Учреждение образования Федерации профсоюзов Беларуси
«Международный университет «МИТСО»

Факультет экономический
Кафедра информационных технологий

СОГЛАСОВАНО
Заведующий кафедрой
_____О.Б.Хорошко
29. 05 2026 г.

СОГЛАСОВАНО
Декан факультета
_____А.В.Ковтунов
07. 07 2026 г.

ЭЛЕКТРОННЫЙ УЧЕБНО-МЕТОДИЧЕСКИЙ КОМПЛЕКС ПО УЧЕБНОЙ ДИСЦИПЛИНЕ

ВЕБ-ДИЗАЙН И ШАБЛОНЫ ПРОЕКТИРОВАНИЯ ВЕБ-ПРИЛОЖЕНИЙ

для специальности 6-05-0611-01 Информационные системы и технологии

Составитель: М.А. Калинин, старший преподаватель кафедры информационных технологий учреждения образования Федерации профсоюзов Беларуси «Международный университет «МИТСО»

Рассмотрено и рекомендовано к утверждению на заседании кафедры информационных технологий учреждения образования Федерации профсоюзов Беларуси «Международный университет МИТСО»
29. 05. 2026 г., протокол № 11

Утверждено на заседании научно-методического совета учреждения образования Федерации профсоюзов Беларуси «Международный университет «МИТСО»
07. 07 2026 г., протокол № 6

РЕЦЕНЗЕНТЫ:
Кафедра микропроцессорных систем и сетей УО «Белорусский государственный университет информатики и радиоэлектроники» (Кашникова И.В., заведующий кафедрой);
Гришко Н.И., доцент кафедры экономики и менеджмента УО ФПБ «Международный университет «МИТСО», к.т.н., доцент

Регистрационный № УД-079-26/э

Регистрационное свидетельство № 1202021654 от 11.02.2020
новый, зарегистрирован как актуализация 20.07.2026

АКТУАЛИЗИРОВАН
заседание кафедры _____
_____ учреждения
образования Федерации профсоюзов Беларуси «Международный университет МИТСО»
_____20_____, протокол №_____

заседание научно-методического совета учреждения образования Федерации профсоюзов Беларуси «Международный университет «МИТСО»
_____ 20__ г., протокол № ___

ОГЛАВЛЕНИЕ

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА 7
Учебная программа по дисциплине 8
I. ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ 23
Введение в веб-дизайн и шаблоны проектирования Веб-приложений 23
Понятия, методология, принципы проектирования веб-приложений 23
Интерфейсы веб-приложений, методы, инструменты и техника разработки веб-приложений 24
Средства проектирования типовых решений, лучшие решения и шаблоны (паттерны) проектирования 25
Методы, средства работы с компьютерной графикой, публикация компьютерной графики в Интернете 26
ТЕМА 1. СОВРЕМЕННЫЕ ТЕХНОЛОГИИ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО АНАЛИЗА И ПРОЕКТИРОВАНИЯ ИНФОРМАЦИОННЫХ СИСТЕМ 28
Предмет объектно-ориентированного анализа и проектирования 28
Понятие повторного использования проектных решений 28
Основные концепции ООП с точки зрения повторного использования 29
Влияние повторного использования на механизмы современных языков программирования 29
Инкапсуляция, наследование, полиморфизм. Достоинства и недостатки 30
Элементарные приемы объектного проектирования: композиция и делегирование 31
Разница между объектным и структурным подходами 31
Тема 2. Интерфейсы. Методология объектно-ориентированного анализа и проектирования 33
Интерфейсы в объектно-ориентированном программировании 33
Интерфейсы и абстрактные классы. Достоинства и недостатки 33
Множественное наследование и реализация интерфейсов 34
Интерфейсы в конкретных языках и системах 35
Виды диаграмм UML. Диаграмма классов 37
Диаграмма активностей 38
Диаграммы взаимодействия. Последовательности 39
Диаграммы вариантов использования 39
Обзор CASE-средств для построения диаграмм UML 40
Тема 3. Шаблоны объектно-ориентированного анализа и проектирования, их классификация 43
Принцип классификации шаблонов проектирования 43
Что такое типовое решение проектирования 43

Форматы описания паттернов проектирования 44
Паттерны проектирования в нотации языка UML 44
Отношения, связи и альтернативы применения паттернов 45
Тема 4. Порождающие шаблоны 46
Обзор применения порождающих типовых решений проектирования 46
Фабричный метод 46
Абстрактная фабрика 47
Строитель 47
Прототип 48
Одиночка 48
Обсуждение порождающих шаблонов проектирования 49
Синергия нескольких порождающих шаблонов 50
Тема 5. Структурные шаблоны 51
Обзор применения структурных решений проектирования 51
Адаптер 51
Мост 52
Компоновщик 52
Декоратор 53
Фасад 53
Заместитель 54
Приспособленец 54
Обсуждение структурных паттернов проектирования 55
Тема 6. Поведенческие шаблоны 56
Обзор применения поведенческих решений проектирования 56
Цепочка обязанностей 56
Команда 57
Интерпретатор 57
Итератор 58
Медиатор 58
Хранитель 59
Состояние 59
Стратегия 60
Посетитель 61
Обсуждение поведенческих шаблонов проектирования 61
Тема 7. Методика применения типовых решений проектирования 62
Пример паттерна проектирования: Модель-вид-контроллер 62
Как выбирать паттерн проектирования 62
Поиск подходящих объектов 63
Определение степени детализации объекта 64
Специфицирование интерфейсов объекта 64
Как пользоваться паттерном проектирования 65

Тема 8. Виды и примеры шаблонов, не зависящих от языка программирования.
Виды и примеры шаблонов, специфичных для языка программирования 66
Архитектурные шаблоны 66
Конструктивные шаблоны 67
Шаблоны интеграции 67
Антишаблоны 68
Идиомы 69
Пример идиомы C++. RAII 69
Преимущества разработки программ с применением шаблонов 70
Тема 9. Основы компьютерной графики 71
Понятие о графическом процессе 71
Понятие о геометрическом моделировании 71
Типы моделей, особенности их получения 71
Основные направления развития и области применения компьютерной графики 72
Классификация графических примитивов, их атрибуты, используемые в компьютерной графике 72
Виды компьютерной графики – растровая, векторная, 3D-графика 73
Принципы представления растровой и векторной информации 73
Особенности изображения в экранном представлении и полиграфии 74
Разрешение 74
Обзор, назначение и возможности пакетов векторной и растровой графики 75
Достоинства и недостатки векторной графики 75
Интеграция векторной и растровой графики 76
Устройства ввода и вывода графической информации 76
Тема 10. Понятие цвета и его представление в компьютерной графике 77
Определение цвета 77
Особенности восприятия цвета 77
Цветовые модели 78
Характеристики цвета 78
Управление цветом, его составляющие 79
Профили 79
Цветовые палитры 80
Основы колористики 80
Тема 11. Веб-дизайн. Основы визуального дизайна. Принципы юзабилити 81
Понятие веб-дизайна 81
Графика в веб-дизайне 81
Информационная архитектура веб-проекта 82
Микро- и макротипографика 82
Свойства зрительного восприятия 83

Основные принципы графического дизайна. Законы и приемы композиции. 83
Понятия: размер, форма, цвет, текстура, размещение, шрифт, композиция, контраст 84
Основные принципы создания графики для Веб 84
Оптимизация графических изображений 85
Размеры и разрешение 85
Уменьшение количества цветов 86
Выбор формата файла 86
Тема 12. Обзор современных методов и технологий проектирования и верстки Веб-сайтов 87
Статические и динамические компоненты представления Веб-приложений 87
Технологии, используемые на стороне сервера: ASP-технология, ASP.NET, PHP, Python, Perl 87
Технологии веб-программирования, применяющиеся на стороне клиента для реализации GUI: HTML, XHTML, CSS 88
Обзор информационных систем управления контентом (CMS) 89
Виды верстки: таблицы, фреймы, блоки 90
Тема 13. назначение и применение CSS 91
Понятие каскадных таблиц стилей, их назначение, определение и использование при форматировании HTML-документа 91
Способы подключения стилей к документу 91
Замена тегов на стилевое описание 92
Стили элементов. Блочные и строковые элементы 92
Стилевое позиционирование 93
Стили классов 93
Внешние стилевые таблицы 94
Добавление фонового рисунка 94
Текст, списки, таблицы 95
Цветовое и декоративное оформление текста 95
Оформление заголовка 96
Контекстный селектор, псевдоэлементы и псевдоклассы, группировка, каскадирование и наследование 96
II. ПРАКТИЧЕСКИЙ РАЗДЕЛ 98
Планы и задания для лабораторных работ 98
III. РАЗДЕЛ КОНТРОЛЯ ЗНАНИЙ 107
Тестовое задание 107
Примерный перечень вопросов к экзамену 111
IV. ВСПОМОГАТЕЛЬНЫЙ РАЗДЕЛ 113
Перечень рекомендуемой литературы 113

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

Электронный учебно-методический комплекс (далее – ЭУМК) подготовлен в соответствии с учебной программой учреждения образования Федерации профсоюзов Беларуси «Международный университет «МИТСО» по дисциплине «Веб-дизайн и шаблоны проектирования» для специальности 6-05-0611-01 «Информационные системы и технологии».

Цель ЭУМК – обеспечение максимальной доступности и конкретности изучаемого материала по дисциплине. Практическая значимость состоит в том, что данное издание позволяет конкретизировать и спланировать учебную деятельность слушателей, разнообразить формы усвоения достаточно сложного материала.

Основными структурными элементами ЭУМК являются:

- учебная программа по дисциплине;
- теоретический раздел, включающий конспект лекций для теоретического изучения учебной дисциплины;
- практический раздел, содержащий учебно-методические материалы для поведения практических занятий, задачи для индивидуальной работы;
- раздел контроля знаний, в котором представлены контрольные вопросы по дисциплине, вопросы для самостоятельного изучения по всем предусмотренным учебной программой темам дисциплины;

Рекомендации по организации работы с ЭУМК. Для лучшего усвоения получаемых знаний слушателям рекомендуется сначала ознакомиться с учебной программой по дисциплине «Веб-дизайн и шаблоны проектирования». Усвоение материала по каждой из тем рекомендуется начинать с изучения соответствующей лекции («Теоретический раздел»), после чего следует проработать практические задания, решить индивидуальные задачи («Практический раздел»). Проверка знаний проводится на основе тестов, перечня вопросов к зачету («Раздел контроля знаний»).

Студентам рекомендуется использовать материалы ЭУМК для осуществления активной самостоятельной работы при изучении дисциплины, подготовки к аудиторным занятиям и экзамену.

Учреждение образования Федерации профсоюзов Беларуси
«Международный университет «МИТСО»

УТВЕРЖДАЮ
Проректор по учебной работе
учреждения образования
Федерации профсоюзов Беларуси
«Международный университет «МИТСО»
_____ М.А.Юрочкин
20.12 2024 г.
Регистрационный № УД-057/02-24/уч.

ВЕБ-ДИЗАЙН И ШАБЛОНЫ ПРОЕКТИРОВАНИЯ ВЕБ-ПРИЛОЖЕНИЙ

Учебная программа учреждения высшего образования
по учебной дисциплине для специальности

6-05-0611-01 Информационные системы и технологии

2024 г.

Контрольный экземпляр

Учебная программа составлена на основе Образовательного стандарта высшего образования ОСВО 6-05-0611-01-2023 (общее высшее образование), для специальности 6-05-0611-01 «Информационные системы и технологии», утвержденного Постановлением Министерства образования Республики Беларусь 10.08.2023 г. № 246, и учебного плана учреждения образования Федерации профсоюзов Беларуси «Международный университет «МИТСО» по специальности(ям) 6-05-0611-01 «Информационные системы и технологии», утвержденного 01.03.2023 рег. № 6-05-0611-01/уч.

СОСТАВИТЕЛИ:
М.А. Калинин, старший преподаватель кафедры информационных технологий учреждения образования Федерации профсоюзов Беларуси «Международный университет «МИТСО»

РЕКОМЕНДОВАНА К УТВЕРЖДЕНИЮ:
Кафедрой информационных технологий учреждения образования Федерации профсоюзов Беларуси «Международный университет «МИТСО»
(протокол № 4 от 28.11.2024);

Научно-методическим советом учреждения образования Федерации профсоюзов Беларуси «Международный университет «МИТСО»
(протокол № 3 от 20.12.2024)

СОГЛАСОВАНО
Декан экономического факультета _____А.В.Ковтунов

Заведующий библиотекой _____О.О.Бабарикина

Нормоконтроль
ведущий специалист УМУ
_____Г.Д.Лагунович

I. ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

Учебная дисциплина «Веб-дизайн и шаблоны проектирования веб-приложений» разработана для студентов учреждений высшего образования, обучающихся по специальности 6-05-0611-01 «Информационные системы и технологии».

Учебная дисциплина «Веб-дизайн и шаблоны проектирования веб-приложений» является актуальной и формирует у студентов навыки владения шаблонами проектирования для разработки программных продуктов с дружественным интерфейсом с использованием современных подходов и технологий объектно-ориентированного программирования; владения приемами веб-дизайна и компьютерной графики.

Цель учебной дисциплины: подготовка специалиста, владеющего основами компьютерной графики, приемами веб-дизайна, инструментальными средствами разработки визуальных компонент представления веб-контента, технологиями представления, создания и использования шаблонов проектирования (спецификаций лучших проектных решений) при разработке сложных объектно-ориентированных приложений, ориентированных на повторное применение кода (методы повторного использования).

Задачи учебной дисциплины:

усвоить теоретические знания в области компьютерной графики и обработки изображения;

изучить методы объектно-ориентированного анализа и эффективного проектирования архитектурных и программных решений на основе шаблонов проектирования.

изучить основы разработки компонент представления веб-контента;

изучить типовые приемы проектирования (шаблонов проектирования) на основе статических и динамических диаграмм UML;

получить навыки разработки элементов дизайна веб-приложений с использованием основных инструментальных визуальных средств проектирования сайтов и графических редакторов;

получить навыки применения типовых решений для разработки и кодирования приложений.

Освоение учебной дисциплины «Веб-дизайн и шаблоны проектирования веб-приложений» должно обеспечить у обучающихся формирование следующих **специализированных компетенций**: разрабатывать дружественные пользовательские интерфейсы с использованием технологических инструментальных средств и паттернов проектирования.

В результате изучения учебной дисциплины студент должен

знать:

общую теорию веб-дизайна и основы взаимодействия в архитектуре клиент-сервер;

работу с графикой, методы, техники и инструменты разработки веб-приложений;

методы проектирования и программной реализации веб-интерфейсов;

общую концепцию, технологию, методы и технику применения и разработки паттернов;

семейство базовых паттернов и паттернов, ориентированных на используемую платформу программных приложений;

уметь:

работать с графическими редакторами, осуществлять компьютерную верстку страниц;

применять технологии и средства разработки расширений функциональности сервера;

применять языки и технологии разработки веб-страниц и веб-сайтов;

применять существующие и разрабатывать паттерны для обеспечения эффективности программных решений;

иметь навыки:

техникой работы с основными графическими пакетами и методами обработки векторной и растровой графики;

технологиями проектирования веб-сайтов, методами, приемами и техникой разработки сложных интерфейсов;

техникой и методами создания графических и стилистических элементов для веб-приложений приемами верстки, подготовки и размещения на сервере графики и контента;

методами абстрагирования и разработки проектных решений программных систем.

Распределение аудиторных часов по видам занятий и семестрам.
Формы промежуточной аттестации

Семестр	Количество академических часов							Самост. работа	Форма промежуточной аттестации
	Всего	Аудит.	Из них						
			Лекции	Лабор. занятия	Практ. занятия	Семи-нары	УСР[1]		
Очная (дневная) форма получения высшего образования									
4	110	56	14	30			12	54	экз.
Итого	110	56	14	30			12	54	

[1] Управляемая самостоятельная работа

II. СОДЕРЖАНИЕ УЧЕБНОЙ ДИСЦИПЛИНЫ

Введение

Понятия, методология, принципы проектирования веб-приложений. Интерфейсы веб-приложений, методы, инструменты и техника разработки вебприложений. Средства проектирования типовых решений, лучшие решения и шаблоны (паттерны) проектирования. Методы, средства работы с компьютерной графикой, публикация компьютерной графики в Интернете.

Раздел 1. Шаблоны проектирования

Тема 1.1. Современные технологии объектно-ориентированного анализа и проектирования информационных систем

Предмет объектно-ориентированного анализа и проектирования. Понятие повторного использования проектных решений. Основные концепции ООП с точки зрения повторного использования. Влияние повторного использования на механизмы современных языков программирования. Инкапсуляция, наследование, полиморфизм. Достоинства и недостатки. Элементарные приемы объектного проектирования: композиция и делегирование. Разница между объектным и структурным подходами.

Тема 1.2. Интерфейсы. Методология объектно-ориентированного анализа и проектирования

Интерфейсы в объектно-ориентированном программировании. Интерфейсы и абстрактные классы. Множественное наследование и реализация интерфейсов. Интерфейсы в конкретных языках и системах. Основные концепции унифицированного языка моделирования: визуализация, прямое и обратное проектирование, специфицирование, документирование. Виды диаграмм UML: диаграмма классов, диаграмма активностей, диаграммы взаимодействия, диаграммы вариантов использования. Обзор CASE-средств для построения диаграмм UML.

Тема 1.3. Шаблоны объектно-ориентированного анализа и проектирования, их классификация

Принцип классификации шаблона проектирования. Что такое типовое решение проектирования. Форматы описания паттернов проектирования. Паттерны проектирования в нотации языка UML. Отношения, связи и альтернативы применения паттернов.

Тема 1.4. Порождающие шаблоны

Обзор применения порождающих типовых решений проектирования. Порождающие шаблоны: абстрактная фабрика, строитель, фабричный метод, прототип, одиночка. Обсуждение порождающих шаблонов проектирования. Синергия нескольких порождающих шаблонов.

Тема 1.5. Структурные шаблоны

Обзор применения структурных решений проектирования. Структурные паттерны: адаптер, мост, компоновщик, декоратор, фасад, заместитель, приспособленец. Обсуждение структурных паттернов проектирования.

Тема 1.6. Поведенческие шаблоны

Обзор применения поведенческих решений проектирования. Поведенческие паттерны: цепочка обязанностей, команда, интерпретатор, итератор, медиатор, хранитель, посредник, состояние, стратегия, фабричный метод, посетитель. Обсуждение поведенческих шаблонов проектирования.

Тема 1.7. Методика применения типовых решений проектирования

Пример паттерна проектирования: Модель-вид-контроллер. Как выбирать паттерн проектирования. Поиск подходящих объектов. Определение степени детализации объекта. Специфицирование интерфейсов объекта. Как пользоваться паттерном проектирования.

Тема 1.8. Виды и примеры шаблонов, не зависящих от языка программирования. Виды и примеры шаблонов, специфичных для языка программирования

Архитектурные шаблоны. Конструктивные шаблоны. Шаблоны интеграции. Антишаблоны. Идиомы. Пример идиом C++. Преимущества разработки программ с применением шаблонов.

Раздел 2. Веб-дизайн и компьютерная графика

Тема 2.1. Основы компьютерной графики

Понятие о графическом процессе. Понятие о геометрическом моделировании. Типы моделей, особенности их получения. Основные направления развития и области применения компьютерной графики. Классификация графических примитивов, их атрибуты, используемые в компьютерной графике. Виды компьютерной графики – растровая, векторная, 3D-графика. Принципы представления растровой и векторной информации. Особенности изображения в экранном представлении и полиграфии. Разрешение. Обзор, назначение и возможности пакетов векторной и растровой графики. Достоинства и недостатки векторной графики. Интеграция векторной и растровой графики. Устройства ввода/вывода графической информации.

Тема 2.2. Понятие цвета и его представление в компьютерной графике

Определение цвета. Особенности восприятия цвета. Цветовые модели: RGB, CMYK, CIE, Lab, HSB и другие. Характеристики цвета: глубина, динамический диапазон, гамма цветов устройств, цветовой охват. Управление цветом, его составляющие. Профили. Цветовые палитры. Основы колористики.

Тема 2.3. Веб-дизайн. Основы визуального дизайна. Принципы ЮЗАБИЛИТИ

Понятие веб-дизайна. Графика в веб-дизайне. Информационная архитектура веб-проекта. Микро- и макро- типографика. Свойства зрительного восприятия. Основные принципы графического дизайна. Законы и приемы композиции и др. Понятия: размер, форма, цвет, текстура, размещение, шрифт, композиция, контраст. Основные принципы создания графики для Веб. Оптимизация графических изображений. Размеры и разрешение. Уменьшение количества цветов. Выбор формата файла.

Тема 2.4. Обзор современных методов и технологий проектирования и верстки веб-сайтов

Статические и динамические компоненты представления веб-приложений.

Технологии, используемые на стороне сервера: ASP-технология, ASP.net, PHP, Python, Perl. Технологии веб-программирования применяющиеся на стороне клиента для реализации GUI: HTML, XHTML, CSS.

Технологии веб-программирования применяющиеся на стороне клиента для формирования и обработки запросов, создания интерактивного и независимого от браузера интерфейса: JavaScript, Java, Апплты Java, Active-X, Adobe Flash\Flex, Silverlight, AJAX. Обзор информационных систем управления контентом (CMS). Виды верстки: таблицы, фреймы, блоки.

Тема 2.5. Назначение и применение CSS

Понятие каскадных таблиц стилей, их назначение, определение и использование при форматировании HTML-документа. Способы подключения стилей к документу. Замена тегов на стилевое описание.

Стили элементов. Блочные и строковые элементы. Стилевое позиционирование. Стили классов. Внешние стилевые таблицы. Добавление фонового рисунка. Текст, списки, таблицы. Цветовое и декоративное оформление текста. Оформление заголовка. Контекстный селектор, псевдоэлементы и псевдоклассы, группировка, каскадирование и наследование

III. УЧЕБНО-МЕТОДИЧЕСКАЯ КАРТА ДИСЦИПЛИНЫ

Очная (дневная) форма получения высшего образования

Номер раздела, темы	Название раздела, темы	Всего часов	Количество аудиторных часов					Самостоятельная работа	Литература	Форма контроля
			Лекции	Практические занятия	Семинарские занятия	Лабораторные занятия	Количество часов УСР			
4 семестр										
	Введение	3	1					2	[1]	
1.	**Шаблоны проектирования**									
1.1	Современные технологии объектно-ориентированного анализа и проектирования информационных систем	6	1				1	4	[1]	УО, УД
1.2	Интерфейсы. Методология объектно-ориентированного анализа и проектирования	7	1				2	4	[1]	УО, УД
1.3	Шаблоны объектно-ориентированного анализа и проектирования, их классификация	10	1			4	1	4	[1]	УО, РПЗ, Р, УД
1.4	Порождающие шаблоны	10	1			4	1	4	[1]	УО, РПЗ, Р, УД
1.5	Структурные шаблоны	10	1			4	1	4	[1]	УО, РПЗ, Р, УД
1.6	Поведенческие шаблоны	10	1			4	1	4	[1]	УО, РПЗ, Р, УД
1.7	Методика применения типовых решений проектирования	7	1			2		4	[1]	УО, РПЗ, Р, УД
1.8	Виды и примеры шаблонов, не зависящих от языка программирования. Виды и примеры шаблонов, специфичных для языка программирования	7	1			2		4	[1]	УО, РПЗ, Р(ТА)
2	**Веб-дизайн и компьютерная графика**									
2.1	Основы компьютерной графики	8	1			2	1	4	[2]	УО, РПЗ, Р, УД
2.2	Понятие цвета и его представление в компьютерной графике	6	1				1	4	[2]	УО, РПЗ, Р, УД
2.3	Веб-дизайн. Основы визуального дизайна. Принципы ЮЗАБИЛИТИ	10	1			4	1	4	[2]	УО, РПЗ, Р, УД
2.4	Обзор современных методов и технологий проектирования и верстки веб-сайтов	6	1				1	4	[2]	УО, РПЗ, Р, УД
2.5	Назначение и применение CSS	10	1			4	1	4	[2]	УО, РПЗ, Р, УД
	Всего по дисциплине	**110**	**14**			**30**	**12**	**54**		**экз.**

IV. ИНФОРМАЦИОННО-МЕТОДИЧЕСКАЯ ЧАСТЬ

ОСНОВНАЯ ЛИТЕРАТУРА

1. Гамма, Э. Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. – СПб: «Питер», 2016. – 366 с.

2. Лупптон, Э. Графический дизайн. Базовые концепции / Э. Лупптон. – СПб. : Питер, 2019. – 256 с.

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

3. Сирых, Ю. Современный веб-дизайн. Настольный и мобильный / Ю. Сирых. – Киев : Диалектика, 2019. – 384 с.

4. Филичи, Дж. Типографика: шрифт, верстка, дизайн / Дж. Филичи. – СПб. : БХВ-Петербург, 2020. – 496 с.

СРЕДСТВА ОБУЧЕНИЯ

Необходимыми средствами для реализации успешного формирования профессиональных компетенций у студентов являются:

оборудованный интернет-доступом компьютерный класс с установленным специальным программным обеспечением;

мультимедийная аппаратура;

электронные учебно-наглядные пособия (презентации, методические материалы).

СРЕДСТВА ДИАГНОСТИКИ РЕЗУЛЬТАТОВ УЧЕБНОЙ ДЕЯТЕЛЬНОСТИ

Диагностика результатов образовательной деятельности обучающихся осуществляется в ходе проведения всех видов занятий, самостоятельной работы и текущей аттестации по учебной дисциплине.

Текущая аттестация обучающихся проводится в течение семестра в целях периодического контроля и оценки результатов их учебной деятельности по учебной дисциплине с выставлением отметок.

Основными **формами контроля** знаний по учебной дисциплине являются:

устный опрос	УО;
устный доклад	УД;
реферат	Р;
решение практических задач	РПЗ.

Текущий контроль проводится строго в соответствии с учебно-методической картой дисциплины. Обучающийся должен в обязательном порядке участвовать во всех контрольных мероприятиях текущего контроля, предусмотренных учебной программой дисциплины.

Форма текущей аттестации по учебной дисциплине:

реферат Р.

К промежуточной аттестации по учебной дисциплине, модулю обучающиеся допускаются при условии успешного прохождения текущей аттестации, предусмотренной учебной программой в текущем семестре.

Формой промежуточной аттестации является:

экзамен экз.

ПЕРЕЧЕНЬ ТЕМ ЛАБОРАТОРНЫХ ЗАНЯТИЙ

Основная цель проведения лабораторных занятий состоит в закреплении теоретического материала курса, приобретении навыков выполнения эксперимента, обработки экспериментальных данных, анализа результатов, грамотного оформления отчетов.

Лабораторная работа 1. **Тема 1**. Шаблоны объектно-ориентированного анализа и проектирования, их классификация

Обеспеченность занятия: ПК,Visual Studio Code.

Лабораторная работа 2. **Тема 2**. Порождающие шаблоны

Обеспеченность занятия: ПК,Visual Studio Code.

Лабораторная работа 3. **Тема 3**. Структурные шаблоны

Обеспеченность занятия: ПК,Visual Studio Code.

Лабораторная работа 4. **Тема 4**. Поведенческие шаблоны

Обеспеченность занятия: ПК,Visual Studio Code.

Лабораторная работа 5. **Тема 5**. Методика применения типовых решений проектирования

Обеспеченность занятия: ПК,Visual Studio Code.

Лабораторная работа 6. **Тема 6**. Виды и примеры шаблонов, не зависящих от языка программирования. Виды и примеры шаблонов, специфичных для языка программирования

Обеспеченность занятия: ПК,Visual Studio Code.

Лабораторная работа 7. **Тема 7**. Основы компьютерной графики

Обеспеченность занятия: ПК,Visual Studio Code.

Лабораторная работа 8. **Тема 8**. Веб-дизайн. Основы визуального дизайна. Принципы ЮЗАБИЛИТИ

Обеспеченность занятия: ПК,Visual Studio Code.

Лабораторная работа 9. **Тема 9**. Назначение и применение CSS

Обеспеченность занятия: ПК,Visual Studio Code.

