КАЗАНСКИЙ КООПЕРАТИВНЫЙ ИНСТИТУТ (ФИЛИАЛ)
АВТОНОМНОЙ НЕКОММЕРЧЕСКОЙ ОБРАЗОВАТЕЛЬНОЙ
ОРГАНИЗАЦИИ ВЫСШЕГО ОБРАЗОВАНИЯ
ЦЕНТРОСОЮЗА РОССИЙСКОЙ ФЕДЕРАЦИИ
«РОССИЙСКИЙ УНИВЕРСИТЕТ КООПЕРАЦИИ»

ОЦЕНОЧНЫЕ МАТЕРИАЛЫ

ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

Специальность: 10.05.01 Компьютерная безопасность
Специализация: «Разработка защищенного программного обеспечения»
Формы обучения: очная
Объем дисциплины:
в зачетных единицах: 2 з.е.
в академических часах: 72 ак.ч.

Форма промежуточной аттестации: зачет

2025

Оценочные материалы по дисциплине Паттерны проектирования

обсуждены и рекомендованы к утверждению решением кафедры гуманитарных дисциплин и иностранных языков от 21 марта 2025 г., протокол № 7

одобрены Научно-методическим советом Казанского кооперативного института (филиала) от «2» апреля 2025 г., протокол № 3.

СОДЕРЖАНИЕ

Целью оценочных материалов по дисциплине (модулю) (далее –ОМ) является комплексная оценка результатов обучения по дисциплине Паттерны проектирования и установление уровня сформированности компетенций обучающегося.

ОМ предназначены для проведения текущего контроля успеваемости и промежуточной аттестации.

ОМ включают оценочные средства для проведения текущего контроля успеваемости и промежуточной аттестации.

Оценочные материалы разработаны в соответствии с рабочей программой дисциплины Паттерны проектирования.

1. Паспорт оценочных материалов по дисциплине «Паттерны проектирования»

Формируемые компетенции (код и наименование компетенции)	Индикаторы достижения компетенций	Планируемые результаты обучения	Контролируемые разделы, темы дисциплины	Наименование оценочного средства[1]	
				Текущий контроль	Промежуточная аттестация
1	2	3	4	5	6
ПК-3.4 Способен анализировать уязвимости и угрозы информационной безопасности в компьютерных системах и сетях	ПК-3.4.1 Анализирует компьютерную систему с целью определения уровня защищённости и доверия	Знать: основные концепции и стандарты информационной безопасности, типы угроз и уязвимостей компьютерных систем, а также методы их выявления и классификации, методологии и инструменты для оценки уровня защищённости и доверия компьютерных систем. Уметь: идентифицировать компоненты компьютерной системы, подлежащие анализу, проводить анализ рисков для компьютерной системы, выявлять уязвимости и определять потенциальные угрозы, использовать специализированные инструменты и программное обеспечение для сканирования, тестирования и анализа безопасности компьютерных систем. Владеть: навыками применения методов и инструментов анализа защищённости компьютерных систем, навыками выявления, оценки и классификации угроз и уязвимостей компьютерных систем, навыками разработки и реализации рекомендаций по повышению уровня	Тема 1. Объектно-ориентированный подход к программированию Тема 2. Порождающие паттерны. Тема 3. Структурные паттерны Тема 4. Паттерны поведения	Реферат (Тема 1-4) Тест (тема 1-4) Домашние задания по практическим занятиям (тема 1-4)	Тест (П 1-10 тестового задания)

		защищённости и доверия компьютерных систем.			
	ПК-3.4.2 Демонстрирует умение проводить анализ безопасности компоненты программныхи программно-аппаратных средств защиты информации в компьютерных системах	Знать: основные подходы и методики анализа безопасности компонентов программных и программно-аппаратных средств защиты информации, типовые виды атак и сценарии компрометации компонент защитных механизмов, методы верификации и аттестации средств защиты информации, современные средства и технологии мониторинга и аудита безопасности компонент защиты информации. Уметь: проводить детальное исследование функциональности и архитектуры компонент программных и аппаратных средств защиты информации, выявлять возможные уязвимости и слабые места в компонентах защитной инфраструктуры, оценивать эффективность используемых средств защиты и степень соответствия установленным требованиям и стандартам. Владеть: навыками практической работы с современными инструментами и методами анализа безопасности программных и аппаратных решений, способностью разрабатывать рекомендации по устранению выявленных недостатков и оптимизации настроек существующих средств защиты, методиками документального оформления результатов анализа и оценивания степени риска для	Тема 1. Объектно-ориентированный подход к программированию Тема 2. Порождающие паттерны. Тема 3. Структурные паттерны Тема 4. Паттерны поведения	Реферат (Тема 1-4) Тест (тема 1-4) Домашние задания по практическим занятиям (тема 1-4)	Тест (П 11-20 тестового задания)

		автоматизированных систем обработки информации.			
	ПК-3.4.3. Формирует предложения по устранению выявленных уязвимостей	Знать: основные подходы и принципы устранения уязвимостей программного и аппаратного обеспечения, рекомендации по выбору оптимальных способов нейтрализации выявленных слабых мест, законодательные требования и регламенты, регламентирующие процесс исправления дефектов и повышение уровня защищенности информационных ресурсов. Уметь: анализировать выявленные уязвимости и оценивать их критичность для функционирования системы, определять наиболее эффективные пути ликвидации выявленных проблем и предотвращения возможных инцидентов, составлять аргументированные предложения по модернизации существующей системы защиты. Владеть: навыками разработки конкретных мероприятий по устранению выявленных уязвимостей, умениями формулировать конкретные рекомендации и технические задания на устранение найденных дефектов, наличием опыта управления проектами по улучшению уровня защиты ИТ-инфраструктуры организаций.	Тема 1. Объектно-ориентированный подход к программированию Тема 2. Порождающие паттерны. Тема 3. Структурные паттерны Тема 4. Паттерны поведения	Реферат (Тема 1-4) Тест (тема 1-4) Домашние задания по практическим занятиям (тема 1-4)	Тест (П 21-30 тестового задания)

2. Оценочные материалы по дисциплине «Паттерны проектирования»

2.1 Оценочные материалы для проведения текущего контроля успеваемости

Тематика рефератов

Тема 1. Объектно-ориентированный подход к программированию

1. **Основные принципы ООП: сравнительный анализ реализации в Ruby и Kotlin**
   - Сравнение синтаксиса и семантики
   - Особенности наследования в разных языках
   - Практические примеры инкапсуляции и полиморфизма
2. **Динамические возможности Ruby: метапрограммирование и его применение**
   - Открытые классы и monkey patching
   - Методы send, define_method, method_missing
   - Практические примеры использования
3. **Система типов в Kotlin: null safety и расширения функций**
   - Nullable и non-null типы
   - Extension functions и properties
   - Сравнение с динамической типизацией Ruby
4. **Работа со строками и массивами: особенности в Ruby и Kotlin**
   - Строковые интерполяции и методы
   - Функциональные операции с коллекциями
   - Производительность и лучшие практики
5. **Создание Web-сервисов на Ruby: Sinatra и Rails в сравнении**
   - Простые сервисы на Sinatra
   - Полнофункциональные приложения на Rails
   - REST API и микросервисная архитектура
6. **Функциональное программирование в ООП-контексте: Kotlin vs Ruby**
   - Lambda-выражения и higher-order functions
   - Immutable data structures
   - Композиция функций и потоковая обработка
7. **Модули и mixins в Ruby: альтернатива множественному наследованию**
   - Принцип включения модулей
   - Пространства имен и организация кода
   - Сравнение с интерфейсами в Kotlin
8. **Корутины и асинхронное программирование в Kotlin**
   - Принципы работы корутин
   - Отмена и обработка ошибок
   - Практическое применение в веб-приложениях

9. **Тестирование в Ruby и Kotlin: RSpec vs JUnit**
    - Методологии TDD и BDD
    - Моки и стабы
    - Интеграционное тестирование
10. **Создание DSL (Domain Specific Language) на Ruby**
    - Блоки и замыкания
    - Построение внутренних DSL
    - Практические примеры
11. **Рефлексия и интроспекция в обоих языках**
    - Получение информации о классах и методах
    - Динамическое создание и вызов методов
    - Применение в фреймворках

Тема 2. Порождающие паттерны

1. **Паттерн Abstract Factory: создание семейств связанных объектов**
    - Структура и участники
    - Пример реализации на Ruby и Kotlin
    - Применение в UI-библиотеках
2. **Паттерн Builder: пошаговое конструирование сложных объектов**
    - Fluent interface и цепочки вызовов
    - Реализация в Kotlin с apply/also
    - Пример построения SQL-запросов
3. **Паттерн Factory Method: виртуальные конструкторы**
    - Иерархии классов и фабрики
    - Параметризованные фабричные методы
    - Применение в фреймворках
4. **Паттерн Prototype: клонирование объектов**
    - Глубокое и поверхностное копирование
    - Регистр прототипов
    - Примеры в графических редакторах
5. **Паттерн Singleton: глобальный доступ к экземпляру**
    - Потокобезопасные реализации
    - Антипаттерны и критика
    - Альтернативы (Dependency Injection)
6. **Сравнительный анализ порождающих паттернов**
    - Критерии выбора паттерна
    - Производительность и поддерживаемость
    - Пример комплексного применения
7. **Паттерн Object Pool: управление дорогостоящими объектами**
    - Принципы работы пула
    - Примеры: подключения к БД, потоки
    - Ограничения и лучшие практики
8. **Паттерн Lazy Initialization: отложенная инициализация**
    - Ленивые вычисления в функциональном стиле
    - Thread-safe реализации

- Применение в Kotlin (lazy delegate)

9. **Dependency Injection как современная альтернатива порождающим паттернам**
   - Принципы IoC (Inversion of Control)
   - Фреймворки для DI
   - Сравнение с традиционными подходами
10. **Паттерн Multiton: управление множеством одиночек**
    - Регистр именованных экземпляров
    - Примеры использования
    - Отличия от Singleton
11. **Реализация порождающих паттернов в веб-фреймворках**
    - Фабрики контроллеров и моделей
    - Построители форм и валидаторов
    - Практические кейсы

