адаптировать.

3. Адаптер (*Adapter*) – класс, который реализует интерфейс цели и переводит вызовы клиента в формат, понятный адаптируемому объекту.

4. Клиент (*Client*) – объект, который взаимодействует с адаптером, пользуясь интерфейсом цели.

Основные отношения между сущностями в UML-диаграмме классов для данного паттерна, следующие: ассоциация – клиент взаимодействует с адаптером; реализация – адаптер реализует интерфейс, ожидаемый клиентом; композиция – адаптер содержит ссылку на адаптируемый класс.

Результаты паттерна:

Адаптеры объектов и классов обладают разными достоинствами и недостатками.

1. Адаптер класса:

– адаптирует *Adaptee* к *Target*, перепоручая действия конкретному классу *Adaptee*. Поэтому данный паттерн не будет работать, если требуется одновременно адаптировать класс и его подклассы;

– позволяет адаптеру *Adapter* заместить некоторые операции адаптируемого класса *Adaptee*, так как *Adapter* есть не что иное, как подкласс *Adaptee*;

– вводит только один новый объект. Чтобы добраться до адаптируемого класса, не нужно никакого дополнительного обращения по указателю.

2. Адаптер объектов:

– позволяет одному адаптеру *Adapter* работать со многим адаптируемыми объектами *Adaptee*, то есть с самим *Adaptee* и его подклассами (если таковые имеются). Адаптер может добавить новую функциональность сразу всем адаптируемым объектам;

– затрудняет замещение операций класса *Adaptee*. Для этого потребуется породить от *Adaptee* подкласс и заставить *Adapter* ссылаться на этот подкласс, а не на сам *Adaptee*.

Недостатки паттерна, это увеличение сложности системы, особенно если используются каскадные адаптеры (один адаптер адаптирует другой). Может привести к снижению производительности, если адаптация требует преобразований данных. Адаптеры усложняют структуру кода и могут скрыть изъяны дизайна.

Пример из реальной жизни: имеется европейская вилка и американская розетка. Для того чтобы подключить вилку к розетке, нужен переходник (адаптер). Этот адаптер позволяет использовать европейскую вилку в американской розетке, не изменяя ни вилку, ни розетку.

Есть библиотека для работы с файлами в старом формате, но проект использует новый формат файлов. Вместо того чтобы переписывать всю библиотеку или проект, можно создать адаптер, который будет преобразовывать запросы нового формата в старый и наоборот.

4.1.2 Пример на Python

Имеется система для работы с данными клиентов, которая возвращает данные клиента. Однако есть старая библиотека, который возвращает данные в другом формате. Чтобы соединить их, нужно использовать *Adapter*.

```
class OldCustomerAPI:
    def fetch_customer_info(self):
        return {
            "name": "Sergey M",
            "age": 28,
            "address": "Samara"
        }
class Client:
    def __init__(self, adapter):
        self.adapter = adapter
    def get_client_data(self):
        return self.adapter.get_data()
class CustomerAdapter:
    def __init__(self, old_api):
        self.old_api = old_api
    def get_data(self):
        old_data = self.old_api.fetch_customer_info()
        new_data = {
            "full_name": old_data["name"],
            "age": old_data["age"],
            "location": old_data["address"]
        }
        return new_data
old_api = OldCustomerAPI()
adapter = CustomerAdapter(old_api)
client = Client(adapter)
client_data = client.get_client_data()
print(client_data)
```

Пояснение:

– старый класс *OldCustomerAPI*, где метод *fetch_customer_info* возвращает данные клиента в формате, который не подходит для нашего нового интерфейса;

– новый интерфейс *Client* ожидает, что данные клиента будут получены через метод *get_client_data*. Для этого ему передается адаптер, у которого есть метод get_data, возвращающий данные в нужном формате;

– в адаптере *CustomerAdapter* конструктор *__init__* принимает экземпляр *OldCustomerAPI* и сохраняет его для использования. Метод *get_data* получает данные из старого API, используя метод *fetch_customer_info* и преобразует их в формат, который ожидает *Client* (меняет ключи и формат данных).

4.1.3 Пример на Java

Аналогичный пример использования паттерна Адаптер на языке Java.

```
class OldCustomerAPI {
    public Map<String, String> fetchCustomerInfo() {
        Map<String, String> oldData = new HashMap<>();
        oldData.put("name", "Sergey M");
        oldData.put("age", "28");
```

```
        oldData.put("address", "Samara"
        return oldData;
    }
}
interface ClientData {
    Map<String, String> getClientData();
}
class CustomerAdapter implements ClientData {
    private OldCustomerAPI oldApi;
    public CustomerAdapter(OldCustomerAPI oldApi) {
        this.oldApi = oldApi;
    }
    @Override
    public Map<String, String> getClientData() {
        Map<String, String> oldData = oldApi.fetchCustomerInfo();
        Map<String, String> newData = new HashMap<>();
        newData.put("full_name", oldData.get("name"));
        newData.put("age", oldData.get("age"));
        newData.put("location", oldData.get("address"));
        return newData;
    }
}
class Client {
    private ClientData data;
    public Client(ClientData data) {
        this.data = data;
    }
    public void printClientData() {
        Map<String, String> clientData = data.getClientData();
        System.out.println("Client Data: " + clientData);
    }
}
public class AdapterExample {
    public static void main(String[] args) {
        OldCustomerAPI oldApi = new OldCustomerAPI();
        ClientData adapter = new CustomerAdapter(oldApi);
        Client client = new Client(adapter);
        client.printClientData();
    }
}
```

4.2 Паттерн Bridge (Мост)

4.2.1 Описание паттерна Bridge (Мост)

Паттерн, который разделяет абстракцию и реализацию, позволяя им изменяться независимо друг от друга. Т.е. выделяет абстракцию и реализацию в отдельные иерархии, соединяя их через мост (интерфейс или ссылку на объект реализации). Это позволяет изменять или расширять их независимо друг от друга. Он используется, когда необходимо обеспечить гибкость в расширении функциональности классов, избежать увеличения количества подклассов и сократить связанность между ними. Также применяется в случаях, когда требуется поддерживать несколько возможных реализаций одной абстракции.

Основные условия для применения паттерна мост:

1. Требуется избежать постоянной привязки абстракции к реализации. Так, например, бывает, когда реализация должна выбираться во время выполнения программы.

2. И абстракции, и реализации должны расширяться новыми подклассами. В таком случае паттерн Мост позволяет комбинировать разные абстракции и реализации и изменять их независимо.

3. Изменения в реализации абстракции не должны отражаться на клиентах, то есть клиентский код не должен перекомпилироваться.

4. (только для C++!) Требуется полностью скрыть от клиентов реализацию абстракции. В C++ представление класса видимо через его интерфейс.

5. Число классов стремительно разрастается. Это признак того, что иерархию следует разделить на две части.

6. Реализация должна совместно использоваться несколькими объектами (например, на базе подсчета ссылок), и этот факт должен быть скрыт от клиента.

Основные элементы паттерна:

1. Абстракция (*Abstraction*) – определяет высокоуровневый интерфейс и хранит ссылку на объект реализации.

2. Уточненная абстракция (*RefinedAbstraction*) – конкретные реализации интерфейса абстракции, которые используют объект реализации.

3. Интерфейс реализации (*Implementor*) – определяет интерфейс для всех возможных реализаций, которые могут быть подключены к абстракции.

4. Конкретная реализация (*ConcreteImplementor*) – классы, реализующие интерфейс реализации и предоставляющие конкретные способы работы.

Основные отношения между сущностями в UML-диаграмме классов для данного паттерна, следующие: ассоциация – абстракция содержит ссылку на реализацию (*Implementor*); реализация – конкретные реализации (*ConcreteImplementor*) реализуют интерфейс (*Implementor*).

Результаты паттерна:

Отделение реализации от интерфейса. Реализация больше не имеет постоянной привязки к интерфейсу. Реализация абстракции может настраиваться во время выполнения. Объект может даже динамически изменять свою реализацию.

Разделение классов *Abstraction* и *Implementor* устраняет также зависимости от реализации, устанавливаемые на этапе компиляции. Чтобы изменить класс реализации, не обязательно перекомпилировать класс *Abstraction* и его клиентов. Это свойство особенно важно, если необходимо обеспечить двоичную совместимость между разными версиями библиотеки классов.

Кроме того, такое разделение облегчает разбиение системы на слои и тем самым позволяет улучшить ее структуру. Высокоуровневые части системы должны знать только о классах *Abstraction* и *Implementor*.

– повышение степени расширяемости. Иерархии классов *Abstraction* и *Implementor* могут расширяться независимо;

– сокрытие деталей реализации от клиентов. Клиентов можно изолировать от таких подробностей реализации, как совместное использование объектов класса *Implementor* и сопутствующего механизма подсчета ссылок.

Недостатки паттерна, это увеличение сложности кода из-за введения дополнительных классов и уровней абстракции. Может быть избыточным для простой системы с небольшой иерархией классов. Сложность при изменении интерфейса как реализации, так и абстракции.

Пример из реальной жизни: есть универсальный пульт дистанционного управления, который может управлять разными устройствами: телевизором, аудиосистемой, кондиционером и т.д. Пульт не знает заранее, с каким именно устройством ему придется работать, но поддерживает базовые команды (включение, выключение, увеличение и уменьшение громкости и т.п.), которые должны работать с любым подключенным устройством. Пульт не реализует конкретные функции для каждого устройства напрямую, а вместо этого через интерфейс отправляет команды устройству, а устройство само решает, как их исполнять.

4.2.2 Пример на Python

Разработка формы оповещения в различных приложениях. Нужно отправлять оповещения разными способами, например, по SMS и Email. Также можно отправлять оповещения разного уровня, например, информационные или критические.

```
class NotificationSender(ABC):
    @abstractmethod
    def send(self, message: str):
        pass
class SmsNotificationSender(NotificationSender):
    def send(self, message: str):
        print(f"Отправка SMS: {message}")
class EmailNotificationSender(NotificationSender):
    def send(self, message: str):
        print(f"Отправка Email: {message}")
class Alert(ABC):
    def __init__(self, sender: NotificationSender):
        self.sender = sender
    @abstractmethod
    def send_alert(self, message: str):
        pass
class InfoAlert(Alert):
    def send_alert(self, message: str):
        self.sender.send(f"Информационное оповещение: {message}")
class CriticalAlert(Alert):
    def send_alert(self, message: str):
        self.sender.send(f"Критическое оповещение: {message}")
sms_sender = SmsNotificationSender()
email_sender = EmailNotificationSender()
info_alert_sms = InfoAlert(sms_sender)
info_alert_sms.send_alert("Все системы работают нормально")
critical_alert_email = CriticalAlert(email_sender)
critical_alert_email.send_alert("Система перегружена!")
```

Пояснение:

– определение интерфейса *NotificationSender* с методом *send*, который принимает сообщение и отправляет его определенным способом (SMS или Email). Все отправители будут следовать этому интерфейсу, что делает возможным замену одного отправителя на другой без изменения основной логики;

– *SmsNotificationSender* и *EmailNotificationSender* являются конкретными реализациями *NotificationSender*, каждая из которых отправляет сообщение по-своему. Например, *SmsNotificationSender* отправляет сообщение как SMS, а *EmailNotificationSender* отправляет его по электронной почте;

– *Alert* – это абстрактный класс для оповещений. Он имеет ссылку на объект *NotificationSender*, который передается через конструктор. Благодаря этому любой класс, который наследует *Alert*, сможет отправлять оповещения, используя любой из доступных способов отправки (SMS, Email), переданных при создании;

– *InfoAlert* и *CriticalAlert* – это конкретные реализации *Alert*. Они используют свой метод *send_alert*, чтобы отправить сообщение с помощью предоставленного *NotificationSender*. *InfoAlert* может использовать SMS или Email для отправки сообщения, и то же самое касается *CriticalAlert*;

– создаются отправители *sms_sender* и *email_sender*, которые реализуют различные способы отправки сообщений. Затем создаются объекты *InfoAlert* и *CriticalAlert*, передавая в них нужного отправителя. Это позволяет выбрать, каким способом будет отправлено конкретное сообщение.

4.2.3 Пример на Java

В этом примере также будут оповещения (*Alert*) и разные способы отправки оповещений (SMS и Email), аналогичные примеру на Python.

```
interface NotificationSender {
    void send(String message);
}
class SmsNotificationSender implements NotificationSender {
    @Override
    public void send(String message) {
        System.out.println("Отправка SMS: " + message);
    }
}
class EmailNotificationSender implements NotificationSender {
    @Override
    public void send(String message) {
        System.out.println("Отправка Email: " + message);
    }
}
abstract class Alert {
    protected NotificationSender sender;
    public Alert(NotificationSender sender) {
        this.sender = sender;
    }
    public abstract void sendAlert(String message);
}
class InfoAlert extends Alert {
    public InfoAlert(NotificationSender sender) {
        super(sender);
    }
    @Override
    public void sendAlert(String message) {
        sender.send("Информационное оповещение: " + message);
    }
}
class CriticalAlert extends Alert {
    public CriticalAlert(NotificationSender sender) {
        super(sender);
    }
    @Override
```

```
    public void sendAlert(String message) {
        sender.send("Критическое оповещение: " + message);
    }
}
public class Main {
    public static void main(String[] args) {
        NotificationSender smsSender = new SmsNotificationSender();
        NotificationSender emailSender = new EmailNotificationSender();
        Alert infoAlertSms = new InfoAlert(smsSender);
        infoAlertSms.sendAlert("Все системы работают нормально");
        Alert criticalAlertEmail = new CriticalAlert(emailSender);
        criticalAlertEmail.sendAlert("Система перегружена!");
    }
}
```

4.3 Паттерн Composite (Компоновщик)

4.3.1 Описание паттерна Composite (Компоновщик)

Паттерн Компоновщик относится к структурным паттернам проектирования и позволяет организовать объекты в виде древовидной структуры для представления иерархий типа «часть-целое», в которой объекты организованы так, что одни объекты являются составными элементами других. В такой иерархии элементы объединены по принципу включения одних объектов (частей) в состав других объектов (целых), образуя систему, где более крупные и сложные объекты состоят из более мелких и простых.

Этот паттерн позволяет пользователям одинаково обращаться как с отдельными объектами, так и с группами объектов.

Компоновщик помогает структурировать объекты в виде дерева, где каждое звено может быть либо отдельным элементом (листом), либо составным элементом (узлом). Каждый узел может содержать как листья, так и другие узлы.

Основные условия для применения паттерна компоновщик:

1) требуется представить иерархию объектов вида «часть – целое»;

2) клиенты должны по единым правилам работать с составными и индивидуальными объектами.

Основные элементы паттерна:

1. *Component* (Компонент) – общий интерфейс или абстрактный класс, который определяет методы для работы как с отдельными объектами, так и с их составами. Например, метод execute() для выполнения команды.

2. *Leaf* (Лист) – конечный элемент, который не содержит вложенных объектов. Реализует методы, объявленные в Component.

3. *Composite* (Контейнер) – компонент, который содержит другие компоненты (как листья, так и другие контейнеры). Обычно реализует методы для добавления и удаления компонентов.

4. *Client* (Клиент) – манипулирует объектами композиции через интерфейс *Component*.

Основные отношения между сущностями в UML-диаграмме классов для данного паттерна, следующие: композиция – компоновщик (*Composite*) содержит коллекцию компонентов (*Component*); наследование – Лист (*Leaf*) и компоновщик наследуют общий интерфейс (*Component*).

Результаты паттерна:

1. Определяет иерархии классов, состоящие из примитивных и составных объектов. Из примитивных объектов можно составлять более сложные, которые, в свою очередь, участвуют в более сложных композициях и так далее. Любой клиент, ожидающий получить примитивный объект, может работать и с составным.

2. Упрощает архитектуру клиента. Клиенты могут единообразно работать с индивидуальными и объектами и с составными структурами. Обычно клиенту неизвестно, взаимодействует ли он с листовым или составным объектом. Это упрощает код клиента, поскольку нет необходимости писать функции, ветвящиеся в зависимости от того, с объектом какого класса они работают.

3. Облегчает добавление новых видов компонентов. Новые подклассы классов Composite или Leaf будут автоматически работать с уже существующими структурами и клиентским кодом. Изменять клиент при добавлении новых компонентов не нужно.

4. Способствует созданию общего дизайна. Впрочем, такая простота добавления новых компонентов имеет и свои отрицательные стороны: становится трудно установить ограничения на то, какие объекты могут входить в состав композиции. Иногда бывает нужно, чтобы составной объект мог включать только определенные виды компонентов. Паттерн Компоновщик не позволяет воспользоваться для реализации таких ограничений статической системой типов. Вместо этого приходится проводить проверки во время выполнения.

Основными недостатками паттерна заключаются в том, что добавление иерархической структуры усложняет код, особенно при большом количестве классов компонентов и компоновщиков. *Composite* позволяет работать с объектами через общий интерфейс, но это делает сложнее ограничить типы компонентов для определенных контейнеров (например, чтобы контейнер содержал только конкретные типы). Также может привести к избыточным операциям и высокому потреблению памяти при глубокой вложенности, поскольку каждое действие итерируется по всем элементам дерева. Становится труднее контролировать специфические особенности отдельных объектов, так как общий интерфейс абстрагирует их уникальные свойства.

Пример из реальной жизни: крупная компания, состоящая из нескольких подразделений, где каждый департамент включает разные команды, а команды состоят из отдельных сотрудников. При этом каждый сотрудник, команда или департамент – часть компании в целом. Компания может проводить общие мероприятия или собирать отчёты, и иногда нужно выполнить эту задачу для всех сотрудников сразу. Целое (*Composite*) – это департамент или компания, которые могут содержать подчиненные элементы: департаменты могут включать команды, а компании состоят из департаментов. Части (*Leaf*) – отдельные сотрудники, у которых нет подчиненных. Операция – например, запрос «предоставить отчёт». Паттерн Компоновщик позволяет отправить запрос одновременно всем сотрудникам, независимо от уровня вложенности. Каждый департамент пересылает запрос дальше, вплоть до индивидуальных сотрудников.

4.3.2 Пример на Python

Есть система, где требуется построить структуру папок и файлов. Папка может содержать другие папки и файлы, а файл – это конечный объект, который не может содержать ничего. Необходимо создать интерфейс, чтобы к папкам и файлам можно было обращаться одинаково.

```
class FileSystemItem(ABC):
    @abstractmethod
```

```
    def show_info(self, indent=0):
        pass
class File(FileSystemItem):
    def __init__(self, name, size):
        self.name = name
        self.size = size
    def show_info(self, indent=0):
        print(' ' * indent + f"File: {self.name}, Size: {self.size}KB")
class Directory(FileSystemItem):
    def __init__(self, name):
        self.name = name
        self.children = []
    def add(self, item: FileSystemItem):
        self.children.append(item)
    def remove(self, item: FileSystemItem):
        self.children.remove(item)
    def show_info(self, indent=0):
        print(' ' * indent + f"Directory: {self.name}")
        for child in self.children:
            child.show_info(indent + 2)
file1 = File("file1.txt", 100)
file2 = File("file2.txt", 200)
file3 = File("file3.txt", 300)
root_dir = Directory("root")
sub_dir1 = Directory("sub_dir1")
sub_dir2 = Directory("sub_dir2")
root_dir.add(file1)
root_dir.add(sub_dir1)
sub_dir1.add(file2)
sub_dir1.add(sub_dir2)
sub_dir2.add(file3)
root_dir.show_info()
```

Пояснение:

– *FileSystemItem* – это абстрактный базовый класс, представляющий Component. Он определяет общий метод *show_info*, который должен быть реализован в каждом подклассе;

– *File* – это класс, представляющий конечный объект (лист) в структуре, который не может содержать другие элементы. Он реализует метод *show_info*, который выводит информацию о файле с отступом, чтобы структура была читаемой;

– *Directory* – это составной объект, который может содержать как файлы, так и другие папки. В нем есть методы *add* и *remove* для добавления и удаления вложенных элементов, а также метод *show_info* для рекурсивного отображения всей структуры папок и файлов.

