Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Кубанский государственный технологический университет»
(ФГБОУ ВО «КубГТУ»)

Кафедра информационных систем и программирования

ПРОЕКТИРОВАНИЕ ПРОГРАММНЫХ КОМПЛЕКСОВ

Методические указания по выполнению лабораторных работ
для студентов всех форм обучения
направления 09.03.04 Программная инженерия

Краснодар
2019

Составитель**:** канд. техн. наук, доцент, Янаева Марина Викторовна

Проектирование программных комплексов: методические указания по выполнению лабораторных работ для студентов всех форм обучения направления: 09.03.04 Программная инженерия. / Сост. М.В. Янаева; Кубан. гос. технол. ун-т. Кафедра информационных систем и программирования. – Краснодар. 2019. – 111 с. Режим доступа: http://moodle.kubstu.ru (по паролю).

Методические указания по выполнению лабораторных работ дисциплины «Проектирование программных комплексов» для студентов составлены в соответствии с требованиями к обязательному минимуму содержания дисциплины, входящей в основную образовательную программу подготовки студентов направления 09.03.04 Программная инженерия государственного образовательного стандарта высшего профессионального образования, и в соответствии с рабочей программой дисциплины.

Рецензенты: Руководитель отдела телекоммуникаций Краснодарского регионального информационного центра сети «Консультант Плюс», канд.техн.наук. Н.Ф. Григорьев;
д-р. техн. наук, профессор каф. ИСП КубГТУ
В.Н. Марков

Содержание

Вводная лабораторная работа - Ознакомление и установка ПО.

Данный курс лабораторных работ предназначен для формирования понимания предназначения проектирования программных комплексов, а также для формирования первичных практических навыков.

Перед началом введения в практическую часть стоит дать пояснение зачем необходим текущий предмет, а также его наибольшую сферу влияния. Идея о формировании подходов для проектирования программных продуктов возникла, когда разработчики многих компаний столкнулись со сложностями масштабирования текущих проектов, а также параллельно возникающие ошибок, из-за чего компании теряли внутренние ресурсы.

Спустя время были сформированы рекомендации по проектированию высоко нагруженных программных комплексов. Большинство данных рекомендаций возникли из практик проектирования корпоративных продуктов, где программное обеспечение должно иметь длительный жизненный цикл, где программное обеспечение должно быть постоянно исполняемым и отказоустойчивым. Примерами областей бизнеса существования данного ПО являются:

- телекоммуникационные компании;
- компании банковской сферы;
- крупные сети розничной торговли.

Из приведённого списка можно сделать вывод, что проектирование ПО является лишь уделом крупного корпоративного сектора. Это не совсем так. Программные продукты в данном секторе бизнеса нуждаются в самом тщательном подходе к проектированию программных комплексов ввиду ответственности за те бизнес-процессы, которые автоматизирует данное ПО.

Практически любой относительно комплексный проект нуждается в первоначальном концептуальном проектировании. Таким образом можно определить весь необходимый функционал проекта, а также систематизировать все условия для команды разработчиков. Отличия в проектировании программного продукта для крупной корпоративной среды и любого независимого программного проекта заключается в детальности абстракций этапов проектирования. Для корпоративного продукта необходим спроектировать каждый процесс со всеми условиями, для иного проекта абстракция одного процесса может подразумевать несколько вытекающих или схожих процессов. Отсюда можно сделать вывод, что проектирование небольшого проекта допускает гибкость в определении абстракций, тогда как модель корпоративного продукта должна строго описывать все процессы, задействованные в реализации данной задачи.

В данном курсе проектирования программных комплексов будут использоваться подходы проектирования с помощью унифицированного языка моделирования (UML — Unified Modeling Language). Данная технология была разработана с целью построения концептуальных моделей, реализуемых в объектно-ориентируемой нотации. Иными словами, модели, разработанные с помощью UML, в основном, реализуются на таких объектно-ориентированных языках как Java, C#, C++ и Python. Данный факт

не означает, что проектирование на UML можно использовать только в связке с ООП языками. Команда разработчиков может принять решение реализовать программный компонент с помощью функциональной парадигмы. В таком случае UML отлично подойдёт для описания сценариев функционирования программных компонентов, но при таком сценарии не применимы модели классов программного компонента, а также диаграмма пакетов.

UML как язык без графических абстракций имеет строгую семантику описания структуры модели. На начальном этапе проектирования, запомнить все правила кодирования UML моделей является трудной задачей. Для фокусировки на проектирования модели, а не запоминания нотаций языка UML существуют решения для построения моделей с помощью элементов графического интерфейса. Из примеров данного ПО стоит отметить:

— StarUML;
— Archimate;
— Lucidchart.

Выше приведён лишь малый список из всех существующих программных решений. В рамках данного курса будет использоваться Umbrello для выполнения задания к лабораторному практикуму. Umbrello является кросс-платформенным программным обеспечением с открытым исходным кодом, что позволяет производить процесс моделирования системы вне зависимости от платформы, на которую нацелено Ваше будущее ПО. По ходу курса Вы ознакомитесь как структурировать различные диаграммы в единый проект Umbrello. Но для начала необходимо произвести установку самого ПО.

Версия для ОС семейства Windows можно скачать пройдя по ссылке и выбрав разрядность версии: https://download.kde.org/stable/umbrello/latest/

Версия OS X является портируемым образом системы MacPorts: https://www.macports.org/ports.php?by=name&substr=umbrello

Версия для Linux доступна для всех семейств операционных систем, поддерживающих пакетный менеджер Snap, ссылка на страницу в репозитории: https://snapcraft.io/umbrello

Umbrello позволяет проектировать модели с использованием графического интерфейса. Но использование графического интерфейса не единственный способ для проектирования UML диаграмм. Одним из самых распространённых плагинов для составления диаграмм внутри текстового редактора или среды разработки — PlantUML. Проектирование диаграмм производится с помощью специальных текстовых нотаций. Разработка модели доступна в нескольких форматах:

— .wsd;
— .pu;
— .puml;
— plantuml;
— .iuml.

В данном руководстве описываются действия по установке данного лпагина для среды разработки JetBrains Intellij IDEA и текстового редактора Visual Studio Code.

Для установки плагина в среде Intellij IDEA необходимо зайти в настройки (File → Settings), далее переходим на вкладку плагинов (Plugins) и в строке поиска вводим «PlantUML» и устанавливаем плагин с названием «PlantUML Integration» и устанавливаем его нажатием кнопки Install.

Для установки внутри текстового редактора Visual Studio Code необходимо перейти на вкладку расширений (Extensions, иконка с квадратами). В открывшейся строке поиска вводим PlantUML. Устанавливаем плагин на самой верхней позиции поиска.

Вопросы для самоконтроля

1. По какой причине возникла потребность в первоначальном проектировании ПО?
2. Для каких сфер бизнеса наиболее применимы практики проектирования ПО?
3. Что такое UML?
4. UML строго привязан к проприетарному ПО для моделированию на данном языке?
5. Какими способами можно производить моделирование в нотации UML?

Задание

Установите ПО Umbrello и PlantUML в соответствии с используемой операционной системой, используя ссылки на скачивание, приведённые в данной главе. Придумайте информационную систему, которую Вы будете проектировать в рамках данного лабораторного практикума или выберите пример приведённый из таблицы ниже.

Таблица 1 – Примеры информационных систем

№	Название информационной системы
1	Система управления умным домом
2	Онлайн сервис аренды автомобилей
3	Онлайн сервис бронирования отеля
4	Автоматизированная система розничной торговли
5	Система ведения прямых трансляций (стриминговый сервис)
6	Онлайн кинотеатр
7	Социальная сеть
8	Мессенджер
9	Интернет-магазин
10	Служба доставки еды
11	CRM-система (от англ. Customer Relationship Management – Система управления взаимоотношения с клиентами)
12	CMS-система (от англ. Content Management System – Система управления контентом)
13	Автоматизированная система медицинского учреждения
14	Интерактивная система онлайн-обучения
15	Автоматизированная система бронирования авиа-билетов

Лабораторная работа №1 - Начальный этап проектирования. Проектирование прецедентов, составление диаграммы

Прецеденты – это последовательные истории об использовании системы, которые широко используются для осмысления и формулировки требований. Стоит отметить, что основная задача проектировщика является не черчение диаграммы, а составление текста для описания сути работы каждого «сценария». Модель диаграммы прецедентов оперирует абстракциями верхнего уровня. В данной модели нет строго выявления абстракций функционала системы, на данном этапе выявляется самый базовый функционал системы. «Системными» исполнителями могут быть как отдельно разработанные классы или пакеты, или могут быть составными из нескольких элементов системной архитектуры. Данная диаграмма ориентирована на пользователей системы, на данном этапе проектирования выявляются ключевые особенности взаимодействия пользователей с системой, а также выявляются базовые требования к реализации системного функционала.

Исполнителем (actor) – называется сущность, обладающая поведением, например, человека (идентифицируемого по роли, к примеру, кассира), компьютерную систему или организацию.

Сценарий (scenario) – это специальная последовательность действий или взаимодействий между исполнителем и системой. Его иногда также называют *экземпляром прецедента* (use case instance). Это один конкретный пример сценарий использования системы либо один проход прецедента, например, сценарий успешной регистрации на ресурсе при помощи аккаунтов социальных сетей.

Стоит отметить, каким образом строятся зависимости между сценариями и исполнителями. Обычный сценарий связывается с сущностью исполнителя при обычной прямой линии без стрелочки. Расширение сценария делится на *включаемые прецеденты* (include), *расширяемые* (extend), а также *генирализуемые.*

Включаемые прецеденты – это прецеденты, которые наступают одновременно с возникновение определённого сценария. Например, после создания комнаты обязательно произойдёт событие по перенаправлению пользователя на страницу комнаты.

Расширяемые прецеденты – это прецеденты, которые могут осуществляться при работе определённого сценария, но не являются обязательными к исполнению. Для примера можно привести сценарий «редактирования плейлиста» при выполнении сценария «управления комнатой». Администратор может изменить очередь треков, но это не всегда происходит во время его управления комнатой.

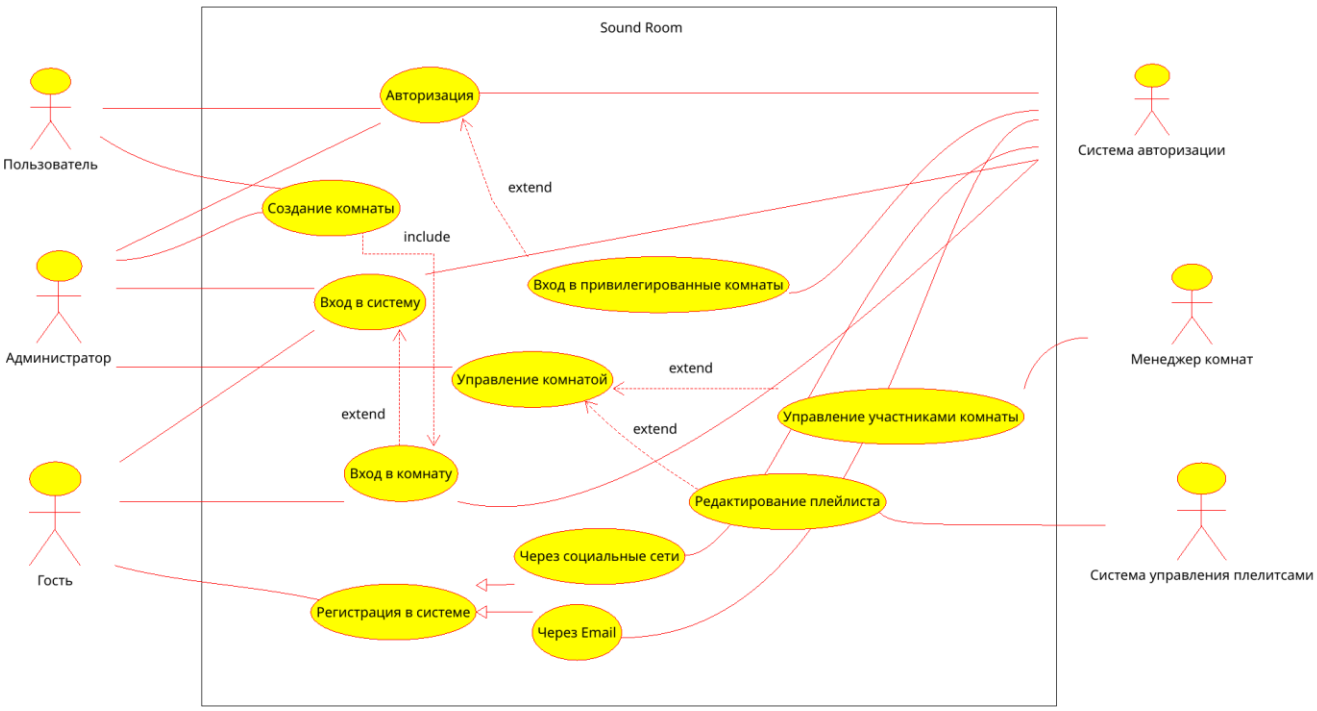


Рисунок 1.1 – Диаграмма прецедентов проекта Sound Room

Роли актёров расположены по бокам объектной модели. Исполнители-пользователи, проводящие основное взаимодействие с системой расположены слева от модели системы. Исполнители, являющиеся системными сущностями программного обеспечения расположены справа от объектной модели.

Пользователь - ключевой исполнитель данной системы. Абсолютно все исполнители всех ролей пользователей (пользователь, администратор и гость) имеют возможность входа в систему по заранее сохранённым персональным данным или войти как анонимный гость. Каждый пользователь имеет возможность производить авторизацию в систему для осуществления всех остальных действий, поэтому данный прецедент выведен на верхний уровень модели прецедентов. После авторизации пользователь имеет возможность авторизации в привилегированные комнаты, в случае если пользователь обладает данным правом. Находясь в комнате системы все пользователи имеют доступ к прослушиванию аудио контетна, а привилегированные пользователи имеют права добавлять собственные аудиодорожки в плейлист комнаты.

Администратор - пользователь, обладающий правом управления активностью внутри выделенной комнаты. Управление комнатой заключается в возможности добавления или удаления треков из очереди плейлиста комнаты, а также управлением участниками:

- выдача прав пользователям комнаты на добавление треков;
- удаление участников комнаты;
- добавление пользователей в чёрный список комнаты.

Гость - неавторизованный пользователь системы. Данный исполнитель может производить вход в открытые комнаты, но не имеет права вести любую деятельность в комнате кроме прослушивания аудио контента. Любой незарегистрированный в системе пользователь имеет возможность произвести регистрацию в системе удобным ему способом – через персональный адрес электронной почты или используя данные аккаунтов социальных сетей.

Система авторизации – системный исполнитель, в обязанности которого входит проверка корректности вводимых данных пользователей при авторизации в системе, доступу по паролю в закрытые (привилегированные), выдаче анонимным пользователя временных данных для идентификации, а также осуществление регистрации новых пользователей в системе. Для регистрации новых пользователей данный исполнитель производит новые записи в базу данных системы, а также имеет доступ к системным интерфейсам социальных сетей для возможности регистрации новых пользователей используя персональные данные социальных сетей.

Менеджер комнат – к задачам данного системного исполнителя относятся реализация программного функционала для ведения администраторами функций управления пользователями комнат.

Система управления плейлистами – системный исполнитель, реализующий систему очередей треков в каждой комнате. Также в задачи данного исполнителя входит реализация программного функционала для предоставления возможности пользователям добавлять новые треки в очередь, а администраторами предоставить возможность удалять треки из очереди.

Диаграмма описания прецедентов ставит задачу к систематизации требований к первоначальному функционалу системы, а также выделению ролей данной системы, поэтому данную модель строят на первых стадиях проектирования. Для удобства проектировщика, текстовое описание прецедентов носит свободный характер, но при этом проектировщик обязуется донести до разработчиков начальное понимание функционирования системы. Главной задачей диаграммы прецедентов является понятность и читаемость. Не стоит выделять небольшую задачу или функционал системы в отдельный прецедент, в таком случае модель приобретёт избыточность и потеряет свойство читаемости.

Для того чтобы создать диаграмму прецедентов в системе Umbrello необходимо создать новый проект, или продолжить существующий.

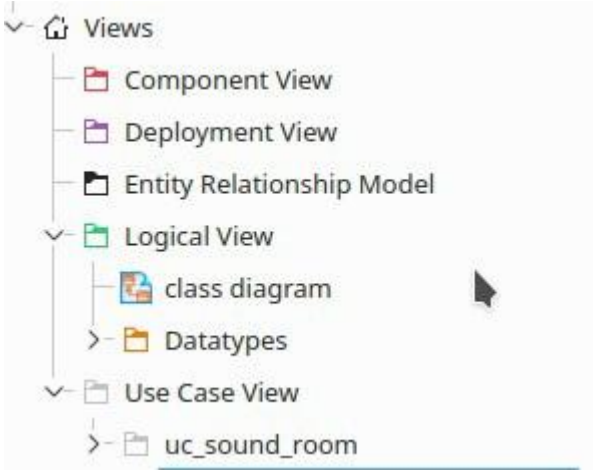


Рисунок 1.2 – Структура проекта Umbrello

Проектировать диаграммы прецедентов необходимо внутри каталога «Use Case View». Для лучшей структуризации проекта, для каждой диаграммы необходимо создавать собственный подкаталог, в данном примере – «uc_sound_room». Следующим шагом необходимо добавить объект диаграммы. Щёлкнув правой кнопкой мыши по созданному подкаталогу, в контекстном меню перейдите на пункт «New» и в раскрывшемся списке выберете пункт «Use Case Diagram...».

В графическом определении диаграммы прецедентов существую 3 сущности: проектируемая система, сценарии взаимодействия, участники. В Umbrello система не выделяется в отдельный графический объект в дереве структуры проекта, поэтому для выделения рабочей области системы необходим добавить фигуру прямоугольника (рис. 1.3).

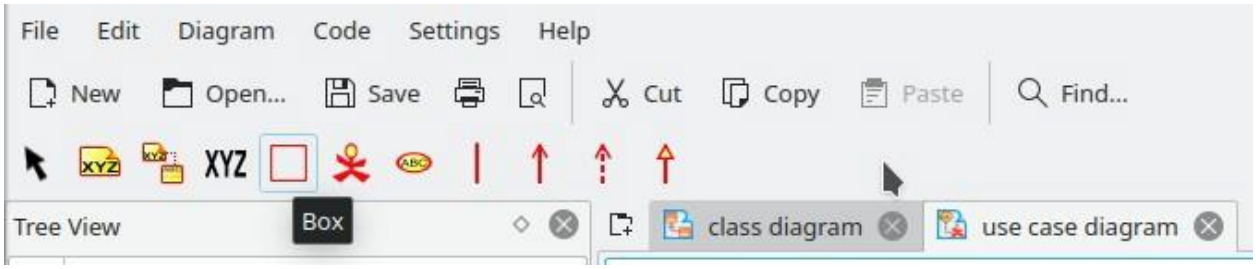


Рисунок 1.3 – Добавление контейнера рабочего пространства

Внутри подкаталога проекта для проведений лучшей структуризации проекта необходим выделить ещё 2 подкаталога для «сценариев» (use cases) и для «исполнителей» (actors). Добавление новых элементов в соответствующие категории осуществляется аналогично добавлению диаграммы с выбором соответствующих элементов, «Actor» — для исполнителей, «Use case» - для сценариев.

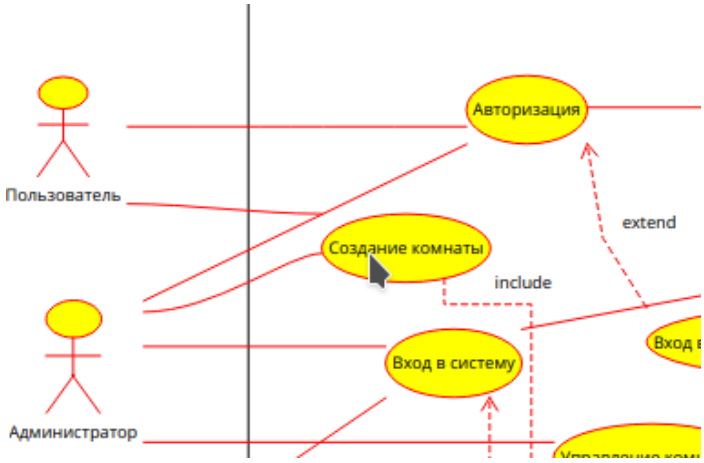


Рисунок 1.4 – Ассоциация исполнителей и сценариев

После определения исполнителей и сценариев необходим определить зависимости и ассоциации между ними. Чтобы установить ассоциацию необходимо нажать на кнопку ассоциации (Association) (рис. 5), затем нажать левой кнопкой мыши по первому объекту (в данном случае исполнитель) и по второму (сценарию).

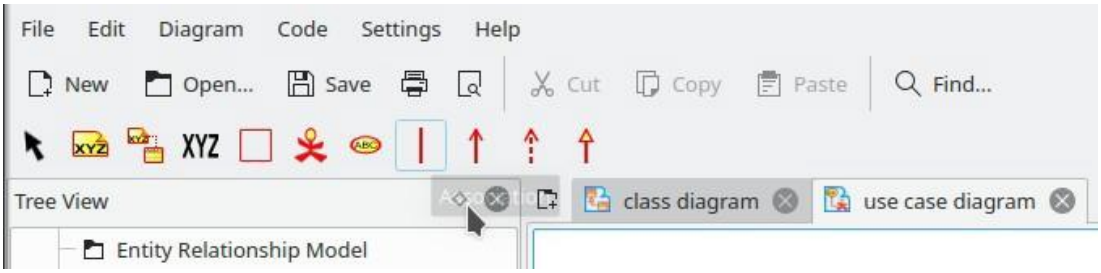


Рисунок 1.5 – Кнопка ассоциации

Проведённая автоматически линия ассоциации может быть довольно короткой и наслаиваться на другие элементы, из-за чего визуально может вызвать трудность в чтении. Для устранения данной проблемы, построенную линию необходимо перевести в режим отображения «кривой» (Spline). Для этого необходим выделить соответствующую зависимость левой клавишей мыши, нажатием правой клавиши вызвать контекстное меню вы выбрать пункт «Layout», в отобразившемся списке выбрать пункт «Spline».

Следующим этапом происходит выделение «зависимых» (include) и «ситуативных» (extend) сценариев. Зависимые прецеденты — это те прецеденты, которые всегда срабатывают при выполнении выполнении прецедента, от которого они зависят. На примере проекта Sound Room есть сценарий «создания комнаты», после срабатывания которого, срабатывает сценарий перемещения пользователя в интерфейс новой комнаты (рис. 6).

