

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
СТАРООСКОЛЬСКИЙ ТЕХНОЛОГИЧЕСКИЙ ИНСТИТУТ ИМ. А.А. УГАРОВА
(филиал) федерального государственного автономного образовательного учреждения
высшего образования
«Национальный исследовательский технологический университет «МИСИС»
СТИ НИТУ «МИСИС»

Рабочая программа утверждена
решением Ученого совета
СТИ НИТУ «МИСИС»
от «23» июня 2026 г.
протокол № 38

Рабочая программа дисциплины

Архитектура ЭВМ

Закреплена за кафедрой	<u>Кафедра автоматизированных и информационных систем управления</u>
Направление подготовки	<u>им. Ю.И. Еременко</u>
Профиль	09.03.02 Информационные системы и технологии
	Информационные системы и технологии
Квалификация	<u>Бакалавр</u>
Форма обучения	<u>Очная</u>

Общая трудоемкость 4 ЗЕТ

Часов по учебному плану	<u>144</u>
в том числе:	
аудиторные занятия	<u>42</u>
самостоятельная работа	<u>66</u>
часов на контроль	<u>36</u>

Формы контроля в семестре: 3
Экзамен,

Семестр	4		Итого	
Вид занятий	УП	РП	УП	РП
Лекции	34	34	34	34
Лабораторные	17	17	17	17
Практические занятия	17	17	17	17
Сам. работа	40	40	40	40
Часов на контроль	36	36	36	36
Итого:	144	144	144	144

Год набора 2026г.

Программу составил: ассистент кафедры. АИСУ им.
Ю.И. Еременко
к.т.н., Жуков Пётр Игоревич

Должность, уч. ст., уч. зв. ФИО полностью



подпись

Рабочая программа дисциплины

Архитектура ЭВМ

наименование

Разработана в соответствии с ОС ВО НИТУ «МИСИС»:

Самостоятельно устанавливаемый образовательный стандарт высшего образования – бакалавриат
Федеральное государственное автономное образовательное учреждение высшего образования
«Национальный исследовательский технологический университет «МИСИС» по направлению подготовки
09.03.02 Информационные системы и технологии (приказ от 05.03.2020 г. № 95 о.в.)

Составлена на основании учебного плана 2026 года набора.

09.03.02 Информационные системы и технологии

Профиль: Информационные системы и технологии, утвержденного Ученым советом СТИ НИТУ
«МИСИС» 23.06.2026 г., протокол №38.

Рабочая программа одобрена на заседании кафедры

Автоматизированные и информационные системы управления

Протокол от «22» июня 2026 г. № 6

и.о. зав.
кафедрой

АИСУ им Ю.И.
Еременко

аббревиатура наименования кафедры



подпись

Д.А. Полещенко
И.О. Фамилия

«22» июня 2026г.

Руководитель ОПОП ВО

и.о. зав. кафедрой, кандидат технических
наук, доцент

должность, уч. ст., уч. зв.



подпись

Д.А. Полещенко
И.О. Фамилия

«22» июня 2026 г.

1. ЦЕЛИ ОСВОЕНИЯ	
Цель дисциплины – получить теоретические знания и практические навыки по основам проектирования и создания программных архитектур на ЭВМ. Сформировать представления об основных методологиях и принципах построения поддерживаемых, гибких программных архитектур ПО.	
Задачи дисциплины:	
1. Познакомить обучающихся с основными методологиями и принципами построения чистых программных архитектур ПО на современных архитектурах ЭВМ; 2. Научить обучающихся анализу и синтезу исходного кода на уровне его архитектуры в соответствии с внутренними принципами работы ЭВМ; 3. Научить обучающихся проектированию и реализации чистых архитектур ПО для современных ЭВМ.	

2. МЕСТО ДИСЦИПЛИНЫ В СТРУКТУРЕ ОПОП ВО	
Цикл (раздел) ОП:	Б1.В.
2.1	Требования к предварительной подготовке обучающихся:
2.1.1	Компьютерное обеспечение специальности
2.1.2	Проектирование информационных систем
2.1.3	Современные информационные технологии
2.1.4	Системный анализ
2.1.5	Интеллектуальные системы управления
2.2	Дисциплины и практики, для которых освоение данной дисциплины необходимо как предшествующее:
2.2.1	Научно-исследовательская работа
2.2.2	Преддипломная практика для выполнения выпускной квалификационной работы
2.2.3	Защита выпускной квалификационной работы, включая подготовку к процедуре защите и процедуру защиты

3. ИНДИКАТОРЫ ФОРМИРОВАНИЯ КОМПЕТЕНЦИЙ, СОВМЕЩЕННЫЕ С РЕЗУЛЬТАТАМИ ОБУЧЕНИЯ ПО ДИСЦИПЛИНЕ	
УК-3: Способен эффективно обмениваться информацией, идеями, проблемами и решениями с инженерным сообществом и обществом в целом, осуществлять социальное взаимодействие и реализовывать свою роль в команде	
Знать:	УК-3-31 – Устройство программной архитектуры ПО и современные принципы программирования.
Уметь:	УК-3-У1 – Применять средства анализа и описания программной архитектуры ПО.
Владеть:	УК-3-В1 – Способами проектирования и организации программной архитектуры ПО.
ОПК-6 Способен выбирать и применять методики проектирования и актуальные инструментальные средства, разрабатывать алгоритмы и программы, пригодные для практического применения в области информационных систем и технологий	
Знать:	ОПК-6-31 – Современный подход к описанию программной архитектуры ПО.
Уметь:	ОПК-6-У1 – Проектировать и документировать программную архитектуру ПО.
Владеть:	ОПК-6-В1 – Унифицированным языком моделирования для описания программной архитектуры ПО.
ОПК-2: Способен понимать принципы работы современных информационных технологий и программных средств, в том числе отечественного производства, и использовать их при решении задач профессиональной деятельности	
Знать:	ОПК-2-31 – Устройство интегрированных сред разработки программного архитектуры ПО.
Уметь:	ОПК-2-У1 – Проводить отладку программной архитектуры на уровне компиляции и исполнения.
Владеть:	ОПК-2-В1 – Навыками работы с инструментарием отслеживания и устранения ошибок в коде.
ПК-1: Способен участвовать в работах по:	
• доводке и освоению информационных технологий в ходе внедрения и эксплуатации информационных технологий; • поддержанию работоспособности информационных систем и технологий в заданных функциональных характеристиках и в соответствии с критериями качества; • управлению программно-аппаратными средствами информационных служб инфокоммуникационной системы организации; - обеспечению безопасности и целостности данных информационных систем и технологий, а также условий жизненного цикла информационных систем; • адаптации приложений к изменяющимся условиям функционирования; • администрированию сетевой подсистемы инфокоммуникационной системы организации.	
Использовать технологии разработки объектов профессиональной деятельности в различных областях.	
Знать:	ПК-1-31 – Основные принципы эффективной чистой программной архитектуры
Уметь:	ПК-1-У1 – Находить нарушение принципов чистой архитектуры и устранять их
Владеть:	ПК-1-В1 – Базовыми навыками программирования на языке высокого уровня

4. СТРУКТУРА И СОДЕРЖАНИЕ						
Код занятия	Наименование разделов и тем /вид занятия/	Семестр	Часов	Компетенции	Литература и эл. ресурсы	Примечание
	Раздел 1. Введение в объекты ООП					
1.1	Основные понятия архитектуры. Качество кода. Проблемы и пути устранения /Лек/	3	4	УК-3-31 ОПК-6-31	Л 2.1, Л 3.1, Э 1, Э 3, Э 4	
1.2	Ссылочные и значимые типы / Пр /	3	2	ПК-1-В1 УК-3-В1 ОПК-6-У1 ОПК-6-В1	Л 3.1, Л 2.1, Э 2, Э 4	
1.3	Знакомство с эnumераторами в C# / Лаб /	3	4	ОПК-2-31 ПК-1-В1 УК-3-В1 ОПК-2-У1 ОПК-2-В1	Л 3.1, Л 2.1, Э 2, Э 3, Э 4	
1.4	Абстракции и интерфейсы /Пр /	3	2	ПК-1-В1 УК-3-В1 ОПК-6-У1 ОПК-6-В1	Л 3.1, Л 2.1, Э 2, Э 3, Э 4	
1.5	Мутабельные и имутабельные типы данных /Лаб/	3	4	ОПК-2-31 ПК-1-В1 УК-3-В1 ОПК-2-У1 ОПК-2-В1	Л 3.1, Л 2.1, Э 2, Э 3, Э 4	
1.6	Связи классов друг с другом. Агрегация и композиция / Пр /	3	2	ПК-1-В1 УК-3-В1 ОПК-6-У1 ОПК-6-В1	Л 3.1, Л 2.1, Э 2, Э 3, Э 4	
1.7	Изучение обобщенных типов / Лаб /	3	5	ОПК-2-31 ПК-1-В1 УК-3-В1 ОПК-2-У1 ОПК-2-В1	Л 3.1, Л 2.1, Э 2, Э 3, Э 4	
1.8	Статические объекты. Методы расширения / Пр /	3	2	ПК-1-В1 УК-3-В1 ОПК-6-У1 ОПК-6-В1	Л 3.1, Л 2.1, Э 2, Э 3, Э 4	
1.9	Знакомство с LINQ / Лаб /	3	4	ОПК-2-31 ПК-1-В1 УК-3-В1 ОПК-2-У1 ОПК-2-В1	Л 3.1, Л 2.1, Э 2, Э 3, Э 4	
1.10	Подготовка к защите лабораторных и практических работ / СР /	3	10	ОПК-2-31 ПК-1-В1 УК-3-В1 ОПК-2-У1 ОПК-2-В1	Л 3.1, Л 2.1, Э 2, Э 3, Э 4	
1.11	Выполнение домашнего задания «Реализация классовой структуры на основе UML-диаграммы» / СР /	3	10	ПК-1-В1 УК-3-В1 УК-3-У1 ОПК-6-У1 ОПК-6-В1	Л 3.1, Л 2.1, Э 2, Э 3, Э 4	
	Раздел 2. Принципы программной архитектуры					
2.1	ООП, инверсия зависимостей, SOLID, DRY, YAGNI /Лек/	3	10	УК-3-31 ОПК-6-31 ПК-1-31	Л 1.1, Л 2.1, Л 3.1, Э 1, Э 2, Э 3, Э 4	
2.2	Принципы дизайна кода: «CCP, REP и CRP». Принципы использования кода: «ADP, SDP, SAP» / Лек /	3	10	УК-3-31 ОПК-6-31 ПК-1-31	Л 1.1, Л 2.1, Л 3.1, Э 1, Э 2, Э 3, Э 4	
2.3	Некоторые методологии разработки программного обеспечения: «EDM, DDD, TDD» / Лек /	3	10	УК-3-31 ОПК-6-31 ПК-1-31	Л 1.1, Л 2.1, Л 3.1, Э 1, Э 2, Э 3, Э 4	
2.4	Изучение принципов SRP, OCP и LSP / Пр /	3	3	ПК-1-В1 УК-3-В1 УК-3-У1 ОПК-6-У1	Л 3.1, Л 2.1, Э 2, Э 3, Э 4	