4.3.3 Пример на Java

Ниже приведен пример, аналогичный предыдущему, который реализует иерархию файлов и папок.

```
interface FileSystemItem {
    void showInfo(String indent);
}
class File implements FileSystemItem {
    private String name;
    private int size;
    public File(String name, int size) {
```

```
        this.name = name;
        this.size = size;
    }
    @Override
    public void showInfo(String indent) {
        System.out.println(indent + "File: " + name + ", Size: " + size +
"KB");
    }
}
class Directory implements FileSystemItem {
    private String name;
    private List<FileSystemItem> children = new ArrayList<>();
    public Directory(String name) {
        this.name = name;
    }
    public void add(FileSystemItem item) {
        children.add(item);
    }
    public void remove(FileSystemItem item) {
        children.remove(item);
    }
    @Override
    public void showInfo(String indent) {
        System.out.println(indent + "Directory: " + name);
        for (FileSystemItem child : children) {
            child.showInfo(indent + "  ");
        }
    }
}
public class Main {
    public static void main(String[] args) {
        File file1 = new File("file1.txt", 100);
        File file2 = new File("file2.txt", 200);
        File file3 = new File("file3.txt", 300);
        Directory rootDir = new Directory("root");
        Directory subDir1 = new Directory("sub_dir1");
        Directory subDir2 = new Directory("sub_dir2");
        rootDir.add(file1);
        rootDir.add(subDir1);
        subDir1.add(file2);
        subDir1.add(subDir2);
        subDir2.add(file3);
        rootDir.showInfo("");
    }
}
```

4.4 Паттерн Decorator (Декоратор)

4.4.1 Описание паттерна Decorator (Декоратор)

Это структурный паттерн проектирования, который позволяет динамически добавлять объектам новое поведение, не изменяя исходный код их классов. Это достигается за счёт обёртывания объекта в другой объект-декоратор, который расширяет или изменяет его поведение. Паттерн Декоратор помогает обойти ограничение жёсткой статической структуры классов. Вместо наследования, он использует композицию: объект оборачивается в несколько «слоёв» декораторов, каждый из которых добавляет новую функциональность.

Основные условия для применения паттерна декоратор:

1) динамическое, прозрачное для клиентов добавление обязанностей объектам (не затрагивающее другие объекты);

2) реализация обязанностей, которые могут быть сняты с объекта;

3) расширение путем порождения подклассов по каким-то причинам неудобно или невозможно. Иногда приходится реализовывать много независимых расширений, так что порождение подклассов для поддержки всех возможных комбинаций приведет к стремительному росту их числа. В других случаях определение класса может быть скрыто или почему-либо еще недоступно, так что породить от него подкласс нельзя.

Основные элементы паттерна:

1. Компонент (*Component*) – базовый интерфейс или абстрактный класс, описывающий поведение, которое может быть расширено. Например, базовый метод *operation*().

2. Конкретный компонент (*ConcreteComponent*) – класс, реализующий базовое поведение компонента, которое затем будет расширяться декораторами.

3. Базовый декоратор (*Decorator*) – абстрактный класс или интерфейс, реализующий тот же интерфейс, что и компонент. Содержит ссылку на объект компонента, который декорируется. В базовом декораторе методы интерфейса делегируются компоненту, что позволяет расширять их в наследниках.

4. Конкретные декораторы (*ConcreteDecorators*) – классы, расширяющие функциональность компонента. Они изменяют или дополняют базовое поведение, добавленное к объекту, и могут включать дополнительные поля и методы для управления своим поведением.

Основные отношения между сущностями в UML-диаграмме классов для данного паттерна, следующие: композиция – декоратор содержит ссылку на базовый компонент; реализация – декораторы и основной компонент реализуют общий интерфейс.

Результаты паттерна:

1. Большая гибкость, нежели у статического наследования. Паттерн Декоратор позволяет более гибко добавлять объекту новые обязанности, чем было бы возможно в случае статического (множественного) наследования. Декоратор может добавлять и удалять обязанности во время выполнения программы. С другой стороны, при использовании наследования требуется создавать новый класс для каждой дополнительной обязанности, что ведет к увеличению числа классов и, как следствие, к возрастанию сложности системы. Кроме того, применение нескольких декораторов к одному компоненту позволяет формировать произвольные комбинации обязанностей.

2. Декораторы также позволяют легко добавить одно и то же свойство дважды. Например, чтобы окружить объект *TextView* двойной рамкой, нужно просто добавить два декоратора *BorderDecorators*. Двойное наследование классу *Border* в лучшем случае чревато ошибками.

3. Позволяет избежать перегруженных функциями классов на верхних уровнях иерархии. Декоратор разрешает добавлять новые обязанности по мере необходимости. Вместо того чтобы пытаться поддержать все мыслимые возможности в одном сложном, допускающем разностороннюю настройку классе, можно определить простой класс и постепенно наращивать его функциональность с помощью декораторов. В результате приложение уже не перегружается неиспользуемыми функциями. Нетрудно также определять новые виды декораторов независимо от классов, которые они расширяют, даже если первоначально такие расширения не планировались. При расширении же сложного класса обычно приходится вникать в детали, не имеющие отношения к добавляемой функции.

4. Декоратор и его компонент не идентичны. Декоратор действует как прозрачное обрамление. Но декорированный компонент все же не идентичен исходному. При использовании декораторов это следует иметь в виду.

5. Множество мелких объектов. При использовании в проекте паттерна декоратор нередко формируется система, составленная из большого числа мелких объектов, похожих друг на друга. Такие объекты различаются только способом взаимосвязи, а не классом и не значениями своих атрибутов (полей). Хотя такие системы легко настраиваются проектировщиком, хорошо разбирающимся в их строении, изучать и отлаживать их очень тяжело.

Основными недостатками паттерна заключаются в том, что использование паттерна может привести к большому количеству маленьких классов, что усложняет код. Сложнее отладить, так как многоуровневые обёртки могут затруднять понимание.

Пример из реальной жизни может служить добавление различных ингредиентов к основному блюду в ресторане, например, к чашке кофе. Клиент заказывает чашку кофе в кафе, и есть возможность добавить к нему различные ингредиенты: молоко, шоколад, сироп и т.д. Каждый из этих ингредиентов увеличивает стоимость кофе и добавляет дополнительные вкусы, но базовый кофе остаётся тем же. Кофе – это базовый напиток, который можно модифицировать. Ингредиенты – это «декораторы», которые добавляются к базовому напитку, чтобы изменить его вкус и стоимость.

4.4.2 Пример на Python

Требуется создать декоратор, который будет выводить в консоль сообщение перед и после вызова функции, показывая, что выполнение функции началось и закончилось. Такой декоратор может быть полезен для отладки.

```
def my_decorator(func):
    def wrapper(*args, **kwargs):
        print("Начало выполнения функции...")
        result = func(*args, **kwargs)
        print("Выполнение функции завершено.")
        return result
    return wrapper
@my_decorator
def say_hello():
    print("Привет, мир!")
say_hello()
```

Пояснение:

– функция-декоратор *my_decorator*(*func*) принимает функцию *func*, которую нужно декорировать. Внутри декоратора создаётся вложенная функция *wrapper*(**args*, ***kwargs*), которая «оборачивает» логику основной функции и может добавлять дополнительный код;

– функция-обёртка *wrapper*(**args*, ***kwargs*), использует параметры **args* и ***kwargs*, чтобы поддерживать передачу любого количества аргументов в функцию. Внутри обёртки выполняется: сообщение о начале выполнения, вызов исходной функции и сохранение её результата в *result*, сообщение о завершении и возврат результата выполнения исходной функции;

– запись *@my_decorator* перед определением функции *say_hello*, Python автоматически оборачивает *say_hello* с помощью декоратора *my_decorator*. Когда происходит вызов *say_hello*(),

фактически выполняется код в *wrapper*, который сначала выводит сообщение, затем вызывает *say_hello*, а затем выводит финальное сообщение.

4.4.3 Пример на Java

В отличие от Python, Java не поддерживает функции как объекты, и декораторы реализуются как классы, которые оборачивают другие классы и добавляют к ним дополнительное поведение.

Реализация аналогичного примера, где нужно добавить дополнительное поведение до и после выполнения метода, используя паттерн Декоратор. Допустим, есть базовый интерфейс *Service*, который определяет метод *execute*. Нужно создать декоратор, который будет выводить сообщение перед началом выполнения и после выполнения метода *execute*.

```
public interface Service {
    void execute();
}
public class SimpleService implements Service {
    @Override
    public void execute() {
        System.out.println("Привет, мир!");
    }
}
```

Создание декоратора *LoggingDecorator*, который будет добавлять поведение перед и после вызова *execute* у основного сервиса. Декоратор реализует интерфейс *Service*, но внутри он будет использовать другой объект типа *Service*:

```
public class LoggingDecorator implements Service {
    private Service wrapped;
    public LoggingDecorator(Service wrapped) {
        this.wrapped = wrapped;
    }
    @Override
    public void execute() {
        System.out.println("Начало выполнения функции...");
        wrapped.execute();
        System.out.println("Выполнение функции завершено.");
    }
}
```

Использование декоратора, для начала нужно создать объект *SimpleService* и обернуть его в *LoggingDecorator*. При вызове метода *execute* у *LoggingDecorator* будут выполнены дополнительные действия, а затем вызов метода у оригинального сервиса:

```
public class Main {
    public static void main(String[] args) {
        Service service = new SimpleService();
        Service decoratedService = new LoggingDecorator(service);
        decoratedService.execute();
    }
}
```

Пояснение:

– интерфейс *Service* – определяет метод *execute*(), который нужно реализовать для всех сервисов;

– класс *SimpleService* – это базовая реализация интерфейса *Service*. Метод *execute*() в этом классе просто выводит сообщение «Привет, мир!»;

– класс *LoggingDecorator* принимает в конструкторе объект *Service*, который будет оборачиваться. Реализует метод *execute*(), добавляя логику до и после вызова *execute*() оборачиваемого объекта. Таким образом, *LoggingDecorator* позволяет расширить функциональность любого объекта, реализующего *Service*, не изменяя его исходный код.

4.5 Паттерн Facade (Фасад)

4.5.1 Описание паттерна Facade (Фасад)

Это структурный паттерн проектирования, который предоставляет простой интерфейс к сложной системе классов, библиотек или фреймворков. Паттерн упрощает использование системы, скрывая её внутреннюю сложность и объединяя функциональность в одном месте. Фасад создаёт высокоуровневый интерфейс, который содержит необходимый набор методов и позволяет пользователям не погружаться в детали внутренней структуры системы.

Основные условия для применения паттерна Фасад:

1) упрощение интерфейса для взаимодействия с подсистемами или сложной логикой;

2) создание единой точки доступа к сложной системе, позволяя клиентам не взаимодействовать с многочисленными классами и их методами;

3) разделение уровней подсистем для уменьшения связности между клиентами и сложной логикой.

Основные элементы паттерна:

1. Фасад (*Facade*) – класс, который обеспечивает простой интерфейс к сложной системе, инкапсулируя взаимодействие с различными подсистемами. Клиенты обращаются к фасаду, чтобы выполнять комплексные операции, не обращаясь напрямую к внутренним классам.

2. Подсистемы (*Subsystems*) – классы, реализующие основную функциональность системы. Эти классы могут быть сложными и иметь сложные зависимости. Фасад вызывает методы подсистем в правильном порядке и комбинирует результаты, чтобы упростить их для клиента.

Основные отношения между сущностями в UML-диаграмме классов для данного паттерна, следующие: ассоциация – фасад взаимодействует с подсистемами; зависимость – клиенты зависят от фасада, а не от подсистем напрямую.

Результаты паттерна:

1. Он изолирует клиентов от компонентов подсистемы, уменьшая тем самым число объектов, с которыми клиентам приходится иметь дело, и упрощая работу с подсистемой.

2. Позволяет ослабить связанность между подсистемой и ее клиентами. Зачастую компоненты подсистемы сильно связаны. Слабая связанность позволяет видоизменять компоненты, не затрагивая при этом клиентов. Фасады помогают разложить систему на уровни и структурировать зависимости между объектами, а также избежать сложных и циклических зависимостей. Это может оказаться важным, если клиент и подсистема реализуются независимо.

3. Фасад не препятствует приложениям напрямую обращаться к классам подсистемы, если это необходимо. Таким образом, у вас есть выбор между простотой и общностью.

Недостаток паттерна в том, что он может затруднить доступ к расширенной функциональности. Когда вся сложная логика завёрнута в фасад, иногда это ограничивает возможности

клиента в точной настройке параметров и использовании расширенных функций. Возможные проблемы с обновлением. В больших системах фасад может постепенно становиться перегруженным, если в него добавляются новые методы для доступа ко всем возможностям подсистем. Это может привести к трудностям в поддержке и разрастанию его функциональности.

Пример из реальной жизни: фасадом можно считать универсальный пульт дистанционного управления для бытовой техники. Вместо того чтобы каждый раз искать нужный пульт для телевизора, кондиционера или музыкального центра, можно использовать универсальный пульт, который содержит основные функции для управления всеми устройствами. Пульт является фасадом, упрощающим доступ к сложным и различным системам, такими как телевизор или кондиционер, объединяя их под одним интерфейсом для удобства клиента.

4.5.2 Пример на Python

Есть система домашнего кинотеатра с несколькими компонентами: телевизор, усилитель и DVD-плеер. Требуется создать фасад, который позволит запустить все эти устройства одной командой.

```
class TV:
    def on(self):
        print("Телевизор включен.")
    def off(self):
        print("Телевизор выключен.")
class Amplifier:
    def on(self):
        print("Усилитель включен.")
    def setVolume(self, level):
        print(f"Громкость установлена на уровень {level}.")
    def off(self):
        print("Усилитель выключен.")
class DVDPlayer:
    def on(self):
        print("DVD-плеер включен.")
    def play(self, movie):
        print(f"Воспроизведение фильма: '{movie}'")
    def off(self):
        print("DVD-плеер выключен.")
class HomeTheaterFacade:
    def __init__(self, tv: TV, amplifier: Amplifier, dvd: DVDPlayer):
        self.tv = tv
        self.amplifier = amplifier
        self.dvd = dvd
    def watchMovie(self, movie: str):
        print("Подготовка к просмотру фильма...")
        self.tv.on()
        self.amplifier.on()
        self.amplifier.setVolume(5)
        self.dvd.on()
        self.dvd.play(movie)
        print("Наслаждайтесь фильмом!\n")
    def endMovie(self):
        print("Завершение просмотра фильма...")
        self.dvd.off()
        self.amplifier.off()
        self.tv.off()
```

```
        print("Система выключена.\n")
tv = TV()
amplifier = Amplifier()
dvd = DVDPlayer()
home_theater = HomeTheaterFacade(tv, amplifier, dvd)
home_theater.watchMovie("Интерстеллар")
home_theater.endMovie()
```

Пояснение:

– классы компонентов (*TV*, *Amplifier*, *DVDPlayer*) представляют собой отдельные устройства системы домашнего кинотеатра. Каждый класс имеет методы *on*() и *off*() для включения и выключения устройства, а также другие специфические методы, например, *play*() у DVD-плеера и *setVolume*() у усилителя;

– класс *HomeTheaterFacade* является фасадом, он содержит методы *watchMovie*() и *endMovie*(), которые упрощают использование системы. В методе *watchMovie*() фасад последовательно включает все устройства и подготавливает их к воспроизведению фильма. В методе *endMovie*() фасад выполняет обратные действия для завершения просмотра и выключения системы;

– вместо того чтобы управлять каждым компонентом по отдельности, используется фасад (*home_theater*), который делает всю работу. Вызывая *home_theater.watchMovie*("Интерстеллар"), клиентский код упрощает взаимодействие с системой до одной строки, без необходимости заботиться о внутренних деталях.

4.5.3 Пример на Java

Реализация фасада для управления домашним кинотеатром, где отдельные компоненты – телевизор, усилитель и DVD-плеер – будут объединены в одном классе фасада.

```
class TV {
    public void on() {
        System.out.println("Телевизор включен.");
    }
    public void off() {
        System.out.println("Телевизор выключен.");
    }
}
class Amplifier {
    public void on() {
        System.out.println("Усилитель включен.");
    }
    public void setVolume(int level) {
        System.out.println("Громкость установлена на уровень " + level +
".");
    }
    public void off() {
        System.out.println("Усилитель выключен.");
    }
}
class DVDPlayer {
    public void on() {
        System.out.println("DVD-плеер включен.");
    }
    public void play(String movie) {
        System.out.println("Воспроизведение фильма: '" + movie + "'");
```

```
    }
    public void off() {
        System.out.println("DVD-плеер выключен.");
    }
}
class HomeTheaterFacade {
    private TV tv;
    private Amplifier amplifier;
    private DVDPlayer dvd;
    public HomeTheaterFacade(TV tv, Amplifier amplifier, DVDPlayer dvd) {
        this.tv = tv;
        this.amplifier = amplifier;
        this.dvd = dvd;
    }
    public void watchMovie(String movie) {
        System.out.println("Подготовка к просмотру фильма...");
        tv.on();
        amplifier.on();
        amplifier.setVolume(5);
        dvd.on();
        dvd.play(movie);
        System.out.println("Наслаждайтесь фильмом!\n");
    }
    public void endMovie() {
        System.out.println("Завершение просмотра фильма...");
        dvd.off();
        amplifier.off();
        tv.off();
        System.out.println("Система выключена.\n");
    }
}
public class Main {
    public static void main(String[] args) {
        TV tv = new TV();
        Amplifier amplifier = new Amplifier();
        DVDPlayer dvd = new DVDPlayer();
        HomeTheaterFacade homeTheater = new HomeTheaterFacade(tv, ampli-
fier, dvd);
        homeTheater.watchMovie("Интерстеллар");
        homeTheater.endMovie();
    }
}
```

4.6 Паттерн Flyweight (Приспособленец)

4.6.1 Описание паттерна Flyweight (Приспособленец)

Структурный паттерн проектирования, который позволяет экономить память при работе с большим количеством объектов, представляющих собой повторяющиеся или схожие состояния. Паттерн выделяет повторяющиеся свойства объектов и переносит их в одно место, что позволяет уменьшить затраты на хранение и обработку этих свойств. Это особенно полезно в случаях, когда необходимо создать огромное количество небольших объектов, которые содержат дублирующиеся данные.

Основные условия для применения паттерна Приспособленец:

1) применение, если необходимо работать с множеством однотипных объектов, создание которых затратно по памяти;

2) уменьшение нагрузки на память за счёт вынесения повторяющихся данных в отдельные структуры;

3) разделение свойств объектов на постоянные и изменяющиеся: постоянные свойства могут храниться в одном месте и разделяться между объектами.

Основные элементы паттерна:

1. Приспособленец (*Flyweight*) – базовый интерфейс или абстрактный класс, представляющий объект с разделяемым состоянием. Этот интерфейс предоставляет методы для работы с объектом и управления общими (разделяемыми) и уникальными (независимыми) свойствами.

2. Конкретный приспособленец (*ConcreteFlyweight*) – класс, реализующий приспособленец и хранящий разделяемое состояние, которое может быть использовано многими объектами. Этот класс минимизирует использование памяти, сохраняя только общие данные.

3. Нелегковесный (*UnsharedFlyweight*) – классы, представляющие объекты с уникальным состоянием, не разделяемым с другими объектами. Они могут использоваться как дополнение к конкретному приспособленцу, обеспечивая гибкость в работе с уникальными свойствами.

4. Фабрика приспособленцев (*FlyweightFactory*) – класс, отвечающий за создание и управление приспособленцами. При запросе объекта фабрика проверяет, существует ли уже нужный объект с разделяемым состоянием. Если объект найден, он возвращается, если нет – создаётся новый.

5. Клиент (*Client*) хранит ссылки на одного или нескольких приспособленцев. Вычисляет или хранит внешнее состояние приспособленцев.

Основные отношения между сущностями в UML-диаграмме классов для данного паттерна, следующие: ассоциация – клиенты используют приспособленцы (*Flyweight*); композиция – приспособленец содержит разделяемое внутреннее состояние, а внешнее состояние управляется клиентами.

Результаты паттерна:

При использовании приспособленцев возможны затраты на передачу, поиск или вычисление внутреннего состояния на стадии выполнения, особенно если раньше оно хранилось как внутреннее. Однако такие затраты с лихвой компенсируются экономией памяти за счет совместного использования объектов-приспособленцев.

Экономия памяти обусловлена несколькими причинами:

1) уменьшение общего числа экземпляров;

2) сокращение объема памяти, необходимого для хранения внутреннего состояния;

3) вычисление, а не хранение внешнего состояния (если это действительно так).

Чем выше степень совместного использования приспособленцев, тем существеннее экономия. С увеличением объема совместного состояния экономия также возрастает. Самого большого эффекта удается добиться, когда суммарный объем внутренней и внешней информации о состоянии велик, а внешнее состояние вычисляется, а не хранится. Тогда совместное использование уменьшает стоимость хранения внутреннего состояния, а за счет вычислений сокращается память, отводимая под внешнее состояние.

Паттерн Приспособленец часто применяется вместе с компоновщиком для представления иерархической структуры в виде графа с совместно используемыми листовыми узлами. Из-за

разделения указатель на родителя не может храниться в листовом узле-приспособленце, а должен передаваться ему как часть внешнего состояния. Это оказывает заметное влияние на способ взаимодействия объектов иерархии между собой.

Недостатки паттерна, это усложнение кода, так как требуется выделения состояний на уникальные и общие. В сложных системах это может создать дополнительные трудности для разработки и поддержки. Доступ к объектам становится менее удобным, так как разделение состояния требует дополнительных проверок и вызовов. Особенно это касается ситуаций, когда объекты могут изменяться во времени. Неприменим, если объекты имеют мало общего состояния или их состояния сильно отличаются, так как при этом экономия памяти будет незначительной.

