КАЗАНСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

А.А. АНДРИАНОВА

ПРАКТИКУМ ПО КУРСУ «ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ И ПРОЕКТИРОВАНИЕ»

Казань – 2025

УДК 004.415.2
ББК 32.973.44

Принято на заседании учебно-методической комиссии ИВМиИТ
Протокол № 4 от 20 декабря 2024 года

Рецензенты:
Старший преподаватель **Т.М. Мухтарова;**
кандидат физико-математических наук, доцент **В.В. Бандеров**

Андрианова А.А.
Практикум по курсу «Объектно-ориентированный анализ и проектирование» / А.А. Андрианова – Казань: Казанский федеральный университет, 2025. – 72 с.

Разработка программного обеспечения – сложный многоэтапный процесс, который представляет собой командную работу специалистов различной специализации, причем непосредственно написание программного кода в этом процессе занимает вовсе не преимущественную долю. Данное учебно-методическое пособие содержит описание практикума по дисциплине «Объектно-ориентированный анализ и проектирование». Практикум предоставляет основные теоретические сведения и описание практических приемов создания программного проекта, включая приемы на этапах работы по анализу требований и формированию проекта программного обеспечения на основе объектно-ориентированной концепции в рамках технологии UML.

Учебно-методическое пособие разработано для студентов, обучающихся по направлению «Фундаментальная информатика и информационные технологии», но может представлять интерес и для студентов других направлений бакалавриата, которые специализируются на разработке программного обеспечения.

УДК 004.415.2
ББК 32.973.44

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Программное обеспечение – это особый вид продукта, который имеет свои специфические свойства, отличающие его от других видов результатов производства. Наиболее важными отличительными чертами программного обеспечения как продукта является то, что большая часть программного обеспечения обладает уникальным набором функций, ориентированным на требования заказчика, а также развитие продукта в процессе эксплуатации, его многоверсионность, обновляемость, распространение путем автоматизированного обновления.

Эти свойства накладывают дополнительную сложность на организацию процесса создания программного обеспечения, так как этот процесс должен учитывать то, что в процессе эксплуатации будут возникать требования на изменения и внесение этих изменений должно учитывать преемственность и согласование версий продукта, чтобы не вызвать сложностей для пользователей, которые уже участвуют в его эксплуатации.

Учет этих аспектов в первую очередь ложится не на программистов, а на архитекторов программного обеспечения – специалистов, которые занимаются проектированием программного обеспечения с точки зрения формирования его архитектуры – структуры и определения взаимосвязей отдельных частей программного обеспечения, распределения функций между этими частями. Именно продуманная архитектура позволяет быстро вносить изменения в уже работающее программное обеспечение, определить принципы этих изменений и тем самым уменьшить временные и трудовые затраты на создание новой версии программного обеспечения.

Архитектуры работают не с непосредственным программным продуктом, а с его проектом еще до начала его создания. Программный проект и его составляющие являются исходными данными не только для работы программистов-кодировщиков, но и, например, для тестировщиков для формирования набора тестов для проверки выполнения требований заказчика, или для проектировщиков пользовательского

интерфейса, которые позволят сделать программное обеспечение удобным и интуитивно понятным для пользователя. Поэтому работу по проектированию программного обеспечения можно считать ключевой в разработке программного продукта.

Описание программного проекта включает множество различных факторов. Оно должно отражать структуру элементов программного кода (модулей, подсистем, классов и пр.), определение их взаимосвязей между собой, зависимостей и видимости элементов программного кода, распределение функций между элементами программного кода, соответствие функций требованиям заказчика, требованиям к используемым алгоритмам реализации функций, форматам хранения и передачи данных и другие вопросы.

Поскольку программное обеспечение является результатом командной работы, проект должен однозначно пониматься всеми членами команды разработчиков независимо от их роли в команде, поэтому в настоящее время существует несколько унифицированных методологий, определяющих принципы описания программного проекта. Большинство из этих методологий представляют проект в виде набора графических элементов. Визуализации является наиболее удобным способом представления сложной структурированной информации, так как известно, что большую часть информации человек получает именно визуальным образом. Именно этот тип информации является наиболее удобным и интуитивно понятным для человека. В рамках данного учебно-методического пособия будет рассматриваться одна из таких методологий, основанная на применении объектно-ориентированной концепции и универсального языка моделирования UML, который является наиболее популярном в настоящее время при проектировании программного обеспечения.

Данное пособие адресовано студентам, изучающим дисциплину «Объектно-ориентированный анализ и проектирование» по направлению подготовки бакалавриата «Фундаментальная информатика и информационные технологии», реализуемой в Институте Вычислитель-

ной математики и информационных технологий Казанского (Приволжского) федерального университета, но может быть интересно и студентам других направлений, специализирующихся на разработке программного обеспечения.

В пособии собраны основные теоретические сведения о программном проекте, языке UML и его применении для описания программного проекта, описание и методические рекомендации по выполнению этапов командного проекта, который студенты реализуют в рамках лабораторных работ в течение семестра для создания проекта программного обеспечения. Результатом проекта является документация, описывающая список требований к программному обеспечению, результаты его анализа и набор диаграмм проекта, созданных с помощью технологии UML.

ФОРМАЛИЗАЦИЯ И АНАЛИЗ ТРЕБОВАНИЙ

Формализацией и анализом требований занимаются обычно бизнес-аналитики. Их главная роль в проекте – общение с заказчиком и своеобразный перевод требований с языка заказчика на язык разработчика. Результаты работы бизнес-аналитика является исходными данными для архитекторов и проектировщиков, поэтому принципы формализации и анализа требований должны быть известны и этим представителям команды разработчиков.

Бизнес-аналитик определяет требования исходя их анализа и исследования предметной области разработки. Совместно с представителями заказчика бизнес-аналитик согласовывает техническое задание на разработку программного обеспечения, которое уже содержит описание требований к программному обеспечению. Требования разделяют на функциональные и нефункциональные. Первый тип является наиболее важным, так как определяет, что какой функционал программное обеспечение должно предоставлять пользователю. К нефункциональным требованиям относятся разные виды требований, обеспечивающих качество выполнения функций программного обеспечения, например, требования безопасности, требования по производительности и масштабируемости, требования к дизайну пользовательского интерфейса и другие характеристики, которые при создании проекта, конечно, играют роль, но внимание на них уделяется уже в контексте созданного проекта и выбранной архитектуры. Поэтому основное внимание будем уделять в рамках этой дисциплины функциональным требованиям.

Любое требование должно быть формализовано в ожидаемой для других членов команды форме. Требование представляется в текстовой форме и представляется в виде предложений, сформулированных таким образом, чтобы удовлетворить следующим свойствам:

1. Ясность, недвусмысленность. Требование определено без обращения к техническому жаргону, акронимам и другим скрытым

формулировкам. Оно выражает объективные факты, а не субъективные мнения. Возможна одна и только одна интерпретация требования. Определение не содержит нечетких фраз. Использование отрицательных и составных утверждений запрещено, так как подобные формулировки могут затруднить их проверку или сместить внимание на неважную часть требования.

2. Завершенность. Требование полностью определено в одном пункте. Вся необходимая разработчикам информация присутствует в нем, чтобы не было необходимости обращаться к другим связанным с ним требованиям.

3. Проверяемость. То, что требование было реализовано в проекте и программном обеспечении, может быть однозначно проверено с помощью одного из четырех возможных методов: осмотр, демонстрация, тест или анализ. Любое требование должна иметь способ проверки его реализации.

4. Необходимость. Требование представляет собой определенную заинтересованным лицом характеристику, отсутствие которой приведет к неполноценности созданного продукта.

5. Осуществимость. Требование может быть реализовано в пределах проекта. Ценность требования сопоставима с затратами на его реализацию.

6. Отслеживаемость. Требование соответствует деловым нуждам заинтересованных лиц, подробно документировано, имеет идентификатор и связи с другими артефактами проекта. По документации и программному коду и комментариям в нем можно отследить все элементы, реализующие данное требование.

7. Корректность, согласованность. Требование не противоречит другим требованиям.

8. Актуальность. Требование не устаревает с течением времени, имеет явную ценность для пользователя.

9. Атомарность. Требование атомарно, то есть оно не может быть разбито на ряд более детальных требований без потери свойства его завершенности.

Если в результате интервью с заказчиком или других методов анализа предметной области разработки был определен некоторый текст, описывающий требования заказчика, его еще нельзя использовать как набор требований, так как чаще всего этот текст, как и любой текст на естественном языке, не удовлетворяет тем свойствам, которые указаны выше. Для формализации требований согласно указанным свойствам, бизнес-аналитик должен провести анализ, который можно разбить на несколько этапов:

1. Описание автоматизируемого бизнес-процесса «как есть». В случае, если заказ заключается в модернизации существующего программного решения, требуется обнаружить его недостатки и выявить те слабые места, которые можно было бы устранить в новой версии. Описание бизнес-процесса обычно проводят либо в текстовой форме, либо в виде диаграммы, напоминающей блок-схему алгоритмов. Главная цель подобного описания – определить последовательность действий, которая должна быть выполнена при реализации бизнес-процесса и назначить роли действующих лиц, несущих ответственность за выполнение отдельных его этапов.

2. Описание автоматизируемого бизнес-процесса «как должно быть». В данном пункте требуется проведение аналогичной деятельности, только важно теперь иметь представление о том, как заказчик видит проведение бизнес-процесса. Сравнение описаний, выполненных на этом и предыдущем этапах, дает ответ на вопрос о том, что заказчик будет считать наиболее важными требованиями.

3. Формирование карточки проекта. В карточке проекта в систематизированном виде обычно описываются текущая ситуация по той деятельности, которую надо изменить, и концепция целевого решения. Особое внимание следует уделить формализации важных для заказчика проблем и тех преимуществ и рисков, которые могут возникнуть от внедрения программного решения в реализацию данного бизнес-процесса.

4. Описание контекста используемого программного решения. Любой программный продукт будет использоваться в определенной

среде (контексте) – важно и аппаратное обеспечение, на котором будет установлено программного обеспечение, его характеристики, и пользователи, которые будут с ним работать, их квалификация, формат используемых ими данных, определение потоков входных данных и их источников, а также потоки выходных данных и их представление. Характеристики контекста следует учитывать при формировании требований к разработанному программному продукту как с точки зрения формулировки функциональных, так и нефункциональных требований. Именно на этом этапе можно определить роли пользователей и основные ограничения, которые они могут испытывать при использовании программного продукта.

5. Формализация функциональных требований. Именно на этом этапе производится формулировка требований в той форме, которая требуется для дальнейшей разработки.

Основная форма описания требований должна укладываться в следующую схему: «[Предмет] должен позволять [Роль] [Глагол] [Объект] [Условие]». Здесь в роли предмета выступает какой-то элемент программного обеспечения (подсистема, модуль, элемент пользовательского интерфейса), т.е. то, к чему должно относиться требование. Роль отражает тип или типы пользователей, которым доступна функциональная опция, реализующая данное требование. Глагол отражает непосредственно действие, выполняемое в рамках функциональной опции над объектом с учетом выполнения указанных в требовании условий, то, что требует автоматизации.

Приведем пример формулировки требования: *Информационная система должна предоставлять зарегистрированному пользователю осуществлять поиск товара в электронном каталоге по названию или производителю.*

Подобная формулировка облегчает поиск информации в требовании о важных элементах требования – кому функциональная опция предоставляется, что позволяет сделать, с какой информацией работает, какой элемент программного решения обеспечивает данный функционал. Становится возможным как «беглое» чтение требований

с целью поиска, так и автоматизация процесса поиска требований по важным элементам описания требований.

Для систематизации списка требований можно группировать требования по подсистемам (реализации определенного функционала) или по роли пользователя, для которого они предназначены. Это также позволит сделать эффективнее дальнейшую работу с требованиями.

Для бизнес-аналитика работа с требованиями не заканчивается их формулировкой. Для дальнейшего облегчения работы проектировщика бизнес-аналитики по формулировке требований могут составлять предварительную модель данных (на основе анализа объектов, которые указаны в формулировках требований), создать словарь терминов (особенно важно для предметных областей со сложной предметной терминологией), а также участвовать в первом этапе разработки проекта программного обеспечения – формулировки функциональной модели программного обеспечения в виде диаграммы вариантов использования и формировании пользовательских сценариев (прецедентов) реализации выделенных вариантов использования.

ЛАБОРАТОРНАЯ РАБОТА. ЭТАП 1 КОМАНДНОГО ПРОЕКТА «ФОРМУЛИРОВКА ТРЕБОВАНИЙ». МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ

Студенты в рамках группы разделяются на команды по 2-3 человека. Состав команды должен быть стабильным, так как проект будет выполняться в течение всего семестра.

Этап 1 состоит из определения предметной области проекта и выполнения работ по формулировке требований к проектируемому программному обеспечению.

Темой проекта может быть любое программное обеспечение, обслуживающее некоторую деятельность. Выбор предметной области проекта остается на усмотрение команды (интернет-магазин, инфор-

мационная система некоторого учреждения, социальная сеть, программное обеспечение, обеспечивающее какой-то удаленный сервис, например, игровое приложение, графический редактор и пр).

Программное обеспечение должно предусматривать участие пользователей с несколькими ролями, обладающими несколькими функциональными опциями.

Результатом этапа 1 должны стать два документа - документ представления предметной области проекта в виде текстового описания и документ со списком функциональных требований, сформулированных согласно принципам работы бизнес-аналитика. Текстовое описание можно рассматривать как конспект интервью с заказчиком и оно может быть оформлено как эссе на естественном языке. Документ с требованиями будет являться исходными данными для последующих этапов.

При выполнении задания роли участников команды могут быть разделены. Предварительно обсудив концепцию, члены команды могут разделиться - кто-то берет ответственность за формирование текстового описания предметной области, другой член команды занимается формулировкой требований. Другой вариант разделения обязанностей может подразумевать, что каждый член команды работает над обоими документами в части описания функциональных опций отдельной(ых) роли(ей).

Для совместной работы над командным проектом создается общее пространство в виде облачной папки проекта с предоставлением полного доступа всех членов команды и преподавателя. В целях контроля версий создаваемых документов можно пользоваться средствами контроля версий, предоставляемыми облачными хранилищами, или договариваться об организации своего протокола работы с документами. Так, можно договориться о принципах именования файлов для раздельной работы студентов. В рамках очных лабораторных занятий можно осуществлять слияние отдельных документов в общий для сдачи этапа путем обсуждения всех версий и выбора лучших реализаций фрагментов документов.

ЯЗЫК МОДЕЛИРОВАНИЯ UML (UNIFIED MODELING LANGUAGE) – УНИВЕРСАЛЬНЫЙ СПОСОБ ОПИСАНИЯ ПРОГРАММНОГО ПРОЕКТА

Унифицированный язык моделирования UML возник как ответ на широкое распространения объектно-ориентированных языков программирования в 80-90-х гг. XX века. В результате начали создаваться методы проектирования, которые позволяли описать объектно-ориентированную программу так, чтобы относительно легко можно было бы переводить программу с одного языка объектно-ориентированного программирования на другой. В начале 90-х гг. выделилось новое поколение методов разработки, среди которого особую популярность приобрели метод Буча, созданный Якобсоном Object-Oriented Software Engineering (OOSE) и разработанный Рамбо Object Modeling Technique (OMT). В 1994 году в рамках компании Rational Software были объединены эти методы, что привело к появлению языка UML в 1996 году. Язык UML версии 2.4.1 принят в качестве международного стандарта ISO/IEC 19505-1, 19505-2.