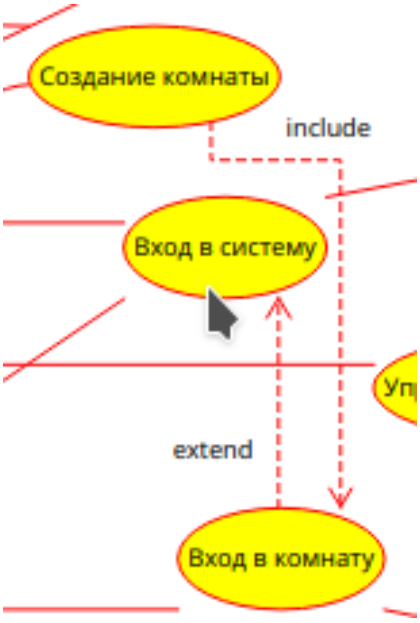


Рисунок 1.6 – Зависимый и ситуативный прецеденты

Для того чтобы создать зависимость такого рода необходимо в верхнем меню элементов нажать на кнопку «Dependency» (рис. 1.7), нажать на зависимый элемент, а вторым щелчком нажать на элемент от которого он будет зависеть. Чтобы добавить обозначение, что данная связь относится к типу «include», необходим добавить текстовую метку рядом с линией ассоциации. Для этого необходимо нажать на кнопку текстовой метки «XYZ», нажать на область, где будет находиться метка и дать название метки (в данном случае include).

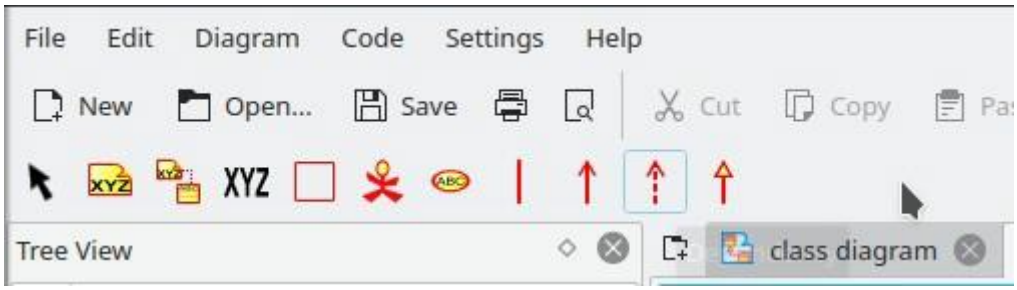


Рисунок 1.7 – Панель графических элементов

Аналогичным методом производится добавление ситуативных ассоциаций (extend). Ситуативные прецеденты могут срабатывать при определённых обстоятельствах, после срабатывания сценария, с которым они ассоциируются. На примере системы Sound Room это можно увидеть по прецедентам «вход в систему» и «вход в комнату». Не каждый пользователь, который зашёл систему, перманентно переходит в определённую комнату. Пользователь может выйти из систему или перейти на иной раздел системы. Поэтому данный случай может быть ситуативным.

Стоит отметить, что ассоциация зависимых и ситуативных прецедентов строится пунктирной линией от зависимого/ситуативного к прецеденту, который должен быть совершён для их исполнения. Следующим элементом построения диаграммы прецедентов является «обобщение» (generaliztion). Данный процесс связан с разделением сценариев и исполнителей на подгруппы. Когда сценарий имеет разные пути для достижения результата. На примере Sound Room — «Способ регистрации» в системе: через email или через аккаунт в социальных сетях (рис. 1.8).

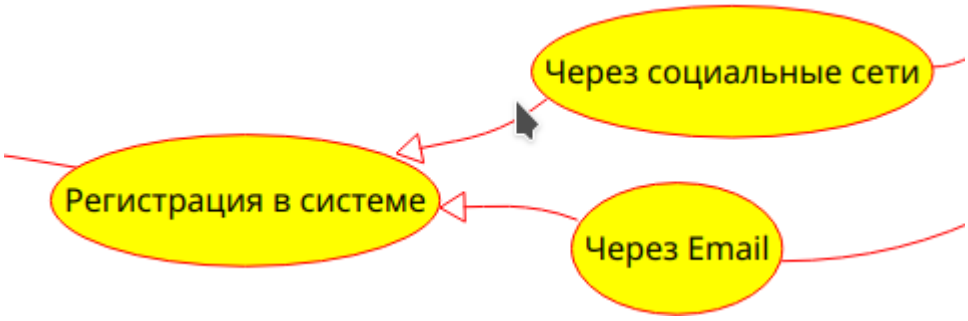


Рисунок 1.8 – Обобщение прецедентов

Для того чтобы построить ассоциацию такого типа необходим из меню графических элементов нажать на кнопку «Generalization» и соединить разветвлённые прецеденты с образующим их прецедентом.

Вопросы для самоконтроля

1. Какие основные объекты используются для построения диаграммы прецедентов?
2. Дайте определение включаемым прецедентам.
3. Дайте определение расширяемым прецедентам.
4. Верно ли утверждение, что роль *исполнителей* могут осуществлять только люди, участвующие в жизненном цикле работы системы?
5. Дайте определение генерализуемым прецедентам.
6. Верно ли утверждение, что диаграмма прецедентов представляет низкоуровневую абстракцию работы системы?
7. Построение прецедентов ограничивается диаграммой?
8. Для каких целей осуществляется проектирование прецедентов?
9. Чем является сценарий в диаграмме прецедента?
10. По какому принципу организованы исполнители на диаграмме?

Задание

С проектируйте диаграмму прецедентов, также их описание по системе, выбранной Вами ранее. На данном этапе Вам необходимо выделить все основные процессы взаимодействия, пользователей с вашей системой.

Лабораторная работа №2 - Проектирование модели предметной области

Модель предметной области – одна из важнейших моделей объектно-ориентированного анализа. Она отображает основные (с точки зрения моделирующего) классы понятий (концептуальные классы) предметной области. На рис. 1 представлена схема взаимодействий и отношений на начальных этапах проектирования.

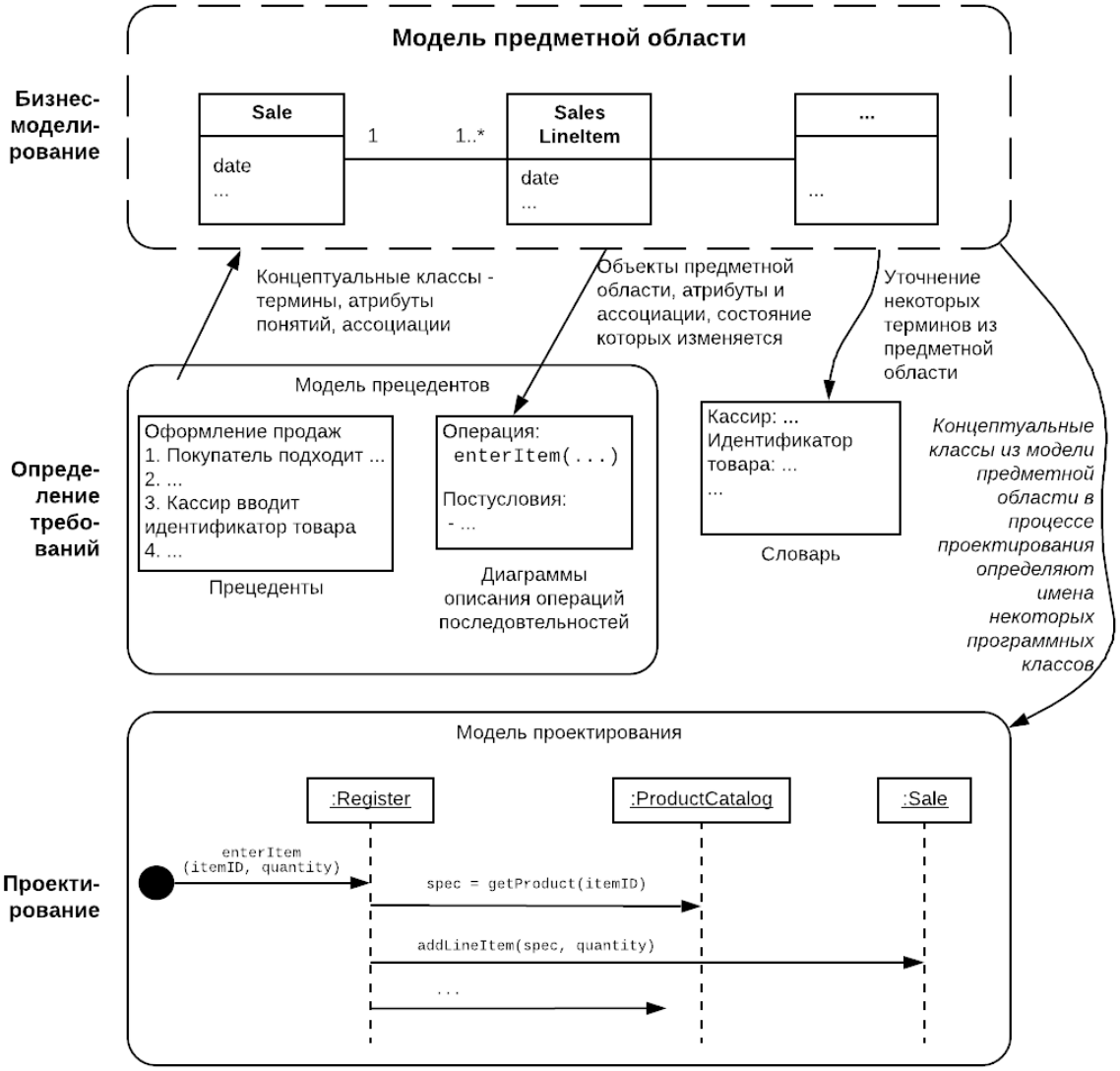


Рисунок 2.1 – Схема представления отношений и связей начального этапа проектирования

Как представлено на рис. 2.1, каждой итерации соответствует своя модель предметной области, отражающая реализуемые на данном этапе сценарии прецедентов. Таким образом, модель предметной области эволюционирует в процессе разработки системы. Модель предметной области связана с моделью проектирования, особенно программными объектами, относящимися к уровню предметной области, т.к. объекты данной модели являются прообразом полноценных программных

компонентов. Модель предметной областью является промежуточным звеном между моделью прецедентов, которая формирует требования к системе, а также определяет роли и иерархию пользователей и исполнителей, и моделью низкоуровневого проектирования (диаграмма последовательностей, диаграмма классов).

На рис. 2.2 представлен фрагмент модели приложения Sound Room в виде диаграммы классов. Из рисунка ясно, что с точки зрения предметной области *концептуальными классами* (conceptual class) являются User (пользователь), Lobby (страница комнаты размещения треков), Queue (очередь треков) и Track (трек) и Black List (чёрный список или бан-лист). Как видно из диаграммы, эти понятия связаны между собой, и понятию Lobby соответствуют определённые названия комнаты и прав доступа к ней. С точки зрения проектирования сущностей объектной модели, свойства объекта реализуются через свойства класса. Насколько можно видеть из данной модели, каждый объект имеет имя, связь, а также несколько характеризующих его параметров.

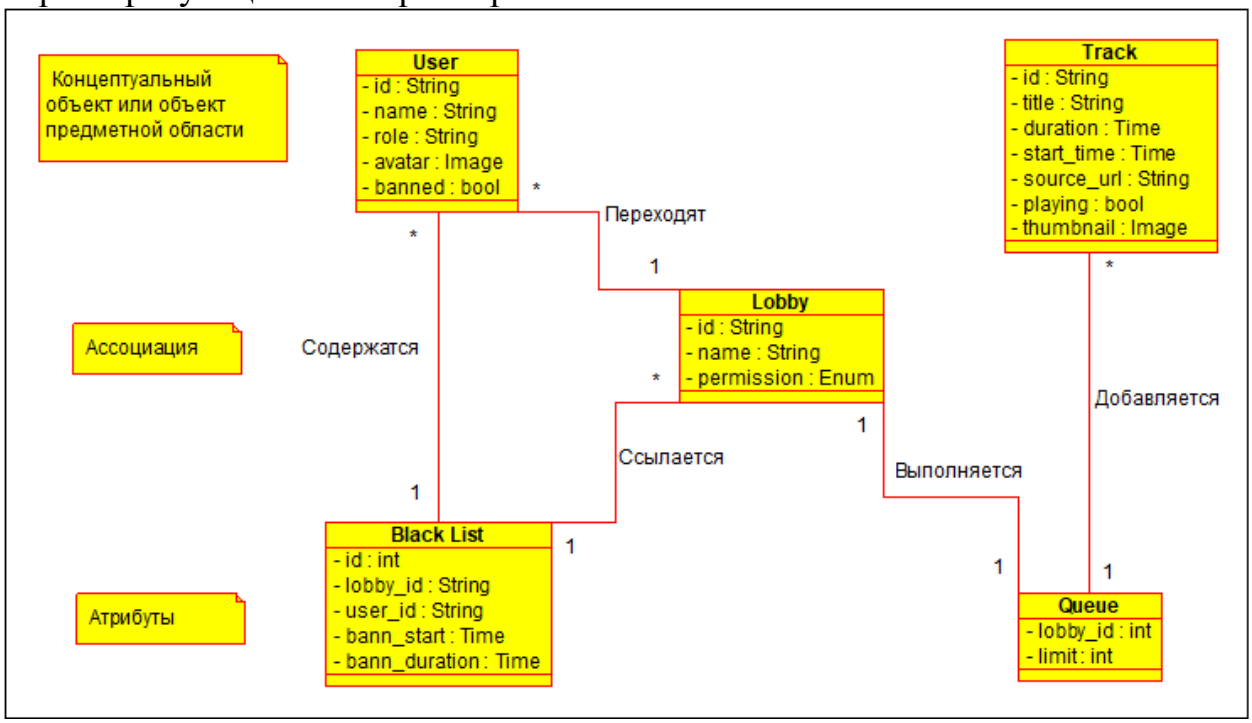


Рисунок 2.2 – Фрагмент модели предметной области – визуальный словарь

Диаграмма классов в обозначении UML обеспечивают концептуальную перспективу модели.

Идентификация набора концептуальных классов – основная задача объектно-ориентированного анализа. На начальных этапах построение модели предметной области у разработчика может занять всего несколько часов, но на последующих этапах, когда требования к системе определяются более чётко, уточнение модели предметной области потребует значительно больше времени.

При построении диаграммы модели предметной области избегайте попыток построения полной модели на начальных этапах разработки. Такой

стиль присущ каскадному процессу разработки, когда модель предметной области строилась на этапе анализа без учёта обратной связи.

Основной составляющей объектно-ориентированного анализа или исследования является декомпозиция проблемы на отдельные классы понятий (концептуальные классы) или объекты.

Модель предметной области – это *визуальное* представление концептуальных классов или объектов реального мира в терминах предметной области. Такие модели называют также *концептуальными моделями*, или *объектными моделями анализа*.

Объектные модели анализа также связаны с моделями взаимоотношений концептуальных сущностей, отображающими только концептуальное представление предметной области, однако интерпретируемыми в более широком смысле как модели данных для разработки баз данных. Модели предметной области – это не модели данных.

Основная идея модели предметной области – это представление классов понятий реального мира, а не программные компоненты. Это *не* набор диаграмм, описывающих программные классы или программные объекты с их обязанностями.

В ином контексте модель предметной области – это один из артефактов, создаваемых в рамках дисциплины бизнес-моделирования. Более точно: модель предметной области является конкретизацией более общего понятия *модели бизнес-объектов* (business object model - BOM), обеспечивающей представление понятий, играющих важную роль в данной предметной области, например, связанные с розничной торговлей. В более сложных случаях модель BOM является мультидисциплинарной моделью и описывает предметную область, включающую множество подобластей.

На языке UML модель предметной области представляется в виде набора *диаграмм классов*, на которых не определены никакие операции (работа методов, наследование). Модель предметной области может отображать следующее:

- объекты предметной области или концептуальные классы;
- ассоциации между концептуальными классами;
- атрибуты концептуальных классов.

Модель предметной области – это результат визуализации понятий реального мира в терминах предметной области, а не программных элементов, таких как классы Java или C#. Следовательно, в модели предметной области не используются следующие элементы:

1. Артефакты программирования наподобие окон или баз данных, если только разрабатываемая система не является моделью программного средства, например моделью графического интерфейса пользователя.
2. Обязанности или методы.

Иллюстрационная часть модели предметной области отведена концептуальным классам или словарю предметной области. Неформально: *концептуальный класс* – это представление идеи или объекта. Если говорить более строго, то концептуальный класс можно рассматривать в терминах символов, содержания и расширения.

- *символы* (symbol) – слова или образы, представляющие концептуальный класс.
- *содержание* (intension) – определение концептуального класса.
- *расширение* (extension) – набор примеров, к которым применим концептуальный класс.

Модель предметной области не является *моделью данных* (data model), к которой по определению относятся данные, сохраняемые на постоянном носителе, поскольку неизвестно, нужно ли сохранять в базе данных информацию о классах. Концептуальные классы могут вообще не содержать атрибутов и играть в предметной области чисто поведенческую, а не информационную роль.

Для создания модели предметной области на данной итерации выполните следующие действия:

1. Выделите концептуальные классы.
2. Отобразите их в модели предметной области в виде классов на диаграмме UML.
3. Добавьте необходимые ассоциации и атрибуты.

Основной проблемой построения модели предметной области является выделение концептуальных классов. Для упрощения и систематизации данного процесса можно воспользоваться двумя подходами:

1. Повторное использование или модификация существующих моделей.
2. На основе выделения существительных.

В *первом случае* стоит обратиться к уже спроектированным моделям предметных областей. Это может быть модель, из другого проекта или модель, описываемая в учебных руководствах или источниках документирования сторонних систем. Данный способ является самым простым при проектировании данной модели, но не стоит вслепую копировать сущности. Сперва стоит обратиться к диаграмме прецедентов для более точной формализации концептуальных классов.

Второй способ идентификации концептуальных классов основан на лингвистическом анализе. Он состоит в выделении **существительных** из текстовых описаний предметной области и их выборе в качестве кандидатов в концептуальные классы или атрибуты. Такой способ часто называют *моделированием естественного языка*.

Данный метод следует использовать с осторожностью. Между существительными и концептуальными классами нет взаимно однозначного соответствия, а слова естественного языка могут иметь по несколько значений. Для реализации подобного подхода удобно использовать *развёрнуты описания прецедентов*, например основной сценарий прецедента *Оформление продажи*.

Основной успешный сценарий (или основной процесс)

1. **Покупатель** подходит к **кассовому аппарату POS-системы** с выбранными **товарами**.

2. **Кассир** открывает новую **продажу**.
3. **Кассир** вводит **идентификатор товара**.
4. Система записывает **наименование товара** и выдаёт его **описание**, **цену** и **общую стоимость**. **Цена** вычисляется на основе набора правил.

Кассир повторяет действия описанные в пп. 3-4 для каждого наименования товара.

5. Система вычисляет общую стоимость **покупки** с **налогом**.
6. **Кассир** сообщает покупателю **общую стоимость** и предлагает оплатить **покупку**.
7. **Покупатель** оплачивает **покупку**, система обрабатывает **платёж**.
8. Система регистрирует **продажу** и отправляет информацию о ней внешней **бухгалтерской системе** (для обновления бухгалтерских документов и начисления **комиссионных**) и **системе складского учёта** (для обновления данных).
9. Система выдаёт **товарный чек**.
10. **Покупатель** покидает магазин с **чеком** и **товарами** (если он что-то купил).

Расширения (или альтернативные потоки)

…

7а. Оплата наличными

1. Кассир вводит предложенную покупателем **сумму**.
2. Система вычисляет положенную **сдачу** и открывает **кассу с наличностью**.
3. **Кассир** складывает полученные деньги и выдаёт **сдачу покупателю.**
4. Система регистрирует **платёж** наличными.

Модель предметной области – это визуализация важных понятий из словаря предметной области. Откуда брать необходимые термины? Из описания прецедентов. Эти описания – богатый источник для идентификации существительных.

Одни из этих существительных могут быть представлены в виде концептуальных классов, другие могут представлять концептуальные классы не имеющие отношения к данной итерации (например, «бухгалтерская система» или «комиссионные»), а третьи – в виде атрибутов этих концептуальных классов.

Недостатки данного подхода является выразительность естественного языка. Для описания одного и того же концептуального класса или атрибута могут использоваться различные существительные, и в то же время некоторые существительные могут иметь по нескольку значений.

В процессе разработки модели предметной области необходимо идентифицировать связи (ассоциации) между концептуальными классами, удовлетворяющие информационным потребностям разрабатываем на

текущей итерации сценариев, а также выделить те из них, которые способствуют лучшему пониманию модели предметной области.

Ассоциация (association) – это отношение между классами (или точнее, экземплярами этих классов), отражающая некоторые значимые и полезные связи между ними.

В языке UML ассоциации описываются как «семантические взаимосвязи между двумя или несколькими классификаторами и их экземплярами».

В модель предметной области целесообразно включать следующие ассоциации:

- ассоциации, знания о которых нужно сохранять в течении некоторого периода (важные ассоциации);
- ассоциации, производные от содержащихся в списке стандартных ассоциаций.

Ассоциация обозначается проведённой между классами линией, с которой связано определённое имя, начинающееся с большой буквы (рис. 2.2).

На концах линии, которая обозначает ассоциацию, могут содержаться выражения, определяющие количественную связь между экземплярами классов.

Обычно ассоциация является двунаправленной. Это означает, что от одного объекта любого типа возможен логический переход к другому объекту. Такой переход является абсолютно абстрактным. Он не определяет тип взаимосвязей между программными сущностями.

В случае добавления дополнительной стрелки рядом с именем ассоциации указывает, в каком направлении нужно читать её имя. Она не определяет направление видимости или перемещения. Если такая стрелка отсутствует, то имена ассоциаций следует читать с использованием общепринятых соглашений, а именно – слева направо и сверху вниз. Однако в языке UML в явной форме это правило отсутствует.

Кратность (multiplicity) определяет, сколько экземпляров класса A может быть ассоциировано с одним экземпляром класса B (рис. 2).

Например, один экземпляр класса *Queue (очередь)* может быть ассоциирован с несколькими (ни с одним или с несколькими, что отображается символом «*») экземпляра класса *Track (аудио-трек)*.

Некоторые примеры кратных ассоциаций представлены на рис. 2.3.

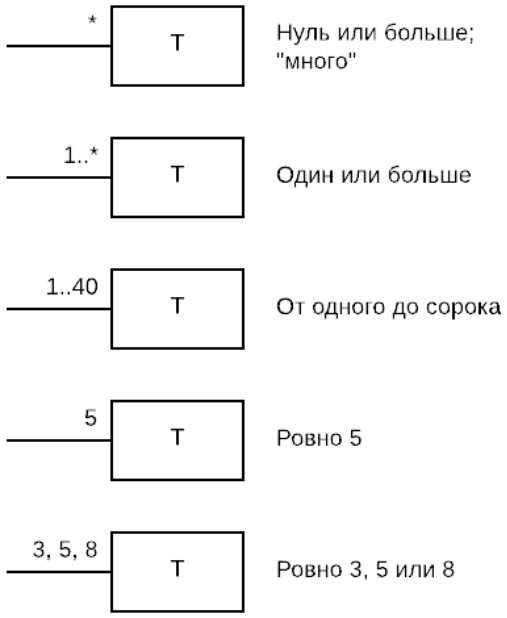


Рисунок 2.3 – Значения кратности

Значение кратности определяет, сколько экземпляров одного класса может быть корректно связано с экземпляром другого класса в некоторый конкретный момент, а не на всём промежутке времени. Например, некий подержанный автомобиль через определённые промежутки может быть многократно продан одному и тому же продавцу подержанных автомобилей. Однако в каждый конкретный момент этот автомобиль продан только *одному* владельцу. Автомобиль не может быть продан *нескольким* владельцам одновременно.