ПРИМЕРНЫЙ ПЕРЕЧЕНЬ ЗАДАНИЙ И КОНТРОЛЬНЫХ МЕРОПРИЯТИЙ УСР

№ темы, (раздела)	Тема УСР	Кол-во часов	Метод. обеспечение	Форма контроля
2 курс, 3 семестр (12 часов)				
1.1	Современные технологии объектно-ориентированного анализа и проектирования информационных систем	1	Интернет-ресурсы	Реферат, доклад, сообщение
1.2	Интерфейсы. Методология объектно-ориентированного анализа и проектирования	2	Интернет-ресурсы	Реферат, доклад, сообщение
1.3	Шаблоны объектно-ориентированного анализа и проектирования, их классификация	1	Интернет-ресурсы	Реферат, доклад, сообщение
1.4	Порождающие шаблоны	1	Интернет-ресурсы	Реферат, доклад, сообщение
1.5	Структурные шаблоны	1	Интернет-ресурсы	Реферат, доклад, сообщение
1.6	Поведенческие шаблоны	1	Интернет-ресурсы	Реферат, доклад, сообщение
2.1	Основы компьютерной графики	1	Интернет-ресурсы	Реферат, доклад, сообщение
2.2	Понятие цвета и его представление в компьютерной графике	1	Интернет-ресурсы	Реферат, доклад, сообщение
2.3	Веб-дизайн. Основы визуального дизайна. Принципы ЮЗАБИЛИТИ	1	Интернет-ресурсы	Реферат, доклад, сообщение
2.4	Обзор современных методов и технологий проектирования и верстки веб-сайтов	1	Интернет-ресурсы	Реферат, доклад, сообщение
2.5	Назначение и применение CSS	1	Интернет-ресурсы	Реферат, доклад, сообщение

ТЕМАТИКА РЕФЕРАТОВ УСР

1. Поведенческие паттерны: цепочка обязанностей, команда, интерпретатор, итератор, медиатор, хранитель, посредник, состояние, стратегия, фабричный метод, посетитель.
2. Пример паттерна проектирования: Модель-вид-контроллер.
3. Антишаблоны.
4. Классификация графических примитивов, их атрибуты, используемые в компьютерной графике.
5. Цветовые модели: RGB, CMYK, CIE, Lab, HSB и другие.
6. Микро- и макро- типографика.
7. Технологии, используемые на стороне сервера: ASP-технология, ASP.net, PHP, Python, Perl.
8. Технологии веб-программирования применяющиеся на стороне клиента для формирования и обработки запросов, создания интерактивного и независимого от браузера интерфейса: JavaScript, Java, Апплты Java, Active-X, Adobe Flash\Flex, Silverlight, AJAX.

ПРИМЕРНЫЙ ПЕРЕЧЕНЬ ВОПРОСОВ ПРОМЕЖУТОЧНОЙ АТТЕСТАЦИИ (ЭКЗАМЕН)

1. Понятия, методология, принципы проектирования веб-приложений.
2. Интерфейсы веб-приложений, методы, инструменты и техника разработки веб-приложений.
3. Средства проектирования типовых решений, лучшие решения и шаблоны (паттерны) проектирования.
4. Методы, средства работы с компьютерной графикой, публикация компьютерной графики в Интернете.
5. Предмет объектно-ориентированного анализа и проектирования. Понятие повторного использования проектных решений.
6. Основные концепции ООП с точки зрения повторного использования. Влияние повторного использования на механизмы современных языков программирования. Инкапсуляция, наследование, полиморфизм. Достоинства и недостатки.
7. Элементарные приемы объектного проектирования: композиция и делегирование.
8. Разница между объектным и структурным подходами.
9. Интерфейсы в объектно-ориентированном программировании.
10. Интерфейсы и абстрактные классы. Множественное наследование и реализация интерфейсов.
11. Интерфейсы в конкретных языках и системах.
12. Основные концепции унифицированного языка моделирования: визуализация, прямое и обратное проектирование, специфицирование, документирование.
13. Виды диаграмм UML: диаграмма классов, диаграмма активностей, диаграммы взаимодействия, диаграммы вариантов использования.
14. Обзор CASE-средств для построения диаграмм UML.
15. Принцип классификации шаблона проектирования.
16. Что такое типовое решение проектирования.
17. Форматы описания паттернов проектирования.
18. Паттерны проектирования в нотации языка UML.
19. Отношения, связи и альтернативы применения паттернов.
20. Обзор применения порождающих типовых решений проектирования.
21. Порождающие шаблоны: абстрактная фабрика, строитель, фабричный метод, прототип, одиночка.
22. Обсуждение порождающих шаблонов проектирования.
23. Синергия нескольких порождающих шаблонов.
24. Обзор применения структурных решений проектирования.
25. Структурные паттерны: адаптер, мост, компоновщик, декоратор, фасад, заместитель, приспособленец.
26. Обсуждение структурных паттернов проектирования.
27. Обзор применения поведенческих решений проектирования.
28. Поведенческие паттерны: цепочка обязанностей, команда,

интерпретатор, итератор, медиатор, хранитель, посредник, состояние, стратегия, фабричный метод, посетитель.

29. Обсуждение поведенческих шаблонов проектирования.
30. Пример паттерна проектирования: Модель-вид-контроллер.
31. Как выбирать паттерн проектирования.
32. Поиск подходящих объектов.
33. Определение степени детализации объекта.
34. Специфицирование интерфейсов объекта.
35. Как пользоваться паттерном проектирования.
36. Архитектурные шаблоны.
37. Конструктивные шаблоны.
38. Шаблоны интеграции.
39. Антишаблоны.
40. Идиомы.
41. Пример идиом C++.
42. Преимущества разработки программ с применением шаблонов.
43. Понятие о графическом процессе.
44. Понятие о геометрическом моделировании.
45. Типы моделей, особенности их получения.
46. Основные направления развития и области применения компьютерной графики.
47. Классификация графических примитивов, их атрибуты, используемые в компьютерной графике.
48. Виды компьютерной графики – растровая, векторная, 3D-графика.
49. Принципы представления растровой и векторной информации.
50. Особенности изображения в экранном представлении и полиграфии.
51. Разрешение.
52. Обзор, назначение и возможности пакетов векторной и растровой графики.
53. Достоинства и недостатки векторной графики.
54. Интеграция векторной и растровой графики.
55. Устройства ввода/вывода графической информации.
56. Определение цвета.
57. Особенности восприятия цвета.
58. Цветовые модели: RGB, CMYK, CIE, Lab, HSB и другие.
59. Характеристики цвета: глубина, динамический диапазон, гамма цветов устройств, цветовой охват.
60. Управление цветом, его составляющие.
61. Профили.
62. Цветовые палитры.
63. Основы колористики.
64. Понятие веб-дизайна.
65. Графика в веб-дизайне.
66. Информационная архитектура веб-проекта.
67. Микро- и макро- типографика.

68. Свойства зрительного восприятия.

69. Основные принципы графического дизайна.

70. Законы и приемы композиции и др.

71. Понятия: размер, форма, цвет, текстура, размещение, шрифт, композиция, контраст.

72. Основные принципы создания графики для Веб.

73. Оптимизация графических изображений.

74. Размеры и разрешение.

75. Уменьшение количества цветов.

76. Выбор формата файла.

77. Статические и динамические компоненты представления веб-приложений.

78. Технологии, используемые на стороне сервера: ASP-технология, ASP.net, PHP, Python, Perl.

79. Технологии веб-программирования применяющиеся на стороне клиента для реализации GUI: HTML, XHTML, CSS.

80. Технологии веб-программирования применяющиеся на стороне клиента для формирования и обработки запросов, создания интерактивного и независимого от браузера интерфейса: JavaScript, Java, Апплты Java, Active-X, Adobe Flash\Flex, Silverlight, AJAX.

81. Обзор информационных систем управления контентом (CMS).

82. Виды верстки: таблицы, фреймы, блоки.

83. Понятие каскадных таблиц стилей, их назначение, определение и использование при форматировании HTML-документа.

84. Способы подключения стилей к документу.

85. Замена тегов на стилевое описание.

86. Стили элементов.

87. Блочные и строковые элементы.

88. Стилевое позиционирование.

89. Стили классов.

90. Внешние стилевые таблицы.

91. Добавление фонового рисунка.

92. Текст, списки, таблицы.

93. Цветовое и декоративное оформление текста.

94. Оформление заголовка.

95. Контекстный селектор, псевдоэлементы и псевдоклассы, группировка, каскадирование и наследование

V. ПРОТОКОЛ СОГЛАСОВАНИЯ УЧЕБНОЙ ПРОГРАММЫ

Название учебной дисциплины, с которой требуется согласование	Название кафедры	Предложения об изменениях в содержании учебной программы учреждения высшего образования по учебной дисциплине	Решение, принятое кафедрой, разработавшей учебную программу (с указанием даты и номера протокола)
Основы компьютерной графики	кафедра информационных технологий	Нет	протокол заседания кафедры информационных технологий № 4 от 28.11.2024)
Объектно-ориентированное проектирование и программирование	кафедра информационных технологий	Нет	протокол заседания кафедры информационных технологий № 4 от 28.11.2024)

I. ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ

ВВЕДЕНИЕ В ВЕБ-ДИЗАЙН И ШАБЛОНЫ ПРОЕКТИРОВАНИЯ ВЕБ-ПРИЛОЖЕНИЙ

Понятия, методология, принципы проектирования веб-приложений

Веб-приложение – это клиент-серверное программное обеспечение, в котором клиентом выступает веб-браузер, а взаимодействие пользователя с приложением происходит через интернет или интранет. В отличие от статических веб-сайтов, которые просто отображают одинаковую информацию для всех посетителей, веб-приложения являются интерактивными: они обрабатывают вводимые пользователем данные, выполняют бизнес-логику на сервере, обращаются к базам данных и динамически формируют ответ. Примерами веб-приложений служат интернет-магазины, социальные сети, почтовые сервисы, онлайн-банки, системы управления проектами и многие другие.

Методология проектирования веб-приложений представляет собой совокупность подходов, этапов и рекомендаций по созданию таких систем. Как правило, процесс включает сбор требований, анализ предметной области, проектирование архитектуры, проектирование пользовательского интерфейса, реализацию, тестирование, развёртывание и сопровождение. В последние годы широкое распространение получили гибкие методологии, особенно Scrum и Kanban, поскольку веб-разработка часто характеризуется быстро меняющимися требованиями и необходимостью частых релизов. Однако выбор конкретной методологии зависит от масштаба проекта, команды и требований заказчика.

Принципы проектирования веб-приложений можно разделить на несколько групп. Принципы архитектурного проектирования включают разделение ответственности, например модель MVC, отделяющую данные от представления и логики управления, а также принцип единственной ответственности и инверсию зависимостей. Принципы пользовательского интерфейса ориентированы на удобство: понятность, предсказуемость, обратную связь на действия пользователя, минимизацию когнитивной нагрузки, доступность для людей с ограниченными возможностями. Принципы безопасности включают защиту от распространённых атак, таких как межсайтовый скриптинг и подделка межсайтовых запросов, а также валидацию и экранирование всех пользовательских данных. Принципы производительности требуют минимизации времени загрузки страниц, оптимизации запросов к базе данных, использования кэширования и сжатия ресурсов. Наконец, принципы

поддерживаемости предполагают чистоту и документированность кода, модульность, тестируемость и использование систем контроля версий.

Интерфейсы веб-приложений, методы, инструменты и техника разработки веб-приложений

Интерфейсы веб-приложений обычно делят на два типа. Пользовательский интерфейс, или frontend, – это всё, что видит и с чем взаимодействует пользователь в браузере: веб-страницы, формы, кнопки, анимация, диалоговые окна. Программный интерфейс, или API, используется для взаимодействия между различными программными компонентами: браузер взаимодействует с сервером через HTTP-запросы, а сервер может предоставлять RESTful или GraphQL API для мобильных приложений и других сервисов. В современных веб-приложениях также широко применяются веб-сокеты для двусторонней связи в реальном времени, например в чатах или онлайн-играх.

Методы разработки веб-приложений охватывают как клиентскую, так и серверную стороны. На клиентской стороне обязательным является знание HTML для структуры документа, CSS для стилизации и JavaScript для интерактивности. Сегодня активно используются фреймворки и библиотеки, такие как React, Angular или Vue.js, которые позволяют создавать сложные одностраничные приложения, управляя состоянием и реактивно обновляя интерфейс. На серверной стороне доступен широкий выбор языков и сред: Node.js, Python с Django или Flask, Java с Spring, PHP с Laravel, Ruby on Rails, C# с ASP.NET Core и другие. Выбор зависит от требований к производительности, экосистемы и предпочтений команды. Методы также включают использование систем управления базами данных, кэширующих серверов, очередей сообщений и контейнеризации.

Инструменты разработки веб-приложений делятся на редакторы кода, например Visual Studio Code, Sublime Text, IntelliJ IDEA; инструменты для отладки, встроенные в браузеры, такие как Chrome DevTools; системы контроля версий, прежде всего Git; инструменты автоматизации сборки, такие как Webpack, Vite, Gulp; системы непрерывной интеграции и развёртывания, например Jenkins, GitLab CI, GitHub Actions. Для работы с базами данных используются административные инструменты, такие как phpMyAdmin, pgAdmin, DBeaver. Для тестирования применяются фреймворки Jest, Mocha, Selenium, Postman для тестирования API. Важной частью инструментария являются менеджеры пакетов, такие как npm, yarn, pip, composer, которые упрощают управление зависимостями.

Техника разработки веб-приложений включает в себя практики написания кода, такие как семантическая вёрстка с использованием HTML5-элементов, адаптивный и отзывчивый дизайн с помощью медиа-запросов и

флексбоксов/гридов, прогрессивное улучшение, когда базовая функциональность работает даже без JavaScript, и постепенная деградация. Используются методологии именования CSS-классов, например BEM. В области программирования применяются паттерны проектирования, о которых речь пойдёт ниже, а также функциональное и объектно-ориентированное программирование в зависимости от языка. Современная техника также включает написание автоматических тестов, как модульных, так и интеграционных и end-to-end, а также использование систем логирования и мониторинга для выявления проблем в рабочей среде.

Средства проектирования типовых решений, лучшие решения и шаблоны (паттерны) проектирования

Для ускорения разработки веб-приложений широко используются средства проектирования типовых решений. К ним относятся фреймворки, которые предоставляют готовую архитектуру и набор компонентов для решения типичных задач. На клиентской стороне это Bootstrap, Tailwind CSS для стилизации, а также упомянутые React, Angular, Vue для построения интерфейсов. На серверной стороне фреймворки предоставляют ORM для работы с базами данных, маршрутизацию, шаблонизаторы, middleware для обработки запросов. Также существуют CMS, такие как WordPress, Drupal, или конструкторы сайтов, позволяющие создавать веб-приложения без глубокого программирования. Кроме того, используются библиотеки компонентов, например Material-UI, Ant Design, которые содержат готовые кнопки, формы, диалоги, соответствующие принципам Material Design или другим дизайн-системам.

Лучшие решения в веб-разработке часто оформляются в виде паттернов проектирования. Паттерны архитектуры приложений включают модель MVC, которая разделяет данные, представление и контроллер, и её вариации, такие как MVVM. Паттерны организации кода на клиентской стороне включают модульный паттерн, позволяющий инкапсулировать логику, паттерн наблюдатель для реактивных обновлений, паттерн контейнер-презентация для разделения логики и отображения в React-приложениях. Паттерны серверной архитектуры включают репозиторий для абстракции доступа к данным, сервисный слой для бизнес-логики, внедрение зависимостей. Паттерны интеграции включают шлюз для унифицированного доступа к внешним API, агрегатор для объединения данных из нескольких источников. Также существуют паттерны безопасности, такие как валидация входных данных с использованием белых списков, защита от CSRF с помощью токенов, контроль доступа на основе ролей. Знание этих паттернов позволяет разработчикам не изобретать велосипед, а использовать проверенные решения типовых проблем.

Методы, средства работы с компьютерной графикой, публикация компьютерной графики в Интернете

Компьютерная графика является неотъемлемой частью современных веб-приложений. Для работы с графикой используются растровые редакторы, такие как Adobe Photoshop, GIMP, для обработки фотографий, ретуши, создания коллажей. Векторные редакторы, например Adobe Illustrator, Inkscape, Figma, позволяют создавать логотипы, иконки, интерфейсные элементы, которые затем масштабируются без потери качества. Для создания трёхмерной графики и анимации применяются Blender, Autodesk Maya, а также библиотеки для веба, такие как Three.js для рендеринга 3D в браузере с использованием WebGL. Средства для работы с графикой также включают инструменты для сжатия изображений, например TinyPNG, для оптимизации размера файлов без видимого ухудшения качества.

Методы работы с графикой для веба включают выбор подходящего формата в зависимости от типа контента. Фотографии и сложные изображения с градиентами сохраняют в формате JPEG с регулируемой степенью сжатия. Изображения с прозрачностью, логотипы, скриншоты – в формате PNG, который обеспечивает сжатие без потерь. Для анимированной простой графики используется формат GIF, хотя его всё чаще заменяют видео в форматах MP4 или WebM, которые дают лучшее качество при меньшем размере. Современным форматом является WebP, который обеспечивает лучшее сжатие как с потерями, так и без потерь по сравнению с JPEG и PNG. Для иконок и простых иллюстраций идеален векторный формат SVG, который может быть встроен непосредственно в HTML и стилизован с помощью CSS. Также существуют новые форматы, такие как AVIF, обеспечивающие ещё более высокую степень сжатия.

Публикация компьютерной графики в Интернете включает несколько этапов. Подготовка изображений подразумевает их оптимизацию для веба: изменение размера до необходимых значений, сжатие, выбор формата, добавление атрибутов alt для доступности. Затем изображения размещаются на веб-сервере и встраиваются в HTML-страницы с помощью тега img, CSS-свойства background-image или, для адаптивных изображений, с использованием тегов picture и srcset для загрузки разных версий в зависимости от разрешения экрана. Для ускорения загрузки применяются техники ленивой загрузки, то есть отложенной загрузки изображений, которые находятся вне видимой области. Также используются CDN, или сети доставки контента, для кэширования изображений на серверах, расположенных географически ближе к пользователю. Важно учитывать правовые аспекты публикации: использование изображений, на которые у автора нет прав, запрещено, поэтому следует использовать стоковые фотографии с открытыми лицензиями или собственные материалы.

Современные веб-приложения всё чаще работают с графикой не только как с пассивным контентом, но и как с объектами для интерактива. Это может быть редактирование изображений в браузере с помощью Canvas API, наложение фильтров с помощью WebGL, создание динамических диаграмм и графиков с помощью библиотек типа D3.js, а также интеграция с нейросетями для распознавания или генерации изображений непосредственно на стороне клиента или сервера. Понимание методов работы с графикой и способов её эффективной публикации является необходимым навыком для современного веб-разработчика, поскольку именно визуальная составляющая во многом определяет пользовательский опыт и восприятие приложения.

ТЕМА 1. СОВРЕМЕННЫЕ ТЕХНОЛОГИИ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО АНАЛИЗА И ПРОЕКТИРОВАНИЯ ИНФОРМАЦИОННЫХ СИСТЕМ

Предмет объектно-ориентированного анализа и проектирования

Объектно-ориентированный анализ и проектирование представляют собой методологию создания информационных систем, в основе которой лежит представление предметной области в виде совокупности взаимодействующих объектов. Каждый объект обладает состоянием, то есть набором данных, и поведением, то есть набором операций, которые могут быть выполнены над этими данными. Предметом объектно-ориентированного анализа является выявление и описание классов и объектов, их свойств, связей и поведения в исследуемой предметной области безотносительно к конкретной программной реализации. Объектно-ориентированное проектирование переводит результаты анализа в архитектуру программной системы, определяя интерфейсы классов, наследование, полиморфные связи и механизмы взаимодействия. Ключевая идея заключается в том, что модель системы строится из тех же понятий, что и реальный мир, что облегчает понимание, коммуникацию между заказчиком и разработчиком, а также последующее сопровождение и модификацию системы.

Понятие повторного использования проектных решений

Повторное использование проектных решений, или reuse, является одной из фундаментальных целей объектно-ориентированного подхода. Вместо того чтобы каждый раз создавать программное обеспечение с нуля, разработчик может применять уже существующие, проверенные и отлаженные решения. Повторное использование может быть нескольких уровней. На самом низком уровне повторно используются отдельные классы или библиотеки функций. На более высоком уровне повторно используются целые компоненты или фреймворки. На уровне проектирования повторное использование выражается в применении паттернов проектирования, то есть типовых архитектурных решений для часто встречающихся проблем. Наконец, на самом высоком уровне повторно используются целые архитектурные стили или даже готовые программные продукты. Повторное использование сокращает время разработки, повышает надёжность, поскольку проверенный код содержит меньше ошибок, унифицирует стиль и облегчает сопровождение. Однако оно требует дисциплины, документации и культуры проектирования «на вырост», чтобы создаваемые компоненты действительно могли быть использованы в других проектах.

Основные концепции ООП с точки зрения повторного использования

С точки зрения повторного использования, три основные концепции объектно-ориентированного программирования – инкапсуляция, наследование и полиморфизм – играют ключевую роль. Инкапсуляция позволяет скрыть внутреннюю реализацию объекта и предоставить чётко определённый интерфейс. Это делает объект «чёрным ящиком», который можно использовать, не зная его внутреннего устройства. При повторном использовании такой объект надёжен, потому что его внутренности не могут быть случайно повреждены внешним кодом. Наследование позволяет создавать новый класс на основе существующего, добавляя или уточняя функциональность. Это механизм повторного использования на уровне исходного кода: дочерний класс автоматически получает все методы и поля родительского, что избавляет от необходимости переписывать их заново. Полиморфизм позволяет объектам разных классов отвечать на один и тот же запрос, но по-своему. Это обеспечивает гибкость при повторном использовании: код, написанный для работы с базовым классом, автоматически работает и со всеми его наследниками. Таким образом, новые классы могут быть подставлены вместо старых без изменения клиентского кода, что является мощной формой повторного использования на уровне интерфейсов, а не реализации.

Влияние повторного использования на механизмы современных языков программирования

Стремление к повторному использованию оказало глубокое влияние на развитие языков программирования. Языки, поддерживающие объектно-ориентированную парадигму, такие как Java, C++, C#, Python, внедрили механизмы, облегчающие создание повторно используемых компонентов. Это, прежде всего, системы модулей и пакетов, позволяющие группировать связанные классы и управлять их видимостью. Механизмы обобщённого программирования, или generics, дают возможность создавать классы и методы, работающие с различными типами данных, что является формой повторного использования без привязки к конкретному типу. Интерфейсы и абстрактные классы позволяют определять контракты, которым могут следовать множество конкретных реализаций, что облегчает замену одной реализации другой. Механизмы отражения, или reflection, позволяют анализировать и создавать объекты во время выполнения, что используется в фреймворках для автоматической сборки компонентов. Управляемые среды исполнения, такие как виртуальная машина Java или среда .NET CLR, обеспечивают безопасность типов и сборку мусора, что снижает количество ошибок при повторном использовании чужого кода. Современные языки также внедряют средства аннотаций или

атрибутов для метапрограммирования, что позволяет фреймворкам автоматически подключать повторно используемые компоненты без явного кода.

Инкапсуляция, наследование, полиморфизм. Достоинства и недостатки

Инкапсуляция – это объединение данных и методов, работающих с этими данными, внутри одного объекта, а также сокрытие внутренней реализации от внешнего мира. Достоинствами инкапсуляции являются повышение надёжности за счёт контроля доступа к состоянию объекта, упрощение понимания, поскольку объект предстаёт как целостная сущность, возможность изменения внутренней реализации без затрагивания клиентского кода, если интерфейс остаётся неизменным, а также устранение побочных эффектов при доступе к данным. Недостатком инкапсуляции можно считать некоторое усложнение кода из-за необходимости писать геттеры и сеттеры, а также потенциальное снижение производительности по сравнению с прямым доступом к данным, хотя современные компиляторы и JIT-компиляторы часто устраняют эти накладные расходы.

Наследование позволяет строить иерархии классов, где дочерний класс расширяет или переопределяет поведение родительского. Достоинства наследования: повторное использование кода, поскольку методы родителя автоматически доступны потомку; логическая классификация сущностей, соответствующая иерархии понятий в предметной области; возможность полиморфной обработки семейства классов. Недостатки наследования: жёсткая связь между родителем и потомком, изменения в родителе могут неожиданно повлиять на всех потомков; нарушение инкапсуляции, поскольку потомок часто должен знать о внутреннем устройстве родителя; хрупкость иерархий при глубоком и широком наследовании; проблема «ромбовидного наследования» в языках с множественным наследованием.

Полиморфизм означает способность объектов разных классов отвечать на один и тот же вызов метода соответствующим их типу образом. Достоинства полиморфизма: унификация кода, когда клиент работает с объектами через общий интерфейс, не заботясь об их конкретном типе; расширяемость системы, поскольку новые классы могут быть добавлены без изменения существующего кода, который их использует; поддержка принципа подстановки Барбары Лисков, являющегося основой объектно-ориентированного проектирования. Недостатки полиморфизма: динамическое связывание может вносить небольшие накладные расходы по сравнению со статическими вызовами; сложность отладки, поскольку по коду не всегда очевидно, какой именно метод будет вызван; неверное использование полиморфизма может привести к запутанной архитектуре, где трудно отследить поток управления.

Элементарные приемы объектного проектирования: композиция и делегирование

Композиция и делегирование являются альтернативами наследованию для достижения повторного использования и гибкости. Композиция означает, что объект одного класса содержит в себе объекты других классов в качестве своих частей. Вместо того чтобы наследовать поведение, класс включает в себя экземпляр другого класса и переадресует ему соответствующие запросы. Преимущества композиции перед наследованием заключаются в уменьшении связности, поскольку класс не обязан знать внутреннее устройство своего компонента, достаточно знать его интерфейс. Композиция обеспечивает большую гибкость: поведение можно менять во время выполнения, заменяя один компонент другим. Она также решает проблему хрупких иерархий. Недостатком является то, что код может стать более многословным, поскольку каждый переадресованный метод нужно явно написать.

Делегирование тесно связано с композицией. При делегировании объект, получив запрос, пересылает его другому объекту, который и выполняет работу. При этом отправляющий объект может добавить свою логику до или после вызова. Делегирование часто реализуется через композицию. Например, класс Car содержит объект Engine. Вместо того чтобы наследовать метод start от какого-то абстрактного механизма, Car делегирует вызов своему двигателю. Делегирование является фундаментальным принципом многих паттернов проектирования, таких как Стратегия, Декоратор, Адаптер. Оно позволяет использовать преимущества повторного использования без фиксации иерархии наследования на этапе компиляции. Однако делегирование делает программу более динамичной, что может усложнить её понимание и отладку по сравнению со статическим наследованием.

Разница между объектным и структурным подходами

Объектный и структурный подходы к анализу и проектированию информационных систем принципиально различаются по способу представления системы. Структурный подход, также известный как функционально-ориентированный или процедурный, рассматривает систему как иерархию функций, преобразующих входные данные в выходные. Центральными понятиями здесь являются данные и процессы, которые над ними выполняются. Модель в структурном подходе описывается диаграммами потоков данных, диаграммами «сущность-связь» для данных и структурными картами для функций. Процессы и данные разделены. Изменение формата данных часто приводит к необходимости переписывать многие функции.

Объектный подход, напротив, объединяет данные и операции над ними внутри объектов. Система представляется как множество взаимодействующих объектов, каждый из которых отвечает за свою часть поведения и состояния. Центральными понятиями являются классы, объекты, наследование, полиморфизм, инкапсуляция. Модель в объектном подходе описывается диаграммами классов, диаграммами взаимодействия, диаграммами состояний и другими диаграммами языка UML. В объектном подходе данные и процессы неразрывны, что делает систему более устойчивой к изменениям: если изменится формат данных, то изменения могут быть локализованы внутри одного класса, а не разнесены по множеству функций.