Программный проект на языке UML представляет собой набор диаграмм, каждая из которых рассматривает проект с отдельной стороны, отражая его отдельные характеристики. Можно сказать, что каждая диаграмма представляет собой отдельный взгляд на весь проект или на его часть. Согласно сути этого «отдельного взгляда» производится основная классификация диаграмм UML. Основными типами диаграмм являются структурные и поведенческие диаграммы. Как и в случае объектно-ориентированной концепции, в которой модель класса представляется структурными (переменные) и поведенческими (методы) свойствами, программный продукт может быть рассмотрен с точки зрения его структуры (что часто называют архитектурой), описывающей отдельные элементы программного обеспечения и их взаимосвязи, и с точки зрения реализации его поведения (функций и реализующих их алгоритмов).

Среди структурных диаграмм можно выделить:

– диаграммы классов (class diagrams) предназначены для моделирования архитектуры объектно-ориентированной программы, здесь основными элементами являются классы с описанием их состава (атрибутов-переменных и методов), а также взаимосвязей между объектами и классами;

– диаграммы компонент (component diagrams) используются при моделировании компонентной структуры распределенных приложений, использующих возможно для работы нескольких устройств, каждая компонента представляет собой отдельный артефакт (приложение, модуль, сценарий и пр.), каждый из которых может иметь сложную внутреннюю структуру;

– диаграммы объектов (object diagrams) применяются для моделирования фрагментов работающей системы, отображая реально существующие в режиме выполнения экземпляры классов, значения их атрибутов и принципы их формирования и вычисления;

– диаграммы развертывания (deployment diagrams) предназначены для моделирования аппаратной части системы, с которой программное обеспечение непосредственно связано (например, размещено или взаимодействует);

– диаграммы пакетов (package diagrams) служат для разбиения объемных моделей на составные части (например, подсистемы), удобны для представления фрагментов диаграммы классов, которые содержатся в отдельном артефакте программного обеспечения.

Среди поведенческих диаграмм можно выделить:

– диаграммы активностей или деятельности (activity diagrams) используются для спецификации бизнес-процессов, которые должно автоматизировать разрабатываемое программное обеспечения, а также для описания сложных алгоритмов;

– диаграммы вариантов использования (use case diagrams) предназначены для визуализации функциональной структуры разрабатываемого программного обеспечения в виде систематизации требова-

ний к программному обеспечению путем их представления в виде взаимосвязанных функциональных опций (вариантов использования), предоставляемых отдельным ролям пользователей;

– диаграммы конечных автоматов (state machine diagram) применяются для отслеживания жизненного цикла объекта класса и принципов изменения реакции объекта на различные события;

– диаграммы последовательностей (sequence diagram) используются для моделирования временных аспектов взаимодействия отдельных объектов для реализации определенного функционала и принципов передачи управления между участвующими в реализации функционала объектами.

Указанный список включает набор диаграмм, который является стандартом и поддерживается всеми редакторами UML-диаграмм. Тем не менее, в различных спецификациях существуют и другие виды диаграмм, раскрывающих более детальные аспекты представления программного обеспечения.

Другая классификация диаграмм базируется на том, какой уровень детализации рассматривается в диаграмме. Разделяют диаграммы логического уровня (описывающее программное обеспечение в терминах функционала, реализованных алгоритмов, используемых структур данных) и диаграммы физического уровня (описывающее программное обеспечение в терминах артефактов и способов взаимодействия с аппаратным обеспечением). К диаграммам логического уровня относят диаграммы вариантов использования, классов, последовательностей, активностей, конечного автомата. К диаграммам физического уровня относятся диаграммы компонентов и развертывания.

Для формирования диаграмм UML можно использовать как обычные графические редакторы, так и специализированные редакторы, среди которых можно выделить:

– Microsoft Visio – графический редактор, входящий в состав Microsoft Office, содержит все основные средства для создания UML-диаграмм (web-версия доступна при наличии лицензии Microsoft 365 https://www.microsoft.com/ru-ru/microsoft-365/visio/flowchart-software);

– StarUML (https://staruml.io/) – условно свободный редактор, специализирующийся именно на создании UML-диаграмм;

– Visual Paradigm (https://online.visual-paradigm.com/) – бесплатная версия имеет некоторые ограничения по возможностям интерфейса, но для учебных проектов инструмент достаточно удобный;

– Lucidchart (https://www.lucidchart.com) – имеет возможность бесплатного аккаунта, но с ограничениями на количество и типы используемых диаграмм;

– Google Draw.io – бесплатный редактор в рамках сервисов, предоставляемых Google Disk.

Большая часть данных инструментов является графическим редакторами. Их удобно использовать для быстрых набросков диаграмм, хотя их последующее редактирование в случае создания больших диаграмм, в которых следует располагать большое количество элементов, которые могут много раз редактироваться и перегруппировываться в рамках построенного изображения, может быть значительно затруднено и соответственно не удобно.

Как способ устранить подобное неудобство был создан специализированный open-source-проект, называемый PlantUML, представляющий собой декларативный язык описания UML-диаграмм (и не только их), который позволяет генерировать изображение по созданному текстовому описанию. Для создания UML- диаграмм на основе данного языка имеются бесплатные online-редакторы, например:

– https://www.planttext.com/ ,
– https://plantuml-editor.kkeisuke.com/ ,
– http://www.plantuml.com/plantuml/uml.

Интерфейс указанных web-приложений аналогичен. Например, на интерфейс ресурса https://plantuml-editor.kkeisuke.com/ представлен на рис. 1. В рамках интерфейса представлено окно для ввода кода описания uml-диаграммы, на основе которого рядом генерируется изображение диаграммы, которую можно сохранить в графическом формате и скачать в виде, например, файла формата png.

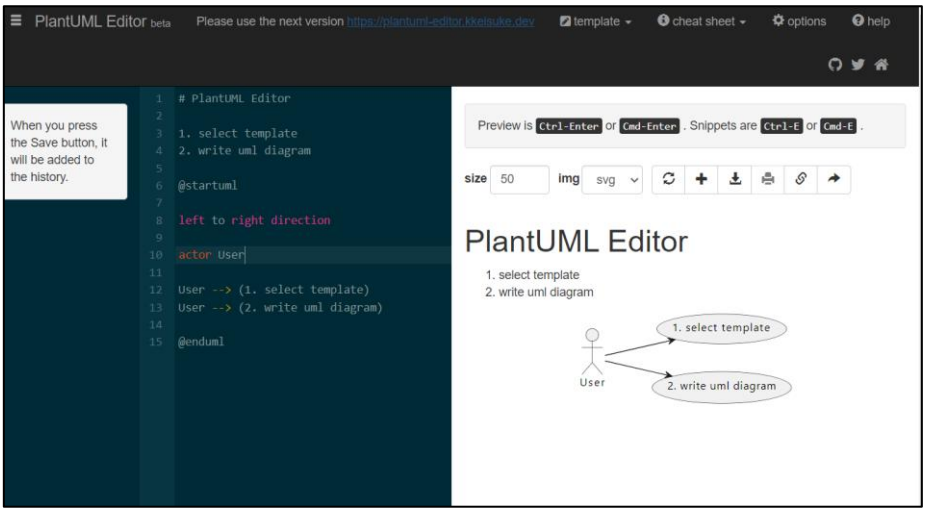

Рис. 1. Пользовательский интерфейс ресурса https://plantuml-editor.kkeisuke.com/

ДИАГРАММА ВАРИАНТОВ ИСПОЛЬЗОВАНИЯ

Диаграмма вариантов использования является первой диаграммой, создаваемой в рамках проектирования программного обеспечения и призвана показать функциональную структуру приложения, систематизировать функциональные опции, которые предоставляются разным ролям пользователей. Диаграмму вариантов использования могут использовать и бизнес-аналитики, в том числе, для показа заказчику, чтобы уточнить состав требований и конкретизировать техническое задание.

Основными элементами диаграммы вариантов использования являются акторы (действующие лица) и варианты использования. Актором является любая внешняя по отношению к проектируемой системе сущность, которая взаимодействует с программной системой и использует ее функциональные возможности для достижения определенных целей. Изображается на диаграмме в виде человечка. Актором может быть пользователь с определенной ролью, но может быть и другая информационная система, которая является внешним ресурсом и которая может инициировать действия в разрабатываемой системе.

Вариант использования представляет собой спецификацию некоторого целевого действия, выполняемого системой, по инициативе одного или нескольких видов акторов. Целевое действие связано с получением явного видимого результата, по которому можно понять, что вариант использования был выполнен. Изображается на диаграмме в виде овала, в котором в виде глагола или отглагольного существительного указывается целевое действие варианта использования.

Простой пример диаграммы вариантов использования можно увидеть на рис. 2. На ней актор «Пользователь» имеет два варианта использования «Просмотр каталога товаров» и «Покупка товаров».

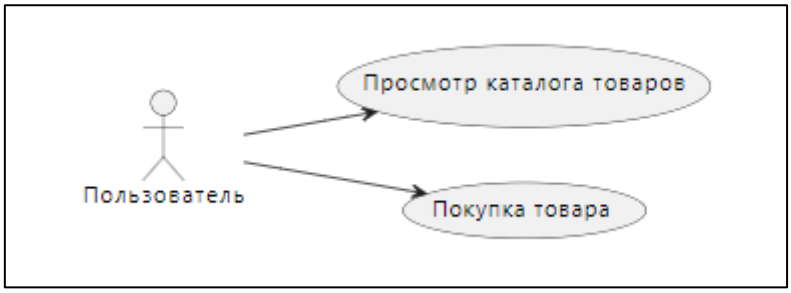

Рис. 2. Пример диаграммы вариантов использования

Между элементами диаграммы вариантов использования существуют взаимосвязи нескольких видов:

– Ассоциация – обозначает связь между актором и вариантом использования. Это наиболее частая связь, присутствующая на диаграмме вариантов использования. Указывается стрелкой от инициирующего актора к варианту использования;

– Включение – связь между вариантами использования, определяющая, что один вариант использования обязательно вызывается и используется в другом варианте использования. Обозначается надписью <<include>> над стрелкой, обозначающей связь от включающего варианта использования к включенному. Например, такое вариант использования «Регистрация» может включать вариант использования «Подтверждение номера телефона» как обязательный элемент. Такое разделение имеет смысл производить, если вариант использования «Подтверждение номера телефона» может быть использован повторно в других вариантах использования;

– Расширение – связь между вариантами использования, определяющая, что один вариант использования может быть расширен выполнением другого варианта использования при выполнении определенных условий, т.е. расширение в отличие от включения не является обязательным. Обозначается надписью <<extend>> над стрелкой, обозначающей связь от расширяющего варианта использования к расширяемому. Примером может являться расширение варианта использования «Оплата заказа» вариантом использования «Предоставление скидки», который может использоваться, например, при условии, если сумма заказа больше некоторой заданной или если оплата производится определенным образом, например, через банк-партнер;

– Обобщение – отношение между однородными элементами (двумя акторами или двумя вариантами использования). Фактически обобщение является своеобразным наследованием, обозначающим, что, например, один актор является частным случаем другого. Обозначается с помощью стрелки с незакрашенным наконечником, направленным в сторону более общего элемента диаграммы. Примерами авторов, находящихся в отношении обобщения, могут быть «Сотрудник» и «Менеджер» – менеджер является частным случаем сотрудника и ему доступны все варианты использования, доступные сотруднику, плюс свои дополнительные варианты использования. Между вариантами использования тоже может быть отношение обобщения – например, вариант использования «Оплата заказа» может иметь частный случай «Оплата заказа по безналичному расчету».

Каждый вариант использования, фактически, соответствует одному или нескольким функциональным требованиям. Для конкретизации вариант использования дополняется пользовательским сценарием (прецедентом) – описанием типичных действий пользователя, соответствующих этому варианту использования, при условии его успешного совершения, т.е. выполнению целевого действия, связанного с этим вариантом использования.

Прецедент описывается в алгоритмическом виде как последовательность действий акторов и системы и передачи управления между

пунктами. Кроме алгоритма действий, прецедент содержит описание выполняющего актора, предусловия (условия запуска прецедента), постусловия (условия, определяющие успешное завершение прецедента), описание альтернативных потоков, соответствующих ошибочным действиям пользователя и реакции системы на них.

Приведем пример прецедента. В данном прецеденте пункт 4 вызывает передачу управления на альтернативные потоки. Этот прием напоминает использование механизма обработки исключительных ситуаций в программировании – главная последовательность прерывается, если какой-то пункт требует перехода на альтернативный поток. Каждый альтернативный поток связан с определенным условием, при выполнении которого невозможно выполнение главной последовательности. Так, все альтернативные потоки, указанные в примере прецедента, связаны с проверками пункта 4 и в зависимости от условия, которое было нарушено (указывается как условие передачи управления альтернативному потоку), запускается тот или иной альтернативный поток. Альтернативные потоки могут быть привязаны к различным пунктам главной последовательности – тогда следует явно указать на номер альтернативного потока, к которому следует осуществить переход в пункте главной последовательности.

Также следует обратить внимание на обязательные реквизиты описания прецедента. Название прецедента должно соответствовать названию варианта использования на диаграмме вариантов использования. Действующие лица соответствуют акторам диаграммы варианты использования. Если вариант использования доступен нескольким акторам, можно описывать его единообразно, указав всех возможных акторов в реквизите прецедента, но в случае, когда существуют отличия в реализации прецедента для разных видов акторов, каждая реализация прецедента описывается отдельно. Цель или постусловие формулируется в виде условий, которые можно проверить для того, чтобы убедиться, что целевое действие прецедента было успешно выполнено. Очень важно сформулировать постусловие таким образом, чтобы тестировщики впоследствии по данному описанию прецедента

смогли легко сформулировать тестовую процедуру для проверки того, что функциональное требование, реализуемое данным прецедентом, было успешно выполнено. Предусловия описывают условия, которые должны быть выполнены системой или актором, чтобы прецедент мог быть выполнен. В случае невыполнения предусловий главная последовательность прецедента даже не должна запускаться на выполнение. Это также следует отслеживать тестировщикам при формировании тестовых процедур для проверки данного прецедента.

Название прецедента: *регистрация*

Действующее лицо: *незарегистрированный пользователь*

Цель (Постусловие): *сохранение информации о пользователе в базе данных*

Предусловия: *отсутствуют*

Главная последовательность:

1. *Пользователь нажимает на кнопку «Регистрация»;*
2. *Система показывает пользователю окно регистрации, в котором следует указать логин пользователь, пароль, повторить пароль в отдельном поле редактирования и другие необходимые поля ввода (определяются карточкой пользователя);*
3. *Пользователь вводит информацию в форму регистрации и нажимает кнопку «Зарегистрироваться»;*
4. *Система проверяет корректность введенной информации, в случае некорректной информации переход к альтернативной последовательности;*
5. *Система сохраняет корректные данные о пользователе в базе данных;*
6. *Система выводит пользователю сообщение об успешной регистрации в системе.*

Альтернативная последовательность *(введены некорректные данные пользователя – логин уже занят в системе):*

1. *Система выводит сообщение о том, что указанный логин уже занят, и требуется придумать другой.*

2. Система указывает пользователю о необходимости изменения логина и повторного нажатия кнопки «Зарегистрироваться».