Значение кратности зависит от интересов разработчика модели и программы, поскольку ограничение предметной области должны быть отражены в программном обеспечении.

Рассмотрим способ построение диаграмму модели предметной области в среде Umbrello. Для диаграммы предметной области нет отдельного раздела в инструментах Umbrello, как это и не предназначено во многих других инструментах. Для построения диаграмму предметной области воспользуемся инструментами диаграммы классов. Для этого необходим в разделе Logical View добавить новую диаграмму классов, вызвав правой кнопкой мыши контекстное меню и выбрав New → Class Diagram (рис. 2.4).

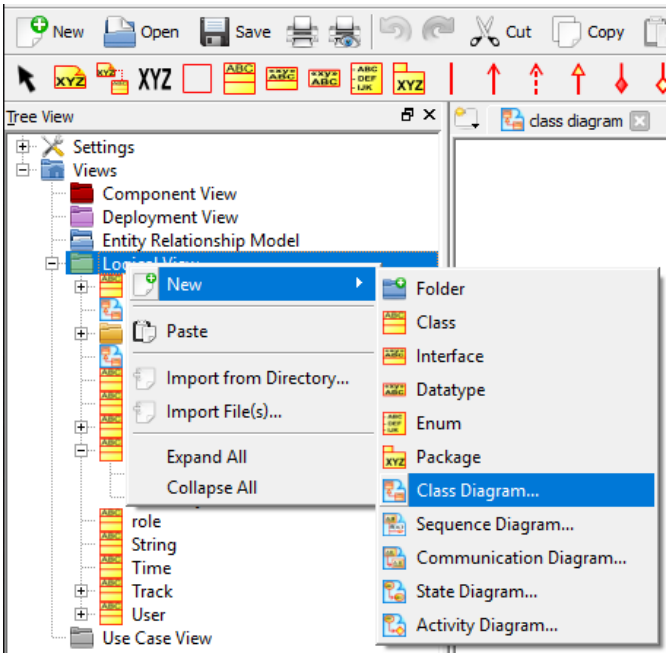


Рисунок 2.4 – Создание новой диаграммы предметной области

Чтобы не путаться в именах диаграммы директории Logical View хорошей практикой будет давать названия новым диаграммам на английском языке с названием типа диаграммы, а также постфиксом (если необходимо) с указанием проекта или области проектирования. Для удобства назовём новую диаграмму Domain Model (от англ. – предметная область) (рис. 2.5).

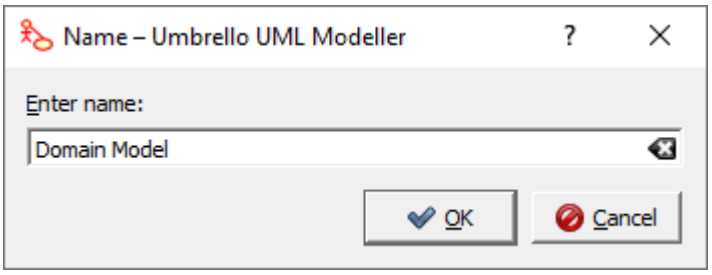


Рисунок 2.5 – Именование новой диаграммы

Создав новую диаграмму, двойным щелчком левой кнопки мыши, переходим на рабочую область диаграммы. Прежде чем создавать концептуальные классы модели, необходим добавить контейнер, в рамках которого будет производиться графическое проектирование. Производится это путём нажатия кнопки «Box» графической панели элементов Umbrello и после перевода курсора в режим добавления контейнеров на рабочую область, добавить 1 щелчком левой клавиши мыши по любому месту рабочей области.

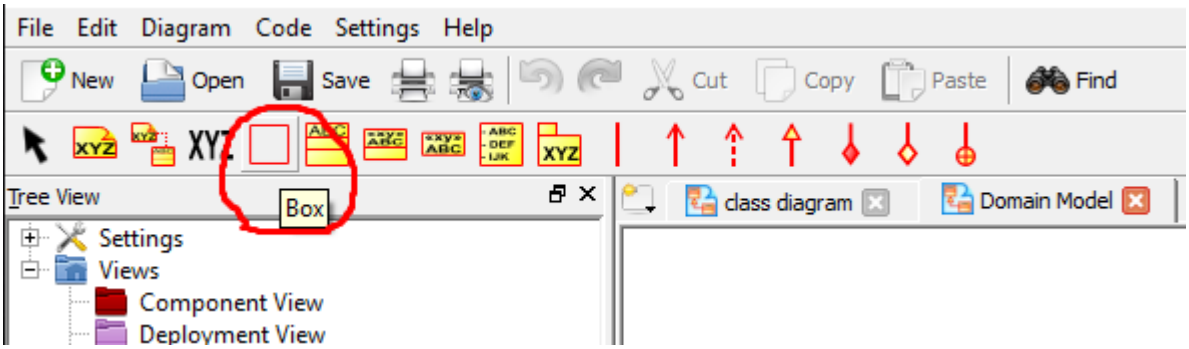


Рисунок 2.6 – Выбор контейнера диаграммы

Теперь есть возможность добавлять новые концептуальные классы. Производится это путём нажатия левой клавиши мыши (ЛКМ) по элементу «Class» на панели элементов Umbrello (рис. 2.6). После перехода курсора в режим добавления элемента, щелчком ЛКМ добавить элемент в область контейнера. В появившемся окне ввести новое имя концептуального класса согласно Вашей системе. Чтобы добавить параметры концептуальном классу, необходимо правой клавишей мыши (ПКМ) нажать на объект класса и выбрать пункт «Properties» (рис. 2.7).

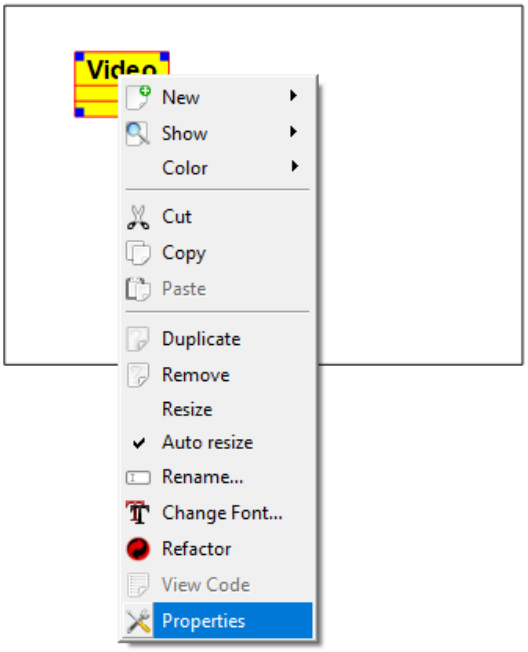


Рисунок 2.7 – Настройка объекта класса

В появившемся коне необходим выбрать пункт «Attributes». Далее нажать на кнопку «New Attribute» (рис. 2.8).

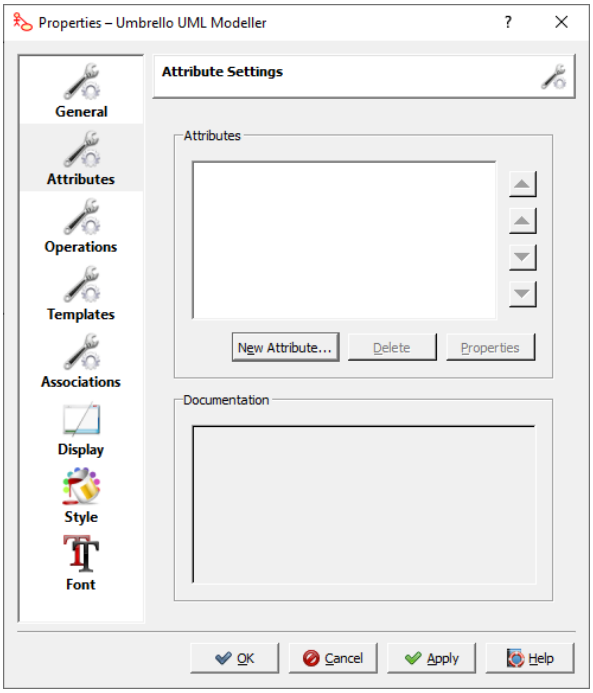


Рисунок 2.8 – Окно настройки объекта класса

В появившемся окне в поле «Name» необходимо ввести имя свойство, и, если есть необходимость в поле «Type» выбрать тип свойства или добавить собственный (рис. 2.9).

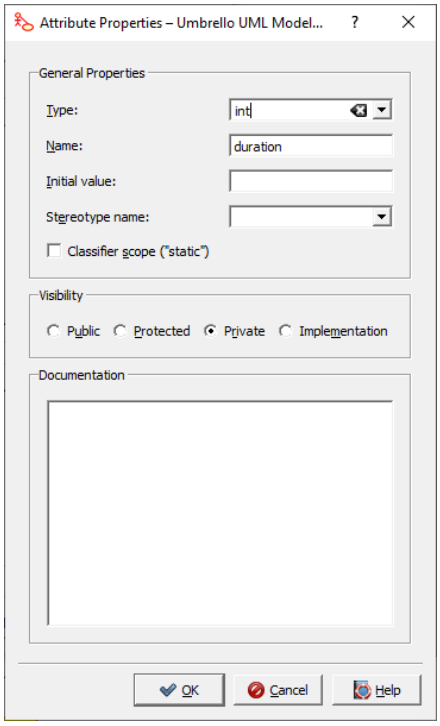


Рисунок 2.9 – Определение нового атрибута

После определение концептуальных классов и определения у них свойств, необходим провести ассоциации между соответствующими классами. Для этого необходим выбрать элемент «Association» в панели элементов Umbrello (рис. 2.10).

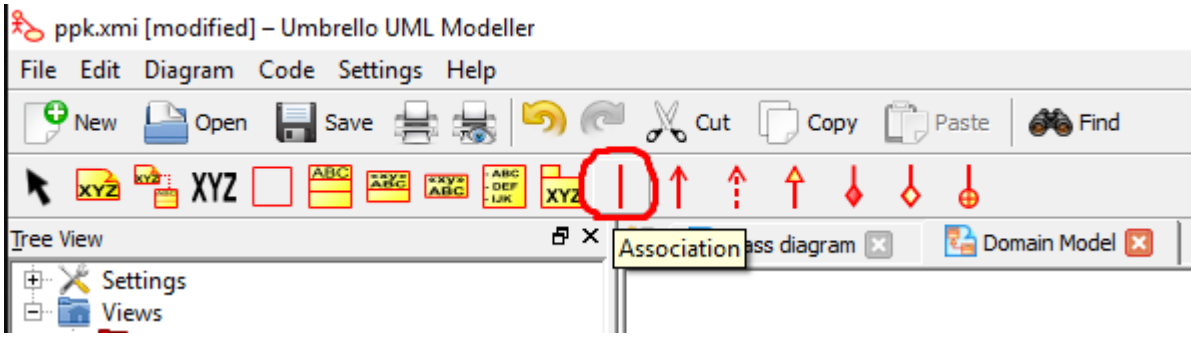


Рисунок 2.10 – Элемент ассоциации панели Umbrello

После перевода курсора в режим проведения ассоциаций, провести линию от одного класса к другому (рис. 2.11).

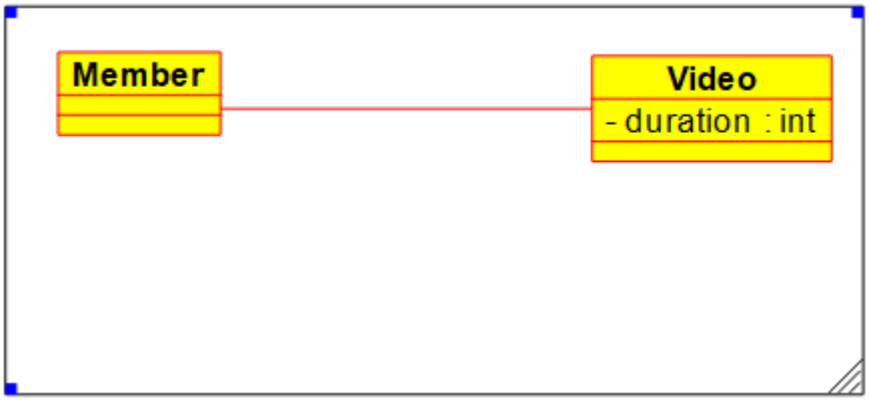


Рисунок 2.11 – Проведение ассоциации

Следующим шагом необходим добавить кратность данной ассоциации, а также дать ей имя. Производятся все данные действия путём добавления текстовых элементов вблизи элементов.

Добавление метки производится аналогично добавлением подобных элементов из панели графических элементов Umbrello (рис 2.12).

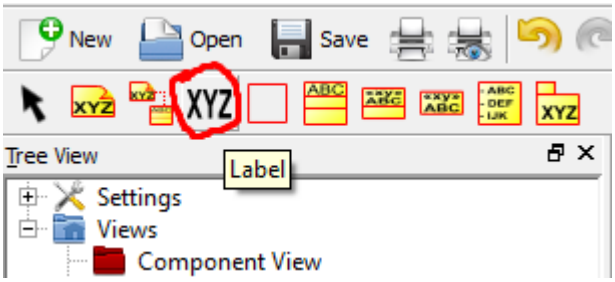


Рисунок 2.12 – Добавление текстовой метки

Введя имя ассоциации, а также кратность ассоциации, диаграмма должна иметь следующий вид (рис. 2.13):

Рисунок 2.13 – Готовая ассоциация концептуальных классов

Подобным образом производится добавления всех остальных подобных элементов диаграммы.

Вопросы для самоконтроля

1. С какой моделью связана модель предметной области?
2. Какая задача стоит перед моделью предметной области?
3. Модель предметной области является начальной программной реализации функционала будущей системы, так ли это?
4. Дайте определение модели предметной области.
5. Верно ли утверждение, что модель предметной области используется также для определения модели данных?
6. Что такое концептуальный класс?
7. Всегда ли концептуальные классы должны содержать атрибуты?
8. Какие существуют практики для определения концептуальных классов?
9. Что такое ассоциация концептуальных классов?
10. Обязательно ли указывать стрелку для проведения ассоциации между концептуальными классами?
11. Что такое кратность ассоциации?
12. Какой тип диаграммы следует использовать чтобы проектировать диаграмму предметной области?
13. Чем является результат построения модели предметной области?

Задание

Постройте модель предметной области в среде Umbrello в соответствии с вариантом Вашей системы из таблицы «Лаб_0» или используйте собственную систему для моделирования. Построения диаграмм должно производиться в рамках единого проекта Umbrello.

Лабораторная работа №3 - Проектирование диаграмм последовательностей.

На этапе, когда необходимо моделировать хронологическую последовательность действий пользователя и поведения системы, применяются *диаграммы последовательности*.

Диаграмма последовательности отображает взаимодействие между объектами в хронологическом порядке, т.е. определены на диаграмме в том порядке, в котором выполняется вызов событий во времени. Термины или сценарии из диаграммы видов деятельности также могут быть задействованы при проектировании диаграммы последовательности. Аналогично может быть задействовано при обратном сценарии проектировки. Данный вид диаграмм имеет более низкоуровневый вид описания взаимодействия процессов внутри системы. Стоит различать 2 типа диаграмм последовательности:

– *стандартные диаграммы последовательности* – данный вид диаграмм используется для моделирования хронологических процессов без привязки к конкретному языку программирования и первоначальной реализации классов и методов;

– *системные диаграммы последовательности* – данный вид диаграмм ставит целью первоначального определением классов и методов в контексте их хронологического вызова при определённых действиях пользователя.

В рамках данной лабораторной работы будут рассмотрены стандартные диаграммы последовательностей.

Рассмотрим нотации, применяемые в стандартных диаграммах последовательностей.

Исполнитель – представляет в UML нотации диаграммы последовательностей тип роли, который взаимодействует с системой и её объектами. Важно ответить, что исполнитель всегда находится вне основной группы объектов системы.

Линии жизненного цикла – именованный элемент, который описывает отдельный объект на диаграмме. Каждый объект диаграммы последовательности описывается при помощи линии жизненного цикла. Все элементы (классы) расположены в самой верхней части диаграммы. Линии жизни всегда отображают внутренние объекты, по отношению к системе, тогда как исполнители используются для отображения внешних объектов по отношению к системе.

Сообщения – связь между объектами системы. Сообщения изображаются в хронологическом порядке. Вверху – самые ранние процессы, внизу – самые поздние процессы. Сообщения можно разделить на следующие категории:

– *Синхронные сообщения*. Ожидают ответа от объектов, прежде чем взаимодействие от данного объекта перейдёт к следующему. Отправитель ожидает, пока получатель завершит обработку сообщения.

Отправитель продолжает деятельность только в том случае, когда получатель обработал предыдущее сообщение, т.е. отправил ответ на сообщение. Большинство операций в объектно-ориентированном программировании являются синхронными. Изображаются в виде прямой линии со стрелкой (отправленное сообщение) и пунктирной линией (ответ) (рис 3.1).

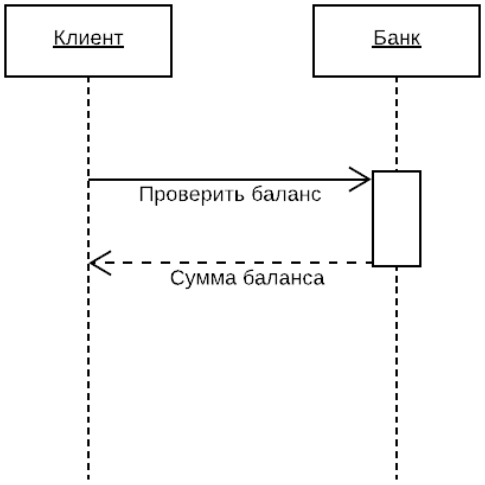


Рисунок 3.1 – Синхронное сообщение

– *Асинхронные сообщения*. Асинхронные сообщения не ждут ответа от получателя. Взаимодействие продвигается независимо от того, обрабатывает ли получатель предыдущее сообщение или нет. Для изображения используется только прямая линия со стрелкой (рис. 3.2).

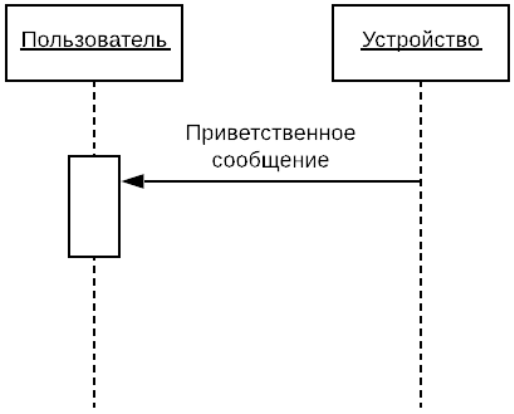


Рисунок 3.2 – Асинхронное сообщение

– *Собственное сообщение.* В некоторых сценариях необходимо отправить сообщение самому себе. Допустим, когда необходимо при определённом сообщение вызвать собственный метод для продолжения взаимодействия. Отображается в виде петлеобразной стрелки (рис. 3.3).

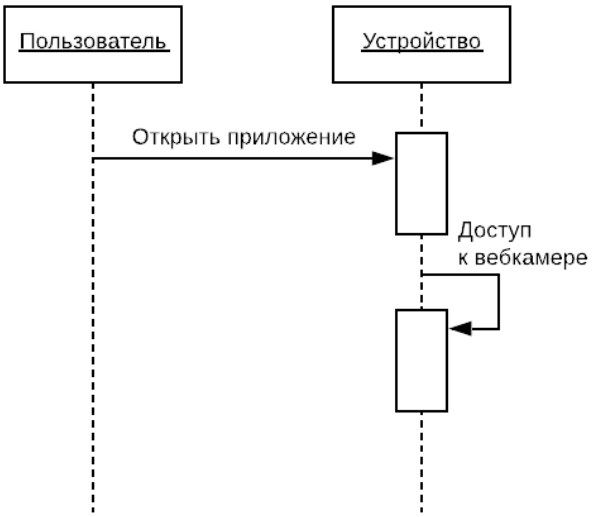


Рисунок 3.3 – Собственное сообщение

– *Найденное сообщение.* Используется для изображения сценария, где неизвестный или неопределённый сервис отправляет сообщение. Изображается в виде прямой линии исходящей от чёрного круга (рис. 3.4).

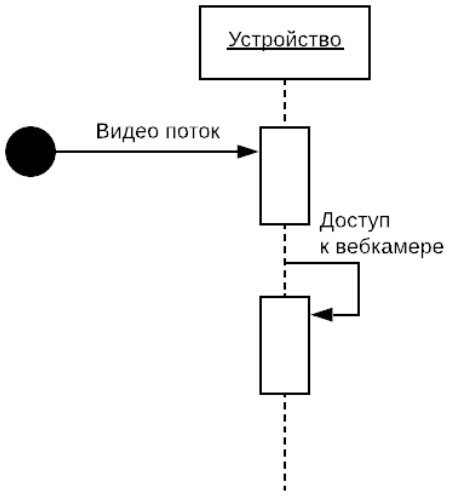


Рисунок 3.4 – Найденное сообщение

Рассмотрим диаграмму последовательностей приложения Sound Room (рис. 3.5), смоделированную в середе Umbrello.

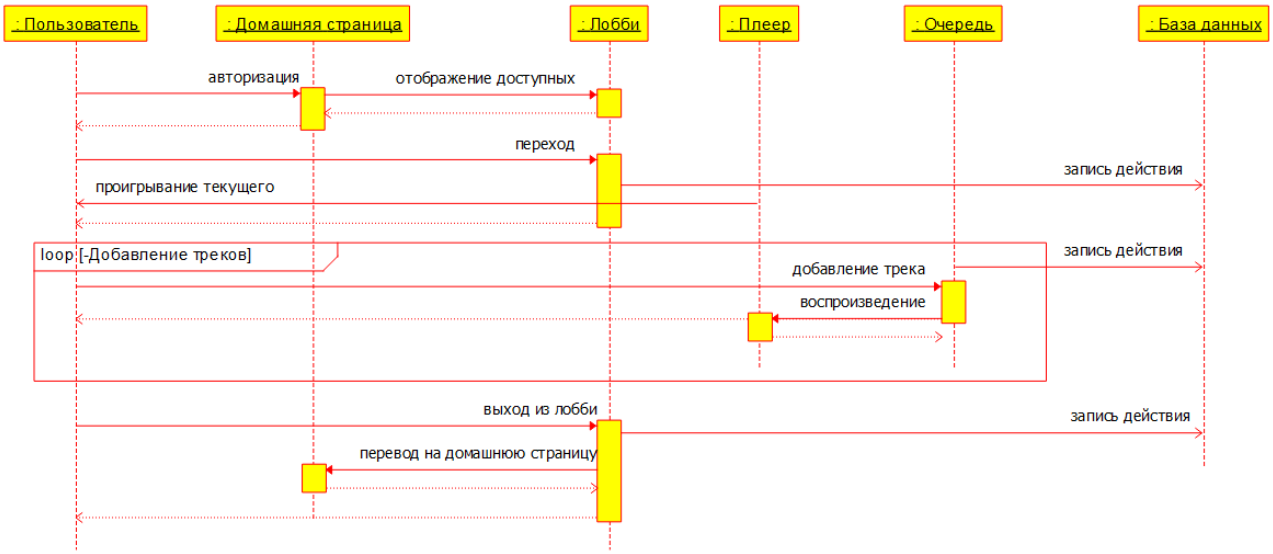


Рисунок 3.5 – Диаграмма последовательности Sound Room

Опишем каждый класс из диаграммы последовательности (рис. 3.5).