Тема 3. Структурные паттерны

1. **Паттерн Adapter: согласование интерфейсов**
   - Object adapter vs Class adapter
   - Примеры адаптации сторонних библиотек
   - Применение в миграции legacy-кода
2. **Паттерн Bridge: разделение абстракции и реализации**
   - Мост между высокоуровневой и низкоуровневой логикой
   - Пример: кроссплатформенные UI-фреймворки
   - Расширяемость и тестируемость
3. **Паттерн Composite: древовидные структуры объектов**
   - Рекурсивные структуры данных
   - Пример: файловая система, UI-компоненты
   - Операции обхода дерева
4. **Паттерн Decorator: динамическое добавление обязанностей**
   - Цепочка декораторов
   - Пример: системы кэширования, логирования
   - Отличия от наследования
5. **Паттерн Facade: унифицированный интерфейс к подсистеме**
   - Упрощение сложных API
   - Пример: клиенты для внешних сервисов
   - Рефакторинг legacy-систем
6. **Паттерн Flyweight: эффективная поддержка множества объектов**
   - Разделение состояния
   - Пример: текстовые редакторы, игровые движки
   - Управление внутренним и внешним состоянием
7. **Паттерн Proxy: суррогат другого объекта**
   - Virtual proxy, protection proxy, remote proxy
   - Пример: ленивая загрузка, доступ к удаленным объектам
   - Динамические прокси в Kotlin
8. **Сравнение структурных паттернов для решения типовых задач**

- Критерии выбора паттерна
- Комбинации паттернов
- Антипаттерны и типичные ошибки

9. **Структурные паттерны в веб-разработке**
   - Adapter для интеграции с внешними API
   - Decorator для middleware в веб-фреймворках
   - Composite для построения сложных UI
10. **Реализация структурных паттернов в функциональном стиле**
    - Композиция функций как альтернатива Decorator
    - Type classes как аналог Adapter
    - Практические примеры на Kotlin
11. **Паттерн Module: организация кода в модули**
    - Инкапсуляция реализации
    - Публичный интерфейс модуля
    - Примеры в современных фреймворках

Тема 4. Паттерны поведения

1. **Паттерн Chain of Responsibility: цепочка обработчиков**
   - Динамическое построение цепочек
   - Пример: middleware в веб-фреймворках
   - Обработка событий и запросов
2. **Паттерн Command: инкапсуляция запроса как объекта**
   - Отмена и повтор операций
   - Пример: системы управления задачами
   - Очереди команд и планировщики
3. **Паттерн Interpreter: интерпретатор языков**
   - Дерево синтаксического разбора
   - Пример: DSL, правила валидации
   - Производительность и ограничения
4. **Паттерн Iterator: последовательный доступ к элементам агрегата**
   - Внутренние и внешние итераторы
   - Ленивые вычисления и потоки данных
   - Пример: обработка больших коллекций
5. **Паттерн Mediator: централизованное взаимодействие объектов**
   - Снижение связанности компонентов
   - Пример: системы событий, чат-комнаты
   - Реализация в UI-фреймворках
6. **Паттерн Memento: сохранение и восстановление состояния**
   - Снимки состояния объекта
   - Пример: системы отмены действий, контроль версий
   - Управление памятью и производительность
7. **Паттерн Observer: механизм подписки и уведомлений**
   - Push и pull модели
   - Пример: реактивное программирование, системы событий
   - Потокобезопасные реализации

8. **Паттерн State: изменение поведения при изменении состояния**
    - Конечные автоматы
    - Пример: workflow-системы, игровые AI
    - Переходы между состояниями
9. **Паттерн Strategy: семейство алгоритмов**
    - Взаимозаменяемые алгоритмы
    - Пример: системы сортировки, валидации
    - Функциональные интерфейсы в Kotlin
10. **Паттерн Template Method: шаблонный алгоритм**
    - Переиспользование структуры алгоритма
    - Хуки и абстрактные операции
    - Пример: фреймворки для тестирования
11. **Паттерн Visitor: операции над структурами объектов**
    - Двойная диспетчеризация
    - Пример: компиляторы, сериализаторы
    - Расширяемость без изменения классов
12. **Сравнительный анализ поведенческих паттернов**
    - Критерии выбора для типовых задач
    - Комбинации паттернов в реальных проектах
    - Производительность и поддерживаемость
13. **Реактивное программирование как современная альтернатива поведенческим паттернам**
    - Потоки данных и наблюдаемые последовательности
    - Операторы преобразования и комбинации
    - Примеры на RxJava/RxRuby

Критерии оценки выполнения рефератов по учебной дисциплине

Оценка «отлично» выставляется студенту, если тема реферата раскрыта в полной мере и все требования по написанию реферата соблюдены.

Оценка «хорошо» выставляется студенту, если недостаточно раскрыта тема реферата, но все требования по написанию реферата соблюдены.

Оценка «удовлетворительно» выставляется студенту, если недостаточно раскрыта тема реферата, не все требования по написанию реферата соблюдены, присутствуют ошибки и неточности в работе.

Оценка «неудовлетворительно» выставляется студенту, если отсутствует реферат.

Рекомендации по написанию реферата:

Реферат — письменная работа, выполняемая обучающимся в течение определенного срока.

Реферат — краткое точное изложение сущности какого-либо вопроса, темы на основе одной или нескольких книг, монографий или других первоисточников. Реферат должен содержать основные фактические сведения

и выводы по рассматриваемому вопросу.

Реферат отвечает на вопрос — что содержится в данной публикации (публикациях). Реферат — не механический пересказ работы, а изложение ее существа.

Кроме реферирования прочитанной литературы, от обучающегося требуется аргументированное изложение собственных мыслей по рассматриваемому вопросу. Тему реферата предлагает преподаватель или сам обучающийся, в последнем случае она должна быть согласованна с преподавателем.

В реферате нужны развернутые аргументы, рассуждения, сравнения. Материал подается не столько в развитии, сколько в форме констатации или описания.

Содержание реферируемого произведения излагается объективно от имени автора. Если в первичном документе главная мысль сформулирована недостаточно четко, в реферате она должна быть конкретизирована и выделена.

Структура реферата:

1. Титульный лист.

2. После титульного листа на отдельной странице следует оглавление (план, содержание), в котором указаны названия всех разделов (пунктов плана) реферата и номера страниц, указывающие начало этих разделов в тексте реферата.

3. После оглавления следует введение. Объем введения составляет 1,5-2 страницы.

4. Основная часть реферата может иметь одну или несколько глав, состоящих из 2-3 параграфов (подпунктов, разделов) и предполагает осмысленное и логичное изложение главных положений и идей, содержащихся в изученной литературе. В тексте обязательны ссылки на первоисточники. В том случае если цитируется или используется чья-либо неординарная мысль, идея, вывод, приводится какой-либо цифрой материал, таблицу - обязательно сделайте ссылку на того автора у кого вы взяли данный материал.

5. Заключение содержит главные выводы, и итоги из текста основной части, в нем отмечается, как выполнены задачи и достигнуты ли цели, сформулированные во введении.

6. Приложение может включать графики, таблицы, расчеты.

7. Библиография (список литературы) здесь указывается реально использованная для написания реферата литература. Список составляется согласно правилам библиографического описания.

Этапы работы над рефератом.

Работу над рефератом можно условно подразделить на три этапа:

1. Подготовительный этап, включающий изучение предмета исследования;

2. Изложение результатов изучения в виде связного текста;

3. Устное сообщение по теме реферата.

1. Подготовительный этап работы.

Формулировка темы. Подготовительная работа над рефератом

начинается с формулировки темы. Тема в концентрированном виде выражает содержание будущего текста, фиксируя как предмет исследования, так и его ожидаемый результат. Для того чтобы работа над рефератом была успешной, необходимо, чтобы тема заключала в себе проблему, скрытый вопрос (даже если наука уже давно дала ответ на этот вопрос, студент, только знакомящийся с соответствующей областью знаний, будет вынужден искать ответ заново, что даст толчок к развитию проблемного, исследовательского мышления).

Поиск источников. Грамотно сформулированная тема зафиксировала предмет изучения; задача обучающегося — найти информацию, относящуюся к данному предмету и разрешить поставленную проблему. Выполнение этой задачи начинается с поиска источников. На этом этапе необходимо вспомнить, как работать с энциклопедиями и энциклопедическими словарями.

Работа с источниками. Работу с источниками надо начинать с ознакомительного чтения, т. е. просмотреть текст, выделяя его структурные единицы. При ознакомительном чтении закладками отмечаются те страницы, которые требуют более внимательного изучения. В зависимости от результатов ознакомительного чтения выбирается дальнейший способ работы с источником. Если для разрешения поставленной задачи требуется изучение некоторых фрагментов текста, то используется метод выборочного чтения. Если в книге нет подробного оглавления, следует обратить внимание на предметные и именные указатели.

Избранные фрагменты или весь текст (если он целиком имеет отношение к теме) требуют вдумчивого, неторопливого чтения с «мысленной проработкой» материала. Такое чтение предполагает выделение: 1) главного в тексте; 2) основных аргументов; 3) выводов. Особое внимание следует обратить на то, вытекает тезис из аргументов или нет. Необходимо также проанализировать, какие из утверждений автора носят проблематичный, гипотетический характер и уловить скрытые вопросы.

Понятно, что умение таким образом работать с текстом приходит далеко не сразу. Наилучший способ научиться выделять главное в тексте, улавливать проблематичный характер утверждений, давать оценку авторской позиции — это сравнительное чтение, в ходе которого студент знакомится с различными мнениями по одному и тому же вопросу, сравнивает весомость и доказательность аргументов сторон и делает вывод о наибольшей убедительности той или иной позиции.

Создание конспектов для написания реферата.

Подготовительный этап работы завершается созданием конспектов, фиксирующих основные тезисы и аргументы. Здесь важно вспомнить, что конспекты пишутся на одной стороне листа, с полями и достаточным для исправления и ремарок межстрочным расстоянием (эти правила соблюдаются для удобства редактирования). Если в конспектах приводятся цитаты, то непременно должно быть дано указание на источник (автор, название, выходные данные, № страницы).

По завершении предварительного этапа можно переходить непосредственно к созданию текста реферата.

2. Создание текста.

Общие требования к тексту.

Текст реферата должен подчиняться определенным требованиям: он должен раскрывать тему, обладать связностью и цельностью.

Раскрытие темы предполагает, что в тексте реферата излагается относящийся к теме материал и предлагаются пути решения содержащейся в теме проблемы; связность текста предполагает смысловую соотносительность отдельных компонентов, а цельность - смысловую законченность текста.

С точки зрения связности все тексты делятся на тексты-констатации и тексты-рассуждения. Тексты-констатации содержат результаты ознакомления с предметом и фиксируют устойчивые и несомненные суждения. В текстах-рассуждениях одни мысли извлекаются из других, некоторые ставятся под сомнение, дается им оценка, выдвигаются различные предположения.

План реферата.

Универсальный план реферата – введение, основной текст и заключение.

Требования к введению.

Во введении аргументируется актуальность исследования, - т. е. выявляется практическое и теоретическое значение данного исследования. Далее констатируется, что сделано в данной области предшественниками; перечисляются положения, которые должны быть обоснованы. Введение может также содержать обзор источников или экспериментальных данных, уточнение исходных понятий и терминов, сведения о методах исследования. Во введении обязательно формулируются цель и задачи реферата.

Объем введения - в среднем около 10% от общего объема реферата.

Основная часть реферата.