Структурный подход хорошо подходит для задач с чётко определёнными, стабильными алгоритмами и входными данными, например для инженерных расчётов. Он проще для понимания и отладки в таких условиях. Объектный подход лучше подходит для сложных, развивающихся систем, особенно когда предметная область естественно иерархична или требует повторного использования компонентов. Он даёт более высокую степень модульности, поддерживает инкапсуляцию и наследование, но требует более высокой квалификации разработчиков и может приводить к излишней абстракции в простых задачах. На практике современные информационные системы часто сочетают оба подхода, используя объектную организацию на верхнем уровне и структурные алгоритмы для реализации отдельных методов. Однако объектно-ориентированный анализ и проектирование стали доминирующей парадигмой в разработке крупных информационных систем благодаря своей гибкости, масштабируемости и поддержке повторного использования.

ТЕМА 2. ИНТЕРФЕЙСЫ. МЕТОДОЛОГИЯ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО АНАЛИЗА И ПРОЕКТИРОВАНИЯ

Интерфейсы в объектно-ориентированном программировании

Интерфейс в объектно-ориентированном программировании представляет собой формальное описание набора методов, которые класс, реализующий интерфейс, обязательно должен предоставить. Иными словами, интерфейс определяет контракт или обязательство класса перед его клиентами. Главное отличие интерфейса от класса заключается в том, что интерфейс не содержит реализации методов. Он лишь задаёт сигнатуры методов, их параметры и типы возвращаемых значений.

Интерфейсы играют ключевую роль в проектировании крупных программных систем. Они позволяют отделить спецификацию того, что должен делать класс, от того, как именно он это делает. Благодаря этому разные классы могут реализовывать один и тот же интерфейс совершенно разными способами, и код, использующий интерфейс, не зависит от конкретной реализации. Это явление называется полиморфизмом на уровне интерфейсов.

Интерфейсы особенно полезны в ситуациях, когда разные, семантически не связанные классы должны предоставлять одинаковый набор операций. Например, в языке Java интерфейс Comparable содержит метод compareTo. Классы, реализующие этот интерфейс, могут быть отсортированы стандартными средствами. Ими могут быть строки, числа, даты, пользовательские классы – совершенно разные типы, но все они реализуют один и тот же контракт. Без интерфейсов каждая из этих сущностей нуждалась бы в индивидуальной обработке, что сделало бы универсальные алгоритмы сортировки невозможными.

Интерфейсы также являются механизмом, обеспечивающим подстановку Барбары Лисков, один из пяти принципов SOLID. Согласно этому принципу, если класс реализует интерфейс, то объект этого класса может быть использован везде, где ожидается этот интерфейс. Это позволяет создавать взаимозаменяемые компоненты и строить системы из слабо связанных частей.

Интерфейсы и абстрактные классы. Достоинства и недостатки

Абстрактный класс – это класс, который не может быть инстанциирован, то есть от него нельзя создать объект. Он может содержать как абстрактные методы, то есть методы без реализации, так и обычные методы с конкретной реализацией, а также поля, конструкторы и даже статические члены. Интерфейс же в своей классической форме не содержит ничего, кроме абстрактных методов и констант, и не может иметь конструкторов или полей с состоянием.

Ключевое различие между абстрактным классом и интерфейсом заключается в том, что абстрактный класс фокусируется на том, что представляет собой класс, то есть на его идентичности и общей сущности, в то время как интерфейс фокусируется на том, что класс может делать, то есть на его способностях и возможностях. Например, абстрактный класс Vehicle описывает само понятие транспортного средства и может содержать общие поля, такие как скорость и вес, а также общие методы, например move. Интерфейсы же, такие как Electric или Flyable, добавляют к конкретным классам дополнительные возможности. Класс ElectricCar наследует от Vehicle и реализует Electric, а класс FlyingCar мог бы наследовать от Vehicle и реализовывать оба интерфейса.

Абстрактные классы предпочтительны, когда существует общая иерархия, где классы действительно являются разновидностями одного базового типа, и когда предполагается, что в будущем в эту иерархию могут быть добавлены новые общие методы с реализацией по умолчанию. Интерфейсы же лучше подходят для описания возможностей, которые могут быть реализованы совершенно разными классами, не связанными иерархически. Например, интерфейс Serializable в Java могут реализовать любые классы, которым нужна поддержка сериализации, независимо от их места в иерархии.

Достоинством абстрактных классов является возможность предоставлять частичную реализацию, что уменьшает дублирование кода в наследниках. Эволюция абстрактного класса проще: добавление нового метода с реализацией по умолчанию не нарушит существующие классы-наследники. Недостатком является жёсткая иерархическая связь. Поскольку язык Java, C и другие поддерживают только одиночное наследование классов, класс может наследовать только одному абстрактному классу, что ограничивает гибкость.

Достоинство интерфейсов заключается в том, что класс может реализовывать множество интерфейсов. Это обеспечивает большую гибкость и позволяет описывать классы через множество независимых аспектов их поведения. Недостаток интерфейсов в их классической форме заключается в невозможности предоставить реализацию по умолчанию, хотя в современных языках, таких как Java начиная с версии 8, появились методы по умолчанию в интерфейсах, частично стирающие грань между интерфейсами и абстрактными классами. Другим недостатком интерфейсов является то, что любое изменение интерфейса, добавление нового метода, нарушает все существующие реализации, что требует модификации всех классов, реализующих этот интерфейс.

Множественное наследование и реализация интерфейсов

Множественное наследование – это возможность класса наследовать поведение и состояние от нескольких родительских классов. Такая возможность поддерживается в языках C++ и Python, но отсутствует в Java и C. Проблемы

множественного наследования хорошо известны. Главная из них – это проблема ромбовидного наследования, когда класс наследует от двух классов, имеющих общего предка, и возникает неоднозначность, какой именно метод предка следует использовать. Также возникают сложности с коллизией имён и с тем, как управлять состоянием, унаследованным от разных родителей.

Интерфейсы предлагают альтернативный механизм, который часто называется наследованием типа, или type inheritance, в отличие от наследования реализации. Класс может реализовывать множество интерфейсов, но при этом он не наследует никакой реализации от них. Это решает проблему ромбовидного наследования, поскольку интерфейсы не содержат полей и, следовательно, не создают конфликтов состояния. Если два интерфейса содержат метод с одинаковой сигнатурой, класс, реализующий оба интерфейса, должен предоставить одну реализацию, которая удовлетворяет обоим контрактам.

В языках, где множественное наследование допустимо, например в C++ и Python, интерфейсы как отдельная языковая конструкция отсутствуют, но их роль выполняют абстрактные классы, содержащие только абстрактные методы и не имеющие состояния. В таких языках класс может наследовать нескольким таким абстрактным классам, что эквивалентно реализации нескольких интерфейсов. Однако в этих языках существует и полное множественное наследование с реализацией, что создаёт упомянутые выше проблемы.

В контексте проектирования информационных систем, реализация множества интерфейсов является предпочтительным способом наделить класс несколькими независимыми типами поведения. Это позволяет создавать гибкие и расширяемые системы, где классы могут участвовать в различных контекстах, не будучи жёстко привязанными к конкретной иерархии.

Интерфейсы в конкретных языках и системах

В языке Java интерфейс объявляется с помощью ключевого слова interface. До версии Java 8 интерфейсы могли содержать только абстрактные методы и константы. Начиная с Java 8, интерфейсы получили методы по умолчанию, помечаемые ключевым словом default, и статические методы. Это было сделано для эволюционного расширения интерфейсов без нарушения существующего кода, например при добавлении метода forEach в интерфейс Iterable. С Java 9 появились приватные методы в интерфейсах, позволяющие разделить общую логику между методами по умолчанию. Класс в Java может реализовывать любое количество интерфейсов, но наследовать только одному классу.

В языке C# интерфейс как отдельная конструкция существует в виде специального типа interface. Он похож на интерфейсы в Java, но с тем отличием, что может содержать свойства в дополнение к методам. В C#, как и в Java, класс может реализовывать множество интерфейсов, но наследовать только одному классу. В C# интерфейсы играют ключевую роль в механизме LINQ, где

стандартные интерфейсы IEnumerable позволяют единообразно обрабатывать различные коллекции данных.

В языке C++ нет специального ключевого слова для интерфейса. Роль интерфейса выполняет абстрактный класс, все методы которого являются чисто виртуальными, то есть не имеют реализации. В C++ класс может наследовать множеству таких абстрактных классов, что эквивалентно реализации нескольких интерфейсов. Кроме того, C++ поддерживает полное множественное наследование классов с реализацией, что даёт разработчику больше гибкости, но требует внимательности при проектировании.

В языке Python интерфейс как отдельная синтаксическая конструкция официально отсутствует. Вместо этого используется концепция утиной типизации: если объект имеет нужные методы, его можно использовать в нужном контексте независимо от его типа. Однако для явного описания интерфейсов в Python существует модуль abc, абстрактные базовые классы, который позволяет объявлять абстрактные методы. Кроме того, в языке широко используются протоколы, такие как итератор, контекстный менеджер, которые являются неформальными интерфейсами. Начиная с Python 3.8, протоколы могут быть формализованы через модуль typing с помощью класса Protocol.

Основные концепции унифицированного языка моделирования: визуализация, прямое и обратное проектирование, специфицирование, документирование.

Унифицированный язык моделирования, или UML, представляет собой стандартизированный графический язык для визуализации, специфицирования, конструирования и документирования артефактов программных систем. UML не является методологией или процессом разработки, это язык, который можно использовать с любым процессом, от водопадного до гибкого. Основные концепции UML включают четыре ключевых аспекта использования.

Визуализация означает, что UML предоставляет графические нотации для выражения различных аспектов системы. Визуальное представление сложной системы часто более понятно, чем объёмный текстовый код. UML-диаграммы позволяют увидеть структуру системы, поведение, взаимодействие компонентов и другие характеристики. Визуализация особенно важна при общении между разработчиками, аналитиками, заказчиками и другими стейкхолдерами, которые могут не владеть деталями языка программирования.

Специфицирование означает, что UML может быть использован для создания точных, однозначных и полных моделей системы. В отличие от разговорного языка или неформального рисунка, UML-диаграммы имеют строгую семантику. Спецификации, выраженные в UML, могут быть использованы для генерации кода, проверки согласованности модели и для документирования архитектурных решений.

Прямое проектирование, или forward engineering, представляет собой процесс преобразования UML-модели в программный код. Многие CASE-средства позволяют по диаграмме классов сгенерировать скелет кода на Java, C, Python или другом языке. Это ускоряет разработку и гарантирует соответствие кода проектным решениям. Прямое проектирование особенно эффективно при создании новых систем, где архитектура продумывается на уровне моделей до начала кодирования.

Обратное проектирование, или reverse engineering, является процессом, противоположным прямому: по существующему программному коду создаются UML-диаграммы. Это полезно для восстановления архитектуры унаследованных систем, для онбординга новых разработчиков в проект, а также для получения общей картины, когда документация отсутствует или устарела. Некоторые инструменты способны выполнять полное обратное проектирование, анализируя код и строя диаграммы классов, последовательности и зависимостей.

Документирование означает, что UML-модели служат формой документации, которая иллюстрирует различные аспекты системы. Хорошая UML-документация включает структурные и поведенческие диаграммы, отражающие архитектуру системы на разных уровнях абстракции. В отличие от отдельных текстовых файлов, UML-модель часто хранится в едином репозитории, что облегчает её поддержку в актуальном состоянии. Для этого используется циклическое проектирование, когда модель и код поддерживаются в синхронизированном состоянии. Инструменты, поддерживающие циклическое проектирование, автоматически обновляют модель при изменении кода и наоборот, сохраняя согласованность между проектными артефактами и реализацией.

Виды диаграмм UML. Диаграмма классов

Диаграмма классов является основным видом диаграмм в UML. Она относится к структурным диаграммам, то есть показывает статическую структуру системы, а не динамику её поведения. Диаграмма классов изображает классы, их атрибуты и операции, а также связи между ними.

Класс на диаграмме изображается в виде прямоугольника, разделённого на три секции. Верхняя секция содержит имя класса, средняя секция перечисляет атрибуты, а нижняя секция – операции или методы. Каждый атрибут и операция могут сопровождаться индикаторами видимости. Плюс означает публичный член, доступный всем. Минус означает приватный член, доступный только внутри класса. Окружность означает защищённый член, доступный внутри класса и его наследникам.

Связи между классами могут быть различных типов. Ассоциация показывает структурную связь между объектами двух классов и обозначается сплошной линией. В общем случае ассоциация – это не просто линия между

двумя классами, а связь, которая может иметь направление, имя и кратность на каждом конце, например, ученик может быть привязан к одному классному руководителю, а учитель может обучать многих учеников. Агрегация и композиция являются частными случаями ассоциации, выражающими отношение «часть-целое». Агрегация означает, что часть может существовать независимо от целого и обозначается пустым ромбом со стороны целого. Композиция же означает, что часть не может существовать без целого и обозначается закрашенным ромбом. Наследование, или обобщение, показывает, что один класс является подтипом другого, и изображается сплошной линией с пустым треугольником на конце, указывающим на родительский класс. Зависимость, изображаемая пунктирной линией с открытой стрелкой, показывает, что изменение одного элемента может повлиять на другой.

Диаграмма активностей

Диаграмма активностей относится к поведенческим диаграммам и используется для моделирования потока управления от одного действия к другому. Она похожа на классические блок-схемы, но обладает более богатой семантикой, поддерживая параллелизм, синхронизацию и сложные ветвления.

Основным элементом диаграммы активностей является действие, изображаемое в виде прямоугольника с закруглёнными углами. Действие представляет собой атомарную единицу работы, например вызов метода, присваивание переменной или отправку сообщения. Начальное состояние изображается в виде закрашенного кружка и показывает, с какого действия начинается выполнение. Конечное состояние изображается в виде закрашенного кружка внутри круга и отмечает завершение потока.

Узлы принятия решений, изображаемые в виде ромба, имеют один входящий поток и несколько исходящих, каждый из которых связан с определённым условием. Разделительная полоса, или вертикальная черта, используется для разделения потока на несколько параллельных ветвей, выполняющихся одновременно. Завершающая полоса объединяет несколько параллельных ветвей обратно в один поток. Синхронизация означает, что все входящие потоки должны быть завершены, прежде чем выполнение продолжится дальше.

Диаграммы активностей широко используются для моделирования бизнес-процессов, алгоритмов работы системы и сценариев использования на высоком уровне абстракции. Они также полезны для документирования сложной логики внутри одного варианта использования, где несколько участников системы взаимодействуют друг с другом.

Диаграммы взаимодействия. Последовательности

Диаграммы взаимодействия являются поведенческими диаграммами, которые показывают, как объекты взаимодействуют друг с другом во времени. Существует несколько видов диаграмм взаимодействия, из которых наиболее распространённой является диаграмма последовательности. Другие виды включают диаграмму коммуникации и диаграмму синхронизации.

Диаграмма последовательности изображает обмен сообщениями между объектами в хронологическом порядке. На диаграмме объекты располагаются горизонтально, каждый в своей головной части прямоугольником с именем объекта и класса. Вертикально от каждого объекта вниз идёт линия жизни, тонкая пунктирная или сплошная линия, обозначающая существование объекта во времени. Прямоугольники активности, расположенные поверх линии жизни, обозначают периоды, когда объект выполняет некоторую операцию или находится в состоянии выполнения.

Сообщения изображаются в виде стрелок от линии жизни одного объекта к линии жизни другого объекта. Стрелка обозначает передачу управления от отправителя к получателю. Пунктирная стрелка возврата обозначает возврат управления обратно, обычно вместе с возвращаемым значением. Сообщения могут быть синхронными, когда отправитель ждёт ответа, или асинхронными, когда отправитель продолжает выполнение, не дожидаясь ответа.

Диаграммы последовательности особенно полезны для детальной спецификации сценариев вариантов использования. Они показывают, какие объекты участвуют в сценарии, в каком порядке они вызывают методы друг у друга, и какие значения передаются. Такие диаграммы помогают выявлять пропущенные операции, оптимизировать взаимодействие и документировать протоколы обмена.

Диаграмма коммуникации решает похожую задачу, но акцент делает на связях между объектами, а не на временной последовательности. В диаграмме коммуникации сообщения нумеруются в порядке их отправки. Этот тип диаграмм полезен, когда важнее структурные отношения между участниками взаимодействия.

Диаграммы вариантов использования

Диаграмма вариантов использования является поведенческой диаграммой, которая описывает функциональные требования к системе с точки зрения её пользователей. В отличие от диаграмм последовательности или активностей, диаграмма вариантов использования показывает не то, как система работает внутри, а то, что именно она должна делать. Это самый высокоуровневый вид UML-диаграмм, предназначенный в первую очередь для общения с заказчиком и бизнес-аналитиками.

Вариант использования, или use case, изображается в виде эллипса с коротким глагольным словосочетанием внутри, например «Заказать товар» или «Зарегистрироваться в системе». Каждый вариант использования описывает законченную и ценную для пользователя функциональность, которую система предоставляет.

Актор, или actor, изображается в виде стилизованного человечка и представляет собой роль пользователя или внешней системы, которая взаимодействует с системой. Акторы всегда находятся вне системы, то есть за её границами. Один физический человек может играть несколько ролей, и наоборот, роль может быть реализована разными людьми. Например, в банковской системе роль «Клиент» и роль «Сотрудник банка» будут разными акторами, хотя один человек не может одновременно быть клиентом и сотрудником.

Отношения между акторами и вариантами использования обозначаются простыми линиями, соединяющими их. Эти линии показывают, что актор инициирует вариант использования или участвует в нём. Варианты использования могут быть связаны между собой отношениями включения и расширения. Включение показывает, что один вариант использования всегда вызывает другой, и изображается пунктирной стрелкой со стереотипом include. Расширение показывает, что один вариант использования может дополнить другой при определённых условиях, и изображается пунктирной стрелкой со стереотипом extend. Например, вариант использования «Войти в систему» может включать вариант «Проверить учётные данные», и никогда не выполняется без него. А вариант использования «Оплатить заказ» может расширять вариант «Выбрать способ оплаты» при использовании определённого платёжного шлюза.

Диаграмма вариантов использования служит контрактом между заказчиком и разработчиком. По ней можно оценить объём функциональности системы, выделить приоритеты и составить план разработки. Поскольку она не содержит деталей реализации, её легко понимают не технические специалисты, что делает её идеальным инструментом для первого этапа сбора требований.

Обзор CASE-средств для построения диаграмм UML

CASE-средства, или Computer Aided Software Engineering, представляют собой программные инструменты, предназначенные для автоматизации процессов анализа, проектирования и разработки программного обеспечения. В контексте UML, CASE-средства позволяют создавать, редактировать и поддерживать диаграммы различных типов, а также выполнять прямое и обратное проектирование. На рынке существует множество CASE-средств, которые можно классифицировать по функциональности, модели распространения и платформе.

К крупным профессиональным CASE-средствам относятся продукты, которые долгое время были стандартом в промышленной разработке. IBM

Rational Rose, хотя сегодня считается устаревшим, сыграл ключевую роль в популяризации UML и поддержке циклического проектирования. Он позволял строить все типы диаграмм UML и генерировать код на различных языках, включая C++ и Java. Более современным и мощным средством является Enterprise Architect от Sparx Systems. Этот инструмент поддерживает UML 2.5, а также смежные нотации, такие как BPMN, SysML и ArchiMate. Enterprise Architect предоставляет возможности для управления требованиями, отслеживания изменений, генерации документации и работы в команде через репозиторий. Другим представителем этого класса является Visual Paradigm, который сочетает в себе мощный редактор UML с поддержкой Agile-методологий, включая инструменты для пользовательских историй и управления бэклогом.

Среди легковесных и бесплатных инструментов особенно популярен StarUML. Это десктопное приложение с открытым API, позволяющее расширять функциональность с помощью JavaScript и Node.js. StarUML поддерживает все основные типы диаграмм UML 2.x и современный интерфейс. Ещё одним известным инструментом является Dia, который прост и бесплатен, но поддерживает лишь ограниченный набор UML-диаграмм и менее удобен для сложных моделей. Для пользователей ОС Linux также доступны такие средства, как ArgoUML и Umbrello, хотя по функциональности они уступают коммерческим продуктам.

Отдельную нишу занимают инструменты с текстовым описанием диаграмм. PlantUML является лидером в этой категории. Он позволяет описывать диаграммы с помощью простого текстового языка. Например, с помощью текста можно описать классы, методы и связи между ними, а PlantUML сгенерирует соответствующий графический файл. Текстовый подход имеет ряд преимуществ. Диаграммы могут храниться в системе контроля версий наравне с кодом, они легче поддаются ревью и автоматической обработке. PlantUML интегрируется с различными средами разработки, документационными системами и CI/CD-конвейерами. Недостатком является более крутая кривая обучения по сравнению с графическими редакторами.

В последние годы значительную популярность приобрели онлайн-инструменты для совместной работы. draw.io, в настоящее время известный как diagrams.net, является бесплатным онлайн-редактором с поддержкой UML-диаграмм, интеграцией с облачными хранилищами и возможностью командной работы. Lucidchart – это более функциональный коммерческий продукт, ориентированный на корпоративных пользователей. Он предлагает широкий набор шаблонов, интеграции с популярными сервисами и развитые средства для совместного редактирования. Такие инструменты удобны тем, что не требуют установки, обеспечивают облачную синхронизацию и поддерживают реальное взаимодействие нескольких пользователей над одной диаграммой.

Следует упомянуть и программные средства, встроенные в среды разработки. Например, IntelliJ IDEA и другие IDE от JetBrains имеют встроенные средства визуализации и редактирования UML-диаграмм, хотя их функциональность обычно ограничена диаграммами классов и последовательности. Такие инструменты удобны тем, что всегда синхронизированы с кодом и не требуют импорта или экспорта.

При выборе CASE-средства для учебных или профессиональных целей следует руководствоваться несколькими критериями. Поддержка требуемых типов диаграмм: достаточно ли инструмент покрывает те диаграммы, которые планируется использовать. Возможность командной работы, если проект ведёт группа разработчиков. Простота освоения: легковесные инструменты удобнее для обучения, но уступают в функциональности профессиональным. Интеграция со средой разработки: возможность прямого и обратного проектирования может быть критичной. И наконец, стоимость и модель лицензирования: существуют как полностью бесплатные решения, так и дорогие корпоративные пакеты. Для образовательных целей часто рекомендуются StarUML или draw.io как достаточно мощные и при этом бесплатные, в то время как PlantUML полезен для привития навыков проектирования через текстовое описание моделей.

ТЕМА 3. ШАБЛОНЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО АНАЛИЗА И ПРОЕКТИРОВАНИЯ, ИХ КЛАССИФИКАЦИЯ

Принцип классификации шаблонов проектирования

Классификация шаблонов проектирования, предложенная «бандой четырёх» (Gang of Four, GoF) – Эрихом Гаммой, Ричардом Хелмом, Ральфом Джонсоном и Джоном Влиссидесом в их книге «Design Patterns: Elements of Reusable Object-Oriented Software», является наиболее признанной и широко используемой. Эта классификация разделяет двадцать три шаблона на три основные группы в зависимости от их назначения и решаемой задачи. Ключевым принципом этой классификации является выделение трёх аспектов проектирования: создания объектов, построения структур и определения поведения.

Также существует дополнительная классификация шаблонов, делящая их на шаблоны уровня класса и шаблоны уровня объекта. Различие между ними определяется основным механизмом, используемым для достижения гибкости: наследованием или композицией. Шаблоны уровня класса опираются на отношения наследования между классами, что позволяет переопределять поведение на уровне всей иерархии, но делает связи статическими, заданными на этапе компиляции. В отличие от них, шаблоны уровня объекта используют механизмы композиции и делегирования, что позволяет динамически изменять структуру и поведение системы во время выполнения, обеспечивая большую гибкость.

Третьим измерением классификации может быть масштаб решаемой задачи: от шаблонов, работающих на уровне отдельных классов и объектов, до архитектурных шаблонов, определяющих структуру целых приложений.

Что такое типовое решение проектирования

В основе понятия шаблона проектирования лежит концепция типового решения. Образец, или паттерн, представляет собой проверенный и документированный способ решения типовой, часто встречающейся задачи проектирования в определённом контексте, с учётом действующих в системе ограничений и набором требований к решению. Это не готовый фрагмент кода, а общая схема или «каркас» решения, который необходимо адаптировать под конкретную ситуацию.

Шаблон проектирования задаёт структуру взаимодействующих компонентов программной системы и решает типичную проблему организации связей между ними. Ключевой характеристикой такого типового решения является его многократная применимость. Использование шаблонов позволяет избежать «изобретения велосипеда», снизить количество ошибок, повысить

читаемость и поддерживаемость кода, а также ускорить процесс разработки за счёт применения проверенных идей.

Форматы описания паттернов проектирования

Для того чтобы паттерном можно было эффективно пользоваться, его описание должно быть формализовано и стандартизировано. Наиболее известным и влиятельным является формат описания, введённый «бандой четырёх». Каждый паттерн в книге GoF имеет одинаковую структуру, что облегчает его изучение, сравнение и применение. Это описание включает в себя несколько обязательных разделов. Ключевым элементом является определение контекста, в котором существует проблема, и действующих в этом контексте сил, то есть противоречивых требований, которые необходимо сбалансировать.

Формат GoF включает имя и классификацию, дающую уникальный идентификатор, позволяющий разработчикам ссылаться на него, а также определяющую его принадлежность к одной из трёх основных категорий. В разделе «Назначение» кратко описывается, какую именно проблему решает паттерн и какова его цель. Далее следует подробное описание проблемы: ситуации, в которой применение паттерна будет оправданным. Также вводится общее описание решения, описывающего структуру паттерна в виде диаграммы классов и абстрактного кода. Обязательным является раздел с последствиями, анализирующим плюсы и минусы применения паттерна, а также влияние на гибкость, расширяемость и переносимость системы. В дополнение к этому приводится известный пример реализации и описание связей паттерна с другими паттернами проектирования.

Паттерны проектирования в нотации языка UML

Язык UML является основным инструментом для визуализации и документирования паттернов проектирования. Поскольку паттерн описывает отношения между классами и объектами, диаграмма классов UML становится идеальным и наиболее распространённым способом его графического представления. Нотация UML позволяет наглядно показать статическую структуру паттерна: какие классы в нём участвуют, какие у них атрибуты и операции, и, самое главное, как они связаны друг с другом.

На UML-диаграмме классов, описывающей паттерн, обычно выделяют роли, которые играют в нём различные классы. Например, в паттерне «Стратегия» (Strategy) будут показаны абстрактный класс стратегии с виртуальным методом algorithm, конкретные классы-стратегии, реализующие этот метод, и класс-контекст, который содержит ссылку на объект стратегии. При этом связи между классами изображаются в соответствии со стандартной нотацией UML. Стрелка с пустым треугольником обозначает наследование, а

линия с ромбом – композицию или агрегацию. Использование UML позволяет сделать абстрактное описание паттерна предельно наглядным, показать на диаграмме последовательности динамику взаимодействия объектов, а также служить «чертежом» для генерации кода.

Использование UML делает описание паттерна стандартизированным и понятным для всех разработчиков, независимо от языка программирования, который они используют.

Отношения, связи и альтернативы применения паттернов

Понимание связей между паттернами и умение выбирать альтернативные решения – признак высокой квалификации проектировщика. Паттерны редко используются изолированно. В реальных проектах они взаимодействуют и комбинируются. Некоторые паттерны являются альтернативой друг другу, другие дополняют друг друга, а третьи могут быть реализованы на основе четвёртых, более простых.

Ключевой принцип объектного проектирования, который тесно связан с выбором паттернов, гласит: «предпочитайте композицию наследованию» (Favor Composition over Inheritance). Многие паттерны, особенно структурные, такие как «Мост» (Bridge) или «Декоратор» (Decorator), были созданы именно для того, чтобы предоставить альтернативу жёсткому наследованию и сделать систему более гибкой. В то время как наследование создаёт статическую связь на уровне классов, композиция и делегирование позволяют динамически изменять поведение объекта. Именно поэтому шаблоны уровня класса (в основном, порождающие) используют наследование, а большинство структурных и поведенческих шаблонов уровня объекта основываются на композиции, чтобы изменять поведение во время выполнения. Например, паттерн «Стратегия» полностью построен на композиции: контекст содержит ссылку на объект стратегии и делегирует ему выполнение алгоритма.