***Альтернативная последовательность** (введены некорректные данные пользователя – пароль не удалось подтвердить):*

1. Система выводит сообщение о том, что введенный пароль в поле подтверждения пароля не совпадает с указанным ранее.

2. Система указывает пользователю о необходимости повторного ввода пароля и нажатия кнопки «Зарегистрироваться».

***Альтернативная последовательность** (введены некорректные данные пользователя – пароль не удовлетворяет требованиям безопасности):*

1. Система выводит сообщение о том, что введенный пароль не удовлетворяет требованиям безопасности.

2. Система указывает пользователю о необходимости повторного ввода пароля и нажатия кнопки «Зарегистрироваться».

Диаграмму вариантов использования часто используют как один из инструментов анализа требований и показа проекта бизнес-аналитику и заказчику. Очевидность и малое разнообразие элементов диаграммы делает ее интуитивно понятной, поэтому заказчик может наглядно представить функционал будущего приложения и соответствие своего взгляда взглядам проектировщика. Пользовательские истории в силу того, что они описываются на естественном языке, также доступны для понимания заказчику. Поэтому этот этап необходимо выполнять очень тщательно, чтобы максимально убедиться, что заказчик удовлетворен функциональной моделью приложения, выполненной на этом этапе.

Также этот этап важен для работы тестировщиков. Прецеденты являются исходными данными для формирования тестовых процедур функциональных и приемосдаточных тестов. Поэтому тестировщики уже после выполнения этого этапа проектирования могут начать ра-

боту по созданию тестового набора для функционального тестирования, создавая тестовые процедуры на основании предполагаемых действий пользователя.

Диаграмма вариантов использования строится одна на весь проект и показывает весь функционал, который должен быть в приложении. Очевидно, что для сложных приложений диаграмма может быть достаточно громоздкой. Для удобного представления ее можно разделить согласно разным подсистемам или разным типам пользователей. Но тогда необходимо отслеживать, чтобы одни и те же элементы на разных частях диаграммы были названы одинаково.

Разберем теперь правила создания диаграммы вариантов использования на PlantUML.

Любое описание uml -иаграммы описывается в декларирующих «скобках», записываемых с помощью ключевых слов:

```
@startuml
...
@enduml
```

Прецеденты в описании заключаются в круглые скобки. Можно также использовать ключевое слово usecase для предварительной декларации прецедента и повторного использования по псевдониму, который можно указать после указания имени прецедента, используя конструкцию as keyword.

Определить актора можно, заключив его значение между двоеточиями. Также можно использовать ключевое слово actor для предварительной декларации актора. Актору также можно задать псевдоним.

Для соединения акторов и прецедентов, используется стрелка «-->». Чем большее количество символа тире используется в стрелке, тем она будет длиннее. Можно добавить метку на стрелку, добавив символ «:» при определении стрелки.

Пунктирную линию по необходимости можно провести с помощью символов «..». Отношение обобщения между акторами и вариантами использования обозначается «<|—».

На следующей диаграмме вариантов использования (рис. 3) демонстрируется использование нескольких вариантов использования с отношением включения и расширения. Также демонстрируются визуальные приемы, которые позволяют объединить несколько вариантов использования в подсистему (объединение в прямоугольную область) и использование комментариев, которые можно присоединить к любому элементу диаграммы.

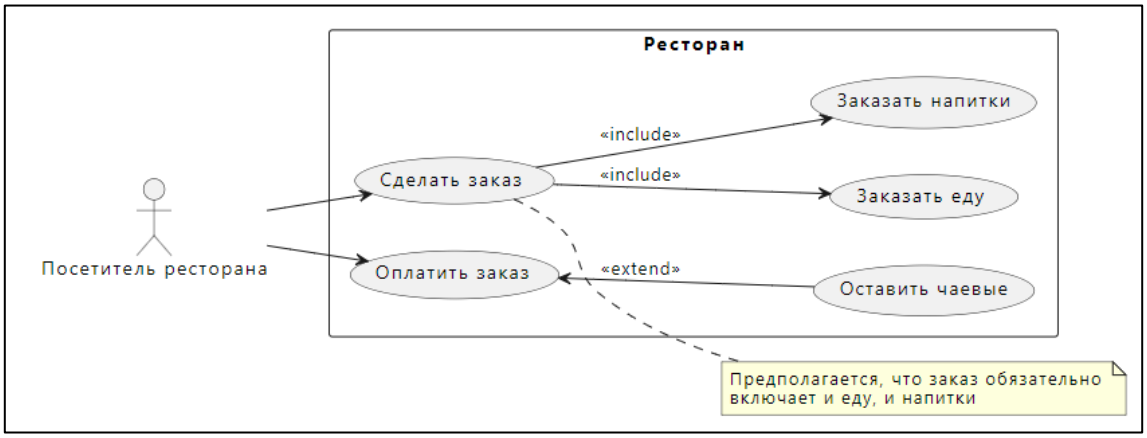

Рис. 3. Диаграмма вариантов использования для посетителя ресторана

Программный код, описывающий данную диаграмму вариантов использования, выглядит следующим образом:

```
@startuml
left to right direction

actor "Посетитель ресторана" as client
rectangle Ресторан {
usecase "Сделать заказ" as UC1
usecase "Заказать еду" as UC2
usecase "Заказать напитки" as UC3
usecase "Оплатить заказ" as UC4
usecase "Оставить чаевые" as UC5
}
client --> UC1
UC1 --> UC2 :<<include>>
UC1 --> UC3 :<<include>>
client --> UC4
UC4 <-- UC5 :<<extend>>
note right of UC1
```

```
Предполагается, что заказ обязательно
включает и еду, и напитки
end note
@enduml
```

При описании элементов диаграммы можно установить стилевые установки в виде указания цветовых и иных характеристик.

Общий формат определения стилевых характеристик стрелок:

```
#color;line.[bold|dashed|dotted];text:color
```

Общий формат определения стилевых характеристик акторов или прецедентов:

```
#[color|back:color];line:color;line.[bold|dashed|dotted];text:color
```

ЛАБОРАТОРНАЯ РАБОТА. ЭТАП 2 КОМАНДНОГО ПРОЕКТА «СОЗДАНИЕ ДИАГРАММЫ ВАРИАНТОВ ИСПОЛЬЗОВАНИЯ». МЕТОДИЧЕСКИЕ УКАЗАНИЯ

На основании разработанных на этапе 1 требований к программному обеспечению, указанных в них модулей, акторов и функциональных опций, необходимо создать диаграмму вариантов использования и описание прецедентов (8 и более самых важных по проекту).

При выполнении задания команде рекомендуется совместно обсудить основной принцип построения диаграммы вариантов использования, возможную декомпозицию диаграммы для лучшего визуального восприятия, принцип описания прецедента, после чего можно разделить части диаграммы (варианты использования отдельных акторов или подсистем) и прецеденты между членами команды и синхронизировать выполненную работу к сроку сдачи.

Следует обратить внимание на единообразие именования вариантов использования и синхронизацию используемой терминологии на диаграмме вариантов использования и в описания прецедентов, особенно в том случае, если диаграмма вариантов использования будет декомпозицирована на несколько диаграмм.

Также следует обратить внимание не то, что диаграмма должна соответствовать сформулированным на этапе 1 функциональным требованиям. Так, роли пользователей при формулировке требований должны соответствовать акторам на диаграмме использования, каждому выделенному требованию должен соответствовать вариант использования на диаграмме.

ДИАГРАММА КЛАССОВ

Диаграмма классов – следующий уровень конкретизации после диаграммы вариантов использования. Возможно, это самая важная диаграмма, которая задает архитектуру программной системы, ее внутренний состав и взаимосвязи.

Диаграмма классов (class diagram) – диаграмма, предназначенная для представления модели статической структуры программной системы в терминологии классов объектно-ориентированного программирования. Диаграмма классов представляет собой граф, вершинами или узлами которого являются классы, интерфейсы или перечисления. Взаимосвязи в графе определяют отношения между классами или объектами классов в зависимости от типа взаимосвязи.

С точки зрения объектно-ориентированного подхода любой класс имеет ряд характеристик:

– структурные характеристики (атрибуты), которые показывают, из чего класс состоит (переменные класса с точки зрения объектно-ориентированного программирования);

– поведенческие характеристики (операции, методы), которые показывают, что можно делать с объектом или что может делать сам объект со своими или полученными данными.

Каждый класс отображается на диаграмме с помощью прямоугольника, разделенного на 3 горизонтальные части – для описания имени класса, его атрибутов и методов соответственно.

Атрибуты и методы могут быть указаны в краткой форме, либо с подробным описанием их свойств.

Форма записи атрибута имеет следующий вид:

```
<атрибут>::=[<видимость>] [/] <имя> [: <тип атрибута>] [[<кратность>]] [= <значение по умолчанию>] [{<список модификаторов атрибута>*}]
```

В записи атрибута обязательным является только его имя. Видимость атрибута означает уровень доступа к атрибуту извне класса. Видимость соответствует основным режимам доступа в объектно-ориентированном программировании, обозначается для краткости символами – «+» (public), «-» (private), «#» (protected), «~» (package, internal). Последний режим доступа специфичен для разных языков программирования и означает видимость элемента класса в пределах модуля, пакета, пространства имен – некоторого объединения программных элементов в рамках приложения.

Использование символа «/» означает, что атрибут является производным, то есть является избыточным и может быть получен из значений других атрибутов. Кратность атрибута указывается в квадратных скобках и характеризует общее количество конкретных значений для атрибута, которые могут быть заданы для объектов данного класса.

Модификаторы атрибуты – это различные его важные свойства, которые важны для понимания его использования. Модификаторы указываются в фигурных скобках через запятую. Среди популярных видов модификаторов атрибутов можно выделить модификатор «только для чтения» (readOnly), переопределение унаследованного атрибута (redefines <имя атрибута>), хранение множественного атрибута в отсортированном виде (ordered), уникальность значения атрибута

среди всех объектов класса (unique). Также можно вводить свои модификаторы атрибута, задающие некоторые условия, которым должен удовлетворять атрибут.

Методы также имеют форму описания, конкретизирующую применение метода в программе:

```
<метод>::=[<видимость>] <метод> ( [<список параметров>] ) [: [<тип возвращаемого результата>] [{<список свойств метода>}
```

Основным элементом формы записи метода является его прототип. Параметры метода также требуют описания, которое осуществляется по следующей форме:

```
<параметр>::=[<направление>] <имя параметра>: <выражение типа> [[<кратность>]] [=<значение по умолчанию>] [{<список свойств параметра>}]
```

Основным свойством параметра является его направление. Направление задает способ использования параметра. Возможны следующие значения:

– in (по умолчанию)– значения этого параметра являются входными данными для метода;

– inout – значения такого параметра используются как входные данные метода, но могут быть изменены внутри метода с сохранением сделанных изменений и передачей новых значений в место вызова метода;

– out – такие параметры используются для возвращения результатов выполнения методов;

– return – используется для конкретизации возвращаемого значения метода.

Кроме прототипа метода могут быть указаны дополнительные свойства методов, например, переопределение родительского метода (redefines <имя метода>), запрет метода изменять состояние объекта (изменять значения атрибутов объекта) (query) и т.д.

Помимо состава используемых классов на диаграмме должны быть указаны взаимосвязи между классами или объектами классов, которые могут возникнуть в приложении. Отношение между классами формулируется на основе отношений между типами данных. Отношения между объектами означает, что в него вступают отдельные объекты классов. Имеются следующие виды взаимосвязей:

– Ассоциация - произвольное отношение или взаимосвязь между объектами двух или более классов. Ассоциация определяет логическую связь, которую обычно можно выразить глаголом или существительным, которое характеризует роль объекта класса в этом отношении. Ассоциация обозначается сплошной соединительной линией или сплошной линией с направленной стрелкой для обозначения активного объекта, от которого исходит инициация ассоциации. Также на концах линии ассоциации обычно указывают кратность объектов ассоциации. Ассоциации часто возникают между объектами, хранящими данными, и объектами, которые осуществляют их обработку. В этом случае линия ассоциации может быть отображена стрелкой, ведущей от класса обработки к классу данных.

– Обобщение – отношения между классами, соответствующие принципу наследования (“is a” – «является», “is like a” – «является похожим»). Наиболее понятное с точки зрения объектно-ориентированного программирования. Обобщение отображается с помощью сплошной линии с треугольным незакрашенным наконечником на стороне родительского класса.

– Агрегация (“is part of”) – направленное отношение между объектами двух классов, предназначенное для представления отношения «часть-целое», то есть показывающее, что объекты одного из классов являются частями объекта другого класса. Следует обратить внимание, что объект-часть может существовать и после уничтожения объекта-целого. Отображается сплошной линией с наконечником в виде незакрашенного ромба к объекта-целого.

– Композиция (частный случай агрегации) предназначена для спецификации более сильной формы отношения «часть-целое», при

которой с уничтожением объекта класса-целого уничтожаются и все объекты, являющимися его составными частями. Отображается аналогично агрегации с закрашенным ромбом в качестве наконечника.

– Зависимость – самый неформализованный вид отношения между классами. Обозначается на диаграмме пунктирной линией. Зависимости помещают на диаграмму в редких случаях и прежде всего, чтобы подчеркнуть ее важность для дальнейшей разработки проекта. Зависимость характеризует возможные изменения в одном из классов при модификации в другом. Она может возникнуть, если объекты двух классов возникают в одном и том же фрагменте программного кода, например, объект одного класса передается в метод другого класса в качестве параметра или является его возвращаемым значением, или в одном методе используются объекты обоих классов.

Разберем правила создания диаграммы классов на PlantUML.

Описание каждого класса похоже на его описание в языке программирования. Можно также создавать абстрактные классы, интерфейсы, перечисления. Дополнительно можно использовать описатели {static} или {abstract}, чтобы показать то, что переменная или метод были статическими или метод был абстрактным. Каждый элемент класса может иметь видимость, которую можно указать ключевыми словами (public, private и т.д.) или соответствующими им в UML символами (+, - и т.д.).

Имя класса в дальнейшем является его идентификатором и позднее может быть использован для описания разных видов отношений. Связь при наследовании организуется с помощью последовательности символов «<|--», агрегация – «o-- », композиция – «*--», ассоциация – «--», зависимость – «..». Кратность указывается на соответствующем конце соединительной линии в виде строки. Также связь может быть дополнена надписью над линией связи.

В качестве примера рассмотрим фрагмент диаграммы классов для информационной системы ресторана (рис.4).