Пользователь – концептуальный класс, с которого начинается формирование первого сообщения и активизация всех остальных событий диаграммы. Отправляет такие действия как:

– *авторизация* – вход в систему для дальнейшего взаимодействия со всем функционалом;

– *переход* – действие, которое активизирует сервис маршрутизации и переводит пользователя на другой раздел приложения, в данном случае переход в определённое *лобби*;

– *добавление трека* – вставка ссылки на трек из определённого сервиса (YouTube или SoundCloud);

– *выход из лобби* – задействование сервиса маршрутизации для возвращения пользователя на домашнюю страницу приложения.

Домашняя страница – концептуальный класс, реализующий начальную страницу приложения, которая содержит список лобби для дальнейшего перехода пользователей. Отправляет сообщения:

– *отображение доступных лобби* – действие, выводящее на экран список доступных лобби для перехода.

Лобби – концептуальный класс, реализующий страницу, где пользователи находятся для совместного прослушивания музыки из общего плейлиста (очереди). Реализует следующие сообщения:

– *запись действия* – задействование метода для фиксирования действия в базу данных.

Плеер – концептуальный класс, реализующий функционал элемента воспроизведения треков в каждом лобби. Обладает панелью управления и окном для отображения обложки альбома или сопутствующего видеоряда. Реализует следующие сообщения:

– *проигрывание текущего* – асинхронное сообщение, реализующее автовоспроизведение трека, когда пользователь перешёл на страницу лобби.

Очередь – концептуальный класс, реализующий объект хранения треков, добавляемыми участниками. Обладаем сервисами по управлению треками. В рамках жизненного цикла нахождения пользователя в лобби реализует следующие сообщения:

– *воспроизведение* – сообщение реализующее воспроизведение трека, следующего на очереди после окончания предыдущего.

База данных – концептуальный класс, обозначающий сервис, через который происходит запись информации в базу данных.

В диаграмме последовательности область, которая может повторяться определённое количество раз помещается в отдельный контейнер, которому даётся название. В данном случае это процесс добавления треков в очередь (рис. 3.6).

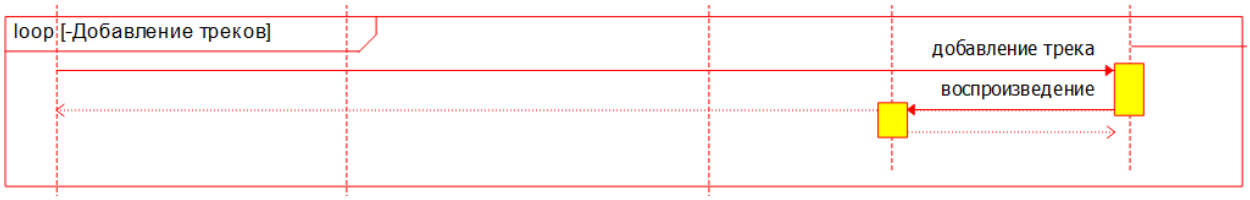


Рисунок 3.6 – Цикл добавления треков

Для того чтобы создать диаграмму последовательностей в проекте Umbrello, создайте новую диаграмму последовательностей (Sequence Diagram…) в разделе Logical View (рис. 3.7).

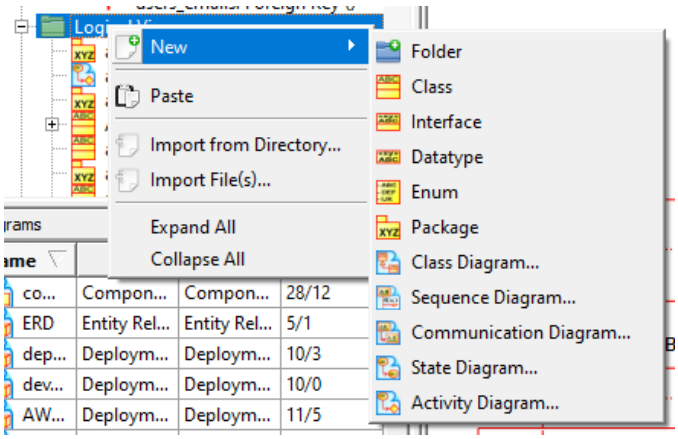


Рисунок 3.7 – Добавление диаграммы последовательнсотей

В рабочей области диаграммы последовательности щелчком правой кнопки мыши добавьте новый объект на диаграмму (рис. 8).

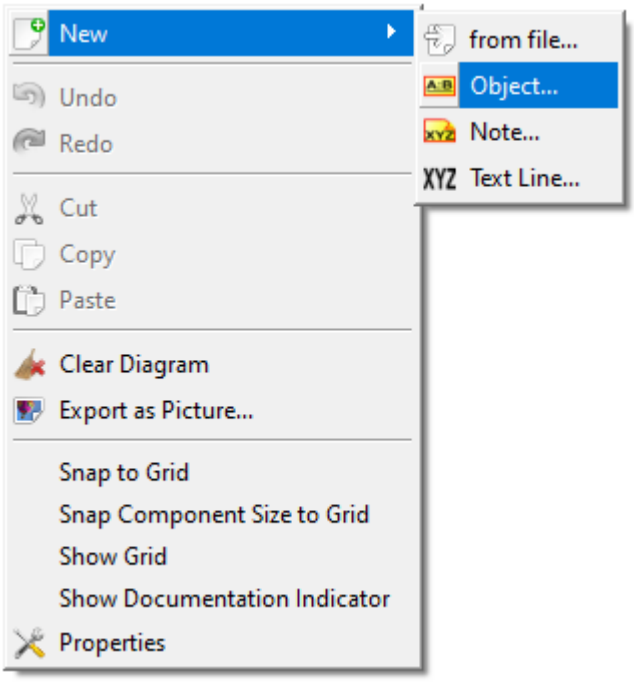


Рисунок 3.8 – Добавление нового объекта

Дайте название новому объекту (рис. 3.9).

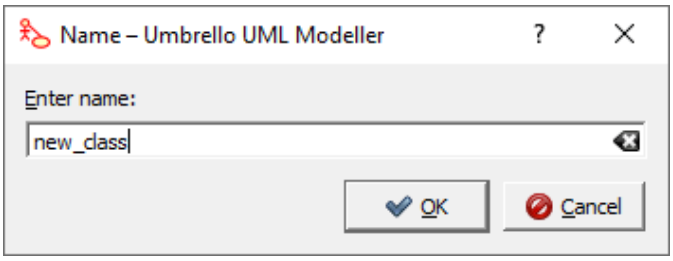


Рисунок 3.9 – Именование нового объекта

Для добавления синхронного сообщения от одного класса к другому нажмите на соответствую кнопку на панели элементов (рис. 3.10).

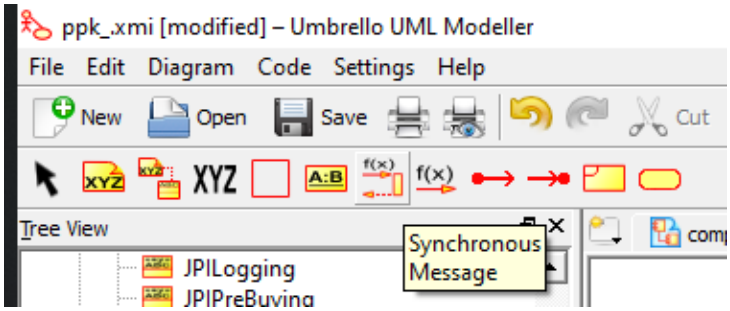


Рисунок 3.10 – Выбор синхронного сообщения

Для того чтобы провести сообщение, для начала выберете линию жизненного цикла первого концептуального класса и проведите линии ко второму концептуальному классу. Далее возникнет окно, в котором необходимо указать действие, которое совершается при отправке сообщения (рис. 3.11). Название необходимо указать в поле (Custom operation).

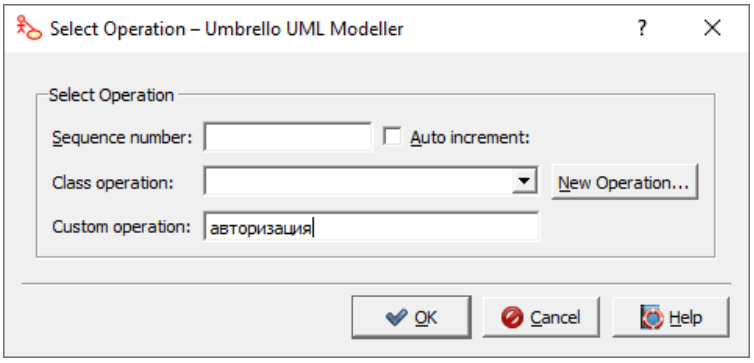


Рисунок 3.11 – Окно именования сообщения

В итоге должна получиться связь, изображённая на рисунке 3.12.

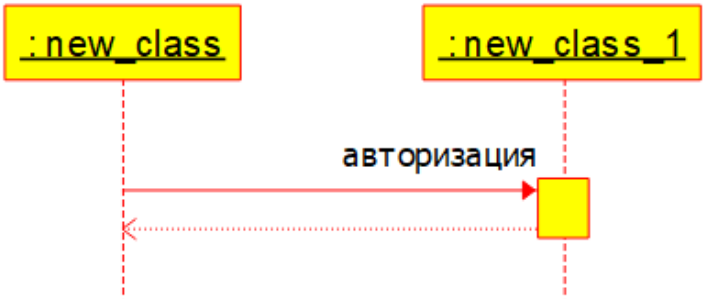


Рисунок 3.12 – Связь между двумя классами

Аналогичная операция производится, когда вам необходим указать асинхронное сообщение или продолжить добавлять объекты для обозначения взаимосвязи отправки сообщений.

Вопросы для самоконтроля.

1. Что описывают стандартные диаграммы последовательности?
2. Что описывают системные диаграммы последовательности?
3. Дайте определение исполнителю.
4. Чем являются сообщения в диаграмме последовательности?
5. Дайте определение линии жизненного цикла.
6. Дайте определение синхронному сообщению.
7. Дайте определение асинхронному сообщению.
8. Дайте определение собственному сообщению.
9. Дайте определение найденному сообщению.
10. На что влияет порядок объектов (концептуальных классов) на диаграмме последовательности?

Задание.

Спроектируйте диаграмму последовательности по системе Вашей предметной области. Разработайте диаграмму в среде Umbrello и добавьте описание всех связей и сущностей в отчёте к лабораторной работе.

Лабораторная работа №4 - Проектирование диаграммы компонентов.

Для создания конкретной физической системы необходимо реализовать все элементы логического представления в конкретные материальные сущности. Для описания таких реальных сущностей предназначен другой аспект модельного представления, а именно – физическое представление модели. В контексте языка UML это означает совокупность связанных физических сущностей, включая программное и аппаратное обеспечение, а также персонал, которые организованы для выполнения специальных задач.

Физическая система (physical system) — реально существующий прототип модели системы.

С тем чтобы пояснить отличие логического и физического представлений, необходимо в общих чертах рассмотреть процесс разработки программной системы. Ее исходным логическим представлением могут служить структурные схемы алгоритмов и процедур, описания интерфейсов и концептуальные схемы баз данных. Однако для реализации этой системы необходимо разработать исходный текст программы на языке программирования. При этом уже в тексте программы предполагается организация программного кода, определяемая синтаксисом языка программирования и предполагающая разбиение исходного кода на отдельные модули.

Однако исходные тексты программы еще не являются окончательной реализацией проекта, хотя и служат фрагментом его физического представления. Программная система может считаться реализованной в том случае, когда она будет способна выполнять функции своего целевого предназначения. А это возможно, только если программный код системы будет реализован в форме исполняемых модулей, библиотек классов и процедур, стандартных графических интерфейсов, файлов баз данных. Именно эти компоненты являются базовыми элементами физического представления системы в нотации языка UML.

Полный проект программной системы представляет собой совокупность моделей логического и физического представлений, которые должны быть согласованы между собой. В языке UML для физического представления моделей систем используются так называемые диаграммы реализации, которые включают в себя две отдельные канонические диаграммы: диаграмму компонентов и диаграмму развертывания.

Диаграмма компонентов, в отличие от ранее рассмотренных диаграмм, описывает особенности физического представления системы. Диаграмма компонентов позволяет определить архитектуру разрабатываемой системы, установив зависимости между программными компонентами, в роли которых может выступать исходный, бинарный и исполняемый код. Во многих средах разработки модуль или компонент соответствует файлу. Пунктирные стрелки, соединяющие модули, показывают отношения взаимозависимости, аналогичные тем, которые имеют место при компиляции

исходных текстов программ. Основными графическими элементами диаграммы компонентов являются компоненты, интерфейсы и зависимости между ними.

Компонент – это несколько размытое понятие UML, поскольку их можно использовать для моделирования тех же сущностей, что и классы. Можно привести цитату Румбаха (Rumbaugh), одного из создателей UML:

Различи между структурированным классом и компонентом является весьма относительным и определяется, скорее, предпочтениями, чем строгой семантикой.

Обратимся к определению компонента из спецификации UML:

Компонент (component) представляет собой отдельно размещаемую и заменяемую часть системы, инкапсулирующую своё содержимое. Поведение компонента определяется в терминах реализации интерфейсов. Таким образом, компонент служит типом, определяемым с помощью реализуемых интерфейсов.

Данную идею можно проиллюстрировать с помощью обычного класса UML и реализуемых им интерфейсов. Классы UML для моделирования программного элемента любого уровня, от целой системы или подсистемы до небольшого вспомогательного объекта.

Также термин компонент можно интерпретировать следующим образом.

Компонент (component) — физически существующая часть системы, которая обеспечивает реализацию классов и отношений, а также функционального поведения моделируемой программной системы.

Компонент предназначен для представления физической организации ассоциированных с ним элементов модели. Дополнительно компонент может иметь текстовый стереотип и помеченные значения, а некоторые компоненты – собственное графическое представление. Компонентом может быть исполняемый код отдельного модуля, командные файлы или файлы, содержащие интерпретируемые скрипты.

При использовании компонентов на диаграммах UML акцент делается на *интерфейсах*, модульности и заменяемости. Это означает, что компонент практически не зависит от других внешних элементов (за исключением, возможно, стандартных библиотек) и является относительно самостоятельным модулем.

Компонент UML рассматривается на уровне проектирования и не существуюе в конкретно программной реализации. Однако они соответствуют конкретным артефактам, например наборам файлов.

Хорошей аналогией для моделирования программных компонентов служат современные домашние кинотеатры, которых легко можно заменить медиа-плеер или колонки. Такой кинотеатр состоит из отдельных заменяемых модулей, взаимодействующих друг с другом через стандартный интерфейс. Например, на высоком уровне абстракции в качестве компонента можно рассматривать СУБД SQL, поскольку одну и ту же версию SQL могут поддерживать различные базы данных. В более мелком масштабе в качестве

компонента можно рассматривать стандартный программный интерфейс Java Message Service.

Модуль (module) — часть программной системы, требующая памяти для своего хранения и процессора для исполнения.

В этом случае верхний маленький прямоугольник концептуально ассоциировался с данными, которые реализует этот компонент (иногда он изображается в форме овала). Нижний маленький прямоугольник ассоциировался с операциями или методами, реализуемыми компонентом. В простых случаях имена данных и методов записывались явно в маленьких прямоугольниках, однако в языке UML они не указываются.

Имя компонента подчиняется общим правилам именования элементов модели в языке UML и может состоять из любого числа букв, цифр и знаков препинания. Отдельный компонент может быть представлен на уровне типа или экземпляра. И хотя его графическое изображение в обоих случаях одинаково, правила записи имени компонента несколько отличаются.

Правила именования объектов в языке UML требуют подчеркивания имени отдельных экземпляров, но применительно к компонентам подчеркивание их имени часто опускают. В этом случае запись имени компонента со строчной буквы характеризует компонент уровня примеров.

В качестве собственных имен компонентов принято использовать имена исполняемых файлов, динамических библиотек, web-страниц, текстовых файлов или файлов справки, файлов баз данных или файлов с исходными текстами программ, файлов скриптов и другие.

В отдельных случаях к простому имени компонента может быть добавлена информация об имени объемлющего пакета и о конкретной версии реализации данного компонента. Необходимо заметить, что в этом случае номер версии записывается как помеченное значение в фигурных скобках. В других случаях символ компонента может быть разделен на секции, чтобы явно указать имена реализованных в нем классов или интерфейсов. Такое обозначение компонента называется расширенным.

Поскольку компонент как элемент модели может иметь различную физическую реализацию, иногда его изображают в форме специального графического символа, иллюстрирующего конкретные особенности реализации. Строго говоря, эти дополнительные обозначения не специфицированы в нотации языка UML. Однако, удовлетворяя общим механизмам расширения языка UML, они упрощают понимание диаграммы компонентов, существенно повышая наглядность графического представления.

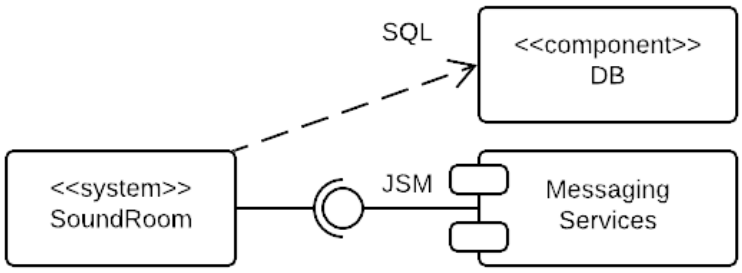


Рисунок 4.1 – Представление видов компонентов и интерфейсов на диаграмме компонентов

На рисунке 4.1 представлено каким образом могут быть изображены обозначения компонентов и интерфейсов. В среде Umbrello обозначение компонентам имеет унифицированный формат, но для общего понимания также будет рассмотрен данный пример, в виду того, что построение диаграммы компонентов в разных средах отличается.

Классическое обозначение компонента представлено у сервиса Messaging Services, обычно это прямоугольник, у которого имеется два прямоугольника на левой грани. Интерфейс подключения может быть обозначен как дуга полукруга (сервис, который подключается) или как круг (сервис, к которому подключаются). Но в некоторых в сервисах (в том числе Umbrello) может быть представлено в виде направленно пунктирной стрелке, где указатель обозначает сервис, к которому подключаются.

Отношение зависимости служит для представления факта наличия специальной формы связи между двумя элементами модели, когда изменение одного элемента модели оказывает влияние или приводит к изменению другого элемента модели. Отношение зависимости на диаграмме компонентов изображается пунктирной линией со стрелкой, направленной от клиента или зависимого элемента к источнику или независимому элементу модели.

Зависимости могут отражать связи отдельных файлов программной системы на этапе компиляции и генерации объектного кода. В других случаях зависимость может указывать на наличие в независимом компоненте описаний классов, которые используются в зависимом компоненте для создания соответствующих объектов. Применительно к диаграмме компонентов зависимости могут связывать компоненты и импортируемые этим компонентом интерфейсы, а также различные виды компонентов между собой.

В этом случае рисуют стрелку от компонента-клиента к импортируемому интерфейсу. Наличие такой стрелки означает, что компонент не реализует соответствующий интерфейс, а использует его в процессе своего выполнения. При этом на этой же диаграмме может присутствовать и другой компонент, который реализует этот интерфейс. Отношение реализации интерфейса обозначается на диаграмме компонентов обычной линией без стрелки.

Рассмотрим диаграмму компонентов клиентской логики сервиса Sound Room, разработанного в среде Umbrello (рис. 4.2).

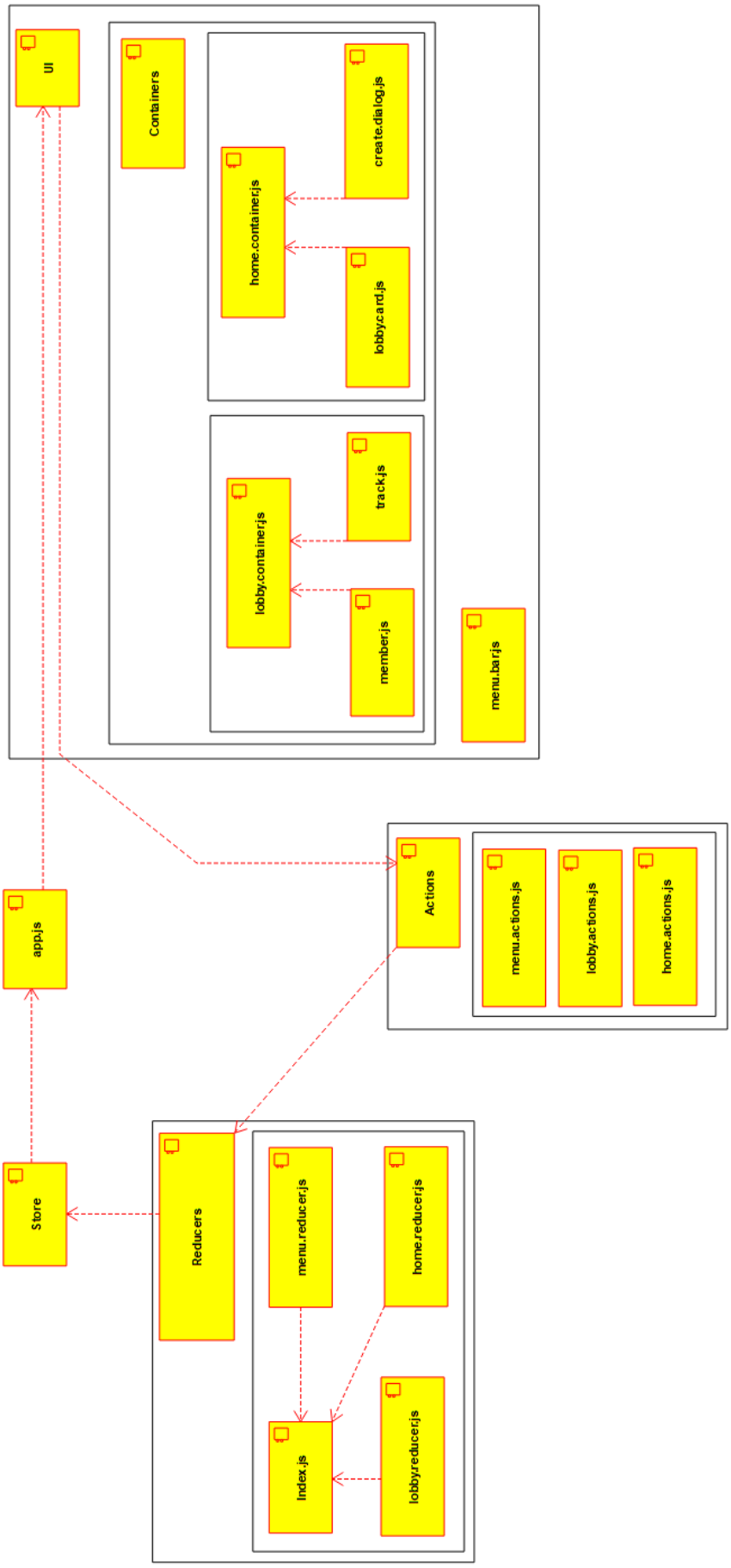


Рисунок 4.2 – Диаграмма компонентов Sound Room

На диаграмме (рис. 4.2) отображены 3 контейнера, внутри которых в правом верхнем углу располагается компонент. Данный компонент является

названием всего контейнера, который обозначает составной компонент, в который входят элементы данного компонента, которые также являются компонентами. Данный способ применяется в виду того, что в среде Umbrello на данный момент нет возможности проектировать многоуровневые компоненты для описания структуры приложения.