Основная часть реферата раскрывает содержание темы. Она наиболее значительна по объему, наиболее значима и ответственна. В ней обосновываются основные тезисы реферата, приводятся развернутые аргументы, предполагаются гипотезы, касающиеся существа обсуждаемого вопроса. Важно проследить, чтобы основная часть не имела форму монолога. Аргументируя собственную позицию, можно и должно анализировать, и оценивать позиции различных исследователей, с чем-то соглашаться, чему-то возражать, кого-то опровергать. Текст основной части делится на главы, параграфы, пункты. План основной части может быть составлен с использованием различных методов группировки материала: классификации (эмпирические исследования), типологии (теоретические исследования), периодизации (исторические исследования).

Заключение.

Заключение — последняя часть научного текста. В ней краткой и сжатой форме излагаются полученные результаты, представляющие собой ответ на главный вопрос исследования. Здесь же могут намечаться и дальнейшие перспективы развития темы. Небольшое по объему сообщение также не может обойтись без заключительной части - пусть это будут две-три фразы. Но в них должен подводиться итог проделанной работы.

Список использованной литературы.

Реферат любого уровня сложности обязательно сопровождается списком используемой литературы. Названия книг в списке располагают по алфавиту с указанием выходных данных использованных книг.

Требования, предъявляемыек оформлению реферата

Объемы рефератов – от 10-18 машинописных страниц. Работа выполняется на одной стороне листа стандартного формата. По обеим сторонам листа оставляются поля размером 35 мм. слева и 15 мм. справа, рекомендуется шрифт 12-14, интервал - 1,5. Все листы реферата должны быть пронумерованы. Каждый вопрос в тексте должен иметь заголовок в точном соответствии с наименованием в плане-оглавлении. При написании и оформлении реферата следует избегать типичных ошибок, например, таких:

– поверхностное изложение основных теоретических вопросов выбранной темы, когда автор не понимает, какие проблемы в тексте являются главными, а какие второстепенными,

– в некоторых случаях проблемы, рассматриваемые в разделах, не раскрывают основных аспектов выбранной для реферата темы,

– дословное переписывание книг, статей, заимствования рефератов из интернета и т. д.

При проверке реферата преподавателем оцениваются:

1. Знания и умения на уровне требований стандарта конкретной дисциплины: знание фактического материала, усвоение общих представлений, понятий, идей.

2. Характеристика реализации цели и задач исследования (новизна и актуальность поставленных в реферате проблем, правильность формулирования цели, определения задач исследования, правильность выбора методов решения задач и реализации цели; соответствие выводов решаемым задачам, поставленной цели, убедительность выводов).

3. Степень обоснованности аргументов и обобщений (полнота, глубина, всесторонность раскрытия темы, логичность и последовательность изложения материала, корректность аргументации и системы доказательств, характер и достоверность примеров, иллюстративного материала, широта кругозора автора, наличие знаний интегрированного характера, способность к обобщению).

4. Качество и ценность полученных результатов (степень завершенности реферативного исследования, спорность или однозначность выводов).

5. Использование литературных источников.

6. Культура письменного изложения материала.

7. Культура оформления материалов работы.

Комплект типовых домашних заданий по практическим занятиям

Тема 1. Объектно- ориентированный подход к программированию.

Задание 1.1: Основы ООП в Ruby - создание класса "Банковский счет"

Цель: Освоить базовые принципы ООП: классы, объекты, инкапсуляция

Задача:

1. Создайте класс BankAccount в Ruby со следующими характеристиками:
   - Приватные поля: @account_number, @balance, @owner_name
   - Публичные методы: deposit(amount), withdraw(amount), display_balance
   - Конструктор для инициализации счета
2. Реализуйте:
   - Валидацию входных данных (сумма не может быть отрицательной)
   - Проверку достаточности средств при снятии
   - Геттер для номера счета с ограниченным доступом

Требования к коду:

```ruby

# Пример использования
account = BankAccount.new("123456", 1000, "Иван Иванов")
account.deposit(500)
account.withdraw(200)
account.display_balance # => "Баланс счета 123456: 1300 руб."
```

Задание 1.2: Наследование и полиморфизм - система сотрудников компании

Цель: Понять принципы наследования и полиморфизма

Задача:

1. Создайте иерархию классов в Ruby:
   - Базовый класс Employee с полями: name, position, salary
   - Производные классы: Manager, Developer, Designer
2. Реализуйте полиморфный метод calculate_bonus:
   - Manager: 20% от зарплаты
   - Developer: 15% + 1000 руб. за каждый завершенный проект
   - Designer: 10% + 500 руб. за каждый утвержденный дизайн

Пример реализации:

```ruby

employees = [
  Manager.new("Анна", 50000),
  Developer.new("Петр", 40000, 3),
  Designer.new("Мария", 35000, 5)
]

employees.each { |emp| puts emp.calculate_bonus }
```

Задание 1.3: Работа со строками и массивами в Ruby
Цель: Освоить методы работы со строками и массивами
Задача:

1. Создайте класс TextProcessor с методами:
    - analyze_text(text) - возвращает статистику (количество слов, символов, предложений)
    - find_most_common_words(text, n) - находит n самых частых слов
    - palindrome?(text) - проверяет, является ли строка палиндромом
2. Реализуйте модуль ArrayUtils с методами:
    - flatten_array(nested_array) - преобразует многомерный массив в одномерный
    - group_by_length(words_array) - группирует слова по длине

Требования:

- Использовать блоки и итераторы
- Обработать исключительные ситуации

Задание 1.4: Ввод-вывод и файловые операции в Ruby
Цель: Научиться работать с файлами и пользовательским вводом
Задача:

1. Создайте программу для управления списком задач (To-Do List):
    - Добавление, удаление, отметка выполнения задач
    - Сохранение в файл и загрузка из файла
    - Поиск и фильтрация задач
2. Реализуйте:
    - Чтение/запись в формате JSON
    - Валидацию пользовательского ввода
    - Интерфейс командной строки

Пример структуры:

```ruby
ruby

todo = TodoList.new
todo.add_task("Изучить ООП в Ruby", "2024-12-31")
todo.complete_task(1)
todo.save_to_file("tasks.json")
```

Задание 1.5: Простой Web-сервис на Sinatra
Цель: Создать базовый веб-сервис с использованием ООП
Задача:

1. Разработайте веб-приложение для учета книг в библиотеке:
    - GET /books - список всех книг
    - POST /books - добавление новой книги
    - GET /books/:id - информация о конкретной книге
    - PUT /books/:id - обновление информации
2. Создайте класс Book с полями:

- title, author, isbn, year, available

Пример реализации:

```ruby
require 'sinatra'
require 'json'

class Book
  attr_accessor :title, :author, :isbn, :year, :available

  def initialize(title, author, isbn, year)
    @title = title
    @author = author
    @isbn = isbn
    @year = year
    @available = true
  end
end
```

Задание 1.6: ООП в Kotlin - класс "Умный дом"

Цель: Освоить ООП в Kotlin, сравнить с Ruby

Задача:

1. Создайте систему "Умный дом" на Kotlin:
   - Базовый класс SmartDevice
   - Производные: SmartLight, SmartThermostat, SmartLock
   - Интерфейс Controllable с методами turnOn(), turnOff(), getStatus()
2. Реализуйте:
   - Data class для хранения состояния устройств
   - Object для управления всеми устройствами (Singleton)
   - Sealed class для типов событий

Пример на Kotlin:

kotlin

```kotlin
interface Controllable {
    fun turnOn()
    fun turnOff()
    fun getStatus(): String
}

class SmartLight(val name: String) : Controllable {
    private var isOn = false
    private var brightness = 0

    override fun turnOn() {
        isOn = true
        brightness = 50
    }

    override fun getStatus() = "Light $name: ${if (isOn) "ON" else "OFF"}"
}
```

Задание 1.7: Динамические механизмы в Ruby

Цель: Изучить метапрограммирование и динамические возможности Ruby

Задача:

1. Создайте модуль AttributeValidator для динамической валидации атрибутов:
   - Валидация присутствия (validates_presence_of)
   - Валидация формата (validates_format_of)
   - Валидация длины (validates_length_of)
2. Реализуйте использование через:
   - method_missing для динамического создания валидаторов
   - define_method для генерации методов проверки

Пример:

```ruby
ruby


class User
  include AttributeValidator

  validates_presence_of :name, :email
  validates_format_of :email, with: /\A[^@\s]+@[^@\s]+\z/

  attr_accessor :name, :email
end
```

Задание 1.8: Расширения в Kotlin

Цель: Освоить extension functions и свойства
Задача:

1. Создайте расширения для стандартных классов:
    - String.isPalindrome() - проверка палиндрома
    - List<T>.second() - получение второго элемента
    - Int.isPrime() - проверка простого числа
2. Реализуйте DSL для построения HTML:
    - Использование лямбд с receiver
    - Type-safe построение дерева элементов

Пример DSL:

```kotlin
kotlin

fun html(block: HTML.() -> Unit): HTML {
    return HTML().apply(block)
}

val page = html {
    head {
        title { +"My Page" }
    }
    body {
        h1 { +"Hello World" }
    }
}
```

Задание 1.9: Сравнительный анализ Ruby и Kotlin
Цель: Сравнить реализации ООП в двух языках
Задача:

1. Реализуйте одну и ту же систему на Ruby и Kotlin:
    - Класс Bank с методами управления счетами
    - Класс Transaction для операций
    - Модуль/интерфейс Reportable для генерации отчетов
2. Проанализируйте:
    - Различия в синтаксисе
    - Подходы к наследованию и композиции
    - Системы типов и их влияние на дизайн

Задание 1.10: Комплексный проект - система бронирования
Цель: Применить все изученные концепции в комплексном проекте
Задача:

1. Разработайте систему бронирования отелей:
    - Ruby backend с Sinatra
    - Kotlin mobile client (консольное приложение)
    - Обмен данными через JSON API
2. Модули системы:
    - Управление отелями и номерами

- Бронирование и отмена
- Система пользователей и аутентификации
- Генерация отчетов и статистики

Требования:

- Использование ООП принципов
- Proper error handling
- Валидация данных
- Юнит-тесты для критической функциональности

Тема 2. Порождающие паттерны.