На диаграмме рассмотрены классы, которые включаются в порядок формирования заказа посетителем. Актор, который участвует в процессе здесь единственный. Для него сформировано два класса – класс данных Пользователь и класс пользовательского интерфейса ОкноПользователя. Такое разделение соответствует разделению обязанностей в архитектурах наподобие архитектуры MVC (Model View Controller). Здесь класс Пользователь относится к подсистеме модели, отвечает за обеспечение доступа к данным и синхронизацию информации с базой данных. Аналогично, классами подсистемы модели являются классы Товар, Блюдо и Напиток. Следует обратить внимание, что последние три класса образуют иерархию наследования, т.е. вступают в отношение обобщения. Класс ОкноПользователя является классом из подсистемы представления (View). Оно включает объекты классов Пользователь и Заказ с помощью отношения агрегации (имеет соответствующие поля) и реализует варианты использования актора Пользователь (часть из них можно наблюдать на диаграмме вариантов использования на рис.3). Так как для формирования заказа объекту класса ОкноПользователя требуется информация о товаре, между классами ОкноПользователя и Товар существует отношение зависимости. В принципе, можно было бы использовать отношение ассоциации, однако выбор чаще всего зависит от принципов реализации данного отношения. Также отношение агрегации существует между классами Заказ и Товар – в объект класса Заказ как его часть внедрен список из ссылок на товары, в который могут входить как блюда, так и напитки из-за реализованного отношения обобщения в иерархии товаров.

Программный код, описывающий данную диаграмму классов, выглядит следующим образом:

```
@startuml
class Пользователь
{
  - Имя: string
  - Логин: string
  - ХэшПароля: string
```

```
    - ДанныеКарты: string
    + Пользователь(имя, логин, пароль)
    + ПроверкаУчетныхДанных(логин, пароль): bool
    + СохранитьвБД()
    + ЗагрузитьизБД()
  }
```

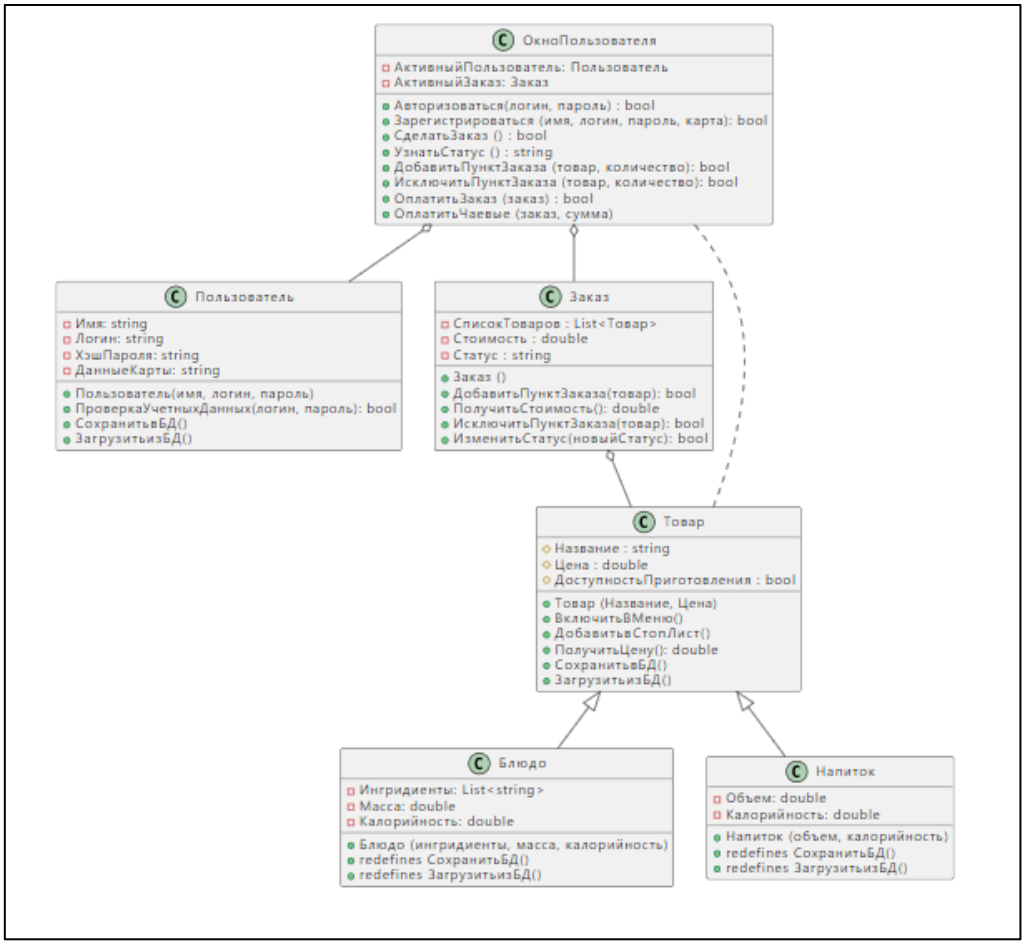

Рис. 4. Диаграмма классов для информационной системы ресторана

```
  class ОкноПользователя
  {
    - АктивныйПользователь: Пользователь
    - АктивныйЗаказ: Заказ
    + Авторизоваться(логин, пароль) : bool
    + Зарегистрироваться (имя, логин, пароль,
карта): bool
    + СделатьЗаказ () : bool
    + УзнатьСтатус () : string
    + ДобавитьПунктЗаказа (товар, количество): bool
    + ИсключитьПунктЗаказа (товар, количество):
bool
```

```
  + ОплатитьЗаказ (заказ) : bool
  + ОплатитьЧаевые (заказ, сумма)
}
class Заказ {
  - СписокТоваров : List<Товар>
  - Стоимость : double
  - Статус : string
  + Заказ ()
  + ДобавитьПунктЗаказа(товар): bool
  + ПолучитьСтоимость(): double
  + ИсключитьПунктЗаказа(товар): bool
  + ИзменитьСтатус(новыйСтатус): bool
}
class Товар
{
  # Название : string
  # Цена : double
  # ДоступностьПриготовления : bool
  + Товар (Название, Цена)
  + ВключитьВМеню()
  + ДобавитьвСтопЛист()
  + ПолучитьЦену(): double
  + СохранитьвБД()
  + ЗагрузитьизБД()
}
class Блюдо
{
  - Ингридиенты: List<string>
  - Масса: double
  - Калорийность: double
  + Блюдо (ингридиенты, масса, калорийность)
  + redefines СохранитьБД()
  + redefines ЗагрузитьизБД()
}
class Напиток
{
  - Объем: double
  - Калорийность: double
  + Напиток (объем, калорийность)
  + redefines СохранитьБД()
  + redefines ЗагрузитьизБД()
}

Товар <|-- Блюдо
Товар <|-- Напиток
```

```
    Заказ o-- Товар
    ОкноПользователя o-- Пользователь
    ОкноПользователя o-- Заказ
    ОкноПользователя .. Товар
@enduml
```

ЛАБОРАТОРНАЯ РАБОТА. ЭТАП 3 КОМАНДНОГО ПРОЕКТА «СОЗДАНИЕ ДИАГРАММЫ КЛАССОВ». МЕТОДИЧЕСКИЕ УКАЗАНИЯ

На основании диаграммы вариантов использования и описанных прецедентов следует разработать первоначальный вариант диаграммы классов. Впоследствии диаграмма классов может уточняться и изменяться в связи с получение новой дополнительной информации.

На первом варианте диаграммы требуется определить основную архитектуру проекта, основные классы, интерфейсы, состав переменных классов и основных методов. Обратите внимание, что очень важно всех акторов на диаграммы вариантов использования перенести в классы, а варианты использования – в методы классов-акторов или классов обеспечения их интерфейса.

Кроме того, могут быть явно введены и детализированы классы с данными (например, в описании прецедентов есть упоминание этих данных) или классы для обеспечения функционирования (специализированные классы для хранения, обеспечения пользовательского интерфейса, обработки данных).

Следует понимать, что диаграмма классов одна, но она также как диаграмма вариантов использования может быть декомпозицирована на несколько диаграмм по подсистемам классов, выполняющих определенную функцию или близкий по смыслу набор функций.

Рекомендуется вести документ, в котором описываются свойства атрибутов и методов классов с учетом их специфики и характеристик, чтобы не загромождать основную диаграмму ключевыми словами и обозначениями.

Необходимо обратить внимание на обеспечение синхронизации диаграммы классов с диаграммой вариантов использования с точки зрения используемых имен – каждый элемент диаграммы вариантов использования должен быть отображен на диаграмме классов.

При выполнении задания команде рекомендуется совместно обсудить основной принцип построения диаграммы классов, а непосредственное выполнение деталей различных подсистем распределить между членами команды и синхронизировать выполненную работу к сроку сдачи.

ИСПОЛЬЗОВАНИЕ ШАБЛОНОВ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОЕКТИРОВАНИЯ

Шаблоны (паттерны) объектно-ориентированного проектирования – это набор образцов, по которому строится программная система в некоторых ситуациях. Использование шаблонов проектирования облегчает чтение и модификацию программных систем при необходимости новых версий. Кроме того, использование шаблонов позволяет документировать опыт разработки объектно-ориентированных программ и упрощать повторное использование архитектурных решений в похожих проектах.

Существует общепринятая система объектно-ориентированных паттернов, применение которых стало стандартом при разработке объектно-ориентированных программ.

Все шаблоны проектирования разделены на три группы: порождающие, структурные и поведенческие паттерны. Порождающие паттерны предназначены для гибкого создания объектов классов. Структурные паттерны предназначены для возможности гибкой смены

структуры используемых классов, отвечают за наполнение и взаимодействие подсистем. Поведенческие паттерны предназначены для гибкого описания поведения объектов. Всего шаблонов объектно-ориентированного проектирования более двадцати. Они носят универсальный характер. Тем не менее, в некоторых технологиях и языках программирования имеются и свои специализированные шаблоны проектирования.

Опишем несколько наиболее популярных в проектировании паттернов, принадлежащих различным типам.

1. **Фабричный метод (Factory Method).** Данный шаблон проектирования определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой объект какого класса создать. Данный паттерн является порождающим и часто в литературе называется также виртуальным конструктором, так как фактически является его заменой в клиентской подсистеме. Схема взаимодействия участвующих при использовании паттерна «Фабричный метод» классов представлена на рис. 5.

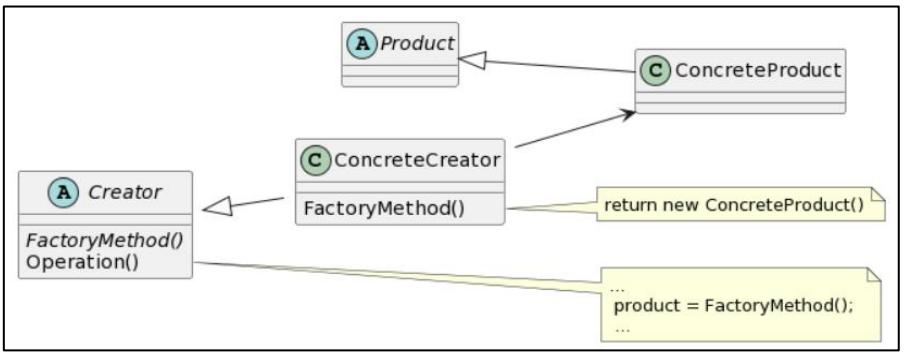

Рис. 5. Схема использования паттерна «Фабричный метод»

Абстрактный класс Product здесь определяет интерфейс объектов, создаваемых фабричным методом – любой его наследник может быть создан фабричным методом. Таким наследником на схеме является класс ConcreteProduct. Абстрактный класс Creator предоставляет интерфейс, доступный клиентским приложениям. В его состав входит, в частности, фабричный метод FactoryMethod(). На данной схеме от него наследует класс, в котором определенная конкретная реализация

фабричного метода, которая внутри себя по значениям параметра создает объект из иерархии и возвращает его как конструктор, т.е. фабричный метод инкапсулирует внутри себя конструктор. Прототип фабричного метода должен возвращать объект базового абстрактного типа Product. В зависимости от параметров фабричного метода или других характеристик (например, состояния объекта-создателя) происходит передача управления тому подклассу класс-создателя, который может создать нужный объект-наследник класса Product.

В более простых случаях не требуется создавать иерархию классов, имеющих и переопределяющих фабричный метод. Более того, фабричный метод можно поместить в класс Product как статический метод, который можно вызвать без создания объекта, что вполне возможно для абстрактного класса. Тогда вся логика создания объектов инкапсулируется в нем.

Применение данного шаблона проектирования позволяет осуществить гибкое создание объектов, инкапсулируя в одном месте знание о существующих типах, соответственно, давая возможность расширять спектр типов, которые будет создавать фабричный метод.

2. **Одиночка (Singleton)**. Это порождающий паттерн, применение которого гарантирует, что у класса будет только один объект, и предоставляет к этому единственному объекту точку доступа. Схем реализации представлена на рис. 6.

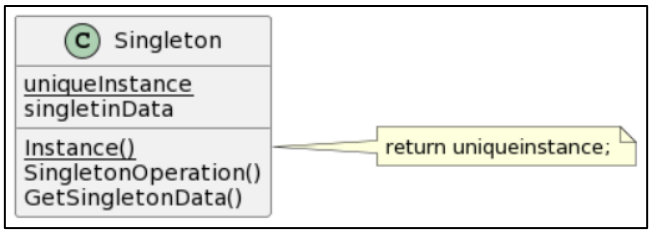

Рис. 6. Схема использования паттерна «Одиночка»

Все манипуляции осуществляются с одним классом. Ссылка на единственный экземпляр класса формируется как статическое поле класса Singleton. Для создания и доступа к этому единственному объекту в класс Singleton добавляется статический метод Instance(). Метод

создает единственный объект при первом обращении к методу и возвращает ссылку на этот единственный объект при любом обращении к методу. Этот метод является единственным способом доступа извне класса к объекту.

Типичным примером использования этого паттерна являются создание затратных объектов, определяющих ресурсы, которые требуются в разных частях приложения. Например, таким ресурсом может быть соединение с базой данных при частых обращениях к ней.

3. **Адаптер (Adapter).** Паттерн «Адаптер» (другое название паттерна «Обертка») является способом преобразования программного интерфейса в другой, более удобный для использования. Например, такая задача часто стоит в случаях использования в программном обеспечении прикладного уровня некоторых сторонних библиотек для решения системных задач или написанных на другой технологии программирования, неудобной для места использования. Также часто используется в клиент-серверных технологиях для общения клиентской и серверной части, которые могут быть написаны с помощью разных технологий разработки и использующих сложный протокол обмена. В этом случае адаптер занимается своеобразным переводом с языка одной стороны взаимодействия на язык другой. Схема использования паттерна «Адаптер» представлена на рис. 7.

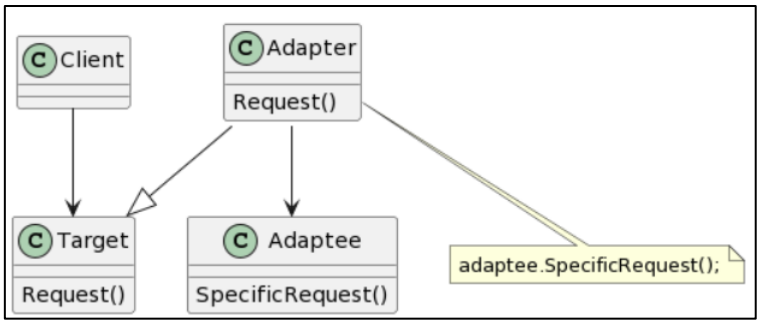

Рис. 7. Схема использования паттерна «Адаптер»

В данной схеме клиентскому приложению (класс Client) требуется некоторый функционал (класс Target метод Request()). Программ-

ный интерфейс метода зависит от предметной области решаемой задачи и сформулирован в терминах технологии разработки прикладного уровня. Непосредственно этот функционал удобно выполнить, например, с помощью ранее разработанного или библиотечного класса Adaptee, в котором данную задачу решает метод SpecificRequest(), неудобный для использования непосредственно в классе Client. Непосредственно класс Target или его наследники (например, в случае возможности использования нескольких альтернативных вариантов реализации) реализуют интерфейс, удобный для использования, но либо по принципу композиции включают в себя объект класса Adaptee, либо в некотором методе вступают во взаимодействие по принципу ассоциации (такая связь показана на схеме на рис.7).