				ОПК-6-В1		
2.5	Изучение принципов <i>ISP</i> и <i>DIP</i> / Пр /	3	3	ПК-1-В1 УК-3-В1 УК-3-У1 ОПК-6-У1 ОПК-6-В1	Л 3.1, Л 2.1, Э 2, Э 3, Э 4	
2.6	Рефакторинг кода / Пр /	3	4	ПК-1-В1 ПК-1-У1 УК-3-У1 УК-3-В1	Л 3.1, Л 2.1, Э 2, Э 3, Э 4	
2.7	Подготовка к защите лабораторных и практических работ / СР /	3	10	ОПК-2-31 ПК-1-В1 УК-3-В1 ОПК-2-У1 ОПК-2-В1	Л 3.1, Л 2.1, Э 2, Э 3, Э 4	
2.8	Выполнение домашнего задания «Рефакторинг нерабочего кода» / СР /	3	10	ПК-1-В1 УК-3-В1 УК-3-У1 ОПК-6-У1 ОПК-6-В1	Л 3.1, Л 2.1, Э 2, Э 3, Э 4	

5. ФОНД ОЦЕНОЧНЫХ МАТЕРИАЛОВ

5.1. Вопросы для самостоятельной подготовки к экзамену (зачёту с оценкой)

5.1.1. Перечень вопросов для подготовки к экзамену. (промежуточный контроль успеваемости)

Раздел 1. Введение в объекты ООП (УК-3-31, ОПК-6-31)

1. Что такое объектно-ориентированное программирование (ООП) и какие основные принципы лежат в его основе?
2. Что такое класс и объект в контексте объектно-ориентированного программирования?
3. Что такое инкапсуляция и как она реализуется в ООП?
4. Что такое наследование и как оно используется в ООП?
5. Что такое полиморфизм и какие его виды существуют в ООП?
6. Что такое класс в C# и как он отличается от структуры (struct)?
7. Какие особенности и ограничения у структур в C#?
8. Как осуществляется инициализация полей в структурах и классах в C#?
9. Почему рекомендуется использовать свойства вместо публичных полей в структурах?
10. Как передаются объекты классов и структуры по значению в C#?
11. Какие основные виды связей существуют между классами в объектно-ориентированном программировании?
12. Как реализуется ассоциация между классами и какие её разновидности бывают?
13. Что такое агрегация и композиция как виды связи между классами, и в чем их различия?
14. Как осуществляется наследование и какие типы наследования существуют в объектно-ориентированном программировании?
15. Что такое зависимость между классами и как она проявляется в коде?
16. Что такое статические объекты и как они отличаются от обычных экземпляров классов в C#?
17. Как и где можно использовать статические методы в C# и какие у них ограничения?
18. Что такое методы расширения и в каких случаях их применение наиболее целесообразно?
19. Как реализуются статические конструкторы и для чего они используются в классах?
20. Можно ли создавать статические классы и какие особенности, и ограничения у таких классов в C#?

Раздел 2. Принципы программной архитектуры (УК-3-31, ОПК-6-31, ПК-1-31)

1. Что такое инверсия зависимостей и как она реализуется в программировании?
2. Какие преимущества предоставляет принцип инверсии зависимостей в объектно-ориентированном программировании?
3. Как принцип *SOLID* помогает в создании гибкого и поддерживаемого кода?
4. Что означает принцип *DRY* (*Don't Repeat Yourself*) и как его можно применять в разработке программного обеспечения?
5. Какие проблемы решает принцип *DRY* и как он влияет на качество кода?
6. В чем заключается принцип *YAGNI* (*You Aren't Gonna Need It*) и как он помогает в управлении техническим долгом?
7. Почему важно избегать жёсткого связывания в зависимостях и как это связано с принципом инверсии зависимостей?
8. Как принцип разделения интерфейсов (*Interface Segregation Principle*) помогает в проектировании классов?
9. Что такое принцип подстановки Лисков и как он влияет на наследование в объектно-ориентированном программировании?
10. Как принцип открытости/закрытости (*Open-Closed Principle*) способствует поддержке и расширению кода?
11. Какие преимущества даёт использование интерфейсов в реализации принципа инверсии зависимостей?
12. Как принцип *YAGNI* влияет на процесс разработки и принятия решений в программировании?
13. Почему важно следовать принципу *DRY* при работе с кодом и какие последствия могут быть при его нарушении?
14. Как инверсия зависимостей помогает в тестировании кода и какие преимущества это предоставляет?
15. Какие примеры нарушения принципов *SOLID* и *DRY* можно привести и как их избежать в практике разработки?
16. Что такое принцип *CCP* (*Common Closure Principle*) и как он применяется в дизайне кода?
17. Как принцип *CCP* помогает в управлении зависимостями и поддержке кода?
18. В чем заключается принцип *REP* (*Release Equivalence Principle*) и как он влияет на модульность в программном обеспечении?
19. Почему важно следовать принципу *REP* при разработке библиотек и фреймворков?
20. Как принцип *CRP* (*Common Reuse Principle*) способствует повторному использованию кода в различных контекстах?
21. Какие преимущества предоставляет принцип *CRP* при проектировании архитектуры системы?
22. Что такое принцип *ADP* (*Abstract Data Principle*) и как он используется для скрытия деталей реализации?
23. Как принцип *ADP* помогает в упрощении интерфейса и улучшении тестируемости кода?
24. В чем заключается принцип *SDP* (*Stable Dependencies Principle*) и как он влияет на управление зависимостями?
25. Почему важно следовать принципу *SDP* при выборе библиотек и фреймворков для проекта?
26. Как принцип *SAP* (*Stable Abstractions Principle*) способствует созданию гибких и устойчивых абстракций?
27. Какие преимущества предоставляет принцип *SAP* при разработке расширяемых систем?
28. Что такое принцип *CCP* и как он связан с организацией классов и модулей в проекте?
29. Как применение принципов *REP* и *CRP* может повлиять на структуру и взаимодействие компонентов в системе?
30. Почему принципы *ADP*, *SDP* и *SAP* важны для обеспечения гибкости и масштабируемости кода в долгосрочной перспективе?

31. Что такое *EDM (Entity Data Model)* и как он используется в проектировании баз данных?
32. Какие преимущества предоставляет паттерн *EDM* при работе с большими объемами данных?
33. Как паттерн *DDD (Domain-Driven Design)* помогает в моделировании бизнес-логики?
34. Какие ключевые элементы включает в себя паттерн *DDD* и как они взаимодействуют между собой?
35. В чем заключаются основные принципы *TDD (Test-Driven Development)* и как они влияют на процесс разработки?
36. Как *TDD* помогает в обеспечении качества и надежности кода?
37. Какие этапы включает в себя цикл разработки в *TDD* и почему они важны?
38. Что такое агрегаты в контексте *DDD* и как они управляют транзакциями?
39. Как паттерн *DDD* способствует лучшему пониманию и управлению доменными моделями?
40. Какие преимущества дает использование *EDM* для моделирования данных в корпоративных приложениях?
41. Как *TDD* влияет на дизайн и структуру тестов в проекте?
42. Что такое ограниченные контексты в *DDD* и как они помогают разделять ответственность в больших системах?
43. Как паттерн *EDM* может быть использован для интеграции различных источников данных?
44. Почему важно следовать принципам *DDD* при разработке сложных бизнес-приложений?
45. Как *TDD* способствует улучшению взаимодействия между разработчиками и тестировщиками в команде?

5.1.2. Перечень контрольных заданий, используемых при формировании экзаменационных билетов при оценке знаний обучающихся на экзамене

Раздел 1. Введение в объекты ООП (УК-3-31, ОПК-6-31)

1. Напишите программу, содержащую абстрактный базовый класс с защищенным полем, являющимся ссылкой на целочисленный массив. Класс должен иметь конструктор с одним аргументом (определяет размер массива и создает его), целочисленное свойство (значение - размер массива), абстрактный метод (без аргументов, не возвращает результат) и индексатор с целочисленным индексом (доступен для чтения и записи). В производном классе опишите абстрактный метод из базового класса, чтобы он отображал в консоли содержимое массива;
2. Создайте интерфейс с одним методом, который принимает целочисленный аргумент и возвращает целое число. Реализуйте этот интерфейс в двух различных классах, каждый из которых должен предоставить свою реализацию метода;
3. Напишите программу, содержащую абстрактный класс с двумя защищенными целочисленными полями, и конструктор с двумя целочисленными аргументами. В классе должен быть объявлен абстрактный индексатор с целочисленным индексом. Опишите интерфейс, в котором есть метод с целочисленным аргументом и целочисленным результатом. Опишите класс, который наследует абстрактный базовый класс и реализует интерфейс. В этом классе опишите индексатор так, чтобы при четном индексе выполнялось обращение к первому полю, а при нечетном индексе обращение выполнялось ко второму полю;
4. Создайте класс, который реализует два интерфейса. Оба интерфейса должны содержать метод с одинаковой сигнатурой. Убедитесь, что класс корректно реализует оба метода;
5. Создайте интерфейс *Comparable* с методом *CompareTo*. Реализуйте этот интерфейс в классе *Person*, добавив метод *CompareTo*, который сравнивает объекты класса *Person* по возрасту;
6. Создайте интерфейс *IDrawable* с методом *Draw* и интерфейс *IShape* с методом *GetArea*. Реализуйте оба интерфейса в классе *Circle*, обеспечив реализацию методов *Draw* и *GetArea*.
7. Определите классы *Book* и *Author*. Установите связь один-ко-многим между ними, добавив свойство *List<Author>* в класс *Book* и свойство *Book* в класс *Author*. Реализуйте методы для добавления и удаления авторов у книги.
8. Создайте классы *Student* и *Course*. Определите ассоциацию многие-ко-многим с помощью класса-посредника *Enrollment*. Реализуйте методы для добавления и удаления студентов в курсе и наоборот.
9. Создайте класс *Car*, который содержит объект *Engine*. Реализуйте композицию, обеспечив корректное создание и удаление объектов *Engine* при создании и уничтожении объектов *Car*.
10. Создайте интерфейс *IWalkable* с методом *Walk*. Реализуйте этот интерфейс в абстрактном классе *Animal*, добавив абстрактный метод *Walk*. Создайте производные классы *Dog* и *Cat*, которые реализуют метод *Walk*.
11. Определите класс *University* с коллекцией студентов *List<Student>*. Установите связь один-ко-многим между университетом и студентами, реализовав методы для добавления и удаления студентов.
12. Создайте класс *Employee*, который реализует интерфейсы *Comparable* и *IDisposable*. Реализуйте методы *CompareTo* и *Dispose* в классе *Employee*.
13. Создайте интерфейс *IShape* с абстрактным методом *CalculateArea*. Реализуйте этот интерфейс в классах *Circle* и *Rectangle*, обеспечив реализацию метода *CalculateArea* для каждого из них.
14. Определите интерфейс *IEventHandler* с событием *BookReleased*. Реализуйте этот интерфейс в классе *Publisher*. Создайте класс *Subscriber*, который подписывается на событие *BookReleased* и выполняет определенные действия при его возникновении.