Пример из реальной жизни: примером может быть текстовый редактор, который работает с большим количеством символов. Вместо создания отдельных объектов для каждой буквы, редактор может создать объект для каждого символа, который будет хранить только изменяющиеся данные (например, положение и цвет), тогда как постоянные данные (например, форма символа) будут разделяться между всеми объектами одной буквы.

4.6.2 Пример на Python

Разрабатывается программа для рисования деревьев в парке. Может быть множество деревьев, и каждое дерево имеет такие свойства, как «тип дерева» (например, «дуб», «сосна»), «цвет», «текстура листьев». Эти свойства одинаковы для всех деревьев одного типа, поэтому нет смысла создавать отдельный объект для каждого дерева с такими же параметрами. Можно создать один объект для каждого уникального типа дерева и использовать его для рисования деревьев этого типа. Это и будет применением паттерна Приспособленец.

```
class TreeType:
    def __init__(self, name, color, texture):
        self.name = name
        self.color = color
        self.texture = texture
    def draw(self, x, y):
        print(f"Рисуем {self.name} дерева с цветом {self.color} и тексту-
рой {self.texture} на позиции ({x}, {y})")
# Класс Tree представляет конкретное дерево
class Tree:
    def __init__(self, x, y, tree_type):
        self.x = x
        self.y = y
        self.tree_type = tree_type
    def draw(self):
        self.tree_type.draw(self.x, self.y)
class TreeFactory:
    tree_types = {}
    @staticmethod
    def get_tree_type(name, color, texture):
        if (name, color, texture) not in TreeFactory.tree_types:
            print(f"Создаем новый тип дерева: {name}, цвет: {color}, тек-
стура: {texture}")
            TreeFactory.tree_types[(name, color, texture)] = Tree-
Type(name, color, texture)
        return TreeFactory.tree_types[(name, color, texture)]
class Forest:
```

```
    def __init__(self):
        self.trees = []
    def plant_tree(self, x, y, name, color, texture):
        tree_type = TreeFactory.get_tree_type(name, color, texture)
        tree = Tree(x, y, tree_type)
        self.trees.append(tree)
    def draw(self):
        for tree in self.trees:
            tree.draw()
forest = Forest()
forest.plant_tree(1, 1, "Дуб", "зеленый", "широкие листья")
forest.plant_tree(2, 2, "Дуб", "зеленый", "широкие листья")
forest.plant_tree(3, 3, "Сосна", "темно-зеленый", "иголки")
forest.plant_tree(4, 4, "Сосна", "темно-зеленый", "иголки")
forest.draw()
```

Пояснение:

– *TreeType* – класс представляет собой «приспособленца». Он хранит свойства дерева, которые не меняются между экземплярами одного типа (имя, цвет, текстура). Содержит метод *draw*(), который выводит на экран данные о дереве с заданными координатами;

– *Tree* – класс представляет конкретное дерево, имеющее уникальные координаты (*x*, *y*). Класс *Tree* использует объект *TreeType* для хранения общих свойств дерева. Он вызывает метод *draw*() своего *TreeType*, передавая свои координаты;

– *TreeFactory* – статический класс, который управляет объектами *TreeType*. Метод *get_tree_type*() проверяет, существует ли уже объект *TreeType* с указанными свойствами. Если нет, он создает новый объект *TreeType* и сохраняет его в словаре. Это позволяет избежать дублирования объектов *TreeType* для одинаковых деревьев;

– *Forest* – класс управляет всеми деревьями и позволяет их сажать (добавлять) и рисовать. Когда вызывается *plant_tree*(), класс *Forest* обращается к *TreeFactory* для получения подходящего объекта *TreeType* и создает объект *Tree* с нужными координатами.

4.6.3 Пример на Java

Реализация аналогичного примера на Java, чтобы лучше понять, как работает паттерн Приспособленец.

```
class TreeType {
    private String name;
    private String color;
    private String texture;
    public TreeType(String name, String color, String texture) {
        this.name = name;
        this.color = color;
        this.texture = texture;
    }
    public void draw(int x, int y) {
        System.out.println("Рисуем " + name + " с цветом " + color +
                           " и текстурой " + texture + " на позиции (" + x
+ ", " + y + ")");
    }
}
class Tree {
    private int x;
    private int y;
```

```
    private TreeType treeType;
    public Tree(int x, int y, TreeType treeType) {
        this.x = x;
        this.y = y;
        this.treeType = treeType;
    }
    public void draw() {
        treeType.draw(x, y);
    }
}
class TreeFactory {
    private static HashMap<String, TreeType> treeTypes = new HashMap<>();
    public static TreeType getTreeType(String name, String color, String
texture) {
        String key = name + "-" + color + "-" + texture;
        if (!treeTypes.containsKey(key)) {
            System.out.println("Создаем новый тип дерева: " + name + ",
цвет: " + color + ", текстура: " + texture);
            treeTypes.put(key, new TreeType(name, color, texture));
        }
        return treeTypes.get(key);
    }
}
class Forest {
    private List<Tree> trees = new ArrayList<>();
    public void plantTree(int x, int y, String name, String color, String
texture) {
        TreeType treeType = TreeFactory.getTreeType(name, color, texture);
        Tree tree = new Tree(x, y, treeType);
        trees.add(tree);
    }
    public void draw() {
        for (Tree tree : trees) {
            tree.draw();
        }
    }
}
public class Main {
    public static void main(String[] args) {
        Forest forest = new Forest();
        forest.plantTree(1, 1, "Дуб", "зеленый", "широкие листья");
        forest.plantTree(2, 2, "Дуб", "зеленый", "широкие листья");
        forest.plantTree(3, 3, "Сосна", "темно-зеленый", "иголки");
        forest.plantTree(4, 4, "Сосна", "темно-зеленый", "иголки");
        forest.draw();
    }
}
```

4.7 Паттерн Proxy (Заместитель)

4.7.1 Описание паттерна Proxy (Заместитель)

Структурный паттерн проектирования, который предоставляет суррогатный объект, управляющий доступом к другому объекту. Паттерн Заместитель позволяет контролировать доступ, управление и дополнительное поведение, не изменяя код самого объекта. Это достигается

путём создания объекта-заместителя, который оборачивает оригинальный объект и перехватывает обращения к нему. Паттерн помогает разделить логику доступа и выполнения операций на два класса: основной объект и заместитель.

Паттерн Заместитель применим во всех случаях, когда возникает необходимость сослаться на объект более гибким или нетривиальным способом, чем при использовании простого указателя. Несколько типичных ситуаций, в которых заместитель может оказаться полезным:

1. Необходимость контроля доступа к объекту, например, для проверки прав доступа или логирования.

2. Оптимизация ресурсов, когда создание или использование реального объекта дорогостоящи или нежелательны (например, работа с удалёнными ресурсами).

3. Обеспечение безопасного доступа к объекту в многопоточной среде.

4. Выполнение действий до или после передачи вызова реальному объекту (например, кэширование результатов).

Основные элементы паттерна:

1. Интерфейс (*Subject*) – общий интерфейс для реального объекта и его заместителя. Например, метод request(), предоставляющий общую функциональность.

2. Реальный объект (*RealSubject*) – класс, реализующий основное поведение и функциональность.

3. Заместитель (*Proxy*) – класс, реализующий тот же интерфейс, что и реальный объект. Содержит ссылку на реальный объект и управляет вызовами к нему.

Варианты заместителей:

1. Виртуальный заместитель – отложенная инициализация объекта. Заместитель создаёт реальный объект только при необходимости, экономя ресурсы.

2. Удалённый заместитель – обёртка для взаимодействия с объектами, находящимися в другом адресном пространстве (например, на удалённом сервере).

3. Защищающий заместитель – проверяет права доступа перед передачей вызова реальному объекту.

4. Кэширующий заместитель – хранит результаты вызовов для оптимизации повторных запросов.

Основные отношения между сущностями в UML-диаграмме классов для данного паттерна, следующие: ассоциация – прокси содержит ссылку на реальный объект; реализация – прокси и реальный объект реализуют общий интерфейс.

Результаты паттерна:

1. Контроль доступа. Паттерн позволяет перехватывать вызовы к объекту, чтобы проверять права доступа или добавлять вспомогательную логику (например, логирование).

2. Оптимизация ресурсов. Заместитель позволяет создавать ресурсоёмкий объект только по мере необходимости. Это может быть полезно для объектов, требующих сложной инициализации.

3. Снижение задержек в сетевых взаимодействиях. Удалённый заместитель минимизирует издержки за счёт упаковки и отправки запросов.

4. Чистота архитектуры. Паттерн помогает выделить дополнительную логику (например, кэширование или синхронизацию) из основного класса, что улучшает читаемость и тестируемость кода.

Недостатки паттерна, усложнение системы, добавляя дополнительные уровни абстракции. Может вызвать затруднения при отладке и тестировании из-за дополнительной логики в заместителе. Заместитель и реальный объект могут казаться идентичными снаружи, но их поведение при ошибках или в сложных сценариях может отличаться.

Пример из реальной жизни – банковский сейф. Чтобы получить доступ к сейфу, клиент должен обратиться к банковскому служащему. Служащий проверяет документы и право доступа, после чего предоставляет ключ или открывает сейф. Здесь сам сейф – это реальный объект, а служащий – заместитель, который контролирует доступ.

4.7.2 Пример на Python

Есть объект, представляющий секретные данные. Доступ к этим данным можно получить только с правильным паролем. Используем Заместитель для проверки пароля.

```
from abc import ABC, abstractmethod
class SecretDataInterface(ABC):
    @abstractmethod
    def get_data(self) -> str:
        pass
class SecretData(SecretDataInterface):
    def __init__(self):
        self._data = "Это секретные данные!"
    def get_data(self) -> str:
        return self._data
class SecretDataProxy(SecretDataInterface):
    def __init__(self, password: str):
        self._real_data = None
        self._password = password
    def get_data(self) -> str:
        if self._password == "correct_password":
            if not self._real_data:
                self._real_data = SecretData()
            return self._real_data.get_data()
        else:
            return "Доступ запрещён: неверный пароль."
def main():
    print("=== Попытка доступа с неверным паролем ===")
    proxy1 = SecretDataProxy("wrong_password")
    print(proxy1.get_data())
    print("\n=== Попытка доступа с правильным паролем ===")
    proxy2 = SecretDataProxy("correct_password")
    print(proxy2.get_data())
```

Пояснение:

– *SecretDataInterface* – интерфейс, который определяет метод *get_data*(). Это контракт, которому должны следовать и реальный объект, и заместитель;

– *SecretData* – класс с "тяжёлой" инициализацией, содержащий секретные данные;

– *SecretDataProxy* – заместитель, который проверяет пароль перед доступом к данным. Он создаёт объект *SecretData* только при первом запросе с правильным паролем (объект *SecretData* создаётся только в случае правильного пароля).

4.7.3 Пример на Java

Ниже представлен аналогичный пример на Java, где используется Заместитель для контроля доступа к секретным данным.

```
interface SecretDataInterface {
    String getData();
}
class SecretData implements SecretDataInterface {
    private String data;
    public SecretData() {
        this.data = "Это секретные данные!";
    }
    @Override
    public String getData() {
        return data;
    }
}
class SecretDataProxy implements SecretDataInterface {
    private SecretData realData;
    private String password;
    public SecretDataProxy(String password) {
        this.password = password;
    }
    @Override
    public String getData() {
        if ("correct_password".equals(password)) {
            if (realData == null) {
                realData = new SecretData();
            }
            return realData.getData();
        } else {
            return "Доступ запрещён: неверный пароль.";
        }
    }
}
public class ProxyExample {
    public static void main(String[] args) {
        System.out.println("=== Попытка доступа с неверным паролем ===");
        SecretDataInterface proxy1 = new SecretDataProxy("wrong_pass-
word");
        System.out.println(proxy1.getData());
        System.out.println("\n=== Попытка доступа с правильным паролем
===");
        SecretDataInterface proxy2 = new SecretDataProxy("correct_pass-
word");
        System.out.println(proxy2.getData());
    }
}
```

4.8 Контрольные вопросы для самоподготовки

Тема 1. Проектирование и реализация паттерна «Адаптер» (Adapter)

Цель самоподготовки: практическое освоение механизмов интеграции существующих компонентов с несовместимыми интерфейсами в новую систему без модификации их исходного кода.

1. Проектирование целевого контракта (*Target*). Каким образом выделение абстракции *Target* позволяет стандартизировать взаимодействие клиента с системой? Опишите процесс описания методов, которые ожидаются клиентским кодом в рамках вашей предметной области.

2. Анализ адаптируемого компонента (*Adaptee*). С какими ограничениями сталкивается разработчик при интеграции сторонних библиотек или унаследованного (*legacy*) кода? Обоснуйте выбор компонента *Adaptee*, чья бизнес-логика необходима системе, но интерфейс является несовместимым с целевым.

3. Архитектура объектного адаптера. Спроектируйте класс *Adapter*. Почему в современной разработке предпочтительнее использовать «объектный адаптер» на основе композиции/агрегации, а не «адаптер класса» на основе множественного наследования? Опишите механизм инкапсуляции экземпляра *Adaptee* внутри адаптера.

4. Механика трансляции вызовов. Опишите алгоритм преобразования данных внутри методов адаптера. Каким образом осуществляется перенаправление вызова от интерфейса *Target* к специфическим методам внутреннего объекта *Adaptee*? Приведите пример возможной трансформации типов данных в этом процессе.

5. Обеспечение прозрачности интеграции. Спроектируйте клиентский код, взаимодействующий с системой. Как гарантировать на уровне архитектуры, что клиент остается в неведении относительно существования *Adaptee* и работает исключительно через *Target*? В чем преимущество такой изоляции?

6. Визуализация иерархии зависимостей в UML. Какие типы связей должны быть использованы на диаграмме классов для отображения паттерна «Адаптер»? Объясните разницу между отношением реализации (между адаптером и целевым интерфейсом) и отношением направленной ассоциации (между адаптером и адаптируемым классом).

Тема 2. Проектирование и реализация паттерна «Мост» (Bridge)

Цель самоподготовки: Практическое освоение архитектурного приема разделения предметной абстракции и ее физической реализации на две независимые иерархии классов для предотвращения их экспоненциального разрастания.

1. Анализ ортогональных измерений системы. Опишите проблему «комбинаторного взрыва» количества классов при использовании классического наследования. Каким образом выделение двух независимых векторов изменения (например, «интерфейс» и «платформа») позволяет оптимизировать структуру системы?

2. Проектирование иерархии реализации (*Implementor*). Спроектируйте базовый интерфейс *Implementor*. Почему он должен содержать только примитивные операции? Объясните, как конкретные реализаторы (*ConcreteImplementor*) адаптируют эти операции под специфику различных платформ или API.

3. Структура базовой абстракции (Abstraction). Какова роль класса *Abstraction* в данной архитектуре? Обоснуйте необходимость инкапсуляции поля ссылочного типа, указывающего на *Implementor*, и опишите механизм передачи конкретного объекта реализации в этот класс (например, через внедрение зависимостей в конструкторе).

4. Механизм делегирования поведения. Опишите процесс выполнения высокоуровневых методов в классе абстракции. Каким образом работа делегируется объекту-реализатору? Приведите пример того, как один высокоуровневый метод абстракции может быть реализован через серию вызовов примитивных операций реализатора.

5. Развитие уточненных абстракций (*Refined Abstraction*). Спроектируйте подкласс *RefinedAbstraction*. Каким образом обеспечивается независимость иерархий: почему добавление нового функционала в иерархию абстракций не требует изменения кода в иерархии реализаций (и наоборот)?

6. Визуализация параллельных иерархий в UML. Как на UML-диаграмме классов наглядно отобразить «мост» между абстракцией и реализацией? Объясните значение отношения агрегации (или композиции) между базовым классом абстракции и интерфейсом реализации как ключевого элемента данного паттерна.

Тема 3. Проектирование и реализация паттерна «Компоновщик» (Composite)

Цель самоподготовки: практическое освоение архитектурного подхода для построения древовидных структур объектов и обеспечения единообразного взаимодействия клиентского кода как с отдельными объектами, так и с их композициями.

1. Определение базового контракта (*Component*). Каким образом проектирование общего интерфейса *Component* позволяет реализовать полиморфизм для простых и составных элементов? Опишите набор бизнес-операций, которые должны быть применимы к любому узлу древовидной структуры.

2. Специфика простых элементов (*Leaf*). В чем заключается роль классов «листьев» в иерархии? Объясните, почему эти элементы должны самостоятельно выполнять конечную бизнес-логику и как они сигнализируют системе об отсутствии вложенных объектов.

3. Архитектура составных элементов (*Composite*). Спроектируйте класс узла-контейнера. Какую структуру данных целесообразно использовать для хранения коллекции ссылок на дочерние элементы? Обоснуйте выбор методов для управления иерархией (добавление, удаление, получение узлов).

4. Механизм рекурсивного делегирования. Опишите алгоритм выполнения бизнес-операций в классе *Composite*. Каким образом реализуется итерирование по коллекции и делегирование вызовов дочерним элементам? В каких случаях необходимо аккумулировать результаты выполнения методов от всех потомков?

5. Прозрачность взаимодействия в клиентском коде. Спроектируйте фрагмент кода, создающий многоуровневое дерево. Как обеспечить работу клиента с корнем дерева исключительно через абстракцию *Component*? Почему для клиента должна отсутствовать необходимость проверки конкретных типов узлов перед вызовом методов?

6. Визуализация рекурсивных связей в UML. Какая графическая особенность на диаграмме классов является ключевой для паттерна «Компоновщик»? Объясните значение рефлексивного отношения агрегации, направленного от класса *Composite* к интерфейсу *Component*, и его сочетание с отношением реализации базового контракта.

Тема 4. Проектирование и реализация паттерна «Декоратор» (Decorator)

Цель самоподготовки: практическое освоение архитектурного приема динамического расширения функциональности объектов во время выполнения программы как гибкой альтернативы статическому наследованию.

1. Проектирование единого контракта (*Component*). Какова роль общего интерфейса *Component* в обеспечении возможности бесконечного вложения декораторов друг в друга? Опишите, как этот интерфейс уравнивает в правах базовые объекты и их расширения.

2. Реализация целевого объекта (*ConcreteComponent*). Спроектируйте класс *ConcreteComponent*. Опишите его ответственность как носителя основной бизнес-логики и объясните, почему этот класс не должен знать о существовании будущих декораторов.

3. Архитектура базового декоратора. Обоснуйте необходимость создания абстрактного класса *Decorator*. Каким образом инкапсуляция ссылки на *Component* и реализация делегирования по умолчанию позволяют строить цепочки вызовов? Почему декоратор сам должен реализовывать интерфейс компонента?

4. Механизм добавления новой функциональности. Опишите процесс расширения поведения в классах *ConcreteDecorator*. В какой момент (до или после делегирования вызова обернутому объекту) целесообразно добавлять новую логику? Приведите пример расширения состояния объекта через декоратор.

5. Динамическое конструирование объектов. Спроектируйте фрагмент клиентского кода, создающий цепочку из нескольких декораторов. Как обеспечить прозрачность взаимодействия, чтобы клиент работал с итоговой конструкцией исключительно через базовую абстракцию, не зная о количестве и типах «оберток»?

6. Визуализация рекурсивной структуры в UML. Какие ключевые связи на диаграмме классов отличают «Декоратор» от обычного наследования? Объясните значение сочетания отношения реализации интерфейса *Component* и отношения агрегации (направленного к этому же интерфейсу) в классе *Decorator*.

Тема 5. Проектирование и реализация паттерна «Фасад» (Facade)

Цель самоподготовки: практическое освоение архитектурного приема инкапсуляции сложной подсистемы за единым упрощенным интерфейсом для снижения связности системы и зависимости клиентского кода от низкоуровневых компонентов.

1. Анализ сложности подсистемы. Опишите признаки «сложной подсистемы», которые указывают на необходимость внедрения фасада. Почему прямое взаимодействие клиента с группой взаимозависимых классов или сторонних API считается плохой практикой проектирования в контексте сопровождения кода?

2. Проектирование интерфейса фасада. Спроектируйте высокоуровневый класс *Facade*. Каким образом следует определять ограниченный набор методов фасада, ориентируясь на наиболее востребованные клиентские сценарии (*use cases*)?

3. Реализация делегирования и оркестрации. Опишите внутреннюю логику методов фасада. Как фасад должен обеспечивать правильный порядок инициализации и конфигурации объектов подсистемы? Объясните механизм трансляции одного простого запроса клиента в последовательность вызовов методов различных компонентов подсистемы.

4. Изоляция и снижение связности (*Coupling*). Каким образом использование фасада позволяет реорганизовать клиентский код так, чтобы полностью исключить его зависимость от интерфейсов внутренней подсистемы? К каким положительным эффектам это приводит при необходимости замены или обновления компонентов подсистемы?

5. Визуализация единой точки входа в UML. Спроектируйте UML-диаграмму классов, демонстрирующую роль фасада. Как графически отобразить отношение зависимости клиента от фасада и отношения направленной ассоциации (или композиции) от фасада ко всем скрытым за ним классам?