Таким образом, не затрагивая клиентскую часть приложения на уровне класса-адаптера инкапсулируется весь алгоритм адаптации технологий, что дает возможность изменения реализующего функционал объекта (библиотеки, технологии) без изменений на клиентской стороне, обеспечивая изоляцию изменений.

4. **Компоновщик (Composite).** Это структурный паттерн, который позволяет удобный образом компоновать сложные иерархические структуры объектов, которые требуется обрабатывать единообразно. Иерархическая организация объектов используется довольно часто. Типовыми примерами являются структура каталогов на диске, элементы управления в пользовательском интерфейсе. Схема применения паттерна представлена на рис. 8.

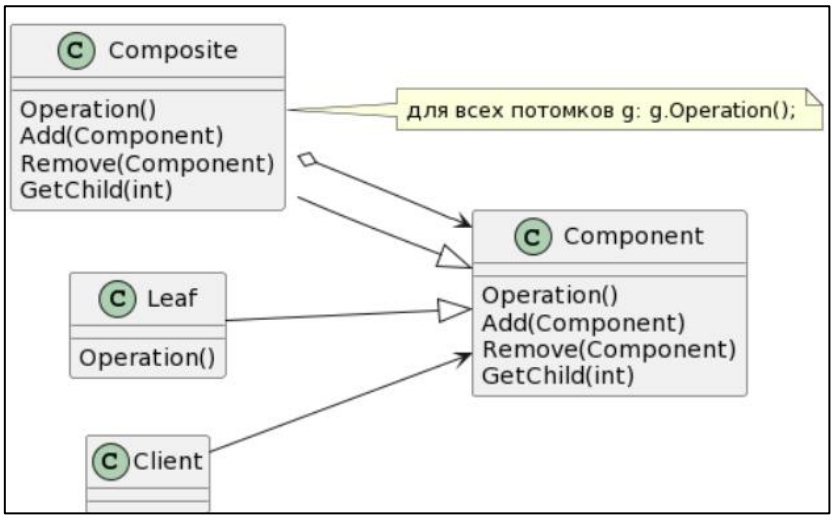

Рис. 8. Схема использования паттерна «Компоновщик»

В данном случае класс Component является вершиной иерархии разных компонентов, определяющий общий интерфейс обработки (в данном случае это представлена методом Operation()) и интерфейс для формирования подчиненных элементов (ведение списка подчиненных элементов, добавление и удаление из него, получение конкретного подчиненного элемента). Класс Component является родительским для составных (имеющих подчиненные) (класс Composite) и листовых (класс Leaf) объектов. С классом Composite у класса Component есть еще отношение агрегации, так как составной объект как свою часть может иметь как составной объект, так и листовой.

Принцип единообразной обработки операций для всех элементов структуры обеспечивается возможностью в методе обработки класса Composite вызова методов, реализующих ту же операцию, для всех подчиненных элементов. Например, таким образом, определяется, какому объекту иерархии элементов управления окна направляется сообщение мыши – по координатам мыши определяется сначала главное окно приложение, потом происходит обращение к дочерним элементами, например, панели инструментов, и далее, пока не будет найден элемент управления, например, кнопка, на которую действительно нажал пользователь.

5. **Фасад (Facade)**. Структурный паттерн Фасад является самым используемым, особенно в случае, когда создается многокомпонентное приложения, состоящее из нескольких подсистем. Фасад предоставляет унифицированный интерфейс вместо набора интерфейсов некоторой подсистемы, являясь своеобразной «входной дверью» в подсистему. Обычно этот интерфейс определяется в терминах не реализации подсистемы, а в терминах более высокого уровня, который отражает сервисы, которые предоставляет подсистема вовне. Схема применения представлена на рис. 9.

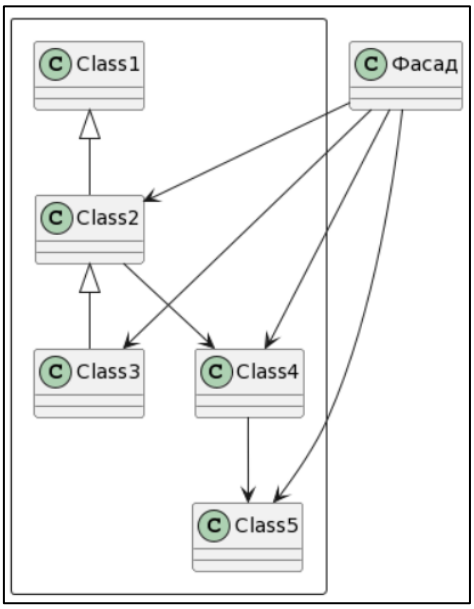

Рис. 9. Схема использования паттерна «Фасад»

Все классы подсистемы, которые могут быть недоступны извне. Между ними могут быть организованы очень сложные взаимозависимости. Благодаря использованию класса Фасад клиентской части приложения не необходимости знать сложную систему классов подсистемы – общение будет происходить через объект класса Фасад.

Часто класс Фасад может выполнять и функцию адаптера, переводя интерфейс терминов реализации подсистемы в термины предметной области клиентской части приложения.

6. **Команда (Command).** Реализация сложного поведения, особенно при сложной архитектуре, когда обработка может выполняться другими, независимыми компонентами, например, серверами, нередко

требует сохранения информации о команде, формирования очереди выполнения команд, поступающих от разных компонентов, их протоколирования, отмены. Реализация этих задач становится возможной благодаря использованию поведенческого шаблона проектирования «Команда». Схема использования паттерна «Команда» представлена ниже на рис. 10.

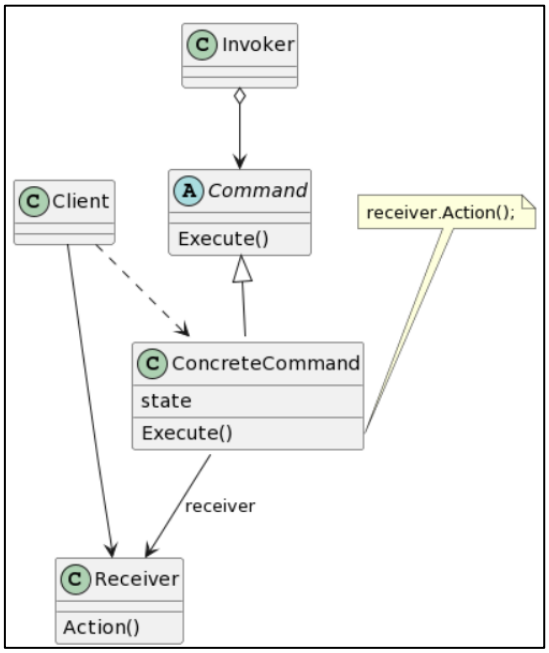

Рис. 10. Схема использования паттерна «Команда»

В данной схеме абстрактный класс Command определяет интерфейс выполнения команд, возможно, привлекающим класс Invoker – класс-инициатор выполнения команд. Например, типовым примером использования инкапсуляции сведений о команде в отдельном классе является организация общения с серверной базой данных. В этом случае на месте класса Invoker находится классы, обеспечивающие работу с провайдером СУБД. Конкретные типы команд могут быть реализованы отдельными классами в иерархии и иметь отдельные данные и методы. Создание команд может осуществляться как классом Client, если команда высокоуровневая, так и отдельным классом-получателем результата запроса Receiver, который может выполнять роль адаптера в переводе с терминов предметной области на термины формирования

и выполнения команд, ведения очереди выполнения команд и отложенного выполнения. Тут же могут быть организованы средства протоколирования выполнения команд, через которых можно осуществить откат или повторение команд.

7. **Итератор (Iterator).** Еще одним поведенческим шаблоном проектирования, который активно применяется, в том числе в библиотеках реализации сложных по организации составных структур данных (коллекций), является паттерн «Итератор». Часто необходимо обработать все элементы структуры, обеспечив обход каждой из его составляющей один раз. Паттерн «Итератор» обеспечивает универсальный способ обеспечения последовательного доступа ко всем элементам составного объекта, не раскрывая его внутреннего представления. Схема применения паттерна «Итератор» представлена на рис. 11.

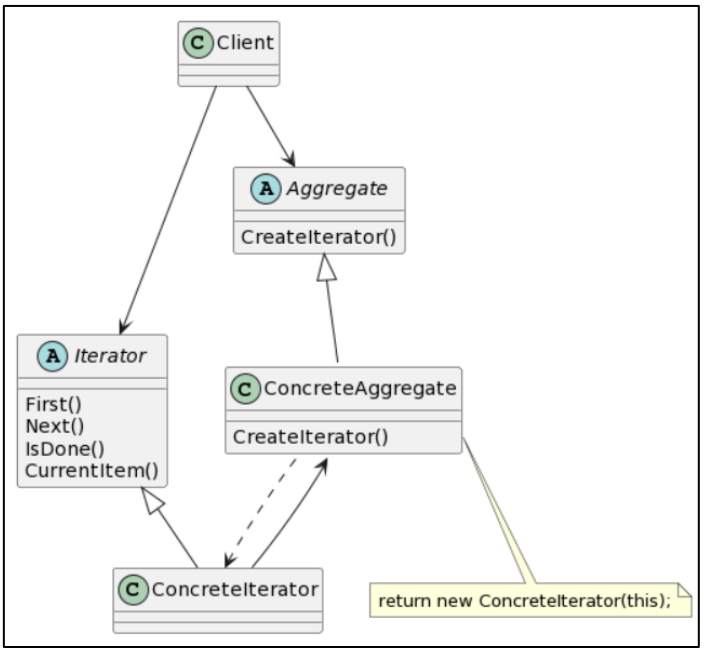

Рис. 11. Схема использования паттерна «Итератор»

Класс Aggregate определяет сложную структуру элементов, которую использует класс Client, которому требуется произвести обход включенных в структуру элементов. Чтобы не раскрывать структуру класса-агрегата, для каждого типа класса-агрегата создается класс ConcreteIterator, раскрывающий единообразный интерфейс Iterator, ко-

торый известен классу Client. Интерфейс итератора обеспечивает клиента методами для получения первого элемента сложной структуры, для перехода к следующему элементу, для получения текущего элемента и метода, с помощью которого можно узнать, что элементы в структуре закончились. Объект класса ConcreteIterator создается объектом-агрегатом с помощью фабричного метода CreateIterator(). Соответственно, получив итератор, клиент может использовать итератор для получения всех элементов агрегата последовательно. При этом между классами итератора и агрегата существует сильная связь, а между клиентом и агрегатом, несмотря на возможность получения отдельных его элементов – слабая связь.

Итераторы и принципы их применения мы видим в разных языках программирования при реализации классов-коллекций с точностью до именования методов, которые обеспечивают схему использование итератора.

8. **Наблюдатель (Observer).** Поведенческий паттерн «Наблюдатель» также имеет много реализаций в библиотеках на разных языках программирования. Применение этого паттерна является основой для обработки событий и изменений в зависимых объектах на основе каких-то определенных изменений в другом, наблюдаемом объекте. Схема применения паттерна «Наблюдатель» представлена на рис. 12.

Принцип реализации паттерна «Наблюдатель» основан на схеме «источник-адресат». Источником является какой-то класс, с которым могут возникать некоторые события. Таким классом на Рис. 11 является класс ConcreteSubject, реализующий интерфейс абстрактного класса Subject. Данный интерфейс предполагает некоторую структуру хранения ссылок на объекты-адресаты, которым необходимо узнать, что событие произошло.

Класс наблюдателя (адресата) ConcreteObserver, раскрывающий интерфейс абстрактного класса Observer, определяет методы, которые должны быть вызваны при возникновении события. На схеме этим методом является метод Update(). Класс источника позволяет классу адресату осуществить подписку на событие с помощью метода класса

источника Attach(). Таким образом класс источника будет знать список всех заинтересованных объектов, которым необходимо сообщить о возникновении события путем вызова метода Notify(), внутри которого для всех объектов-адресатов вызывается метод обновления (Update()) и, возможно, передается какая-то дополнительная информация о возникновении события.

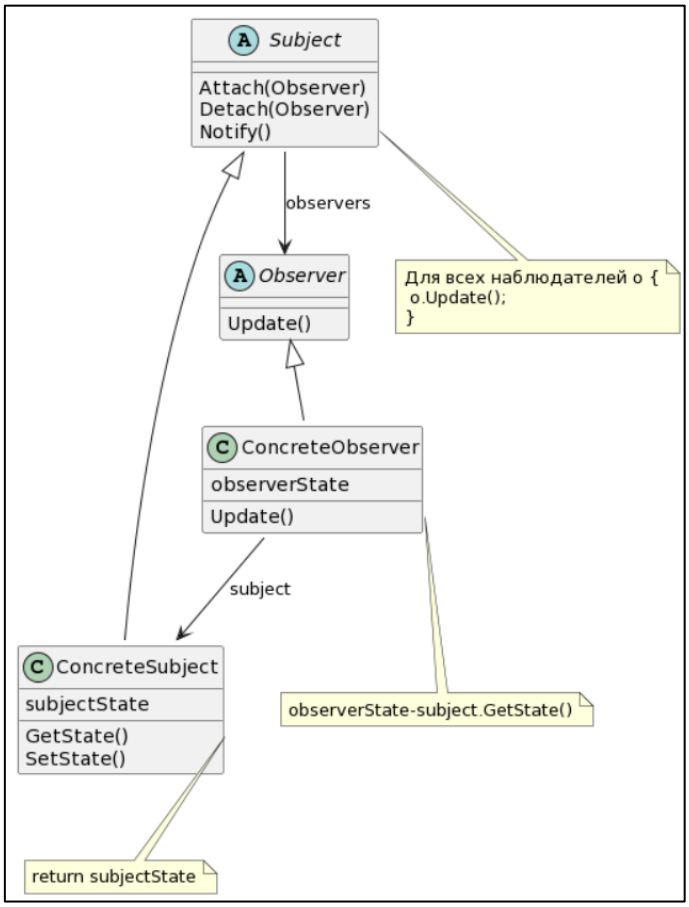

Рис. 12. Схема использования паттерна «Наблюдатель»

Обратим внимание, что помимо метода Attach() класс источника имеет также метод Detach(), обеспечивающий удаление из списка наблюдателей. Таким образом, списком наблюдателей можно гибко управлять по ходу работы программного обеспечения.

9. **Цепочка обязанностей (Chain of Responsibility).** Последним поведенческим шаблон проектирования, который мы упомянем в данном пособии, является паттерн «Цепочка обязанностей». Данный паттерн в случае сложной структуры программного обеспечения позволяет избежать явной привязки объекта, которому требуется выполнить

некоторую обработку данных, и объекта, который эту обработку будет выполнять. В частности, это может использоваться в различных распределенных структурах, когда несколько объектов-исполнителей, для которых следует сбалансировать нагрузку для устойчивости работы, или использоваться в ситуациях, когда обработка зависит от специфических условий, которые объекту-отправителю запроса неизвестны. Нередко данный паттерн используют для организации конвейерной обработки информации, когда весь алгоритм обработки представляет себя последовательность действий, которые должны выполняться разными объектами-исполнителями. Схема применения паттерна «Цепочка обязанностей» представлена на рис. 13.