Раздел 2. Принципы программной архитектуры (УК-3-31, ОПК-6-31, ПК-1-31)

1. Создайте класс *Car*, который отвечает за управление автомобилем. Убедитесь, что класс не содержит методов, не связанных напрямую с управлением автомобилем (например, методов для расчета налогов).
2. Создайте базовый класс *Shape* с методом *CalculateArea*. Реализуйте этот метод в производных классах *Circle* и *Rectangle*. Убедитесь, что можно добавлять новые фигуры, наследуя от *Shape*, без изменения исходного кода *Shape*.
3. Создайте базовый класс *Animal* с методом *MakeSound*. Создайте производные классы *Dog* и *Cat*, которые реализуют метод *MakeSound*. Убедитесь, что объекты классов *Dog* и *Cat* могут быть использованы вместо объектов класса *Animal* без изменения логики программы.

4. Создайте интерфейс *I LogManager* с методом *ReadLog*. Реализуйте два класса *JSONLogManager* и *FileLogManager*, которые будут читать лог из JSON файла и бинарного файла соответственно. Убедитесь, что классы использующие *I LogManager* зависят от абстракций, а не от конкретных реализаций.
5. Разделите класс *Employee* на два класса: *EmployeeData* для хранения данных о сотруднике и *EmployeeLogic* для бизнес-логики. Убедитесь, что каждый класс имеет одну ответственность и инкапсулирует свои обязанности.
6. Создайте базовый класс *Shape* с методом *CalculateArea*. Реализуйте этот метод в производных классах *Circle* и *Rectangle*. Добавьте новый класс *Triangle*, наследуя от *Shape*, и убедитесь, что его можно добавить без изменения кода *Shape*.
7. Создайте базовый класс *Shape* с методом *GetArea*. Реализуйте этот метод в производных классах *Circle* и *Rectangle*. Убедитесь, что объекты классов *Circle* и *Rectangle* могут быть использованы в коде, который ожидает объекты класса *Shape*.
8. Создайте интерфейс *IDrawable* с методами *Draw* и *Erase*. Создайте классы *Circle* и *Rectangle*, которые реализуют только те методы интерфейса *IDrawable*, которые им необходимы. Убедитесь, что изменение методов в интерфейсе *IDrawable* не влияет на классы *Circle* и *Rectangle*.
9. Создайте интерфейс *I Logger* с методом *Log*. Создайте классы *ConsoleLogger* и *FileLogger*, которые реализуют интерфейс *I Logger*. Убедитесь, что классы, использующие *I Logger*, зависят от абстракции, а не от конкретных реализаций.
10. Создайте интерфейс *I Logger* с методом *Log*. Создайте классы *ConsoleLogger* и *FileLogger*, которые реализуют интерфейс *I Logger*. Убедитесь, что классы, использующие *I Logger*, зависят от абстракции, а не от конкретных реализаций.
11. Разработайте доменную модель для управления библиотекой. Определите основные сущности (например, *Book*, *Reader*, *Library*), их атрибуты и связи между ними. Убедитесь, что модель отражает реальные бизнес-правила и инварианты. Затем создайте репозитории для работы с созданными классами. Убедитесь, что репозиторий отделен от конкретной реализации хранения данных и может работать с различными источниками.
12. Разработайте слой сервисов для управления библиотекой, который будет содержать бизнес-логику высокого уровня. Определите сервисы (например, *CheckOutBook*, *ReturnBook*, *SearchBooks*), которые будут использовать репозитории для выполнения операций. Убедитесь, что сервисы не содержат логики доступа к данным и фокусируются на бизнес-правилах.
13. Создайте интерфейс *PaymentGateway* и две реализации этого интерфейса: *CreditCardGateway* и *BankTransferGateway*. Напишите классы, которые зависят от интерфейса *PaymentGateway*, но не от конкретных реализаций. Убедитесь, что ваши классы используют абстракции, а не конкретные реализации.
14. Создайте интерфейс *IDataAccess* с несколькими методами для работы с данными. Разделите этот интерфейс на несколько более мелких интерфейсов (*IReadData*, *IWriteData*, *ICalculateData*), чтобы классы реализовывали только те методы, которые им необходимы. Убедитесь, что зависимости классов минимальны и специфичны.
15. Создайте структуру зависимостей, где высокоуровневые компоненты зависят от низкоуровневых через абстракции. Убедитесь, что зависимости направлены сверху вниз, минимизируя циклические зависимости и улучшая понимание структуры проекта.
16. Создайте интерфейс *IOrderService* с методами для обработки заказов. Реализуйте этот интерфейс в классе *DefaultOrderService*. Напишите класс *OrderProcessor*, который зависит от интерфейса *IOrderService*. Убедитесь, что изменения в одной реализации не влияют на другую реализацию, и каждая реализация может быть заменена независимо.

5.2. Перечень работ, выполняемых по дисциплине

В рамках данного курса предполагается выполнение следующей учебной нагрузки.

Лабораторные работы (ОПК-2-31, ПК-1-В1, УК-3-В1, ОПК-2-У1, ОПК-2-В1):

- Лабораторная работа №1. Знакомство с эnumераторами в C#;
- Лабораторная работа №2. Знакомство с мутабельными и иммутабельными типами;
- Лабораторная работа №3. Знакомство с обобщенными типами;
- Лабораторная работа №4. Знакомство с *LINQ*;

Практические работы (ПК-1-В1, УК-3-В1, УК-3-У1, ОПК-6-У1, ОПК-6-В1):

- Практическая работа №1. Ссылочные и значимые типы;
- Практическая работа №2. Абстракции и интерфейсы;
- Практическая работа №3. Абстракции и интерфейсы;
- Практическая работа №4. Статические объекты. Методы расширения;
- Практическая работа №5. Изучение принципов *SRP*, *ОСР* и *LSP*;
- Практическая работа №6. Изучение принципов *ISP* и *DIP*;
- Практическая работа №7. Рефакторинг кода;

Домашнее задание (ПК-1-В1, УК-3-В1, УК-3-У1, ОПК-6-У1, ОПК-6-В1)

- Домашнее задание №1 Реализация классовой структуры на основе *UML*-диаграммы;
- Домашнее задание №2 Рефакторинг нерабочего кода.

5.2.1. ПЕРЕЧЕНЬ ВОПРОСОВ К ЗАЩИТЕ ЛАБОРАТОРНЫХ И ПРАКТИЧЕСКИХ РАБОТ

Практическая работа №1 «Ссылочные и значимые типы»

1. Чем различаются значимые и ссылочные типы данных? К какому типу данных относятся перечисления в C#?
2. Что такое *stackalloc* в C#? Когда его можно использовать?

3. В чем существенный недостаток ссылочных типов?
4. В чем существенный недостаток значимых типов?
5. В чем преимущества значимых и ссылочных типов?

Практическая работа №2 «Абстракции и интерфейсы»

1. Чем различаются значимые и ссылочные типы данных? К какому типу данных относятся перечисления в C#?
2. Что такое `stackalloc` в C#? Когда его можно использовать?
3. В чем существенный недостаток ссылочных типов?
4. В чем существенный недостаток значимых типов?
5. В чем преимущества значимых и ссылочных типов?

Практическая работа №3 «Связи классов друг с другом. Агрегация и композиция»

1. Объясните концепцию композиции и агрегации. Как они влияют на жизненный цикл объектов?
2. Что такое ассоциация и какие типы ассоциаций существуют?
3. Объясните разницу между наследованием и реализацией?

Практическая работа №4 «Статические объекты. Методы расширения»

1. Что такое статический класс и какие особенности его использования в C#? Можно ли создавать экземпляры статического класса?
2. Как и для чего применяются статические методы в C#? Приведите пример. В чем преимущество вызова статического метода без создания объекта?
3. Как и для чего применяются статические методы в C#? Приведите пример кода. В чем преимущество вызова статического метода без создания объекта?
4. Какие ограничения накладываются на статические методы и данные в C#? Почему эти ограничения важны? Можно ли вызвать нестатический метод из статического метода в том же классе?
5. В каких случаях рекомендуется использовать статические классы и методы расширения? Как статические методы могут улучшить производительность приложения?

Практическая работа №5 «Изучение принципов SRP OCP и LSP»

1. Что гласит принцип единственной ответственности (Single Responsibility Principle, SRP)?
2. Почему объединение нескольких сущностей, которые могут меняться по разным причинам, считается плохим проектным решением в контексте SRP?
3. Как можно изменить поведение класса, не внося физических изменений в сам класс, следуя принципу открытости/закрытости (Open Closed Principle, OCP)?
4. Что означает фраза “программные сущности должны быть открыты для расширения, но закрыты для изменения” в контексте принципа OCP?
5. Как принцип подстановки Барбары Лисков (Liskov’s Substitution Principle) связан с принципами SRP и OCP в контексте объектно-ориентированного программирования?
6. Какие проблемы могут возникнуть при нарушении принципа LSP в иерархии наследования?
7. Как принцип LSP влияет на полиморфное поведение объектов в программе?
8. Какие практические примеры нарушения принципа LSP можно привести и как это влияет на код?
9. Как принцип LSP соотносится с концепцией абстракции в объектно-ориентированном программировании?
10. Какие рекомендации можно дать разработчикам для соблюдения принципа LSP при проектировании классов и интерфейсов?