Тема 6. Проектирование и реализация паттерна «Приспособленец» (Flyweight)

Цель самоподготовки: практическое освоение архитектурного приема оптимизации потребления оперативной памяти путем разделения состояния объектов и совместного использования общих данных множеством легковесных экземпляров.

1. Анализ и разделение состояний. Опишите критерии, по которым атрибуты объекта разделяются на внутреннее (*intrinsic state*) и внешнее (*extrinsic state*) состояния. Почему для эффективной оптимизации важно, чтобы внутреннее состояние было неизменяемым и одинаковым для большого массива объектов?

2. Проектирование контракта приспособленца. Спроектируйте интерфейс *Flyweight*. Каким образом должны быть описаны его методы, чтобы они могли оперировать внешним состоянием, не сохраняя его внутри самого объекта? Объясните роль аргументов вызова в данной архитектуре.

3. Реализация конкретных приспособленцев. Опишите процесс создания класса *ConcreteFlyweight*. Каким образом обеспечивается строгая иммутабельность внутреннего состояния после инициализации? Почему недопустимо изменять поля приспособленца после его попадания в пул?

4. Механика работы фабрики приспособленцев. Спроектируйте класс *FlyweightFactory*. Опишите алгоритм кэширования: как фабрика должна проверять наличие существующего экземпляра в пуле и в какой момент принимается решение об инстанцировании нового объекта? Какую роль здесь играет «ключ» внутреннего состояния?

5. Перенос ответственности в клиентском коде. Каким образом должна измениться логика клиента при переходе на использование паттерна «Приспособленец»? Объясните, почему ответственность за вычисление и хранение внешнего (контекстного) состояния теперь ложится на клиентский код.

6. Визуализация системы управления ресурсами в UML. Какие ключевые зависимости должны быть отражены на диаграмме классов для паттерна «Приспособленец»? Опишите связь между клиентом, фабрикой и интерфейсом приспособленца, а также способ отображения агрегации пула объектов внутри фабрики.

Тема 7. Проектирование и реализация паттерна «Заместитель» (Proxy)

Цель самоподготовки: практическое освоение архитектурного приема создания объекта-суррогата для перехвата клиентских вызовов и контроля доступа к целевому объекту без модификации его исходного кода.

1. Проектирование общего контракта (*Subject*). Зачем необходимо выделение единого интерфейса *Subject* как для реального сервиса, так и для его заместителя? Опишите, как общность интерфейса обеспечивает прозрачность подмены объектов для клиентского кода.

2. Реализация целевого объекта (*RealSubject*). Какие типы задач целесообразно инкапсулировать в классе *RealSubject*? Обоснуйте, почему ресурсоемкие операции (например, тяжелые вычисления или запросы к БД) делают применение паттерна «Заместитель» архитектурно оправданным.

3. Структура класса-заместителя. Опишите процесс проектирования класса *Proxy*. Каким образом инкапсуляция ссылки на *RealSubject* позволяет заместителю управлять жизненным циклом реального объекта?

4. Механизмы перехвата и служебной логики. Проанализируйте различные сценарии реализации перехвата:

– виртуальный заместитель – как реализовать «ленивую инициализацию» объекта только в момент первого обращения?

– защищающий заместитель – как организовать проверку прав доступа перед делегированием вызова?

– логирующий заместитель – в какой момент (до или после вызова метода оригинала) следует фиксировать информацию о запросе?

5. Обеспечение прозрачности интеграции. Спроектируйте клиентский код так, чтобы инстанцирование заместителя не влияло на логику взаимодействия с сервисом. Как гарантировать, что клиентская часть работает исключительно через абстракцию *Subject* и «не замечает» наличия суррогата?

6. Визуализация контролирующих связей в UML. Какие ключевые элементы должны присутствовать на диаграмме классов для паттерна *Proxy*? Опишите отношение реализации общего интерфейса обоими классами и специфику отношения направленной ассоциации (или композиции), исходящей от заместителя к реальному объекту.

5 ПАТТЕРНЫ ПОВЕДЕНИЯ

Паттерны поведения связаны с алгоритмами и распределением обязанностей между объектами. Речь в них идет не только о самих объектах и классах, но и о типичных схемах взаимодействия между ними. Паттерны поведения характеризуют сложный поток управления, который трудно проследить во время выполнения программы. Внимание акцентируется не на схеме управления как таковой, а на связях между объектами.

В паттернах поведения уровня объектов используется не наследование, а композиция. Некоторые из них описывают, как с помощью кооперации множество равноправных объектов справляется с задачей, которая ни одному из них не под силу. Важно здесь то, как объекты получают информацию о существовании друг друга. Одноранговые объекты могут хранить ссылки друг на друга, но это увеличит степень связанности системы. При максимальной степени связанности каждому объекту пришлось бы иметь информацию обо всех остальных.

Их цель – упрощение и улучшение взаимодействия между объектами в системе. Эти паттерны помогают организовать эффективное, гибкое и устойчивое взаимодействие объектов, определяя способы передачи сообщений, управления состояниями и выполнения операций.

Преимущества поведенческих паттернов:

1. Упрощают управление сложными объектами, их состояниями и взаимосвязями.

2. Уменьшают зависимости между классами, повышая модульность системы.

3. Легко добавлять новые поведения без изменения существующего кода.

4. Поведенческие аспекты системы становятся более понятными благодаря унифицированным подходам.

Основные сферы применения:

1. Когда объекты в системе должны взаимодействовать сложным образом, но при этом быть слабо связанными.

2. Когда поведение объектов может меняться в зависимости от их состояния или внешних условий.

3. Когда необходимо определить стандартный процесс выполнения операций, который может изменяться в деталях.

Таким образом, поведенческие паттерны помогают организовать взаимодействие объектов, делая систему более гибкой, понятной и простой для модификации.

5.1 Паттерн Chain of Responsibility (Цепочка обязанностей)

5.1.1 Описание паттерна Chain of Responsibility (Цепочка обязанностей)

Это поведенческий паттерн проектирования, который позволяет передавать запросы последовательно по цепочке обработчиков, пока один из них не выполнит запрос или цепочка не закончится. Паттерн Цепочка обязанностей помогает организовать динамическую последовательность обработки запросов, избегая жёсткой привязки отправителя к конкретным обработчикам. Каждый обработчик в цепочке решает, сможет ли он обработать запрос, либо передаёт его дальше.

Основные условия для применения паттерна цепочка обязанностей:

1. Запрос может быть обработан более чем одним объектом, причем настоящий обработчик заранее неизвестен и должен быть найден автоматически.

2. Запрос должен быть отправлен одному из нескольких объектов, без явного указания, какому именно.

3. Набор объектов, способных обработать запрос, должен задаваться динамически.

Основные элементы паттерна:

1. Запрос (*Request*) – объект, содержащий информацию для обработки (например, данные о событии или запросе пользователя).

2. Базовый обработчик (*Handler*) – абстрактный класс или интерфейс, задающий общий метод для обработки запросов и хранения ссылки на следующий обработчик в цепочке.

3. Конкретные обработчики (*ConcreteHandlers*) – классы, которые реализуют обработку запроса. Каждый обработчик может либо обработать запрос самостоятельно, либо передать его следующему обработчику в цепочке.

4. Клиент (*Client*) – инициатор, который отправляет запрос первому обработчику в цепочке.

Основные отношения между сущностями в UML-диаграмме классов для данного паттерна, следующие: ассоциация – каждый обработчик содержит ссылку на следующий обработчик в цепочке; реализация – конкретные обработчики реализуют общий интерфейс (*Handler*).

Результаты паттерна:

1. Ослабление связанности. Паттерн освобождает объект от необходимости знать, кто конкретно обработает его запрос. Отправитель и получатель ничего не знают друг о друге, а включенный в цепочку объект – о структуре цепочки.

В результате цепочка обязанностей помогает упростить взаимосвязи между объектами. Вместо хранения ссылок на все объекты, которые могут стать получателями запроса, объект должен располагать информацией лишь о своем ближайшем преемнике.

2. Дополнительная гибкость при распределении обязанностей между объектами. Цепочка обязанностей позволяет повысить гибкость распределения обязанностей между объектами. Добавить или изменить обязанности по обработке запроса можно, включив в цепочку новых участников или изменив ее каким-то другим образом. Этот подход можно сочетать со статическим порождением подклассов для создания специализированных обработчиков.

3. Получение не гарантировано. Поскольку у запроса нет явного получателя, то нет и гарантий, что он вообще будет обработан: он может достичь конца цепочки и пропасть. Необработанным запрос может оказаться и в случае неправильной конфигурации цепочки.

Недостатки паттерна, когда запрос передается по цепочке, сложно отследить, кто именно обработал запрос. Если цепочка слишком длинная, это может увеличить время обработки. Клиент не знает, кто именно обработал запрос, и что произошло, если запрос остался необработанным.

Пример из реальной жизни – система технической поддержки. Пользователь отправляет запрос, который сначала обрабатывается операторами начального уровня. Если они не могут решить проблему, запрос передаётся дальше – специалисту средней квалификации, затем старшему инженеру. Здесь каждый уровень технической поддержки – это обработчик в цепочке, а запрос пользователя – это объект, передаваемый по цепочке.

5.1.2 Пример на Python

Есть система, которая применяет скидки в зависимости от уровня клиента: новый клиент, постоянный клиент или VIP-клиент. Каждый тип клиента имеет свой обработчик скидки.

```
class DiscountHandler:
    def __init__(self, next_handler=None):
        self.next_handler = next_handler
    def handle_request(self, amount):
        raise NotImplementedError("Метод handle_request должен быть пере-
определен.")
class NewCustomerDiscount(DiscountHandler):
    def handle_request(self, amount):
        if amount < 100:
            print(f"Новый клиент получает скидку 5% на {amount}.")
        elif self.next_handler:
            print("Новый клиент не подходит для скидки, передаем запрос
дальше.")
            self.next_handler.handle_request(amount)
class RegularCustomerDiscount(DiscountHandler):
    def handle_request(self, amount):
        if amount >= 100 and amount < 500:
            print(f"Постоянный клиент получает скидку 10% на {amount}.")
        elif self.next_handler:
            print("Постоянный клиент не подходит для скидки, передаем за-
прос дальше.")
            self.next_handler.handle_request(amount)
class VIPCustomerDiscount(DiscountHandler):
    def handle_request(self, amount):
        if amount >= 500:
            print(f"VIP клиент получает скидку 20% на {amount}.")
        elif self.next_handler:
            print("VIP клиент не подходит для скидки, передаем запрос
дальше.")
            self.next_handler.handle_request(amount)
vip_discount = VIPCustomerDiscount()
regular_discount = RegularCustomerDiscount(next_handler=vip_discount)
new_customer_discount = NewCustomerDiscount(next_handler=regular_discount)
amount = 250
print(f"Запрос на скидку для суммы {amount}:")
new_customer_discount.handle_request(amount)
```

Пояснение:

– *DiscountHandler* – абстрактный класс, который объявляет метод *handle_request*(), который должен быть реализован в конкретных классах обработчиков. Также в нем есть поле *next_handler*, которое указывает на следующий обработчик в цепочке;

– *NewCustomerDiscount*, *RegularCustomerDiscount* и *VIPCustomerDiscount* – это конкретные классы обработчиков, каждый из которых реализует логику скидки для своего типа клиента. *NewCustomerDiscount* – обрабатывает скидки для новых клиентов, если сумма покупки меньше 100. *RegularCustomerDiscount* – обрабатывает скидки для постоянных клиентов, если сумма покупки от 100 до 500. *VIPCustomerDiscount* – обрабатывает скидки для VIP-клиентов, если сумма покупки больше или равна 500.

Каждый обработчик в цепочке проверяет, может ли он обработать запрос. Если да – он применяет скидку, если нет – передает запрос следующему обработчику.

5.1.3 Пример на Java

Аналогичный пример реализации паттерна Цепочка обязанностей на языке Java.

```
abstract class DiscountHandler {
    protected DiscountHandler nextHandler;
    public DiscountHandler(DiscountHandler nextHandler) {
        this.nextHandler = nextHandler;
    }
    public abstract void handleRequest(double amount);
}
class NewCustomerDiscount extends DiscountHandler {
    public NewCustomerDiscount(DiscountHandler nextHandler) {
        super(nextHandler);
    }
    @Override
    public void handleRequest(double amount) {
        if (amount < 100) {
            System.out.println("Новый клиент получает скидку 5% на " +
amount);
        } else if (nextHandler != null) {
            System.out.println("Новый клиент не подходит для скидки, пере-
даем запрос дальше.");
            nextHandler.handleRequest(amount);
        }
    }
}
class RegularCustomerDiscount extends DiscountHandler {
    public RegularCustomerDiscount(DiscountHandler nextHandler) {
        super(nextHandler);
    }
    @Override
    public void handleRequest(double amount) {
        if (amount >= 100 && amount < 500) {
            System.out.println("Постоянный клиент получает скидку 10% на "
+ amount);
        } else if (nextHandler != null) {
            System.out.println("Постоянный клиент не подходит для скидки,
передаем запрос дальше.");
            nextHandler.handleRequest(amount);
        }
    }
}
class VIPCustomerDiscount extends DiscountHandler {
    public VIPCustomerDiscount(DiscountHandler nextHandler) {
        super(nextHandler);
    }
    @Override
    public void handleRequest(double amount) {
        if (amount >= 500) {
            System.out.println("VIP клиент получает скидку 20% на " +
amount);
        } else if (nextHandler != null) {
            System.out.println("VIP клиент не подходит для скидки, пере-
даем запрос дальше.");
            nextHandler.handleRequest(amount);
        }
    }
```

```
}
public class ChainOfResponsibilityExample {
    public static void main(String[] args) {
        DiscountHandler vipDiscount = new VIPCustomerDiscount(null);
        DiscountHandler regularDiscount = new RegularCustomerDis-
count(vipDiscount);
        DiscountHandler newCustomerDiscount = new NewCustomerDiscount(reg-
ularDiscount);
        double amount = 250;
        System.out.println("Запрос на скидку для суммы " + amount + ":");
        newCustomerDiscount.handleRequest(amount);
    }
}
```

5.2 Паттерн Command (Команда)

5.2.1 Описание паттерна Command (Команда)

Паттерн проектирования, который превращает запросы в объекты, позволяя параметризовать объекты по действиям и поддерживать очереди запросов. Паттерн Команда позволяет инкапсулировать все детали запроса в объекте, а затем передавать этот объект для выполнения. Такой подход позволяет изолировать инициатора запроса от его конкретной реализации и позволяет легко добавлять новые команды, не изменяя существующий код.

Основные условия для применения паттерна Команда:

1. Необходимость инкапсулировать запросы как объекты, чтобы передавать их, хранить и выполнять позже.

2. Требуется поддержка отмены, повторного выполнения или отсрочки запросов.

3. Необходимо создание параметризуемых объектов команд для гибкой обработки различных типов запросов.

4. Потребность в разделении ответственности между отправителями и получателями запросов.

Основные элементы паттерна:

1. Команда (*Command*) – абстрактный интерфейс или класс, который определяет метод для выполнения действия, обычно *execute*(). Команда инкапсулирует запрос и параметры, необходимые для его выполнения.

2. Конкретная команда (*ConcreteCommand*) – класс, реализующий интерфейс команды и содержащий ссылку на объект, который выполняет действие. Каждая конкретная команда определяет, какое действие нужно выполнить на получателе.

3. Получатель (*Receiver*) – объект, на котором выполняется действие. Он содержит всю логику, необходимую для выполнения запроса.

4. *Invoker* – объект, который инициирует выполнение команды, но не знает, как именно она будет выполнена. Он передает команды на выполнение в зависимости от ситуации.

5. Клиент (*Client*) – создает объекты команд и настраивает их параметры.

Основные отношения между сущностями в UML-диаграмме классов для данного паттерна, следующие: ассоциация – каждый обработчик содержит ссылку на следующий обработчик в цепочке; реализация – конкретные обработчики реализуют общий интерфейс (*Handler*).

Основные результаты применения паттерна команда:

1. Инкапсуляция запроса. Все параметры запроса (например, команда и её данные) заключаются в объект, что упрощает их передачу, выполнение и отложенное выполнение.

2. Гибкость в управлении запросами. Легко добавить новые команды, не изменяя существующий код.

3. Поддержка отмены и повторного выполнения. Можно реализовать операции отмены и повторного выполнения, поскольку команда хранит все данные, необходимые для выполнения действия.

4. Разделение обязанностей. Инициатор запроса (*Invoker*) отделён от получателя, что снижает связанность системы и позволяет менять логику обработки без изменения её вызова.

Недостатки паттерна, когда команд слишком много, код может стать тяжёлым для поддержки и понимания. Для реализации отмены команд нужно хранить историю выполненных действий, что может увеличить сложность. В случае повторного выполнения команды может потребоваться дополнительная логика для предотвращения нежелательных побочных эффектов.

Пример из реальной жизни – пульт управления для включения различных приборов в доме. Когда происходит нажатие на кнопку на пульте, это инициирует команду, которая отправляется в соответствующее устройство (например, лампу или вентилятор). Пульт – это инвокер, команда – это нажатая кнопка, а устройства – получатели, которые выполняют действие. При этом пульт не знает, как именно работает устройство, он только отправляет команду на выполнение.

5.2.2 Пример на Python

Есть умный дом, который управляет светом. Использование паттерна *Command*, чтобы включать/выключать свет.

```
class Command:
    def execute(self):
        pass
    def undo(self):
        pass
class Light:
    def turn_on(self):
        print("Свет включен!")
    def turn_off(self):
        print("Свет выключен!")
class TurnOnLightCommand(Command):
    def __init__(self, light: Light):
        self.light = light
    def execute(self):
        self.light.turn_on()
    def undo(self):
        self.light.turn_off()
class TurnOffLightCommand(Command):
    def __init__(self, light: Light):
        self.light = light
    def execute(self):
        self.light.turn_off()
    def undo(self):
        self.light.turn_on()
class RemoteControl:
    def __init__(self):
        self.history = []
```

```
    def press_button(self, command: Command):
        command.execute()
        self.history.append(command)
    def press_undo(self):
        if self.history:
            last_command = self.history.pop()
            last_command.undo()
        else:
            print("Нечего отменять!")
if __name__ == "__main__":
    light = Light()
    turn_on = TurnOnLightCommand(light)
    turn_off = TurnOffLightCommand(light)
    remote = RemoteControl()
    remote.press_button(turn_on)
    remote.press_button(turn_off)
    remote.press_undo()
    remote.press_undo()
```

Пояснение:

– Класс команды – это интерфейс с методами *execute* (выполнить) и *undo* (отменить). *TurnOnLightCommand* и *TurnOffLightCommand*, которые управляют светом;

– Класс приемник – это объект, который знает, как выполнить запрос. Есть класс *Light*, который реализует логику включения/выключения света;

– Класс инвокер – это объект, который вызывает команды. *RemoteControl*, чтобы управлять выполнением и отменой команд.

5.2.3 Пример на Java

Пример реализации паттерна *Command* на Java, аналогичный приведенному выше коду на Python.

```
interface Command {
    void execute();
    void undo();
}
class Light {
    public void turnOn() {
        System.out.println("Свет включен!");
    }
    public void turnOff() {
        System.out.println("Свет выключен!");
    }
}
class TurnOnLightCommand implements Command {
    private Light light;
    public TurnOnLightCommand(Light light) {
        this.light = light;
    }
    @Override
    public void execute() {
        light.turnOn();
    }
    @Override
    public void undo() {
        light.turnOff();
```

```
    }
}
class TurnOffLightCommand implements Command {
    private Light light;
    public TurnOffLightCommand(Light light) {
        this.light = light;
    }
    @Override
    public void execute() {
        light.turnOff();
    }
    @Override
    public void undo() {
        light.turnOn();
    }
}
// Создание класса инвокера (Invoker)
class RemoteControl {
    private Command lastCommand;
    public void pressButton(Command command) {
        command.execute();
        lastCommand = command;
    }
    public void pressUndo() {
        if (lastCommand != null) {
            lastCommand.undo();
        } else {
            System.out.println("Нечего отменять!");
        }
    }
}
public class CommandPatternExample {
    public static void main(String[] args) {
        Light light = new Light();
        Command turnOn = new TurnOnLightCommand(light);
        Command turnOff = new TurnOffLightCommand(light);
        RemoteControl remote = new RemoteControl();
        remote.pressButton(turnOn);
        remote.pressButton(turnOff);
        remote.pressUndo();
        remote.pressUndo();
    }
}
```

5.3 Паттерн Interpreter (Интерпретатор)

5.3.1 Описание паттерна Interpreter (Интерпретатор)

Это поведенческий паттерн проектирования, который определяет представление грамматики для заданного языка и интерпретатор для анализа и выполнения выражений в этом языке. Паттерн Интерпретатор позволяет создать простую систему обработки заданных выражений, где каждое правило грамматики представлено отдельным классом.