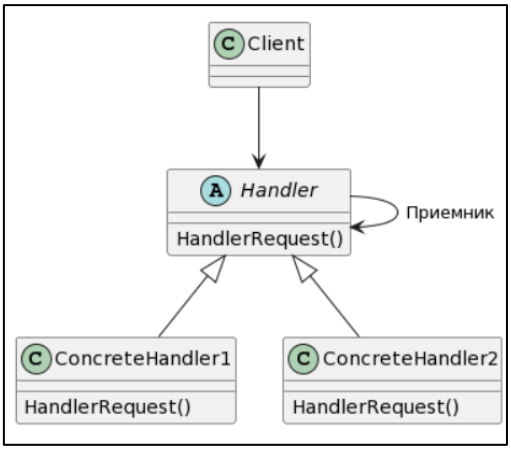

Рис. 13. Схема использования паттерна
«Цепочка обязанностей»

На схеме абстрактный класс Handler задает интерфейс обработки запроса и организации списковой структуры выстраивания цепочки объектов-обработчиков запроса. Каждый конкретный класс для обработки запроса или части запроса является его наследником. Когда клиент инициирует запрос, он продвигается по списку объектов обработки запроса, пока некоторый конкретный обработчик не возьмет на себя ответственность за работу. Обычно хранение списка организовано в вершине иерархии, чтобы обеспечить гибкость в общении с классом Client. Тем не менее, универсализация интерфейса обработки запроса совместно с применением паттерна итератор может обеспечить гибкость обработки и на стороне клиента.

ДИАГРАММЫ ЛОГИЧЕСКОГО УРОВНЯ

После разработки диаграмм вариантов использования и диаграммы классов начинается детальное проектирование на логическом и физическом уровнях, показывающие свойства реализации различных вариантов использования и других программных элементов.

Диаграммы логического уровня, в первую очередь, призваны отразить алгоритмы реализации вариантов использования и других важных для приложения алгоритмов, а также влияние этих алгоритмов на состояние участвующих в алгоритмах объектов. Варианты использования выходят на первый план из-за важности именно реализации их алгоритмов для приложения. Тем не менее, для любого вспомогательного алгоритма или класса могут создаваться диаграммы этих типов. Классическими диаграммами логического уровня являются диаграмма последовательности и диаграмма активности (или деятельности). Они создаются для каждого алгоритма (варианта использования), который необходимо реализовать в проекте.

Для отслеживания изменений в объектах классов под влиянием выполнения операций над ними создаются диаграммы состояний (или диаграммы конечных автоматов). Несмотря на то, что основным предметом рассмотрения на этой диаграмме является состояние объекта класса, оно рассматривается в динамике, показывая под воздействием каких внешних факторов и с помощью каких методов происходят изменения состояния объектов и каким образом объект будет работать в каждом из его состояний.

Разберем последовательно принципы реализации каждого из этих трех видов диаграмм.

1. **Диаграмма последовательности.**

Диаграмма последовательности – это диаграмма, которая служит для представления взаимодействия элементов модели (объектов) в форме последовательности сообщений и соответствующих событий на линиях жизни объектов. Обычно диаграмма последовательности стро-

ится для каждого из вариантов использования, конкретизируя его выполнение (конкретизирует в терминах взаимодействующих объектов конкретный прецедент).

Диаграмма последовательности показывает принципы функционирования приложения в конкретизации времени, но масштаб времени при этом не считается важным – имеет значение только последовательность передаваемых сообщений, но не длительность промежутков между передачей сообщений.

Основными элементами на диаграмме последовательностей являются объекты и их линии жизни. Объекты отображаются в виде прямоугольника с указанным в нем именем и/или типом объекта в верхней части диаграммы. Если в алгоритме варианта использования участвует единственный объект какого-то класса, достаточно указать только имя класса для него, подчеркивая, что нет явного значения в принципе его именования. Если в алгоритме варианта использования используется два и более объекта одного и того же типа, то требуется определить их различия – это различие достигается использованием разных имен объектов. Линия жизни исходит из прямоугольника объекта вниз. На линии жизни будут обозначаться фрагменты активности объекта. Они отображаются прямоугольниками, расположенными на линии жизни. Этот прямоугольник соответствует фрагменту времени на линии жизни объекта, в котором основной поток команд выполняет именно этот объект в рамках выполнения определенного метода. В момент уничтожения объекта во время выполнения прецедента, может быть реализовано прерывание линии жизни, что отражается символом «×» в конце линии жизни.

Передача управления между объектами отображается посредством стрелок, обозначающих передачу сообщений. Сообщения могут быть синхронными (с ожиданием ответа) (сплошная линия с закрашенной стрелкой) и асинхронными (посылающий сообщение объект не ожидает ответа, а продолжает свою работу) (сплошная линия со стрелкой в виде буквы v). Ответный сигнал обозначается пунктирной линией. Выделяют также сообщения, которые уходят вовне диаграммы

(в другую подсистему) или приходят извне. Они обозначаются стрелкой, в начале или в конце которой ставится закрашенный кружок, обозначающий внешнюю сторону. Каждое сообщение представляет собой вызов метода принимающего сообщение объекта, ответное сообщение – возвращаемое значение этого метода. Имена методов и при необходимости параметров и возвращаемых значений указываются как сопровождающая надпись радом со стрелкой сообщения.

Основные обозначения диаграммы последовательности представлены на рис. 14.

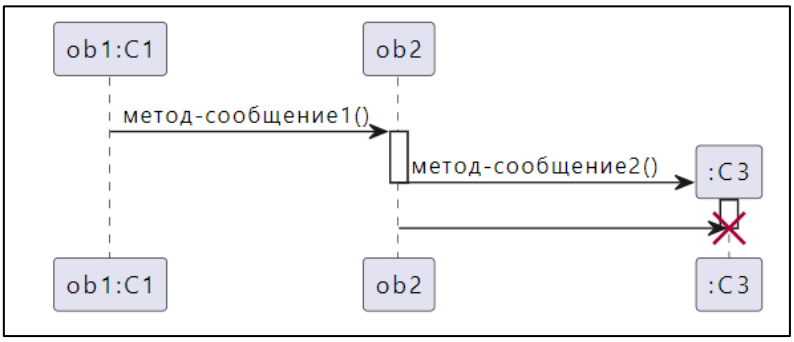

Рис. 14. Основные обозначения диаграммы последовательности

Сложные и вариативные варианты последовательностей обозначаются комбинированными фрагментами, которые заключаются в прямоугольники с указанием типа комбинированного фрагмента. С понятием комбинированного фрагмента связаны термины оператора взаимодействия (правила выполнения фрагментов), операнда взаимодействия (содержимое фрагмента) и ограничения (условия, при котором выполняется фрагмент или его операнд) (рис.15).

На рисунке представлены два типа комбинированных фрагментов (alt и loop) – тип фрагмента (оператор взаимодействия) указывается в верхнем левом углу прямоугольника фрагмента. Если у комбинированного фрагмента есть несколько операндов взаимодействия, они разделяют прямоугольник фрагмента пунктирными линиями на независимые фрагменты, означающие область действия операнда. У каждого операнда имеется сторожевое условие – указывается в квадратных скобках в начале операнда.

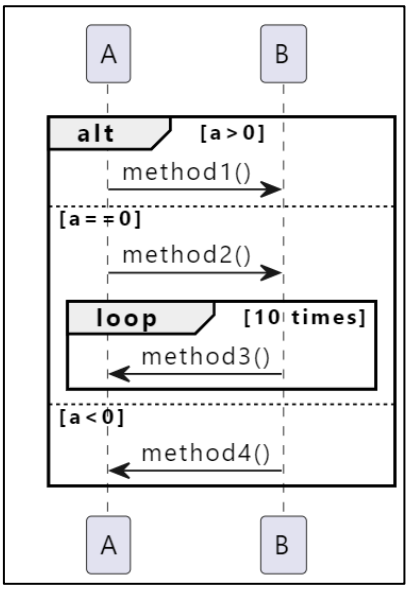

Рис. 15. Демонстрация обозначений
комбинированных фрагментов

Существует несколько типовых видов операторов взаимодействия внутри комбинированных фрагментов:

1. Альтернативы (alt) – способ указания выполнения одного из нескольких альтернативных вариантов последовательностей действий. Каждый вариант описывается сторожевым условием – условием, при котором управление переходит именно в этот фрагмент. Необходимо обеспечить, чтобы сторожевые условия всех операндов обеспечивали все возможные случаи и разбивали их на непересекающиеся множества. Является аналогом условного оператора (if) или оператора (switch) в языках программирования.

2. Завершение (break) – способ завершения прецедента при определенных условиях, можно сказать аварийного завершения прецедента. Обычно содержит описание действий некоторого альтернативного потока прецедента. Обязательно имеет сторожевое условие, обозначающее, в каком случае требуется передача управления на этот альтернативный поток. С этой точки зрения данный комбинированный фрагмент является аналогом использования генерации и обработки исключительных ситуаций или оператора break при выходе из цикла в языках программирования.

3. Параллельность (par) – определяет параллельное выполнение своих операндов.

4. Критический регион (critical) – комбинированный фрагмент, который используется совместно с комбинированным фрагментом параллельности и описывает операнд, который своеобразным образом блокирует выполнение других операндов комбинированного фрагмента параллельности, обозначая, что важно, чтобы действия, попавшие в критический регион, были выполнены без параллельного выполнения других операндов.

5. Рассмотрение (consider) и игнорирование (ignore). Данные комбинированные фрагменты позволяют описать ограничения на вызов методов внутри комбинированного фрагмента, акцентируя внимание на разных моментах. Рассмотрение указывает только разрешенные для вызова методы, игнорирование – только запрещенные методы.

6. Цикл (loop) – <цикл>::=loop[(<minint> [, <maxint>])]. Описывает фрагмент, который будет выполняться при истинности сторожевого условия несколько раз (согласно указанным в скобках границам).

7. Отрицание (neg) – определяет последовательность действий, которую следует считать запрещенной, акцентируя внимание на действиях, которые требуется запретить.

8. Необязательный (opt) – определяет последовательность действий, которую выполняется в случае выполнения сторожевого условия или не выполняется в противном случае. Это упрощенный вариант альтернативного фрагмента, соответствующий условному оператору if без ветви else в языках программирования.

На диаграмме последовательности можно указать дополнительную информацию выполнения прецедента. Так, например, в начале линии жизни объекта можно указать условие, которое определяет состояние объекта для корректного выполнения прецедента. Это указание должно трактоваться так, что, если объект не находится в указанном состоянии, прецедент вообще не должен выполняться.

Другим примером является указание рядом с сообщением временных требований на его передачу и получение ответа на него. Это важно, например, для алгоритмов, которые требуют быстрого выполнения, чтобы можно было их использовать в online-режиме.

2. **Диаграмма активности (диаграмма деятельности).**

Диаграмма активности изображает поведение объекта или системы во время выполнения варианта использования с использованием моделей потока управления. Единицами описания алгоритма являются действия. Действие (action) представляет собой элементарную единицу спецификации поведения, которая не может быть далее декомпозирована в форме деятельности. Это обычно или отдельные операторы, или вызов некоторой функции, которая может быть описана отдельной диаграммой активности.

Диаграмма представляет собой своеобразную блок-схему, на которой обозначен поток передачи управления с узлами принятия решений (условия), узлами разделения и слияния для параллельного выполнения. При ветвлении следует указывать сторожевые условия для определения того, в каком случае процесс должен идти по той или иной ветви действий.

В целом, диаграммы активности очень похожи на классические блок-схемы описания алгоритмов. Поэтому более подробные описания особенностей ее формирования не требуются.

3. **Диаграмма состояний (диаграмма конечного автомата).**

Диаграмма состояний призвана отслеживать жизненный цикл объекта класса и изменения в его поведении на различных этапах. В объектно-ориентированном программировании состояние объекта рассматривает как совокупность значений полей класса. Похожие по поведению объекта совокупности значений полей класса объединяются в группу, которая будет является отдельным состоянием на диаграмме состояний. Обычно такая группа описывается набором условий, которым должны удовлетворять поля класса, чтобы объект принадлежал этому состоянию.

Диаграмма состояний является графом, узлами которого являются выделенные состояния для объекта класса. Внутри состояния может указываться информация о методах, соответствующих событиям, происходящим с объектом, находящимся в этом состоянии. Существуют общепринятые события состояний, например, entry – событие входа объекта в состояние, exit – событие выхода объекта из состояния, do – событие пребывания объекта в состоянии. Каждое из событий указывается вместе с вызовом метода объекта, который является его обработчиком. Также можно описать собственные события, которые следует отслеживать, когда объект находит в указанном состоянии.

Связи в графе определяют переходы объекта из одного состояния в другой и условиях их реализации. Переходы могут быть триггерными и нетриггерными. Триггерный переход связан с внешним воздействием на объект, который заключается в вызове извне метода класса при выполнении определенных условий, которые задаются сторожевым условием перехода. Нетриггерный переход обусловлен завершением do-деятельности объекта, также может иметь сторожевое условие. Сторожевое условие может использовать в своей формулировке поля класса и/или параметры и возвращаемое значение связанного с переходом метода.

Состояния на диаграмме обозначается прямоугольником с закругленными углами, переходы обозначаются стрелками. Отличие триггерного и нетриггерного переходов отображается наличием в надписи надо стрелкой указания вызова метода, вызвавшего переход. Так как do-деятельность определяется методом, указанном в самом состоянии объекта, указание метода на самом переходе означает, что переход является триггерным.

На диаграмме выделяют также псевдосостояния – состояния, которым не соответствует поведение. Например, такими состояниями являются начальное псевдосостояние, переход из которого означает вызов конструктора объекта, и конечное псевдосостояние, переход в ко-

торое означает вызов деструктора объекта. Кроме того, можно создавать сложные композитные состояния, обозначающие собственное сложное поведение.

Основные обозначения диаграммы состояний, включая разные виды состояний и переходы между ними, представлены на рис. 16.

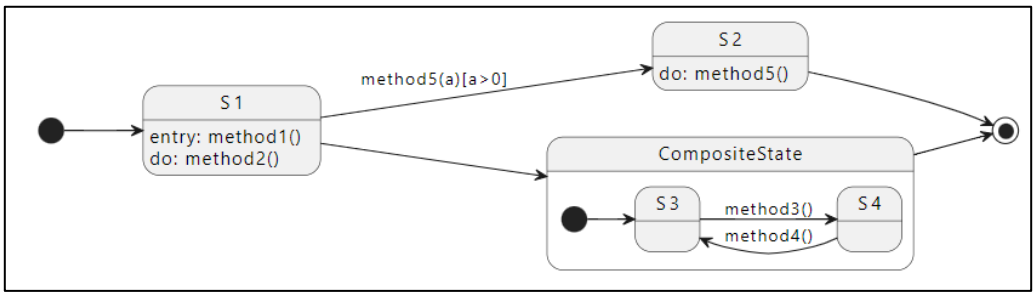

Рис. 16. Основные обозначения диаграммы состояний

На рис.16 представлены состояния S1 и S2, имеющие описания деятельности при нахождении объекта в этом состоянии (do-деятельность). Переходы без этих состояний без пометок являются нетриггерными (переход из состояния S1 в составное состояние CompositeState или переход из S2 к деструктору). Переход из состояния S1 в состояние S2 является триггерным – он вызывается посредством вызова метода method5() с параметром a при сторожевом условии a>0.