Практическая работа №6 «Изучение принципов ISP и DIP»

1. В чем заключается принцип разделения интерфейса (Interface Segregation Principle, ISP) и как он помогает в поддержании гибкости системы?
2. Почему использование “толстых” интерфейсов считается нежелательным в объектно-ориентированном программировании согласно принципу ISP?
3. Как принцип инверсии зависимостей (Dependency Inversion Principle, DIP) влияет на структуру зависимостей в проекте?
4. Какие преимущества дает применение принципа DIP в проектировании, особенно в контексте устойчивости к изменениям?
5. Как принцип разделения интерфейсов (ISP) связан с принципом инверсии зависимостей (DIP) в процессе разработки программного обеспечения?

Практическая работа №7 «Рефакторинг кода»

1. Что такое рефакторинг кода и какие основные цели он преследует?
2. Какие основные причины побуждают разработчиков проводить рефакторинг существующего кода?
3. Какие преимущества дает рефакторинг кода для улучшения читаемости и тестируемости?
4. Какие методы и техники рефакторинга можно использовать для снижения технического долга в проекте?
5. Какие риски и побочные эффекты могут возникнуть при проведении рефакторинга и как их можно минимизировать?

Лабораторная работа № 1 «Знакомство с эnumераторами в C#»

1. Что такое интерфейсы “`IEnumerable`” и “`IEnumerator`” в C# и как они используются для реализации перечисления коллекций?
2. Как работает метод “`GetEnumerator`” в контексте получения эnumератора для коллекции и как он связан с интерфейсом “`IEnumerable`”?

3. Какие методы должен реализовывать интерфейс "IEnumerator" и для чего они предназначены (например, "MoveNext" и "Current")?
4. В чем заключается роль эnumераторов при обходе коллекций в многопоточных сценариях и как они представляют состояние обхода?
5. Как коллекции в C# реализуют концепцию полиморфизма при работе с эnumераторами и какие преимущества это предоставляет разработчикам?
6. Как работает конструкция "foreach" в C# и какие типы данных она может обрабатывать (например, массивы, списки и т.д.)?
7. Какие неявные преобразования происходят при использовании "foreach" и как они связаны с интерфейсами "IEnumerable" и "IEnumerator"?
8. Как осуществляется итерация по элементам коллекции в "foreach" и какие методы вызываются за кулисами?
9. Можно ли использовать "foreach" для изменения элементов коллекции и какие ограничения при этом существуют?
10. В чем заключаются преимущества использования "foreach" по сравнению с традиционными циклами "for" и "while" в C#?

Лабораторная работа №2 «Знакомство с мутабельными и иммутабельными типами»

1. Чем отличаются иммутабельные и мутабельные типы в C#? Объясните основные различия между иммутабельными и мутабельными типами, приводя примеры кода.
2. Почему строки (string) являются иммутабельными в C#? Опишите, почему строки в C# иммутабельны и как это влияет на производительность и управление памятью.
3. Как создать иммутабельный класс в C#? Приведите пошаговую инструкцию по созданию иммутабельного класса, включая использование readonly переменных и конструкторов.
4. В чем преимущества использования мутабельных типов, таких как StringBuilder? Перечислите преимущества мутабельных типов на примере StringBuilder и как они влияют на производительность приложения.
5. Как мутабельные типы влияют на память в сравнении с иммутабельными? Объясните, как мутабельные и иммутабельные типы управляют памятью: создание новых экземпляров, использование одной и той же памяти и влияние на «сборщик мусора».
6. В каких случаях предпочтительнее использовать иммутабельные классы вместо мутабельных? Приведите примеры ситуаций, где использование иммутабельных классов может быть более выгодным, и объясните, почему.

Лабораторная работа №3 «Знакомство с обобщенными типами»

1. Как правильно определить и использовать generic методы в C#?
2. Какие ограничения можно накладывать на generic типы в C# и зачем они нужны?
3. Как реализовать интерфейсы в generic методах и какие преимущества это дает?
4. В чем разница между использованием where T : class и where T : struct в generic ограничениях?
5. Как обрабатывать случаи, когда необходимо ограничить generic тип базовым классом или интерфейсом?
6. Что такое рефлексия в C# и какие основные задачи она позволяет решать?
7. Какие основные компоненты включает в себя система рефлексии в C#?
8. Как с помощью рефлексии можно получить информацию о классе, методах и полях во время выполнения программы?
9. Что такое динамически создаваемые объекты и как рефлексия помогает в их создании?
10. Как использовать рефлексия для вызова методов по их имени, определенному во время выполнения?
11. Какие преимущества предоставляет рефлексия для повышения гибкости и настраиваемости приложений?
12. Какие основные классы и интерфейсы используются в рефлексии C# и какие методы они предоставляют?

Лабораторная работа №4 «Знакомство с LINQ»

1. Что такое LINQ и какие основные задачи он выполняет в языке C#?
2. Какие основные типы LINQ существуют и для работы с какими данными они предназначены?
3. В чем разница между LINQ to Objects, LINQ to SQL, LINQ to Entities и другими разновидностями LINQ?
4. Что такое лямбда-выражения и как они используются в LINQ-запросах?
5. Какие операторы запросов поддерживаются в LINQ и как они помогают в построении сложных запросов?
6. Что такое анонимные методы и как они применяются в контексте LINQ?
7. Какие преимущества дает использование LINQ по сравнению с традиционными методами работы с данными в C#?
8. Что такое отложенные запросы в LINQ и в чем их основное отличие от немедленных запросов? Объясните концепцию отложенных запросов и приведите пример, когда они могут быть полезны.
9. Как работают отложенные запросы при изменении источника данных после их создания? Опишите, как изменение исходного списка данных может повлиять на результаты отложенного запроса и почему это происходит.
10. Можете ли вы объяснить, в каких случаях выполнение запроса откладывается до момента перечисления результата? Приведите примеры кода, где выполнение запроса откладывается, и объясните, когда именно происходит выполнение.
11. Как отложенные запросы влияют на производительность по сравнению с немедленными запросами? Обсудите, как использование отложенных запросов может повлиять на производительность приложения и в каких ситуациях это может быть выгодно.
12. Как можно использовать отложенные запросы для оптимизации работы с большими объемами данных? Приведите пример, где применение отложенных запросов позволяет улучшить производительность при работе с большими списками данных.

5.2.2 ПЕРЕЧЕНЬ ВОПРОСОВ ДЛЯ ЗАЩИТЫ ДОМАШНЕГО ЗАДАНИЯ

Домашнее задание №1

1. Какие основные элементы включает в себя диаграмма классов UML и как они используются для моделирования структуры системы?
2. Как изображается ассоциация между двумя классами на UML диаграмме и какие типы ассоциаций существуют?
3. В чем заключается принцип композиции и как он отличается от агрегации в контексте UML диаграмм?
4. Как реализуются интерфейсы в UML диаграммах классов и какие преимущества они предоставляют при проектировании объектно-ориентированных систем?
5. Какие типы отношений наследования существуют в UML и как они помогают в организации иерархии классов?

Домашнее задание №2

1. Какие основные признаки указывают на необходимость проведения рефакторинга в коде, и как их можно идентифицировать на ранних стадиях разработки?
2. Какие техники рефакторинга наиболее эффективны для улучшения читаемости и поддерживаемости кода, и как их правильно применять на практике?
3. Как определить и устранить дублирование кода в проекте, чтобы избежать проблем с его сопровождением и обновлением в будущем?
4. Какие инструменты и методы могут помочь автоматизировать процесс рефакторинга, снижая риск внесения ошибок и ускоряя разработку?
5. Как правильно оценить влияние рефакторинга на производительность и стабильность системы, чтобы убедиться в отсутствии негативных последствий после внесения изменений?
6. Что такое Domain-Driven Design (DDD) и какие основные задачи он призван решать?
7. Каковы ключевые особенности доменно-ориентированного дизайна и как они помогают в разработке сложных бизнес-приложений?
8. Какие основные концепции включает в себя модель DDD, такие как контекст, домен и язык?
9. Какие стратегии применяются в DDD для поддержания целостности и управляемости модели, особенно в крупных проектах?
10. Какие известные инструменты и фреймворки используются для поддержки DDD в различных проектах?
11. В каких случаях применение DDD является наиболее целесообразным, а в каких его использование может быть неоправданным?
12. Как DDD способствует взаимодействию между техническими специалистами и экспертами в предметной области для создания эффективных программных решений?