Основные условия для применения паттерна Интерпретатор:

Используйте паттерн Интерпретатор в ситуациях, когда имеется интерпретируемый язык, конструкции которого можно представить в виде абстрактных синтаксических деревьев. Этот паттерн лучше всего работает в следующих случаях:

1. Грамматика проста. Для сложных грамматик иерархия классов становится слишком громоздкой и неуправляемой. В таких случаях лучше применять парсеры-генераторы, поскольку они могут интерпретировать выражения без построения абстрактных синтаксических деревьев, что экономит память (а возможно, и время).

2. Эффективность не является главным критерием. Наиболее эффективные интерпретаторы обычно не работают непосредственно с деревьями, а сначала транслируют их в другую форму. Так, регулярное выражение часто преобразуется в конечный автомат. Но даже в этом случае сам транслятор можно реализовать с помощью паттерна интерпретатор.

Основные элементы паттерна:

1. Контекст (*Context*) – объект, содержащий глобальные данные, необходимые для интерпретации (например, переменные или функции).

2. Абстрактное выражение (*AbstractExpression*) – интерфейс или базовый класс, определяющий метод *interpret*(*Context context*), который должен быть реализован конкретными выражениями.

3. Конкретные выражения (*TerminalExpression*, *NonTerminalExpression*):

– терминальное выражение (*TerminalExpression*) – выражение, представляющее конкретный элемент языка (например, числа, переменные);

– нетерминальное выражение (*NonTerminalExpression*) – выражение, которое состоит из других выражений и описывает правила грамматики (например, операции сложения или умножения).

4. Клиент (*Client*) – строит синтаксическое дерево из выражений и передаёт его для интерпретации.

Основные отношения между сущностями в UML-диаграмме классов для данного паттерна, следующие: ассоциация – объект инициатора (*Invoker*) содержит команду; реализация – команды реализуют общий интерфейс (*Command*), а конкретные команды (*ConcreteCommand*) обращаются к получателям (*Receiver*).

Результаты паттерна:

1. Упрощение обработки языка. Каждое правило грамматики представлено отдельным классом, что упрощает добавление новых правил.

2. Расширяемость. Легко добавить поддержку новых конструкций языка путём создания дополнительных классов.

3. Поддержка сложных выражений. Синтаксическое дерево позволяет обрабатывать сложные вложенные выражения рекурсивно.

Недостатки паттерна, когда язык становится сложным, количество классов может значительно увеличиться, что делает систему трудно поддерживаемой. Из-за рекурсивной природы синтаксического дерева интерпретация может быть медленной для сложных выражений. Паттерн подходит для простых языков, но для сложных грамматик обычно применяются другие подходы, такие как генераторы парсеров.

Пример из реальной жизни: создаётся система фильтрации электронных писем. Создаются правила на простом языке: «Если отправитель – boss и тема содержит ‘важно’, то переме-

стить в папку ‘Работа’». Здесь каждое условие – терминальное выражение, а их сочетание с операторами «и»/«или» – нетерминальное выражение.

5.3.2 Пример на Python

Создание простого интерпретатора для выполнения математических операций: сложение и умножение.

```
class Expression(ABC):
    @abstractmethod
    def interpret(self, context):
        pass
class Number(Expression):
    def __init__(self, value):
        self.value = value
    def interpret(self, context):
        return self.value
class Add(Expression):
    def __init__(self, left, right):
        self.left = left
        self.right = right
    def interpret(self, context):
        return self.left.interpret(context) + self.right.interpret(con-
text)
class Multiply(Expression):
    def __init__(self, left, right):
        self.left = left
        self.right = right
    def interpret(self, context):
        return self.left.interpret(context) * self.right.interpret(con-
text)
if __name__ == "__main__":
    addition = Add(Number(5), Number(3))
    multiplication = Multiply(addition, Number(2))
    result = multiplication.interpret(None)
    print(f"Результат: {result}")
```

Пояснение:

– абстрактный класс *Expression* – определяет общий интерфейс для всех выражений. В данном случае метод *interpret*;

– класс *Number* – представляет числовое значение (терминальное выражение);

– класс *Add* – представляет операцию сложения. У него есть два подвыражения: *left* и *right*;

– класс *Multiply* – представляет операцию умножения. Также работает с двумя подвыражениями.

5.3.3 Пример на Java

Аналогичный пример реализации паттерна *Interpreter* на Java.

```
interface Expression {
    int interpret();
}
class Number implements Expression {
    private final int value;
```

```
    public Number(int value) {
        this.value = value;
    }
    @Override
    public int interpret() {
        return value;
    }
}
class Add implements Expression {
    private final Expression leftExpression;
    private final Expression rightExpression;
    public Add(Expression leftExpression, Expression rightExpression) {
        this.leftExpression = leftExpression;
        this.rightExpression = rightExpression;
    }
    @Override
    public int interpret() {
        return leftExpression.interpret() + rightExpression.interpret();
    }
}
class Multiply implements Expression {
    private final Expression leftExpression;
    private final Expression rightExpression;
    public Multiply(Expression leftExpression, Expression rightExpression)
{
        this.leftExpression = leftExpression;
        this.rightExpression = rightExpression;
    }
    @Override
    public int interpret() {
        return leftExpression.interpret() * rightExpression.interpret();
    }
}
public class InterpreterPatternExample {
    public static void main(String[] args) {
        Expression addition = new Add(new Number(5), new Number(3));
        Expression multiplication = new Multiply(addition, new Number(2));
        int result = multiplication.interpret();
        System.out.println("Результат: " + result);
    }
}
```

5.4 Паттерн Iterator (Итератор)

5.4.1 Описание паттерна Iterator (Итератор)

Предоставляет способ последовательного доступа ко всем элементам коллекции (структура данных, которая хранит набор элементов и предоставляет способ получить итератор для последовательного доступа к этим элементам), не раскрывая её внутреннего устройства. Паттерн Итератор позволяет обойти коллекцию различными способами, инкапсулируя логику обхода в отдельный объект.

Основные условия для применения паттерна Итератор:

1. Необходимо предоставить универсальный интерфейс для обхода коллекций, независимо от их реализации.

2. Нужно реализовать разные способы обхода элементов одной коллекции (например, прямой или обратный обход).

3. Требуется сократить зависимость между коллекцией и кодом, использующим её элементы.

Основные элементы паттерна:

1. Итератор (*Iterator*) – общий интерфейс, задающий методы для обхода коллекции.

2. Конкретный итератор (*ConcreteIterator*) – класс, реализующий интерфейс Итератора. Хранит текущее состояние обхода (например, текущий индекс) и предоставляет логику для перехода к следующему элементу.

3. Коллекция (*Aggregate*) – интерфейс или абстрактный класс, определяющий метод для создания итераторов.

4. Конкретная коллекция (*ConcreteAggregate*) – класс, реализующий интерфейс коллекции и предоставляющий метод для создания конкретного итератора.

Основные отношения между сущностями в UML-диаграмме классов для данного паттерна, следующие: ассоциация – итератор взаимодействует с коллекцией; реализация – конкретные итераторы (*ConcreteIterator*) реализуют общий интерфейс (*Iterator*).

Результаты паттерна:

1. Поддержка разных способов обхода коллекции. Сложные коллекции можно обходить по-разному. Например, для генерации кода и семантических проверок нужно обходить деревья синтаксического разбора. Генератор кода может обходить дерево во внутреннем или прямом порядке. Итераторы упрощают изменение алгоритма обхода – достаточно просто заменить один экземпляр итератора другим. Для поддержки новых видов обхода можно определить и подклассы класса *Iterator*.

2. Упрощение интерфейса класса *Aggregate*. Наличие интерфейса для обхода в классе *Iterator* делает излишним дублирование этого интерфейса в классе *Aggregate*. Тем самым интерфейс коллекции упрощается.

3. Возможность наличия нескольких активных обходов для данной коллекции. Итератор следит за инкапсулированным в нем самом состоянием обхода, поэтому одновременно могут существовать несколько обходов коллекции.

Недостатки паттерна, когда структура коллекции изменяется во время обхода, итератор может работать некорректно. Для каждой коллекции может потребоваться собственный итератор, что увеличивает число классов в системе.

Пример из реальной жизни: есть книга с оглавлением. Необходимо просмотреть главы по порядку или в обратном порядке. Здесь: Книга – это коллекция, Глава – элемент коллекции и Указатель на текущую главу – итератор.

5.4.2 Пример на Python

Создание итератор для перебора списка книг.

```
class Book:
    def __init__(self, title: str):
        self.title = title
    def __repr__(self):
        return f"Book('{self.title}')"
class BookCollection:
    def __init__(self):
```

```
        self._books: List[Book] = []
    def add_book(self, book: Book):
        self._books.append(book)
    def __iter__(self) -> Iterator[Book]:
        return BookIterator(self._books)
class BookIterator:
    def __init__(self, books: List[Book]):
        self._books = books
        self._index = 0
    def __iter__(self):
        return self
    def __next__(self):
        if self._index < len(self._books):
            book = self._books[self._index]
            self._index += 1
            return book
        else:
            raise StopIteration
library = BookCollection()
library.add_book(Book("Harry Potter"))
library.add_book(Book("The Hobbit"))
library.add_book(Book("1984"))
print("Список книг в библиотеке:")
for book in library:
    print(book)
```

Пояснение:

– класс *Book* описывает объект книги с атрибутом *title*. Реализует метод *__repr__*, чтобы удобно отображать объект;

– класс *BookCollection* – это коллекция книг, реализованная как список *_books*. Метод *add_book* добавляет книги в коллекцию. Метод *__iter__* возвращает экземпляр *BookIterator*;

– класс *BookIterator* получает список книг и отслеживает текущую позицию с помощью *_index*. Метод *__next__* – возвращает текущий элемент и увеличивает индекс. Если элементов больше нет, вызывает исключение *StopIteration*.

5.4.3 Пример на Java

Аналогичный пример на языке Java, который будет демонстрировать паттерн Итератор для коллекции книг.

```
class Book {
    private String title;
    public Book(String title) {
        this.title = title;
    }
    public String getTitle() {
        return title;
    }
    @Override
    public String toString() {
        return "Book('" + title + "')";
    }
}
interface BookCollection {
    void addBook(Book book);
    Iterator<Book> createIterator();
```

```
}
class Library implements BookCollection {
    private List<Book> books;
    public Library() {
        this.books = new ArrayList<>();
    }
    @Override
    public void addBook(Book book) {
        books.add(book);
    }
    @Override
    public Iterator<Book> createIterator() {
        return new BookIterator(books);
    }
}
class BookIterator implements Iterator<Book> {
    private List<Book> books;
    private int currentIndex;
    public BookIterator(List<Book> books) {
        this.books = books;
        this.currentIndex = 0;
    }
    @Override
    public boolean hasNext() {
        return currentIndex < books.size();
    }
    @Override
    public Book next() {
        if (hasNext()) {
            return books.get(currentIndex++);
        }
        throw new IndexOutOfBoundsException("No more books in the collec-
tion.");
    }
}
public class IteratorPatternExample {
    public static void main(String[] args) {
        Library library = new Library();
        library.addBook(new Book("Harry Potter"));
        library.addBook(new Book("The Hobbit"));
        library.addBook(new Book("1984"));
        Iterator<Book> iterator = library.createIterator();
        System.out.println("Список книг в библиотеке:");
        while (iterator.hasNext()) {
            Book book = iterator.next();
            System.out.println(book);
        }
    }
}
```

5.5 Паттерн Mediator (Посредник)

5.5.1 Описание паттерна Mediator (Посредник)

Паттерн Посредник используется для упрощения взаимодействия между объектами, которые должны взаимодействовать друг с другом, устраняя их прямую зависимость. Этот паттерн

определяет объект, который инкапсулирует способ взаимодействия множества объектов. Благодаря этому, объекты взаимодействуют опосредованно через посредника, а не напрямую, что снижает связанность и упрощает сопровождение кода.

Основные условия для применения паттерна Посредник:

1. Необходимо минимизировать количество прямых связей между объектами, чтобы упростить их взаимодействие.

2. Повторное использование объекта затруднено, поскольку он обменивается информацией со многими другими объектами.

3. Поведение, распределенное между несколькими классами, должно настраиваться без порождения множества подклассов.

Основные элементы паттерна:

1. Посредник (*Mediator*) – интерфейс или абстрактный класс, определяющий методы взаимодействия для связанных объектов.

2. Конкретный посредник (*ConcreteMediator*) – реализация интерфейса посредника, инкапсулирующая логику взаимодействия между объектами. Он хранит ссылки на взаимодействующие объекты и управляет их коммуникацией.

3. Компоненты (*Colleague*) – объекты, взаимодействующие друг с другом через посредника. Они не знают о существовании друг друга и обращаются к посреднику для выполнения необходимых операций.

Основные отношения между сущностями в UML-диаграмме классов для данного паттерна, следующие: ассоциация – посредник (*Mediator*) управляет взаимодействием между коллегами (*Colleague*); реализация – конкретные посредники (*ConcreteMediator*) реализуют общий интерфейс.

Результаты применения паттерна:

1. Снижение связности между объектами. Компоненты взаимодействуют через посредника, а не друг с другом напрямую, что уменьшает зависимости и упрощает модификацию или замену компонентов.

2. Централизация управления. Посредник содержит всю логику взаимодействия между компонентами, что позволяет легко изменять её в одном месте.

3. Улучшение читаемости и поддерживаемости кода. Убирается сложная сеть взаимных зависимостей, что делает систему более предсказуемой.

Недостатки паттерна, когда в сложных системах посредник может стать перегруженным логикой взаимодействий, превращаясь в «божественный объект». Все компоненты должны быть связаны с конкретным посредником, что может усложнить замену его реализации.

Пример из реальной жизни: аэропорт, где диспетчер управляет взлётами и посадками самолётов. Диспетчер – посредник. Самолёты – компоненты, которые не взаимодействуют друг с другом напрямую. Диспетчер координирует их действия, решая, когда и какой самолёт может взлетать или садиться, избегая столкновений.

5.5.2 Пример на Python

Необходимо разработать систему обмена сообщениями, где несколько участников могут общаться друг с другом через общего посредника (например, чат-комнату). Взаимодействие между участниками должно осуществляться через центральный объект-посредник, который отвечает за передачу сообщений.

```
class Mediator(ABC):
    @abstractmethod
    def send_message(self, message, sender):
        pass
class ChatRoom(Mediator):
    def __init__(self):
        self.participants = []
    def register_participant(self, participant):
        if participant not in self.participants:
            self.participants.append(participant)
            participant.set_chat_room(self)
    def send_message(self, message, sender):
        for participant in self.participants:
            if participant != sender:
                participant.receive_message(message, sender)
class Participant:
    def __init__(self, name):
        self.name = name
        self.chat_room = None
    def set_chat_room(self, chat_room):
        self.chat_room = chat_room
    def send_message(self, message):
        if self.chat_room:
            print(f"{self.name} отправляет сообщение: {message}")
            self.chat_room.send_message(message, self)
        else:
            print(f"{self.name} не подключён к чат-комнате.")
    def receive_message(self, message, sender):
        print(f"{self.name} получил сообщение от {sender.name}: {mes-
sage}")
if __name__ == "__main__":
    chat_room = ChatRoom()
    alice = Participant("Alice")
    bob = Participant("Bob")
    charlie = Participant("Charlie")
    chat_room.register_participant(alice)
    chat_room.register_participant(bob)
    chat_room.register_participant(charlie)
    alice.send_message("Привет всем!")
    bob.send_message("Привет, Alice!")
    charlie.send_message("Добрый день!")
```

Пояснение:

– посредник – абстрактный класс *Mediator* определяет метод *send_message*, который будет реализован в конкретном посреднике;

– конкретный посредник – класс *ChatRoom* реализует функционал передачи сообщений между участниками. У него есть метод *register_participant* для регистрации новых участников. Метод *send_message* пересылает сообщение всем участникам, кроме отправителя;

– участники – класс *Participant* представляет участников чата. Каждый участник знает только о посреднике (chat_room), через которого отправляет и получает сообщения.

5.5.3 Пример на Java

Создание аналогичной системы чат-комнаты, где участники взаимодействуют через посредника.

```
interface Mediator {
    void sendMessage(String message, Participant sender);
    void registerParticipant(Participant participant);
}
class ChatRoom implements Mediator {
    private List<Participant> participants = new ArrayList<>();
    @Override
    public void registerParticipant(Participant participant) {
        if (!participants.contains(participant)) {
            participants.add(participant);
            participant.setChatRoom(this);
        }
    }
    @Override
    public void sendMessage(String message, Participant sender) {
        for (Participant participant : participants) {
            if (participant != sender) {
                participant.receiveMessage(message, sender);
            }
        }
    }
}
abstract class Participant {
    protected Mediator chatRoom;
    protected String name;
    public Participant(String name) {
        this.name = name;
    }
    public void setChatRoom(Mediator chatRoom) {
        this.chatRoom = chatRoom;
    }
    public abstract void sendMessage(String message);
    public abstract void receiveMessage(String message, Participant
sender);
}
class User extends Participant {
    public User(String name) {
        super(name);
    }
    @Override
    public void sendMessage(String message) {
        if (chatRoom != null) {
            System.out.println(name + " отправляет сообщение: " + mes-
sage);
            chatRoom.sendMessage(message, this);
        } else {
            System.out.println(name + " не подключён к чат-комнате.");
        }
    }
    @Override
    public void receiveMessage(String message, Participant sender) {
        System.out.println(name + " получил сообщение от " + sender.name +
": " + message);
    }
}
public class MediatorExample {
    public static void main(String[] args) {
        ChatRoom chatRoom = new ChatRoom();
```

```
        Participant alice = new User("Alice");
        Participant bob = new User("Bob");
        Participant charlie = new User("Charlie");
        chatRoom.registerParticipant(alice);
        chatRoom.registerParticipant(bob);
        chatRoom.registerParticipant(charlie);
        alice.sendMessage("Привет всем!");
        bob.sendMessage("Привет, Alice!");
        charlie.sendMessage("Добрый день!");
    }
}
```

5.6 Паттерн Memento (Хранитель)

5.6.1 Описание паттерна Memento (Хранитель)

Паттерн Хранитель предоставляет способ сохранения и восстановления состояния объекта без нарушения инкапсуляции. Он используется, когда необходимо откатить объект к предыдущему состоянию, сохраняя при этом скрытыми детали его реализации.

Основные условия для применения паттерна Хранитель:

1. Необходимость сохранения снимка состояния объекта (или его части), чтобы впоследствии объект можно было восстановить в том же состоянии.

2. Прямой интерфейс для получения этого состояния привел бы к раскрытию подробностей реализации и нарушению инкапсуляции объекта.

Основные элементы паттерна:

1. Создатель (*Originator*) – объект, состояние которого нужно сохранять и восстанавливать. Он создаёт объект-хранитель с копией своего состояния и использует его для восстановления.

2. Хранитель (*Memento*) – объект, хранящий моментальный снимок состояния создателя. Он предоставляет методы для сохранения и извлечения состояния, но не раскрывает структуру или детали сохранённых данных.

3. Опекун (*Caretaker*) – объект, управляющий хранителями. Он сохраняет их и передаёт обратно создателю для восстановления состояния. Опекун не имеет доступа к содержимому хранителя.

Основные отношения между сущностями в UML-диаграмме классов для данного паттерна, следующие: ассоциация – опекун (*Caretaker*) хранит снимок; зависимость – снимок (*Memento*) взаимодействует с объектом-источником (*Originator*).

Результаты применения паттерна:

1. Состояние объекта сохраняется и восстанавливается без раскрытия его структуры другим объектам.

2. Паттерн позволяет легко реализовать функции «отмены» и «повтора» (*Undo*/*Redo*) в приложениях.

3. Опекун может решать, сколько состояний хранить, и управлять их жизненным циклом.

Недостатки паттерна, каждое сохранённое состояние требует памяти, что может стать проблемой при большом количестве состояний или при больших объёмах данных. Если состояния сохраняются слишком часто, это может привести к избыточным данным и усложнению управления.

Пример из реальной жизни: текстовый редактор с функцией отмены. Здесь: Документ – это создатель, который сохраняет своё текущее состояние (например, текст и его форматирование) в хранителе. Сохранённое состояние документа – это хранитель, в котором содержится моментальный снимок текста. История изменений – это опекун, который управляет списком хранителей, позволяя пользователю вернуться к любому из предыдущих состояний.