Задание 2.1: Паттерн Singleton - система логирования

Цель: Освоить принципы работы Singleton и его практическое применение

Задача:

1. Реализуйте класс Logger как Singleton на Ruby и Kotlin
2. Функциональность:
   - Запись логов разных уровней (DEBUG, INFO, WARN, ERROR)
   - Ротация логов по размеру файла
   - Потокобезопасная запись

Требования к реализации:

```ruby
# Ruby
logger = Logger.instance
logger.debug("Отладочное сообщение")
logger.error("Ошибка в модуле X")
```

```kotlin
// Kotlin
val logger = Logger.getInstance()
logger.info("Информационное сообщение")
```

Критерии оценки:

- Гарантирован единственный экземпляр
- Потокобезопасность
- Гибкая конфигурация

Задание 2.2: Паттерн Factory Method - система уведомлений

Цель: Понять принцип виртуальных конструкторов

Задача:

1. Создайте систему отправки уведомлений:
   - Базовый класс Notification и метод send()
   - Производные: EmailNotification, SMSNotification, PushNotification

- Класс NotificationFactory с фабричным методом

2. Реализуйте на Ruby и Kotlin:

```ruby
# Ruby
factory = NotificationFactory.new
notification = factory.create_notification(:email)
notification.send("Заголовок", "Текст сообщения")
```

```kotlin
// Kotlin
val factory = NotificationFactory()
val notification = factory.createNotification(NotificationType.SMS)
notification.send("Заголовок", "Текст")
```

Задание 2.3: Паттерн Abstract Factory - кроссплатформенные UI компоненты

Цель: Освоить создание семейств связанных объектов

Задача:

1. Создайте Abstract Factory для UI компонентов:
   - Интерфейсы: Button, Checkbox, TextField
   - Фабрики: WindowsFactory, MacOSFactory, LinuxFactory
2. Реализация:

```kotlin
// Kotlin
interface GUIFactory {
    fun createButton(): Button
    fun createCheckbox(): Checkbox
}

class WindowsFactory : GUIFactory {
    override fun createButton() = WindowsButton()
    override fun createCheckbox() = WindowsCheckbox()
}
```

```ruby
# Ruby
class Application
  def initialize(factory)
    @factory = factory
    create_ui
  end

  def create_ui
    button = @factory.create_button
    checkbox = @factory.create_checkbox
  end
end
```

Задание 2.4: Паттерн Builder - построение сложных объектов
Цель: Научиться пошаговому конструированию объектов
Задача:

1. Реализуйте Builder для класса Computer:
   - Обязательные параметры: CPU, RAM
   - Опциональные: GPU, SSD, HDD, монитор
   - Валидация конфигурации
2. Fluent interface:

```ruby
# Ruby
computer = ComputerBuilder.new
  .with_cpu("Intel i7")
  .with_ram("16GB")
  .with_ssd("512GB")
  .with_gpu("NVIDIA RTX 3080")
  .build
```

```kotlin
// Kotlin
val computer = Computer.Builder()
    .cpu("Intel i7")
    .ram("16GB")
    .ssd("512GB")
    .gpu("NVIDIA RTX 3080")
    .build()
```

Задание 2.5: Паттерн Prototype - система клонирования документов
Цель: Изучить механизмы клонирования объектов
Задача:

1. Создайте систему документов с поддержкой клонирования:
   - Базовый класс Document с методом clone()
   - Производные: Report, Contract, Invoice
   - Регистр прототипов
2. Реализация:

```kotlin
// Kotlin
abstract class Document : Cloneable {
    var title: String = ""
    var content: String = ""
```

```
    public override fun clone(): Document {
        val cloned = super.clone() as Document
        cloned.title = this.title
        cloned.content = this.content
        return cloned
    }
}

class Report : Document() {
    var charts: List<Chart> = emptyList()

    override fun clone(): Report {
        val cloned = super.clone() as Report
        cloned.charts = this.charts.toList()
        return cloned
    }
}
```

Задание 2.6: Object Pool - пул соединений с базой данных
Цель: Освоить управление дорогостоящими ресурсами
Задача:

1. Реализуйте пул соединений с БД:
    - Ограничение максимального количества соединений
    - Проверка валидности соединений
    - Таймауты ожидания
2. Интерфейс:

```ruby
ruby

# Ruby
class ConnectionPool
  def initialize(max_size: 10)
    @max_size = max_size
    @available = []
    @in_use = []
  end

  def acquire
    # логика получения соединения
  end

  def release(connection)
    # возврат соединения в пул
  end
end
```

Задание 2.7: Сравнение Factory Method и Abstract Factory
Цель: Научиться выбирать подходящий паттерн
Задача:

1. Реализуйте обе фабрики для одной предметной области:
   - Factory Method: создание различных типов отчетов
   - Abstract Factory: создание UI компонентов для разных платформ
2. Проанализируйте:
   - Когда использовать каждый подход
   - Преимущества и недостатки
   - Сложность расширения

Задание 2.8: Lazy Initialization - отложенная загрузка ресурсов
Цель: Освоить ленивую инициализацию
Задача:

1. Реализуйте класс HeavyResource с отложенной загрузкой:
   - Загрузка происходит только при первом обращении
   - Thread-safe инициализация
   - Возможность сброса и повторной загрузки
2. Реализация в Kotlin:

```kotlin
kotlin

class HeavyResource {
    private val _data by lazy(LazyThreadSafetyMode.SYNCHRONIZED) {
        loadHeavyData()
    }

    val data: String
        get() = _data

    private fun loadHeavyData(): String {
        // имитация тяжелой загрузки
        Thread.sleep(5000)
        return "Загруженные данные"
    }
}
```

Задание 2.9: Dependency Injection - система плагинов
Цель: Понять принципы инверсии управления
Задача:

1. Создайте систему плагинов с DI:
   - Интерфейс Plugin с методом execute()
   - Класс PluginManager для управления плагинами
   - Внедрение зависимостей через конструктор
2. Реализация:

```ruby
# Ruby
class PluginManager
  def initialize(plugins = [])
    @plugins = plugins
  end

  def register_plugin(plugin)
    @plugins << plugin
  end

  def execute_all
    @plugins.each(&:execute)
  end
end

# Использование
manager = PluginManager.new([LoggerPlugin.new, NotifierPlugin.new])
manager.execute_all
```

Задание 2.10: Комплексная система - конфигуратор автомобилей
Цель: Применить несколько порождающих паттернов в одном проекте
Задача:

1. Разработайте систему конфигурации автомобилей:
   - Builder для пошагового конструирования
   - Factory Method для создания различных комплектаций
   - Prototype для быстрого клонирования популярных конфигураций
   - Singleton для глобального доступа к каталогу деталей
2. Функциональность:

```kotlin
// Kotlin
val director = CarConfigurationDirector()

val sportCar = director.buildSportsCar()
val familyCar = director.buildFamilyCar()

// Клонирование конфигурации
val anotherSportCar = sportCar.clone()
```

```ruby
ruby

# Ruby
class CarBuilder
  def initialize
    reset
  end

  def with_engine(type)
    @car.engine = EngineFactory.create_engine(type)
    self
  end

  def with_interior(style)
    @car.interior = InteriorFactory.create_interior(style)
    self
  end

  def build
    car = @car
    reset
    car
  end
end
```

Тема 3. Структурные паттерны

Задание 3.1: Паттерн Adapter - интеграция платежных систем
Цель: Освоить принцип согласования интерфейсов
Задача:

1. Создайте адаптеры для различных платежных систем:
   - Единый интерфейс PaymentProcessor с методом process_payment(amount)
   - Внешние сервисы: StripeAPI, PayPalSDK, BankTransfer с разными интерфейсами
   - Адаптеры: StripeAdapter, PayPalAdapter, BankAdapter

Требования к реализации:
ruby
 # Ruby
class PaymentProcessor
  def process_payment(amount)
    raise NotImplementedError
  end

```
end

class StripeAdapter < PaymentProcessor
  def initialize(stripe_api)
    @stripe = stripe_api
  end

  def process_payment(amount)
    @stripe.charge(amount * 100, 'usd') # Конвертация в центы
  end
end

# Использование
processors = [
  StripeAdapter.new(Stripe::Charge),
  PayPalAdapter.new(PayPal::Payment)
]

processors.each { |p| p.process_payment(100.0) }
```

```
// Kotlin
interface PaymentProcessor {
    fun processPayment(amount: Double): Boolean
}

class StripeAdapter(private val stripe: StripeAPI) : PaymentProcessor {
    override fun processPayment(amount: Double): Boolean {
        return stripe.charge((amount * 100).toInt(), "USD")
    }
}
```

Задание 3.2: Паттерн Bridge - система рендеринга графики

Цель: Разделить абстракцию и реализацию

Задача:

1. Создайте систему рендеринга с поддержкой разных API:
    - Абстракция: Shape (Circle, Rectangle, Triangle)
    - Реализация: Renderer (VectorRenderer, RasterRenderer)
    - Мост между фигурами и рендерерами

Реализация:

```
// Kotlin
interface Renderer {
    fun renderCircle(radius: Double)
    fun renderRectangle(width: Double, height: Double)
}
```

```
class VectorRenderer : Renderer {
    override fun renderCircle(radius: Double) {
        println("Rendering vector circle with radius $radius")
    }

    override fun renderRectangle(width: Double, height: Double) {
        println("Rendering vector rectangle ${width}x$height")
    }
}

abstract class Shape(protected val renderer: Renderer) {
    abstract fun draw()
    abstract fun resize(factor: Double)
}

class Circle(renderer: Renderer, private var radius: Double) : Shape(renderer)
{
    override fun draw() {
        renderer.renderCircle(radius)
    }

    override fun resize(factor: Double) {
        radius *= factor
    }
}
```

Задание 3.3: Паттерн Composite - файловая система

Цель: Освоить построение древовидных структур

Задача:

1. Реализуйте модель файловой системы:
    - Компонент: FileSystemComponent
    - Листья: File (размер, расширение)
    - Композиты: Directory (коллекция компонентов)
    - Операции: get_size(), find(name), display()

Реализация:

ruby

```
# Ruby
class FileSystemComponent
  def name
    raise NotImplementedError
  end

  def size
    raise NotImplementedError
```

```
  end

  def display(indent = 0)
    raise NotImplementedError
  end
end

class File < FileSystemComponent
  def initialize(name, size)
    @name = name
    @size = size
  end

  def display(indent = 0)
    puts "#{' ' * indent} 📄 #{@name} (#{@size} bytes)"
  end
end

class Directory < FileSystemComponent
  def initialize(name)
    @name = name
    @children = []
  end

  def add(component)
    @children << component
  end

  def size
    @children.sum(&:size)
  end

  def display(indent = 0)
    puts "#{' ' * indent} 📁 #{@name} (#{size} bytes)"
    @children.each { |child| child.display(indent + 2) }
  end
end

# Использование
root = Directory.new("root")
docs = Directory.new("Documents")
docs.add(File.new("report.pdf", 1024))
docs.add(File.new("presentation.pptx", 2048))
root.add(docs)
```

root.display

Задание 3.4: Паттерн Decorator - система уведомлений с дополнительной функциональностью

Цель: Научиться динамически добавлять обязанности

Задача:

1. Создайте систему уведомлений с декораторами:
   - Базовый компонент: Notifier с методом send(message)
   - Базовые реализации: EmailNotifier, SMSNotifier
   - Декораторы: EncryptionDecorator, CompressionDecorator, LoggingDecorator