5.2.3 ПЕРЕЧЕНЬ ТЕСТОВЫХ ВОПРОСОВ

Раздел 1. Введение в объекты ООП (УК-3-31, ОПК-6-31)

- 1) Что такое класс в объектно-ориентированном программировании?
 - Множество объектов, характеризующаяся общими событиями.
 - Совокупность объектов, характеризующаяся общей структурой.
 - Множество объектов, имеющих общую структуру и общее поведение.
 - Переменные в программе, которым присвоены одинаковые свойства.
- 2) Как называется метод класса, который вызывается для создания объекта этого класса?
 - Инициализация
 - Конструктор
 - Деструктор
 - Полиморфизм
- 3) Что называется объектом в ООП?
 - То, что обладает состоянием и поведением.
 - То, что имеет четкие границы и обладает состоянием и поведением.
 - То, что имеет определенную структуру и свойства.
 - То, что обладает определенными свойствами.
- 4) Как называется механизм в ООП, который позволяет описать новый класс на основе существующего (родительского)?
 - Полиморфизм
 - Наследование
 - Инкапсуляция
 - Абстракция
- 5) Что называется методом в ООП?
 - Это процедура или функция, принадлежащая классу объектов.
 - Действие, которое может выполнять отдельный объект класса.
 - Это совокупность свойств объектов класса.
 - Свойство объектов, при котором действие с разными объектами вызывают одинаковое поведение.
- 6) Как называется свойство объектов, при котором действие с одинаковыми именами вызывает различное поведение для различных объектов?
 - Полиморфизм
 - Инкапсуляция

- Наследование
 - Абстракция
- 7) Что такое интерфейс в ООП?
- Описание конкретного объекта в памяти.
 - Абстрактный класс, который не может быть инстанцирован.
 - Совокупность свойств объектов класса.
 - Метод, который вызывается для создания объекта.
- 8) Что называется свойством в C#?
- Поле, доступ к которому контролируется.
 - Метод, который возвращает значение поля.
 - Абстрактный класс.
 - Интерфейс.
- 9) Что такое инкапсуляция в ООП?
- Скрытие внутреннего устройства объектов.
 - Создание новых классов на основе существующих.
 - Свойство объектов, при котором действие с разными объектами вызывают одинаковое поведение.
 - Абстракция данных.
- 10) Какая связь между классами называется агрегацией?
- Слабая связь, при которой один класс включает в себя другой.
 - Связь, при которой объекты одного класса принадлежат другому классу.
 - Связь, при которой объекты классов взаимодействуют на равных.
 - Сильная связь, при которой один класс зависит от другого.
- 11) Что такое композиция в контексте связей между классами?
- Слабая связь, при которой один класс включает в себя другой.
 - Связь, при которой объекты одного класса принадлежат другому классу.
 - Связь, при которой объекты классов взаимодействуют на равных.
 - Сильная связь, при которой один класс зависит от другого.
- 12) Какая связь называется ассоциацией в ООП?
- Слабая связь, при которой один класс включает в себя другой.
 - Связь, при которой объекты одного класса принадлежат другому классу.
 - Связь, при которой объекты классов взаимодействуют на равных.
 - Сильная связь, при которой один класс зависит от другого.
- 13) Что такое зависимость между классами?
- Слабая связь, при которой один класс включает в себя другой.
 - Связь, при которой объекты одного класса принадлежат другому классу.
 - Связь, при которой объекты классов взаимодействуют на равных.
 - Сильная связь, при которой один класс зависит от другого.
- 14) Какая связь называется наследованием?
- Связь, при которой один класс наследует свойства и методы другого класса.
 - Связь, при которой объекты одного класса принадлежат другому классу.
 - Связь, при которой объекты классов взаимодействуют на равных.
 - Сильная связь, при которой один класс зависит от другого.
- 15) Что такое интерфейс в контексте связей между классами?
- Абстрактный класс, который не может быть инстанцирован.
 - Класс, который может быть инстанцирован.
 - Метод, который вызывается для создания объекта.
 - Свойство, которое возвращает значение поля.
- 16) Какая связь называется реализацией в ООП?
- Связь, при которой класс реализует интерфейс.
 - Связь, при которой объекты одного класса принадлежат другому классу.
 - Связь, при которой объекты классов взаимодействуют на равных.
 - Сильная связь, при которой один класс зависит от другого.
- 17) Что такое ассоциация “один ко многим” в контексте связей между классами?
- Связь, при которой один объект связан с несколькими другими объектами.
 - Связь, при которой несколько объектов связаны с одним объектом.
 - Связь, при которой объекты классов взаимодействуют на равных.
 - Сильная связь, при которой один класс зависит от другого.
- 18) Какая связь называется ассоциацией “многие ко многим”?
- Связь, при которой один объект связан с несколькими другими объектами.
 - Связь, при которой несколько объектов связаны с одним объектом.
 - Связь, при которой объекты классов взаимодействуют на равных.
 - Связь, при которой объекты классов связаны через промежуточную сущность.
 - Что такое агрегация в контексте связей между классами?
- 19) Слабая связь, при которой один класс включает в себя другой.
- Связь, при которой объекты одного класса принадлежат другому классу.
 - Связь, при которой объекты классов взаимодействуют на равных.

- Сильная связь, при которой один класс зависит от другого.
- 20) Что такое статический метод в C#?
- Метод, который может быть вызван без создания экземпляра класса.
 - Метод, который вызывается для создания объекта.
 - Метод, который возвращает значение поля.
 - Метод, который вызывается при удалении объекта.
- 21) Как создать статический метод в C#?
- Использовать ключевое слово *static* перед определением метода.
 - Использовать ключевое слово *virtual* перед определением метода.
 - Использовать ключевое слово *abstract* перед определением метода.
 - Использовать ключевое слово *override* перед определением метода.
- 22) Что такое статическое поле в C#?
- Поле, доступ к которому контролируется.
 - Поле, которое принадлежит конкретному экземпляру класса.
 - Поле, которое может быть изменено только статическими методами.
 - Поле, которое принадлежит всем экземплярам класса.
- 23) Как создать статическое поле в C#?
- Использовать ключевое слово *static* перед определением поля.
 - Использовать ключевое слово *virtual* перед определением поля.
 - Использовать ключевое слово *abstract* перед определением поля.
 - Использовать ключевое слово *override* перед определением поля.
- 24) Что такое метод расширения в C#?
- Метод, который позволяет добавлять новые методы в существующие типы.
 - Метод, который вызывается для создания объекта.
 - Метод, который возвращает значение поля.
 - Метод, который вызывается при удалении объекта.
- 25) Как создать метод расширения в C#?
- Создать статический класс и объявить статический метод с первым параметром, имеющим префикс *this*.
 - Создать экземпляр класса и вызвать метод.
 - Использовать ключевое слово *override* для расширения метода.
 - Использовать ключевое слово *abstract* для создания метода.
- 26) Что такое статический конструктор в C#?
- Конструктор, который вызывается при создании экземпляра класса.
 - Конструктор, который вызывается при первом обращении к статическому члену класса.
 - Конструктор, который вызывается при удалении объекта.
 - Конструктор, который вызывается при изменении статического поля.
- 27) Как создать статический конструктор в C#?
- Использовать ключевое слово *static* перед определением конструктора.
 - Использовать ключевое слово *virtual* перед определением конструктора.
 - Использовать ключевое слово *abstract* перед определением конструктора.
 - Использовать ключевое слово *override* перед определением конструктора.
- 28) Что такое статический класс в C#?
- Класс, который не может иметь экземпляров.
 - Класс, который может иметь только один экземпляр.
 - Класс, который может быть унаследован.
 - Класс, который всегда является абстрактным.
- 29) Как создать статический класс в C#?
- Использовать ключевое слово *static* перед определением класса.
 - Использовать ключевое слово *virtual* перед определением класса.
 - Использовать ключевое слово *abstract* перед определением класса.
 - Использовать ключевое слово *override* перед определением класса.
- 30) Что такое частичный класс в C#?
- Класс, который может быть разделен на несколько файлов кода.
 - Класс, который используется только для создания интерфейсов.
 - Класс, который всегда является абстрактным.
 - Класс, который может иметь только статические методы.
- 31) Как разделить определение частичного класса на несколько файлов в C#?
- Использовать ключевое слово *partial* перед каждым определением класса.
 - Использовать ключевое слово *abstract* перед каждым определением класса.
 - Использовать ключевое слово *sealed* перед каждым определением класса.
 - Использовать ключевое слово *virtual* перед каждым определением класса.
- 32) Почему использование частичных классов может быть полезным в разработке?
- Для разделения логики класса на разные файлы.
 - Для создания нескольких экземпляров одного класса.
 - Для обеспечения инкапсуляции данных.
 - Для создания абстрактных классов.

- 33) Можно ли определить разные методы в разных файлах для одного частичного класса?
- Да, можно определять методы в разных файлах.
 - Можно определять только статические методы.
 - Можно определять только конструкторы.
 - Нет, методы должны быть определены в одном файле.
- 34) Как компилятор обрабатывает частичные классы в C#?
- Объединяет все части в один класс во время компиляции.
 - Создает несколько экземпляров класса для каждой части.
 - Выбрасывает ошибку, если части не совпадают.
 - Ничего не делает, оставляет части разделенными.
- 35) Можно ли наследовать от частичного класса в C#?
- Да, можно наследовать от частичного класса.
 - Можно наследовать только от абстрактных классов.
 - Можно наследовать только от статических классов.
 - Нет, нельзя наследовать от частичного класса.
- 36) Что происходит, если два файла определяют одинаковые члены для частичного класса?
- Компилятор выдает ошибку.
 - Компилятор объединяет определения.
 - Компилятор выбирает одно из определений случайным образом.
 - Компилятор игнорирует одно из определений.
- 37) Можно ли использовать частичные классы для реализации интерфейсов?
- Да, можно использовать частичные классы для реализации интерфейсов.
 - Можно использовать только для создания абстрактных классов.
 - Можно использовать только для статических методов.
 - Нет, нельзя использовать для реализации интерфейсов.
- 38) Как частичные классы помогают в генерации кода в C#?
- Они позволяют автоматически генерировать код на основе шаблонов.
 - Они обеспечивают инкапсуляцию данных.
 - Они создают несколько экземпляров класса.
 - Они используются только для создания интерфейсов.
- 39) Можно ли использовать частичные классы для разделения логики между разработчиками в одной команде?
- Да, можно использовать для разделения работы между разработчиками.
 - Можно использовать только для создания статических методов.
 - Можно использовать только для абстрактных классов.
 - Нет, нельзя использовать для разделения работы между разработчиками.

Раздел 2. Принципы программной архитектуры (УК-3-31, ОПК-6-31, ПК-1-31)

- 40) Что такое принцип единственной ответственности (*SRP*) в объектно-ориентированном программировании?
- Каждый объект должен иметь только одну обязанность и выполнять только одну функцию.
 - Каждый объект должен иметь несколько обязанностей и выполнять несколько функций.
 - Объекты должны быть максимально абстрактными.
 - Объекты должны наследовать как можно больше методов.
- 41) Какое преимущество дает соблюдение принципа *SRP*?
- Упрощает тестирование и делает код более понятным.
 - Усложняет тестирование и делает код менее понятным.
 - Увеличивает количество кода.
 - Уменьшает количество кода.
- 42) Что такое принцип открытости/закрытости (*ОСР*)?
- Код должен быть открыт для изменений и закрыт для расширений.
 - Код должен быть закрыт для изменений и открыт для расширений.
 - Код должен быть максимально абстрактным.
 - Код должен наследовать как можно больше методов.
- 43) Какое преимущество дает соблюдение принципа *ОСР*?
- Усложняет поддержку и расширение кода.
 - Делает код менее гибким и масштабируемым.
 - Позволяет легко добавлять новый функционал без изменения существующего кода.
 - Увеличивает количество ошибок в коде.
- 44) Как можно применить принцип *SRP* при разработке нового функционала?
- Добавить новый функционал в существующий объект.
 - Создать новый объект, который будет отвечать за новую функцию.
 - Использовать наследование для добавления нового функционала.
 - Использовать полиморфизм для добавления нового функционала.
- 45) Как принцип *ОСР* помогает в долгосрочной поддержке программного обеспечения?
- Усложняет добавление нового функционала.
 - Позволяет легко добавлять новый функционал без изменения существующего кода.
 - Увеличивает количество ошибок в коде.