Отношения между паттернами могут быть иерархическими. Более крупные и сложные паттерны, например, архитектурные, часто строятся на основе более простых фундаментальных паттернов. Понимание этих связей позволяет разработчику видеть общую картину и не изобретать велосипед для каждой подзадачи, а выстраивать элегантные, согласованные системы из проверенных строительных блоков.

ТЕМА 4. ПОРОЖДАЮЩИЕ ШАБЛОНЫ

Обзор применения порождающих типовых решений проектирования

Порождающие шаблоны проектирования представляют собой одну из трёх основных категорий, описанных «бандой четырёх». Их основная цель заключается в абстрагировании процесса создания объектов. Вместо того чтобы создавать объекты непосредственно с помощью оператора new, что приводит к жёсткой привязке к конкретным классам, порождающие шаблоны предлагают гибкие механизмы, которые позволяют выбирать тип создаваемого объекта во время выполнения, скрывать сложность создания, контролировать количество экземпляров и повторно использовать существующие объекты. Применение порождающих шаблонов особенно актуально в системах, где требуется независимость от того, как именно создаются, компонуются и представляются объекты. Они помогают сделать систему менее зависимой от конкретных классов, повышают её гибкость и облегчают сопровождение, поскольку изменения в логике создания локализуются внутри шаблона, а не размазываются по всему коду. Порождающие шаблоны можно разделить на два подтипа. Шаблоны уровня класса используют наследование для изменения создаваемого класса, например фабричный метод. Шаблоны уровня объекта используют композицию и делегирование, как абстрактная фабрика, строитель или прототип, что даёт большую гибкость во время выполнения.

Фабричный метод

Фабричный метод, также известный как виртуальный конструктор, является порождающим шаблоном уровня класса. Он определяет интерфейс для создания объекта, но позволяет подклассам решать, какой именно класс инстанциировать. Иными словами, фабричный метод делегирует создание объекта наследникам родительского класса. Этот шаблон применяется в ситуациях, когда классу заранее неизвестно, объекты каких подклассов ему нужно создавать, или когда класс спроектирован так, что создаваемые им объекты должны определяться подклассами.

Структура фабричного метода включает создателя, который объявляет абстрактный фабричный метод, возвращающий объект типа продукта. Конкретные создатели переопределяют этот метод, возвращая экземпляр конкретного продукта. Ключевое свойство фабричного метода заключается в том, что он полагается на наследование: создание объектов делегируется подклассам. Таким образом, код создателя остаётся независящим от конкретных классов продуктов. Достоинствами фабричного метода являются устранение жёсткой привязки к конкретным классам продуктов, упрощение добавления новых типов продуктов без изменения существующего кода создателя, а также

реализация принципа открытости-закрытости. Недостатком является то, что для каждого нового типа продукта, возможно, потребуется создать новый подкласс создателя, что может привести к разрастанию иерархии.

Абстрактная фабрика

Абстрактная фабрика – это порождающий шаблон уровня объекта, который предоставляет интерфейс для создания семейств связанных или взаимозависимых объектов без спецификации их конкретных классов. В отличие от фабричного метода, который создаёт объекты одного типа, абстрактная фабрика создаёт набор объектов, которые должны работать вместе. Этот шаблон полезен, когда система должна быть независимой от способа создания, компоновки и представления входящих в неё продуктов, или когда необходимо обеспечить использование только одного из нескольких семейств продуктов.

Структура абстрактной фабрики включает абстрактную фабрику, объявляющую интерфейсы для создания каждого типа продукта из семейства, и конкретные фабрики, реализующие эти интерфейсы для создания продуктов конкретных семейств. Также существуют абстрактные объявления продуктов и их конкретные реализации. Клиент работает только с абстрактными интерфейсами и не знает, какие конкретные фабрики и продукты используются. Это позволяет легко заменять целые семейства продуктов, изменяя всего один объект фабрики. Достоинства абстрактной фабрики включают гарантию совместимости создаваемых продуктов, поскольку фабрика создаёт только согласованные между собой объекты, а также изоляцию клиента от конкретных классов продуктов. Недостатком является сложность расширения фабрики для поддержки новых типов продуктов: добавление нового продукта требует изменения интерфейса абстрактной фабрики и всех её конкретных реализаций.

Строитель

Шаблон строитель отделяет конструирование сложного объекта от его представления, позволяя одному и тому же процессу конструирования создавать различные представления. Он используется для пошагового создания сложных объектов, особенно когда объект может иметь множество различных конфигураций или когда процесс создания состоит из множества этапов. Типичный пример – построение сложного документа, парсера или графического объекта.

Структура строителя включает строителя, который объявляет интерфейс для построения частей сложного объекта, и конкретных строителей, реализующих этот интерфейс и собирающих отдельные части в итоговый продукт. Директор управляет процессом конструирования, вызывая методы строителя в определённой последовательности. Клиент создаёт директора и

нужного строителя, после чего запускает процесс конструирования и получает готовый объект. Основное достоинство строителя заключается в возможности изменять внутреннее представление продукта без изменения кода директора, а также в детальном контроле над процессом конструирования. Шаблон позволяет повторно использовать один и тот же процесс для создания разных представлений. Недостатком является усложнение кода за счёт введения дополнительных классов, особенно если объект не слишком сложен.

Прототип

Прототип – это порождающий шаблон, который создаёт новые объекты путём копирования существующего объекта-прототипа, вместо того чтобы создавать экземпляр класса через конструктор. Этот шаблон полезен, когда создание объекта с помощью конструктора является дорогостоящим или сложным, когда необходимо избежать жёсткой зависимости от конкретных классов, или когда система должна быть независимой от того, как её продукты создаются, компонуются и представляются.

Структура прототипа включает абстрактный прототип, объявляющий метод клонирования, и конкретные прототипы, реализующие этот метод, создавая копию самого себя. Клиент создаёт новые объекты, вызывая метод clone на существующем объекте-прототипе вместо использования оператора new. Различают поверхностное и глубокое копирование. Поверхностное копирование копирует только сам объект, но не объекты, на которые ссылаются его поля. Глубокое копирование рекурсивно копирует все вложенные объекты. Достоинствами прототипа являются сокрытие сложности создания объектов от клиента, возможность добавлять и удалять классы продуктов во время выполнения путём изменения набора прототипов, а также снижение количества подклассов. Недостатки связаны с реализацией глубокого копирования, которая может быть сложной, особенно при наличии циклических ссылок, а также с тем, что некоторые языки могут не поддерживать клонирование для всех типов объектов.

Одиночка

Одиночка, или синглтон, гарантирует, что класс имеет только один экземпляр, и предоставляет глобальную точку доступа к этому экземпляру. Этот шаблон используется для доступа к ресурсам, которые должны быть единственными в системе, например к конфигурации приложения, пулу соединений с базой данных, менеджеру печати или реестру. Одиночка также позволяет управлять доступом к общему ресурсу и избегать конфликтов.

Типичная реализация одиночки включает приватный конструктор, статическую переменную, хранящую единственный экземпляр, и статический

метод, возвращающий этот экземпляр, создавая его при первом вызове. В многопоточной среде необходима потокобезопасная реализация, например с использованием двойной проверки блокировки или статической инициализации. Достоинства одиночки включают контролируемый доступ к единственному экземпляру, сокращение глобального пространства имён и возможность уточнения операций инстанцирования. Однако одиночка имеет серьёзные недостатки. Он нарушает принцип единственной ответственности, поскольку отвечает как за свою основную функцию, так и за управление своим жизненным циклом. Одиночка создаёт скрытые зависимости, затрудняет тестирование, поскольку трудно изолировать классы, использующие одиночку, и заменять его заглушкой. В больших системах одиночка может приводить к трудноустранимым ошибкам, связанным с порядком инициализации. Многие современные практики рекомендуют использовать внедрение зависимостей вместо одиночки для управления общими ресурсами.

Обсуждение порождающих шаблонов проектирования

Каждый из рассмотренных порождающих шаблонов решает свою специфическую задачу, и выбор между ними зависит от конкретной ситуации. Фабричный метод является самым простым и использует наследование. Он подходит, когда есть одна иерархия создаваемых продуктов и требуется лишь небольшая гибкость. Абстрактная фабрика сложнее, но позволяет создавать целые семейства продуктов и легко переключаться между ними. Она удобна, когда система должна работать с несколькими вариантами конфигурации, например с разными графическими интерфейсами для разных операционных систем. Строитель применяется, когда объект сложен и его создание требует пошаговой сборки с возможностью изменения внутреннего представления. Прототип полезен, когда создание экземпляра дорого или когда классы неизвестны до выполнения. Одиночка – самый спорный шаблон: его применяют с осторожностью, когда действительно требуется строго один экземпляр, например для глобальной конфигурации или логирования, но часто от него стараются уходить в пользу более тестируемых решений.

Важно также различать, когда использовать фабричный метод, а когда абстрактную фабрику. Фабричный метод создаёт один продукт, но оставляет возможность подклассам переопределить его тип. Абстрактная фабрика создаёт семейство продуктов. Часто эти шаблоны комбинируются: абстрактная фабрика может быть реализована с помощью фабричных методов. Строитель и абстрактная фабрика могут показаться похожими, но их различие в том, что абстрактная фабрика возвращает готовый объект сразу, а строитель возвращает объект пошагово и позволяет менять процесс сборки. Прототип и фабричный метод оба создают объекты, но прототип делает это через клонирование, а не

через вызов конструктора, что может быть эффективнее, если клонирование дешевле создания.

Синергия нескольких порождающих шаблонов

В реальных сложных системах порождающие шаблоны редко используются по отдельности. Напротив, они часто комбинируются для достижения нужной гибкости. Один из классических примеров синергии – реализация абстрактной фабрики с помощью фабричных методов. Вместо того чтобы создавать продукты в конкретной фабрике напрямую, можно определить для каждого продукта отдельный фабричный метод, который переопределяют наследники. Это даёт дополнительную гибкость и позволяет переиспользовать общую логику.

Строитель часто сочетается с прототипом. Если процесс строительства создаёт объекты одинаковой структуры, но с разными значениями параметров, можно хранить прототипы уже собранных объектов и клонировать их, что ускоряет создание. Одиночка иногда используется совместно с абстрактной фабрикой. Например, в системе может быть только одна фабрика для текущей платформы, и к ней нужен глобальный доступ. Тогда фабрику реализуют как одиночку. Фабричный метод может использовать прототип для создания объектов, если клонирование оказывается эффективнее инстанцирования.

На уровне фреймворков и библиотек порождающие шаблоны переплетаются. Фреймворк может предоставлять абстрактную фабрику для создания своих компонентов, реализовывать строитель для сложной конфигурации, а конкретные фабрики внутри фреймворка могут быть одиночками. Понимание синергии шаблонов позволяет проектировать более элегантные и гибкие системы, где каждый шаблон отвечает за свой аспект создания объектов, а вместе они образуют целостный механизм. Проектировщик должен не просто знать каждый шаблон по отдельности, но и уметь комбинировать их, выбирая оптимальное сочетание под конкретную задачу.

ТЕМА 5. СТРУКТУРНЫЕ ШАБЛОНЫ

Обзор применения структурных решений проектирования

Структурные шаблоны проектирования составляют вторую из трёх основных категорий по классификации «банды четырёх». Если порождающие шаблоны отвечают за гибкое создание объектов, то структурные шаблоны решают задачи гибкой компоновки классов и объектов в более крупные структуры. Они позволяют строить сложные системы из отдельных компонентов, обеспечивая при этом возможность изменения связей между ними без переписывания всего кода. Основная идея заключается в том, чтобы найти простой способ выразить отношения между сущностями, скрыть сложность, преобразовать один интерфейс в другой, добавить новые обязанности или оптимизировать использование памяти.

Структурные шаблоны делятся на два подтипа. Шаблоны уровня класса используют наследование для объединения классов, например, шаблон адаптер при работе с классами. Шаблоны уровня объекта используют композицию и делегирование для объединения объектов, что обеспечивает большую гибкость во время выполнения. К этой категории относятся большинство структурных паттернов: адаптер, мост, компоновщик, декоратор, фасад, заместитель и приспособленец. Применение структурных шаблонов позволяет создавать системы, где компоненты слабо связаны, легко модифицируются, тестируются и переиспользуются. Они помогают соблюдать принцип открытости-закрытости, когда новые функции добавляются через расширение, а не через изменение существующего кода.

Адаптер

Шаблон адаптер, также известный как обёртка, предназначен для преобразования интерфейса одного класса в интерфейс, ожидаемый клиентом. Адаптер позволяет классам с несовместимыми интерфейсами работать вместе, выступая в роли посредника. Этот шаблон особенно полезен при интеграции legacy-кода, использовании сторонних библиотек или когда необходимо унифицировать работу с несколькими классами, имеющими похожую функциональность, но разные протоколы вызова. Существует две разновидности адаптера. Адаптер объекта использует композицию: адаптер содержит ссылку на адаптируемый объект и делегирует ему выполнение операций, преобразуя вызовы клиента в формат, понятный адаптируемому классу. Адаптер класса использует множественное наследование, когда адаптер наследует одновременно и целевой интерфейс, и адаптируемый класс. На практике адаптер объекта встречается чаще из-за гибкости и отсутствия проблем множественного наследования. Достоинства адаптера заключаются в возможности использовать

существующие классы с неподходящими интерфейсами без их изменения, а также в отделении интерфейса клиента от конкретных реализаций. Недостатком является некоторое усложнение системы за счёт введения дополнительных классов, особенно если требуется множество адаптеров для разных типов.

Мост

Шаблон мост разделяет один большой класс или группу тесно связанных классов на две отдельные иерархии: абстракцию и реализацию. Это позволяет изменять и расширять каждую иерархию независимо друг от друга. Мост решает проблему, когда класс необходимо расширять в двух независимых измерениях, например, разные типы окон и разные операционные системы, или разные виды устройств и разные пульты дистанционного управления. Вместо создания комбинаторного взрыва подклассов для каждого варианта, мост предлагает отделить абстракцию, которая содержит высокоуровневую логику, от реализации, которая выполняет конкретные действия. Абстракция содержит ссылку на объект реализации и делегирует ей все вызовы, связанные с платформозависимыми операциями. Клиент работает только с абстракцией. Когда нужно добавить новый вид абстракции, создаётся новый подкласс абстракции; когда нужно добавить новую реализацию, создаётся новый подкласс реализации. Преимущества моста включают возможность комбинировать разные абстракции с разными реализациями во время выполнения, уменьшение количества подклассов и упрощение расширения системы. Недостатком является усложнение кода, поскольку в систему вводятся дополнительные уровни косвенности, и проектировщик должен заранее определить, какие аспекты системы являются независимыми измерениями.

Компоновщик

Шаблон компоновщик позволяет сгруппировать объекты в древовидную структуру для представления иерархии «часть-целое». Он даёт клиенту возможность единообразно работать как с отдельными объектами, так и с их композициями. Классический пример – графические примитивы, где точка, отрезок и группа фигур должны поддерживать одни и те же операции, такие как рисование, перемещение или масштабирование. Компоновщик определяет единый интерфейс для всех компонентов дерева. Листовые узлы реализуют этот интерфейс непосредственно для простых элементов. Составные узлы, или контейнеры, содержат список дочерних компонентов и делегируют выполнение операций им, а также могут добавлять, удалять и получать дочерние элементы. Клиент работает с компонентами через общий интерфейс и не знает, является ли конкретный компонент листом или контейнером. Это упрощает клиентский код и позволяет строить произвольно сложные иерархии. Достоинства

компоновщика включают единообразную обработку отдельных и составных объектов, упрощение добавления новых типов компонентов и поддержку рекурсивных структур. Недостатком является то, что интерфейс компонента должен быть достаточно общим, чтобы подходить и для листьев, и для контейнеров, что иногда требует определения методов, которые не имеют смысла для листьев, например добавление дочернего элемента. Эту проблему можно решить с помощью предоставления реализации по умолчанию, выбрасывающей исключение.

Декоратор

Шаблон декоратор позволяет динамически добавлять объектам новые обязанности без изменения их исходного кода. Декоратор является гибкой альтернативой наследованию, поскольку наследование добавляет функциональность статически во время компиляции и для всех экземпляров класса одновременно, а декоратор позволяет навешивать дополнительные поведения на отдельные объекты во время выполнения. Декоратор реализует тот же базовый интерфейс, что и декорируемый объект, и содержит ссылку на него. При вызове операции декоратор может выполнить дополнительные действия до или после делегирования вызова вложенному объекту. Декораторы можно вкладывать друг в друга, комбинируя несколько расширений. Классический пример – графические компоненты, где к ним можно добавить рамку, тень, прокрутку и другие эффекты. То же применимо к потокам ввода-вывода, где различные декораторы добавляют буферизацию, сжатие или шифрование. Достоинства декоратора включают бóльшую гибкость по сравнению с наследованием, возможность добавлять ответственность во время выполнения и устранять её, а также отсутствие взрывного роста количества подклассов. Недостатки: код может стать перегруженным множеством мелких классов-декораторов, особенно если их много, а также сложность конфигурации, когда декораторы накладываются в несколько слоёв. Кроме того, декоратор может усложнить отладку, поскольку поток вызовов проходит через цепочку объектов.

Фасад

Шаблон фасад предоставляет унифицированный высокоуровневый интерфейс к сложной подсистеме, состоящей из множества классов, компонентов или библиотек. Фасад скрывает сложность подсистемы, упрощает её использование для клиента и снижает количество зависимостей между клиентским кодом и внутренними компонентами. При этом фасад не ограничивает доступ к полной функциональности подсистемы: клиент, если ему нужны специфические возможности, может обращаться к низкоуровневым классам напрямую. Фасад часто используется на этапе проектирования для

разбиения системы на слои и создания точки входа в каждый слой. Типичный пример – компилятор, где фасад предоставляет один метод compile, который внутри запускает лексический анализ, синтаксический анализ, семантический анализ, генерацию кода и оптимизацию. Клиенту не нужно знать об этих внутренних шагах. Другой пример – библиотека для работы с видеоформатами, которая через фасад предоставляет простые методы для открытия, конвертации и сохранения видео, скрывая сотни классов для обработки разных кодеков и контейнеров. Достоинства фасада заключаются в уменьшении связности между клиентом и подсистемой, упрощении клиентского кода, упрощении тестирования и возможности локально изменять подсистему без влияния на клиента. Недостатком может стать то, что фасад становится «божественным объектом», знающим слишком много о внутренностях подсистемы, что нарушает принцип единственной ответственности.

Заместитель

Шаблон заместитель, или прокси, предоставляет суррогатный объект, который контролирует доступ к другому объекту, называемому реальным субъектом. Заместитель имеет тот же интерфейс, что и реальный объект, поэтому клиент не замечает подмены. Существует несколько разновидностей заместителя, каждая из которых решает свою задачу. Виртуальный заместитель создаёт реальный объект только тогда, когда он действительно понадобится, экономя ресурсы при работе с тяжёлыми объектами, например большими изображениями или трёхмерными моделями. Защитный заместитель проверяет права доступа и решает, можно ли клиенту выполнить ту или иную операцию над реальным объектом. Удалённый заместитель скрывает тот факт, что объект находится в другом адресном пространстве или на другом компьютере, предоставляя локальный интерфейс для удалённого объекта, как в технологиях CORBA или Java RMI. Логирующий заместитель записывает вызовы методов для отладки или аудита. Кеширующий заместитель сохраняет результаты дорогостоящих операций и возвращает их при повторных вызовах. Достоинства заместителя включают разделение ответственности, возможность добавлять новые виды контроля без изменения реального объекта, а также реализацию ленивой загрузки и отложенных вычислений. Недостатки связаны с усложнением системы из-за дополнительных классов и возможным снижением производительности, особенно если заместитель выполняет много проверок или синхронизации.

Приспособленец

Шаблон приспособленец, или легковес, служит для эффективной поддержки большого количества мелких объектов за счёт разделения общего

состояния между ними. Приспособленец применим, когда программа создаёт огромное количество похожих объектов, которые потребляют много памяти, причём большую часть их состояния можно вынести за пределы объектов и сделать общим. Классический пример – редактор, отображающий текстовый документ. Каждый символ может быть представлен объектом, но вместо создания миллиона объектов для каждого символа, можно создать один объект для каждого символа в кодировке, а в документе хранить только позиции и ссылки на общие объекты-приспособленцы. Состояние приспособленца делится на внутреннее, которое является общим для многих объектов и не меняется, например код символа и шрифт, и внешнее, которое зависит от контекста и хранится вне приспособленца, например координаты на странице. Фабрика приспособленцев управляет пулом общих объектов и возвращает существующий объект, если он уже был создан. Достоинства приспособленца очевидны: значительная экономия памяти при большом количестве однородных объектов. Недостатки: усложнение кода, необходимость разделять состояние на внутреннее и внешнее, а также возможное снижение производительности, если часто приходится конвертировать внешнее состояние при вызовах методов.

Обсуждение структурных паттернов проектирования

Все структурные паттерны так или иначе используют механизм композиции и делегирования для достижения гибкости. Их различие заключается в решаемых задачах. Адаптер нужен, когда интерфейсы несовместимы. Мост нужен, когда необходимо разделить абстракцию и реализацию. Компоновщик позволяет работать с древовидными структурами единообразно. Декоратор динамически добавляет обязанности. Фасад упрощает сложную подсистему. Заместитель контролирует доступ. Приспособленец экономит память за счёт разделения состояния.

При выборе структурного паттерна важно понимать, какую именно проблему нужно решить. Если проблема в несовместимости интерфейсов – адаптер. Если требуется независимо развивать абстракцию и реализацию – мост. Если нужно представить иерархию «часть-целое» – компоновщик. Если нужно добавлять поведение без создания множества подклассов – декоратор. Если нужно скрыть сложность – фасад. Если нужен контроль доступа или ленивая инициализация – заместитель. Если нужно сэкономить память при большом количестве одинаковых объектов – приспособленец.

Часто структурные паттерны комбинируются. Например, фасад может скрывать систему, внутри которой используется мост для разделения абстракции и реализации. Декоратор может быть реализован с помощью компоновщика, когда декоратор оборачивает целую группу компонентов. Заместитель часто сочетается с приспособленцем, чтобы управлять созданием и разделением тяжёлых объектов. Понимание этих связей позволяет проектировать

масштабируемые, гибкие и эффективные системы, где каждый паттерн выполняет свою чёткую роль, а вместе они обеспечивают элегантную архитектуру.

ТЕМА 6. ПОВЕДЕНЧЕСКИЕ ШАБЛОНЫ

Обзор применения поведенческих решений проектирования

Поведенческие шаблоны проектирования – это третья большая группа паттернов, описанная «бандой четырёх». Если порождающие шаблоны отвечают за создание объектов, а структурные – за их компоновку, то поведенческие паттерны решают задачи эффективного распределения обязанностей между объектами и алгоритмизации взаимодействия. Они описывают типовые способы организации связей между объектами, позволяя легко изменять поведение системы, не переписывая её код, и управлять сложными потоками управления. Поведенческие паттерны фокусируются не на статической структуре, а на динамике: как объекты общаются друг с другом, как передают запросы, как синхронизируют состояние и как инкапсулируют алгоритмы. Основные преимущества применения поведенческих шаблонов заключаются в повышении гибкости, снижении связанности и улучшении повторного использования кода. Они особенно полезны в системах, где поведение может меняться во время выполнения, где требуется обработка событий, реализация конечных автоматов или сложная координация множества объектов. Поведенческие паттерны делятся на две подгруппы: шаблоны уровня класса, использующие наследование для распределения поведения, и шаблоны уровня объекта, использующие композицию и делегирование. Большинство поведенческих паттернов относятся ко второй группе, что делает их более гибкими.

Цепочка обязанностей

Шаблон цепочка обязанностей позволяет передавать запрос последовательно по цепочке обработчиков, пока один из них не обработает запрос. Каждый обработчик решает, обработать запрос самостоятельно или передать его следующему в цепочке. Это даёт возможность избежать жёсткой привязки отправителя запроса к его получателю, а также динамически изменять состав и порядок обработчиков. Типичный пример – системы контекстной помощи, где запрос на справку передаётся от элемента интерфейса к его контейнеру, затем к окну и так до глобального обработчика. Другой пример – фильтры обработки данных в веб-приложениях. Участники цепочки обязанностей включают абстрактный обработчик, объявляющий метод обработки и ссылку на следующий обработчик. Конкретные обработчики реализуют логику: либо обрабатывают запрос, либо передают дальше. Клиент

строит цепочку, связывая обработчики в нужном порядке. Достоинства шаблона заключаются в уменьшении связности, гибкости перестройки цепочки во время выполнения и соблюдении принципа единственной ответственности. Недостатком является то, что запрос может остаться необработанным, если ни один обработчик его не принял.

Команда

Шаблон команда превращает запрос в самостоятельный объект, содержащий всю информацию для его выполнения. Это позволяет параметризовать объекты запросами, ставить запросы в очередь, логировать их, поддерживать отмену операций. Команда инкапсулирует действие и его параметры, отделяя объект, инициирующий операцию, от объекта, который её выполняет. Классический пример – реализация отмены действий в текстовом редакторе. Каждая операция вставки или удаления текста упаковывается в команду, которая знает, как выполнить действие и как его отменить. Команда также используется в графических интерфейсах для связывания кнопок с действиями. Структура команды включает интерфейс команды с методом execute, конкретные команды, реализующие этот интерфейс и содержащие ссылку на получателя – объект, который реально выполняет действие. Инициатор, например кнопка или пункт меню, вызывает execute у команды. Клиент создаёт команды и связывает их с инициаторами. Достоинства команды: возможность отмены и повтора операций, постановка команд в очередь и логирование, отделение инициации операции от её реализации, а также поддержка макрокоманд. Недостатки – увеличение количества классов и возможное усложнение кода.

Интерпретатор

Шаблон интерпретатор определяет грамматику некоторого языка и предоставляет интерпретатор для её разбора и выполнения. Он используется для решения повторяющихся задач, которые можно описать формальным языком, например для обработки регулярных выражений, выполнения математических формул, компиляции простых языков запросов. Шаблон применяется, когда грамматика языка относительно проста и её эффективность не является критическим фактором. Структура интерпретатора включает абстрактное выражение, объявляющее метод интерпретации, и набор конкретных выражений для каждого правила грамматики: терминальные выражения, соответствующие элементарным символам, и нетерминальные выражения, представляющие правила композиции. Контекст содержит глобальную информацию, необходимую при интерпретации. Клиент строит абстрактное синтаксическое дерево из выражений и вызывает метод интерпретации. Достоинства

интерпретатора: простота изменения и расширения грамматики, поскольку добавление нового правила требует лишь создания нового класса выражения. Недостатки: сложность поддержки сложных грамматик, когда количество классов выражений растёт, и потенциально низкая производительность. На практике интерпретатор часто заменяют более эффективными инструментами, такими как генераторы парсеров.

Итератор

Шаблон итератор предоставляет способ последовательного доступа ко всем элементам составного объекта, не раскрывая его внутреннего представления. Итератор выносит логику обхода коллекции из самой коллекции в отдельный объект, что позволяет использовать разные алгоритмы обхода для одной и той же коллекции и одновременно выполнять несколько обходов. Практически все современные языки программирования встроили итератор в свои стандартные библиотеки. В Java это интерфейсы Iterator и Iterable, в C – IEnumerable и IEnumerator, в Python – протокол итератора с методами iter и next. Структура итератора включает абстрактный итератор, объявляющий методы для перехода к следующему элементу и проверки наличия элементов, и конкретные итераторы, реализующие обход для определённых коллекций. Абстрактная коллекция объявляет метод создания итератора, а конкретные коллекции возвращают соответствующий итератор. Клиент работает с коллекцией через итератор, не зная её внутренней структуры. Достоинства итератора: унификация способа обхода, сокрытие внутреннего представления коллекции, возможность реализовывать различные стратегии обхода и упрощение кода клиента. Недостатков практически нет, кроме разве что небольшого оверинга при создании объектов-итераторов.