Опишем каждый из компонентов.

Store – компонент, хранящий состояние передаваемое редьюсерами. Обрабатывает все изменения и изменяет состояние компонентов, после передачи нового состояния редьюсером.

app.js – главный компонент приложения, с которого производится запуск приложения активация всех основных компонентов.

UI – компонент, который содержит все графические элементы, разделяется на компоненты-контейнеры (*Containers*) и компонента меню приложения (*menu.bar.js*).

Containers – состоящий из *lobby.containers.js* (контейнер элементов лобби) и *home.container.js* (контейнер элементов домашней страницы), который содержит контейнеры для обособленных компонентов.

lobby.container.js – составной компонент контейнера лобби, который содержит в себе компоненты *member.js* (компонент графического отображения пользователя) и *track.js* (графический компонент трека).

home.container.js – составной компонент контейнера домашней страницы, который содержит в себе компоненты *lobby.cards.js* (графический компонент карточек лобби, по нажатию на которую пользователя переносит в раздел данного лобби) и *creat.dialog.js* (компонент диалогового окна пользователя, через которое есть возможность авторизоваться или зарегистрироваться).

Actions – компонент функциональных действий графических компонентов сервиса. Состоит из: *menu.actions.js* (действия компонента меню), *lobby.actions.js* (действия компонента лобби) и *home.actions.js* (действия компонента домашней страницы).

Reducers – составной компонент, который содержит редьюсеры – компоненты обрабатывающие действия всех графических элементов сервиса. Содержит компонент *Index.js*, который содержит в себе зависимости всех редьюсеров: *menu.reducer.js* (обработчик действий меню), *lobby.reducer.js* (обработчик действий лобби) и *home.reducer.js* (обработчик действий домашней страницы).

Для того чтобы создать диаграмму компонентов в среде Umbrello необходимо добавить новую диаграмму компонентов (Componen Diagram…) в разделе Component View (рис. 4.3).

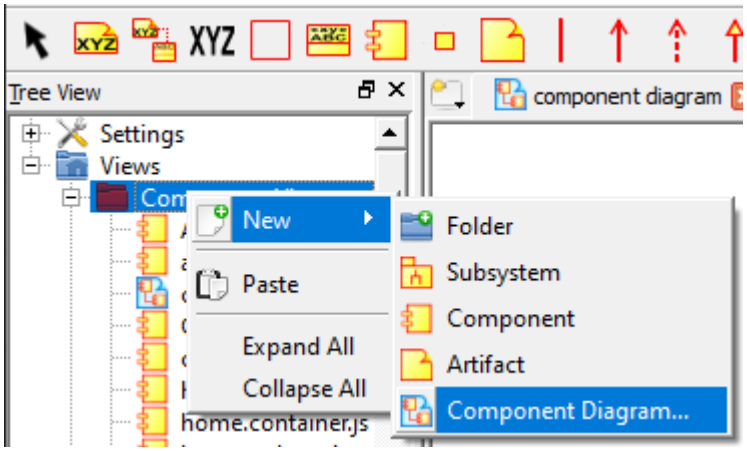


Рисунок 4.3 – Создание новой диаграммы компонентов

Чтобы добавить новый компонент в рабочую область диаграммы, необходимо нажать соответствую иконку из панели инструментов (рис. 4.3) или нажатием правой кнопки мыши выбраться пункт Component… (рис. 4.4).

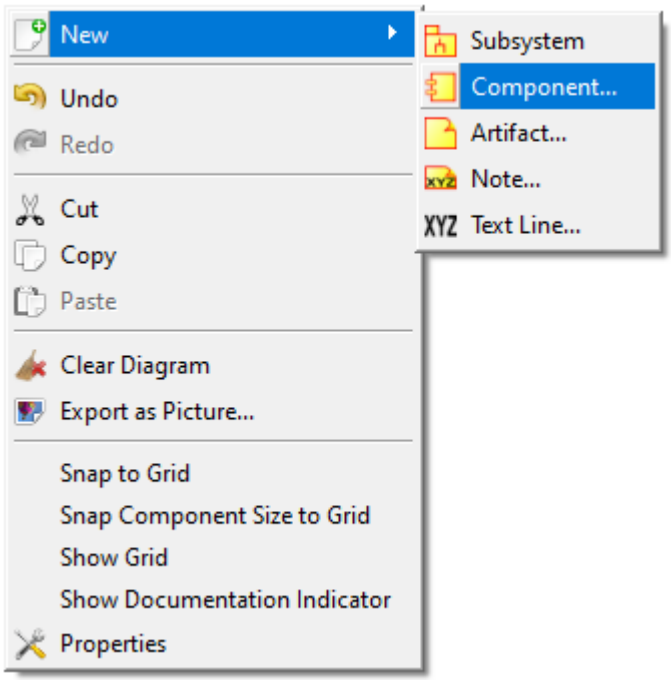


Рисунок 4.4 – Добавление компонента

После добавления элементов и группирования их при помощи стандартного контейнера (иконка красного прямоугольника (рис. 4.1)), необходим провести зависимости (интерфейсы) между компонентами. Для этого необходим выбрать тип линии Dependency… из панели элементов (рис. 4.5) и провести зависимость между элементами (рис. 4.6).

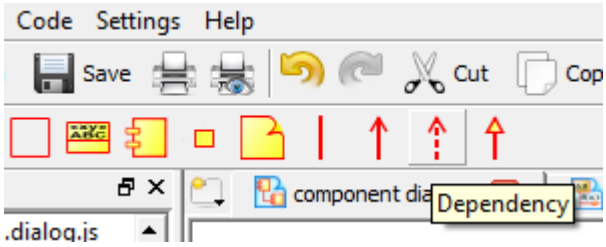


Рисунок 5 – Линия зависимости

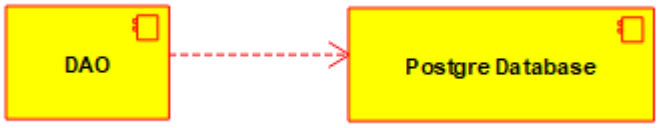


Рисунок 4.6 – Зависимость между компонентами

Вопросы для самоконтроля

1. Зачем необходимо проектировать диаграмму компонентов?
2. Дайте одно из представленных определений компонента.
3. Дайте определение интерфейса.
4. Верно ли утверждение, что компонентом может быть только файл?
5. Верно ли утверждение, что обозначение зависимости (интерфейса) между компонентами является строго унифицированным?

6. Что такое физическая система в термине диаграммы компонентов?

7. Чем является модуль в диаграмме компонентов?

8. Существует ли компонент в конкретной программной реализации?

9. Верно ли утверждение, что диаграмма компонентов – всегда строго определённая нотация?

10. Как обозначается компонент в разных средах проектирования диаграмм (указать 1 из распространённых видов)?

Задание

Спроектируйте диаграмму компонентов в среде Umbrello в соответствии с вашей предметной областью. Опишите связи и предназначение смоделированных компонентов в отчёте к лабораторной работе.

Лабораторная работа №5 - Проектирование системных диаграмм последовательностей.

Основным отличием системной диаграммы последовательностей от стандартной диаграммы заключается в явном представлении абстрактных классов как прототип классов, которые будут реализованы в системе. А за обозначения действий отправления сообщений отвечают методы или процедуры или функции. Вновь дадим определение, что такое диаграмма последовательностей.

Диаграммы последовательности (sequence diagram) – это быстро и легко создаваемый артефакт, иллюстрирующий входные и выходные события, связанные с разрабатываемой системой.

Для иллюстрации событий, связывающих внешних исполнителей с системой, в языке UML существуют специальные обозначения, позволяющие создавать диаграммы последовательностей.

На системной диаграмме последовательностей для определённого хода событий, описанного в прецеденте, отображаются внешние исполнители, которые взаимодействуют непосредственно с системой, сама система (как «чёрный ящик»), а также системные события, инициируемые исполнителями. При этом порядок событий должен соответствовать их последовательности в описании прецедента. Время на диаграмме последовательности изменяется сверху вниз.

Прецеденты определяют, как внешние исполнители взаимодействуют с программной средой. В процессе этого взаимодействия исполнителем генерируется *системные события* (system event), которые представляют собой запросы на выполнение некоторой *системной операции* (system operation). Например, кассир, введя идентификатор товара (метод enterItem). Это событие инициирует в системе выполнение некоторой операции. Метод enterItem вводится в описании прецедента, на диаграмме конкретизируется и формализуется.

В состав языка UML входят обозначения для *диаграммы последовательностей* (sequence diagram), с помощью которых можно проиллюстрировать взаимодействие исполнителя с системой и операции, выполнение которых при этом инициируется.

Системная диаграмма последовательностей (system sequence diagram - SSD) – это схема, которая для определённого сценария прецедента показывает генерируемые внешними исполнителями события, их порядок, а также события, генерируемые внутри самой системы. При этом все системы рассматриваются как «чёрный ящик». Назначение данной диаграмма – отображение событий, передаваемых исполнителями системе через её границы.

При разработке системы возникает важный вопрос: какие события поступают в систему? Проектируемая система обрабатывать эти события (поступающие от мыши, клавиатуры и других интерфейсов взаимодействия) и реагировать на них. Программная среда реагирует на три типа событий: внешние события, инициируемые исполнителями (людьми или

компьютерами), таймерные события и сбои или исключения (зачастую генерируемые внешними источниками).

Поэтому очень важно знать, что следует понимать под внешними, или *системными*, событиями (system event). Они является важной частью анализа поведения системы.

На системной диаграмме последовательностей термины (операции, параметры, возвращаемые значения) отображаются в краткой форме. Поэтому на этапе проектирования могут понадобиться некоторые пояснения этих терминов. Если значение терминов не разъясняется в описании прецедентов, то их нужно вести словарь терминов.

Не создавайте СДП для всех сценариев, поскольку для этого потребуется идентифицировать все системные операции. Лучше построить СДП только для тех сценариев, которые будут реализовываться на следующей итерации.

Диаграммы последовательностей облегчают понимание интерфейса и принципов взаимодействия с существующими системами, а также позволяют документировать архитектуру.

Диаграммы последовательностей – это часть модели прецедентов. Они обеспечивают визуализацию взаимодействия объектов при реализации прецедента. В исходном описании унифицированного процесса диаграммы последовательностей не упоминаются явно. Диаграммы последовательностей – это пример артефактов анализа и проектирования или видов деятельности, не упомянутых ранее в документации.

Рассмотрим системную диаграмму приложения Sound Room (рис. 5.1). В данном разделе будет рассмотрена системная интерпретация ранее созданной диаграммы последовательностей (Лабораторная работа №3). Указания по проектированию диаграммы данного типа также можно найти в Лабораторной работе №3.

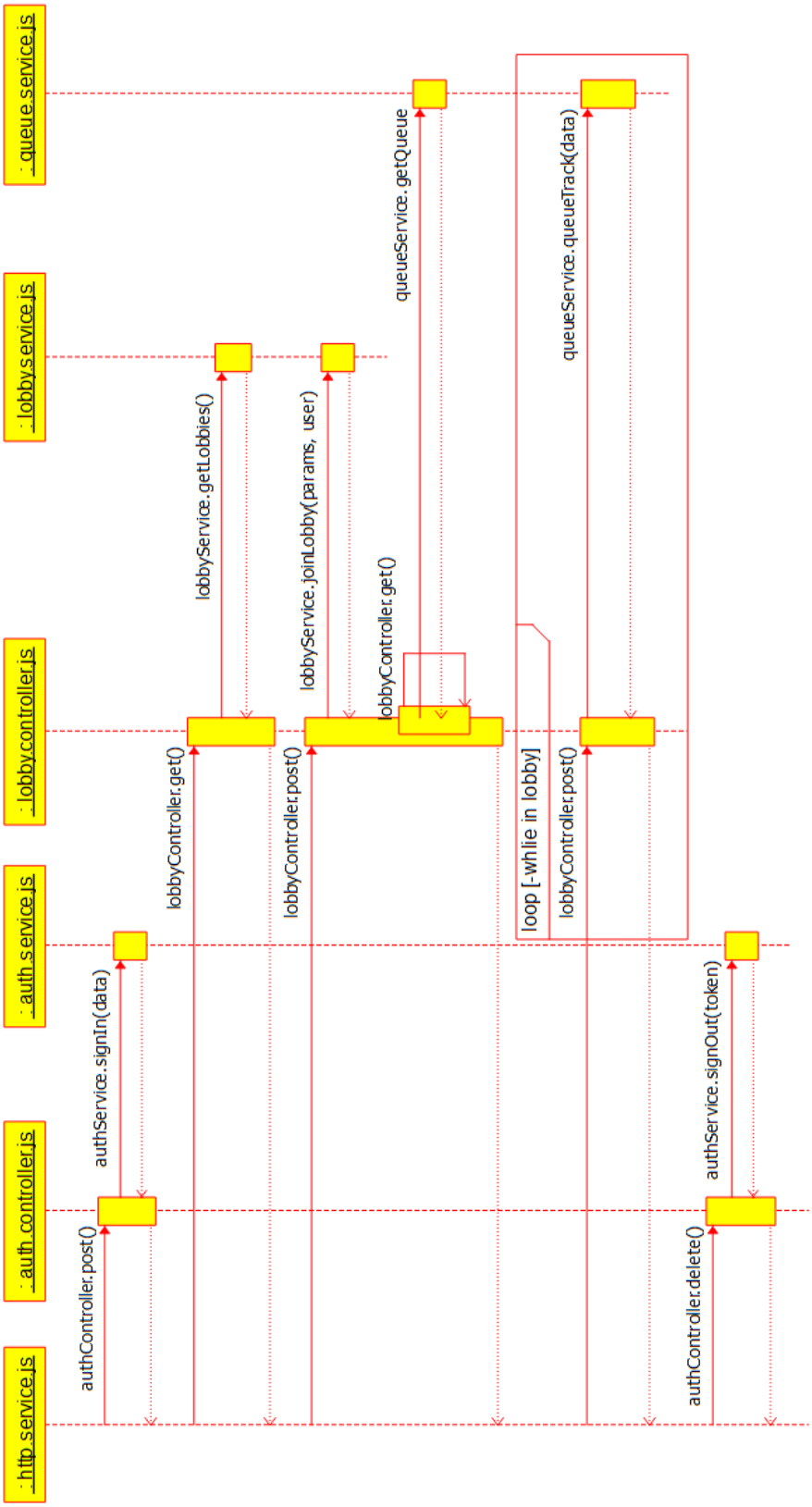


Рисунок 5.1 – Системная диаграмма последовательностей Sound Room

Далее последует описание исполнителей и системных классов, а также операции, которые активизируются после вызова других операций.

http.service.js – класс сервиса веб-запросов. На диаграмме представляет роль исполнителя ввиду того, что любой запрос пользователя всегда проходит через обработчик веб-запросов. Реализует следующие операции (в случае текущей технической реализации – функции).

– *authController.post()* – операция активирующая сервис авторизации пользователя на ресурсе;

– *lobbyController.get()* – операция для получения списка доступных лобби;

– *lobbyController.post()* – операция реализующая переход в лобби, а также, за управлением действий пользователя в лобби;

– *authController.delete()* – операция активизирующая сервис для выхода пользователя из системы.

auth.controller.js – класс вызывающий интерфейс сервиса авторизации. Реализует логику обращения к функциям авторизации пользователя в системе. Все операции работают при вызове от других операций. Исполняет в данном сценарии следующие операции:

– *authService.signIn(data)* – сервис авторизации пользователя на ресурсе;

– *authService.signOut(token)* – сервис завершения сеанса пользователя в системе, т.е. выход из системы.

auth.service.js – класс, реализующий весь функционал авторизации в системе.

lobby.controller.js – класс реализующий функционал по запросам к сервисом для манипуляции элементов лобби. В данной диаграмме задействует следующие операции:

– *lobbyService.getLobbies()* – операция, отображающая список доступных лобби;

– *lobbyService.joinLobby(params, user)* – операция позволяющая пользователю войти в определённое лобби;

– *lobbyController.get()* – операция использующая сервисы для отображения всех графических компонентов в разделе лобби;

– *queuService.getQueue()* – операция вызывающая объект очереди в определённое лобби;

– *queueService.queueTrack(data)* – операция позволяющая пользователям добавлять трек в очередь.

queue.service.js – класс, реализующий функционал для манипуляции с очередью лобби.

В текущем сценарии присутствуют зацикленные операции (рис. 5.2). В данном цикле реализован прецедент добавление пользователем треков до момента покидания лобби.

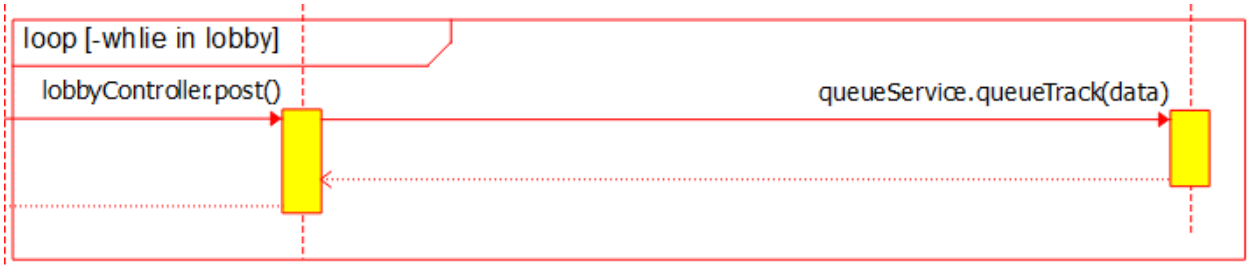


Рисунок 5.2 – Цикл добавление пользователем треков

Вопросы для самоконтроля

11. Дайте определение системной диаграмме последовательностей.
12. Дайте определение исполнителям.
13. Какова роль исполнителей в системе?
14. Что определяют прецеденты?
15. В чём основное назначение системной диаграммы последовательностей?
16. На какие типы программных событий реагирует программная среда?
17. Верно ли утверждение, что системную диаграмму последовательности необходимо реализовывать для всех прецедентов?
18. В качестве какого объекта обычно рассматривается система при построении системных диаграмм последовательности?
19. Что такое системные события?
20. Какую роль играют диаграммы прецедентов в модели прецедентов?

Задание

Спроектируйте системную диаграмму последовательностей в среде Umbrello в соответствии с вашей предметной областью. Опишите связи и предназначение смоделированных компонентов в отчёте к лабораторной работе.

Лабораторная работа №6 - Составление технического задания.

Составление технического задания является важнейшим элементом взаимодействия между заказчиком и разработчиками. Именно данный документ закрепляет в себе все необходимые требования, выставленные заказчиком перед исполнителем, что подразумевает сведение переработок к минимуму.

Стоит также отметить, что составление технического задания, а также документирование в целом не подпадают под ведение проекта по Agile-методологии. Данный вывод можно сделать при прочтении тезисов из манифеста Agile: «Работающий продукт важнее исчерпывающей документации. Сотрудничество с заказчиком важнее согласований условий контракта. Готовность к изменения важнее следования первоначальному плану.» Следовательно, Agile-разработка нацелена на динамичную разработку с постоянно меняющимися требованиями, но данный факт не означает полного отказа от документирования проекта, приоритет отдаётся иным взаимодействиям. Такой подход применяется при разработке *MVP* (Minimum Viable Product – минимально жизнеспособный продукт). Но когда речь заходит о разработке внутреннего корпоративного продукта, документирование итераций разработки является необходимостью. Данная потребность происходит из необходимости длительной поддержки продукта.

В данном руководстве техническое задание будет составляться на разработку приложения «Sound Room». Ввиду того, что данное приложения не является корпоративным продуктом, а также основывается на WEB-технологиях, составление технического задания будет адаптировано под данные условия.

Составляться техническое задание будет по образцу стандарта 830-1998 — IEEE Recommended Practice for Software Requirements Specifications. Вне зависимости от того, что данный стандарт был разработан в 1998 году, он не утерял актуальность и удобство составления в отличии от ГОСТ 34 и ГОСТ 19.

Для составления технического задания по стандарту IEEE 830-1998 — IEEE Recommended Practice for Software Requirements Specifications в документе технического задания должны присутствовать следующие разделы:

1. Введение
    1.1. Назначение
    1.2. Предполагаемая аудитория и предложения по прочтению
    1.3. Определения, акронимы и сокращения
    1.4. Ссылки
    1.5. Краткий обзор
2. Общее описание
    2.1. Взаимодействие продукта (с другими продуктами и компонентами)
    2.2. Функции продукта (краткое описание)

2.3. Характеристики пользователя
2.4. Ограничения
2.5. Допущения и зависимости
3. Детальные требования
3.1. Требования к внешним интерфейсам
3.1.1. Интерфейсы пользователя
3.1.2. Интерфейсы аппаратного обеспечения
3.1.3. Интерфейсы программного обеспечения
3.1.4. Интерфейсы взаимодействия
3.2. Функциональные требования
3.3. Требования к производительности
3.4. Проектные ограничения
3.5. Нефункциональные требования (надёжность, доступность и пр.)

1 Введение

1.1 Назначение

Данное техническое задание описывает функциональные и нефункциональные требования для разработки приложения Sound Room. Приложение Sound Room является онлайн социальным сервисом для прослушивания музыки.

1.2 Предполагаемая аудитория и предложения по прочтению

Данный документ составлен для ознакомления команды разработчиков проекта Sound Room, а также для непосредственного Заказчика приложения Sound Room. Документ составлен для структурирования и согласования требований для разработки проекта Sound Room.

1.3 Определения, акронимы и сокращения

Таблица 1 - Определения

Термин	**Определение**
Пользователь	Человек, взаимодействующий с приложением и зарегистрированный в системе.
Администратор	Человек, зарегистрированный в системе, являющийся создателем лобби или имеющий права управлением лобби.
Гость	Пользователь незарегистрированный в системе.
Лобби	Страница приложения, где располагается плеер, очередь треков, а также чат данного лобби.
БД	База данных.
Сервис	Компонент реализующий определённый функционал

1.4 Ссылки

В данном разделе должны размещаться ссылки на документы, нормативы и ресурсы, используемые при разработке проекта. Для примера

указаны ссылки на стандарт составления документации, а также литературу по проектированию.