5.6.2 Пример на Python

Требуется разработать текстовый редактор с функцией «Отменить».

```
class TextEditor:
    def __init__(self):
        self._text = ""
    def write(self, text):
        self._text += text
    def show_text(self):
        print(self._text)
    def save(self):
        return Memento(self._text)
    def restore(self, memento):
        self._text = memento.get_state()
class Memento:
    def __init__(self, state):
        self._state = state
    def get_state(self):
        return self._state
class Caretaker:
    def __init__(self):
        self._history = []
    def save_state(self, memento):
        self._history.append(memento)
    def restore_state(self):
        if not self._history:
            print("Нет сохранённого состояния для восстановления.")
            return None
        return self._history.pop()
def main():
    editor = TextEditor()
    caretaker = Caretaker()
    editor.write("Привет, мир! ")
    editor.show_text()
    caretaker.save_state(editor.save())
    editor.write("Как дела?")
    editor.show_text()
    last_saved_state = caretaker.restore_state()
    if last_saved_state:
        editor.restore(last_saved_state)
    editor.show_text()
if __name__ == "__main__":
    main()
```

Пояснение:

– класс *TextEditor* содержит текущий текстовый контент. Включает методы: *write* – добавляет текст к текущему состоянию, *show_text* – отображает текущее состояние текста, *save* – создаёт объект хранитель с текущим состоянием, *restore* – восстанавливает состояние из хранителя;

– класс *Memento* инкапсулирует состояние объекта *TextEditor*. Метод *get_state* возвращает сохранённое состояние;

– класс *Caretaker* управляет сохранёнными объектами хранителя с помощью стека *_history*. Включает методы: *save_state* – добавляет хранитель в историю, *restore_state* – извлекает последний объект хранителя из стека.

5.6.3 Пример на Java

Аналогичный пример реализации паттерна Хранитель на Java. Этот пример также моделирует текстовый редактор с функцией «Отменить».

```
class TextEditor {
    private String text = "";
    public void write(String newText) {
        text += newText;
    }
    public void showText() {
        System.out.println(text);
    }
    public Memento save() {
        return new Memento(text);
    }
    public void restore(Memento memento) {
        this.text = memento.getState();
    }
}
class Memento {
    private final String state;
    public Memento(String state) {
        this.state = state;
    }
    public String getState() {
        return state;
    }
}
class Caretaker {
    private final Stack<Memento> history = new Stack<>();
    public void saveState(Memento memento) {
        history.push(memento);
    }
    public Memento restoreState() {
        if (history.isEmpty()) {
            System.out.println("Нет сохранённых состояний для восстановле-
ния.");
            return null;
        }
        return history.pop();
    }
}
public class Main {
    public static void main(String[] args) {
        TextEditor editor = new TextEditor();
        Caretaker caretaker = new Caretaker();
        editor.write("Привет, мир! ");
        editor.showText();
        caretaker.saveState(editor.save());
```

```
        editor.write("Как дела?");
        editor.showText();
        Memento lastSavedState = caretaker.restoreState();
        if (lastSavedState != null) {
            editor.restore(lastSavedState);
        }
        editor.showText();
    }
}
```

5.7 Паттерн Observer (Наблюдатель)

5.7.1 Описание паттерна Observer (Наблюдатель)

Паттерн Наблюдатель определяет зависимость типа «один ко многим» между объектами, благодаря чему при изменении состояния одного объекта все зависимые от него объекты уведомляются и автоматически обновляются. Этот паттерн используется для реализации механизмов подписки, позволяя объектам реагировать на события.

Основные условия для применения паттерна Наблюдатель:

1. У абстракции есть два аспекта, один из которых зависит от другого. Инкапсуляции этих аспектов в разные объекты позволяют изменять и повторно использовать их независимо.

2. При модификации одного объекта требуется изменить другие, и не известно, сколько именно объектов нужно изменить.

3. Один объект должен оповещать других, не делая предположений об уведомляемых объектах. Другими словами, объекты не должны быть тесно связаны между собой.

Основные элементы паттерна:

1. Субъект (*Subject*) – объект, за состоянием которого наблюдают. Он предоставляет интерфейс для добавления, удаления и уведомления наблюдателей.

2. Наблюдатель (*Observer*) – интерфейс или абстрактный класс, который должны реализовать все наблюдатели. Он определяет метод для получения уведомлений от субъекта.

3. Конкретный субъект (*ConcreteSubject*) – объект, реализующий интерфейс субъекта. Он хранит состояние, за которым следят, и отправляет уведомления наблюдателям при изменении этого состояния.

4. Конкретный наблюдатель (*ConcreteObserver*) – объект, который получает уведомления от субъекта и обновляет своё состояние в соответствии с изменениями.

Основные отношения между сущностями в UML-диаграмме классов для данного паттерна, следующие: ассоциация – субъект (*Subject*) содержит ссылки на наблюдателей (*Observer*); реализация – конкретные наблюдатели (*ConcreteObserver*) реализуют интерфейс наблюдателя.

Результаты применения паттерна:

1. Наблюдатели и субъект связаны через интерфейсы, что позволяет легко добавлять, удалять или изменять наблюдателей без модификации кода субъекта.

2. Объекты могут динамически подписываться на события и отписываться от них.

3. Все наблюдатели автоматически получают уведомления и обновляются при изменении состояния субъекта.

Недостатки паттерна, когда при большом количестве наблюдателей становится сложно отслеживать цепочку уведомлений. Если субъект уведомляет слишком много наблюдателей или

обновления происходят слишком часто, это может снизить производительность системы. Если наблюдатели не отписываются от субъекта, их ссылки могут не освобождаться, что приводит к утечкам памяти.

Пример из реальной жизни: система уведомлений в социальных сетях. Пользователь – это субъект, за действиями которого наблюдают подписчики. Подписчики – наблюдатели, которые получают уведомления о новых публикациях, изменениях профиля и других событиях. При публикации нового поста пользователь уведомляет всех своих подписчиков, и они видят это событие в своих лентах.

5.7.2 Пример на Python

Требуется разработать систему, где есть объект «Метеостанция» (издатель), и различные устройства, такие как «Телефон» и «Экран» (наблюдатели), которые получают уведомления об изменении температуры.

```
class Observer:
    def update(self, temperature):
        raise NotImplementedError("Этот метод должен быть переопределён в
подклассе")
class Subject:
    def __init__(self):
        self._observers = []
    def add_observer(self, observer):
        if observer not in self._observers:
            self._observers.append(observer)
    def remove_observer(self, observer):
        if observer in self._observers:
            self._observers.remove(observer)
    def notify_observers(self, temperature):
        for observer in self._observers:
            observer.update(temperature)
class WeatherStation(Subject):
    def __init__(self):
        super().__init__()
        self._temperature = None
    @property
    def temperature(self):
        return self._temperature
    @temperature.setter
    def temperature(self, value):
        print(f"Метеостанция: обновление температуры на {value}°C")
        self._temperature = value
        self.notify_observers(value)
class PhoneDisplay(Observer):
    def update(self, temperature):
        print(f"Телефон: получено уведомление о температуре:
{temperature}°C")
class ScreenDisplay(Observer):
    def update(self, temperature):
        print(f"Экран: получено уведомление о температуре:
{temperature}°C")
if __name__ == "__main__":
    weather_station = WeatherStation()
    phone_display = PhoneDisplay()
    screen_display = ScreenDisplay()
```

```
    weather_station.add_observer(phone_display)
    weather_station.add_observer(screen_display)
    weather_station.temperature = 25
    weather_station.temperature = 30
    weather_station.remove_observer(phone_display)
    weather_station.temperature = 28
```

Пояснение:

– класс *Observer* – определяет метод *update*, который вызывается для уведомления об изменении состояния;

– класс *Subject* – содержит методы для добавления, удаления и уведомления наблюдателей;

– класс WeatherStation – хранит данные (например, температуру) и управляет уведомлением наблюдателей;

– классы *PhoneDisplay* и *ScreenDisplay* – реализуют метод *update*, чтобы реагировать на изменения состояния издателя.

5.7.3 Пример на Java

Аналогичный пример реализации паттерна на Java. Разработка программы, где «Метеостанция» является издателем, а «Телефон» и «Экран» – это наблюдатели.

```
interface Observer {
    void update(float temperature);
}
interface Subject {
    void addObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers();
}
class WeatherStation implements Subject {
    private float temperature;
    private List<Observer> observers = new ArrayList<>();
    @Override
    public void addObserver(Observer observer) {
        observers.add(observer);
    }
    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }
    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(temperature);
        }
    }
    public void setTemperature(float temperature) {
        System.out.println("Метеостанция: обновление температуры на " +
temperature + "°C");
        this.temperature = temperature;
        notifyObservers();
    }
}
class PhoneDisplay implements Observer {
    @Override
    public void update(float temperature) {
```

```
        System.out.println("Телефон: получено уведомление о температуре: "
+ temperature + "°C");
    }
}
class ScreenDisplay implements Observer {
    @Override
    public void update(float temperature) {
        System.out.println("Экран: получено уведомление о температуре: " +
temperature + "°C");
    }
}
public class ObserverPatternExample {
    public static void main(String[] args) {
        WeatherStation weatherStation = new WeatherStation();
        PhoneDisplay phoneDisplay = new PhoneDisplay();
        ScreenDisplay screenDisplay = new ScreenDisplay();
        weatherStation.addObserver(phoneDisplay);
        weatherStation.addObserver(screenDisplay);
        weatherStation.setTemperature(25);
        weatherStation.setTemperature(30);
        weatherStation.removeObserver(phoneDisplay);
        weatherStation.setTemperature(28);
    }
}
```

5.8 Паттерн State (Состояние)

5.8.1 Описание паттерна State (Состояние)

Паттерн Состояние позволяет объекту изменять свое поведение в зависимости от его состояния. Этот паттерн используется, когда объект должен вести себя по-разному в зависимости от состояния, и он изменяет своё поведение в момент перехода из одного состояния в другое, не изменяя свой интерфейс.

Основные условия для применения паттерна Состояние:

1. Поведение объекта зависит от его состояния и должно изменяться вовремя выполнения.

2. Когда в коде операций встречаются состоящие из многих ветвей условные операторы, в которых выбор ветви зависит от состояния. Обычно в таком случае состояние представлено перечисляемыми константами. Часто одна и та же структура условного оператора повторяется в нескольких операциях. Паттерн Состояние предлагает поместить каждую ветвь в отдельный класс. Это позволяет трактовать состояние объекта как самостоятельный объект, который может изменяться независимо от других.

Основные элементы паттерна:

1. Контекст (*Context*) – объект, который использует состояние для изменения своего поведения. Он хранит ссылку на текущее состояние и делегирует выполнение операций состоянию.

2. Состояние (*State*) – интерфейс или абстрактный класс, который определяет методы для работы с различными состояниями.

3. Конкретные состояния (*ConcreteState*) – подклассы, которые реализуют поведение объекта в различных состояниях. Каждый конкретный подкласс состояния изменяет поведение контекста в зависимости от текущего состояния.

Основные отношения между сущностями в UML-диаграмме классов для данного паттерна, следующие: ассоциация – контекст (*Context*) содержит ссылку на текущее состояние; реализация – конкретные состояния (*ConcreteState*) реализуют интерфейс состояния (*State*).

Результаты применения паттерна:

1. Локализация поведения по состояниям. Все поведение, зависящее от состояния, помещается в отдельные объекты (подклассы *State*). Это упрощает добавление новых состояний, так как достаточно создавать новые подклассы, а не менять код в нескольких местах. Это избегает громоздких условных операторов (*if-else* или *switch*) и улучшает структуру кода.

2. Явно выраженные переходы между состояниями. Переходы между состояниями становятся явными благодаря объектам состояния, что защищает контекст от рассогласования внутренних данных и упрощает управление состоянием (требуется изменение только одной переменной, а не нескольких).

3. Совместное использование объектов состояния. Если объект состояния не содержит собственных полей (атрибутов), то его можно использовать в разных контекстах. Это позволяет разделить одно и то же состояние между несколькими контекстами, экономя ресурсы и улучшая производительность.

Недостатки паттерна, для каждого состояния может потребоваться создание нового класса, что может привести к излишнему количеству классов в системе. Иногда может быть сложно управлять большим количеством состояний, особенно если они имеют сложные переходы или взаимозависимости.

Пример из реальной жизни: есть автомат по продаже напитков, который имеет несколько состояний, таких как «Ожидание», «Выбор напитка», «Покупка» и «Отдача товара». Ожидание – автомат в этом состоянии ожидает, когда пользователь начнёт выбор. Выбор напитка – пользователь выбирает напиток, и автомат обновляет своё поведение в соответствии с этим выбором. Покупка – автомат ожидает, когда пользователь оплатит напиток. Отдача товара – автомат отдает выбранный напиток после успешной оплаты.

Каждое состояние изменяет поведение автомата, при этом все состояния можно реализовать как отдельные классы, инкапсулируя поведение, специфичное для каждого состояния.

5.8.2 Пример на Python

Моделирование взаимодействия с музыкальным плеером. У плеера может быть несколько состояний, таких как «Воспроизведение», «Пауза» и «Остановлено». В зависимости от текущего состояния, плеер будет по-разному реагировать на действия пользователя (например, нажать на кнопку «Play»).

```
class State(ABC):
    @abstractmethod
    def press_play(self, player):
        pass
    @abstractmethod
    def press_pause(self, player):
        pass
    @abstractmethod
    def press_stop(self, player):
        pass
class MusicPlayer:
    def __init__(self):
```

```
        self._state = StoppedState()
    def set_state(self, state):
        self._state = state
    def press_play(self):
        self._state.press_play(self)
    def press_pause(self):
        self._state.press_pause(self)
    def press_stop(self):
        self._state.press_stop(self)
class PlayingState(State):
    def press_play(self, player):
        print("Плеер уже воспроизводит музыку!")
    def press_pause(self, player):
        print("Пауза! Музыка приостановлена.")
        player.set_state(PausedState())
    def press_stop(self, player):
        print("Остановка воспроизведения.")
        player.set_state(StoppedState())
class PausedState(State):
    def press_play(self, player):
        print("Воспроизведение продолжено!")
        player.set_state(PlayingState())
    def press_pause(self, player):
        print("Музыка уже на паузе.")
    def press_stop(self, player):
        print("Остановка воспроизведения.")
        player.set_state(StoppedState())
class StoppedState(State):
    def press_play(self, player):
        print("Начинаем воспроизведение!")
        player.set_state(PlayingState())
    def press_pause(self, player):
        print("Музыка не воспроизводится, ничего не происходит.")
    def press_stop(self, player):
        print("Плеер уже остановлен.")
player = MusicPlayer()
player.press_play()
player.press_pause()
player.press_stop()
```

Пояснение:

– интерфейс состояния – абстрактный класс, который определяет методы, которые будут реализованы для каждого состояния. Методы для нажатия на кнопки: *press_play*, *press_pause* и *press_stop*;

– контекст – класс управляет состоянием плеера. У него есть метод *set_state*, который позволяет менять текущее состояние, и методы *press_play*, *press_pause* и *press_stop*, которые делегируют действия конкретному состоянию;

– конкретные состояния – конкретные реализации состояний плеера. Каждый класс (*PlayingState*, *PausedState*, *StoppedState*) реализует поведение для соответствующего состояния. В *PlayingState* при нажатии на *press_pause* плеер переходит в состояние паузы, а при нажатии на *press_stop* – в состояние остановки. В *PausedState* плеер может продолжить воспроизведение или остановиться. В *StoppedState* плеер может начать воспроизведение, но ничего не происходит, если нажать «Pause», так как плеер не воспроизводит музыку.

5.8.3 Пример на Java

Аналогичный пример паттерна «Состояние» на языке Java. Моделирование музыкального плеера с состояниями: Воспроизведение (*Playing*), Пауза (*Paused*) и Остановлено (*Stopped*).

```
interface State {
    void pressPlay(MusicPlayer player);
    void pressPause(MusicPlayer player);
    void pressStop(MusicPlayer player);
}
class MusicPlayer {
    private State state;
    public MusicPlayer() {
        state = new StoppedState();
    }
    public void setState(State state) {
        this.state = state;
    }
    public void pressPlay() {
        state.pressPlay(this);
    }
    public void pressPause() {
        state.pressPause(this);
    }
    public void pressStop() {
        state.pressStop(this);
    }
}
class PlayingState implements State {
    @Override
    public void pressPlay(MusicPlayer player) {
        System.out.println("Плеер уже воспроизводит музыку!");
    }
    @Override
    public void pressPause(MusicPlayer player) {
        System.out.println("Пауза! Музыка приостановлена.");
        player.setState(new PausedState());
    }
    @Override
    public void pressStop(MusicPlayer player) {
        System.out.println("Остановка воспроизведения.");
        player.setState(new StoppedState());
    }
}
class PausedState implements State {
    @Override
    public void pressPlay(MusicPlayer player) {
        System.out.println("Воспроизведение продолжено!");
        player.setState(new PlayingState());
    }
    @Override
    public void pressPause(MusicPlayer player) {
        System.out.println("Музыка уже на паузе.");
    }
    @Override
    public void pressStop(MusicPlayer player) {
        System.out.println("Остановка воспроизведения.");
        player.setState(new StoppedState());
```

```
    }
}
class StoppedState implements State {
    @Override
    public void pressPlay(MusicPlayer player) {
        System.out.println("Начинаем воспроизведение!");
        player.setState(new PlayingState());
    }
    @Override
    public void pressPause(MusicPlayer player) {
        System.out.println("Музыка не воспроизводится, ничего не происхо-
дит.");
    }
    @Override
    public void pressStop(MusicPlayer player) {
        System.out.println("Плеер уже остановлен.");
    }
}
public class Main {
    public static void main(String[] args) {
        MusicPlayer player = new MusicPlayer();
        player.pressPlay();
        player.pressPause();
        player.pressStop();
    }
}
```

5.9 Паттерн Strategy (Стратегия)

5.9.1 Описание паттерна Strategy (Стратегия)

Это поведенческий паттерн проектирования, который определяет семейство алгоритмов, инкапсулирует их и делает взаимозаменяемыми. Паттерн Стратегия позволяет выбирать алгоритм на этапе выполнения программы, не меняя классы, которые используют эти алгоритмы. Это помогает выделить логику поведения и сделать её независимой от кода, который использует это поведение.

Основные условия для применения паттерна Стратегия:

1. Наличие множества родственных классов, отличающихся только поведением. Стратегия позволяет настроить класс одним из многих возможных вариантов поведения.

2. Наличие нескольких разновидностей алгоритма. Например, можно определить два варианта алгоритма, один из которых требует больше времени, а другой – больше памяти. Стратегии разрешается применять, когда варианты алгоритмов реализованы в виде иерархии классов.

3. В алгоритме содержатся данные, о которых клиент не должен «знать». Можно использовать паттерн, чтобы не раскрывать сложные, специфичные для алгоритма структуры данных.

4. В классе определено много вариантов поведения, представленных разветвленными условными операторами. В этом случае проще перенести код из ветвей в отдельные классы стратегий.

Основные элементы паттерна:

1. Контекст (*Context*) – класс, который использует стратегию. Он хранит ссылку на объект стратегии и вызывает её методы для выполнения операций.

2. Интерфейс стратегии (*Strategy*) – общий интерфейс для всех стратегий, определяющий метод (например, *execute*()), который будет реализован конкретными стратегиями.

3. Конкретные стратегии (*ConcreteStrategy*) – классы, реализующие различные алгоритмы, которые можно подставлять в контекст.

Основные отношения между сущностями в UML-диаграмме классов для данного паттерна, следующие: ассоциация – контекст (*Context*) содержит ссылку на стратегию; реализация – конкретные стратегии (*ConcreteStrategy*) реализуют интерфейс (*Strategy*).

Результаты паттерна:

1. Гибкость. Паттерн позволяет менять поведение объектов во время выполнения путём подстановки разных стратегий.

2. Простое добавление новых алгоритмов. Новую стратегию можно добавить без изменения существующего кода.

3. Изолирование алгоритмов. Код алгоритма отделён от кода, который его использует, что упрощает поддержку и тестирование.

Недостатки паттерна, это усложнение структуры при большом количестве стратегий увеличивается количество классов. Если стратегии сильно похожи, код может быть дублированным, что усложняет поддержку.

Пример из реальной жизни – мобильное приложение для навигации, которое предлагает разные маршруты: кратчайший, быстрый, пеший, велосипедный. Здесь контекст – это модуль приложения, отвечающий за построение маршрута. Каждая стратегия (кратчайший маршрут, пеший маршрут и т.д.) – это алгоритм, который используется в зависимости от выбора пользователя.