Медиатор

Шаблон медиатор, также известный как посредник, уменьшает связанность между множеством взаимодействующих объектов, вводя между ними объект-посредник, который координирует их взаимодействие. Вместо того чтобы объекты ссылались друг на друга напрямую, они общаются через медиатора, что делает их независимыми и легко переиспользуемыми. Классический пример – диалоговое окно со множеством элементов управления: кнопки, поля ввода, списки. Вместо того чтобы каждый элемент знал о всех остальных, все изменения передаются медиатору окна, который решает, как обновить другие элементы. Другой пример – система управления полётами, где диспетчер является медиатором между самолётами. Структура медиатора включает абстрактного медиатора, объявляющего интерфейс для отправки сообщений между компонентами. Конкретные медиаторы реализуют координацию, зная о всех

компонентах системы. Компоненты, или коллеги, хранят ссылку на медиатора и при изменении своего состояния уведомляют его. Достоинства медиатора: централизация управления, снижение связанности компонентов, упрощение их повторного использования и облегчение тестирования. Недостаток в том, что медиатор может стать «божественным объектом», накапливающим слишком много знаний о всей системе, что нарушает принцип единственной ответственности.

Хранитель

Шаблон хранитель, также известный как мементо, позволяет сохранять и восстанавливать предыдущее состояние объекта без раскрытия деталей его реализации. Это достигается путём вынесения состояния в отдельный объект-хранитель, который создаётся и читается только исходным объектом, но не раскрывает своё содержимое другим объектам. Типичное применение – поддержка отмены операций, сохранение контрольных точек, реализация транзакций. В отличие от команды, где отмена может быть реализована через обратное действие, хранитель напрямую фиксирует состояние. Структура хранителя включает объект-хозяин, который умеет создавать хранитель со своим текущим состоянием и восстанавливать состояние из хранителя. Хранитель является пассивным контейнером, который не предоставляет доступа к своим внутренним данным никому, кроме хозяина. Управляющий объект отвечает за хранение истории хранителей. Достоинства хранителя: полная инкапсуляция состояния объекта, простота реализации отката и возможность сохранять сложные состояния. Недостатки: затраты памяти на хранение истории, особенно если состояние велико, и потенциальные накладные расходы на сериализацию. Для экономии памяти может использоваться комбинация с командой, где хранится только разница между состояниями.

Наблюдатель (посредник в списке заменён наблюдателем, но у вас «посредник» уже есть – видимо, извините, я следую списку). В вашем списке значится «посредник» (видимо, синоним медиатора, уже рассмотрен). В классических поведенческих паттернах также есть наблюдатель, но он не упомянут. Я пропущу его и перейду к состоянию.

Состояние

Шаблон состояние позволяет объекту изменять своё поведение при изменении его внутреннего состояния, создавая впечатление, что объект изменил свой класс. Это достигается путём вынесения всех поведений, зависящих от состояния, в отдельные классы-состояния. Контекст хранит ссылку на текущий объект состояния и делегирует ему вызовы. При изменении состояния контекст заменяет объект состояния на другой. Типичный пример – конечный автомат для

заказа на сайте: состояния «новый», «оплачен», «отправлен», «доставлен», «отменён». Каждое состояние определяет, какие операции разрешены и к какому новому состоянию они приводят. В отличие от реализации конечного автомата с помощью условных операторов в одном классе, шаблон состояние изолирует поведение каждого состояния в собственный класс, что упрощает добавление новых состояний и делает код более понятным. Структура состояния включает абстрактное состояние, объявляющее методы для всех операций, которые могут зависеть от состояния, и конкретные состояния, реализующие поведение для каждого конкретного состояния. Контекст хранит текущее состояние и делегирует ему вызовы. Достоинства: локализация поведения каждого состояния, упрощение добавления новых состояний и устранение громоздких условных конструкций. Недостатки – увеличение числа классов и некоторое усложнение, если состояний мало и поведение простое.

Стратегия

Шаблон стратегия определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми. Стратегия позволяет изменять алгоритм независимо от клиентов, которые его используют. Это избавляет от необходимости использовать условные операторы и множественное наследование. Типичный пример – сортировка массива с возможностью выбора алгоритма: быстрая сортировка, сортировка слиянием, пузырьковая сортировка. Другой пример – алгоритмы навигации: пешеходный маршрут, автомобильный, велосипедный. Клиент выбирает нужную стратегию и подставляет её в контекст. Контекст хранит ссылку на объект стратегии и делегирует ему выполнение алгоритма. Конкретные стратегии реализуют разные варианты алгоритма. Достоинства стратегии: замена наследования композицией, избавление от условных операторов, возможность выбора алгоритма во время выполнения. Недостаток – клиент должен знать, какие стратегии существуют, и сам выбирать подходящую, а также увеличение числа классов. Стратегия тесно связана с принципом открытости-закрытости: новые алгоритмы добавляются без изменения существующего кода контекста.

Фабричный метод – замечание. Этот шаблон, как уже сказано, относится к порождающим, а не поведенческим. Его цель – создание объектов. Включение его в список поведенческих – распространённая ошибка. В поведенческих паттернах существует шаблонный метод, который часто путают с фабричным. Шаблонный метод определяет скелет алгоритма в методе базового класса, оставляя реализацию некоторых шагов подклассам. Он является поведенческим. Возможно, в исходном списке имелся в виду именно шаблонный метод. Если же речь о фабричном методе, то он используется для делегирования создания объектов подклассам и обсуждался в теме порождающих шаблонов.

Посетитель

Шаблон посетитель позволяет добавить новую операцию к существующей иерархии классов, не изменяя сами эти классы. Это достигается за счёт двойной диспетчеризации: операция инкапсулируется в отдельном классе-посетителе, который имеет метод для каждого типа элементов в иерархии. Элементы реализуют метод accept, принимающий посетителя и вызывающий соответствующий метод посетителя. Эта техника особенно полезна, когда нужно выполнить разнородные операции над объектами сложной структуры, например компилятор, который обрабатывает абстрактное синтаксическое дерево: посещающие методы могут выполнять проверку типов, генерацию кода, оптимизацию и т.д. Недостатком является то, что добавление нового элемента в иерархию требует добавления метода во всех существующих посетителей. Кроме того, посетитель может нарушить инкапсуляцию, так как ему часто нужен доступ к внутренним данным элементов. Достоинства: упрощение добавления новых операций, объединение родственных операций в одном классе, независимость иерархии элементов от операций над ними. Посетитель часто комбинируется с итератором для обхода сложных структур.

Обсуждение поведенческих шаблонов проектирования

Поведенческие паттерны решают разные аспекты взаимодействия объектов. Цепочка обязанностей полезна для обработки событий с несколькими обработчиками. Команда – для параметризации действий и отмены. Интерпретатор – для разбора языков. Итератор – для единообразного обхода коллекций. Медиатор – для централизации сложных взаимодействий. Хранитель – для сохранения состояний. Состояние – для конечных автоматов. Стратегия – для взаимозаменяемых алгоритмов. Посетитель – для операций над иерархиями. Каждый из них имеет свои сценарии применения, сильные и слабые стороны. Выбор правильного поведенческого паттерна требует анализа конкретной задачи и того, какой аспект поведения нужно сделать гибким. Важно помнить, что эти паттерны не конкурируют, а дополняют друг друга. Например, команда может хранить состояние в хранителе для отмены. Медиатор может использовать стратегию для координации компонентов. Итератор может быть реализован с помощью команды для отложенного выполнения. Знание поведенческих паттернов позволяет разработчику мыслить на более высоком уровне абстракции, строить системы, которые легко расширять и поддерживать, и эффективно решать типовые проблемы взаимодействия объектов.

ТЕМА 7. МЕТОДИКА ПРИМЕНЕНИЯ ТИПОВЫХ РЕШЕНИЙ ПРОЕКТИРОВАНИЯ

Пример паттерна проектирования: Модель-вид-контроллер

Паттерн «Модель-вид-контроллер», или MVC, является одним из самых известных архитектурных паттернов, который часто относят к группе поведенческих или архитектурных шаблонов. Он разделяет интерактивное приложение на три взаимосвязанных компонента, каждый из которых отвечает за свою часть функциональности. Модель представляет собой данные и бизнес-логику приложения. Она ничего не знает о пользовательском интерфейсе и может использоваться независимо от него. Модель хранит состояние приложения, реализует бизнес-правила и предоставляет методы для доступа к данным и их изменения. При изменении модели она уведомляет наблюдателей, обычно через механизм событий или паттерн наблюдатель. Вид отвечает за визуальное представление данных модели. Он отображает информацию пользователю и реагирует на изменения модели, обновляя интерфейс. Вид не содержит бизнес-логики и не выполняет обработку пользовательского ввода – он только показывает данные и предоставляет элементы управления, такие как кнопки и поля ввода. Контроллер обрабатывает входящие запросы от пользователя, например нажатия кнопок или переходы по ссылкам. Он интерпретирует действия пользователя, взаимодействует с моделью для изменения данных и выбирает вид для отображения результата. Контроллер служит связующим звеном между видом и моделью. MVC даёт множество преимуществ. Разделение ответственности упрощает сопровождение и тестирование: модель можно тестировать без интерфейса, вид можно изменять, не затрагивая логику. Также поддерживается множественное представление одних и тех же данных – можно иметь несколько видов для одной модели, например табличное и графическое представление. Этот паттерн лёг в основу многих веб-фреймворков, таких как Ruby on Rails, Django, Spring MVC, ASP.NET MVC, и широко используется в разработке графических интерфейсов.

Как выбирать паттерн проектирования

Выбор подходящего паттерна проектирования требует анализа контекста, выявления проблем и понимания целей системы. Не существует универсального правила «какой паттерн лучше», но есть ряд рекомендаций, помогающих принять решение. Прежде всего, необходимо чётко сформулировать проблему. Часто бывает достаточно понять, что именно требуется сделать: создать объекты гибко, скомпоновать классы или организовать взаимодействие. Если проблема связана с созданием объектов, стоит обратиться к порождающим паттернам. Если нужно изменить интерфейс или структуру классов – к структурным. Если требуется

перераспределить обязанности или алгоритмы – к поведенческим. Затем следует изучить, какие паттерны уже известны разработчикам. Использование знакомого паттерна упрощает коммуникацию и сопровождение. Важно рассмотреть, какую именно гибкость необходимо обеспечить. Например, если требуется, чтобы подсистема могла использовать разные алгоритмы, подойдёт стратегия. Если нужно, чтобы система могла обрабатывать запросы с возможностью отмены, поможет команда. Если требуется единый доступ к сложной подсистеме – фасад. Также полезно обратить внимание на то, как паттерн соотносится с принципами объектно-ориентированного проектирования, такими как единственная ответственность, открытость-закрытость, подстановка Лисков, разделение интерфейсов и инверсия зависимостей. Паттерн, который нарушает эти принципы в конкретном контексте, может быть неудачным выбором. Наконец, важно не злоупотреблять паттернами. Иногда простое решение без паттерна оказывается эффективнее и понятнее, чем излишне усложнённая структура с несколькими паттернами. Следует применять паттерн только тогда, когда его преимущества перевешивают сложность, которую он вносит.

Поиск подходящих объектов

Поиск подходящих объектов – это начальный этап объектно-ориентированного анализа и проектирования, предшествующий выбору паттерна и определению связей. Он заключается в выделении из описания предметной области существительных, которые могут стать классами или объектами системы. Обычно объекты соответствуют реальным сущностям, с которыми работает приложение: заказ, клиент, товар, счёт. Также объектами могут быть акторы, выполняющие действия, или служебные сущности, такие как менеджеры, фасады, посредники. При поиске объектов важно учитывать не только те сущности, которые станут хранить данные, но и те, которые будут выполнять бизнес-логику или управлять взаимодействием. Хорошим советом является искать объекты, которые: имеют состояние, требуют сохранения; имеют поведение, которое меняется в зависимости от условий; участвуют в отношениях с другими объектами. При этом следует избегать слишком мелкой гранулярности. Например, для поля «адрес» не обязательно создавать отдельный объект, если адрес никогда не используется вне контекста клиента. Однако если адрес имеет сложную структуру или обрабатывается отдельно, стоит вынести его в отдельный класс. Поиск объектов также тесно связан с выявлением вариантов использования: каждый сценарий использования может подсказать, какие объекты в нём участвуют. В этом процессе помогают диаграммы вариантов использования и последовательности, которые визуализируют взаимодействие объектов. После того как кандидаты в объекты определены, их группируют в классы, уточняют атрибуты и операции, а также связи между ними. Именно на этом этапе часто становится понятно, какой паттерн проектирования применить:

если объекты должны создаваться динамически, поможет фабрика; если объекты образуют иерархию «часть-целое» – компоновщик; если поведение объекта зависит от состояния – паттерн состояние.

Определение степени детализации объекта

Определение степени детализации объекта является критическим решением, влияющим на сложность, производительность и удобство сопровождения системы. Степень детализации определяет, насколько мелко разбита предметная область на классы и объекты. Слишком крупные объекты, обладающие множеством свойств и методов, нарушают принцип единственной ответственности, становятся сложными в тестировании и изменении, а также часто приводят к дублированию кода. Слишком мелкие объекты порождают огромное количество классов, что усложняет понимание системы, увеличивает накладные расходы на создание объектов и взаимодействие между ними. Оптимальная степень детализации зависит от контекста. В базовых операциях, таких как работа с числами и строками, обычно не требуется создавать отдельные классы. В бизнес-логике выгодно выделять сущности, имеющие самостоятельное значение. При проектировании также полезно руководствоваться принципом, что класс должен иметь одну причину для изменения. Если класс меняется по нескольким независимым причинам, его следует разбить на более мелкие классы. Паттерны проектирования часто подсказывают правильную детализацию. Например, паттерн стратегия требует выделения каждого алгоритма в отдельный класс, что увеличивает детализацию, но приносит гибкость. Паттерн фасад, наоборот, скрывает детали за общим интерфейсом, снижая видимую детализацию для клиента. В итоге выбор степени детализации – это баланс между гибкостью и простотой, который приходит с опытом. При сомнениях часто рекомендуется начинать с более крупных объектов, а затем по мере необходимости проводить рефакторинг и выделять новые классы, руководствуясь принципами YAGNI (You Aren't Gonna Need It) и постепенного улучшения.

Специфицирование интерфейсов объекта

Под специфицированием интерфейсов объекта понимается точное определение набора операций, которые объект предоставляет другим компонентам, включая их сигнатуры (имена, типы параметров, возвращаемые значения) и семантику. Хорошо спроектированный интерфейс является контрактом между объектом и его клиентами. Он должен быть минимальным, но достаточным, чтобы обеспечивать все нужные сценарии использования. Важнейшим принципом специфицирования интерфейсов является принцип разделения интерфейсов, который гласит, что клиенты не должны зависеть от

методов, которые они не используют. Это означает, что вместо одного универсального интерфейса лучше иметь несколько узкоспециализированных. При специфицировании интерфейсов также следует учитывать, что они могут быть реализованы разными классами, поэтому в объектно-ориентированном проектировании часто используют абстрактные классы или интерфейсы языка для описания контракта. Хорошо определённый интерфейс облегчает использование паттернов проектирования. Например, паттерн адаптер требует, чтобы целевой интерфейс был ясен, а адаптер реализовывал его. Паттерн стратегия опирается на общий интерфейс всех стратегий. Паттерн заместитель реализует тот же интерфейс, что и реальный объект. При специфицировании интерфейсов важно избегать излишней сложности: методы должны быть логически связаны, иметь понятные имена, не требовать от клиента знания внутреннего состояния объекта. Хорошей практикой является использование иммутабельных объектов там, где это возможно, и явное указание, какие операции могут изменять объект, а какие нет. В языке UML интерфейсы объектов отображаются на диаграммах классов как операции, часто с указанием видимости, а также могут быть выделены как отдельные элементы со стереотипом «interface».

Как пользоваться паттерном проектирования

Использование паттерна проектирования включает несколько этапов, от изучения до интеграции в код. Первым шагом является глубокое понимание проблемы и контекста. Необходимо убедиться, что паттерн соответствует текущей ситуации, а не применяется «для галочки». Следует ознакомиться с известным описанием паттерна: его назначением, структурой, участниками, последствиями, примерами реализации. Затем нужно адаптировать паттерн к конкретным условиям. Паттерны дают шаблон, а не готовый код. Необходимо определить, какие классы будут играть роли, описанные в паттерне, какие методы и атрибуты им понадобятся, какие отношения связывают участников. При этом можно изменять детали паттерна, например добавлять дополнительные методы или упрощать структуру, если это не нарушает основного замысла. Далее следует написать код, реализующий структуру паттерна, с использованием выбранного языка программирования. Важно не забыть про тестирование: паттерны упрощают тестирование, но их неправильная реализация может внести ошибки. После реализации нужно оценить, действительно ли паттерн принёс ожидаемую пользу: снизил связанность, повысил гибкость, упростил добавление новых функций. Если паттерн усложнил систему без весомых причин, возможно, от него стоит отказаться. Паттерны также полезно документировать, указывая, почему выбрано именно это решение. Это поможет другим разработчикам понять архитектуру. При использовании нескольких паттернов в одной системе важно следить за тем, чтобы они не конфликтовали и не создавали излишнего

косвенного взаимодействия. Наконец, следует помнить, что паттерны – это не догма, а инструмент. Опытный разработчик знает, когда паттерн нужен, а когда лучше ограничиться простым решением. Постоянная рефлексия и рефакторинг помогут поддерживать чистоту архитектуры. Практика изучения и применения паттернов показывает, что их эффективное использование приходит с опытом, а начинать лучше с простых, хорошо понятных паттернов, таких как фабричный метод, адаптер или стратегия, постепенно переходя к более сложным.

ТЕМА 8. ВИДЫ И ПРИМЕРЫ ШАБЛОНОВ, НЕ ЗАВИСЯЩИХ ОТ ЯЗЫКА ПРОГРАММИРОВАНИЯ. ВИДЫ И ПРИМЕРЫ ШАБЛОНОВ, СПЕЦИФИЧНЫХ ДЛЯ ЯЗЫКА ПРОГРАММИРОВАНИЯ

Шаблоны проектирования принято разделять по уровню абстракции и степени привязки к конкретному языку или технологии. Одни шаблоны являются универсальными и могут быть реализованы практически на любом языке, поддерживающем объектно-ориентированный подход. Другие же, которые часто называют идиомами, тесно связаны с возможностями конкретного языка и используют его уникальные конструкции. К первым относятся архитектурные шаблоны, конструктивные шаблоны, шаблоны интеграции и антишаблоны. Ко вторым – идиомы, которые представляют собой типовые способы использования языковых средств для решения повторяющихся задач.

Архитектурные шаблоны

Архитектурные шаблоны представляют собой самый высокий уровень абстракции. Они определяют фундаментальную структуру программной системы в целом, включая крупные компоненты, их обязанности и способы взаимодействия. В отличие от шаблонов проектирования, которые решают локальные задачи на уровне нескольких классов, архитектурные шаблоны задают каркас всего приложения. Классическим примером является шаблон «Модель-вид-контроллер», или MVC, который разделяет приложение на три взаимосвязанные части, что позволяет изменять представление данных без изменения бизнес-логики. Другим известным архитектурным шаблоном является «Многоуровневая архитектура», или Layered Architecture, где система разбивается на горизонтальные слои, такие как уровень представления, уровень бизнес-логики, уровень доступа к данным. Каждый слой предоставляет сервисы вышележащему слою и использует сервисы нижележащего. Этот шаблон реализован во многих корпоративных приложениях. Архитектурный шаблон «Микросервисы» предполагает разбиение приложения на набор небольших, независимо развёртываемых сервисов, каждый из которых реализует одну

бизнес-возможность. Этот шаблон стал популярным с развитием облачных технологий и DevOps. «Событийно-ориентированная архитектура», или EDA, основана на асинхронной передаче событий между компонентами, что обеспечивает высокую масштабируемость и слабую связность. Архитектурные шаблоны не зависят от языка реализации; они могут быть воплощены на Java, C, Python, Go или любом другом языке, поддерживающем необходимые средства коммуникации и компоновки.

Конструктивные шаблоны

Термин «конструктивные шаблоны» может использоваться в двух значениях. В широком смысле он охватывает классические шаблоны проектирования, описанные «бандой четырёх», которые часто называют конструктивными, поскольку они предлагают готовые «кирпичики» для построения гибких и поддерживаемых структур. Эти шаблоны делятся на порождающие, структурные и поведенческие, и они не привязаны к какому-то конкретному языку. Фабричный метод, абстрактная фабрика, строитель, одиночка, адаптер, мост, компоновщик, декоратор, фасад, заместитель, приспособленец, стратегия, команда, наблюдатель, состояние, посетитель – все они могут быть реализованы на любом объектно-ориентированном языке. В более узком смысле «конструктивные шаблоны» иногда противопоставляют «архитектурным» как шаблоны, которые решают задачи на уровне отдельных классов и объектов, но не на уровне всей системы. В любом случае, эти шаблоны являются языконезависимыми. Их реализация может отличаться деталями, например, в языке с поддержкой множественного наследования паттерн адаптер может быть реализован проще, а в языке без такой поддержки придётся использовать композицию. Однако концептуальная структура остаётся неизменной. Знание конструктивных шаблонов позволяет разработчикам быстро находить элегантные решения типовых проблем, таких как создание объектов, компоновка интерфейсов или организация взаимодействия, независимо от выбранного языка.

Шаблоны интеграции

Шаблоны интеграции решают задачи взаимодействия между различными программными системами, компонентами или сервисами, особенно в гетерогенных средах. Эти шаблоны не зависят от конкретного языка, так как описывают способы обмена сообщениями, преобразования данных, маршрутизации и управления интеграционными потоками. Наиболее известным источником является книга Грегора Хоупа и Бобби Вульфа «Шаблоны интеграции корпоративных приложений». Среди них выделяется шаблон «Канал сообщения», который обеспечивает передачу данных от отправителя к

получателю через абстрактный канал. «Маршрутизатор сообщений» анализирует содержание сообщения и направляет его в разные каналы в зависимости от заданных правил. «Преобразователь сообщений» изменяет формат или структуру сообщения, чтобы получатель мог его обработать. «Шлюз» скрывает детали взаимодействия с внешней системой за простым интерфейсом. «Коррелятор» связывает запросы с ответами в асинхронных сценариях. Эти шаблоны широко применяются при построении интеграционных шин, микросервисных архитектур и систем обмена данными между унаследованными и современными приложениями. Независимость от языка здесь особенно важна, поскольку интегрируемые системы часто написаны на разных языках. Реализация шаблонов интеграции может использовать очереди сообщений, брокеры, веб-сервисы, REST API, потоковые платформы или другие технологии, но сама структура шаблона остаётся неизменной.

Антишаблоны

Антишаблоны – это шаблоны, но не хороших решений, а типичных ошибок или плохих практик, которые приводят к проблемам, таким как трудно поддерживаемый код, низкая производительность, ненадёжность или сложность внесения изменений. Антишаблоны так же полезны, как и правильные шаблоны, поскольку помогают распознавать и избегать распространённых ловушек. Многие антишаблоны не зависят от языка программирования. «Божественный объект» – это класс, который знает слишком много и делает слишком много, нарушая принцип единственной ответственности. «Золотой молоток» – привычка применять один и тот же инструмент, например паттерн, ко всем задачам, даже когда он не подходит. «Лапша из кода» описывает код с запутанной структурой и плохой организацией. «Магические числа» – использование числовых констант без пояснения их смысла. «Копирование-вставка программирование» приводит к дублированию кода и ошибкам при изменении одной копии. «Узкое место в производительности» – неоптимальное место в коде, которое замедляет всю систему. «Аналитический паралич» – чрезмерный анализ и проектирование без начала реализации. Существуют также антишаблоны, более специфичные для определённых технологий, например «Spaghetti for Breakfast» в многопоточном программировании. Знание антишаблонов помогает при рефакторинге: обнаружив признаки антишаблона, разработчик может применить соответствующий паттерн для его устранения. Антишаблоны также полезны при код-ревью и обучении, так как позволяют называть и обсуждать проблемы стандартизированным способом.

Идиомы

Идиомы – это шаблоны, специфичные для конкретного языка программирования. Они представляют собой типовые, повторяющиеся конструкции, которые используют сильные стороны языка и его особенности для элегантного решения задач, не имеющих универсального представления в других языках. Идиомы часто бывают очень короткими и могут касаться даже одного-двух выражений. В отличие от архитектурных и конструктивных шаблонов, идиомы нельзя механически перенести на другой язык, потому что они опираются на специфические механизмы конкретного языка: управление памятью, систему типов, макросы, метапрограммирование и т.д. Например, в языке C идиома RAII, Resource Acquisition Is Initialization, связывает время жизни ресурса с временем жизни объекта, используя конструктор для захвата ресурса и деструктор для его освобождения. Эта идиома основана на детерминированном вызове деструкторов при выходе объекта из области видимости. В Java или C, где сборщик мусора не гарантирует немедленного вызова финализаторов, RAII в чистом виде не работает, поэтому приходится использовать конструкцию try-with-resources или using. Другой пример идиомы – «Потокобезопасная ленивая инициализация с двойной проверкой блокировки», которая в Java требует аккуратного использования volatile, а в C более сложна. Идиомы также включают типовые способы обработки ошибок, итерации по контейнерам, перегрузку операторов, использование замыканий, метапрограммирование на шаблонах и многое другое.

Пример идиомы C++. RAII

Идиома RAII, Resource Acquisition Is Initialization, является одной из наиболее важных идиом в языке C++. Её суть заключается в том, что управление ресурсами, такими как динамическая память, файловые дескрипторы, сокеты, мьютексы, должно быть инкапсулировано в классах. Ресурс захватывается в конструкторе объекта, а освобождается в деструкторе. Благодаря тому, что деструктор автоматически вызывается, когда объект выходит из области видимости, ресурс гарантированно освобождается даже при возникновении исключений. Это позволяет избежать утечек памяти и других ресурсов. Примером является стандартный класс std::unique_ptr, который владеет динамически выделенной памятью и освобождает её в своём деструкторе. Когда умный указатель выходит из области видимости, он автоматически вызывает delete. Другой пример – std::lock_guard, который захватывает мьютекс в конструкторе и освобождает его в деструкторе, обеспечивая безопасную работу с критическими секциями. RAII породила ряд сопутствующих идиом, таких как правило трёх, правило пяти, копирование при записи. Эта идиома настолько мощна, что многие C-разработчики, переходящие на C++, отмечают её как одну

из главных причин уменьшения количества ошибок управления памятью. RAII не может быть реализована в языках со сборкой мусора и без детерминированных деструкторов в том же виде, что демонстрирует её языковую специфичность.

Другие примеры идиом в разных языках. В Java распространена идиома try-with-resources, которая автоматически закрывает ресурсы, реализующие интерфейс AutoCloseable. В Python существует контекстный менеджер, использующий оператор with и методы enter и exit. В функциональных языках часто встречается идиома конвейера, или pipe, когда результат одной функции передаётся в следующую. В JavaScript популярна идиома асинхронных функций async/await, которая превращает асинхронный код в похожий на синхронный. В Ruby идиома блока позволяет передавать фрагмент кода в метод. Идиомы делают код кратким, выразительным и идиоматичным, то есть соответствующим стилю и соглашениям сообщества.

Преимущества разработки программ с применением шаблонов

Использование шаблонов проектирования, как общих (языконезависимых), так и идиом, даёт разработчикам множество преимуществ. Во-первых, сокращается время разработки. Вместо того чтобы изобретать решение проблемы с нуля, разработчик использует проверенный и документированный шаблон. Во-вторых, повышается качество кода. Шаблоны, особенно идиомы, учитывают тонкости языка и подводные камни, что позволяет избежать распространённых ошибок. В-третьих, улучшается читаемость и поддерживаемость. Когда разработчик видит в коде знакомый шаблон, он быстро понимает, какую задачу решает эта часть программы, не вникая в детали реализации. Это особенно важно в больших командах. В-четвёртых, шаблоны облегчают коммуникацию между разработчиками. Название шаблона служит кратким и точным термином, который понятен всем членам команды: «давай используем стратегию для этого алгоритма» или «применим фасад для сложной подсистемы». В-пятых, шаблоны способствуют повторному использованию. Хотя шаблон не является готовой библиотекой, он даёт схему, которую можно адаптировать в разных проектах. В-шестых, многие шаблоны проектирования упрощают тестирование, так как выделяют отдельные аспекты системы в независимые классы, которые можно тестировать изолированно. Наконец, использование шаблонов и идиом помогает новичкам быстрее освоиться в проекте и в языке. Знание стандартных идиом является признаком профессионализма и позволяет писать не просто работающий, а элегантный и эффективный код. Важно, однако, помнить, что шаблоны – это не панацея. Неуместное применение шаблона может привести к излишней сложности и снижению производительности. Поэтому хороший разработчик должен знать не только шаблоны, но и критерии их применимости, а также быть готовым отказаться от шаблона, если простое решение оказывается уместнее.