1. IEEE 830-1998 — IEEE Recommended Practice for Software Requirements Specifications. URL: https://standards.ieee.org/standard/830-1998.html

2. Крэг Ларман. Применение UML 2.0 и шаблонов проектирования Практическое руководство. 3-е издание. : Пер. с англ. – М. : ООО «И.Д. Вильямс», 2009. – 736 с.

1.5 Краткое обзор

Sound Room является web-приложением для совместного прослушивания музыки из источников медиа-хостинга таких как YouTube и Soundcloud. Пользователь может зарегистрироваться в системе и зайти в уже созданное лобби или создать своё собственное. Незарегистрированный пользователь имеет доступ к открытым комнатам, но не может создать собственную.

2 Общее описание

2.1 Взаимодействие продукта

Приложение взаимодействует с API двух сервисов: YouTube и Soundcloud. При взаимодействии с Soundcloud API приложение будет посылать GET-запрос по ссылке трека, переданного в очередь лобби. В случае YouTube, в место, где расположена обложка альбома, будет размещён iframe с видео из YouTube.

Для регистрации будет использована технология OAuth для возможности регистрации и авторизации пользователя через аккаунты сторонних сервисов таких как Google, Discord, Twitch, Twitter и Soundcloud.

2.2 Функции продукта

Абсолютно любой пользователь имеет доступ к веб-приложению. Новый пользователь может зарегистрироваться, используя персональные email или с помощью авторизации через сторонние сервисы. Любой авторизованный пользователь может создать собственную комнату или заходить в сторонние с условием, что комната является открытой или пользователь знает пароль для доступа.

В каждой комнате реализована система очереди треков. Треки добавляются в очередь путём введения прямой ссылки на трек из сервиса Soundcloud или YouTube. Администратор имеет право управлять как очередью треков (удалять и добавлять), тогда как обычные участники лобби могут только добавлять. Администратор имеет право удалять участников из группы или давать бан на определённое время.

2.3 Характеристики пользователя

Таблица 2 – Описание пользователей

Тип пользователя	**Описание**
Гость	Любой человек, не авторизованный в системе.

Пользователь	Зарегистрированный и авторизованный в системе человек. Основной целевой аудиторией являются меломаны, независимые исполнители, а также организаторы культурных мероприятий
Менеджер проекта (разработка)	Человек, принимающий решения на всех стадиях разработки проекта – от согласования требований с заказчиком проекта до введения приложения в эксплуатацию.
Frontend разработчик (разработка)	Человек, ответственный за разработку графического веб-интерфейса приложения.

Таблица 2 – продолжение

Backend разработчик (разработка)	Человек, ответственный за разработку серверной логики работы приложения
DevOps инженер (разработка)	Человек, ответственный за настройку окружения разработки приложения, а также за настройку окружения, в котором приложение будет хостится.

2.4 Ограничения

Главным ограничением приложения является отсутствие функции работы оффлайн. Веб-приложение подразумевает использование данного продукта через веб-браузер.

Ввиду использования технологии React для разработки графического интерфейса приложения, ограничением является отсутствие поддержки браузеров Internet Explorer любой версии.

На данном этапе мобильное приложения для сервиса не рассматривается, но пользователи смартфонов смогут иметь доступ к приложению через мобильные браузеры.

Стек MERN (MongoDB Express React Node) подразумевает использование NoSQL базу данных MongoDB для хранения активных данных в рабочих сессиях. Реляционная БД PostgreSQL служит в качестве персистентного хранилища данных сессий.

2.5 Допущения и зависимости

Таблица 3 – Допущения и зависимости

ЗАВ-1:	React как библиотека для разработки веб-приложения по технологии SPA (Single Page Application)
ЗАВ-2:	Redux как вспомогательная технология при разработке проекта на React. Данная технология помогает лучше проектировать изменения состояний компонентов.

Таблица 3 – продолжение

ЗАВ-3:	React Router как вспомогательное дополнение к библиотеке React для более эффективной разработки переходов на различные разделы приложения.
ЗАВ-4:	Express как веб-сервер обрабатывающий запросы с клиента и сервера.
ЗАВ-5:	Node.js как технология исполнения логики серверного кода на JavaScript.

ЗАВ-6:	Webpack как инструмент сборки, построения и компиляции исходного кода графического интерфейса.
ЗАВ-7:	NPM как пакетный менеджер для всего проекта.
ЗАВ-8:	MongoDB как NoSQL база данных для хранения активных данных.
ЗАВ-9:	PostgreSQL как база данных для длительного хранения персистентных данных.
ЗАВ-10:	Soundcloud SDK для доступа к API сервиса с возможностью использования функции в собственной реализации логики воспроизведения треков.

3 Детальные требования

3.1 Требования к внешним интерфейсам

3.1.1Интерфейсы пользователя

При переходе пользователя на главную страницу приложения на средней части экрана должны отображаться недавно созданные открытые комнаты в виде небольших прямоугольников с рамкой и уменьшенной версии картинки фона данного лобби.

В верхней части экрана должно находить меню. Переходы на дополнительные разделы приложения реализованы в выпадающем боковом меню, отображающего при нажатии соответствующей кнопки в «шапке» приложения. Справа от данной кнопки указывается название лобби, в котором на данный момент находится пользователь. В правой части «шапки» располагается ссылка на редактирование профиля пользователя.

При переходе в комнату, плеер играющим текущим треком находится в нижней части экрана. В левой части комнаты располагается окно с видеорядом для текущего трека или с обложкой. Вкладка со списком пользователей находится в правой части экрана. Окно со списком треков в очереди, а также с полем для ввода ссылки на добавление пользователя открывается при нажатии на кнопку «очередь» слева над окном виде ряда/изображения альбома.

3.1.2Интерфейсы аппаратного обеспечения

В качестве устройства для доступа к данному приложению используется любой компьютер с современным веб-браузером и доступом в сеть интернет. Также в качестве клиентского устройства может служить веб-браузер смартфона.

Аппаратно-программная среда для исполнения веб-приложения является выделенное место на виртуальном сервере с доступом к вычислительным ресурсам.

3.1.3Интерфейсы программного обеспечения

На данном этапе разработки приложение не будет иметь собственного API для взаимодействия со сторонними сервисами, но в приложении будет реализована логика, использующая сторонние API для реализации логики работы.

Самым первоначальным API является использование технологии OAuth для регистрации пользователя через такие сторонние сервисы как: Google, Twitter, Twitch, Discord и Soundcloud.

Вторым инструментом является API и SDK сервиса Soundcloud, при помощи SDK сервиса имеется возможность легко взаимодействовать с ресурсами сервиса для добавления в собственную логику. В данном случае из ссылки, указанной пользователем, будет извлекаться и воспроизводиться трек со всей сопутствующей информацией. Второй программный интерфейс относится к сервису YouTube. Для реализации логики добавления трека в очередь и дальнейшее его воспроизведения из ссылки на YouTube будет использоваться API элемента Iframe.

3.1.4Интерфейсы взаимодействия

За отображение и рендеринг элементов в приложении отвечает библиотека React, за серверную логику Node.js, а в качестве хранилища активных данных – MongoDB.

Данное приложение реализует концепцию взаимодействия с хранилищем данных через Node.js посредством моделей и контроллеров. Для реализации манипуляции данными БД будут разработаны сервисы, сокеты и контроллеры. React использует данные команды с помощью передачи запросов через Express.

Для реализации персистентности данных из MongoDB в PostgreSQL будет реализовано промежуточное ПО, реализующее синхронизацию данных по заданному расписанию.

3.2 Функциональные требования

Таблица 3 – Функциональные требования

ФТ-1:	Приложение должно иметь единое серверное API для возможности дальнейшего масштабирования.
ФТ-2:	Веб-приложение должно обладать адаптивный дизайн под все современные разрешения экранов мобильных устройств.
ФТ-3:	Каждый сложно составной компонент React должен быть покрыт тестами.
ФТ-4:	Должна быть реализована система журналирования на уровне базы данных.
ФТ-5:	Все сервисы должны быть покрыты Unit-тестами.
ФТ-6:	В интерфейсе очереди должен быть реализован активный поиск по названию трека.
ФТ-7:	Функция авторизация через сторонние сервисы должна осуществляться через следующие платформы: Google, Twitter, Twitch, Discord, Soundcloud.
ФТ-8:	Добавление аудио-визуального контента должно осуществляться через ссылки на источник следующих платформ: YouTube, Soundcloud.

3.3 Требование к производительности

Таблица 4 – Требования к производительности

ТП-1:	При первоначальном релизе приложение должно стабильно работать с условием 10000 единовременных активных пользователях.
ТП-2:	Весь контент, отдаваемый клиенту, должен быть оптимизирован сборщиком. Минифицирован CSS и все отдаваемые JS-скрипты, векторная и растровая графика должна быть оптимизирована и минифицирована.

3.4 Проектные ограничения

Таблица 5 – Проектные ограничения

ПО-1:	Веб-приложение доступно для следующих версий браузеров и новее: Edge 11, Firefox 45, Chrome 49, Safari 9, Opera 36, iOS Safari 9, Chrome for Android, Firefox for Android, Android browser, Samsung Internet 5.
ПО-2:	Северное окружение будет размещаться на машине следующей конфигурации: ЦП 2.1 x4, ОЗУ 16 Гб, SSD 500 Гб, HDD 2 Тб.

3.5 Нефункциональные требования

Таблица 6 – Нефункциональные требования

НТ-1:	Все элементы интерфейса взаимодействия должны быть доступны для навигации с клавиатуры.
НТ-2:	Приложение должно быть доступно для людей, использующие скринридеры для взаимодействия с программными интерфейсами.
НТ-3:	Разрабатываемые React-компоненты должны иметь семантичную разметку.

Вопросы для самоконтроля

1. Для чего необходим составлять техническое задание?
2. В каком разделе необходим размещать информацию об используемых работе источниках (литература, ссылки на нормативы и стандарты)?
3. Что описывается в главе «Детальные требования»?
4. Должно ли техническое строго следовать регламенту составления?
5. Какая информация должна содержаться в главе «Общие описания»?
6. Верно ли высказывание, что в главе «Нефункциональные требования» описываются основные требования кодовой составляющей проекта?
7. Входит ли процесс подробного формирования ТЗ в норму практик Agile-разработки?
8. В чём коренные различия между MVP и корпоративным продуктом?
9. В каком разделе описывается вся терминология и акронимы, использующиеся в проекте?
10. В каком разделе описываются группы пользователей и участников разработки проекта?

Задание

Составьте техническое задание к выбранной Вами информационной системе согласно с приведённым в данном методическом указании примером.

Лабораторная работа №7 - Проектирование диаграмм классов.

Прежде чем приступать к рассмотрению диаграмм классов, дадим определение термину «класс». *Класс* – это «макет», с помощью которого можно создать *объект*. Класс определяет, что может делать объект и какими характеристиками он обладает.

Диаграмма классов даёт статическое представление Вашего приложения. Она описывает типы объектов в системе, а также описывает различные типы связей между классами. При моделировании с помощью представления диаграммы классов, Вам доступны все приёмы и концепции, которые происходят из мира объектно-ориентированного программирования. Класс может ссылаться на иной класс. Класс может иметь собственные объекты или может наследоваться от других классов.

UML диаграмма классов даёт представление о программном обеспечении, отображая классы, атрибуты, операции (методы), а также связи между ними. Все данные концепции определены в диаграмме классов в виде сепарированных сущностей в едином графическом объекте (полное описание класса).

Проектирование диаграмм классов помогает разработчикам структурировать логическую организацию программного обеспечения до непосредственного написания кода. Имея современные инструменты проектирования, Вы имеете возможность экспортировать ваш макет в реальный код, на необходимом вам ООП языке программирования.

Прежде чем углублять подробно в описание концепций проектирования диаграмм классов, перечислим основные преимущества, стоящие за данным подходом:

- Диаграмма классов способна проиллюстрировать модель данных для сколь угодно сложных информационных систем.
- Даёт представление о том, каким образом устроено приложение, прежде чем углубляться в кодовую реализацию, что способствует сокращению времени на решение проблем при поддержке ИС.
- Способствует пониманию базовых концепций функционирования приложения.
- С помощью графического представление объектов, есть возможность выделения первостепенных объектов реализации и вторичных.
- Помогает в ориентировании логики проекта не только разработчикам, но и остальным членам команды.

Основные UML элементы диаграммы классов:

1. Имя класса
2. Атрибуты
3. Операции

Имя класса

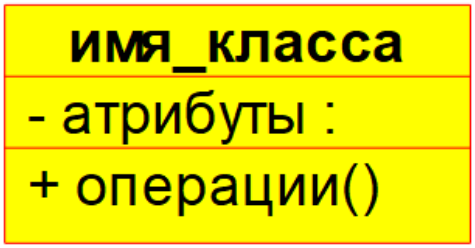


Рисунок 7.1 – Пример объекта класса

Имя класса необходим только для визуального представления класса. Название пишется в самом верхнем поле объекта класса. Класс является «макетом» объекта, который может имплементировать связи, атрибуты, операции и семантику. Класс обозначается как прямоугольник, включающий имя класса, атрибуты и операции, разделённые на отдельные поля.

При создании класса необходим придерживаться следующих правил:

1. Имя класса всегда должно начинаться с заглавной буквы.
2. Имя класса всегда должно находиться по центру поля.
3. Имя класса должно быть написано жирным шрифтом.
4. Имена абстрактных имён классов должны быть выделено курсивом.

Атрибут. Является именованным свойством класса, которое определяет, каким образом будет смоделирован объект. Производный атрибут вычисляется с помощью других атрибутов. Например, возраст может быть легко вычислен по дате рождения.

Характеристики атрибутов:

– Атрибуты в основном описываются для внесения большей видимости в класс диаграммы.

– Публичный, приватный, защищённый и пакетный – 4 типа области видимости, который отображаются соответствующими знаками: +, -, #, ~.

– Область видимости описывает доступность атрибута класса.

– Атрибут должен иметь осмысленно название, полностью описывающее его суть в структуре класса.

Связи. В нотации UML присутствует 3 типа связей:

– зависимости;
– обобщение;
– ассоциации.

Зависимости. Зависимость означает связь между двумя и более классами (рис. 7.2), в которых изменение одного класса может повлиять на изменения в других. Тем не менее связь такого типа является слабой.

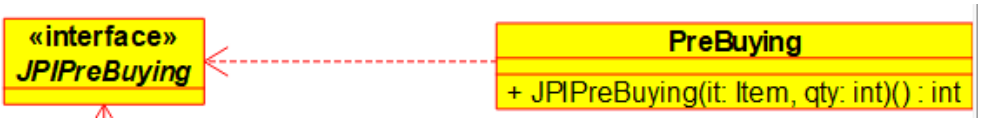


Рисунок 7.2 – Пример зависимости.

Обобщение (генерализация). Обобщение – это таксономическое отношение между более общим элементом и более конкретным. Каждый

экземпляр конкретного элемента также является непрямым экземпляром обобщённого элемента (рис. 7.3). Таким образом, конкретизированный элемент косвенно обладает свойствами обобщённого элемента.

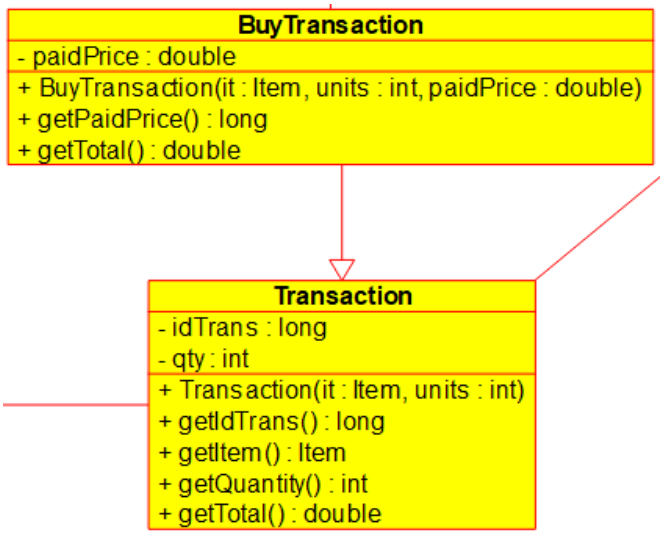


Рисунок 7.3 – Пример обощения

Ассоциация. Данный вид связи представляет статическую связь между двумя классами. Например, *работник* работает в *организации*. Правила построения ассоциации:

– В большинстве случаем, ассоциация является глаголом или глагольным выражением.

– Название ассоциации должно явно выражать роль, играющую для класса, к которому проведена ассоциация.

– Обязательно для рефлексивных типов ассоциации (рис. 7.4).

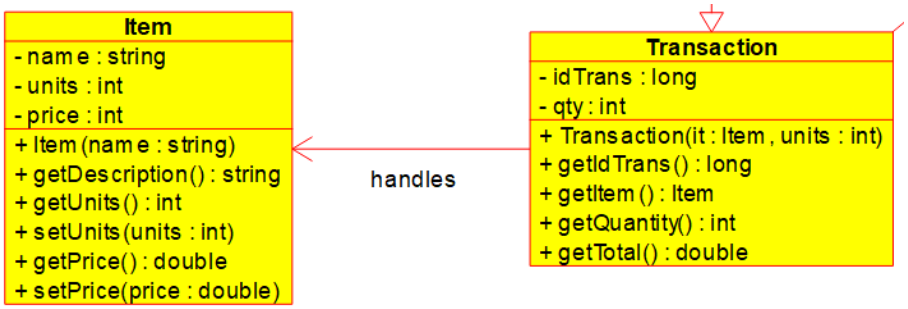


Рисунок 7.4 – Ассоциация между рефлексивными классами

Агрегация. Специальный тип ассоциации, который отображает связь между целым классом и его зависимой частью (рис. 7.5).

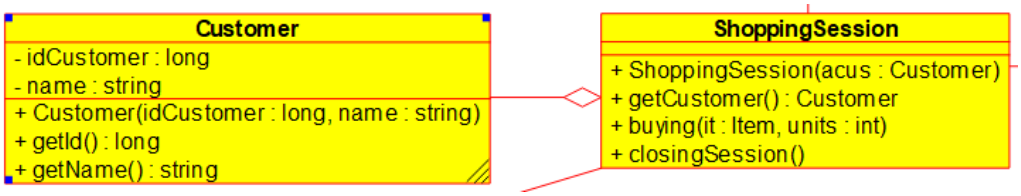


Рисунок 7.5 – Агрегация между классами

Пример из рисунка 7.5 описывает агрегационную связь между классами *покупатель* и *сессия покупки*. В агрегации частичные классы (*покупатель* в текущем примере) никогда полностью не зависят от класса-контейнера (*сессия покупки*) на протяжении всего жизненного цикла системы. В данном примере класс *сессии покупки* будет существовать, даже если ни одного покупателя не будет существовать.

Композиция. Это особый тип агрегации, который обозначает сильную принадлежность между двумя классами, когда один класс является частью другого.

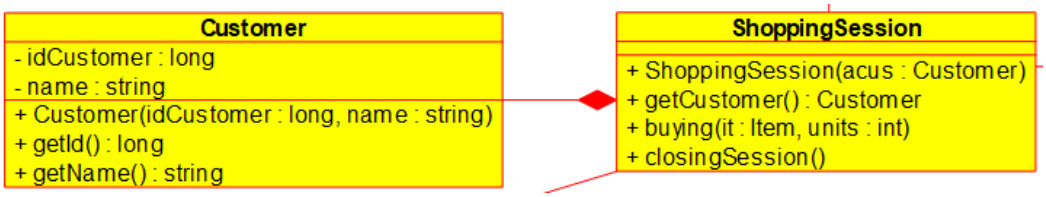


Рисунок 7.6 – Композиция между двумя классами

По примеру из рисунка 6, если *покупатель* является частью класса *сессия покупки*, то 1 экземпляр класса *покупатель* может относиться только к 1 экземпляру класса *сессия покупки*. Если класс *сессия покупки* не функционирует, то все объекты класса *покупатель* также не будут существовать.

Агрегация сообщает о связи, где дочерний класс может существовать отдельно от родительского класса. Например, есть два класса: *Автомобиль* (родитель) и *Седан* (дочерний). Если удалить класс *Автомобиль*, дочерний класс *Седан* будет существовать.

Композиция отображает связь, где объект дочернего класса никогда не будет существовать в отрыве от родительского. Например, есть два класса: *Дом* (родительский) и *Комната* (дочерний). *Комната* никогда не будет отделена от *Дома*.

Диаграмма классов является самой важной UML диаграммой, используемой для описания технической реализации программного продукта. Есть много свойств, которые следует учитывать в процессе моделирования диаграммы классов. Данные свойства представляют различные аспекты программного продукта.

Практики, которые следует учитывать при построении диаграммы классов:

– Имя, данное диаграмме классов, должно полностью описывать суть, моделируемых объектов.

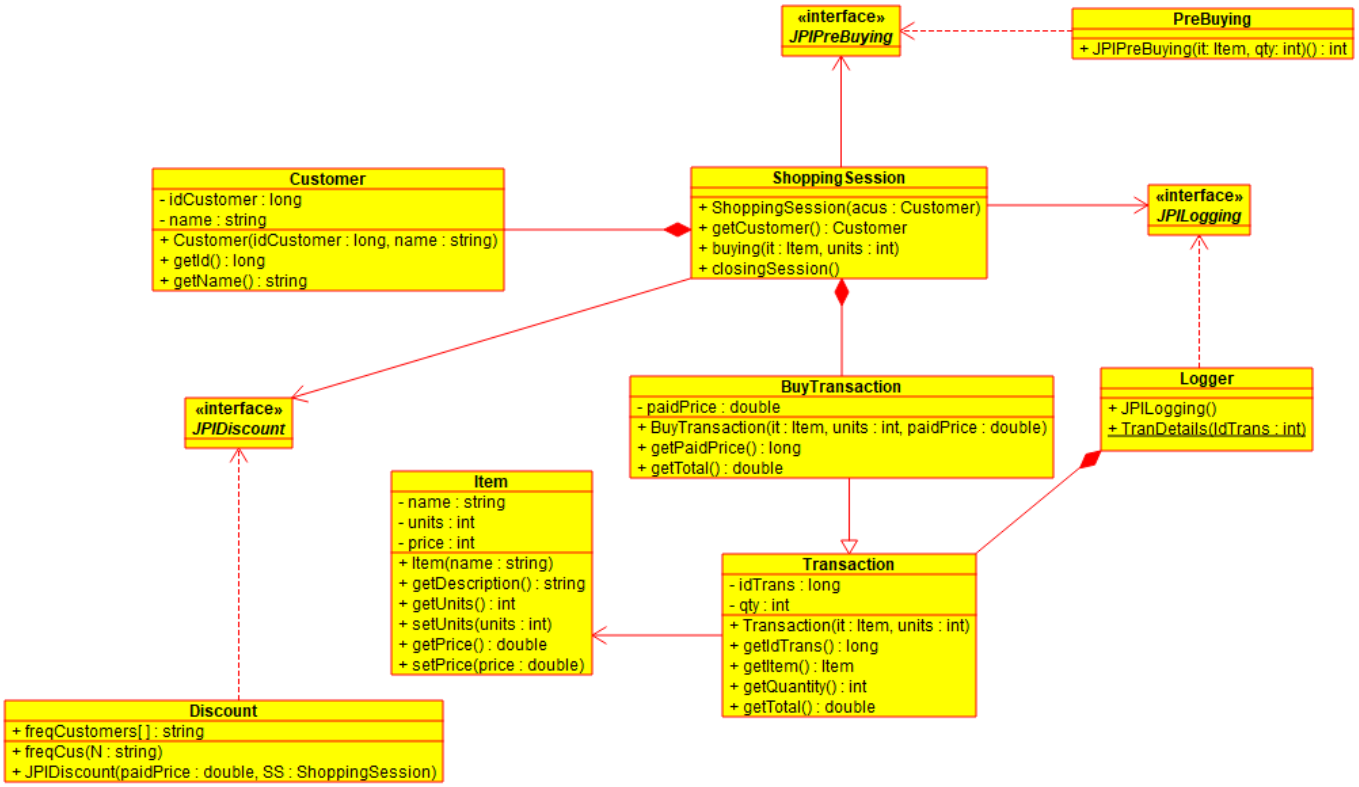
– Связи между каждым элементом должны быть определены заранее.