Диаграмма состояний создается для классов, но так как классов в приложении может быть очень много, не всегда полный охват всего множества классов таким инструментом будет уместен. Обычно диаграмма состояний создается для классов, которые обозначают долговременный процесс с сохранением промежуточных результатов этого процесса (процессные объекты). Такими объектами могут быть объекты заказа, длительной обработки, т.е. любой класс, для которого можно выделить поле статуса, который необходимо отслеживать. Чаще всего состояния для таких объектов и характеризуются этапам выполнения алгоритма или статусом объекта. Тем не менее, ключевым вопросом здесь является именно долговременность существования объекта в каждом отдельном состоянии, поэтому нельзя применять эту диаграмму для описания алгоритмов, которые проводятся быстро и с

оперативным сохранением результатов (результаты хранятся в переменных, время жизни которых – время выполнения алгоритма как последовательности команд).

Другое применение диаграммы состояний – классы для пассивных объектов (классы данных). Условие на переменные класса определяют различные состояния класса. Переходы между состояниями используются в основном триггерного типа, так как объект пассивный и меняется только под воздействие внешней среды (на основе вызова операций класса извне из других объектов).

Рассмотрим правила создания диаграмм логического уровня на PlantUML. Формирование диаграммы последовательности, в целом, похоже на описание алгоритма. Имена объектов не требуют предварительного объявления. Главное здесь – последовательность передачи управления. Тем не менее, можно использовать специальные обозначения для указания, например, актора или базы данных. Тогда предварительно имена должны быть объявлены с указанием их типа:

```
actor a1
database db1
```

Основная часть описания диаграммы последовательности связана с описанием передачи сообщений, который оформляются по следующей схеме:

```
Имя_источника -> Имя_адресата :Сообщение
```

С помощью команд activate и deactivate можно задать период активности любого объекта. Для обозначения источников или адресатов сообщений вовне системы используются символы «[» или «]». С помощью команды create можно создать объект по ходу выполнения последовательности. Прямоугольник, соответствующий этому объекту, будет отображен ниже других объектов, в момент его создания. Уничтожение объекта можно осуществить с помощью команды destroy.

Комбинированный фрагмент обозначается в виде именованной группы, содержащей внутри описание действий внутри фрагмента.

```
group My own label
    A -> B : Message
end
```

Комбинированные фрагменты стандартных типов обозначаются соответствующему типу фрагмента командой. В случае нескольких операндов каждый следующий операнд начинается ключевым словом else с указание далее сторожевого условия для этого операнда.

Синтаксис описания диаграммы активности лучше всего демонстрирует следующий простой пример:

```
@startuml
start
repeat
  :Тестирование;
    if (Что-то пошло не так?) then (нет)
      :Тест пройден;
      break
    endif
    ->ошибка;
    :Сообщение: "Длинное сообщение об ошибке";

repeat while (Что-то пошло не так при\n
          выводе сообщения об ошибке?)
          is (да) not (нет)
:Сообщение: "Всё прошло успешно";
stop
@enduml
```

Здесь мы видим максимальную похожесть на язык программирования или псевдокод. Любое действие определяется сообщением, начинающимся с символа «:». Остальные команды взяты аналогичны использованию в языке программирования. Фрагмент «->ошибка;» определяет нестандартную надпись около стрелки.

В следующих фрагментах кода представлены примеры распараллеливания процесса и применение цикла while:

```
start fork
  :Действие 1;
fork again
  :Действие 2;
end fork
stop
```

```
while (данные доступны?) is (да)
  :чтение данных;
  :построение диаграммы;
  backward :очистка экрана;
end while (нет)
:окончание работы;
```

Ключевое слово is в данном случае обозначает начало тела цикла. Строка в скобках становится надписью рядом со стрелкой входа в итерацию. Ключевое слово backward определяет обратное направление стрелки для возврата к оператору цикла.

При формировании диаграммы состояний акцент ставится на организации связей. Поэтому упоминание имени в обозначении связи означает формирование соответствующего состояния. Тем не менее, удобнее объявить сначала все имена состояний с помощью ключевого слова «state». Любые события состояния описываются через символ «:» после имени принадлежащего состояния. Псевдосостояния описываются группой символов «[*]». Описание составного состояния можно заключить в фигурные скобки.

```
state CompositeState {
[*] --> S1
S1 --> S2 : Message1
S2 --> S1 : Message2
}
```

Для примера рассмотрим построение диаграммы последовательности для варианта использования оформления заказа в информационной системе ресторана. Диаграмма представлена на рис. 17.

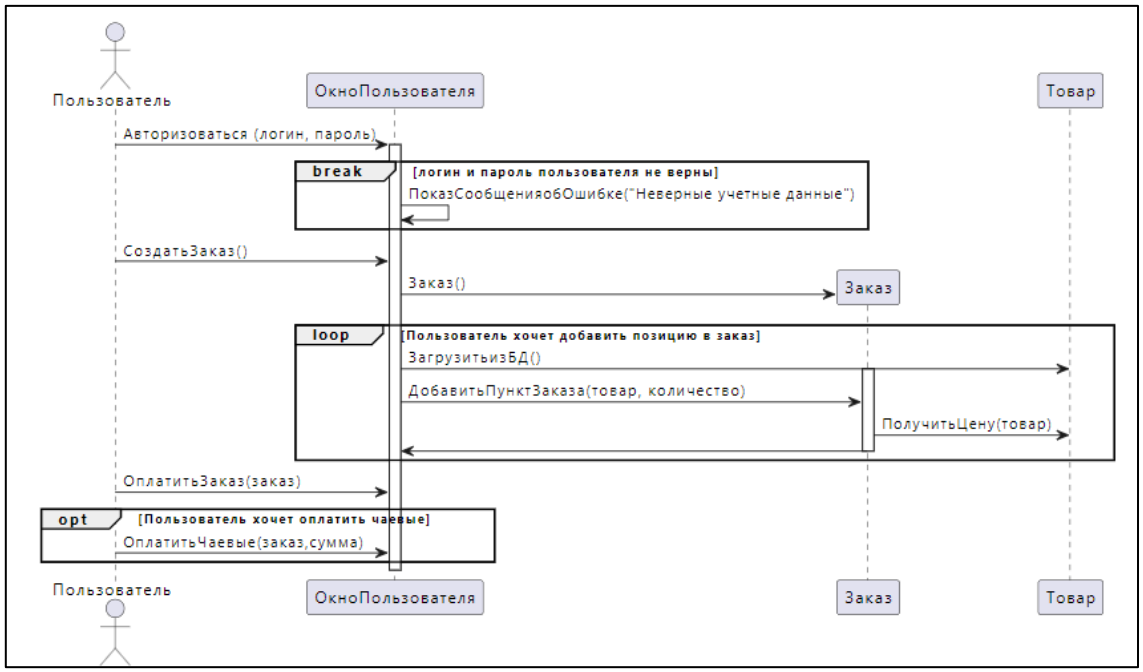

Рис. 17. Диаграмма последовательности для варианта использования оформления заказа в ресторане

В данном случае основным активным объектом является объект класса ОкноПользователя. Это демонстрируется прямоугольником на линии жизни объекта этого класса в течение всего процесса выполнения варианта использования. Актор через пользовательский интерфейс активизирует работу с окном и далее оно все время активно. На диаграмме представлено использование трех комбинированных фрагментов – фрагмента break, который обеспечивает прерывание варианта использования в случае некорректной авторизации пользователя, фрагмента loop для формирования заказа и включения отдельных позиций в заказ, пока есть желание пользователя (следует обратить внимание, что объект класса Заказ создается только перед началом этого цикла), фрагмента opt, действия которого происходят только в случае желания пользователя оплатить чаевые.

Следует обратить внимание на то, что сообщения представляются вызовами методов принимающего класса и большинство методом можно найти на диаграмме классов на Рис.4. Однако имеют методы, отсутствующие на диаграмме классов, например, ПоказСообщениеобОшибке(). Это демонстрирует тот факт, что диаграмма классов в течение периода проектирования постоянно изменяется, дополняется и модифицируется, так как при детальном проектировании алгоритмов может возникнуть необходимость в дополнительной функциональности, реализации дополнительных классов или методов.

Программный код на PlantUML для построения данной диаграммы представлен ниже.

```
@startuml
actor Пользователь

Пользователь -> ОкноПользователя:
                    Авторизоваться (логин, пароль)
activate ОкноПользователя
break логин и пароль пользователя не верны
  ОкноПользователя -> ОкноПользователя:
    ПоказСообщениеобОшибке("Неверные учетные данные")
end
Пользователь -> ОкноПользователя: СоздатьЗаказ()
create Заказ
ОкноПользователя -> Заказ: Заказ()
loop Пользователь хочет добавить позицию в заказ
  ОкноПользователя -> Товар : ЗагрузитьизБД()
  activate Заказ
  ОкноПользователя -> Заказ :
              ДобавитьПунктЗаказа(товар, количество)
  Заказ -> Товар : ПолучитьЦену(товар)
  Заказ -> ОкноПользователя
  deactivate Заказ
end
Пользователь -> ОкноПользователя:ОплатитьЗаказ(заказ)
opt Пользователь хочет оплатить чаевые
  Пользователь -> ОкноПользователя:
                          ОплатитьЧаевые(заказ,сумма)
end
deactivate ОкноПользователя
@enduml
```

Для этого же варианта использования создадим диаграмму активности (рис. 18).

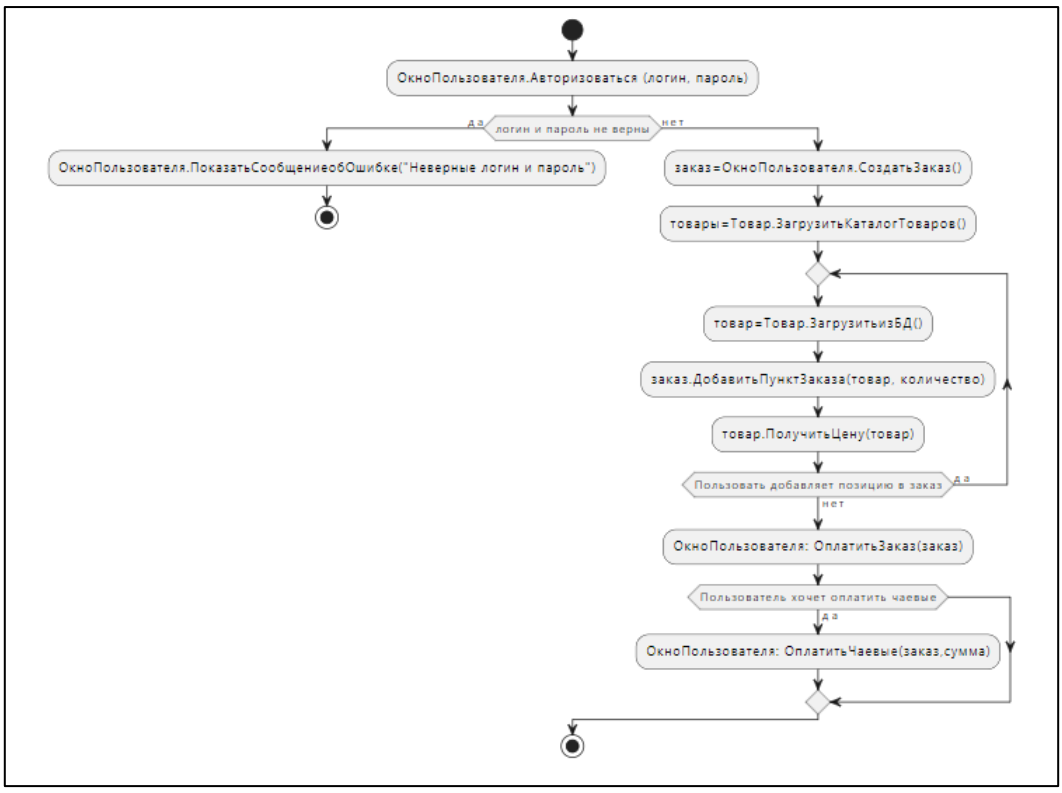

Рис. 18. Диаграмма активности для варианта использования оформления заказа в ресторане

Программный код построения этой диаграммы активности представлен ниже:

```
@startuml
start
:ОкноПользователя.Авторизоваться (логин, пароль);
if (логин и пароль не верны) then (да)
  :ОкноПользователя.ПоказатьСообщениеобОшибке
                    ("Неверные логин и пароль");
  stop
else (нет)
  :заказ=ОкноПользователя.СоздатьЗаказ();
  :товары=Товар.ЗагрузитьКаталогТоваров();
  repeat
    :товар=Товар.ЗагрузитьизБД();
    :заказ.ДобавитьПунктЗаказа(товар, количество);
    :товар.ПолучитьЦену(товар);
  repeat while (Пользовать добавляет позицию в заказ)
is (да) not (нет)
```

```
    :ОкноПользователя.ОплатитьЗаказ(заказ);
    if (Пользователь хочет оплатить чаевые) then (да)
      :ОкноПользователя.ОплатитьЧаевые(заказ,сумма);
    endif
  endif
  stop
  @enduml
```

Здесь команды start и stop означают блоки начала диаграммы деятельности и ее завершения. Каждое действие описывается, начиная с символа «:» и заканчивается символом «;». Остальной поток управления описывается операторами языка, который похож на обычный язык программирования. В условных операторах указания в скобках строк «(да)» или «(нет)» указывают надписи на исходящих ветвях из блока условного оператора.

Для примера диаграммы состояний представим диаграмму класса Очередь, который может использоваться для организации обслуживания заказов в ресторане (так как принадлежит другой подсистеме, не входит в состав диаграммы классов, представленной на Рис.4). Очередь формируется из позиций заказов, которые в определенном порядке должны готовиться на кухне. Диаграмма состояний представлена на рис. 19.

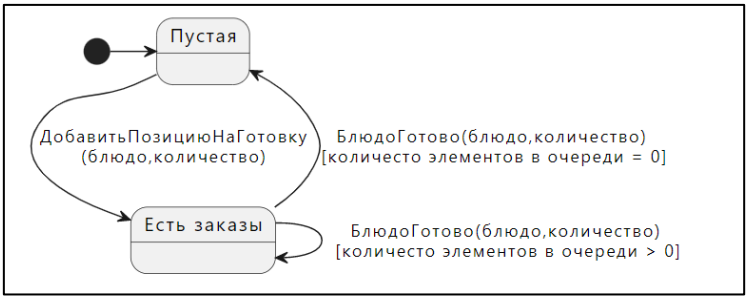

Рис. 19. Диаграмма состояний для класса «Очередь блюд на готовку»

В данном случае есть всего два состояния очереди – пустая очередь и очередь с заказами. Все переходы в диаграмме состояний триггерные, так как очередь сама не имеет активности, а организует хране-

ние информации. При открытии ресторана и запуска приложения (создание объекта очереди) объект переходит в состояние «Пустая», из которого в состояние «Есть заказы» объект переходит, если в очередь была добавлено блюдо для готовки. Обратный переход возможен, когда блюдо было готово, извлечено из очереди и после этого выполняется сторожевое условие отсутствия других элементов в очереди. В остальных случаях непосредственное состояние переменных объекта изменится, но очередь остается непустой, поэтому получается переход-петля из состояния «Есть заказы» в себя.