ТЕМА 9. ОСНОВЫ КОМПЬЮТЕРНОЙ ГРАФИКИ

Понятие о графическом процессе

Графический процесс в компьютерной графике представляет собой последовательность действий по созданию, обработке, хранению и отображению визуальной информации с помощью компьютера. Этот процесс начинается с формирования математического описания объекта или сцены, которое может быть создано вручную, получено с помощью измерительных приборов или синтезировано алгоритмически. Затем это описание преобразуется в форму, пригодную для вывода на экран или печать. Ключевыми этапами графического процесса являются геометрическое моделирование, расчёт освещения и теней, растеризация, сглаживание и вывод изображения. В трёхмерной графике добавляются этапы трансформации вершин, проецирования и текстурирования. Графический процесс может быть интерактивным, когда пользователь управляет сценой в реальном времени, или пакетным, когда изображение рассчитывается заранее, что позволяет достичь более высокого качества, но требует больше времени.

Понятие о геометрическом моделировании

Геометрическое моделирование – это раздел компьютерной графики, занимающийся математическим описанием формы объектов. Модель представляет собой совокупность геометрических данных, которые определяют форму, размеры, положение и взаимное расположение объектов в пространстве. Геометрическое моделирование оперирует такими понятиями, как точки, линии, поверхности и тела. В основе лежат два основных подхода: каркасное моделирование, где объект описывается набором рёбер и вершин, и поверхностное моделирование, где объект представляется совокупностью граней. Наиболее совершенным является твердотельное моделирование, которое описывает не только поверхность, но и внутреннюю структуру объекта, позволяя вычислять массу, объём, моменты инерции и другие физические характеристики. Геометрическое моделирование широко используется в системах автоматизированного проектирования, в производстве, в архитектуре, в компьютерной анимации и в медицинской визуализации.

Типы моделей, особенности их получения

Существует три основных типа геометрических моделей: каркасные, поверхностные и твердотельные. Каркасная модель является самой простой и

представляет собой совокупность точек и соединяющих их отрезков или кривых. Она быстро строится и требует мало памяти, но не позволяет отличить внутреннюю область от внешней и не поддерживает операции вычитания объёмов. Поверхностная модель описывает объект через его грани. Она позволяет получить реалистичное изображение, но всё ещё не даёт информации о внутренней структуре. Твёрдотельная модель является наиболее полной: она определяет не только поверхность, но и внутреннее содержимое объекта, что позволяет выполнять булевы операции, вычислять объём и центры масс. Особенности получения моделей зависят от метода. Прямое моделирование осуществляется интерактивным построением примитивов или сплайнов. Параметрическое моделирование использует размерные зависимости и ограничения. Сканирование реальных объектов с помощью 3D-сканера даёт полигональную сетку. Также модели могут быть получены из томографических данных или синтезированы процедурными алгоритмами.

Основные направления развития и области применения компьютерной графики

Компьютерная графика развивается по нескольким основным направлениям. Это реалистичная визуализация, которая стремится к фотореализму через моделирование глобальной освещённости, трассировку лучей и физически корректные материалы. Это интерактивная графика, ориентированная на достижение высокой частоты кадров для игр, симуляторов и виртуальной реальности. Это научная визуализация, которая превращает большие массивы данных в наглядные образы. Это медицинская визуализация, позволяющая строить трёхмерные модели органов по данным компьютерной томографии. Это компьютерная анимация, используемая в кино и рекламе. Это системы автоматизированного проектирования и геоинформационные системы. Области применения включают промышленность, архитектуру, образование, медицину, киноиндустрию, видеоигры, военное дело, картографию, полиграфию, веб-дизайн и мобильные приложения. С развитием аппаратного обеспечения, особенно графических процессоров, появились новые направления, такие как рендеринг в реальном времени с трассировкой лучей, генерация изображений с помощью нейронных сетей и дополненная реальность.

Классификация графических примитивов, их атрибуты, используемые в компьютерной графике

Графический примитив – это простейший геометрический объект, который может быть отображён графической системой. К основным примитивам относятся точка, отрезок прямой, ломаная линия, прямоугольник, многоугольник, окружность, эллипс, дуга, сектор, а также залитые области и

текст. В трёхмерной графике примитивами являются вершина, ребро, грань, треугольник, четырёхугольник и полигональная сетка. Каждый примитив обладает набором атрибутов, которые определяют его внешний вид. К атрибутам относятся цвет заливки, цвет и толщина контура, тип линии (сплошная, пунктирная, штрихпунктирная), стиль заливки (сплошная, градиентная, текстурная или узорная), прозрачность, порядок отрисовки и трансформации (масштаб, поворот, сдвиг). В современных системах примитивы могут иметь дополнительные атрибуты, такие как сглаживание, тени, свечение, анимация. Набор доступных примитивов и атрибутов зависит от графического API или библиотеки. Стандартные примитивы определены в таких системах, как OpenGL, DirectX, GDI, Cairo, HTML Canvas.

Виды компьютерной графики – растровая, векторная, 3D-графика

Компьютерная графика делится на три основных вида. Растровая графика представляет изображение в виде двумерной матрицы пикселей, каждый из которых имеет свой цвет. Этот формат идеален для фотографий, живописи, сложных текстур и изображений с плавными переходами. Векторная графика описывает изображение с помощью геометрических примитивов: линий, кривых, многоугольников и заливок. Она используется для логотипов, шрифтов, чертежей, диаграмм, элементов интерфейса. Трёхмерная графика работает с трёхмерными моделями, которые проходят этапы моделирования, анимации, расчёта освещения и визуализации (рендеринга). 3D-графика используется в кино, играх, архитектурной визуализации, тренажёрах, медицинской визуализации. Существуют также гибридные подходы, например 2.5D-графика, где используется трёхмерное описание, но камера зафиксирована. Каждый вид имеет свои сильные и слабые стороны, и на практике часто применяется комбинация: векторы для контуров, растр для текстур и трехмерные модели для объектов.

Принципы представления растровой и векторной информации

В основе растровой графики лежит дискретизация: непрерывное изображение разбивается на мелкие элементы – пиксели. Каждый пиксель кодируется одним или несколькими байтами, определяющими его цвет. Чем выше разрешение, тем больше пикселей и тем более детализированным выглядит изображение. Форматы растровой графики делятся на форматы со сжатием без потерь, например PNG, TIFF, BMP, и со сжатием с потерями, например JPEG, где часть информации отбрасывается ради уменьшения размера файла. Принцип векторной графики иной: изображение описывается математическими формулами. Линия задаётся координатами концов, кривая Безье – опорными точками и управляющими рычагами, прямоугольник – координатами углов.

Векторное представление компактно, не зависит от разрешения и легко масштабируется. Программа-рендерер преобразует векторное описание в растровое изображение для вывода на экран или печать в зависимости от текущего масштаба и параметров устройства.

Особенности изображения в экранном представлении и полиграфии

Изображение на экране монитора формируется из пикселей, каждый из которых состоит из трёх субпикселей красного, зелёного и синего цвета. Экранное разрешение измеряется в абсолютных пикселях, например 1920×1080. Пиксели на экране имеют переменный физический размер в зависимости от диагонали монитора и его разрешения. Для экранной графики принято разрешение 72 или 96 точек на дюйм, но это условность, так как важен именно размер в пикселях. В полиграфии изображение воспроизводится на бумаге с помощью красок, чаще всего CMYK. Печатное разрешение измеряется в количестве точек на дюйм, которые способен воспроизвести принтер или печатная машина. Для качественной печати требуется разрешение от 300 точек на дюйм, а для газетной печати достаточно 150. Также различается цветовой охват: экраны воспроизводят яркие насыщенные цвета в модели RGB, при этом бумага обычно не может передать столь же широкий спектр, особенно ярко-синие и ярко-зелёные оттенки. Поэтому при подготовке изображения к печати необходимо выполнить цветоделение и иногда корректировать цвета, чтобы они выглядели адекватно на бумаге.

Разрешение

Разрешение в компьютерной графике – это величина, определяющая количество деталей на единицу длины или на всё изображение. Существует несколько видов разрешения. Пространственное разрешение, или абсолютное, – это размер изображения в пикселях по ширине и высоте. Оно определяет, сколько пикселей содержит изображение. Оптическое разрешение – характеристика сканера или камеры, показывающая, сколько светочувствительных элементов приходится на дюйм поверхности. Физическое разрешение монитора – это количество пикселей, реально отображаемых на экране. Пользовательское разрешение экрана – значение, установленное в операционной системе. Разрешение печати измеряется в точках на дюйм и показывает, сколько мелких точек краски наносит принтер на один дюйм бумаги. Разрешение растрового изображения при печати определяется делением размера изображения в пикселях на размер отпечатка в дюймах. Важно понимать, что высокое разрешение печати требует большого количества пикселей: для отпечатка формата A4 с качеством 300 DPI необходимо изображение размером примерно 2480×3508 пикселей. При

недостаточном разрешении изображение будет выглядеть пикселизированным или размытым.

Обзор, назначение и возможности пакетов векторной и растровой графики

Среди пакетов растровой графики лидирующее положение занимает Adobe Photoshop. Он предоставляет мощные инструменты для ретуши, коррекции цвета, создания коллажей, цифровой живописи, работы со слоями, масками, фильтрами, текстом и 3D-объектами. Бесплатной альтернативой является GIMP, который постепенно приближается к Photoshop по функциональности. Krita специализируется на цифровой живописи и анимации. Paint.NET – простой редактор для базовых задач. Corel Painter имитирует традиционные художественные техники. Пакеты векторной графики: Adobe Illustrator – промышленный стандарт для логотипов, шрифтов, иллюстраций. CorelDRAW популярен в полиграфии и у дизайнеров наружной рекламы. Inkscape – мощный бесплатный векторный редактор с поддержкой формата SVG. Figma и Sketch ориентированы на дизайн интерфейсов и прототипирование. Каждый из этих пакетов поддерживает импорт и экспорт различных форматов, работу со слоями, масками, градиентами, текстом, а также интеграцию с другими программами.

Достоинства и недостатки векторной графики

Векторная графика обладает рядом неоспоримых достоинств. Масштабируемость: векторное изображение можно увеличивать до любых размеров без потери качества, поскольку программа пересчитывает кривые заново. Малый размер файлов: векторное описание занимает значительно меньше места, чем растровое изображение аналогичной сложности, особенно для чёрно-белых чертежей и схем. Легкость редактирования: достаточно переместить опорную точку или изменить атрибут, и вся форма обновится. Независимость от разрешения печати: векторный файл можно напечатать на любом устройстве с наилучшим возможным качеством. К недостаткам векторной графики относят ограниченную реалистичность: с помощью векторов сложно передать плавные цветовые переходы, полутона и фотореализм. Сложность получения фотографического качества: хотя существуют векторные иллюстрации с градиентами и текстурами, они всё равно отличаются от фото. Также векторы не очень подходят для изображений с огромным количеством мелких деталей. Некоторые программы работают с векторной графикой медленнее при большом количестве сложных кривых.

Интеграция векторной и растровой графики

На практике векторная и растровая графика неразрывно связаны и часто используются вместе. Типичный рабочий процесс может включать создание векторного логотипа, который затем экспортируется в растровый формат для размещения на веб-сайте. Фотографии обрабатываются в растровом редакторе, а затем импортируются в векторную иллюстрацию как фоновое изображение или текстура. В программах верстки, таких как Adobe InDesign, могут одновременно присутствовать векторные и растровые объекты. При выводе на печать или в финальный файл все векторные объекты преобразуются в растровую форму на этапе рендеринга. Современные графические пакеты, например Photoshop, умеют создавать и редактировать векторные слои, а Illustrator поддерживает растровые изображения и даже имеет возможности их ретуши. Форматы, такие как PDF, могут содержать как векторные, так и растровые данные. Интеграция позволяет использовать сильные стороны каждого типа: векторы – для чётких контуров и масштабируемости, растр – для фотореалистичных деталей и текстур.

Устройства ввода и вывода графической информации

Устройства ввода предназначены для перевода графической информации в цифровой вид. К ним относятся сканеры, которые построчно считывают изображение с бумажного носителя. Графические планшеты, или дигитайзеры, преобразуют движения пера пользователя в векторные линии. Цифровые фотоаппараты и видеокамеры фиксируют реальные сцены через объектив и матрицу. 3D-сканеры создают трёхмерные модели реальных объектов. Также к устройствам ввода можно отнести мышь и сенсорные экраны. Устройства вывода отображают цифровую графику для восприятия человеком. Мониторы на жидких кристаллах или органических светодиодах являются основным устройством вывода в интерактивном режиме. Принтеры печатают изображения на бумаге или других носителях: струйные – для фотографий и цветной печати, лазерные – для документов и чёрно-белой графики. Плоттеры предназначены для широкоформатной печати чертежей, плакатов и баннеров. Цифровые проекторы выводят изображение на большой экран. Каждое устройство имеет свои характеристики, влияющие на качество графики: разрешение, глубина цвета, точность цветопередачи, скорость работы.

ТЕМА 10. ПОНЯТИЕ ЦВЕТА И ЕГО ПРЕДСТАВЛЕНИЕ В КОМПЬЮТЕРНОЙ ГРАФИКЕ

Определение цвета

Цвет является сложным и многогранным явлением, которое описывается одновременно физическими, физиологическими и психологическими понятиями. С физической точки зрения цвет – это свойство света вызывать определённое зрительное ощущение в соответствии со спектральным составом электромагнитного излучения. Свет представляет собой электромагнитные волны, и человеческий глаз воспринимает лишь узкую область этого спектра – видимый диапазон, приблизительно от 380 до 780 нанометров. Однако один и тот же спектральный состав может восприниматься по-разному в зависимости от условий наблюдения, а разный спектральный состав может давать идентичное цветовое ощущение. С психофизиологической точки зрения цвет – это субъективный образ, возникающий в мозге при обработке сигналов, поступающих от сетчатки глаза. Поэтому цвет не является неотъемлемым свойством объекта, а скорее результатом взаимодействия света, объекта и зрительной системы наблюдателя.

Особенности восприятия цвета

Восприятие цвета человеком начинается с сетчатки глаза, на которой расположены два типа фоторецепторов: палочки и колбочки. Палочки отвечают за восприятие яркости в условиях низкой освещённости и не различают цвет. Колбочки активны при дневном освещении и бывают трёх типов, каждый из которых чувствителен к определённому диапазону длин волн. Тип S чувствителен к коротким волнам, соответствующим синему цвету, тип M – к средним волнам, соответствующим зелёному, тип L – к длинным волнам, соответствующим красному. Эта особенность, называемая трихроматией, лежит в основе большинства цветовых моделей в компьютерной графике. Однако восприятие цвета не сводится только к работе трёх типов колбочек. Важную роль играют процессы обработки сигнала на уровне сетчатки и зрительных центров мозга, где возникают оппонентные механизмы, согласно которым цвет воспринимается по осям красный-зелёный, синий-жёлтый и чёрный-белый. Существует явление метамерии, когда два разных спектральных состава вызывают одинаковое цветовое ощущение. Именно это свойство используется в цветном телевидении и полиграфии: три основных цвета позволяют воспроизвести широкий диапазон оттенков. Также восприятие цвета зависит от окружения, цвета соседних объектов, освещения и индивидуальных особенностей наблюдателя, например, цветовой слепоты, или дальтонизма.

Цветовые модели

Цветовая модель представляет собой математический способ описания цвета с помощью нескольких чисел, называемых цветовыми координатами. RGB является аддитивной моделью, где цвет получается сложением красного, зелёного и синего излучения. Эта модель используется в мониторах, телевизорах, проекторах и любых устройствах, излучающих свет. Канал может иметь разрядность от 8 бит до 16 и более. В модели RGB чёрный цвет соответствует нулевым значениям всех трёх компонент, а белый – максимальным. Модель CMYK является субтрактивной и применяется в полиграфии. Она описывает цвет через количество голубой, пурпурной, жёлтой и чёрной красок, наносимых на бумагу. Буква K обозначает ключевой цвет, чёрный. Смешение CMY теоретически даёт чёрный, но на практике из-за неидеальности красителей получается грязно-коричневый, поэтому добавляется отдельный чёрный компонент. Модель HSB, или HSV, основана на интуитивно понятных характеристиках: цветовой тон, насыщенность и яркость. Эта модель удобна для подбора цвета человеком, так как соответствует естественному восприятию. Модель CIE XYZ была разработана Международной комиссией по освещению в 1931 году как эталонное, аппаратно-независимое цветовое пространство. Она основана на результатах экспериментов по цветовому соответствию и включает все цвета, видимые человеком. Позднее на её основе были разработаны более совершенные модели. Модель CIE Lab является перцепционно-равномерным цветовым пространством, в котором одинаковые евклидовы расстояния между точками примерно соответствуют одинаковой разнице в восприятии цвета. Ось L обозначает светлоту, ось a – переход от зелёного к красному, ось b – переход от синего к жёлтому. Lab широко используется в системах управления цветом. Существуют и другие цветовые модели, например YUV и YCbCr для видеосигналов, а также модель Grayscale для чёрно-белых изображений.

Характеристики цвета

Глубина цвета, или битность, определяет количество бит, выделяемых на хранение информации о цвете одного пикселя. Она может быть распределена по каналам. Например, 24-битный цвет (True Color) использует 8 бит на каждый из каналов RGB, что даёт 16,7 миллионов цветов. 32-битный цвет добавляет альфа-канал для прозрачности. Глубина 48 бит (16 бит на канал) используется в профессиональной фотографии для сохранения большего количества оттенков при постобработке. Динамический диапазон характеризует отношение максимальной яркости к минимальной, которую может зафиксировать устройство или воспроизвести изображение. Стандартные изображения имеют низкий динамический диапазон, или LDR, с яркостью от 0 до 255. Изображения с высоким динамическим диапазоном, или HDR, способны передавать детали

одновременно в ярких светах и глубоких тенях. Гамма цветов устройств, или гамма-коррекция, представляет собой нелинейную зависимость между числовым значением яркости и реальной яркостью свечения пикселя. Эта нелинейность возникает из-за физических свойств электронно-лучевых трубок и сохраняется в современных дисплеях для совместимости. Гамма-коррекция применяется для того, чтобы при ограниченной битности равномерно распределить оттенки, чувствительные для глаза. Цветовой охват, или gamut, – это множество цветов, которое может быть воспроизведено конкретным устройством, например монитором, принтером или проектором. Ни одно устройство не способно воспроизвести все видимые человеком цвета. Цветовые пространства sRGB, Adobe RGB и DCI-P3 имеют разные охваты. При работе с цветом важно понимать, что цвет, отображаемый на мониторе с охватом sRGB, может оказаться за пределами охвата принтера и наоборот.

Управление цветом, его составляющие

Система управления цветом, или CMS, представляет собой совокупность программных и аппаратных средств, обеспечивающих согласованную передачу цвета между различными устройствами: камерами, мониторами, сканерами, принтерами. Основными составляющими CMS являются эталонное цветовое пространство, обычно CIE Lab или CIE XYZ, цветовые профили устройств и модуль преобразования цветов, или CMM. Принцип работы CMS заключается в том, что каждый цвет, входящий в систему, преобразуется из цветового пространства устройства в эталонное пространство, а затем – в цветовое пространство целевого устройства. В процессе преобразования приходится решать проблему, когда цвет исходного устройства отсутствует в охвате целевого. Для этого используются различные рендеринг-интенты, определяющие, как производить сжатие цветового охвата. Перцепционный рендеринг-индент сохраняет относительные отношения между цветами, что подходит для фотографий. Относительный колориметрический рендеринг-индент оставляет точными цвета, попадающие в охват, а непопадающие отображает ближайшими, что используется в полиграфии. Абсолютный колориметрический рендеринг-индент учитывает белое поле устройства и применяется для контроля цветопробы. Насыщенностный рендеринг-индент сохраняет яркость, часто используется для диаграмм и графиков.

Профили

Цветовой профиль – это файл стандарта ICC, который описывает цветовые характеристики конкретного устройства. Для монитора профиль определяет цветовой охват, гамму, цветовую температуру. Для принтера профиль описывает, как смешение красок даёт определённые цвета на определённой бумаге. Профили

делятся на входные (для сканеров и камер), дисплейные и выходные (для принтеров и плоттеров). Профили устройств создаются с помощью калибраторов – аппаратных устройств, измеряющих реальные цвета. Также существуют профили цветовых пространств, такие как sRGB, Adobe RGB, ProPhoto RGB. Встраивание профиля в графический файл позволяет системе управления цветом правильно его отобразить на любом устройстве. Если профиль не встроен, программа предполагает, что изображение находится в стандартном цветовом пространстве, например sRGB. В профессиональной работе с цветом, особенно при подготовке к печати, использование корректных профилей является обязательным.

Цветовые палитры

Цветовая палитра в компьютерной графике – это ограниченный набор цветов, используемых в конкретном изображении или доступных в графической системе. В ранних компьютерных системах из-за ограниченности памяти использовались фиксированные палитры, например, 16, 256 цветов. Формат GIF использует индексированную палитру до 256 цветов. Каждому пикселю ставится в соответствие не цвет, а индекс в палитре, что экономит память. Для фотографических изображений такой подход неприменим, так как требуется значительно больше цветов. Современные системы оперируют полноцветными изображениями без ограничения палитры, но существуют оптимизированные палитры, например, веб-палитра из 216 цветов, которая одинаково отображалась на разных мониторах. В настоящее время цветовые палитры используются в дизайне для создания гармоничных цветовых схем, а также в форматах с ограниченным количеством цветов, например, в иконках и логотипах. Палитра может быть глобальной для всего изображения или локальной, когда каждый фрагмент изображения имеет свою палитру.

Основы колористики

Колористика – это наука о цвете, включающая теорию цветовых гармоний, психологию восприятия цвета и практические рекомендации по составлению цветовых сочетаний. Основой колористики является цветовой круг, который в классическом варианте Иттена содержит 12 цветов. Основными цветами считаются красный, синий и жёлтый. Вторичные цвета получаются смешением основных: зелёный, оранжевый, фиолетовый. Третичные цвета – смешение основного с вторичным. В компьютерной графике чаще используют круг на основе RGB, где основными являются красный, зелёный и синий. Комплементарными, или дополнительными, называются цвета, расположенные друг напротив друга в круге, например красный и зелёный. Их сочетание создаёт контраст и напряжение. Аналоговые цвета расположены рядом, например синий,

сине-зелёный, зелёный. Они создают гармоничную и спокойную гамму. Триада – это три цвета, равноудалённые на круге. Тетрада – четыре цвета, образующие прямоугольник или квадрат. При выборе цветовой схемы учитываются тёплые и холодные цвета. Тёплые ассоциируются с огнём и солнцем, холодные – с водой и небом. Хроматические цвета – цветные, ахроматические – белый, чёрный, серый. В дизайне интерфейсов и веб-приложений колористика помогает создавать удобные, не раздражающие глазу цветовые решения, акцентировать внимание на важных элементах и передавать эмоции. Также важно учитывать культурные ассоциации цветов (например, белый – цвет чистоты на Западе, траура на Востоке) и требования доступности, чтобы люди с нарушениями цветовосприятия могли различать элементы интерфейса.

ТЕМА 11. ВЕБ-ДИЗАЙН. ОСНОВЫ ВИЗУАЛЬНОГО ДИЗАЙНА. ПРИНЦИПЫ ЮЗАБИЛИТИ

Понятие веб-дизайна

Веб-дизайн представляет собой область, связанную с проектированием и созданием веб-сайтов и веб-приложений, объединяющую технические, визуальные и психологические аспекты. Веб-дизайн включает не только эстетическую составляющую, но и продуманную структуру, удобство навигации, скорость загрузки и адаптивность под разные устройства. В отличие от традиционного графического дизайна, веб-дизайн предполагает интерактивность, возможность изменения контента пользователем, а также учёт поведения посетителей. Важнейшей составляющей веб-дизайна является пользовательский интерфейс, через который человек взаимодействует с сайтом. Современный веб-дизайн также включает адаптивный дизайн, обеспечивающий корректное отображение на экранах разного размера, и отзывчивый дизайн, где элементы динамически подстраиваются под поведение пользователя. Процесс веб-дизайна обычно включает исследование аудитории, создание прототипов, визуальный дизайн и тестирование удобства использования.

Графика в веб-дизайне

Графические элементы играют ключевую роль в веб-дизайне, формируя эмоциональное восприятие сайта и помогая пользователю ориентироваться в контенте. К основным видам графики в веб-дизайне относятся фоновые изображения, иконки, логотипы, кнопки, иллюстрации, фотографии, инфографика, анимированные элементы и видео. Графика используется для создания визуальной иерархии, привлечения внимания к важным разделам, иллюстрации сложных понятий и повышения эстетической привлекательности.

Однако избыток или неправильное использование графики может привести к перегруженности интерфейса, замедлению загрузки и ухудшению восприятия. Эффективный веб-дизайн использует графику целенаправленно: каждое графическое решение должно поддерживать задачи пользователя и общую концепцию сайта. В последние годы популярны минимализм, иллюстрации в стиле flat design, крупные фоновые изображения и плавная микроанимация, а также оптимизированные векторные форматы, такие как SVG, которые сохраняют чёткость при любом масштабе и имеют малый вес.

Информационная архитектура веб-проекта

Информационная архитектура – это дисциплина, занимающаяся организацией и структурированием информации на веб-сайте таким образом, чтобы пользователи могли легко находить нужные материалы и выполнять задачи. Она включает в себя проектирование системы навигации, рубрикации, меток, поиска и иерархии страниц. Хорошая информационная архитектура делает сайт интуитивно понятным: пользователь не задумывается о том, куда нажать, чтобы попасть в нужный раздел, и всегда понимает, где находится на сайте. Основными элементами информационной архитектуры являются глобальная навигация, обычно располагаемая в шапке сайта, локальная навигация внутри раздела, контекстные ссылки, хлебные крошки, показывающие путь к текущей странице, карта сайта и поисковая система. При проектировании архитектуры используются такие методы, как карточковая сортировка, построение sitemap, создание прототипов и тестирование сценариев. Информационная архитектура тесно связана с юзабилити: даже красиво оформленный сайт может быть бесполезным, если пользователь не может быстро найти нужную информацию.

Микро- и макротипографика

Типографика в веб-дизайне делится на макротипографику и микротипографику. Макротипографика занимается общим построением текста на странице: выбором гарнитуры для заголовков и основного текста, размерами шрифтов, межстрочным интервалом, длиной строки, полями, выравниванием, иерархией заголовков. Она определяет читаемость и ритм текста. Для экранного чтения предпочтительны шрифты без засечек, такие как Arial, Helvetica, Open Sans, Roboto, но для некоторых проектов могут использоваться и шрифты с засечками, например Times New Roman, Georgia. Микротипографика касается деталей внутри строки: кернинг, трекинг, лигатуры, правильные кавычки, дефисы и тире, висячая пунктуация, устранение «разрывов» коротких слов. Хотя в вебе микротипографика реализована хуже, чем в печати, современные возможности CSS позволяют достаточно тонко управлять этими параметрами.

Принципы типографики включают контраст между заголовками и текстом, достаточный размер шрифта для чтения с экрана (обычно 16 пикселей для основного текста), интерлиньяж около 1.5 от размера шрифта, длину строки 45–75 символов. Кроме того, важно обеспечивать масштабируемость текста для людей с ослабленным зрением. Типографика напрямую влияет на удобство чтения и удержание внимания пользователя.