– Каждый класс должен относиться к своей сфере ответственности в моделируемой системе.

– Для каждого класса должно быть указано минимальное количество свойств. Нежелательные свойства могут легко усложнить диаграмму.

– Для большей конкретики используйте объект *примечание*, в котором подробнее описывается определённые аспекты диаграммы.

– Прежде чем производить моделирование в специализированном программном обеспечении, нарисуйте первоначальный концепт на бумаге,



это может сэкономить определённое количество времени.

Рисунок 7.7 – Диаграмма классов

На рисунке 7.7 представлена диаграмма классов оформления покупки в онлайн-магазине. Система содержит 8 классов и 3 интерфейса. Опишем назначение каждого класса и интерфейса.

JPIPreBuying – интерфейс «предпокупки». Необходим для реализации класса добавления товара в корзину, обладает параметрами идентификатора товара и его количество.

PreBuying – класс, реализующий интерфейс *JPIPrebuying*.

JPILogging – интерфейс журналирования действий. Необходим для фиксирования действий пользователя во время сессии покупок.

Logger – класс, реализующий интерфейс *JPILogging*. Реализует метод описания информации о конкретной транзакции на покупку (принимает идентификатор транзакции в качестве параметра).

JPIDiscount – интерфейс, реализующий формирования скидки на покупку. В качестве параметров принимает сумму всех позиций и объект сессии покупки.

Discount – класс, реализующий интерфейс *JPIDiscount*. Обладает свойством, являющимся массивом пользователей, обладающих скидкой. Обладает методом, определяющим обладает ли пользователь скидкой.

ShoppinSession – класс, формирующий объект сессии клиента, реализует метод определения клиента, оформления покупки, а также метод завершения сессии покупок.

BuyTransaction – класс, формирующий объект транзакции покупки товаров. Реализует метод получения цены позиций товара, метод формирования конечной цены на оплату, а также реализует собственный конструктор, принимающий в качестве параметра список позиций товара и сформированную цену.

Transaction – класс, реализующий объект добавления определённо позиции товара на покупку. Реализует методы по получению идентификатора транзакции, по получению позиции товара, по получению количества товара определённой позиции и по получению цены за количество указанных позиций, обладает параметром идентификатора транзакции количества позиций товара.

Item – класс, реализующий объект товара. Обладает параметрами названия товара, количества товара в 1 позиции, а также цены за единицу товара. Реализует собственный конструктор, принимающий название в качестве параметра. Реализует методы по получению описания товара, по получению и установке количества единиц товара, а также по получению и установке цены товара.

Customer – класс, реализующий объект покупателя. Обладает параметрами идентификатора и названия. Обладает конструктором, принимающим в качестве параметра идентификатор и имя покупателя. Реализует методы по получению идентификатора и п получению имени товара.

От описания сущностей системы, перейдём к описанию связей между ними. Между интерфейсами и классами их реализующих проходит стандартная операция реализации, о чём свидетельствуют пунктирные линии зависимости.

Класс ShoppingSession ссылается на все интерфейсы, приведённые в системе, тем самым обозначая что объявление нового объекта будет производиться через тип интерфейса.

Между классами *ShoppingSession* и *Customer, ShoppingSession* и *BuyTransaction* проходит композитная связь. Это означает, что *Customer* и *BuyTransaction* являются частями реализации *ShoppingSession.*

Transacion также является частью реализации объекта *Logger.*

Между объектами *BuyTransaction* и *Transaction* проходит связь генерализации. Это означает, что каждый экземпляр *BuyTransction* является непрямым экземпляром *Transaction.*

Для того чтобы начать проектировать диаграмму классов в среде Umbrello, для начала необходимо добавить новую диаграмму классов в разделе логических представлений (рис. 7.8).

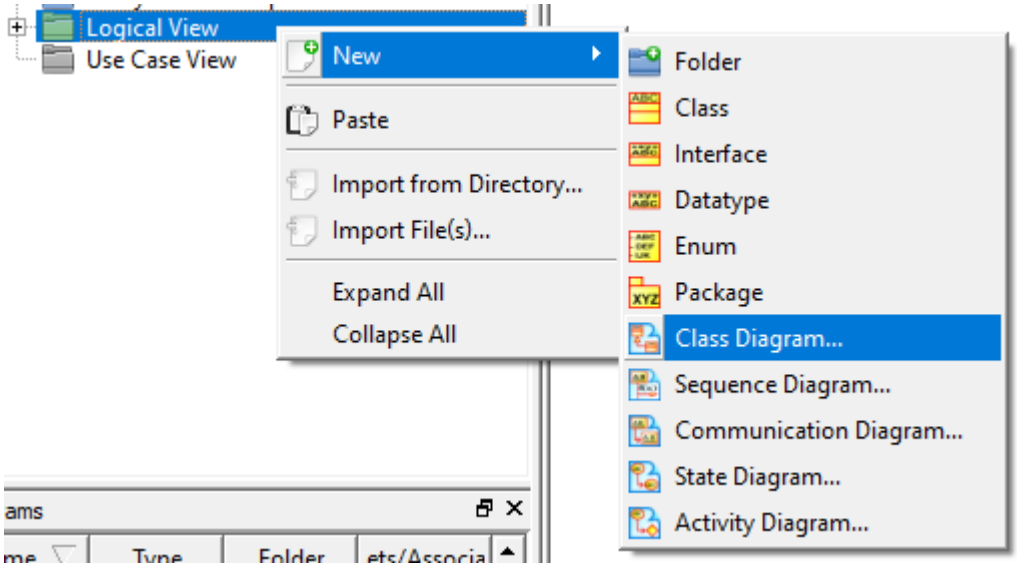


Рисунок 7.8 – Добавление диаграммы классов

После добавления, необходимо добавить классы и интерфейсы для вашей диаграммы. Добавление можно производить нажатием на соответствующую иконку в верхней панели или нажатием правой кнопкой мыши и выбором соответствующего пункта из контекстного меню. Дав имя классу, необходим добавить его атрибуты и методы. Для того чтобы добавить новый атрибут, необходимо нажать правой кнопкой мыши по классу и выбрать из контекстного меню пункт атрибута (рис. 7.9).

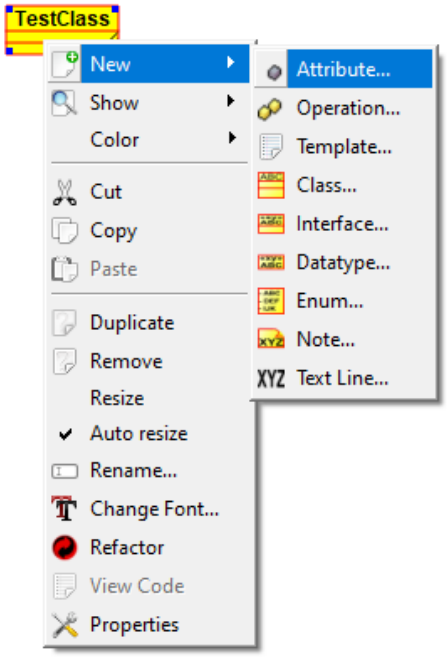


Рисунок 7.9 – Добавление атрибута

Аналогичным образом производится добавлением методов (операций – Operations…).

При добавлении нового атрибута откроется окно, где необходимо указать характеристики добавляемого атрибута (рис. 7.10). Из всех параметров доступных в окне, ключевыми является Type (тип атрибута),

Name (название атрибута), Visibility (область видимости атрибута). Прочие параметры являются не столь существенными при базовом построении диаграммы классов.

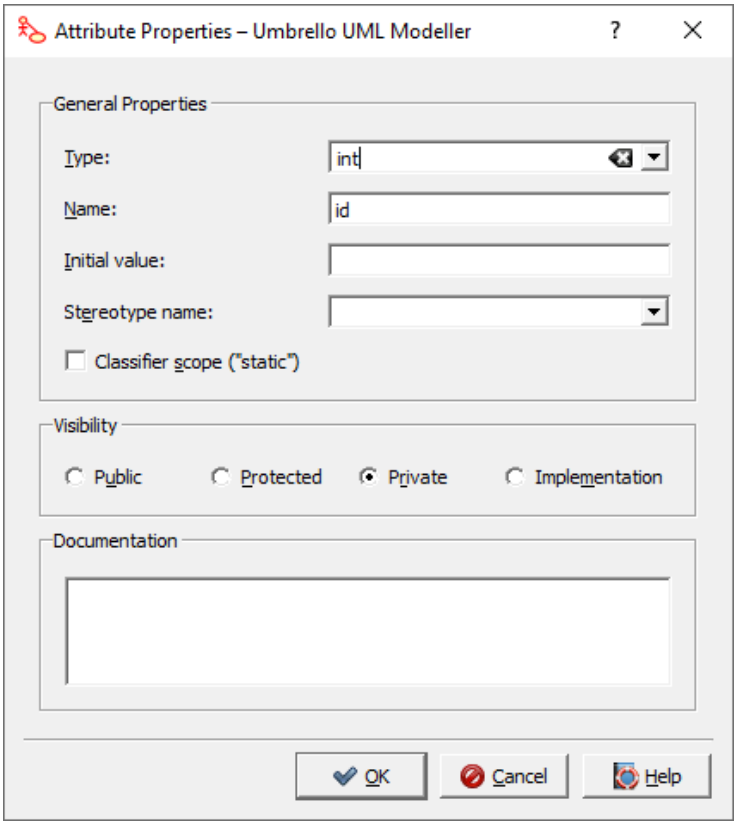


Рисунок 7.10 – Настройка атрибута

Схожее окно будет доступно при добавлении метода класса (рис. 7.11).

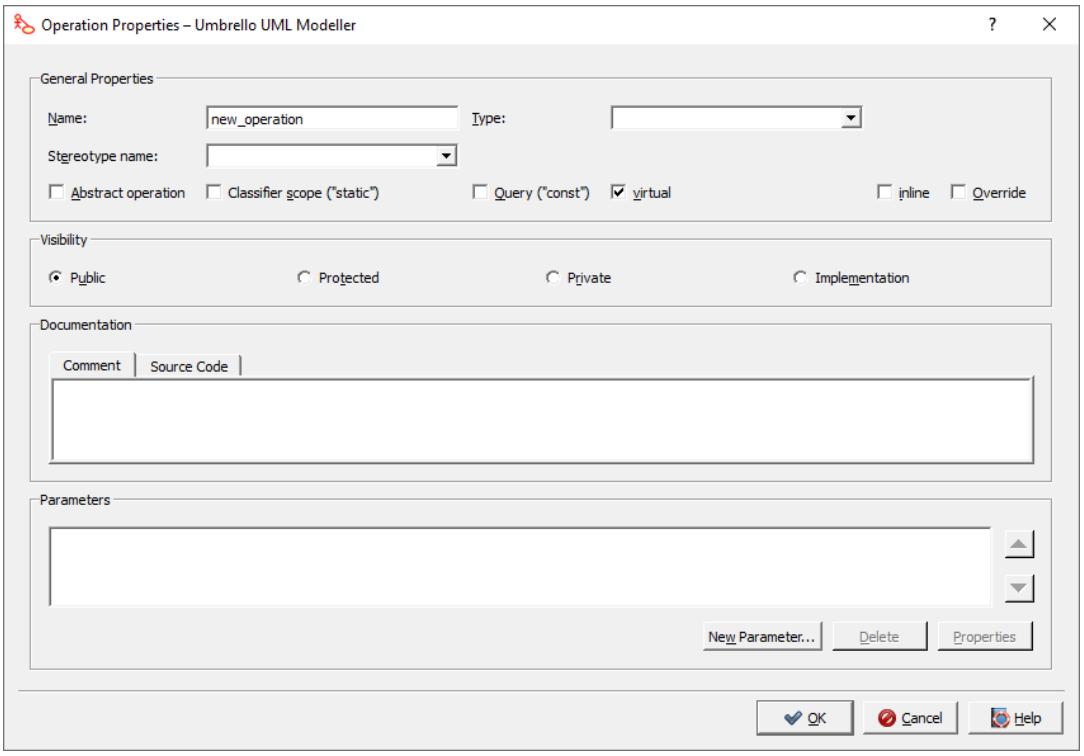


Рисунок 7.11 – Настройка метода

В данном контексте необходим дать название вашему методу (Name), определить возвращаемый тип (Type) и область видимости (Visibility). Прочие параметры остаются для настройки на усмотрения проектировщика, в виду того, что никакой дополнительной визуальной информации о методе класса, данные параметры не содержат. Данные параметры являются ключевыми, если стоит задаче об экспорте класса UML в программный код.

Финальным этапом является проведение зависимостей между классами. Все типы зависимостей расположены на верхней панели управления (рис. 7.12). Их типы слева на право: ассоциация, прямая ассоциация, обобщение (генерализация), композиция, агрегация, содержание.

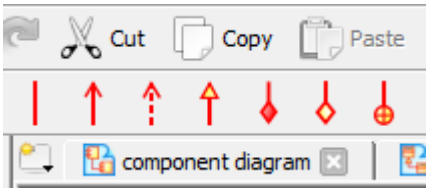


Рисунок 7.12 – Линии ассоциации

Для того чтобы провести ассоциацию нужного вам типа, сначала нажмите левой кнопкой мыши на соответствующую иконку ассоциации, далее на класс, от которого проводить ассоциацию и в конце на класс, к которому проводится ассоциация.

В итоге должна получиться связь по примеру изображённого на рисунке 7.13.

Рисунок 7.13 – Пример проведённой ассоциации

Чтобы дать название действию ассоциации, нажмите правой кнопкой мыши по линии и выберете пункт Properties контекстного (рис. 7.14), после чего откроется окно настроек параметров ассоциации (рис. 7.15).

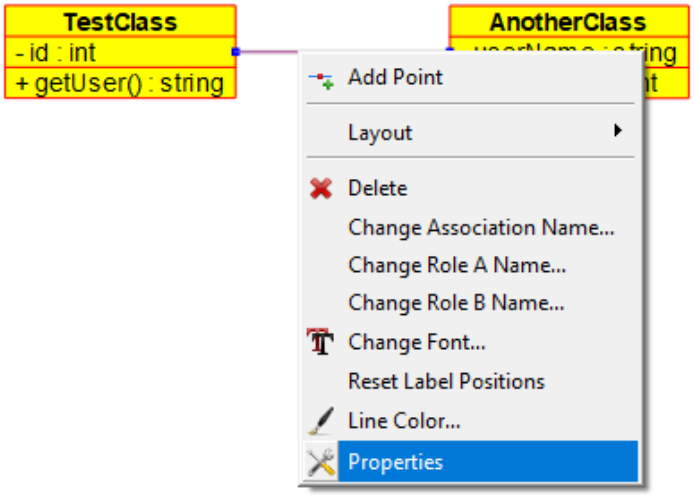


Рисунок 7.14 – Настройка ассоциации

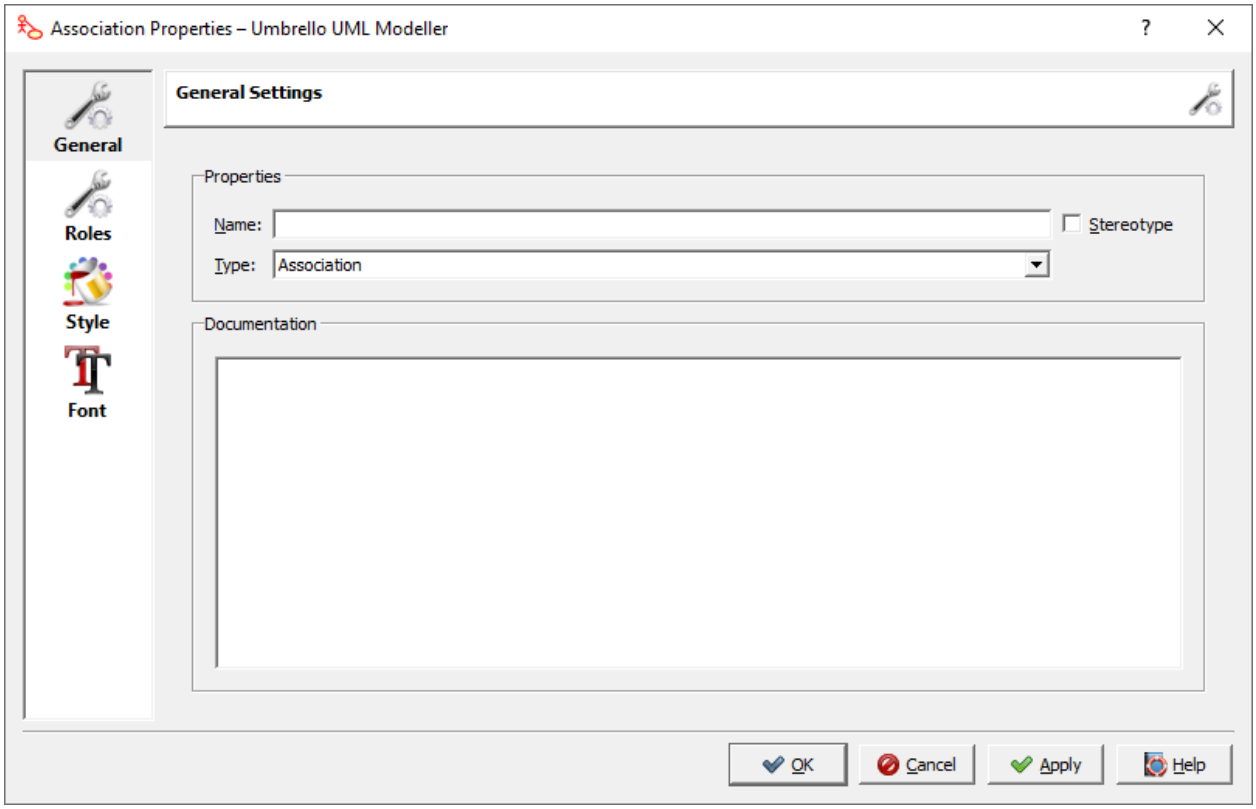


Рисунок 7.15 – Меню настройки ассоциации

Вопросы для самоконтроля.

1. Что такое класс?
2. Какой смысле содержит диаграмма классов?
3. Какими элементами обладает класс?
4. Дайте определение зависимости.
5. Дайте определение ассоциации.
6. Дайте определение обобщению.
7. Дайте определение Агрегации.
8. Дайте определение Композиции.
9. Какое количество свойств должно быть у класса?
10. Как должна именоваться ассоциация?

Задание.

Спроектируйте диаграмму классов для двух прецедентов вашей системы. В случае возникновения сомнения о том, какой тип связи должен проведём между классами в вашей системе, используйте обычную ассоциацию (прямая линия). Дайте описание классов вашей диаграммы, а также из связей.

Лабораторная работа №8 - Проектирование диаграмм видов деятельности.

Диаграммы видов деятельности отображают последовательные и параллельные процессы. Они полезны для моделирования бизнес-процессов, последовательностей выполнения задач, потоков данных и сложных алгоритмов.

Основные обозначения UML для *диаграмм видов деятельности* (activity diagrams) продемонстрированы на рис 8.1. На таких диаграммах отображаются *действия* (action), *точки ветвления* (fork/branch), *разделы* (partition), *объединения* (join) и *объекты* (object). В действительности, на данных диаграммах изображают последовательность действий, часть из которых может выполняться параллельно. Большая часть обозначений довольна очевидна, но некоторые требуют пояснения.

– При завершении вида деятельности выполняется автоматический завершающий переход.

– На диаграммах могут отображаться как потоки управления, так и потоки данных.

Можно построить несколько диаграмм деятельности для одной и той же системы, причем каждая из них будет фокусироваться на разных аспектах системы, показывать различные действия, выполняющиеся внутри ее. Упоминая динамику, мы подразумеваем поведение системы в целом или ее частей. Говоря более формально, диаграммы активности, в общем-то, не имеют монополии на описание поведенческих особенностей динамических частей системы. Для этой же цели могут использоваться еще диаграммы прецедентов, последовательности, кооперации и состояний. Почему же речь идёт именно о диаграмме активности?

Именно на диаграмме деятельности представлены переходы потока управления от одной деятельности к другой. Это, по сути, разновидность диаграммы состояний, где все или большая часть состояний являются некоторыми деятельностями, а все или большая часть переходов срабатывают при завершении определенной деятельности и позволяют перейти к выполнению следующей. Как мы уже говорили (повторение - мать учения), диаграмма деятельности может быть присоединена к любому элементу модели, имеющему динамическое поведение. Кстати, исходя из вышесказанного, логичнее говорить не "диаграмма деятельности", а "диаграмма деятельностей" - во множественном числе. А еще мы предполагаем, что читатель понимает смысл понятий "деятельность", "переход" и "объект". Об объектах как об экземплярах классов мы уже говорили ранее. Понятия же деятельности (activity) как протяженного во времени составного (неатомарного) вычисления (действия, action) и перехода как передачи контроля, надеемся, понятны интуитивно, без дополнительных объяснений.

Диаграммы деятельности позволяют моделировать сложный жизненный цикл объекта, с переходами из одного состояния (деятельности) в другое. Но этот вид диаграмм может быть использован и для описания

динамики совокупности объектов. Они применимы и для детализации некоторой конкретной операции, причем, как мы увидим далее, предоставляют для этого больше возможностей, чем «классическая» блок-схема. Диаграммы деятельности описывают переход от одной деятельности к другой, в отличие от диаграмм взаимодействия, где акцент делается на переходах потока управления от объекта к объекту.

В UML *диаграммы видов деятельности* (activity diagrams) обеспечивают систему обозначений для последовательности видов деятельности. Эти обозначения можно использовать в различных целях (в том числе для визуализации шагов компьютерного алгоритма), однако они особенно полезны для построения диаграммы прецедентов и графиков выполнения проекта.