- Уменьшает количество ошибок в коде.
- 46) Что происходит, если объект нарушает принцип *SRP*?
- Объект становится более гибким и масштабируемым.
 - Объект становится менее гибким и масштабируемым.
 - Объект становится более тестируемым.
 - Объект становится менее тестируемым.
- 47) Что означает “быть открытым для расширений” в контексте принципа *OCP*?
- Возможность изменять существующий код.
 - Возможность добавлять новый код без изменения существующего.
 - Возможность наследовать как можно больше методов.
 - Возможность делать код максимально абстрактным.
- 48) Как соблюдение принципов *SRP* и *OCP* влияет на читаемость кода?
- Ухудшает читаемость кода.
 - Улучшает читаемость кода.
 - Не влияет на читаемость кода.
 - Делает код менее тестируемым.
- 49) Почему важно соблюдать принципы *SRP* и *OCP* в сложных проектах?
- Потому что они усложняют поддержку и расширение кода.
 - Потому что они помогают избежать избыточности кода и упрощают тестирование.
 - Потому что они увеличивают количество ошибок в коде.
 - Потому что они уменьшают количество кода.
- 50) Что такое принцип подстановки Лисков (*LSP*)?
- Объекты в программе должны быть заменяемыми их подтипами без изменения правильности программы.
 - Объекты в программе должны быть заменяемы только супертипами.
 - Объекты в программе должны быть максимально абстрактными.
 - Объекты в программе должны наследовать как можно больше методов.
- 51) Какое ключевое преимущество дает принцип *LSP*?
- Усложняет тестирование кода.
 - Обеспечивает гибкость и возможность расширения кода.
 - Увеличивает количество ошибок в коде.
 - Уменьшает количество кода.
- 52) Что происходит, если принцип *LSP* нарушается в программе?
- Программа становится более гибкой и масштабируемой.
 - Программа становится менее гибкой и может привести к ошибкам при замене подтипов.
 - Программа становится более тестируемой.
 - Программа становится менее абстрактной.
- 53) Как принцип *LSP* влияет на подтипы и супертипы?
- Подтипы не могут расширять поведение супертипов.
 - Подтипы могут изменять поведение супертипов, но только в рамках их контрактов.
 - Подтипы должны полностью заменять супертипы.
 - Подтипы должны наследовать все методы супертипов.
- 54) Что означает “быть заменяемым” в контексте принципа *LSP*?
- Подтип может быть заменен супертипом без изменения поведения программы.
 - Подтип может быть заменен супертипом, только если поведение программы изменяется.
 - Подтип может быть заменен супертипом без изменения его поведения.
 - Подтип всегда должен заменять супертип.
- 55) Как принцип *LSP* помогает в обеспечении надежности программного обеспечения?
- Усложняет тестирование и делает код менее понятным.
 - Обеспечивает, что подтипы могут быть безопасно использованы вместо супертипов без изменения поведения программы.
 - Увеличивает количество ошибок в коде.
 - Уменьшает количество кода.
- 56) Что такое контракт в контексте принципа *LSP*?
- Набор методов, которые должен реализовать подтип.
 - Набор методов, которые должен реализовать супертип.
 - Набор правил и ожиданий относительно поведения объекта.
 - Набор абстрактных классов.
- 57) Как принцип *LSP* соотносится с наследованием?
- Подтипы не могут наследовать поведение супертипов.
 - Подтипы могут изменять поведение супертипов, но только в рамках их контрактов.
 - Подтипы должны полностью заменять супертипы.
 - Подтипы должны наследовать все методы супертипов.
- 58) Почему важно соблюдать принцип *LSP* при разработке библиотек и фреймворков?
- Это усложняет тестирование кода.
 - Это обеспечивает, что пользователи могут безопасно использовать подтипы вместо супертипов.
 - Это увеличивает количество ошибок в коде.

- Это уменьшает количество кода.
- 59) Как принцип *LSP* влияет на расширяемость системы?
- Усложняет добавление новых подтипов.
 - Обеспечивает, что система может быть легко расширена с использованием подтипов.
 - Увеличивает количество ошибок в коде.
 - Уменьшает количество кода.
- 60) Что такое принцип разделения интерфейсов (*ISP*)?
- Интерфейсы должны быть узкими и специализированными.
 - Интерфейсы должны содержать как можно больше методов.
 - Интерфейсы должны наследовать другие интерфейсы.
 - Интерфейсы должны быть максимально абстрактными.
- 61) Какое преимущество дает соблюдение принципа *ISP*?
- Усложняет тестирование кода.
 - Увеличивает связность между компонентами.
 - Уменьшает связность между компонентами и упрощает изменения.
 - Увеличивает количество ошибок в коде.
- 62) Что такое принцип инверсии зависимостей (*DIP*)?
- Модули высшего уровня должны зависеть от модулей низшего уровня.
 - Модули должны зависеть от конкретных реализаций.
 - Модули должны зависеть от абстракций, а не от конкретных реализаций.
 - Модули должны наследовать как можно больше методов.
- 63) Какое ключевое преимущество дает принцип *DIP*?
- Увеличивает связность между компонентами.
 - Усложняет тестирование кода.
 - Уменьшает гибкость системы.
 - Повышает гибкость системы и упрощает ее тестирование.
- 64) Как принцип *ISP* помогает в управлении зависимостями?
- Увеличивает количество ошибок в коде.
 - Уменьшает связность между компонентами.
 - Увеличивает связность между компонентами.
 - Усложняет тестирование кода.
- 65) Как принцип *DIP* влияет на структуру программного обеспечения?
- Увеличивает связность между компонентами.
 - Уменьшает гибкость системы.
 - Уменьшает связность между компонентами и повышает гибкость системы.
 - Увеличивает количество ошибок в коде.
- 66) Что происходит, если интерфейсы нарушают принцип *ISP*?
- Код становится более тестируемым.
 - Код становится менее тестируемым и сложным для изменения.
 - Код становится более гибким.
 - Код становится менее абстрактным.
- 67) Почему важно соблюдать принцип *DIP* при разработке крупных систем?
- Это увеличивает количество ошибок в коде.
 - Это уменьшает гибкость системы.
 - Это повышает гибкость системы и упрощает ее тестирование.
 - Это усложняет тестирование кода.
- 68) Как принцип *ISP* влияет на расширяемость системы?
- Увеличивает связность между компонентами.
 - Уменьшает количество ошибок в коде.
 - Уменьшает связность между компонентами и упрощает добавление новых интерфейсов.
 - Увеличивает количество ошибок в коде.
- 69) Что означает “зависеть от абстракций” в контексте принципа *DIP*?
- Модули должны зависеть от конкретных реализаций.
 - Модули должны зависеть от абстракций, таких как интерфейсы или абстрактные классы.
 - Модули должны наследовать как можно больше методов.
 - Модули должны быть максимально абстрактными.
- 70) Что такое рефакторинг кода?
- Процесс переписывания кода для улучшения его читаемости и структуры.
 - Процесс добавления новых функций в программу.
 - Процесс исправления ошибок в коде.
 - Процесс оптимизации производительности программы.
- 71) Какова основная цель рефакторинга кода?
- Усложнить код для повышения безопасности.
 - Улучшить читаемость и структуру кода.
 - Увеличить количество строк кода.
 - Исправить все ошибки в коде.

- 72) Что не является рефакторингом?
- Переписывание кода для улучшения его структуры.
 - Добавление новых функций в программу.
 - Исправление ошибок в коде.
 - Оптимизация производительности программы.
- 73) Какая методика рефакторинга включает в себя написание юнит-теста перед изменением кода?
- *Red-Green-Refactor*.
 - Абстракция.
 - Фрагментация.
 - Оптимизация.
- 74) Почему рефакторинг кода важен для разработчиков?
- Он усложняет код для повышения безопасности.
 - Он помогает улучшить читаемость и структуру кода, что упрощает его поддержку и расширение.
 - Он увеличивает количество строк кода.
 - Он исправляет все ошибки в коде.
- 75) Какие сложности могут возникнуть при рефакторинге кода?
- Увеличение производительности программы.
 - Риск внесения ошибок и нарушения функциональности программы.
 - Уменьшение количества строк кода.
 - Увеличение количества тестов.
- 76) Как часто рекомендуется проводить рефакторинг кода?
- Раз в полгода.
 - Регулярно, но небольшими порциями.
 - Только при добавлении новых функций.
 - Только при исправлении ошибок.
- 77) Что такое "*Red-Green-Refactor*" в контексте рефакторинга?
- Методика написания тестов после изменения кода.
 - Методика написания кода без тестов.
 - Методика написания юнит-теста перед изменением кода.
 - Методика переписывания кода без тестов.
- 78) Какая стратегия рефакторинга используется для удаления дубликатов кода?
- *Red-Green-Refactor*.
 - Абстракция.
 - Фрагментация.
 - Оптимизация.
- 79) Почему опытные разработчики рекомендуют сопровождать рефакторинг юнит-тестами?
- Для увеличения производительности программы.
 - Для обеспечения безопасности данных.
 - Для предотвращения внесения ошибок и нарушения функциональности программы.
 - Для уменьшения количества строк кода.
- 80) Что такое принцип *CCP (Common Closure Principle)*?
- Классы, изменяющиеся по одним и тем же причинам в одно и то же время, должны быть включены в один компонент.
 - Классы, изменяющиеся по разным причинам, должны быть включены в разные компоненты.
 - Компоненты должны быть максимально абстрактными.
 - Компоненты должны наследовать как можно больше методов.
- 81) Какое преимущество дает соблюдение принципа *CCP*?
- Усложняет повторное использование компонентов.
 - Упрощает повторное использование компонентов, так как изменения затрагивают меньшее количество компонентов.
 - Увеличивает количество ошибок в коде.
 - Уменьшает количество кода.
- 82) Что такое принцип *REP (Reuse/Release Equivalence Principle)*?
- Единица повторного использования должна быть эквивалентна единице выпуска.
 - Единица выпуска должна быть меньше единицы повторного использования.
 - Компоненты должны наследовать как можно больше методов.
 - Компоненты должны быть максимально абстрактными.
- 83) Какое преимущество дает соблюдение принципа *REP*?
- Увеличивает количество ошибок в коде.
 - Усложняет повторное использование компонентов.
 - Упрощает повторное использование компонентов, так как единица повторного использования эквивалентна единице выпуска.
 - Уменьшает количество кода.
- 84) Что такое принцип *CRP (Common Reuse Principle)*?
- Не вынуждайте пользователей компонента зависеть от того, что им не нужно.
 - Компоненты должны быть максимально абстрактными.
 - Компоненты должны наследовать как можно больше методов.