Свойства зрительного восприятия

Зрительное восприятие имеет ряд особенностей, которые должен учитывать веб-дизайнер. Человек не воспринимает страницу как хаотичное нагромождение элементов – он ищет закономерности, группирует объекты, выделяет главное. Закон прегнантности гласит, что восприятие стремится к наиболее простой, устойчивой и симметричной интерпретации. Закон близости: объекты, расположенные близко друг к другу, воспринимаются как группа. Закон сходства: одинаково выглядящие объекты (цвет, форма, размер) объединяются. Закон замкнутости: человек склонен достраивать незавершённые фигуры. Закон продолжения: элементы, расположенные вдоль плавной линии, воспринимаются как связанные. Кроме того, существуют особенности движения глаз: сначала человек просматривает страницу по F-образному паттерну (слева направо, затем вниз и снова слева), потом фиксируется на наиболее контрастных и крупных элементах. Знание этих свойств позволяет дизайнеру управлять вниманием пользователя, направляя его к важным элементам, таким как призывы к действию, формы регистрации, навигация. Также важны психофизические ограничения: различение цветов, влияние яркости и контрастности на утомляемость.

Основные принципы графического дизайна. Законы и приемы композиции

Графический дизайн в вебе опирается на ряд фундаментальных принципов. Баланс может быть симметричным, когда элементы зеркально отражаются относительно оси, или асимметричным, когда равновесие достигается разными по размеру и тону объектами. Контраст выделяет важные элементы через разницу в цвете, размере, форме, насыщенности. Иерархия организует информацию от наиболее важного к наименее важному с помощью размера, цвета, положения. Ритм создаётся чередованием элементов, направляя движение взгляда. Пропорция, в том числе золотое сечение, помогает строить гармоничные сетки. Единство означает, что все части дизайна работают как целое, подчиняясь общей стилистике. К законам композиции относятся закон целостности, когда изображение воспринимается как единая фигура на фоне; закон контраста, выделяющий главное; закон подчинённости, когда второстепенные элементы не

спорят с главными; закон типизации, использующий повторяющиеся паттерны. Приёмы композиции включают правило третей, диагональные линии, рамки, пустое пространство для «дыхания», повторение элементов для создания ритма. В веб-дизайне сетка является основой композиции, особенно модульные сетки, обеспечивающие согласованность между страницами.

Понятия: размер, форма, цвет, текстура, размещение, шрифт, композиция, контраст

Размер в веб-дизайне определяет иерархию и внимание: более крупные элементы воспринимаются как более важные. Форма может быть геометрической (прямоугольники, круги, треугольники) или органической. Прямоугольные формы ассоциируются со стабильностью, круглые – с мягкостью и общностью. Цвет несёт не только эстетическую, но и функциональную нагрузку: выделение ссылок, сигнальные цвета для ошибок, фирменные цвета для узнаваемости. Текстура придаёт поверхности визуальную или тактильную иллюзию: градиенты, тени, имитация бумаги, ткани, металла. В современном веб-дизайне текстуры используются аккуратно, в стилях неоморфизм или стекломорфизм. Размещение элементов на странице определяет их связь: близко расположенные связаны, далеко – не связаны. Шрифт является мощным выразительным средством: гарнитура, начертание, размер, цвет, межбуквенное расстояние формируют настроение и читаемость. Композиция объединяет все элементы в единое целое, создавая визуальную структуру и направляя взгляд. Контраст проявляется в различиях между элементами: тёмное на светлом, крупное на мелком, яркое на приглушённом. Он привлекает внимание и создаёт акценты. Без контраста дизайн становится плоским и невыразительным. Взаимодействие этих понятий лежит в основе каждого удачного веб-проекта.

Основные принципы создания графики для Веб

При создании графики для веба необходимо учитывать не только визуальные качества, но и технические ограничения и особенности использования. Изображения должны быть адаптивными, то есть корректно отображаться на разных экранах. Для этого используется несколько версий одного изображения с разным разрешением, загружаемых через атрибуты srcset и sizes. Важно обеспечивать доступность: все значимые изображения должны иметь текстовое описание в атрибуте alt, а декоративные изображения – пустой alt для игнорирования скринридерами. Графика не должна нарушать восприятие текста: фон должен иметь достаточный контраст, текст поверх изображений должен быть легко читаемым, для чего используются полупрозрачные подложки или затемнение фона. При создании интерфейсных иконок предпочтительны векторные форматы SVG, которые легко масштабировать. Для фотографий

товаров и больших иллюстраций используют JPEG и WebP. Анимация должна быть умеренной, без мерцания, чтобы не вызывать дискомфорт у людей с вестибулярными нарушениями. Также следует учитывать цветовую слепоту: не полагаться только на цвет при передаче важной информации, использовать дополнительные индикаторы. Наконец, графика не должна создавать ложных ожиданий: интерактивные элементы должны выглядеть кликабельными, кнопки – как кнопки.

Оптимизация графических изображений

Оптимизация графики для веба преследует две цели: уменьшение размера файла для быстрой загрузки и сохранение приемлемого визуального качества. Основные методы оптимизации включают выбор правильного формата, сжатие с потерями или без потерь, изменение размера до реально необходимого, удаление метаданных. Для JPEG важно подобрать степень сжатия: слишком высокое сжатие даёт артефакты, слишком низкое – излишний вес. Для PNG с прозрачностью можно уменьшить глубину цвета, например использовать 8-битный PNG вместо 24-битного. Современные инструменты, такие как TinyPNG, ImageOptim, Squoosh, выполняют автоматическую оптимизацию, удаляя лишние данные. Также практикуется отложенная загрузка, или lazy loading, когда изображение загружается только тогда, когда оно появляется в видимой области экрана, что ускоряет начальную загрузку страницы. Для больших фоновых изображений можно использовать технологию критического рендеринга: сначала показать простой цвет фона, а после загрузки полного изображения – плавно его проявить. Оптимизация особенно важна для мобильных устройств с медленным соединением.

Размеры и разрешение

При подготовке графики для веба важно понимать разницу между физическим размером в пикселях и разрешением. Для экрана имеет значение только количество пикселей по ширине и высоте. Обычное абсолютное разрешение экрана составляет от 1366×768 до 3840×2160. Однако современные устройства с Retina-экранами имеют удвоенную плотность пикселей, поэтому изображение, отображаемое как 100×100 точек, должно быть предоставлено в размере 200×200 пикселей, чтобы выглядеть чётким. Это достигается использованием атрибутов srcset и 2x. Для фоновых изображений можно задавать размеры через CSS. Важно не загружать изображение больше, чем требуется: если на сайте фотография отображается в блоке шириной 300 пикселей, нет смысла загружать оригинал в 4000 пикселей. Адаптивный дизайн предусматривает разные версии из одного исходного изображения, уменьшенные для мобильных устройств. При работе с векторной графикой, например SVG,

размер задаётся через viewBox и атрибуты width/height, но качество не страдает при масштабировании.

Уменьшение количества цветов

Уменьшение количества цветов – один из эффективных методов оптимизации графики, особенно для PNG и GIF. Полноцветное изображение может содержать миллионы цветов, но не все из них визуально различимы или необходимы. Преобразование изображения в индексированную палитру с ограниченным числом цветов, например 256 или 64, значительно сокращает размер файла за счёт сокращения информации о цвете. Этот метод хорошо работает для иконок, иллюстраций, скриншотов, логотипов, диаграмм. Для фотографий он не подходит, так как приводит к заметным искажениям, так называемым полосам на градиентах. При индексировании используются алгоритмы, которые подбирают наиболее репрезентативные цвета, а также применяют диффузию ошибок для сглаживания резких переходов. В веб-формате PNG можно установить глубину цвета 2, 4, 8, 16 бит. В GIF максимальное количество цветов – 256. В современных сайтах для простой графики часто выбирают SVG, где цвета задаются декларативно, и проблема оптимизации количества цветов не стоит. Однако при массовом создании контента автоматическое уменьшение палитры может значительно ускорить загрузку страниц.

Выбор формата файла

Выбор правильного формата изображения для веб-страницы критически влияет на скорость загрузки и качество. JPEG подходит для фотографий, сложных градиентов, изображений с непрерывными тонами. Он поддерживает миллионы цветов, но сжатие с потерями может создавать артефакты. PNG используется для изображений с прозрачностью, логотипов, скриншотов, иконок, текста – всего, что требует чётких краёв и сжатия без потерь. PNG-24 даёт высокое качество и полную прозрачность, но весит много; PNG-8 использует палитру до 256 цветов. GIF ограничен 256 цветами и поддерживает простую анимацию, но для длительных или качественных анимаций лучше использовать видеоформаты MP4 или WebM. WebP – современный формат от Google, обеспечивающий лучшее сжатие, чем JPEG и PNG, поддерживает прозрачность и анимацию. Он рекомендуется к использованию, но требует проверки совместимости со старыми браузерами. SVG – векторный формат для иконок, логотипов, иллюстраций, инфографики. Он легковесен, масштабируется, а также может быть стилизован через CSS и анимирован. AVIF – новейший формат на основе кодека AV1, дающий ещё более высокую степень сжатия, но его поддержка растёт. При выборе формата нужно учитывать характер изображения,

необходимость прозрачности, анимации, поддержку браузерами и баланс между качеством и размером. Часто для одного изображения задают несколько форматов с помощью элемента picture, чтобы браузер выбирал самый подходящий.

ТЕМА 12. ОБЗОР СОВРЕМЕННЫХ МЕТОДОВ И ТЕХНОЛОГИЙ ПРОЕКТИРОВАНИЯ И ВЕРСТКИ ВЕБ-САЙТОВ

Статические и динамические компоненты представления веб-приложений

Веб-приложения состоят из статических и динамических компонентов представления. Статические компоненты – это элементы, которые не меняются в зависимости от запроса пользователя и одинаковы для всех посетителей. К ним относятся HTML-страницы, которые хранятся на сервере в готовом виде и отправляются браузеру без какой-либо обработки, CSS-файлы, определяющие внешний вид, клиентские скрипты на JavaScript, а также статические изображения, шрифты и другие медиафайлы. Статические компоненты просты в реализации, легко кэшируются и быстро загружаются. Динамические компоненты, напротив, генерируются на сервере в момент запроса с учётом параметров пользователя, времени, данных из базы и других факторов. Это может быть персонализированная страница интернет-магазина с корзиной конкретного покупателя, результаты поиска, лента новостей, комментарии. Для создания динамических компонентов используются серверные языки программирования и технологии, которые обрабатывают запрос, взаимодействуют с базами данных и формируют HTML, отправляемый клиенту. Граница между статикой и динамикой размывается современными подходами, такими как одностраничные приложения, где часть содержимого генерируется на клиенте с помощью JavaScript, а сервер предоставляет данные через API. Однако традиционное разделение остаётся важным для понимания архитектуры веб-приложений.

Технологии, используемые на стороне сервера: ASP-технология, ASP.NET, PHP, Python, Perl

Серверные технологии отвечают за обработку запросов, выполнение бизнес-логики, взаимодействие с базами данных и генерацию ответа, обычно в виде HTML-страницы. Одной из первых широко распространённых технологий была ASP, или Active Server Pages, разработанная Microsoft. ASP позволяла встраивать скрипты на VBScript или JScript в HTML-страницы, которые выполнялись на сервере. Со временем ASP была заменена более мощной

платформой ASP.NET, которая является частью экосистемы .NET Framework. ASP.NET поддерживает языки C и Visual Basic, использует компиляцию кода, предоставляет обширную библиотеку классов и модель событий, похожую на Windows Forms. Современная версия ASP.NET Core является кросс-платформенной, работает на Windows, Linux и macOS. PHP, или Personal Home Page Tools, ныне рекурсивно расшифровывается как PHP: Hypertext Preprocessor, является интерпретируемым языком, специально созданным для веб-разработки. PHP-код встраивается непосредственно в HTML, что упрощает создание динамических страниц. PHP популярен благодаря низкому порогу входа, огромному сообществу и большому количеству готовых решений, таких как WordPress, Joomla, Drupal. Python используется в веб-разработке с помощью фреймворков, таких как Django и Flask. Django предоставляет полнофункциональную платформу с ORM, админ-панелью, аутентификацией, в то время как Flask является микрофреймворком, дающим базовые средства маршрутизации и расширяемым через плагины. Perl, хотя сейчас менее популярен, исторически сыграл большую роль в раннем вебе. Его мощные средства обработки текста и регулярных выражений делали его удобным для генерации HTML и обработки форм. На Perl написан Moveable Type и многие ранние CGI-скрипты. Сегодня выбор серверной технологии зависит от требований проекта, опыта команды, производительности и экосистемы.

Технологии веб-программирования, применяющиеся на стороне клиента для реализации GUI: HTML, XHTML, CSS

HTML, или HyperText Markup Language, является основным языком разметки для создания веб-страниц. Он определяет структуру документа: заголовки, абзацы, списки, ссылки, изображения, формы и другие элементы. HTML постоянно развивается; современная версия HTML5 добавила семантические элементы, такие как header, footer, article, section, а также встроенную поддержку видео и аудио, canvas для рисования, локальное хранилище. XHTML, или Extensible HyperText Markup Language, представлял собой попытку реформировать HTML в соответствии со строгими правилами XML. В XHTML все теги должны быть закрыты, атрибуты – в кавычках, регистр – единообразным. XHTML был популярен в конце 2000-х, но с выходом HTML5 его использование снизилось, так как HTML5 вобрал лучшие черты XHTML, не требуя излишней строгости. CSS, или Cascading Style Sheets, отвечает за визуальное оформление HTML-документов. CSS отделяет содержимое от представления, позволяя менять внешний вид всех страниц сайта централизованно. Благодаря CSS задаются цвета, шрифты, отступы, позиционирование, анимация, адаптивность под разные экраны. Современный CSS включает flexbox и grid для построения сложных макетов, переменные, медиа-запросы для адаптивного дизайна, анимации и трансформации. Вместе

HTML и CSS образуют фундамент графического интерфейса любого веб-приложения, обеспечивая доступность и совместимость с разными браузерами.

Технологии веб-программирования, применяющиеся на стороне клиента для формирования и обработки запросов, создания интерактивного и независимого от браузера интерфейса: JavaScript, Java, Апплеты Java, Active-X, Adobe Flash/Flex, Silverlight, AJAX.

Клиентские технологии позволяют сделать веб-страницу интерактивной, обрабатывать действия пользователя без перезагрузки, отправлять асинхронные запросы к серверу. Основной и доминирующей технологией сегодня является JavaScript. Изначально созданный для простых скриптов, JavaScript эволюционировал в мощный язык, поддерживающий объектно-ориентированное, функциональное и событийно-ориентированное программирование. С появлением AJAX, асинхронного JavaScript и XML, стало возможным обмениваться данными с сервером без перезагрузки страницы, что привело к созданию одностраничных приложений, таких как Gmail и Google Maps. Современные фреймворки React, Angular, Vue позволяют строить сложные интерфейсы с управлением состоянием, виртуальным DOM и компонентным подходом. Java также использовалась на клиенте в виде апплетов Java. Апплеты были небольшими программами, запускавшимися внутри браузера через виртуальную машину Java. Они позволяли выполнять тяжёлые вычисления и рисовать графику, но требовали установки плагина, долго загружались и имели проблемы безопасности. К концу 2010-х апплеты были полностью вытеснены. Технология Active-X от Microsoft позволяла загружать и выполнять в браузере нативные компоненты Windows. Это давало большие возможности, но было небезопасно и работало только в Internet Explorer, поэтому широкого распространения не получило. Adobe Flash и его фреймворк Flex долгое время были стандартом для анимации, видеопроигрывателей, игр и сложных интерактивных приложений в вебе. Flash использовал векторную графику и ActionScript. Однако из-за проблем безопасности, производительности и закрытости технологии Flash был официально прекращён в 2020 году. Silverlight от Microsoft был попыткой создать конкурента Flash, поддерживал C и XAML, но также требовал плагина и не приобрёл достаточной популярности, особенно с ростом возможностей HTML5. Сегодня JavaScript и его экосистема являются безальтернативным стандартом для клиентской интерактивности, а AJAX реализуется через Fetch API или XMLHttpRequest. Современные веб-приложения обеспечивают полную интерактивность, независимость от браузера и работу на любых устройствах.

Обзор информационных систем управления контентом (CMS)

Системы управления контентом, или CMS, представляют собой программные платформы, которые позволяют создавать, редактировать,

организовывать и публиковать цифровой контент без необходимости глубоких знаний веб-программирования. CMS предоставляют графический интерфейс для добавления текста, изображений, видео, управления структурой сайта, назначения прав доступа. Самым популярным примером является WordPress, который изначально создавался как блоггинг-платформа, но превратился в полноценную CMS, поддерживающую тысячи плагинов и тем. На WordPress работают блоги, новостные сайты, интернет-магазины, корпоративные порталы. Joomla и Drupal являются более мощными и гибкими, но требуют большей технической квалификации. Они подходят для сложных сайтов, сообществ, образовательных порталов. Помимо классических CMS, существуют headless CMS, такие как Contentful, Strapi, которые предоставляют только бэкенд для управления контентом, а фронтенд реализуется отдельно с помощью JavaScript-фреймворков. Это даёт свободу в выборе технологий отображения. Также популярны конструкторы сайтов, например Tilda, Wix, Squarespace, которые позволяют собирать сайты визуально без кода. Выбор CMS зависит от масштаба проекта, требуемой гибкости, бюджета и навыков команды.

Виды верстки: таблицы, фреймы, блоки

Верстка веб-страниц прошла долгий путь эволюции. В ранние годы интернета использовалась табличная вёрстка, где HTML-таблицы служили каркасом для размещения контента. Ячейки таблиц могли объединяться, в них вкладывались изображения, текст и другие таблицы. Табличная вёрстка позволяла создавать сложные макеты, но имела серьёзные недостатки: разметка раздувалась, код становился нечитаемым, было трудно менять структуру адаптивно, а скорость загрузки снижалась. Также использовались фреймы, которые делили окно браузера на независимые области, каждая из которых загружала свой HTML-документ. Фреймы позволяли, например, сделать фиксированную навигацию, не перезагружающуюся при смене контента. Однако у фреймов были проблемы: сложность с букианализацией, невозможность ссылаться на конкретную комбинацию фреймов, проблемы с печатью, трудности с адаптивностью и навигацией для скринридеров. В современной веб-разработке и таблицы, и фреймы считаются устаревшими и крайне редко применяются. Современные стандарты основаны на блочной вёрстке с использованием CSS. Блочная вёрстка использует элементы div и другие семантические теги, позиционируемые с помощью свойств CSS, таких как float, flexbox, grid. Flexbox предназначен для создания одномерных макетов (в строку или в столбец) с выравниванием, распределением пространства. Grid позволяет строить двухмерные сетки, где строки и столбцы задаются явно. Блочная вёрстка отделяет структуру от представления, делает код чистым, поддерживает адаптивность через медиа-запросы, улучшает доступность и производительность. Сегодня при создании сайтов использование семантической

блочной вёрстки с CSS-фреймворками, такими как Bootstrap или Tailwind CSS, является стандартом индустрии.

ТЕМА 13. НАЗНАЧЕНИЕ И ПРИМЕНЕНИЕ CSS

Понятие каскадных таблиц стилей, их назначение, определение и использование при форматировании HTML-документа

Каскадные таблицы стилей, или CSS, представляют собой язык описания внешнего вида документа, написанного на языке разметки, чаще всего HTML или XML. Основное назначение CSS заключается в отделении содержимого веб-страницы от его представления. HTML отвечает за структуру и семантику: заголовки, абзацы, списки, ссылки, формы. CSS отвечает за визуальное оформление: цвета, шрифты, отступы, расположение элементов, анимацию и адаптивность под разные устройства. Такое разделение даёт множество преимуществ. Один CSS-файл может управлять внешним видом всех страниц сайта, что упрощает поддержку и глобальное изменение дизайна. Код становится чище и легче для понимания. Пользователи могут подключать свои стили, например, для увеличения шрифта или изменения контраста, что улучшает доступность. Кроме того, CSS позволяет применять одно и то же форматирование к множеству элементов, сокращая объём передаваемых данных. Современные веб-стандарты требуют использования CSS для всего, что касается представления, оставляя HTML только для структуры и смысловой разметки.

Способы подключения стилей к документу

Существует три основных способа подключения CSS к HTML-документу. Первый способ – внешние таблицы стилей. Стили выносятся в отдельный файл с расширением .css, а в HTML-документе внутри элемента link подключается ссылка на этот файл. Это предпочтительный способ, особенно для многостраничных сайтов. Он обеспечивает максимальное разделение содержания и оформления, позволяет браузеру кэшировать CSS-файл и использовать его для всех страниц. Второй способ – глобальные стили, или встроенные в раздел style. Блок CSS-правил размещается внутри элемента style, который обычно находится в разделе head документа. Этот способ удобен при создании одностраничных макетов или когда нужны стили, специфичные только для одного документа. Однако он не позволяет переиспользовать стили на других страницах. Третий способ – инлайн-стили, или атрибут style. Стили задаются непосредственно внутри открывающего тега элемента с помощью атрибута style. Этот способ имеет самый высокий приоритет, но нарушает принцип разделения и затрудняет поддержку. Инлайн-стили обычно используются для динамического

изменения стилей через JavaScript или в крайних случаях, когда требуется переопределить другие правила. Приоритет применения стилей при конфликтах определяется каскадированием: инлайн-стили имеют наивысший приоритет, затем идут глобальные стили в разделе style, затем внешние таблицы, а стили браузера по умолчанию – самые низкие.

Замена тегов на стилевое описание

Ранние версии HTML содержали множество тегов и атрибутов, отвечающих за оформление, такие как font, center, big, small, bgcolor, align. С появлением CSS эти теги были объявлены устаревшими, и их функциональность перенесена в стилевое описание. Например, вместо тега font для изменения цвета текста используется свойство color, вместо атрибута bgcolor для фона – свойство background-color, вместо тега center – свойство text-align или margin: auto для блочных элементов. Такой подход очищает HTML-код от визуальных деталей, делая его сосредоточенным на структуре. Семантические теги header, nav, main, article, section, aside, footer остались, но их внешний вид полностью задаётся через CSS. Например, можно сделать все элементы header на сайте с синим фоном и белым текстом, при этом сам HTML не содержит никаких указаний на цвета. Замена тегов на стилевое описание также улучшает доступность, так как скринридеры ориентируются на семантику, а не на визуальное оформление. Кроме того, становится возможным быстро менять дизайн, не трогая HTML.

Стили элементов. Блочные и строковые элементы

В CSS каждый элемент HTML имеет тип отображения, определяемый свойством display. Основными значениями являются block, inline и inline-block. Блочные элементы, такие как div, p, h1, ul, li, по умолчанию занимают всю доступную ширину родительского контейнера и начинаются с новой строки. К ним применимы такие свойства, как width, height, margin, padding, border. Строчные элементы, такие как span, a, strong, em, img, располагаются в строке текста и занимают только необходимую ширину. К ним не применимы width и height, а вертикальные отступы (margin-top, margin-bottom) не влияют на расположение соседних элементов. Изменяя свойство display, можно менять поведение элемента. Например, сделать строчный элемент блочным, чтобы задать ему размеры, или наоборот, сделать блочный элемент строчным, чтобы расположить несколько блоков в одну строку. Значение inline-block даёт свойства обоих типов: элемент располагается в строке, но при этом ему можно задавать ширину, высоту и отступы. Понимание блочной и строчной модели необходимо для правильного позиционирования и построения макетов.

Стилевое позиционирование

CSS предоставляет несколько способов позиционирования элементов на странице. Статическое позиционирование, или static, является значением по умолчанию. Элементы располагаются в нормальном потоке документа один за другим в соответствии с их типом отображения (блочные или строчные). Относительное позиционирование, или relative, позволяет сместить элемент относительно его нормальной позиции с помощью свойств top, right, bottom, left. При этом исходное место элемента остаётся зарезервированным, а смещение не влияет на расположение соседних элементов. Абсолютное позиционирование, или absolute, удаляет элемент из нормального потока, и он позиционируется относительно ближайшего позиционированного предка, то есть предка с позиционированием не static. Если такого предка нет, то относительно окна браузера. Абсолютно позиционированные элементы не влияют на расположение других элементов. Фиксированное позиционирование, или fixed, похоже на absolute, но элемент привязывается к окну браузера и не перемещается при прокрутке страницы, что используется для фиксированных меню или кнопок. Липкое позиционирование, или sticky, представляет собой гибрид relative и fixed: элемент ведёт себя как relative до тех пор, пока при прокрутке не достигнет заданной границы, после чего «прилипает» к экрану. Свойство z-index управляет наложением позиционированных элементов по оси Z: чем больше значение, тем выше элемент в стопке. Позиционирование является одной из наиболее мощных и сложных тем в CSS, позволяющей создавать сложные интерфейсы, модальные окна, выпадающие меню и многое другое.

Стили классов

В CSS классы являются одним из основных механизмов для применения одинакового оформления к разным элементам, а также для выбора элементов по семантическому признаку. Класс задаётся в HTML с помощью атрибута class, которому можно присвоить одно или несколько имён, разделённых пробелами. В CSS-правиле класс обозначается точкой перед именем, например .my-class. Классы можно применять к любым элементам, независимо от их типа. Один элемент может иметь несколько классов, что позволяет комбинировать стили: например, класс button задаёт общие свойства кнопки, а класс primary задаёт цветовую схему для главной кнопки. Классы более гибкие, чем селекторы по тегу, поскольку позволяют разметить страницу логически и не привязываться к конкретным HTML-элементам. При использовании методологий именования CSS, таких как БЭМ, классы становятся основным инструментом для создания модульных и переиспользуемых компонентов. Стили классов имеют более высокую специфичность, чем стили тегов, но более низкую, чем стили

идентификаторов и инлайн-стили. Это позволяет легко переопределять базовые стили.

Внешние стилевые таблицы

Внешние таблицы стилей представляют собой отдельные файлы с расширением .css, которые подключаются к HTML-документу через элемент link. Такой подход является наилучшей практикой для проектов любого размера. Внешний CSS-файл может быть подключён к нескольким HTML-страницам, что обеспечивает единообразие дизайна и упрощает внесение глобальных изменений: достаточно отредактировать один файл, и изменения отразятся на всём сайте. Браузер кэширует внешний CSS-файл при первом посещении, что ускоряет загрузку последующих страниц. Внешние таблицы стилей можно использовать для разных медиа-устройств, указывая атрибут media, например, для печати, для экранов, для мобильных устройств. Структура CSS-файла может быть организована иерархически: сброс стилей, переменные, базовые стили, компоненты, утилиты. При разработке часто используются препроцессоры, такие как Sass или Less, которые компилируются во внешние CSS-файлы, добавляя переменные, вложенность, миксины. Внешние таблицы помогают поддерживать чистоту HTML, делают код более доступным и облегчают командную разработку.

Добавление фонового рисунка

CSS позволяет задавать фоновые изображения для любого элемента с помощью свойства background-image, значением которого является URL изображения. Фоновое изображение по умолчанию повторяется по горизонтали и вертикали, заполняя всю область элемента. Это поведение можно изменить с помощью свойства background-repeat, принимающего значения no-repeat, repeat-x, repeat-y, space, round. Свойство background-position задаёт начальную позицию изображения, например top left, center, 10px 20px. Свойство background-size управляет масштабированием: значения cover масштабируют изображение так, чтобы оно полностью покрыло элемент, возможно, обрезая края; contain масштабирует так, чтобы изображение полностью поместилось внутри элемента, возможно, оставляя пустые области. Можно также задать конкретные размеры, например 100px 100px или проценты. Фоновое изображение можно комбинировать с фоновым цветом: цвет будет виден, если изображение не загрузится или имеет прозрачные области. Свойство background-attachment определяет, прокручивается ли фон вместе с элементом (scroll) или остаётся фиксированным (fixed). Сокращённое свойство background позволяет задать все параметры в одной строке: background: url(photo.jpg) no-repeat center / cover #f0f0f0. Фоновые изображения широко используются для декоративных целей,

текстур, градиентов, создания сложных фоновых эффектов без использования html-тегов img.