5.9.2 Пример на Python

Пример программы, где пользователь выбирает способ оплаты.

```
class PaymentStrategy(ABC):
    @abstractmethod
    def pay(self, amount: float) -> None:
        pass
class CreditCardPayment(PaymentStrategy):
    def __init__(self, card_number: str, card_holder: str):
        self.card_number = card_number
        self.card_holder = card_holder
    def pay(self, amount: float) -> None:
        print(f"Оплачено {amount:.2f} кредитной картой. Владелец:
{self.card_holder}, Номер карты: {self.card_number[-4:]}")
class PayPalPayment(PaymentStrategy):
    def __init__(self, email: str):
        self.email = email
    def pay(self, amount: float) -> None:
        print(f"Оплачено {amount:.2f} через PayPal. Аккаунт:
{self.email}")
class CryptoPayment(PaymentStrategy):
    def __init__(self, wallet_address: str):
        self.wallet_address = wallet_address
    def pay(self, amount: float) -> None:
        print(f"Оплачено {amount:.2f} криптовалютой. Кошелек: {self.wal-
let_address}")
class PaymentContext:
```

```
    def __init__(self, strategy: PaymentStrategy):
        self.strategy = strategy
    def set_strategy(self, strategy: PaymentStrategy):
        self.strategy = strategy
    def execute_payment(self, amount: float) -> None:
        self.strategy.pay(amount)
if __name__ == "__main__":
    print("Выберите способ оплаты:")
    print("1: Кредитная карта")
    print("2: PayPal")
    print("3: Криптовалюта")
    choice = int(input("Введите номер способа оплаты: "))
    if choice == 1:
        card_number = input("Введите номер карты: ")
        card_holder = input("Введите имя владельца карты: ")
        strategy = CreditCardPayment(card_number, card_holder)
    elif choice == 2:
        email = input("Введите ваш email PayPal: ")
        strategy = PayPalPayment(email)
    elif choice == 3:
        wallet_address = input("Введите адрес крипто-кошелька: ")
        strategy = CryptoPayment(wallet_address)
    else:
        print("Неверный выбор!")
        exit()
    context = PaymentContext(strategy)
    amount = float(input("Введите сумму оплаты: "))
    context.execute_payment(amount)
```

Пояснение:

– абстрактный класс *PaymentStrategy* содержит метод pay, который должны реализовать все конкретные стратегии. Используется, чтобы гарантировать единообразие интерфейса для всех стратегий;

– конкретные стратегии (*CreditCardPayment*, *PayPalPayment*, *CryptoPayment*) реализуют метод *pay*, предоставляя специфическую логику для оплаты. Например, *CreditCardPayment* принимает номер карты и имя владельца, а *CryptoPayment* – адрес кошелька;

– контекст PaymentContext использует объект стратегии для выполнения конкретного алгоритма. Позволяет менять стратегию оплаты во время выполнения программы с помощью метода *set_strategy*.

5.9.3 Пример на Java

Реализация той же задачи – пользователь выбирает способ оплаты: кредитной картой, через PayPal или криптовалютой.

```
interface PaymentStrategy {
    void pay(double amount);
}
class CreditCardPayment implements PaymentStrategy {
    private String cardNumber;
    private String cardHolder;
    public CreditCardPayment(String cardNumber, String cardHolder) {
        this.cardNumber = cardNumber;
        this.cardHolder = cardHolder;
    }
```

```
    @Override
    public void pay(double amount) {
        System.out.printf("Оплачено %.2f кредитной картой. Владелец: %s,
Номер карты: ****%s%n",
                          amount, cardHolder, cardNumber.substring(card-
Number.length() - 4));
    }
}
class PayPalPayment implements PaymentStrategy {
    private String email;
    public PayPalPayment(String email) {
        this.email = email;
    }
    @Override
    public void pay(double amount) {
        System.out.printf("Оплачено %.2f через PayPal. Аккаунт: %s%n",
amount, email);
    }
}
class CryptoPayment implements PaymentStrategy {
    private String walletAddress;
    public CryptoPayment(String walletAddress) {
        this.walletAddress = walletAddress;
    }
    @Override
    public void pay(double amount) {
        System.out.printf("Оплачено %.2f криптовалютой. Кошелек: %s%n",
amount, walletAddress);
    }
}
class PaymentContext {
    private PaymentStrategy strategy;
    public PaymentContext(PaymentStrategy strategy) {
        this.strategy = strategy;
    }
    public void setStrategy(PaymentStrategy strategy) {
        this.strategy = strategy;
    }
    public void executePayment(double amount) {
        strategy.pay(amount);
    }
}
public class StrategyExample {
    public static void main(String[] args) {
        java.util.Scanner scanner = new java.util.Scanner(System.in);
        System.out.println("Выберите способ оплаты:");
        System.out.println("1: Кредитная карта");
        System.out.println("2: PayPal");
        System.out.println("3: Криптовалюта");
        int choice = scanner.nextInt();
        scanner.nextLine();
        PaymentStrategy strategy = null;
        switch (choice) {
            case 1 -> {
                System.out.print("Введите номер карты: ");
                String cardNumber = scanner.nextLine();
                System.out.print("Введите имя владельца карты: ");
                String cardHolder = scanner.nextLine();
```

```
                strategy = new CreditCardPayment(cardNumber, cardHolder);
            }
            case 2 -> {
                System.out.print("Введите ваш email PayPal: ");
                String email = scanner.nextLine();
                strategy = new PayPalPayment(email);
            }
            case 3 -> {
                System.out.print("Введите адрес крипто-кошелька: ");
                String walletAddress = scanner.nextLine();
                strategy = new CryptoPayment(walletAddress);
            }
            default -> {
                System.out.println("Неверный выбор!");
                return;
            }
        }
        PaymentContext context = new PaymentContext(strategy);
        System.out.print("Введите сумму оплаты: ");
        double amount = scanner.nextDouble();
        context.executePayment(amount);
    }
}
```

5.10 Паттерн Template Method (Шаблонный метод)

5.10.1 Описание паттерна Template Method (Шаблонный метод)

Паттерн проектирования, который определяет общий алгоритм выполнения задачи, оставляя конкретные шаги реализации подклассам. Паттерн Шаблонный метод позволяет задавать скелет алгоритма в базовом классе, обеспечивая возможность изменять некоторые его шаги в наследниках без изменения общей структуры.

Основные условия для применения паттерна Шаблонный метод:

1. Необходимо определить общий алгоритм выполнения задачи, при этом позволяя изменять или уточнять отдельные шаги.

2. Если необходимо избежать дублирования кода в схожих классах, выделив общие шаги в базовый класс.

3. Требуется обеспечить последовательность выполнения шагов алгоритма.

Основные элементы паттерна:

1. Базовый класс (*AbstractClass*) – содержит реализацию шаблонного метода, который задаёт последовательность выполнения шагов алгоритма. Некоторые шаги могут быть реализованы в базовом классе, а некоторые оставлены абстрактными или виртуальными для переопределения.

2. Шаблонный метод (*TemplateMethod*) – метод, описывающий общую структуру алгоритма. Может быть объявлен как финальный, чтобы запретить изменение структуры алгоритма в подклассах.

3. Конкретные шаги (*ConcreteSteps*) – методы, реализованные либо в базовом классе, либо в подклассах. Они отвечают за выполнение специфичных частей алгоритма.

4. Подклассы (*ConcreteClass*) – конкретные классы, которые переопределяют шаги алгоритма, обеспечивая их уникальную реализацию.

Основные отношения между сущностями в UML-диаграмме классов для данного паттерна, следующие: наследование – подклассы переопределяют шаги алгоритма, определенные в базовом классе.

Результаты паттерна:

1. Повторное использование кода. Общая структура алгоритма определяется один раз в базовом классе, что позволяет избежать дублирования в подклассах.

2. Контроль последовательности. Шаблонный метод строго определяет порядок выполнения шагов, гарантируя их неизменность.

3. Гибкость и расширяемость. Подклассы могут изменять только конкретные шаги алгоритма, не влияя на общую логику.

4. Простота добавления нового поведения. Для изменения поведения нужно только переопределить соответствующие шаги.

Недостатки паттерна, когда если структура алгоритма изменится, потребуется вносить изменения в базовый класс, что может затронуть множество подклассов. Наследование вместо композиции. Использование наследования ограничивает гибкость, так как подклассы жёстко связаны с базовым классом. Риск переопределения шаблонного метода. Если шаблонный метод не защищён от переопределения, это может нарушить общую структуру алгоритма.

Пример из реальной жизни – процесс приготовления кофе и чая. Общие шаги включают: кипячение воды, заваривание напитка, добавление дополнительных ингредиентов (например, сахара или молока). Для чая заваривание – это добавление чайного пакетика, а для кофе – использование кофейного порошка. Кипячение воды и добавление сахара могут быть общими шагами. Шаблонный метод определяет общий процесс приготовления напитка, а конкретные классы (кофе или чай) реализуют специфичные шаги.

5.10.2 Пример на Python

Есть различные способы приготовления кофе и чая. Общий процесс (алгоритм) включает: кипячение воды, заваривание напитка, добавление ингредиентов и наливание напитка в чашку. Но детали (например, что именно заваривать и что добавлять) зависят от конкретного напитка.

```
class Beverage(ABC):
    def prepare(self):
        self.boil_water()
        self.brew()
        self.pour_in_cup()
        self.add_ingredients()
    def boil_water(self):
        print("Кипячение воды...")
    def pour_in_cup(self):
        print("Наливание напитка в чашку...")
    @abstractmethod
    def brew(self):
        pass
    @abstractmethod
    def add_ingredients(self):
        pass
class Tea(Beverage):
    def brew(self):
```

```
        print("Заваривание чайного пакетика...")
    def add_ingredients(self):
        print("Добавление лимона...")
class Coffee(Beverage):
    def brew(self):
        print("Заваривание молотого кофе...")
    def add_ingredients(self):
        print("Добавление сахара и молока...")
if __name__ == "__main__":
    print("Приготовим чай:")
    tea = Tea()
    tea.prepare()
    print("\nПриготовим кофе:")
    coffee = Coffee()
    coffee.prepare()
```

Пояснение:

– *Beverage* – это базовый класс, который определяет шаблонный метод *prepare* и два общих метода *boil_water* и *pour_in_cup*. Он также содержит абстрактные методы *brew* и *add_ingredients*, которые обязаны реализовать подклассы;

– *Tea* и *Coffee* – подклассы, которые реализуют конкретные шаги: как заваривать и что добавлять;

– *Prepare* – это шаблонный метод, который определяет последовательность вызова шагов: сначала кипятим воду, затем заваривание (шаг зависит от подкласса), потом наливание в чашку и в конце добавляются ингредиенты (опять зависит от подкласса).

5.10.3 Пример на Java

Реализация паттерна *Template Method* на Java для той же задачи: приготовления чая и кофе.

```
abstract class Beverage {
    public final void prepare() {
        boilWater();
        brew();
        pourInCup();
        addIngredients();
    }
    private void boilWater() {
        System.out.println("Кипячение воды...");
    }
    private void pourInCup() {
        System.out.println("Наливание напитка в чашку...");
    }
    protected abstract void brew();
    protected abstract void addIngredients();
}
class Tea extends Beverage {
    @Override
    protected void brew() {
        System.out.println("Заваривание чайного пакетика...");
    }
    @Override
    protected void addIngredients() {
        System.out.println("Добавление лимона...");
    }
}
```

```
class Coffee extends Beverage {
    @Override
    protected void brew() {
        System.out.println("Заваривание молотого кофе...");
    }
    @Override
    protected void addIngredients() {
        System.out.println("Добавление сахара и молока...");
    }
}
public class Main {
    public static void main(String[] args) {
        System.out.println("Приготовим чай:");
        Beverage tea = new Tea();
        tea.prepare();
        System.out.println("\nПриготовим кофе:");
        Beverage coffee = new Coffee();
        coffee.prepare();
    }
}
```

5.11 Паттерн Visitor (Посетитель)

5.11.1 Описание паттерна Visitor (Посетитель)

Поведенческий паттерн проектирования, который позволяет добавлять новые операции к существующим объектам, не изменяя их структуры. Паттерн Посетитель отделяет алгоритмы от данных, предоставляя возможность выполнять действия над объектами различных классов без нарушения их инкапсуляции. Это достигается путём создания объекта-«посетителя», который реализует нужные операции и передаётся объектам для выполнения.

Основные условия для применения паттерна Посетитель:

1. В структуре присутствуют объекты многих классов с различными интерфейсами, и вы хотите выполнять над ними операции, зависящие от конкретных классов.

2. Над объектами, входящими в состав структуры, должны выполняться разнообразные, не связанные между собой операции, и вы не хотите «засорять» классы такими операциями. Посетитель позволяет объединить родственные операции, поместив их в один класс. Если структура объектов является общей для нескольких приложений, то паттерн Посетитель позволит в каждое приложение включить только относящиеся к нему операции.

3. Классы, определяющие структуру объектов, изменяются редко, но новые операции над этой структурой добавляются часто. При изменении классов, представленных в структуре, придется переопределить интерфейсы всех посетителей, а это может вызвать затруднения. Поэтому если классы меняются достаточно часто, то, вероятно, лучше определить операции прямо в них.

Основные элементы паттерна:

1. Посетитель (*Visitor*) – интерфейс или абстрактный класс, определяющий набор методов для обработки каждого типа элемента структуры. Например, методы *visitConcreteElementA*() и *visitConcreteElementB*().

2. Конкретный посетитель (*ConcreteVisitor*) – классы, реализующие конкретные операции для обработки элементов. Каждая операция определяется отдельно для каждого типа элемента.

3. Элемент (*Element*) – интерфейс или абстрактный класс, определяющий метод accept(Visitor visitor). Этот метод позволяет объекту принять посетителя.

4. Конкретные элементы (*ConcreteElement*) – классы, представляющие элементы структуры. Они вызывают метод посетителя, соответствующий их типу.

5. Структура объектов (*ObjectStructure*) – коллекция или дерево объектов, которые могут быть обработаны посетителями.

Основные отношения между сущностями в UML-диаграмме классов для данного паттерна, следующие: ассоциация – элементы (*Element*) принимают посетителя (*Visitor*); реализация – конкретные посетители (*ConcreteVisitor*) реализуют общий интерфейс.

Результаты паттерна:

1. Гибкость в добавлении операций. Можно добавлять новые операции, не изменяя классы элементов.

2. Разделение ответственности. Логика операций выносится из элементов в посетителей, что упрощает поддержку и тестирование кода.

3. Удобство работы с различными типами элементов. Посетитель позволяет выполнять специфичные операции для каждого типа элемента, не усложняя их код.

Недостатки паттерна, это сложность добавления новых классов элементов. Если появляются новые типы элементов, нужно модифицировать все существующие посетители, добавляя методы для их обработки. Нарушение инкапсуляции. Посетитель может получать доступ к внутренним данным элемента, что делает его зависимым от их структуры. Увеличение связности. Каждый посетитель должен знать о всех типах элементов.

Пример из реальной жизни – музей, в котором работают гиды (посетители). У музея есть разные экспонаты (элементы): картины, скульптуры и другие произведения искусства. Каждый гид рассказывает о каждом типе экспоната по-своему: один акцентирует внимание на историческом значении, другой – на технической стороне. Экспонаты не знают, как их описывать – эту задачу выполняют гиды.

5.11.2 Пример на Python

Есть древовидная модель файловой системы операционной системы, где каждый узел – это либо каталог (содержащий вложенные элементы), либо конечный файл. Необходимо подсчитать общий объем занимаемой памяти и вывести иерархическую структуру элементов.

```
class Visitor(ABC):
    @abstractmethod
    def visit_file(self, file):
        pass
    @abstractmethod
    def visit_folder(self, folder):
        pass
class Element(ABC):
    @abstractmethod
    def accept(self, visitor: Visitor):
        pass
class File(Element):
    def __init__(self, name: str, size: int):
        self.name = name
        self.size = size
```

```
    def accept(self, visitor: Visitor):
        visitor.visit_file(self)
class Folder(Element):
    def __init__(self, name: str):
        self.name = name
        self.children = []
    def add(self, element: Element):
        self.children.append(element)
    def accept(self, visitor: Visitor):
        visitor.visit_folder(self)
class SizeCalculator(Visitor):
    def __init__(self):
        self.total_size = 0
    def visit_file(self, file: File):
        print(f"Размер файла '{file.name}': {file.size}")
        self.total_size += file.size
    def visit_folder(self, folder: Folder):
        print(f"Обходим папку '{folder.name}'")
        for child in folder.children:
            child.accept(self)
class StructurePrinter(Visitor):
    def __init__(self):
        self.depth = 0
    def visit_file(self, file: File):
        print("  " * self.depth + f"Файл: {file.name}")
    def visit_folder(self, folder: Folder):
        print("  " * self.depth + f"Папка: {folder.name}")
        self.depth += 1
        for child in folder.children:
            child.accept(self)
        self.depth -= 1
if __name__ == "__main__":
    root = Folder("root")
    root.add(File("file1.txt", 100))
    root.add(File("file2.txt", 200))
    sub_folder = Folder("sub_folder")
    sub_folder.add(File("file3.txt", 300))
    root.add(sub_folder)
    size_calculator = SizeCalculator()
    root.accept(size_calculator)
    print(f"Общий размер: {size_calculator.total_size}")
    print("\nСтруктура файлов:")
    structure_printer = StructurePrinter()
    root.accept(structure_printer)
```

Пояснение:

– Element и его конкретные реализации (*File*, *Folder*) представляют элементы структуры, которые могут быть обработаны посетителем. Метод *accept* принимает объект Visitor и вызывает его метод, соответствующий типу элемента. Это позволяет отделить логику обработки от структуры;

– Visitor и его конкретные реализации (*SizeCalculator*, *StructurePrinter*) представляют операции, которые можно выполнять над элементами. Каждый тип элемента (файл, папка) имеет отдельный метод в интерфейсе *Visitor*.

5.11.3 Пример на Java

Аналогичный пример: структура файлов, состоящая из папок и файлов. Требуется подсчитать общий размер всех элементов и вывести их структуру.

```
interface Visitor {
    void visitFile(File file);
    void visitFolder(Folder folder);
}
interface Element {
    void accept(Visitor visitor);
}
class File implements Element {
    private String name;
    private int size;
    public File(String name, int size) {
        this.name = name;
        this.size = size;
    }
    public String getName() {
        return name;
    }
    public int getSize() {
        return size;
    }
    @Override
    public void accept(Visitor visitor) {
        visitor.visitFile(this);
    }
}
class Folder implements Element {
    private String name;
    private List<Element> children = new ArrayList<>();
    public Folder(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void add(Element element) {
        children.add(element);
    }
    public List<Element> getChildren() {
        return children;
    }
    @Override
    public void accept(Visitor visitor) {
        visitor.visitFolder(this);
    }
}
class SizeCalculator implements Visitor {
    private int totalSize = 0;
    public int getTotalSize() {
        return totalSize;
    }
    @Override
    public void visitFile(File file) {
```

```
        System.out.println("Размер файла '" + file.getName() + "': " +
file.getSize());
        totalSize += file.getSize();
    }
    @Override
    public void visitFolder(Folder folder) {
        System.out.println("Обходим папку '" + folder.getName() + "'");
        for (Element child : folder.getChildren()) {
            child.accept(this);
        }
    }
}
class StructurePrinter implements Visitor {
    private int depth = 0;
    @Override
    public void visitFile(File file) {
        System.out.println("  ".repeat(depth) + "Файл: " + file.get-
Name());
    }
    @Override
    public void visitFolder(Folder folder) {
        System.out.println("  ".repeat(depth) + "Папка: " + folder.get-
Name());
        depth++;
        for (Element child : folder.getChildren()) {
            child.accept(this);
        }
        depth--;
    }
}
public class VisitorPatternExample {
    public static void main(String[] args) {
        Folder root = new Folder("root");
        root.add(new File("file1.txt", 100));
        root.add(new File("file2.txt", 200));
        Folder subFolder = new Folder("sub_folder");
        subFolder.add(new File("file3.txt", 300));
        root.add(subFolder);
        SizeCalculator sizeCalculator = new SizeCalculator();
        root.accept(sizeCalculator);
        System.out.println("Общий размер: " + sizeCalculator.getTo-
talSize());
        System.out.println("\nСтруктура файлов:");
        StructurePrinter structurePrinter = new StructurePrinter();
        root.accept(structurePrinter);
    }
}
```

5.12 Контрольные вопросы для самоподготовки

Тема 1. Проектирование и реализация паттерна «Цепочка обязанностей» (Chain of Responsibility)

Цель самоподготовки: практическое освоение архитектурного приема организации последовательной обработки запросов, позволяющего избежать жесткой привязки отправителя запроса к его конкретному получателю (обработчику).

1. Формирование контракта обработчика (*Handler*). Какие обязательные элементы должен содержать интерфейс или абстрактный класс *Handler*? Обоснуйте необходимость метода для установки ссылки на следующее звено цепочки (например, *setNext()*) и единого метода обработки.

2. Реализация базового делегирования. Опишите логику поведения по умолчанию в базовом абстрактном классе обработчика. Каким образом проверка наличия ссылки на следующего обработчика избавляет конкретные классы-наследники от дублирования системного кода?

3. Специфика конкретных обработчиков (*ConcreteHandler*). Спроектируйте алгоритм принятия решения внутри конкретного обработчика. В каких случаях узел должен выполнить бизнес-логику самостоятельно, а в каких безусловно или опционально передать запрос дальше по цепи?

4. Конфигурирование и динамика цепи. Опишите процесс сборки цепочки в клиентском или конфигурационном коде. Каким образом порядок связывания объектов влияет на итоговую логику обработки запроса? Можно ли изменять структуру цепи во время выполнения программы?

5. Инициация запросов и независимость клиента. Спроектируйте взаимодействие клиента с первым звеном цепи. Как архитектура паттерна гарантирует, что клиент остается независимым от состава и количества обработчиков в системе? Поясните, как это способствует соблюдению принципа открытости/закрытости (OCP).

6. Визуализация рекурсивных зависимостей в UML. Какой графический элемент на диаграмме классов является определяющим для «Цепочки обязанностей»? Объясните значение рефлексивной направленной ассоциации от интерфейса *Handler* к самому себе и её роль в поддержании структуры связного списка объектов.