- Компоненты должны быть закрытыми для изменений.
- 85) Какое преимущество дает соблюдение принципа *CRP*?
- Увеличивает количество ошибок в коде.
 - Усложняет повторное использование компонентов.
 - Уменьшает количество ошибок в коде.
 - Усложняет тестирование кода.
- 86) Почему важно соблюдать баланс между принципами *CCP*, *REP* и *CRP*?
- Потому что они помогают избежать избыточности кода и упрощают тестирование.
 - Потому что они усложняют повторное использование компонентов.
 - Потому что они увеличивают количество ошибок в коде.
 - Потому что они уменьшают количество кода.
- 87) На каких этапах разработки важнее соблюдать принцип *CCP*?
- На ранних этапах разработки.
 - На поздних этапах разработки.
 - На всех этапах разработки в равной степени.
 - Только при исправлении ошибок.
- 88) Как соблюдение принципов *CCP* и *REP* влияет на выпуск новых версий компонентов?
- Усложняет выпуск новых версий.
 - Упрощает выпуск новых версий, так как изменения затрагивают меньшее количество компонентов и единица повторного использования эквивалентна единице выпуска.
 - Увеличивает количество ошибок в коде.
 - Уменьшает количество кода.
- 89) Почему важно соблюдать принцип *CRP* при разработке библиотек и фреймворков?
- Это увеличивает количество ошибок в коде.
 - Это усложняет тестирование кода.
 - Это уменьшает количество ошибок в коде и упрощает использование компонентов.
 - Это уменьшает количество кода.
- 90) Что такое принцип *ADP* (*Adapter Design Pattern*)?
- Обеспечивает взаимодействие между независимыми интерфейсами.
 - Позволяет объектам изменять свое поведение в зависимости от контекста.
 - Гарантирует, что объекты не будут зависеть от конкретных классов, а от интерфейсов.
 - Определяет, что классы должны иметь только одну ответственность.
- 91) Какое преимущество дает использование адаптера (*ADP*)?
- Увеличивает связность между компонентами.
 - Позволяет компонентам взаимодействовать, несмотря на несовместимые интерфейсы.
 - Увеличивает количество ошибок в коде.
 - Уменьшает количество кода.
- 92) Что такое принцип *SDP* (*Strategy Design Pattern*)?
- Обеспечивает взаимодействие между независимыми интерфейсами.
 - Позволяет объектам изменять свое поведение в зависимости от контекста.
 - Гарантирует, что объекты не будут зависеть от конкретных классов, а от интерфейсов.
 - Определяет, что классы должны иметь только одну ответственность.
- 93) Какое преимущество дает использование стратегии (*SDP*)?
- Увеличивает связность между компонентами.
 - Позволяет легко изменять поведение объектов без изменения их структуры.
 - Увеличивает количество ошибок в коде.
 - Уменьшает количество кода.
- 94) Что такое принцип *SAP* (*Service-Activator Pattern*)?
- Обеспечивает взаимодействие между независимыми интерфейсами.
 - Позволяет объектам изменять свое поведение в зависимости от контекста.
 - Гарантирует, что объекты не будут зависеть от конкретных классов, а от интерфейсов.
 - Определяет, что классы должны иметь только одну ответственность.
- 95) Какое преимущество дает использование сервиса-активатора (*SAP*)?
- Увеличивает связность между компонентами.
 - Позволяет асинхронно активировать сервисы.
 - Увеличивает количество ошибок в коде.
 - Уменьшает количество кода.
- 96) Почему важно соблюдать принцип *ADP* при разработке адаптеров?
- Это увеличивает количество ошибок в коде.
 - Это позволяет компонентам взаимодействовать, несмотря на несовместимые интерфейсы.
 - Это уменьшает количество кода.
 - Это увеличивает связность между компонентами.
- 97) Как принцип *SDP* помогает в управлении поведением системы?
- Увеличивает связность между компонентами.
 - Позволяет легко изменять поведение объектов без изменения их структуры.
 - Увеличивает количество ошибок в коде.

- Уменьшает количество кода.
- 98) В чем основное отличие принципа *ADP* от принципа *SDP*?
- *ADP* обеспечивает взаимодействие между интерфейсами, а *SDP* позволяет изменять поведение объектов.
 - *ADP* позволяет изменять поведение объектов, а *SDP* обеспечивает взаимодействие между интерфейсами.
 - Оба принципа увеличивают количество ошибок в коде.
 - Оба принципа уменьшают количество кода.
- 99) Почему принцип *SAP* важен для асинхронной обработки данных?
- Он увеличивает количество ошибок в коде.
 - Он позволяет асинхронно активировать сервисы, что улучшает производительность системы.
 - Он уменьшает количество кода.
 - Он увеличивает связность между компонентами.
- 100) Что такое паттерн *EDM* (*Event-Driven Microservices*)?
- Паттерн, который организует микросервисы вокруг бизнес-событий.
 - Паттерн, который организует микросервисы вокруг конкретных функций.
 - Паттерн, который организует микросервисы вокруг данных.
 - Паттерн, который организует микросервисы вокруг пользовательского интерфейса.
- 101) Какое преимущество дает использование *EDM*?
- Увеличивает связность между компонентами.
 - Позволяет микросервисам реагировать на события асинхронно.
 - Увеличивает количество ошибок в коде.
 - Уменьшает количество кода.
- 102) Что такое паттерн *DDD* (*Domain-Driven Design*)?
- Подход, фокусирующийся на событиях и их обработке.
 - Подход, фокусирующийся на данных и их хранении.
 - Подход, фокусирующийся на бизнес-логике и моделировании домена.
 - Подход, фокусирующийся на пользовательском интерфейсе.
- 103) Какое ключевое преимущество дает *DDD*?
- Увеличивает связность между компонентами.
 - Позволяет лучше понять и моделировать бизнес-логику.
 - Увеличивает количество ошибок в коде.
 - Уменьшает количество кода.
- 104) Что такое паттерн *TDD* (*Test-Driven Development*)?
- Подход, при котором тесты пишутся после написания кода.
 - Подход, при котором тесты пишутся до написания кода.
 - Подход, при котором тесты пишутся одновременно с кодом.
 - Подход, при котором тесты не пишутся вообще.
- 105) Какое преимущество дает использование *TDD*?
- Увеличивает связность между компонентами.
 - Позволяет писать более надежный и тестируемый код.
 - Увеличивает количество ошибок в коде.
 - Уменьшает количество кода.
- 106) Почему важно использовать *EDM* в микросервисной архитектуре?
- Это увеличивает количество ошибок в коде.
 - Это позволяет микросервисам реагировать на события асинхронно, улучшая масштабируемость.
 - Это уменьшает количество кода.
 - Это увеличивает связность между компонентами.
- 107) Как *DDD* помогает в понимании и моделировании бизнес-логики?
- Увеличивает связность между компонентами.
 - Позволяет лучше понять и моделировать бизнес-логику, делая код более понятным.
 - Увеличивает количество ошибок в коде.
 - Уменьшает количество кода.
- 108) В чем основное отличие *TDD* от других подходов к разработке?
- *TDD* увеличивает количество ошибок в коде.
 - *TDD* позволяет писать тесты до написания кода, что улучшает качество и надежность.
 - *TDD* уменьшает количество кода.
 - *TDD* увеличивает связность между компонентами.
- 109) Почему *DDD* важен для сложных бизнес-приложений?
- Он увеличивает количество ошибок в коде.
 - Он позволяет лучше моделировать и понимать бизнес-логику, что упрощает поддержку и расширение системы.
 - Он уменьшает количество кода.
 - Он увеличивает связность между компонентами.

5.3. Оценочные материалы, используемые для экзамена (описание билетов, тестов и т.п.)

Экзаменационный билет включает в себя часть с 3-мя теоретическими вопросами из установленного перечня вопросов для подготовки к экзамену п.5.1.1, и 1-й прикладной задачи из установленного перечня контрольных вопросов п.5.1.2. Пример экзаменационного билета:

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
СТАРООСКОЛЬСКИЙ ТЕХНОЛОГИЧЕСКИЙ ИНСТИТУТ им. А.А. Угарова
(филиал)
федерального государственного автономного образовательного учреждения
высшего образования
«Национальный исследовательский технологический университет «МИСИС»
Кафедра «Автоматизированные и информационные системы управления им Ю.И. Еременко»
09.03.02 Информационные системы и технологии
Профиль – Информационные системы и технологии

Дисциплина «Архитектура ЭВМ»

ЭКЗАМЕНАЦИОННЫЙ БИЛЕТ № 1

Теоретические вопросы:

1. Что такое класс и объект в контексте объектно-ориентированного программирования?
2. Что такое инкапсуляция и как она реализуется в ООП?
3. Почему рекомендуется использовать свойства вместо публичных полей в структурах?

Практическая задача:

1. Создайте интерфейс с одним методом, который принимает целочисленный аргумент и возвращает целое число. Реализуйте этот интерфейс в двух различных классах, каждый из которых должен предоставить свою реализацию метода.

Экзаменатор _____ П.И. Жуков

Утверждено на заседании кафедры АИСУ им. Ю.И. Еременко

Протоколом № ____ от « ____ » _____ 20__ г.