Программный код на PlantUML для построения этой диаграммы будет выглядеть следующим образом:

```
@startuml
state "Пустая" as S1
state "Есть заказы" as S2
[*] -> S1
S1 -down-> S2 : ДобавитьПозициюНаГотовку
                 \n(блюдо,количество)
S2 -> S2 : БлюдоГотово(блюдо,количество)
           \n [количесто элементов в очереди > 0]
S2 -up-> S1 : БлюдоГотово(блюдо,количество)
            \n[количесто элементов в очереди = 0]
@enduml
```

ЛАБОРАТОРНАЯ РАБОТА. ЭТАП 4 КОМАНДНОГО ПРОЕКТА «СОЗДАНИЕ ДИАГРАММ ЛОГИЧЕСКОГО УРОВНЯ». МЕТОДИЧЕСКИЕ УКАЗАНИЯ

Необходимо получить базовые навыки детализации отдельных элементов проекта системы в виде реализации алгоритмов и описания влияния алгоритмов на состояние класса. Это осуществляется посредством создания диаграмм логического уровня. Так как диаграмм логического уровня в проекте достаточно много, необходимо создать только несколько ключевых диаграмм каждого типа.

Каждый член команды создает по две или три (в зависимости от сложности) диаграмме последовательности, по одной диаграмме состояний, по две или три диаграмме активности.

Диаграммы последовательности и диаграммы активности строятся для вариантов использования. Диаграмма состояний строится для отслеживания жизненного цикла объекта некоторого класса, т.е. строится для класса. При выборе вариантов использования для реализации диаграмм последовательности и активности и классов для диаграммы состояний рекомендуется подбирать специфичные для проекта программные элементы.

Для описания диаграмм последовательности и активности может помочь описание соответствующих прецедентов, так как именно там содержится описание алгоритма выполнения варианта использования, только в терминах пользователя.

Следует обратить внимание, что терминология диаграмм логического уровня использует в первую очередь термины программиста. Основными ключевыми элементами, обозначенными на диаграммах логического уровня, должны быть классы, переменные, методы, указанные на диаграмме классов. Таким образом, диаграммы логического уровня становятся указанием программисту, с каким объектами работать и в какой последовательности вызывать те или иные методы объектов для реализации необходимой функциональности.

При необходимости имеет смысл модифицировать диаграмму классов, созданную на этапе 3, и описание спецификации методов, которые эту диаграмму дополняет.

ДИАГРАММЫ ФИЗИЧЕСКОГО УРОВНЯ

Диаграммы физического уровня делают другой акцент – не на алгоритмическом наполнении, а на том, каким образом созданное программное обеспечение должно быть установлено на аппаратном обеспечении, каким условиям должна удовлетворять среда исполнения созданного программного обеспечения, каковы должны быть механизмы взаимодействия между отдельными (особенно удаленными) компонентами программного обеспечения.

Основными диаграммами физического уровня являются диаграмма компонентов и диаграмма развертывания.

1. **Диаграмма компонентов.**

Компонент – это элемент модели, представляющий некоторую модульную часть системы с инкапсулированным содержимым, спецификация которого является взаимозаменяемой в его окружении. Компонентом может быть подсистема, библиотека, модуль, специализированное программное обеспечение, в ряде случаев отдельный класс.

Наиболее близкая к физическому уровню интерпретация компонента связана с понятием артефакта – отдельного файла, информационного обеспечения и пр., которые надо разместить определенным образом в определенном месте, установить, настроить, чтобы программное обеспечение заработало. Можно сказать, что артефактом представляется любой файл, который необходим для корректной работы программной системы.

Компоненты должны взаимодействовать друг с другом. В отличие от диаграмм логического уровня взаимодействие не трактуется как вызов методов. Взаимодействие производится уже на уровне операционных систем, сетевого взаимодействия, т.е. фактически имеет свой собственный протокол, которым не всегда может управлять программист, который определяется системными настройками.

Компоненты изображаются прямоугольниками со специальной пиктограммой в правом верхнем углу. Тем не менее, для отдельных

видов компонентов, например, для базы данных или других видов хранилищ существуют другие обозначения.

Для изображения на диаграмме компонентов их взаимодействий друг с другом определено понятие интерфейса. Интерфейсы разделяются на два типа. Предоставляемый интерфейс (provided interface) – интерфейс, который компонент предлагает для своего окружения. Предоставляемый интерфейс отображается с помощью исходящей из прямоугольника линии с кругом на конце. Требуемый интерфейс (required interface) – интерфейс, который необходим компоненту от своего окружения для выполнения заявленной функциональности, контракта или поведения. Требуемый интерфейс отображается с помощью исходящей из прямоугольника линии с дугой на конце, которую можно назвать портом. Фактически взаимодействие представляется на диаграмме включением круга предоставляемого интерфейса в дугу требуемого интерфейса.

Так, предоставляемым интерфейсом могут быть данные, хранящиеся в некотором хранилище, выполнение некоторой обработки удаленным компонентом и т.д. Обычно инициатором взаимодействия является тот компонент, которому требуется какой-то интерфейс. Компонент, предоставляющий интерфейс, в некотором смысле является пассивным слушателем, когда другому компоненту потребуется этот интерфейс. Поэтому при определении принципов взаимодействия следует руководствоваться ответами на вопросы, с какой стороны возникает необходимость во взаимодействии, кто является инициатором взаимодействия. Изображения двух компонентов, один из которых предоставляет интерфейс (компонент1), а другой требует интерфейс (компонент2) представлены на рис. 20.

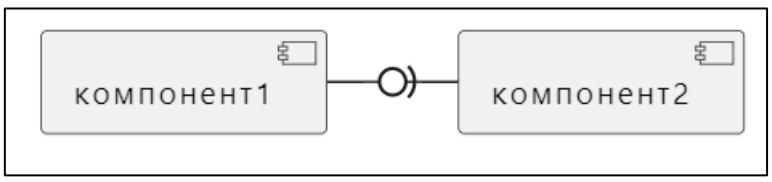

Рис. 20. Обозначения компонентов и их взаимодействии с помощью интерфейсов на диаграмме компонентов

2. Диаграмма развертывания.

Диаграмма развертывания предназначена для представления общей конфигурации или топологии распределенной программной системы и содержит изображение размещения различных артефактов по отдельным узлам системы. Узлами системы являются отдельные устройства. Устройства обычно представляются на диаграмме параллелепипедами. Внутри него могут быть обозначены условия, которым должна удовлетворять среда выполнения приложения на устройстве. Это может быть операционная система, предустановленная среда исполнение, наличие определенного программного обеспечения. Связи между узлами показываются обычно направленными стрелками, обозначая таким образом инициаторов взаимодействия. Отдельные виды связи не выделяются, поскольку организация связи обусловлена техническими условиями взаимодействия определенных типов устройств.

Все артефакты, которые включены в диаграмму компонентов, должны быть распределены по узлам диаграммы развертывания. Таким образом, диаграмма развертывания дает возможность специфицировать физические узлы, необходимые для размещения на них исполнимых компонентов программной системы, показать физические связи между физическими узлами системы на этапе ее исполнения, выявить узкие места системы и реконфигурировать ее топологию для достижения требуемой производительности.

Рассмотрим правила создания диаграмм физического уровня на PlantUML. Общие принципы синтаксиса PlantUML в части описания диаграммы компонентов и диаграммы развертывания касаются только объявления объектов диаграммы. Так, описание компонентов на диаграмме компонентов осуществляется с помощью ключевого слова component, узлы на диаграмме развертывания объявляются с помощью ключевого слова node. Возможно использовать отдельных ключевых слов для обозначения отдельных типов узлов или отдельных типов компонентов, например, cloud для облачных компонентов, database для базы данных. Артефакт на диаграмме развертывания определяется с помощью ключевого слова artefact.

Для объявления связи интерфейсов на диаграмме компонентов используется последовательность символов «-o)-».

Таким образом, общий синтаксис большинства диаграмм, представляющий предмет диаграммы в виде графа, в целом, аналогичен с точностью до обозначений отдельных элементов. Описание же самих связей никаких серьезных особенностей не имеет.

На рис. 21 представлена простая диаграмма развертывания для мобильного приложения на платформе Android с синхронизацией информацией с базой данных, расположенной на сервере MySQL. В данном случае конкретизируется взаимодействие двух устройств – мобильного устройства и сервера, представляющего собой хранилище данных. К условиям предъявляются требования – операционная система Android на мобильном устройстве и установка серверного СУБД MySQL на сервере-хранилище. Непосредственными артефактами разработанной системы является мобильное приложение MyNote и база данных DBNote.

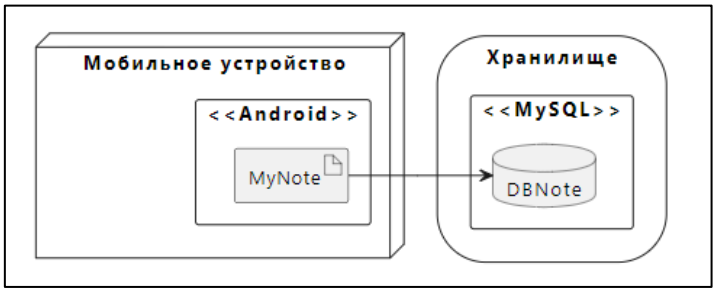

Рис. 21. Пример простой диаграммы развертывания

Программный код на PlantUML для диаграммы, представленной на рис.21, показан ниже:

```
@startuml
node "Мобильное устройство"{
   rectangle "<<Android>>" {
   artifact "MyNote" as app
   }
}

storage "Хранилище" {
  rectangle "<<MySQL>>"{
```

```
  database "DBNote" as db1
 }
}
app -> db1
@enduml
```

Если для этого примера формироваться диаграмму компонентов, то компонентами будут только мобильное приложение и база данных. Тем не менее, можно конкретизировать архитектуру мобильного приложения компонентами-подсистемами. Тогда, к примеру, диаграмма компонентов может иметь следующий вид (рис. 22):

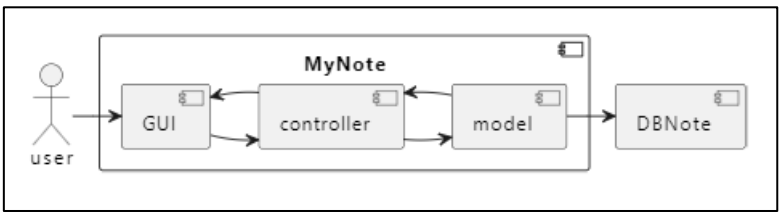

Рис. 22. Диаграмма компонентов для мобильного приложения

На данной диаграмме предполагается, что приложение MyNote было создано в архитектуре на основе выделения подсистем модели, контроллера и представления, где контроллер выполняет связующую функцию между моделью и представлением. Подсистема представления предназначена для формирования графического пользовательского интерфейса GUI и для акцента на этом на диаграмму помещен актор, взаимодействующий с подсистемой GUI. С отдельным компонентом базы данных взаимодействует только подсистема модели приложения MyNote. Так как каждая подсистема приложения может быть представлена отдельным артефактом (например, программным модулем), такая детализация понятия компонента может передать дополнительные акценты при анализе работы приложения на физическом уровне, учитывая видимость друг другу и принципы взаимодействия.

Программный код на PlantUML для реализации диаграммы компонентов, представленной на Рис. 22 выглядит следующим образом:

```
@startuml
component "MyNote" as app{
  component model
  component controller
  component GUI
}

component "DBNote" as db1
actor user

user->GUI
controller-> model
GUI -> controller
controller<- model
GUI <- controller
model -> db1

@enduml
```

ЛАБОРАТОРНАЯ РАБОТА. ЭТАП 5 КОМАНДНОГО ПРОЕКТА «СОЗДАНИЕ ДИАГРАММ ФИЗИЧЕСКОГО УРОВНЯ». МЕТОДИЧЕСКИЕ УКАЗАНИЯ

В рамках завершающего этапа проекта необходимо реализовать диаграммы физического уровня – диаграмму компонентов и диаграмму развертывания.

Предполагается, что эти диаграммы создаются по одной на весь проект, но могут быть декомпозицированы на несколько с учетом акцентов на различных подсистемах. Диаграмму компонентов также можно сделать в разных ракурсах – с точки зрения интерфейса как средства передачи данных (тогда требуемый и предоставляемый интерфейсы – это запросы на определенные данные), а также с точки зрения интерфейса как на сервиса (тогда интерфейс – способ запуска уда-

ленного приложения или вызова метода). Можно воспользоваться любой трактовкой, но так, чтобы явно выражены были используемые компоненты, которые бы определяли артефакты программной системы и распределение классов по этим артефактам. Например, артефакты можно определять как отдельный компонент, реализующий все функции подсистемы программного обеспечения. Таким образом, диаграмма компонентов должна стыковаться с диаграммой классов.

При построении диаграммы развертывания следует обратить внимание на распределение всех компонентов, обозначенных на диаграмме компонентов, по физическим узлам необходимого аппаратного обеспечения. Также следует сделать акцент на необходимых условиях, которым должно удовлетворять аппаратное обеспечение, например, установленной операционной системе или дополнительного системного или служебного программного обеспечения.

ЛИТЕРАТУРА

1) *Антамошкин О.А.* Программная инженерия. Теория и практика/ О.А. Антамошкин. – Красноярск: Сибирский федеральный университет, 2012. – 247 с.

2) *Антипов В.А.* Введение в программную инженерию/ В.А. Антипов, А.А. Бубнов, А.Н. Пылькин, В.К. Столчнев. – М.: КУРС ИНФРА-М, 2019. – 336 с.

3) *Затонский А.В.* Информационные технологии: разработка информационных моделей и систем / А.В. Затонский. – М.: РИОР ИНФРА-М, 2020. – 344 с.

4) *Золотухина Е.Б.* Управление жизненным циклом информационных систем (продвинутый курс) / Е.Б. Золотухина, С.А. Красникова, А.С. Вишня. – М.: НИЦ ИНФРА-М, 2017. – 119 с.

5) Конструирование программного обеспечения / под ред. Л.Г. Гагариной. – М.: ИНФРА-М, 2024. – 319 с.

6) Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон, Д. Влиссидес. – СПб: Питер, 2014. – 366 с.

7) *Миронов В.В.* Профессия «бизнес-аналитика». Краткое пособие для начинающих/ В.В. Миронов. – М.: Олимп-Бизнес, 2022. – 238 с.

8) Документация по PlantUML. – URL: https://plantuml.com/ru/ (дата обращения: 22.11.2024).

Учебное издание

Андрианова Анастасия Александровна

ПРАКТИКУМ ПО КУРСУ «ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ И ПРОЕКТИРОВАНИЕ»

Учебно-методическое пособие

Подписано к использованию хх. хх. 2025 г.
Научная библиотека им. Н.И. Лобачевского