Текст, списки, таблицы

CSS предоставляет богатые возможности для оформления текста, списков и таблиц. Для текста используются свойства font-family для выбора гарнитуры шрифта, font-size для размера, font-weight для насыщенности, font-style для начертания, color для цвета, line-height для межстрочного интервала, letter-spacing для расстояния между буквами, word-spacing для расстояния между словами, text-align для выравнивания, text-decoration для подчёркивания или зачёркивания, text-transform для изменения регистра, text-shadow для тени. Для списков свойства list-style-type позволяют изменить маркеры: диск, круг, квадрат, римские цифры, буквы. list-style-image позволяет задать собственное изображение в качестве маркера. list-style-position определяет, находится ли маркер внутри или снаружи блока списка. Свойство list-style объединяет эти три параметра. Для таблиц CSS позволяет управлять border-collapse: collapse объединяет границы ячеек, border-spacing задаёт расстояние между границами для раздельных границ. Свойства caption-side размещают заголовок таблицы сверху или снизу. Empty-cells определяет, отображать ли границы пустых ячеек. text-align и vertical-align внутри ячеек выравнивают содержимое по горизонтали и вертикали. С помощью свойств padding и border можно задавать внутренние и внешние границы ячеек. Таблицы также могут иметь сложные заголовки, а строки и столбцы – выделяться с помощью псевдоклассов, например tr:hover.

Цветовое и декоративное оформление текста

Цветовое оформление текста включает не только основной цвет, но и фон, тени, градиенты. Цвет задаётся с помощью свойства color, а фон – background-color. Для создания градиентного текста используется техника с background-clip: text. Свойство text-shadow позволяет добавить тень тексту, задавая смещение по x, смещение по y, радиус размытия и цвет тени. Можно задать несколько теней через запятую. Свойство text-decoration также поддерживает линии: underline, overline, line-through. При этом можно менять цвет и стиль линии отдельно с помощью text-decoration-color и text-decoration-style. Свойство text-emphasis добавляет акцентные символы над или под текстом, что используется в японской и китайской типографике. Градиент может быть использован не только для фона, но и для текста, однако это требует дополнительных ухищрений. Также можно применять эффект «выделения» с помощью background с прозрачностью, создавая «маркер» за текстом. Важно при этом обеспечивать достаточный контраст для читаемости, руководствуясь рекомендациями WCAG по доступности.

Оформление заголовка

Заголовки разных уровней от h1 до h6 являются важными элементами структуры страницы. CSS позволяет задавать им индивидуальное оформление, подчеркивая иерархию и привлекая внимание. Стандартная практика – установить для всех заголовков семейство шрифтов, отличное от основного текста, часто с засечками или наоборот, чтобы выделить. Размер шрифта обычно делают крупнее: для h1 – около 2em или 32px, для h2 – 1.5em, и так далее. Поля, или margin, сверху и снизу задают отступы, чтобы заголовки отделялись от окружающего текста. Для особого оформления можно добавить подчёркивание через border-bottom или text-decoration. Также часто применяется свойство text-transform: uppercase для придания заголовкам лаконичности. Свойство letter-spacing позволяет разрядить буквы. Тень, или text-shadow, добавляет объём. Можно также использовать псевдоэлементы ::before и ::after для добавления декоративных элементов, таких как линии или иконки, рядом с заголовком. При работе с отзывчивым дизайном размеры заголовков могут изменяться с помощью медиа-запросов, чтобы на маленьких экранах они не были слишком крупными. Важно сохранять семантическую иерархию: на странице должен быть один h1, обозначающий главный заголовок, а остальные уровни используются для подзаголовков и секций.

Контекстный селектор, псевдоэлементы и псевдоклассы, группировка, каскадирование и наследование

Контекстные селекторы, или комбинаторы, позволяют выбрать элементы в зависимости от их положения в дереве документа. Пробел обозначает выбор потомка, например, div p выбирает все p, находящиеся внутри div. Символ > выбирает только прямых дочерних элементов: div > p. Символ + выбирает соседний элемент, идущий непосредственно после: h1 + p выбирает первый абзац после заголовка. Символ ~ выбирает все соседние элементы после: h1 ~ p выбирает все абзацы, следующие за h1. Псевдоклассы определяют динамическое состояние элемента, например :hover при наведении мыши, :focus при фокусе, :active при нажатии, :visited для посещённых ссылок. Также есть структурные псевдоклассы: :first-child, :last-child, :nth-child, :first-of-type и другие. Псевдоэлементы, обозначаемые двойным двоеточием, позволяют стилизовать части элемента: ::first-line для первой строки текста, ::first-letter для первой буквы, ::before и ::after для вставки генерируемого содержимого до или после содержимого элемента. Группировка селекторов осуществляется через запятую: h1, h2, h3 – одни и те же стили применяются ко всем трём типам заголовков. Каскадирование – это механизм разрешения конфликтов, когда несколько правил

применяются к одному элементу. Приоритет определяется весом селектора: инлайн-стили самые сильные, затем идентификаторы, затем классы и псевдоклассы, затем селекторы по тегу и псевдоэлементы. Важное правило !important может переопределить вес, но его использование не рекомендуется. Наследование – это передача значений свойств от родительского элемента к дочерним. Например, свойство color наследуется, поэтому если задать цвет для body, то все дочерние элементы унаследуют его, если не переопределят. Не все свойства наследуются; свойства, связанные с размерами и позиционированием, обычно не наследуются. Понимание каскадирования и наследования необходимо для эффективной работы с CSS и избежания неожиданных эффектов при переопределении стилей.

II. ПРАКТИЧЕСКИЙ РАЗДЕЛ

ПЛАНЫ И ЗАДАНИЯ ДЛЯ ЛАБОРАТОРНЫХ РАБОТ

Лабораторная работа 1. Шаблоны объектно-ориентированного анализа и проектирования, их классификация

Цель работы: изучить классификацию шаблонов проектирования (порождающие, структурные, поведенческие), научиться различать шаблоны и выбирать подходящий для поставленной задачи.

Задание:

Ознакомиться с теоретическим материалом о трёх группах шаблонов (GoF).

Разработать консольное приложение на JavaScript, которое демонстрирует примеры хотя бы одного шаблона из каждой группы:

- Порождающий – Фабричный метод.
- Структурный – Адаптер.
- Поведенческий – Стратегия.

Реализовать простой калькулятор, в котором алгоритм вычисления (сложение, вычитание, умножение) выбирается динамически с помощью паттерна Стратегия.

Оформить код с комментариями, указав, какой паттерн используется и для чего.

Порядок выполнения:

Создать в VS Code проект (папку) для лабораторной работы.

Написать код на JavaScript (можно использовать Node.js для запуска в терминале).

Реализовать классы:

- Context (калькулятор), который хранит ссылку на стратегию.
- Интерфейс (или абстрактный класс) Strategy с методом execute(a, b).
- Конкретные стратегии AddStrategy, SubtractStrategy, MultiplyStrategy.

Продемонстрировать работу: создать калькулятор, поочерёдно устанавливать разные стратегии и вызывать вычисления.

Добавить пример использования Адаптера (например, адаптировать старый класс OldCalculator к новому интерфейсу).

Добавить пример Фабричного метода (создание разных типов документов или логов).

Проверить выполнение программы, зафиксировать вывод в консоли.

Контрольные вопросы:

1. На какие три категории делятся шаблоны проектирования GoF?
2. В чём отличие шаблона Стратегия от шаблона Состояние?
3. Для каких задач предназначен шаблон Адаптер?
4. Какую проблему решает шаблон Фабричный метод?
5. Какие преимущества даёт использование интерфейсов (или абстрактных классов) в реализации паттернов?

Лабораторная работа 2. Порождающие шаблоны

Цель работы: научиться применять порождающие шаблоны (Одиночка, Строитель, Абстрактная фабрика, Фабричный метод, Прототип) на практике.

Задание:

Реализовать программу на JavaScript, которая демонстрирует работу следующих порождающих шаблонов:

- Одиночка (Singleton) – для глобального конфигуратора.
- Строитель (Builder) – для пошагового создания сложного объекта (например, заказ в ресторане: добавить пиццу, напиток, десерт).
- Прототип (Prototype) – для клонирования объекта (например, клонирование рекламной кампании).

Показать, что Одиночка возвращает один и тот же экземпляр при нескольких обращениях.

Продемонстрировать возможность разных вариантов сборки с помощью Строителя.

Порядок выполнения:

Создать класс Config с приватным конструктором и статическим методом getInstance().

Реализовать класс OrderBuilder с методами addPizza(), addDrink(), addDessert() и методом build(), возвращающим объект Order.

Создать класс Campaign с методом clone(), который выполняет поверхностное (или глубокое) копирование свойств.

В основной функции: создать два экземпляра Config – убедиться, что они равны. Построить два варианта заказа с помощью строителя. Клонировать кампанию и изменить свойства клона, убедиться, что оригинал не изменился.

Вывести результаты в консоль.

Контрольные вопросы:

1. Какие проблемы может вызвать неправильное использование шаблона

Одиночка?

2. В каких случаях удобно использовать Строитель вместо Фабричного метода?
3. Как отличить поверхностное копирование от глубокого?
4. Какой порождающий шаблон помогает создавать семейства связанных объектов?
5. Почему Прототип может быть эффективнее вызова конструктора?

Лабораторная работа 3. Структурные шаблоны

Цель работы: освоить применение структурных паттернов: Декоратор, Компоновщик, Фасад.

Задание:

Написать программу на JavaScript, иллюстрирующую использование:

- Декоратор – для динамического добавления функциональности объекту (например, кофе с добавками: молоко, сахар, шоколад).
- Компоновщик – для работы с деревом элементов (файловая система: папки и файлы).
- Фасад – для упрощения интерфейса к сложной подсистеме (например, запуск компьютера: процессор, память, жесткий диск).

Реализовать иерархию классов для кофе: базовый класс Coffee и декораторы MilkDecorator, SugarDecorator.

Реализовать компоненты File и Folder с общим интерфейсом getSize().

Реализовать класс ComputerFacade, который скрывает детали включения компонентов.

Порядок выполнения:

Создать класс Coffee с методом cost().

Создать абстрактный класс CoffeeDecorator, наследующий от Coffee, и конкретные декораторы, переопределяющие cost() (добавляют стоимость добавки).

В клиентском коде обернуть объект SimpleCoffee несколькими декораторами и вывести итоговую стоимость.

Создать интерфейс FileSystemComponent с методом getSize(). Реализовать классы File и Folder (Folder содержит массив компонентов и суммирует их размеры).

Построить дерево папок и файлов и вычислить общий размер.

Создать классы CPU, Memory, HardDrive с методами start(). ComputerFacade имеет метод turnOn(), который последовательно запускает компоненты.

Продемонстрировать работу фасада.

Контрольные вопросы:

1. В чём отличие Декоратора от наследования?
2. Когда следует применять Компоновщик?
3. Какие преимущества даёт Фасад?
4. Можно ли комбинировать Декоратор с Компоновщиком? Приведите пример.
5. Чем отличается Адаптер от Декоратора?

Лабораторная работа 4. Поведенческие шаблоны

Цель работы: изучить поведенческие шаблоны: Наблюдатель, Команда, Итератор.

Задание:

Реализовать систему на JavaScript, демонстрирующую:

- Наблюдатель (Observer) – уведомление подписчиков об изменении состояния (например, новостной канал).
- Команда (Command) – инкапсуляция запроса в объект (например, действия в текстовом редакторе: копирование, вставка).
- Итератор (Iterator) – перебор коллекции без раскрытия её внутренней структуры.

Для Наблюдателя: создать класс NewsPublisher (издатель) и классы Subscriber (наблюдатели). При публикации новости все подписчики получают уведомление.

Для Команды: реализовать простой редактор с командами CopyCommand, PasteCommand и менеджером команд (возможность отмены).

Для Итератора: создать собственную коллекцию (например, список пользователей) и реализовать итератор для обхода.

Порядок выполнения:

Создать EventEmitter-подобный класс (или использовать нативные механизмы).

Реализовать методы subscribe, unsubscribe, notify.

Создать классы команд с методом execute() и undo().

Класс CommandManager хранит стек выполненных команд и отменяет последнюю.

Создать класс UserList с внутренним массивом и методом createIterator(), возвращающим объект с методами next(), hasNext().

Продемонстрировать работу всех трёх паттернов.

Контрольные вопросы:

1. Какую связь между объектами обеспечивает Наблюдатель?
2. Зачем нужен шаблон Команда, если можно просто вызвать метод?
3. Как Итератор способствует инкапсуляции?
4. В чём разница между Наблюдателем и Медиатором?
5. Как реализовать отмену нескольких команд с помощью Команды?

Лабораторная работа 5. Методика применения типовых решений проектирования

Цель работы: освоить методику выбора и применения паттернов в конкретной задаче (проектирование системы заказа такси).

Задание:

Разработать архитектуру приложения для заказа такси, используя не менее трёх паттернов проектирования (из любых групп).

Обосновать выбор каждого паттерна.

Реализовать прототип на JavaScript (можно без графического интерфейса, консольное приложение).

Документировать архитектуру с помощью диаграммы классов (схематично в тексте или с использованием PlantUML).

Порядок выполнения:

Проанализировать предметную область: заказ такси, расчет стоимости, выбор класса автомобиля, стратегии оплаты, уведомления водителя.

Выбрать паттерны, например:

- Стратегия для расчета стоимости (разные тарифы).
- Наблюдатель для уведомления водителя и клиента.
- Строитель для формирования заказа (опционально).
- Фабричный метод для создания разных типов автомобилей.

Описать, почему выбран именно этот набор.

Написать код: классы заказа, тарифов, диспетчера, наблюдателей.

Продемонстрировать работу: создать заказ, изменить тариф, отправить уведомления.

Приложить текстовое описание применения каждого паттерна.

Контрольные вопросы:

1. Какие критерии влияют на выбор паттерна?
2. Почему не стоит применять паттерны «на всякий случай»?
3. Какие риски возникают при неправильном выборе архитектурного

паттерна?

4. Как проверить, что выбранный паттерн решает проблему, а не усложняет?

5. Приведите пример замены одного паттерна на другой с изменением структуры кода.

Лабораторная работа 6. Виды и примеры шаблонов, не зависящих от языка программирования. Виды и примеры шаблонов, специфичных для языка программирования

Цель работы: изучить разницу между языконезависимыми архитектурными/конструктивными шаблонами и языко-специфичными идиомами на примере C++ или JavaScript.

Задание:

Выбрать один язык программирования (рекомендуется JavaScript).

Реализовать один языконезависимый паттерн (например, Мост) на выбранном языке.

Реализовать идиому, специфичную для этого языка (например, в JavaScript – использование замыканий для приватных полей; в C++ – RAII). В нашем случае использовать JavaScript идиому: модуль с помощью замыкания или класса с приватными полями (ES2022).

Сравнить: как один и тот же концептуальный паттерн Мост выглядит в JavaScript vs в гипотетическом другом языке.

Написать отчёт с примерами кода и комментариями.

Порядок выполнения:

Изучить примеры языконезависимых паттернов и идиом (например, идиома «Модуль» в JavaScript через IIFE или # приватные поля).

Реализовать Мост на JavaScript: абстракция «Пульт управления» и реализация «Устройство» (телевизор, радио).

Реализовать идиому «Приватные поля с использованием замыкания» для создания счётчика с инкапсуляцией.

Объяснить, почему реализация Моста возможна в любом ОО-языке, а идиома зависит от синтаксиса JavaScript.

Продемонстрировать выполнение.

Контрольные вопросы:

1. Что такое идиома программирования?
2. Приведите примеры идиом из C++ (RAII) и Java (try-with-resources).
3. Почему паттерн Адаптер не является идиомой?

4. Как идиомы влияют на читаемость кода?
5. Можно ли перенести идиому JavaScript «замыкание» на Python?

Лабораторная работа 7. Основы компьютерной графики

Цель работы: познакомиться с основами работы с растровой и векторной графикой программными средствами; научиться выполнять базовые операции: изменение цвета, рисование примитивов, наложение текста.

Задание:

Используя HTML5 Canvas API, создать веб-страницу с холстом 800x400.

Реализовать инструменты для рисования:

- Прямоугольник (залитый и контур).
- Линия.
- Окружность.

Реализовать возможность выбора цвета и толщины линии.

Добавить кнопку очистки холста.

Сделать подпись под холстом с пояснением, какой тип графики (растровая или векторная) используется в Canvas.

Порядок выполнения:

Создать HTML-файл с элементом <canvas> и панелью управления.

Написать JavaScript-код для обработки событий мыши (mousedown, mousemove, mouseup).

Реализовать функции рисования примитивов (прямоугольник от точки до точки, линия, окружность по центру и радиусу).

Использовать context.fillStyle, context.strokeStyle для цвета, context.lineWidth для толщины.

Добавить радиокнопки или выпадающий список для выбора фигуры.

Очистка холста – заливка белым цветом.

Проверить работу в браузере.

Контрольные вопросы:

1. Какой тип графики (растровый или векторный) генерируется при рисовании на Canvas? Почему?
2. Какое свойство отвечает за прозрачность линий?
3. Как сохранить рисунок в файл (метод toDataURL)?
4. Какие недостатки у растровой графики при многократном масштабировании?
5. Чем Canvas отличается от SVG?

Лабораторная работа 8-9. Веб-дизайн. Основы визуального дизайна. Принципы юзабилити

Цель работы: разработать макет многостраничного веб-сайта (например, портфолио или интернет-магазина) с учётом принципов визуального дизайна, информационной архитектуры и юзабилити.

Задание:

Создать прототип сайта (не менее 3-х страниц) с использованием HTML5 и CSS.

Реализовать:

- Горизонтальное или вертикальное меню навигации.
- Хлебные крошки.
- Форму обратной связи.
- Карточки товаров или услуг.

Применить правила композиции (сетка, выравнивание), контрастные цветовые схемы, читаемую типографику.

Обеспечить адаптивность (медиа-запросы для планшетов и мобильных).

Провести самооценку юзабилити по контрольному списку (размер шрифта, контраст, обратная связь при наведении и фокусе).

Порядок выполнения:

Разработать структуру папок: css, images, pages.

Создать общий CSS-файл стилей.

Использовать Flexbox или Grid для построения сетки.

Оформить заголовки, основную область, боковую панель, подвал.

Добавить псевдоклассы :hover, :focus для интерактивности.

Протестировать на разных разрешениях с помощью DevTools.

Оформить отчёт: скриншоты страниц, аргументация принятых дизайн-решений.

Контрольные вопросы:

1. Перечислите основные принципы юзабилити.
2. Как влияет цветовая схема на восприятие интерфейса?
3. Зачем нужны хлебные крошки?
4. Какие способы адаптивности вы знаете (резиновая, адаптивная, отзывчивая)?
5. Что такое «F-паттерн» чтения?

Лабораторная работа 10-11. Назначение и применение CSS

Цель работы: углублённое изучение каскадных таблиц стилей, приобретение навыков использования селекторов, псевдоклассов, позиционирования, анимаций.

Задание:

Создать одностраничный сайт-визитку (или интернет-магазин), используя:
Внешнюю таблицу стилей.
Контекстные селекторы и группировку.
Псевдоклассы (:nth-child, :not, :hover).
Псевдоэлементы (::before, ::after).
Анимацию или переходы (transition).
Реализовать следующее:

- Стилизованные списки с нестандартными маркерами.
- Таблицу с зеброй (чередование цвета строк).
- Карточки товара с эффектом поднятия при наведении.
- Блок с фиксированным (position: sticky) меню.
- Фоновое изображение с градиентом поверх.

Проверить валидность CSS и семантическую вёрстку.

Порядок выполнения:

Создать HTML-структуру: шапка, навигация, основная область с карточками, таблица характеристик, подвал.
Подключить внешний CSS.
В CSS задать базовые сбросы (reset или normalize).
Использовать Flex/Grid для расположения карточек.
Применить ::before для добавления иконок к пунктам меню.
Для таблицы использовать tr:nth-child(even).
Для меню задать position: sticky; top: 0;.
Добавить анимацию: плавное появление карточек при загрузке.
Проверить, что изменения стилей происходят без нарушения структуры.

Контрольные вопросы:

1. Как применить стиль ко всем элементам, кроме одного?
2. Чем отличается псевдокласс от псевдоэлемента?
3. Какие единицы измерения (rem, em, vw, vh) предпочтительны для адаптивного дизайна?
4. Что означает каскадирование в CSS?
5. Как разрешается конфликт при равной специфичности?

III. РАЗДЕЛ КОНТРОЛЯ ЗНАНИЙ

ТЕСТОВОЕ ЗАДАНИЕ

Тест

Компьютерная графика, веб-дизайн, технологии и CSS

Вопрос 1. Какой тип компьютерной графики описывает изображение с помощью геометрических примитивов (линий, кривых, многоугольников)?

А) Растровая графика
Б) Векторная графика
В) Фрактальная графика
Г) Пиксельная графика

Вопрос 2. Какая цветовая модель используется в мониторах и других устройствах, излучающих свет?

А) CMYK
Б) HSB
В) RGB
Г) Lab

Вопрос 3. Что из перечисленного является основным назначением каскадных таблиц стилей (CSS)?

А) Создание структуры веб-страницы
Б) Обработка запросов к серверу
В) Описание внешнего вида HTML-документа
Г) Хранение данных на стороне клиента

Вопрос 4. Какое свойство CSS позволяет задать фоновое изображение элемента?

А) background-color
Б) background-image
В) image-source
Г) img-background

Вопрос 5. Какой из перечисленных форматов файлов лучше всего подходит для размещения логотипа на веб-сайте с сохранением чёткости при любом масштабе?

А) JPEG
Б) PNG
В) GIF
Г) SVG

Вопрос 6. Что означает термин «адаптивный веб-дизайн»?

А) Использование только статических страниц
Б) Корректное отображение сайта на разных устройствах и размерах экрана
В) Обязательное использование Flash-анимации
Г) Применение только табличной вёрстки

Вопрос 7. Какой из способов подключения CSS имеет наивысший приоритет при конфликте правил?

А) Внешний файл .css
Б) Внутренний блок <style>
В) Инлайн-стиль (атрибут style)
Г) Стили браузера по умолчанию

Вопрос 8. Какой серверный язык программирования изначально был создан специально для веб-разработки и встраивается непосредственно в HTML?

А) Python
Б) C++
В) PHP
Г) Java

Вопрос 9. Что такое глубина цвета в компьютерной графике?

А) Количество пикселей на дюйм
Б) Количество бит, используемых для хранения цвета одного пикселя
В) Размер изображения в мегабайтах
Г) Яркость белого цвета

Вопрос 10. Какое CSS-свойство управляет типом отображения элемента (блочный, строчный, блочно-строчный)?

А) position
Б) display
В) float
Г) visibility

Вопрос 11. Какой формат графики поддерживает прозрачность и сжатие без потерь, часто используется для скриншотов и иконок?

А) JPEG
Б) BMP
В) PNG
Г) TIFF

Вопрос 12. Какая технология позволила отправлять асинхронные запросы к серверу без перезагрузки страницы, что привело к появлению одностраничных приложений?

А) ActiveX
Б) Java Applets
В) AJAX
Г) Silverlight

Вопрос 13. Что обозначает псевдокласс :hover в CSS?

А) Стиль для первого элемента в списке
Б) Стиль для элемента, на который наведён курсор мыши
В) Стиль для всех элементов с классом hover
Г) Стиль для посещённых ссылок

Вопрос 14. Какая система управления контентом (CMS) является самой популярной в мире и изначально создавалась для блогов?

А) Joomla
Б) Drupal
В) WordPress
Г) Tilda

Вопрос 15. Что такое цветовой охват устройства?

А) Количество цветов, которое устройство может отобразить или воспроизвести
Б) Яркость экрана в люменах
В) Разрешение печати в DPI
Г) Глубина цвета в битах

Вопрос 16. Какой тип позиционирования CSS удаляет элемент из нормального потока и привязывает его к окну браузера (не двигается при прокрутке)?

А) relative
Б) absolute
В) fixed
Г) static

Вопрос 17. Какое из перечисленного является преимуществом векторной графики перед растровой?

А) Лучшая передача фотореализма
Б) Неограниченное масштабирование без потери качества
В) Поддержка всех форматов сжатия
Г) Простота получения с камеры

Вопрос 18. Какой HTML-тег используется для подключения внешнего CSS-файла?

А) <css>
Б) <style>
В) <link>
Г) <script>

Вопрос 19. Что такое информационная архитектура веб-проекта?

А) Дизайн кнопок и иконок
Б) Организация и структурирование информации для удобного поиска и навигации
В) Выбор цветовой гаммы сайта
Г) Написание серверного кода

Вопрос 20. Какое CSS-свойство задаёт межстрочный интервал (расстояние между строками текста)?

А) letter-spacing
Б) line-height
В) text-indent
Г) word-spacing

ПРИМЕРНЫЙ ПЕРЕЧЕНЬ ВОПРОСОВ К ЭКЗАМЕНУ

1. Понятие, принципы и методология проектирования веб-приложений.
2. Интерфейсы, методы, инструменты и техника разработки веб-приложений.
3. Типовые решения и шаблоны проектирования: понятие, классификация, форматы описания.
4. Порождающие, структурные и поведенческие шаблоны проектирования (абстрактная фабрика, строитель, фасад, стратегия и др.).
5. Синергия паттернов, выбор и применение шаблонов проектирования.
6. Архитектурные, конструктивные и интеграционные шаблоны. Антишаблоны и идиомы.
7. Пример архитектурного паттерна – MVC. Преимущества использования шаблонов.
8. Предмет ОО-анализа и проектирования. Повторное использование проектных решений.
9. Основные концепции UML: визуализация, специфицирование, документирование.
10. Виды диаграмм UML: классов, активностей, взаимодействий, вариантов использования.
11. CASE-средства для построения UML-диаграмм.
12. Понятие графического процесса и геометрического моделирования. Типы моделей и их особенности.
13. Основные направления развития и применения компьютерной графики.
14. Виды графики: растровая, векторная, 3D-графика. Принципы представления и публикации.
15. Разрешение, форматы изображений и особенности экранного и полиграфического представления.
16. Программы для работы с графикой: пакеты растровой и векторной графики, интеграция и сравнение.
17. Определение и восприятие цвета.

18. Цветовые модели: RGB, CMYK, Lab, HSB и др.
19. Характеристики и управление цветом: глубина, гамма, профили, палитры.
20. Основы колористики и использование цвета в графике.
21. Понятие веб-дизайна и информационной архитектуры веб-проекта.
22. Основные принципы графического дизайна: композиция, контраст, форма, цвет, шрифт, текстура.
23. Законы и приёмы композиции, микро- и макротипографика.
24. Создание графики для Веб: оптимизация изображений, форматы, разрешение и цвет.
25. Свойства зрительного восприятия и эргономика интерфейсов.
26. Статические и динамические компоненты веб-приложений.
27. Серверные технологии: ASP, ASP.NET, PHP, Python, Perl и др.
28. Клиентские технологии: HTML, CSS, JavaScript, AJAX и современные фреймворки.
29. Информационные системы управления контентом (CMS), виды вёрстки (табличная, блочная, адаптивная).
30. Назначение CSS, способы подключения и основные приёмы оформления: селекторы, классы, псевдоэлементы, каскадирование, позиционирование, оформление текста, списков и таблиц.

IV. ВСПОМОГАТЕЛЬНЫЙ РАЗДЕЛ

ПЕРЕЧЕНЬ РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

1. Гамма, Э. Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. – СПб: «Питер», 2016.-366 с.

2. Даккет, Дж. HTML и CSS. Разработка и дизайн веб-сайтов / Джон Даккет. – СПб: «Питер». – 2014. – 480 с.

3. Лупптон, Э. Графический дизайн. Базовые концепции / Э. Лупптон. – СПб: Питер, 2019. – 256 с.

4. Сирых, Ю. Современный веб-дизайн. Настольный и мобильный / Ю. Сирых. – Киев: «Диалектика», 2019.-384 с.

5. Филичи, Дж. Типографика: шрифт, верстка, дизайн / Дж. Филичи. – СПб: БХВ-Петербург, 2020. – 496 с.