Реализация:

kotlin

```kotlin
// Kotlin
interface Notifier {
    fun send(message: String): String
}

class EmailNotifier : Notifier {
    override fun send(message: String): String {
        return "Sending email: $message"
    }
}

abstract class NotifierDecorator(private val wrapped: Notifier) : Notifier {
    override fun send(message: String): String {
        return wrapped.send(message)
    }
}

class EncryptionDecorator(wrapped: Notifier) : NotifierDecorator(wrapped) {
    override fun send(message: String): String {
        val encrypted = "ENCRYPTED($message)"
        return super.send(encrypted)
    }
}

class LoggingDecorator(wrapped: Notifier) : NotifierDecorator(wrapped) {
    override fun send(message: String): String {
        println("LOG: Sending message - $message")
        val result = super.send(message)
        println("LOG: Message sent successfully")
        return result
    }
}
```

```
// Использование
fun main() {
    val notifier: Notifier = LoggingDecorator(
        EncryptionDecorator(
            EmailNotifier()
        )
    )

    println(notifier.send("Hello World!"))
}
```

Задание 3.5: Паттерн Facade - упрощенный API для работы с базой данных

Цель: Создать унифицированный интерфейс к сложной подсистеме

Задача:

1. Реализуйте фасад для работы с базой данных:
    - Скрыть сложность: подключение, транзакции, ошибки
    - Простой интерфейс: save_user(user), find_user(id), delete_user(id)
    - Внутренняя работа с Connection, Statement, ResultSet

Реализация:

ruby

```
# Ruby
class DatabaseFacade
  def initialize(connection_string)
    @connection = establish_connection(connection_string)
  end

  def save_user(user)
    @connection.transaction do
      validate_user(user)
      sql = "INSERT INTO users (name, email) VALUES (?, ?)"
      @connection.execute(sql, user.name, user.email)
      log_operation("User saved: #{user.name}")
    end
  rescue => e
    handle_error(e)
    false
  end

  def find_user(id)
    sql = "SELECT * FROM users WHERE id = ?"
    result = @connection.execute(sql, id)
    convert_to_user(result.first)
  end
```

```
  private

  def establish_connection(connection_string)
    # Сложная логика подключения
  end

  def validate_user(user)
    # Валидация данных
  end

  def handle_error(error)
    # Обработка ошибок
  end
end

# Использование
db = DatabaseFacade.new("postgresql://localhost/mydb")
db.save_user(User.new("John", "john@example.com"))
```

Задание 3.6: Паттерн Flyweight - система отображения текстового документа

Цель: Оптимизировать использование памяти через разделение состояния

Задача:

1. Создайте систему для отображения текста:
   - Разделяемое состояние: CharacterFlyweight (символ, шрифт, размер)
   - Внешнее состояние: позиция (x, y)
   - Фабрика flyweight-объектов

Реализация:

kotlin

```
// Kotlin
data class CharacterStyle(val font: String, val size: Int, val color: String)

class CharacterFlyweight(private val style: CharacterStyle) {
    fun display(character: Char, x: Int, y: Int) {
        println("Display '$character' at ($x, $y) with style: $style")
    }
}

class FlyweightFactory {
    private val flyweights = mutableMapOf<CharacterStyle, CharacterFlyweight>()
```

```
    fun getFlyweight(style: CharacterStyle): CharacterFlyweight {
        return flyweights.getOrPut(style) { CharacterFlyweight(style) }
    }
}

class Character(private val flyweight: CharacterFlyweight, private val char: Char, private val x: Int, private val y: Int) {
    fun display() {
        flyweight.display(char, x, y)
    }
}

// Использование
val factory = FlyweightFactory()
val style1 = CharacterStyle("Arial", 12, "Black")
val style2 = CharacterStyle("Times New Roman", 14, "Red")

val char1 = Character(factory.getFlyweight(style1), 'H', 0, 0)
val char2 = Character(factory.getFlyweight(style1), 'i', 1, 0) // Тот же flyweight
val char3 = Character(factory.getFlyweight(style2), '!', 2, 0)

listOf(char1, char2, char3).forEach { it.display() }
```

Задание 3.7: Паттерн Proxy - ленивая загрузка изображений
Цель: Контролировать доступ к объектам
Задача:

1. Реализуйте систему загрузки изображений:
    - Реальный субъект: HighResolutionImage (тяжелая загрузка)
    - Прокси: ImageProxy (ленивая загрузка, кэширование)
    - Виртуальный прокси отображает placeholder до загрузки

Реализация:
ruby

```
# Ruby
class Image
  def display
    raise NotImplementedError
  end
end

class HighResolutionImage < Image
  def initialize(filename)
    @filename = filename
    load_image_from_disk
  end
```

```
  def display
    puts "Displaying high resolution image: #{@filename}"
  end

  private

  def load_image_from_disk
    puts "Loading heavy image: #{@filename}"
    sleep(2) # Имитация долгой загрузки
  end
end

class ImageProxy < Image
  def initialize(filename)
    @filename = filename
    @real_image = nil
  end

  def display
    if @real_image.nil?
      @real_image = HighResolutionImage.new(@filename)
    end
    @real_image.display
  end
end

# Использование
image = ImageProxy.new("photo.jpg")
puts "Image proxy created"
image.display # Загрузка происходит здесь
image.display # Повторное использование загруженного изображения
```

Задание 3.8: Сравнение Adapter и Decorator

Цель: Научиться различать и правильно применять паттерны

Задача:

1. Реализуйте оба паттерна для одной предметной области:
   - Adapter: адаптация различных форматов данных (JSON, XML, CSV)
   - Decorator: добавление функциональности (валидация, кэширование, логирование)
2. Проанализируйте различия:
   - Цель применения
   - Структура классов
   - Влияние на интерфейс

Задание 3.9: Композит с Visitor - обход файловой системы
Цель: Комбинировать структурные и поведенческие паттерны
Задача:

1. Расширьте файловую систему из задания 3.3:
   - Добавьте паттерн Visitor для операций над элементами
   - Посетители: SizeCalculator, SearchVisitor, BackupVisitor

Реализация:

kotlin

```kotlin
// Kotlin
interface FileSystemVisitor {
    fun visit(file: File)
    fun visit(directory: Directory)
}

class SizeCalculator : FileSystemVisitor {
    private var totalSize = 0L

    override fun visit(file: File) {
        totalSize += file.size
    }

    override fun visit(directory: Directory) {
        // Директории учитываются в размере файлов
    }

    fun getTotalSize(): Long = totalSize
}

// В композит добавляем метод accept
interface FileSystemComponent {
    fun accept(visitor: FileSystemVisitor)
}

class File : FileSystemComponent {
    // ... остальная реализация

    override fun accept(visitor: FileSystemVisitor) {
        visitor.visit(this)
    }
}
```

Задание 3.10: Комплексная система - графический редактор
Цель: Применить все структурные паттерны в одном проекте
Задача:

1. Разработайте графический редактор:
    - **Composite**: иерархия графических элементов
    - **Bridge**: различные рендереры (vector, raster)
    - **Decorator**: эффекты для элементов (тень, градиент)
    - **Adapter**: импорт из разных форматов
    - **Facade**: упрощенный API для клиента
    - **Flyweight**: оптимизация одинаковых стилей
    - **Proxy**: ленивая загрузка тяжелых ресурсов

Пример архитектуры:

```
ruby
# Ruby
class GraphicEditor
  def initialize
    @document = Document.new
    @renderer = VectorRenderer.new # Bridge
    @import_adapters = [SVGAdapter.new, PNGAdapter.new] # Adapter
  end

  def add_shape(shape)
    decorated_shape = ShadowDecorator.new(
      GradientDecorator.new(shape) # Decorator
    )
    @document.add(decorated_shape)
  end

  def import_graphic(filename)
    adapter = find_adapter_for(filename)
    graphic = adapter.import(filename)
    @document.add(graphic)
  end

  def render
    @document.render(@renderer)
  end
end
```

Тема 4. Паттерны поведения

Задание 4.1: Паттерн Chain of Responsibility - система обработки заказов

Цель: Освоить построение цепочек обработчиков

Задача:

1. Создайте систему валидации заказов с цепочкой проверок:
    - StockValidationHandler - проверка наличия товара
    - PaymentValidationHandler - проверка платежа

- ShippingValidationHandler - проверка доставки
- Каждый обработчик может одобрить или отклонить заказ

Реализация на Ruby:

ruby
```
class OrderHandler
  attr_accessor :next_handler

  def handle(order)
    if next_handler
      next_handler.handle(order)
    else
      puts "Order processed successfully!"
    end
  end
end

class StockValidationHandler < OrderHandler
  def handle(order)
    if order.items.all? { |item| item.in_stock? }
      puts "Stock validation passed"
      super
    else
      puts "Stock validation failed: some items out of stock"
    end
  end
end

class PaymentValidationHandler < OrderHandler
  def handle(order)
    if order.payment_valid?
      puts "Payment validation passed"
      super
    else
      puts "Payment validation failed"
    end
  end
end

# Использование
order = Order.new(items: [item1, item2], payment: payment)
chain = StockValidationHandler.new
chain.next_handler = PaymentValidationHandler.new
chain.next_handler.next_handler = ShippingValidationHandler.new

chain.handle(order)
```

Задание 4.2: Паттерн Command - система управления задачами
Цель: Инкапсулировать запросы как объекты
Задача:

1. Реализуйте систему с поддержкой отмены операций:
   - Базовый класс Command с методами execute() и undo()
   - Конкретные команды: CreateTaskCommand, UpdateTaskCommand, DeleteTaskCommand
   - CommandHistory для хранения истории команд

Реализация на Kotlin:

```
kotlin
interface Command {
    fun execute()
    fun undo()
}

class CreateTaskCommand(private val taskManager: TaskManager, private val task: Task) : Command {
    private var taskId: String? = null

    override fun execute() {
        taskId = taskManager.createTask(task)
        println("Task created with ID: $taskId")
    }

    override fun undo() {
        taskId?.let {
            taskManager.deleteTask(it)
            println("Task $it deleted (undo)")
        }
    }
}

class TaskManager {
    private val tasks = mutableMapOf<String, Task>()
    private val history = CommandHistory()

    fun executeCommand(command: Command) {
        command.execute()
        history.push(command)
    }

    fun undo() {
        if (history.isNotEmpty()) {
```

```
            history.pop().undo()
        }
    }

    fun createTask(task: Task): String {
        val id = generateId()
        tasks[id] = task
        return id
    }

    fun deleteTask(id: String) {
        tasks.remove(id)
    }
}

// Использование
val manager = TaskManager()
val command = CreateTaskCommand(manager, Task("Learn Kotlin"))
manager.executeCommand(command)
manager.undo() // Отмена создания
```

Задание 4.3: Паттерн Interpreter - язык запросов к базе данных
Цель: Реализовать интерпретатор простого языка
Задача:

1. Создайте интерпретатор для SQL-подобного языка:
    - Поддержка операций: SELECT, WHERE, ORDER BY
    - Парсинг и выполнение запросов
    - Контекст с данными для выполнения

Реализация на Ruby:

```
ruby
class Expression
  def interpret(context)
    raise NotImplementedError
  end
end

class SelectExpression < Expression
  def initialize(columns)
    @columns = columns
  end

  def interpret(context)
    context.select(@columns)
  end
end
```

```
class WhereExpression < Expression
  def initialize(condition)
    @condition = condition
  end

  def interpret(context)
    context.where(@condition)
  end
end

class SQLInterpreter
  def initialize(expression)
    @expression = expression
  end

  def execute(context)
    @expression.interpret(context)
  end
end

# Использование
context = DatabaseContext.new(users_table)
expression = SelectExpression.new(["name", "email"])
  .chain(WhereExpression.new("age > 18"))
  .chain(OrderByExpression.new("name"))

interpreter = SQLInterpreter.new(expression)
result = interpreter.execute(context)
```

Задание 4.4: Паттерн Iterator - кастомные коллекции данных
Цель: Освоить механизмы последовательного доступа
Задача:

1. Реализуйте итераторы для различных структур данных:
    - TreeIterator для обхода дерева (in-order, pre-order, post-order)
    - GraphIterator для обхода графа (BFS, DFS)
    - FilteringIterator с предикатом

Реализация на Kotlin:

```
kotlin
interface Iterator<T> {
    fun hasNext(): Boolean
    fun next(): T
}
```

```
class TreeNode<T>(val value: T, val children: List<TreeNode<T>> = emptyList())

class TreeIterator<T>(private val root: TreeNode<T>) : Iterator<T> {
    private val stack = Stack<TreeNode<T>>()
    private var current: TreeNode<T>? = root

    override fun hasNext(): Boolean {
        return current != null || stack.isNotEmpty()
    }

    override fun next(): T {
        while (current != null) {
            stack.push(current)
            current = current!!.children.firstOrNull()
        }

        val node = stack.pop()
        current = node.children.getOrNull(1)
        return node.value
    }
}

class FilteringIterator<T>(
    private val iterator: Iterator<T>,
    private val predicate: (T) -> Boolean
) : Iterator<T> {
    private var nextItem: T? = null
    private var nextReady = false

    override fun hasNext(): Boolean {
        if (!nextReady) {
            findNext()
        }
        return nextItem != null
    }

    override fun next(): T {
        if (!nextReady) {
            findNext()
        }
        nextReady = false
        return nextItem ?: throw NoSuchElementException()
    }
```

```
    private fun findNext() {
        while (iterator.hasNext()) {
            val item = iterator.next()
            if (predicate(item)) {
                nextItem = item
                nextReady = true
                return
            }
        }
        nextItem = null
        nextReady = true
    }
}
```

Задание 4.5: Паттерн Mediator - система чат-комнаты
Цель: Централизовать взаимодействие объектов
Задача:

1. Создайте систему чата с посредником:
    - ChatMediator управляет рассылкой сообщений
    - User участники чата
    - Поддержка разных типов сообщений и комнат

Реализация на Ruby:

```
ruby
class ChatMediator
  def initialize
    @users = {}
    @rooms = {}
  end

  def register_user(user)
    @users[user.name] = user
    user.mediator = self
  end

  def create_room(room_name)
    @rooms[room_name] = ChatRoom.new(room_name)
  end

  def send_message(sender, room_name, message)
    room = @rooms[room_name]
    if room && room.has_user?(sender)
      room.users.each do |user|
        user.receive_message(sender.name, message) unless user == sender
      end
    end
```

```
  end

  def add_user_to_room(user, room_name)
    @rooms[room_name]&.add_user(user)
  end
end

class User
  attr_accessor :mediator
  attr_reader :name

  def initialize(name)
    @name = name
  end

  def send_message(room_name, message)
    puts "#{@name} sends: #{message}"
    mediator.send_message(self, room_name, message)
  end

  def receive_message(sender_name, message)
    puts "#{@name} received from #{sender_name}: #{message}"
  end
end

# Использование
mediator = ChatMediator.new
alice = User.new("Alice")
bob = User.new("Bob")

mediator.register_user(alice)
mediator.register_user(bob)
mediator.create_room("General")
mediator.add_user_to_room(alice, "General")
mediator.add_user_to_room(bob, "General")

alice.send_message("General", "Hello everyone!")
```

Задание 4.6: Паттерн Memento - система сохранения игры
Цель: Реализовать механизм сохранения и восстановления состояния
Задача:

1. Создайте систему для сохранения состояния игры:
    - GameMemento хранит состояние игры
    - GameCaretaker управляет снимками состояния
    - Автосохранение и ручные контрольные точки

Реализация на Kotlin:

kotlin

```
data class GameState(
    val level: Int,
    val score: Int,
    val playerHealth: Int,
    val inventory: List<String>,
    val timestamp: Long = System.currentTimeMillis()
)

class GameMemento(private val state: GameState) {
    fun getState(): GameState = state.copy()
}

class Game {
    private var level: Int = 1
    private var score: Int = 0
    private var playerHealth: Int = 100
    private var inventory: MutableList<String> = mutableListOf()

    fun createMemento(): GameMemento {
        return GameMemento(
            GameState(level, score, playerHealth, inventory.toList())
        )
    }

    fun restoreFromMemento(memento: GameMemento) {
        val state = memento.getState()
        level = state.level
        score = state.score
        playerHealth = state.playerHealth
        inventory = state.inventory.toMutableList()
        println("Game restored to level $level")
    }

    fun play() {
        level++
        score += 100
        playerHealth -= 10
        inventory.add("Item_$level")
        println("Playing... Level: $level, Score: $score, Health: $playerHealth")
    }
}

class GameCaretaker {
```

```
    private val saves = mutableListOf<GameMemento>()

    fun saveState(game: Game) {
        saves.add(game.createMemento())
        println("Game state saved. Total saves: ${saves.size}")
    }

    fun restoreState(game: Game, index: Int = saves.size - 1) {
        if (index in saves.indices) {
            game.restoreFromMemento(saves[index])
        }
    }

    fun showSaves() {
        saves.forEachIndexed { index, memento ->
            val state = memento.getState()
            println("Save $index: Level ${state.level}, Score ${state.score}")
        }
    }
}
```

Задание 4.7: Паттерн Observer - система уведомлений о событиях
Цель: Реализовать механизм подписки и уведомлений
Задача:

1. Создайте систему событий для приложения:
    - EventBus централизованная система событий
    - Подписчики разных типов: EmailSubscriber, LogSubscriber, AnalyticsSubscriber
    - Фильтрация событий по типу

Реализация на Ruby:

```
ruby
module Observable
  def add_observer(observer)
    @observers ||= []
    @observers << observer
  end

  def delete_observer(observer)
    @observers&.delete(observer)
  end

  def notify_observers(event)
    @observers&.each { |observer| observer.update(event) }
  end
end
```

```
class EventBus
  include Observable

  def publish(event)
    puts "Publishing event: #{event.type}"
    notify_observers(event)
  end
end

class Event
  attr_reader :type, :data, :timestamp

  def initialize(type, data = {})
    @type = type
    @data = data
    @timestamp = Time.now
  end
end

class EmailSubscriber
  def update(event)
    if event.type == :user_registered
      send_welcome_email(event.data[:user])
    end
  end

  private

  def send_welcome_email(user)
    puts "Sending welcome email to #{user.email}"
  end
end

# Использование
event_bus = EventBus.new
event_bus.add_observer(EmailSubscriber.new)
event_bus.add_observer(LogSubscriber.new)

event_bus.publish(Event.new(:user_registered, { user: user }))
```

Задание 4.8: Паттерн State - автомат продажи напитков
Цель: Управлять поведением в зависимости от состояния
Задача:

1. Реализуйте автомат по продаже напитков:

- Состояния: ReadyState, PaymentState, DispenseState, SoldOutState
- Переходы между состояниями
- Обработка действий пользователя

Реализация на Kotlin:

kotlin

```
interface VendingMachineState {
    fun insertMoney(amount: Int)
    fun selectProduct(productId: String)
    fun dispenseProduct()
    fun cancel()
}

class ReadyState(private val machine: VendingMachine) : VendingMachineState {
    override fun insertMoney(amount: Int) {
        machine.addBalance(amount)
        println("Balance: ${machine.balance}")
        machine.setState(PaymentState(machine))
    }

    override fun selectProduct(productId: String) {
        println("Please insert money first")
    }

    override fun dispenseProduct() {
        println("Please select product and insert money")
    }

    override fun cancel() {
        println("No operation to cancel")
    }
}

class VendingMachine {
    private var state: VendingMachineState = ReadyState(this)
    var balance: Int = 0
    private val products = mutableMapOf<String, Product>()

    fun setState(newState: VendingMachineState) {
        this.state = newState
    }

    fun addBalance(amount: Int) {
        balance += amount
    }
```

```
    fun resetBalance() {
        balance = 0
    }

    // Делегирование методов состоянию
    fun insertMoney(amount: Int) = state.insertMoney(amount)
    fun selectProduct(productId: String) = state.selectProduct(productId)
    fun dispenseProduct() = state.dispenseProduct()
    fun cancel() = state.cancel()
}
```

Задание 4.9: Паттерн Strategy - система сортировки данных
Цель: Инкапсулировать семейство алгоритмов
Задача:

1. Создайте систему с различными алгоритмами сортировки:
    - BubbleSortStrategy, QuickSortStrategy, MergeSortStrategy
    - Контекст для выбора стратегии
    - Сравнение производительности алгоритмов

Реализация на Ruby:

```
ruby
class SortStrategy
  def sort(data)
    raise NotImplementedError
  end
end

class BubbleSortStrategy < SortStrategy
  def sort(data)
    puts "Sorting using Bubble Sort"
    # Реализация пузырьковой сортировки
    data.dup.sort
  end
end

class QuickSortStrategy < SortStrategy
  def sort(data)
    puts "Sorting using Quick Sort"
    # Реализация быстрой сортировки
    data.dup.sort
  end
end

class Sorter
  attr_accessor :strategy
```

```
  def initialize(strategy = QuickSortStrategy.new)
    @strategy = strategy
  end

  def sort(data)
    @strategy.sort(data)
  end
end

# Использование
data = [5, 2, 8, 1, 9]
sorter = Sorter.new(BubbleSortStrategy.new)
result1 = sorter.sort(data)

sorter.strategy = QuickSortStrategy.new
result2 = sorter.sort(data)
```

Задание 4.10: Комплексная система - workflow обработки заказов
Цель: Применить несколько поведенческих паттернов в одном проекте
Задача:

1. Разработайте систему обработки заказов:
    - **Chain of Responsibility**: цепочка валидации заказа
    - **Command**: операции с заказом (создание, обновление, отмена)
    - **Observer**: уведомления о изменениях статуса
    - **State**: управление статусами заказа
    - **Strategy**: различные стратегии доставки

Реализация на Kotlin:

```
kotlin
class OrderWorkflowSystem {
    private val commandProcessor = CommandProcessor()
    private val eventBus = EventBus()
    private val validationChain = createValidationChain()

    fun processOrder(orderData: OrderData) {
        // Chain of Responsibility - валидация
        if (!validationChain.validate(orderData)) {
            throw ValidationException("Order validation failed")
        }

        // Command - создание заказа
        val command = CreateOrderCommand(orderData, eventBus)
        commandProcessor.execute(command)

        // State - управление статусами
```

```
        val order = command.getOrder()
        order.setState(ProcessingState(order))

        // Observer - уведомления
        eventBus.publish(OrderCreatedEvent(order.id))
    }

    private fun createValidationChain(): OrderValidator {
        return StockValidator()
            .setNext(PaymentValidator())
            .setNext(AddressValidator())
    }
}
```

Критерии оценивания выполнения практического задания:

Оценка «отлично» выставляется, если: работа выполнена в полном объеме с соблюдением необходимой последовательности действий; работа проведена в условиях, обеспечивающих получение правильных результатов и выводов; соблюдены правила техники безопасности; в ответе правильно и аккуратно выполняет все записи, таблицы, рисунки, чертежи, графики, вычисления; правильно выполняет анализ ошибок.