И.о. заведующего кафедрой АИСУ им. Ю.И. Еременко _____ Д.А. Поляшенко

Билеты в бумажном виде, в количестве 30 вариантов, хранятся на кафедре АИСУ им. Ю.И. Еременко и утверждены ее заведующим (или заместителем зав. кафедрой).

5.4. Методика оценки освоения дисциплины

№ п/п	Вид оценочного средства	Критерий	Оценка
1	Выполнение и защита домашних заданий	Обучающийся самостоятельно выполняет полное и аргументированное решение индивидуальных заданий, не допустив ошибок. При защите заданий отвечает развернуто и исчерпывающе на все вопросы.	«Отлично»
		Обучающийся практически самостоятельно выполняет полное решение заданий, но не может аргументировать свое решение. При защите заданий допускает незначительные неточности.	«Хорошо»
		Обучающийся в целом правильно решает задание, но не может аргументировать свое решение. При защите заданий допускает значительные неточности. Обучающийся правильно понимает способ решения заданий, но допускает ошибки при их решении. Задание выполнено частично. При защите заданий допускает значительные неточности.	«Удовлетворительно»
		Обучающийся не может решить задание.	«Неудовлетворительно»
2	Устный опрос по материалам лекционных занятий	Обучающийся уверенно, логически связно, динамично, грамотно отвечает на поставленные вопросы. Демонстрирует абсолютное понимание темы обсуждаемой предметной области, достаточно полные знания в объеме прослушанного теоретического материала. Грамотно владеет и адекватно использует терминологию предметной области.	«Зачтено»
		Обучающийся не верно интерпретирует поставленные вопросы, не владеет терминологией предметной области, не понимает сущности обсуждаемой проблемы. Невнятно, невразумительно, неуверенно, неверно формулирует ответы на дополнительные и наводящие вопросы и/или не отвечает на них.	«Не зачтено»

3	Выполнение и защита лабораторных работ	Обучающийся показывает твердые и достаточно полные знания в объеме прослушанного теоретического материала, владеет требуемым математическим аппаратом, методиками решения, необходимыми остаточными знаниями по изученным фундаментальным дисциплинам (математика, физика); демонстрирует умения и практические навыки владения информационными технологиями, позволяющими оптимизировать экспериментальную и аналитическую часть лабораторного исследования. Логически связно, динамично, грамотно и последовательно излагает методику выполнения лабораторной работы и обработки результатов моделирования. Ошибаясь, уверенно исправляется после дополнительных и наводящих вопросов.	«Зачтено»
		Обучающийся допускает грубые ошибки в ответе, не понимает сущности излагаемого вопроса, не умеет применять теоретические знания на практике и/или не владеет требуемыми знаниями. Невнятно, невразумительно, неуверенно, неверно формулирует ответы на дополнительные и наводящие вопросы и/или не отвечает на них.	«Не зачтено»
4	Экзамен	Компетенции УК-3, ОПК-2, ОПК-6 ПК-1 сформированы. Обучающийся демонстрирует: - глубокие знания содержания изученной дисциплины во взаимосвязи с другими дисциплинами; - способность использовать теоретические знания при выполнении практических заданий; - аргументированные, исчерпывающие ответы на все вопросы по билету, а также дополнительные вопросы экзаменатора; - умение выполнять и обосновывать решение практических заданий высокого уровня сложности; - наличие собственной обоснованной позиции по обсуждаемым вопросам; - свободное использование в ответах на вопросы материалов рекомендованной основной и дополнительной литературы.	«Отлично»
		Компетенции УК-3, ОПК-2, ОПК-6 ПК-1 сформированы. Обучающийся демонстрирует: - знание основных терминов по содержанию изученной дисциплины; - твердые знания теоретического материала; - умение дать четкие ответы на поставленные вопросы; - умение решать практические задания; - владение основной литературой, рекомендованной программой дисциплины. Допускаются незначительные неточности в ответах на теоретические вопросы и при выполнении практических заданий.	«Хорошо»
		Компетенции УК-3, ОПК-2, ОПК-6 ПК-1 сформированы. Обучающийся демонстрирует: - знания теоретического материала по изученной дисциплине; - неполные ответы на основные вопросы, допуская ошибки в ответе; недостаточное понимание сущности излагаемых вопросов; - неточные ответы на дополнительные вопросы; - умение выполнять практические задания без грубых ошибок; - недостаточное владение литературой, рекомендованной программой дисциплины.	«Удовлетворительно»
		Компетенции УК-3, ОПК-2, ОПК-6 ПК-1 не сформированы. Обучающийся демонстрирует: - существенные пробелы в знаниях учебного материала; - принципиальные ошибки при ответе на основные вопросы билета, отсутствие знаний и понимания основных терминов и определений; - непонимание сущности дополнительных вопросов в рамках заданий билета; - отсутствие навыка или существенные ошибки при выполнении практических заданий; - незнание литературы, рекомендованной программой дисциплины.	«Неудовлетворительно»

5	Оценка тестовых заданий	Обучающийся дал 90% и более правильных ответов	«Отлично»
		Обучающийся дал 65-89% правильных ответов	«Хорошо»
		Обучающийся дал 50-64% правильных ответов	«Удовлетворительно»
		Обучающийся дал менее 50% правильных ответов	«Неудовлетворительно»
5.5 Тестовые задания по дисциплине для самопроверки			
См. п. 5.2.3			

6. УЧЕБНО-МЕТОДИЧЕСКОЕ И ИНФОРМАЦИОННОЕ ОБЕСПЕЧЕНИЕ				
6.1. Рекомендуемая литература				
6.1.1. Основная литература				
Обозначение	Авторы, составители	Заглавие	Библиотека	Издательство, год
Л 1.1	Чернышев, С.А.	Принципы, паттерны и методологии разработки программного обеспечения	ЭБС «Юрайт. Образовательная платформа» URL: https://urait.ru/book/principy-patterny-i-metodologii-razrabotki-programmnogo-obespecheniya-544319	М: Юрайт, 2024
Л 1.2	Казанский А.А.	Программирование на C#	ЭБС «Юрайт. Образовательная платформа» URL: https://urait.ru/bcode/569864	М: Юрайт, 2025
6.1.2. Дополнительная литература				
Обозначение	Авторы, составители	Заглавие	Библиотека	Издательство, год
Л 2.1	Р. Мартин	Чистая архитектура. Искусство разработки программного обеспечения	Доступ по ссылке: https://disk.yandex.ru/d/gCgNwPtUqVc1oQ	СПб: Питер, 2021
6.1.3. Методические разработки				
Обозначение	Авторы, составители	Заглавие	Библиотека	Издательство, год
Л 3.1	П.И. Жуков	«Программная архитектура». Учебное пособие.	https://disk.yandex.ru/d/gCgNwPtUqVc1oQ	(в разработке)
6.2. Перечень ресурсов информационно-телекоммуникационной сети «Интернет»				
Э 1	Открытое образование [Электронный ресурс]: https://openedu.ru/			
Э 2	LMS Moodle [Электронный ресурс]: https://newlms.misis.ru			
Э 3	ЭБС «Университетская библиотека онлайн» [Электронный ресурс]: https://biblioclub.ru			
Э 4	Интерактивный учебник «Metanit. Язык C#» [Электронный ресурс]: https://metanit.com/sharp/			
6.3. Перечень программного обеспечения				
П 1	Microsoft Windows			
П 2	Microsoft Office			
П 3	JetBrain's Rider			
6.4. Перечень информационных справочных систем и профессиональных баз данных				
И 1	eLIBRARY.RU - НАУЧНАЯ ЭЛЕКТРОННАЯ БИБЛИОТЕКА [Электронный ресурс]: https://elibrary.ru/			

7. МАТЕРИАЛЬНО-ТЕХНИЧЕСКОЕ ОБЕСПЕЧЕНИЕ	
7.1	Аудитория №404 «Объектно-ориентированного программирования» Перечень основного оборудования, учебно-наглядных пособий: • 25 посадочных мест • 13 мониторов • 13 персональных компьютеров • Проектор Epson EH-TW750 • Экран настенный • Маркерная доска В помещении имеется подключение к сети «Интернет» и доступ в электронную информационно-образовательную среду организации.
7.2	Аудитория №405 «Прикладного программирования» Перечень основного оборудования, учебно-наглядных пособий:

	• 31 посадочное место • 13 мониторов • 13 персональных компьютеров • Проектор Epson EB-485W • Экран настенный • Доска настенная В помещении имеется подключение к сети «Интернет» и доступ в электронную информационно-образовательную среду организации.
--	---

8. МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ОБУЧАЮЩИХСЯ ПО ИЗУЧЕНИЮ ДИСЦИПЛИНЫ

В рамках курса предполагается использование языка программирования C# как одного из современных и популярных ЯП высокого уровня, обладающих целевой направленностью на предмет изучения в рамках дисциплины. Качественное освоение дисциплины возможно только при углубленном самостоятельном изучении тем, затрагиваемых в ней. Настоятельно рекомендуется ознакомиться с главами №1 и №2 интерактивного учебника по C# на ресурсе Metanit (<https://metanit.com/sharp/>).

Материал в рамках данной дисциплины предполагает, что обучающийся уверенно владеет навыками структурного программирования и готов перейти к освоению объектно-ориентированной парадигмы.

Критерии успешного завершения дисциплины:

- Сдача экзамена на оценку «удовлетворительно» и выше;

Критерии допуска к экзамену:

- Успешная защита всех лабораторных работ;
- Успешная защита всех практических работ;
- Успешная защита всех домашних самостоятельных работ.

Критерий успешной защиты лабораторных, практических и домашних самостоятельных работы:

- Полное выполнение задания в рамках работы;
- Устный ответ на 70% вопросов, представленных к конкретному виду работы.

Требования к оформлению отчетов по лабораторным, практическим и домашним заданиям:

- Наличие титульного листа с фамилией обучающегося и преподавателя дисциплины;
- Наличие листа с постановкой задачи;
- Наличие листа с результатами выполнения задания;
- Times New Roman, как основной шрифт, пт.14 или пт.12.

Программный код, представляемый к защите, должен обладать версткой, в соответствии с код-конвенцией C# (<https://learn.microsoft.com/ru-ru/dotnet/csharp/fundamentals/coding-style/coding-conventions>), элементы кода должны иметь цветовую дифференциацию (в том числе, в градациях серого). Код должен быть читаем. Допускается предоставление кода в отдельном репозитории GitHub или аналоге.