Тема 2. Проектирование и реализация паттерна «Команда» (Command)

Цель самоподготовки: практическое освоение архитектурного приема инкапсуляции запроса в виде самостоятельного объекта, что позволяет параметризовать клиенты различными запросами, ставить их в очередь, логировать, а также поддерживать отмену операций.

1. Декларация интерфейса *Command*. Какую роль играет метод *execute()* в базовом контракте команды? Опишите, как добавление метода *undo()* меняет структуру хранения состояния в объекте команды для поддержки обратимости действий.

2. Роль получателя (*Receiver*) в бизнес-логике. Почему в паттерне «Команда» важно отделять логику выполнения (находящуюся в *Receiver*) от самой команды? Опишите типичные обязанности класса-получателя и объясните, почему он не должен знать о существовании объектов команд.

3. Инкапсуляция в конкретных командах (*ConcreteCommand*). Спроектируйте класс конкретной команды. Каким образом внутри него реализуется связь с объектом Receiver и хранение аргументов вызова? Объясните механизм делегирования: как метод *execute()* преобразует абстрактный запрос в конкретные вызовы методов получателя.

4. Функции инициатора (*Invoker*). Какова ответственность класса-отправителя (*Invoker*)? Обоснуйте, почему инициатор должен хранить ссылку на интерфейс *Command*, а не на конкретную реализацию. В какой момент времени и при каких условиях инициатор должен вызывать метод *execute()*?

5. Конфигурирование системы и слабая связанность (*Decoupling*). Опишите процесс сборки системы в клиентском коде: от инстанцирования получателя до связывания команды с

инициатором. Докажите, что в этой архитектуре отправитель запроса полностью изолирован от объекта, выполняющего этот запрос.

6. Визуализация взаимодействия в UML. Спроектируйте диаграмму классов для данного паттерна. Четко разграничьте связи: направленную ассоциацию от *Invoker* к *Command*, отношение реализации и связь (агрегацию или ассоциацию) между *ConcreteCommand* и *Receiver*. Объясните, как эта структура поддерживает динамическую подмену команд у одного инициатора.

Тема 3. Проектирование и реализация паттерна «Интерпретатор» (Interpreter)

Цель самоподготовки: практическое освоение архитектурного подхода для представления грамматики заданного языка программирования или логических выражений и создания механизма их оценки (вычисления).

1. Роль и структура контекста (*Context*). Зачем в процессе интерпретации необходим выделенный класс *Context*? Опишите, какие данные (глобальное состояние, переменные, входной поток) он должен инкапсулировать и как обеспечивается доступ к ним для всех узлов грамматики.

2. Проектирование общей абстракции выражения. Обоснуйте необходимость создания базового интерфейса *AbstractExpression*. Каким образом метод *interpret(Context)* связывает воедино все элементы древовидной структуры языка?

3. Специфика терминальных выражений. Какую роль в иерархии грамматики играют классы *TerminalExpression*? Объясните, почему эти узлы-листья не содержат вложенных выражений и как они реализуют базовые вычисления, используя данные из *Context*.

4. Логика нетерминальных выражений. Спроектируйте структуру класса *NonTerminalExpression*. Каким образом в нем реализуется хранение ссылок на другие выражения и логика рекурсивного вызова интерпретации для дочерних узлов? Опишите процесс комбинирования результатов, полученных от вложенных структур.

5. Формирование и обход абстрактного синтаксического дерева (AST). Опишите процесс построения AST в клиентском коде или специализированном парсере. Каким образом запуск метода *interpret()* у корневого узла инициирует каскадное вычисление по всем ветвям дерева?

6. Архитектурное сходство и визуализация в UML. Проанализируйте сходство паттернов «Интерпретатор» и «Компоновщик». Как на UML-диаграмме классов отобразить отношение агрегации от нетерминального выражения к базовому интерфейсу *AbstractExpression* для обеспечения рекурсивной вложенности?

Тема 4. Проектирование и реализация паттерна «Итератор» (Iterator)

Цель самоподготовки: практическое освоение архитектурного механизма последовательного обхода элементов составных объектов (коллекций) без раскрытия их внутренней структуры и способа представления.

1. Контракт итератора и стандартизация обхода. Какие стандартные операции должен регламентировать базовый интерфейс *Iterator*? Опишите назначение методов для получения следующего элемента, проверки его наличия и отслеживания текущего состояния обхода.

2. Интерфейс агрегата (*Iterable*/*Aggregate*). Зачем коллекции необходим специальный интерфейс для создания итераторов? Обоснуйте применение фабричного метода (например, *createIterator()*) и объясните, почему возвращаемый тип должен быть абстрактным интерфейсом, а не конкретным классом.

3. Инкапсуляция структуры в конкретной коллекции. Спроектируйте класс *ConcreteAggregate*. Каким образом внутренняя структура данных (массив, список или дерево) скрывается от внешнего мира? Опишите процесс реализации фабричного метода, который инстанцирует подходящий итератор для данной структуры.

4. Состояние и логика конкретного итератора. Опишите внутреннее устройство класса *ConcreteIterator*. Каким образом он должен хранить ссылку на коллекцию и инкапсулировать текущую позицию обхода (индекс или указатель)? Как гарантировать, что итератор не нарушит приватность данных коллекции при доступе к элементам?

5. Изоляция клиента от реализации данных. Спроектируйте фрагмент клиентского кода для перебора элементов. Как обеспечить взаимодействие клиента исключительно через абстракции *Iterable* и *Iterator*? В чем заключается преимущество такой независимости при замене одной структуры данных на другую?

6. Визуализация параллельных иерархий в UML. Как на диаграмме классов отобразить параллельные иерархии коллекций и итераторов? Объясните значение направленной ассоциации от конкретного итератора к конкретной коллекции и её роль в процессе извлечения данных в обход публичного интерфейса коллекции.

Тема 5. Проектирование и реализация паттерна «Посредник» (Mediator)

Цель самоподготовки: Практическое освоение архитектурного приема снижения связности системы за счет инкапсуляции способов взаимодействия множества объектов внутри единого объекта-координатора.

1. Декларация контракта посредника. Опишите назначение интерфейса *Mediator*. Каким образом метод обмена сообщениями (например, *notify(sender, event)*) позволяет унифицировать сигналы, поступающие от различных компонентов системы?

2. Архитектура базового компонента. Обоснуйте необходимость инкапсуляции ссылки на интерфейс *Mediator* внутри каждого компонента системы. Почему компоненты должны зависеть от абстракции посредника, а не от его конкретной реализации?

3. Изоляция конкретных компонентов (*ConcreteComponents*). Спроектируйте структуру независимого компонента. Каким образом реализуется принцип «незнания» о других участниках системы? Объясните механизм уведомления посредника при наступлении важных событий внутри бизнес-логики компонента.

4. Логика централизованной маршрутизации. Опишите внутреннее устройство класса *ConcreteMediator*. Каким образом он должен хранить ссылки на управляемые компоненты и как реализуется алгоритм принятия решений в методе *notify()*? Приведите пример того, как посредник перенаправляет вызов от одного компонента к набору методов других.

5. Анализ гибкости и слабой связности. Докажите, что использование посредника упрощает добавление новых компонентов в систему. Почему изменение логики взаимодействия между двумя элементами теперь требует модификации только кода посредника, не затрагивая сами компоненты?

6. Визуализация топологии «звезда» в UML. Как на диаграмме классов отразить централизованную структуру паттерна? Объясните специфику направленных ассоциаций: от базового компонента к интерфейсу *Mediator* (для отправки событий) и от *ConcreteMediator* обратно к конкретным компонентам (для управления ими).

Тема 6. Проектирование и реализация паттерна «Хранитель» (Memento)

Цель самоподготовки: практическое освоение архитектурного механизма фиксации и восстановления предыдущего состояния объекта без нарушения принципа инкапсуляции его внутренних данных.

1. Ограничение интерфейса снимка (*Memento*). В чем заключается концепция «узкого» и «широкого» интерфейсов для объекта *Memento*? Опишите, какие данные могут быть доступны внешним объектам через базовый контракт, а какие должны быть полностью скрыты для обеспечения инкапсуляции.

2. Роль создателя (*Originator*). Спроектируйте структуру целевого класса *Originator*. Каким образом реализуются методы для создания снимка (экспорт состояния) и его восстановления (импорт состояния)? Почему только этот класс должен иметь право определять структуру данных снимка?

3. Обеспечение тесной связи между компонентами. Опишите технические приемы (например, вложенные классы или специфические уровни видимости), которые позволяют классу *ConcreteMemento* быть «открытым» для создателя и «закрытым» для всех остальных. Почему это требование является критическим для архитектуры паттерна?

4. Функции опекуна (*Caretaker*) и управление историей. Какова ответственность класса *Caretaker* в процессе фиксации состояний? Обоснуйте выбор структуры данных (например, «стек») для хранения истории снимков. Объясните, почему опекуну строго запрещено читать или модифицировать содержимое хранимых объектов.

5. Алгоритм восстановления состояния. Опишите последовательность действий при выполнении операции отмены. Каким образом опекун извлекает нужный снимок и как происходит процесс возврата *Originator* к предыдущей контрольной точке?

6. Визуализация уровней доступа в UML. Как на диаграмме классов отобразить разделение уровней доступа к снимку? Опишите связи зависимости опекуна от базового интерфейса *Memento* и создателя от конкретной реализации снимка. Почему на диаграмме важно подчеркнуть, что опекун не имеет прямой связи с внутренними полями *ConcreteMemento*?

Тема 7. Проектирование и реализация паттерна «Наблюдатель» (Observer)

Цель самоподготовки: практическое освоение архитектурного механизма создания динамической подписки, позволяющего объектам реагировать на изменения состояния других объектов при сохранении слабой связности (*loose coupling*) системы.

1. Декларация контракта подписчика. Какую роль играет метод *update()* в базовом интерфейсе *Observer*? Опишите целесообразность передачи параметров (контекста изменения) в этот метод: в каких случаях лучше передавать конкретные данные, а в каких ссылку на сам объект издателя?

2. Управление инфраструктурой подписки в интерфейсе *Subject*. Обоснуйте состав методов базового интерфейса *Subject*. Каким образом методы присоединения (*attach*) и отсоединения (*detach*) позволяют динамически менять состав участников системы во время выполнения программы?

3. Логика работы конкретного издателя (*ConcreteSubject*). Спроектируйте структуру класса *ConcreteSubject*. Каким образом он должен инкапсулировать коллекцию активных подписчиков? Опишите алгоритм работы метода *notify()*: почему итерация по коллекции должна происходить именно при изменении значимого бизнес-состояния?

4. Реализация вариативности подписчиков. Опишите процесс создания классов *ConcreteObserver*. Как реализовать различную реакцию на одно и то же уведомление от издателя в разных классах подписчиков? Приведите примеры таких реакций (например, обновление UI, запись в лог, отправка email).

5. Обеспечение слабой связности (*Loose Coupling*). Докажите, что издатель может успешно оповещать зависимые объекты, абсолютно не зная их конкретных классов. Как использование абстракции *Observer* позволяет расширять систему новыми типами подписчиков без модификации кода издателя?

6. Визуализация динамических связей в UML. Какие типы отношений на диаграмме классов характерны для паттерна «Наблюдатель»? Объясните значение направленной агрегации от класса *Subject* к интерфейсу *Observer* и её роль в представлении списка рассылки уведомлений.

Тема 8. Проектирование и реализация паттерна «Состояние» (State)

Цель самоподготовки: практическое освоение архитектурного приема инкапсуляции зависящего от состояния поведения в самостоятельные классы для динамического изменения поведения объекта при изменении его внутреннего состояния.

1. Проектирование класса Контекст (*Context*). Какова роль класса *Context* как внешнего интерфейса для клиента? Опишите механизм делегирования: каким образом контекст перенаправляет выполнение бизнес-логики текущему объекту состояния через ссылку на интерфейс *State*?

2. Определение базового контракта состояния. Спроектируйте общий интерфейс *State*. Обоснуйте набор методов, поведение которых должно радикально меняться в зависимости от текущей фазы жизненного цикла объекта. Почему интерфейс состояния часто дублирует методы интерфейса контекста?

3. Реализация конкретных состояний (*ConcreteState*). Опишите процесс создания подклассов для каждого возможного состояния системы. Каким образом вынос общей логики в абстрактный базовый класс позволяет избежать дублирования кода при реализации специфичного поведения?

4. Механизмы переходов (*Transitions*). Проанализируйте способы смены состояний. В чем заключаются преимущества и недостатки инициации перехода внутри самого контекста по сравнению с переключением из методов конкретных классов состояний? Опишите назначение метода *setState()*.

5. Устранение условных конструкторов. Докажите на примере, как использование паттерна «Состояние» позволяет избавиться от громоздких операторов *if* или *switch*. Каким образом достигается «прозрачность» переключения поведения для клиентского кода?

6. Визуализация динамического поведения в UML. Какие типы связей на диаграмме классов характеризуют паттерн «Состояние»? Объясните значение отношения агрегации (или композиции) от класса *Context* к интерфейсу *State* и его сходство с паттерном «Стратегия» при принципиально иной семантике использования.

Тема 9. Проектирование и реализация паттерна «Стратегия» (Strategy)

Цель самоподготовки: практическое освоение архитектурного приема инкапсуляции семейства взаимозаменяемых алгоритмов в самостоятельные классы для обеспечения их независимого использования и переключения во время выполнения программы.

1. Декларация интерфейса алгоритма. Спроектируйте общий интерфейс *Strategy*. Каким образом фиксация единой сигнатуры метода позволяет сделать различные алгоритмы взаимозаменяемыми для вызывающей стороны?

2. Вариативность реализаций (*ConcreteStrategy*). Опишите процесс разработки набора классов конкретных стратегий. Приведите примеры предметных областей (например, способы начисления налогов, алгоритмы сжатия данных), где выделение алгоритма в отдельный класс является оправданным.

3. Конфигурирование контекста и внедрение зависимостей. Спроектируйте класс *Context*. Обоснуйте требование, согласно которому контекст не должен содержать реализации алгоритмов. Опишите механизмы внедрения конкретной стратегии в контекст через конструктор или *setter*-методы.

4. Протокол обмена данными при делегировании. Опишите механизм вызова алгоритма внутри бизнес-методов контекста. Каким образом контекст должен передавать данные стратегии (через аргументы метода или путем передачи ссылки на самого себя)? Проанализируйте плюсы и минусы обоих подходов.

5. Соблюдение принципа *Open/Closed*. Докажите, что использование паттерна «Стратегия» позволяет добавлять в систему новые алгоритмы без модификации кода контекста и существующих стратегий. Какую роль в этом играет изоляция клиентского кода?

6. Семантический анализ на UML-диаграмме. Несмотря на визуальную идентичность паттерну «Состояние», в чем заключается ключевое семантическое отличие «Стратегии» на уровне объектных связей? Объясните, почему для этого паттерна нехарактерно наличие логики переходов между объектами-алгоритмами внутри них самих.

Тема 10. Проектирование и реализация паттерна «Шаблонный метод» (Template Method)

Цель самоподготовки: практическое освоение архитектурного приема определения скелета алгоритма в суперклассе с делегированием реализации его отдельных шагов подклассам без изменения общей структуры процесса.

1. Проектирование структуры абстрактного класса. Опишите назначение «шаблонного метода» в базовом классе *AbstractClass*. Каким образом он фиксирует жесткую последовательность (алгоритм) вызовов других методов, представляющих собой отдельные шаги?

2. Защита скелета алгоритма. Какие технические средства языка программирования позволяют исключить возможность переопределения шаблонного метода в подклассах (например, модификаторы *final* или *sealed*)? Обоснуйте важность неизменности структуры алгоритма для стабильности системы.

3. Классификация примитивных операций (шагов). Проанализируйте и опишите различия между тремя категориями шагов алгоритма.

В каких случаях следует использовать каждый из них:

– обязательные (абстрактные) – те, что подклассы обязаны реализовать;

– базовые – имеющие готовую реализацию, общую для всех;

– перехватчики/хуки (*hooks*) – методы с реализацией по умолчанию, предназначенные для расширения логики.

4. Специализация в конкретных классах. Спроектируйте класс *ConcreteClass*. Каким образом реализация абстрактных методов позволяет адаптировать общий алгоритм под специфические требования подкласса без вмешательства в код суперкласса?

5. Анализ инверсии управления. Объясните механизм инверсии контроля при выполнении шаблонного метода. Каким образом базовый класс берет на себя ответственность за управление потоком выполнения и вызов методов подклассов в строго определенном порядке? Докажите, что это предотвращает создание запутанных двусторонних зависимостей.

6. Визуализация иерархии в UML. Как на диаграмме классов наглядно отобразить различие между шаблонным методом и переопределяемыми шагами? Опишите графическое представление отношения наследования (генерализации) между конкретными классами и абстрактным суперклассом.

Тема 11. Проектирование и реализация паттерна «Посетитель» (Visitor)

Цель самоподготовки: практическое освоение архитектурного механизма двойной диспетчеризации для добавления новых операций в существующую гетерогенную иерархию объектов без изменения кода самих этих объектов.

1. Формирование контракта посетителя. Опишите структуру интерфейса *Visitor*. Почему в нем необходимо объявить отдельный метод посещения (*visit*) для каждого конкретного типа элемента существующей иерархии? Объясните связь между набором методов посетителя и стабильностью иерархии элементов.

2. Модификация иерархии элементов. В чем заключается суть метода *accept(Visitor v)*, добавляемого в базовый интерфейс элементов? Обоснуйте утверждение, что это изменение является «единственным и окончательным» для структуры элементов при использовании данного паттерна.

3. Механизм двойной диспетчеризации. Детально опишите процесс взаимодействия в методе *accept()* конкретного элемента. Каким образом вызов *v.visit(this)* позволяет системе определить нужный метод посетителя во время выполнения программы? Объясните роль «двойного прыжка» (диспетчеризации) в этой архитектуре.

4. Реализация новых операций (*ConcreteVisitor*). Спроектируйте класс *ConcreteVisitor*. Каким образом инкапсуляция новой логики в методах *visit()* позволяет расширять функциональность системы, не затрагивая классы самих элементов? Приведите пример операции (например, экспорт в XML или подсчет статистики), которую удобно реализовать таким способом.

5. Организация обхода гетерогенной структуры. Опишите алгоритм взаимодействия клиента с коллекцией разнотипных элементов. Как осуществляется передача объекта-посетителя в структуру данных и как гарантируется, что каждый элемент будет обработан соответствующим методом посетителя?

6. Визуализация перекрестных зависимостей в UML. Как на диаграмме классов отобразить взаимодействие двух независимых иерархий? Объясните, почему метод *accept* должен зависеть от абстракции *Visitor*, в то время как методы *visit* зависят от конкретных типов элементов. Отразите эту циклическую зависимость в архитектурном анализе.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Гамма, Э. Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. – Санкт-Петербург : Питер, 2019. – 368 с. – ISBN 978-5-4461-0867-1.

2. Шалл, Э. Паттерны проектирования: элементы повторно используемого объектно-ориентированного программного обеспечения / Э. Шалл, Э. Гамма. – Москва : Рус-ская редакция, 2016. – 432 с. – ISBN 978-5-906825-05-2.

3. Фаулер, М. Шаблоны корпоративных приложений / М. Фаулер. – Москва : Вильямс, 2018. – 544 с. – ISBN 978-5-8459-1302-9.

4. Буч, Г. Объектно-ориентированный анализ и проектиро-вание с примерами приложений / Г. Буч, Д. Рамбо, И. Джекобсон. – 3-е изд. – Москва : Бином, 2017. – 720 с. – ISBN 978-5-9902126-9-0.

5. Хантер, Дж. Java I/O / Дж. Хантер. – Москва : Вильямс, 2016. – 448 с. – ISBN 978-5-8459-1351-7.

6. Коплиен, Дж. Программируемые шаблоны проектирования / Дж. Коплиен. – Москва : ДМК Пресс, 2018. – 320 с. – ISBN 978-5-94074-877-5.

7. Ларман, К. Применение UML и шаблонов проектирования / К. Ларман. – 3-е изд. – Москва : Вильямс, 2019. – 736 с. – ISBN 978-5-8459-1167-4.

8. Бушнев, А.Г. Паттерны проектирования в реальных проектах / А.Г. Бушнев. – Санкт-Петербург : БХВ-Петербург, 2020. – 304 с. – ISBN 978-5-9775-6317-8.

9. Макконнелл, С. Совершенный код: руководство по разработке программного обес-печения / С. Макконнелл. – Москва : Русская редакция, 2018. – 896 с. – ISBN 978-5-906825-09-0.

10. Мартин, Р. Чистый код: создание, анализ и рефакторинг / Р. Мартин. – Москва : Питер, 2017. – 464 с. – ISBN 978-5-496-02539-3.

11. Фримен, Э. Head First. Паттерны проектирования / Э. Фримен. – Москва : Питер, 2018. – 656 с. – ISBN 978-5-496-00018-3.