Оценка «хорошо» выставляется, если: работа выполнена правильно с учетом 1-2 мелких погрешностей или 2-3 недочетов, исправленных самостоятельно по требованию преподавателя.

Оценка «удовлетворительно» выставляется, если: работа выполнена правильно не менее чем наполовину, допущены 1-2 погрешности или одна грубая ошибка.

Оценка «неудовлетворительно» выставляется, если: допущены две (и более) грубые ошибки в ходе работы, которые обучающийся не может исправить даже по требованию преподавателя или работа не выполнена полностью.

Типовые тесты

Тема 1. Объектно-ориентированный подход к программированию

П1. Что такое инкапсуляция в ООП?

а) Наследование свойств родительского класса

б) Объединение данных и методов в единый объект с контролем доступа ✓

в) Возможность объектов иметь разные формы

г) Создание множества экземпляров класса

П2. Какой принцип ООП позволяет классу-потомку использовать методы родительского класса?

а) Инкапсуляция

б) Полиморфизм

в) Наследование ✓
г) Абстракция

П3. Как создать класс в Ruby?

а) class MyClass {}
б) class MyClass end ✓
в) MyClass = class()
г) def class MyClass

П4. Что выведет код в Ruby: "hello".capitalize**?**

а) HELLO
б) Hello ✓
в) hello
г) hELLO

П5. Как объявить переменную экземпляра в Ruby?

а) @variable_name ✓
б) $variable_name
в) @@variable_name
г) variable_name

П6. Что такое полиморфизм?

а) Сокрытие реализации методов
б) Возможность объектов разных классов отвечать на одинаковые сообщения ✓
в) Создание иерархии классов
г) Защита данных от прямого доступа

П7. Какой метод используется для чтения пользовательского ввода в Ruby?

а) read()
б) input()
в) gets ✓
г) scan()

П8. Что означает ключевое слово open **в Kotlin?**

а) Закрытый класс
б) Класс, который можно наследовать ✓
в) Открытая переменная
г) Публичный метод

П9. Как создать массив в Ruby?

а) array = [1, 2, 3] ✓
б) array = (1, 2, 3)
в) array = {1, 2, 3}
г) array = array(1, 2, 3)

П10. Что такое data class в Kotlin?

а) Класс для работы с базами данных
б) Класс, который автоматически генерирует методы equals, hashCode, toString ✓

в) Абстрактный класс
г) Класс для хранения настроек

Тема 2. Порождающие паттерны

П1. Какой паттерн гарантирует, что у класса будет только один экземпляр?

а) Factory Method
б) Singleton ✓
в) Builder
г) Prototype

П2. Что представляет собой паттерн Abstract Factory?

а) Создание объектов без указания их конкретных классов ✓
б) Пошаговое создание сложных объектов
в) Клонирование существующих объектов
г) Создание семейств связанных объектов

П3. Какой паттерн используется для создания объектов с множеством опциональных параметров?

а) Singleton
б) Factory Method
в) Builder ✓
г) Prototype

П4. Что является главной целью паттерна Factory Method?

а) Создание единственного экземпляра класса
б) Определение интерфейса для создания объекта, но оставление решения о классе создаваемого объекта подклассам ✓
в) Построение сложных объектов шаг за шагом
г) Создание копий существующих объектов

П5. Какой паттерн позволяет копировать объекты без знания их конкретных классов?

а) Singleton
б) Builder
в) Prototype ✓
г) Abstract Factory

П6. В каком случае следует использовать паттерн Singleton?

а) Когда нужно создавать много похожих объектов
б) Когда класс должен иметь только один экземпляр ✓
в) Когда объекты имеют сложный процесс создания
г) Когда нужно создавать семейства связанных объектов

П7. Что такое "ленивая инициализация" в контексте Singleton?

а) Создание объекта при первом обращении ✓
б) Быстрое создание объекта
в) Создание объекта в фоновом потоке
г) Отложенное создание объекта на несколько минут

П8. Какой паттерн лучше всего подходит для создания различных UI компонентов для разных операционных систем?

а) Builder

б) Prototype

в) Abstract Factory ✓

г) Singleton

П9. Что характеризует паттерн Builder?

а) Использование одного и того же класса для создания разных объектов

б) Разделение процесса построения объекта от его представления ✓

в) Создание объектов через клонирование

г) Ограничение создания только одного экземпляра

П10. Какой метод обычно используется в Prototype для создания копий?

а) create()

б) build()

в) clone() ✓

г) copy()

Тема 3. Структурные паттерны

П1. Какой паттерн позволяет объектам с несовместимыми интерфейсами работать вместе?

а) Decorator

б) Adapter ✓

в) Bridge

г) Composite

П2. Что представляет собой паттерн Bridge?

а) Объединение объектов в древовидные структуры

б) Разделение абстракции и реализации ✓

в) Динамическое добавление новой функциональности объектам

г) Преобразование интерфейса одного класса в интерфейс другого

П3. Какой паттерн позволяет сгруппировать объекты в древовидные структуры?

а) Adapter

б) Decorator

в) Composite ✓

г) Bridge

П4. Что делает паттерн Decorator?

а) Изменяет интерфейс объекта

б) Динамически добавляет новую функциональность объекту ✓

в) Создает мост между абстракцией и реализацией

г) Объединяет объекты в иерархии

П5. В чем основное отличие Adapter от Decorator?

а) Adapter изменяет интерфейс, Decorator добавляет функциональность ✓

б) Decorator изменяет интерфейс, Adapter добавляет функциональность
в) Оба делают одно и то же
г) Adapter для классов, Decorator для объектов

П6. Когда следует использовать паттерн Composite?

а) Когда нужно работать с иерархиями объектов "часть-целое" ✓
б) Когда нужно добавить новую функциональность объекту
в) Когда нужно разделить абстракцию и реализацию
г) Когда нужно адаптировать несовместимые интерфейсы

П7. Какой паттерн помогает избежать наследования для расширения функциональности?

а) Adapter
б) Bridge
в) Composite
г) Decorator ✓

П8. Что такое "компонент" в паттерне Composite?

а) Базовый класс для всех объектов в структуре ✓
б) Декоратор для добавления функциональности
в) Адаптер для преобразования интерфейса
г) Мост между абстракциями

П9. Какой паттерн использует принцип "предпочтение композиции перед наследованием"?

а) Singleton
б) Decorator ✓
в) Factory Method
г) Prototype

П10. В чем преимущество паттерна Bridge?

а) Позволяет создавать сложные объекты пошагово
б) Разделяет абстракцию и реализацию, позволяя им изменяться независимо ✓
в) Гарантирует единственный экземпляр класса
г) Позволяет клонировать объекты

Тема 4. Паттерны поведения

П1. Какой паттерн передает запрос по цепочке обработчиков?

а) Command
б) Chain of Responsibility ✓
в) Mediator
г) Iterator

П2. Что инкапсулирует паттерн Command?

а) Запрос как объект ✓
б) Состояние объекта
в) Алгоритмы сортировки
г) Итерацию по коллекции

П3. Какой паттерн используется для оценки языковых грамматик?

а) Iterator

б) Mediator
в) Interpreter ✓
г) Chain of Responsibility

П4. Что предоставляет паттерн Iterator?

а) Способ последовательного доступа к элементам коллекции ✓
б) Централизованное управление взаимодействием объектов
в) Механизм отмены операций
г) Обработку запросов через цепочку

П5. Какой паттерн определяет объект, инкапсулирующий способ взаимодействия множества объектов?

а) Command
б) Chain of Responsibility
в) Mediator ✓
г) Iterator

П6. В чем основное преимущество паттерна Command?

а) Позволяет откладывать выполнение запросов и поддерживать отмену операций ✓
б) Обеспечивает последовательный доступ к элементам
в) Упрощает грамматики языков
г) Централизует управление взаимодействием

П7. Когда следует использовать паттерн Chain of Responsibility?

а) Когда нужно выполнить запрос, но неизвестно, какой объект должен его обработать ✓
б) Когда нужно инкапсулировать запрос как объект
в) Когда нужно предоставить последовательный доступ к коллекции
г) Когда нужно определить грамматику языка

П8. Что такое "получатель" в паттерне Command?

а) Объект, который знает, как выполнить операцию ✓
б) Объект, который хранит историю команд
в) Объект, который инициирует запрос
г) Объект, который инкапсулирует запрос

П9. Какой паттерн полезен для реализации различных алгоритмов обхода коллекций?

а) Mediator
б) Iterator ✓
в) Interpreter
г) Chain of Responsibility

П10. В чем заключается основная идея паттерна Mediator?

а) Инкапсулировать запрос как объект
б) Предоставить последовательный доступ к элементам
в) Убрать прямые связи между объектами, перенеся взаимодействие через посредника ✓
г) Передавать запрос по цепочке обработчиков

Критерии оценивания тестовых заданий для текущего контроля

Максимальное количество баллов за решение одного теста составляет 5 баллов, которые переводятся в академические оценки следующим образом:

- оценка «отлично» (5 баллов) выставляется при условии правильного ответа студента не менее чем на 85% контрольных заданий

- оценка «хорошо» (4 балла) выставляется при условии правильного ответа студента не менее чем на 70% контрольных заданий

- оценка «удовлетворительно» (3 балла) выставляется при условии правильного ответа студента не менее чем на 51% контрольных заданий

- оценка «неудовлетворительно» (1-2 балла) выставляется при условии правильного ответа студента менее чем на 50% контрольных заданий

2.2 Оценочные материалы для проведения промежуточной аттестации

ПК-3.4.1 Анализирует компьютерную систему с целью определения уровня защищённости и доверия

П1. Какой из порождающих паттернов может представлять угрозу безопасности при неправильном использовании в многопоточных системах защиты информации?

а) Factory Method;

б) Singleton при отсутствии потокобезопасной реализации; [✓]

в) Builder;

г) Prototype.

Ответ: б) Singleton при отсутствии потокобезопасной реализации

П2. Какой структурный паттерн может быть использован для анализа безопасности доступа к компонентам системы через единый интерфейс?

а) Adapter;

б) Facade для централизованного контроля доступа; [✓]

в) Decorator;

г) Composite.

Ответ: б) Facade для централизованного контроля доступа

П3. При анализе защищенности системы, какой поведенческий паттерн помогает выявить цепочки обработки конфиденциальных данных?

а) Iterator;

б) Chain of Responsibility для трассировки потока данных; [✓]

в) Command;

г) Mediator.

Ответ: б) Chain of Responsibility для трассировки потока данных

П4. Какой аспект ООП в Ruby требует особого внимания при анализе защищенности из-за динамической природы языка?

а) Статическая типизация;

б) Открытые классы и monkey patching; [✓]

в) Строгая инкапсуляция;
г) Компиляция в байт-код.
Ответ: б) Открытые классы и monkey patching

П5. Установите соответствие между паттерном проектирования и потенциальной уязвимостью при анализе безопасности:

Паттерн	Потенциальная уязвимость
1. Singleton	а) Глобальное состояние и точки отказа
2. Decorator	б) Цепочка доверия при обработке данных
3. Iterator	в) Несанкционированный доступ к элементам коллекции

Правильный ответ:
1 – а, 2 – б, 3 – в

П6. Установите соответствие между элементом ООП и методом анализа защищенности:

Элемент ООП	Метод анализа
1. Наследование	а) Проверка целостности иерархии классов
2. Инкапсуляция	б) Аудит контроля доступа к данным
3. Полиморфизм	в) Анализ динамической диспетчеризации методов

Правильный ответ:
1 – а, 2 – б, 3 – в

П7. Установите правильную последовательность анализа уровня защищенности компьютерной системы:
а) Идентификация критических активов и данных.
б) Анализ архитектуры системы и точек входа.
в) Оценка механизмов контроля доступа.
г) Тестирование на уязвимости.
д) Формирование отчета о уровне доверия.
Ответ: АБВГД

П8. Установите правильную последовательность оценки защищенности распределенной системы:
а) Анализ сетевых взаимодействий и протоколов.
б) Проверка аутентификации и авторизации.
в) Оценка шифрования передаваемых данных.
г) Тестирование устойчивости к атакам.
д) Определение общего уровня доверия к системе.
Ответ: БАВГД

П9. Как называется метод анализа безопасности, при котором система рассматривается как набор взаимодействующих компонентов?
Правильный ответ: Компонентный анализ безопасности (Component Security Analysis)

П10. Как называется характеристика системы, определяющая ее способность противостоять атакам?
Правильный ответ: Уровень защищенности (Security Level) или Устойчивость к атакам (Attack Resilience)

ПК-3.4.2 Демонстрирует умение проводить анализ безопасности компонентов программных и программно-аппаратных средств защиты информации в компьютерных системах

П1. При анализе безопасности паттерна Composite в системе управления файлами, на что следует обратить особое внимание?

а) На возможность несанкционированного доступа к отдельным компонентам дерева; [✓]

б) На скорость выполнения операций;

в) На использование памяти;

г) На сложность реализации.

Ответ: а) На возможность несанкционированного доступа к отдельным компонентам дерева

П2. Какой аспект паттерна Adapter требует проверки при анализе безопасности интеграции с внешними системами?

а) Производительность преобразования данных;

б) Корректность и безопасность преобразования интерфейсов; [✓]

в) Количество адаптеров в системе;

г) Сложность кода адаптера.

Ответ: б) Корректность и безопасность преобразования интерфейсов

П3. При анализе безопасности реализации паттерна Command в системе управления доступом, что является критически важным?

а) Скорость выполнения команд;

б) Гарантия невозможности подмены или перехвата команд; [✓]

в) Количество поддерживаемых команд;

г) Удобство интерфейса.

Ответ: б) Гарантия невозможности подмены или перехвата команд

Р4. Какой механизм в Kotlin требует особого внимания при анализе безопасности программно-аппаратных средств защиты?

а) Extension functions;

б) Inline classes для типобезопасных оберток критических данных; [✓]

в) Data classes;

г) Coroutines.

Ответ: б) Inline classes для типобезопасных оберток критических данных

П5. Установите соответствие между компонентом системы и методом анализа его безопасности:

Компонент	Метод анализа безопасности
1. Singleton	а) Проверка потокобезопасности и контроля доступа
2. Factory Method	б) Анализ валидации создаваемых объектов
3. Observer	в) Проверка безопасности каналов уведомлений

Правильный ответ:

1 – а, 2 – б, 3 – в

П6. Установите соответствие между уязвимостью и паттерном, который может ее устранить:

Уязвимость	Защитный паттерн

1. Прямой доступ к данным	а) Facade для скрытия сложности
2. Несанкционированные вызовы методов	б) Proxy для контроля доступа
3. Утечка через сложный интерфейс	в) Decorator для добавления проверок

Правильный ответ:

1 – б, 2 – в, 3 – а

П7. Установите правильную последовательность анализа безопасности программного компонента:

а) Изучение архитектуры и интерфейсов компонента.

б) Анализ обработки входных данных и валидации.

в) Проверка механизмов контроля доступа.

г) Тестирование на граничные условия и инъекции.

д) Оценка устойчивости к атакам.

Ответ: АБВГД

П8. Установите правильную последовательность тестирования безопасности аппаратного компонента:

а) Анализ спецификаций и документации.

б) Тестирование физических интерфейсов ввода-вывода.

в) Проверка защищенных областей памяти.

г) Анализ устойчивости к side-channel атакам.

д) Оценка общего уровня защищенности.

Ответ: АБВГД

П9. Как называется анализ, направленный на поиск уязвимостей, связанных с временными характеристиками выполнения операций?

Правильный ответ: Анализ временных атак (Timing Attack Analysis) или Side-channel анализ

П10. Как называется метод проверки безопасности, при котором компонент тестируется в условиях, максимально приближенных к реальным?

Правильный ответ: Функциональное тестирование безопасности (Security Functional Testing) или Penetration testing

ПК-3.4.3 Формирует предложения по устранению выявленных уязвимостей

П1. При обнаружении уязвимости в Singleton из-за отсутствия потокобезопасности, какое решение является наиболее эффективным?

а) Удаление паттерна из системы;

б) Использование Double-Checked Locking или thread-safe инициализации; [✓]

в) Добавление комментариев в код;

г) Увеличение скорости выполнения.

Ответ: б) Использование Double-Checked Locking или thread-safe инициализации

П2. Какое предложение по устранению уязвимости следует внести при обнаружении возможности несанкционированного доступа через паттерн

Composite?
а) Упрощение иерархии классов;
б) Внедрение механизма проверки прав доступа для каждого компонента; [✓]
в) Увеличение производительности;
г) Изменение структуры данных.
Ответ: б) Внедрение механизма проверки прав доступа для каждого компонента

П3. При выявлении уязвимости в цепочке обработки данных (Chain of Responsibility), какое улучшение следует предложить?
а) Увеличение количества обработчиков;
б) Внедрение валидации и санитизации данных на каждом этапе; [✓]
в) Ускорение обработки запросов;
г) Упрощение цепочки.
Ответ: б) Внедрение валидации и санитизации данных на каждом этапе

П4. Какое предложение по устранению уязвимости эффективно для паттерна Decorator при обнаружении возможности подмены декораторов?
а) Удаление декораторов из системы;
б) Внедрение механизма цифровой подписи декораторов; [✓]
в) Увеличение скорости выполнения;
г) Изменение цветовой схемы интерфейса.
Ответ: б) Внедрение механизма цифровой подписи декораторов

П5. Установите соответствие между выявленной уязвимостью и предлагаемым решением:

Уязвимость	Предлагаемое решение
1. Состояние гонки в Observer	а) Синхронизация методов и использование thread-safe коллекций
2. Инъекция через Command	б) Валидация и санитизация параметров команд
3. Несанкционированный доступ через Iterator	в) Контроль доступа к методам итерации

Правильный ответ:
1 – а, 2 – б, 3 – в

П6. Установите соответствие между недостатком безопасности и улучшающим паттерном:

Недостаток безопасности	Улучшающий паттерн
1. Прямой доступ к сложной подсистеме	а) Facade с контролем доступа
2. Отсутствие контроля за созданием объектов	б) Factory Method с валидацией
3. Небезопасное хранение состояния	в) Memento с шифрованием

Правильный ответ:
1 – а, 2 – б, 3 – в

П7. Установите правильную последовательность устранения выявленных уязвимостей:
а) Приоритизация уязвимостей по уровню критичности.

б) Разработка плана исправления для каждой уязвимости.
в) Реализация защитных механизмов и исправлений.
г) Тестирование исправлений на корректность и безопасность.
д) Документирование изменений и обновление документации.
Ответ: АБВГД

П8. Установите правильную последовательность внедрения улучшений безопасности:
а) Анализ воздействия изменений на систему.
б) Разработка миграционного плана.
в) Поэтапное внедрение улучшений.
г) Мониторинг работы системы после изменений.
д) Корректировка улучшений по результатам мониторинга.
Ответ: АБВГД

П9. Как называется процесс постепенного внедрения улучшений безопасности с минимальным воздействием на работу системы?
Правильный ответ: Поэтапное развертывание (Phased Deployment) или Инкрементальное внедрение

П10. Как называется документ, описывающий план устранения выявленных уязвимостей с указанием сроков и ответственных?
Правильный ответ: План исправления уязвимостей (Vulnerability Remediation Plan) или Дорожная карта безопасности

Критерии оценки для проведения зачета по дисциплине (тестирование)

Шкала оценивания результатов промежуточной аттестации и критерии выставления оценок.

Оценка	Уровень сформированности компетенции	Критерии
«Отлично» («зачтено»)	Высокий	Выполнено более 80% заданий предложенного теста
«Хорошо» («зачтено»)	Хороший	Выполнено 60-80% заданий предложенного теста
«Удовлетворительно» («зачтено»)	Достаточный	Выполнено 35-60% заданий предложенного теста
«Неудовлетворительно» («не зачтено»)	Недостаточный	Выполнено менее 35% заданий предложенного теста

Обновление оценочных материалов дисциплины

обсуждено и рекомендовано к утверждению решением кафедры наименование кафедры
от ___ _____ 20_ г., протокол № ___

одобрено Научно-методическим советом _____ от ___ _____ 20__ г., протокол №____