Приведём несколько рекомендаций, касающихся моделирования видов деятельности.

– Данная методология оправдана для визуализации очень сложных процессов, в которых задействовано множество участников. Простые процессы лучше представлять в виде описания прецедентов.

– При моделировании бизнес-процесса используйте переходы между уровнями. На нулевом уровне можно показать действия на самом высоком уровне абстракции. Тогда диаграмма останется простой и понятной. Детализацию можно выполнить на уровнях 1, 2 и далее.

– При построении конкретной диаграммы придерживайтесь одного уровня абстракции. Не следует смешивать на одной диаграмме понятия разного уровня.

Рассмотрим диаграмму видов деятельности приложения Sound Room.

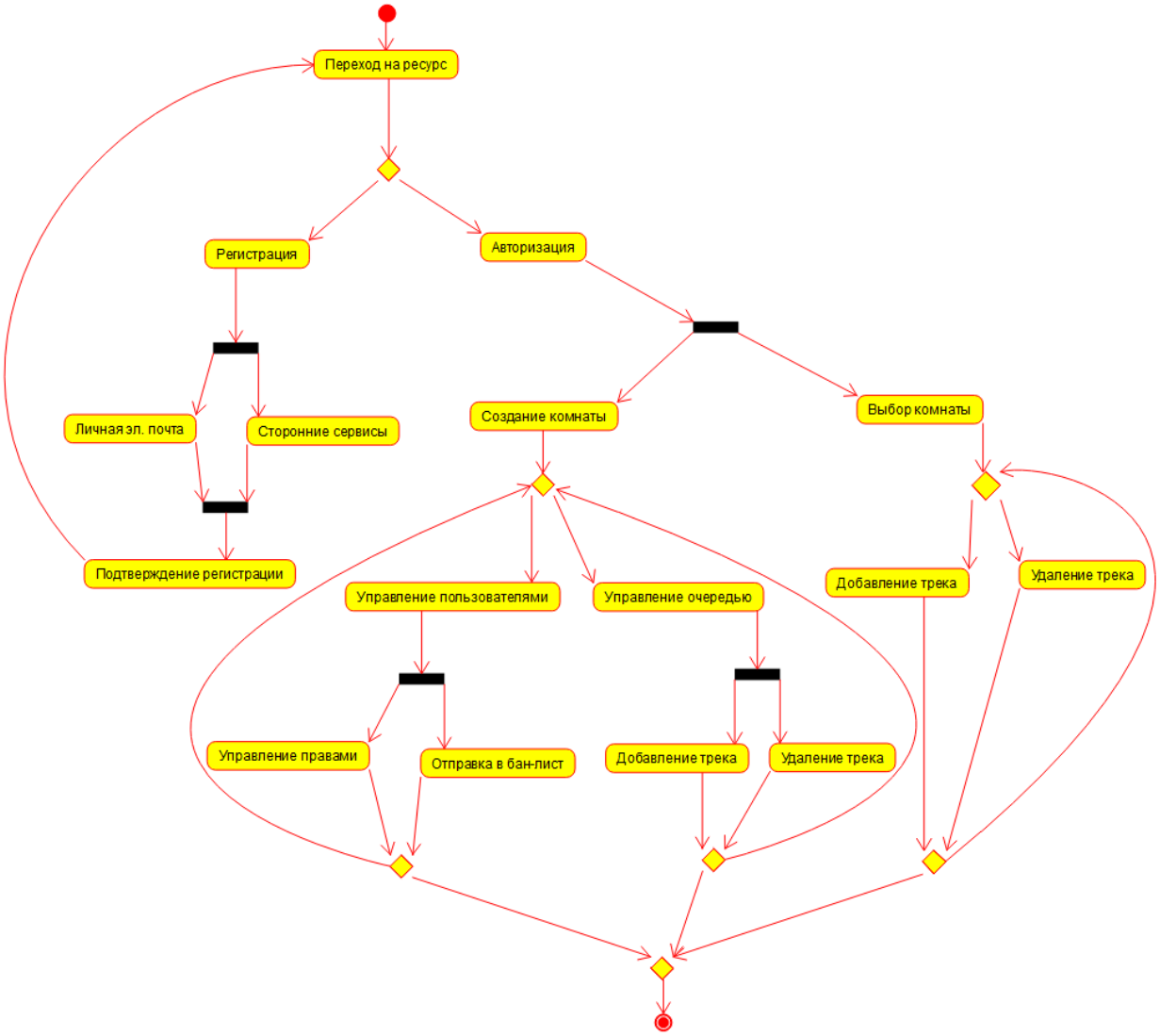


Рисунок 8.1 – Диаграмма видов деятельности Sound Room

Данная диаграмма описывает жизненный цикл нахождения пользователя на ресурсе Sound Room.

На самом верхнем уровне находится блок входа в систему, который обозначается красным закрашенным кругом. Вне зависимости от цветовой палитры в различных средах проектирования, объекты UML имеют унифицированный вид.

После запуска системы, система переводит пользователя на главную страницу ресурса. Следующим этапом пользователь выбирает 1 из сценариев. Выбор сценария изображён в виде точки ветвления (branch), которая обозначается на диаграмме чёрным прямоугольником. Точка ветвления определяет, что пользователь может зарегистрироваться на ресурсе или авторизоваться со своей учётной записью.

В рамках диаграммы видов деятельности существует 2 типа ветвления и объединения. В русском языке данные термины переводятся идентично, поэтому для лучшей ясности в описании диаграммы будем указывать английское слово типа связи. Ветвление делится на fork и branch. Fork (чёрный прямоугольник на схеме) следует использовать тогда, когда идёт разветвление схожих по поведение действий и в сценарий использования не

подразумевает обращения к данному блоку ветвления из других частей диаграммы. Branch следует использовать, когда происходит разделение на отличающиеся по поведению действия, а также в том случае, когда к данному блоку будет производится обращение из других уровней диаграммы. По этой же аналогии отличаются типы объединений join и merge. Join используется, когда происходит объединение схожих по поведению действий без слияния действий из других уровней или веток. Соответственно branch используется, когда объединяются действия из разных веток или отличающиеся по поведению.

Сценарий регистрации. После выбора данного действия, пользователю снова дают выбор, что отображено в виде точке ветвления. Пользователь может зарегистрироваться используя личную почту или используя аккаунт сторонних сервисов (социальные сети).

Вне зависимости от выбора пользователя, в случае успешной регистрации происходит объединение событий (join), которое также, как и точка ветвления обозначается чёрным прямоугольником, и пользователя перемещают к действию перехода на главную страницу ресурса.

При выборе авторизации происходит разделение типа fork на создание комнаты (лобби) или выбора комнаты.

Создание комнаты. После создание комнаты, пользователь автоматически становится её администратором, после чего ему доступны 2 действия: управление пользователями и управление очередью. Данное разветвление имеет тип branch.

Управление пользователями. При выборе данного действия пользователю доступны 2 действия над пользователем: управление его правами или отправка в бан-лист. Данное разветвление имеет тип fork.

После осуществления необходимого действия над пользователем комнаты, администратор переходит на блок разветвления и одновременно объединения. Данный тип связи является совмещённым branch/merge. После которого пользователь может вновь вернуться к блоку выбора действий в комнате или перейти к блоку завершения деятельности в системе.

Управление очередью. Данный блок действий зеркален блоку, находящемуся на том же уровне «управление пользователями». Отличие заключается в действиях, которые есть у пользователя для взаимодействия с очередью: добавление трека или удаление трека. По окончанию любого из действий пользователь также может вернуться в блок выбора действий или перейти к этапу завершения деятельности.

Выбор комнаты. При данном сценарии пользователь изначально не имеет административных прав в комнате, поэтому ему доступны 2 действия: *добавление трека* или *удаление терка*. Тип ветвления – Branch.

По завершению одного из действий, пользователь может повторить данный блок ветвления или перейти на этап завершения деятельности. Тип связи - совмещённым branch/merge.

Завершением деятельности пользователя внутри ресурса является закрытие веб-браузера, которое провоцирует действие выхода из системы.

Вопросы для самоконтроля

11. Дайте определение диаграмме видов деятельности в нотации UML.
12. Чем диаграммы деятельности отличаются от блок-схем?
13. Применимы ли диаграммы деятельности безотносительно к ООП?
14. Какие существуют основные обозначения в диаграмме видов деятельности?
15. Приведите типы ветвления, объясните их применение.
16. Приведите типы объединения, объясните их смысл.
17. Для чего применяют диаграмму видов деятельности?
18. В каких случаях уместно моделирование при помощи диаграммы видов деятельности?
19. На каких уровнях лучше всего детализировать описание действий системы?
20. Что может подразумеваться в объекте действия (activity) диаграммы видов деятельности?

Задание

Спроектируйте диаграмму видов деятельности для системы Вашей предметной области. После моделирования в среде Umbrello, приведите описание связей, блоков и действий в Вашей диаграмме по примеру, приведённого в данной главе.

Лабораторная работа №9 - Проектирование объектов GRASP

Процесс объектного проектирования иногда описывают следующим образом.

Сначала определяются требования и создаётся модель предметной области, затем добавляются методы программных классов, описывающие передачу сообщения между объектами для удовлетворения требований.

Такое описание мало полезно на практике, поскольку в нём не учтены основополагающие принципы и вопросы, лежащие в основе этого процесса. Вопрос определения способов взаимодействия объектов и принадлежности методов чрезвычайно важен и отнюдь не тривиален. Освоение ООП требует изучения большого количества принципов, определяющих достаточно большое количество степеней свободы. В этом поможет применение паттернов (шаблонов) – именованных описаний типичных реализаций.

Один из принципов написания шаблонов имеет аббревиатуру GRASP (англ. General Responsibility Assignment Software Patterns – общие шаблоны распределения ответственностей).

Ответственность в программном обеспечении является очень важной концепцией и касается не только классов, но и модулей, а порой и целых систем. Мышление с точки зрения ответственности – популярный способ думать об архитектуре программного обеспечения. Для того чтобы думать в таком направлении нужно задаваться вопросами:

– За что ответственен этот класс / модуль / компонент / система?

– За какую реализацию ответственен этот объект?

– Нарушается ли принцип единой ответственности в данном конкретном контексте?

Но для того, чтобы иметь ответ на данные вопросы, необходим иметь понимание фундаментального вопроса о том, что такое ответственность в контексте программного обеспечения.

Одно из определений данного термина можно интерпретировать следующим образом.

Ответственность в контексте ПО – это обязанность выполнять задачу или знать информацию.

Из данного определения можно чётко выделить различия между *поведением* (действием) и *данными* (знанием).

Выполнение ответственности объекта рассматривается как:

1. Самостоятельные действия – создать объект, обработать данные, произвести некоторые вычисления;
2. Инициировать и координировать действия с другими объектами.

Знание ответственности объекта может быть определено как:

1. Данные частного и публичного объекта;
2. Ссылки на связанные объекты;
3. Данные, которые он может извлечь.

Для демонстрации реализации данных тезисов, проанализируйте листинг, представленный на рис. 9.1 и 9.2.

```
Entity,  // знание
IAggregateRoot  // знание
{
    public Guid Id { get; private set; } // знание

    public string Email { get; private set; } // знание

    public string Name { get; private set; } // знание

    private readonly List<Order> _orders; // знание

    private Customer()
    {
        this._orders = new List<Order>();
    }

    // самостоятельные дейтсвия
    public Customer(string email, string name, ICustomerUniquenessChecker customerUniquenessChecker)
    {
        this.Email = email;
        this.Name = name;

        // действие - инициация и координация действий с другими объектами
        var isUnique = customerUniquenessChecker.IsUnique(this);
        if (!isUnique)
        {
            throw new BusinessRuleValidationException("Customer with this email already exists.");
        }

        this.AddDomainEvent(new CustomerRegisteredEvent(this));
    }
```

Рисунок 9.1 – Листинг реализации разделения ответственности

```
    // самостоятельные дейтсвия
    public void AddOrder(Order order)
    {
        // действие - инициация и координация действий с другими объектами
        if (this._orders.Count(x => x.IsOrderedToday()) >= 2)
        {
            throw new BusinessRuleValidationException("You cannot order more than 2 orders on the same day");
        }

        this._orders.Add(order);

        this.AddDomainEvent(new OrderAddedEvent(order));
    }

    // самостоятельные дейтсвия
    public void ChangeOrder(
        Guid orderId,
        List<OrderProduct> products,
        List<ConversionRate> conversionRates)
    {
        var order = this._orders.Single(x => x.Id == orderId);

        // действие - инициация и координация действий с другими объектами
        order.Change(products, conversionRates);

        this.AddDomainEvent(new OrderChangedEvent(order));
    }

    // самостоятельные дейтсвия
    public void RemoveOrder(Guid orderId)
    {
        var order = this._orders.Single(x => x.Id == orderId);

        // действие - инициация и координация действий с другими объектами
        order.Remove();

        this.AddDomainEvent(new OrderRemovedEvent(order));
    }

    // самостоятельные дейтсвия
    public GetOrdersTotal(Guid orderId)
    {
        return this._orders.Sum(x => x.Value);
    }
}
```

Рисунок 9.2 - Листинг реализации разделения ответственности

Существуют 9 паттернов GRASP:

1. Информационный эксперт (Information Expert);
2. Создатель (Creator);
3. Контроллер (Controller);
4. Слабое зацепление (Low Coupling);
5. Высокая связанность (High Cohesion);
6. Посредник (Indirection);
7. Полиморфизм (Polymorphism);
8. Чистая выдумка (Pure Fabrication);
9. Устойчивость к изменениям (Protected Variations).

1. Информационный эксперт (Information Expert)

Проблема: Каков основной принцип, по которому можно распределять обязанности между объектами?

Решение: возложите ответственность на класс, обладающий информацией, необходимой для ее выполнения.

В приведённом примере класс *Customer* ссылается на все *Orders* (заказы), поэтому класс *Customer* является естественным кандидатом по принятью ответственности за расчёт общей стоимости заказов (рис. 9.3).

```
1   public class Customer : Entity, IAggregateRoot
2   {
3       private readonly List<Order> _orders;
4
5       public GetOrdersTotal(Guid orderId)
6       {
7           return this._orders.Sum(x => x.Value);
8       }
9   }
```

Рисунок 9.3 – Листинг «Информационный эксперт»

Это самый базовый принцип. Идея заключается в том, что, если нет необходимых данных, следовательно, не выполняются требования и нет возможности распределить ответственность.

2. Создатель (Creator)

Проблема: кто создает объект A?

Решение: возложить на класс B ответственность за создание объекта A, если один из них является истинным (чем больше, тем лучше)

– B содержит или объединяет A;
– B записывает A;
– B тесно использует A;
– B имеет данные инициализации для A.

Пример:

```
1   public class Customer : Entity, IAggregateRoot
2   {
3       private readonly List<Order> _orders;
4
5       public void AddOrder(List<OrderProduct> orderProducts)
6       {
7           var order = new Order(orderProducts); // Creator
8
9           if (this._orders.Count(x => x.IsOrderedToday()) >= 2)
10          {
11              throw new BusinessRuleValidationException("You cannot order more than 2 orders on the same day");
12          }
13
14          this._orders.Add(order);
15
16          this.AddDomainEvent(new OrderAddedEvent(order));
17      }
18  }
```

Рисунок 9.4 – Листинг «Создатель»

Как вы можете видеть выше, класс *Customer* объединяет *Orders* (заказы) (нет заказов без клиентов), записывает заказы, тщательно использует заказы и имеет инициализирующие данные, передаваемые параметрами метода. Идеальный кандидат для «Создателя заказа».

3. Контроллер (Controller)

Проблема: Какой первый объект за пределами пользовательского интерфейса получает и координирует «управление» работой системы?

Решение: Назначьте ответственность для объекта, представляющего один из следующих вариантов:

– Представляет общую «систему», «корневой объект», устройство, в котором работает программное обеспечение, или основную подсистему (все это разновидности контроллера фасада);

– Представляет сценарий варианта использования, в котором происходит работа системы (сценарий использования или контроллер сеанса).

Реализация этого принципа зависит от высокого уровня проектирования системы, но в целом всегда необходимо определять объект, который управляет бизнес-транзакцией. На первый взгляд может показаться, что MVC Controller в веб-приложениях / API - отличный пример (даже имя одно и то же), но это не так. Конечно, он получает входные данные, но он не должен координировать системную операцию - он должен делегировать ее отдельной службе или обработчику команд (рис. 9.5):

```
public class CustomerOrdersController : Controller
{
    private readonly IMediator _mediator;

    public CustomerOrdersController(IMediator mediator)
    {
        this._mediator = mediator;
    }

    /// <summary>
    /// Add customer order.
    /// </summary>
    /// <param name="customerId">Customer ID.</param>
    /// <param name="request">Products list.</param>
    [Route("{customerId}/orders")]
    [HttpPost]
    [ProducesResponseType((int)HttpStatusCode.Created)]
    public async Task<IActionResult> AddCustomerOrder(
        [FromRoute]Guid customerId,
        [FromBody]CustomerOrderRequest request)
    {
       await _mediator.Send(new AddCustomerOrderCommand(customerId, request.Products));

       return Created(string.Empty, null);
    }
}

public class AddCustomerOrderCommandHandler : IRequestHandler<AddCustomerOrderCommand>
{
    private readonly ICustomerRepository _customerRepository;
    private readonly IProductRepository _productRepository;
    private readonly IForeignExchange _foreignExchange;

    public AddCustomerOrderCommandHandler(
        ICustomerRepository customerRepository,
        IProductRepository productRepository,
        IForeignExchange foreignExchange)
    {
        this._customerRepository = customerRepository;
        this._productRepository = productRepository;
        this._foreignExchange = foreignExchange;
    }

    public async Task<Unit> Handle(AddCustomerOrderCommand request, CancellationToken cancellationToken)
    {
        // обработка...
    }
}
```

Рисунок 9.5 – Листинг «Контроллер»

4. Слабое зацепление (Low Cohesion)

Проблема: как уменьшить влияние изменений? Как поддержать низкую зависимость и увеличение повторного использования?

Решение: распределите обязанности так, чтобы (ненужная) связь оставалась низкой. Используйте этот принцип для оценки альтернатив.

Зацепление – это мера того, как один элемент связан с другим. Чем выше зацепление, тем больше зависимость одного элемента от другого.

Слабое зацепление означает, что наши объекты более независимы и изолированы. Если что-то изолировано, мы можем изменить его, не беспокоясь о том, что нам нужно что-то изменить или еще что-нибудь сломать. Использование принципов SOLID - отличный способ поддерживать низкий уровень сцепления. Как видно из приведенного выше примера, связь между *CustomerOrdersController* и *AddCustomerOrderCommandHandler* остается низкой - им нужно только согласовать структуру объекта команды. Такое низкое зацепление возможно благодаря схеме косвенного воздействия.

5. Высокая связанность (High Cohesion)

Проблема: как сохранить объекты сфокусированными, понятными, управляемыми и в качестве побочного эффекта поддерживать слабое зацепление?

Решение: возьмите на себя ответственность, чтобы сплоченность оставалась высокой. Используйте данный подход, для оценки альтернатив.

Связанность – это мера того, насколько тесно связаны все обязанности элемента. Другими словами, в какой степени части внутри элемента принадлежат друг другу.

Классы с низкой связанностью имеют несвязанные данные и / или несвязанное поведение. Например, класс *Customer* обладает высокой сплоченностью, потому что теперь он делает только одно - управляет заказами. Если добавить к этому классу управление ответственностью за цены на продукты, сплоченность этого класса значительно снизилась бы, потому что прайс-лист не имеет прямого отношения к самому покупателю (*Customer*).

6. Посредник (Indirection)

Проблема: где возложить ответственность, чтобы избежать прямой связи между двумя или более вещами?

Решение: возложите ответственность на промежуточный объект, чтобы он осуществлял связь между другими компонентами или службами, чтобы они не были напрямую связаны.

Пример паттерна *посредник* вместо прямой связи:

```
1  public class CustomerOrdersController : Controller
2  {
3      private readonly IOrdersService _ordersService;
4
5      public CustomerOrdersController(IOrdersService ordersService)
6      {
7          this._ordersService = ordersService;
8      }
9  }
```

Рисунок 9.10 – Листинг «Посредник»

можно использовать объект-посредник и связываться между объектами:

```
public class CustomerOrdersController : Controller
{
    private readonly IMediator _mediator;

    public CustomerOrdersController(IMediator mediator)
    {
        this._mediator = mediator;
    }

    public async Task<IActionResult> AddCustomerOrder(
        [FromRoute]Guid customerId,
        [FromBody]CustomerOrderRequest request)
    {
       await _mediator.Send(new AddCustomerOrderCommand(customerId, request.Products));

       return Created(string.Empty, null);
    }
}
```

Рисунок 9.11 – Листинг «Посредник»

Посредник поддерживает слабую связь, но снижает читабельность и понимание обо всей системе. Неизвестно, какой класс обрабатывает команду из определения Controller. Это компромисс, чтобы принять во внимание.

7. Полиморфизм (Polymorphism)

Проблема: как обрабатывать альтернативы на основе типа?

Решение: когда связанные альтернативы или поведение различаются по типу (классу), переложите ответственность за поведение (используя операции polymorphi) на типы, для которых поведение изменяется.

Полиморфизм является фундаментальным принципом объектно-ориентированного проектирования. В этом контексте принцип тесно связан с (среди прочего) паттерном стратегии.

Как было показано выше, конструктор класса Customer принимает интерфейс ICustomerUniquenessChecker в качестве параметра:

```
public Customer(string email, string name, ICustomerUniquenessChecker customerUniquenessChecker)
{
    this.Email = email;
    this.Name = name;

    var isUnique = customerUniquenessChecker.IsUnique(this); // doing - initiate and coordinate actions with other objects
    if (!isUnique)
    {
        throw new BusinessRuleValidationException("Customer with this email already exists.");
    }

    this.AddDomainEvent(new CustomerRegisteredEvent(this));
}
```

Рисунок 9.12 – Листинг «Полиморфизм»

Можно предоставить различные реализации этого интерфейса в зависимости от требований. В целом, это очень полезный подход, когда в системе используются разные алгоритмы с одинаковым входом и выходом (с точки зрения структуры).

8. Чистая выдумка (Pure Fabrication)

Проблема: Какой объект должен нести ответственность, если вы не хотите нарушать высокую связанность и слабое зацепление, но решения, предлагаемые другими принципами, не подходят?

Решение: Присвойте очень связанный набор ответственностей искусственному или вспомогательному классу, который не представляет концепцию проблемной области.

Иногда очень трудно понять, где должна быть ответственность. Вот почему в предметно-ориентированном проектировании существует концепция доменного сервиса. Доменные сервисы содержат логику, которая не связана с одним, конкретным объектом.

Например, в системах электронной коммерции необходимо часто конвертировать одну валюту в другую. Иногда трудно сказать, где это поведение должно быть размещено, поэтому лучший вариант - создать новый класс и интерфейс:

```
1   public interface IForeignExchange
2   {
3       List<ConversionRate> GetConversionRates();
4   }
5
6   public class ForeignExchange : IForeignExchange
7   {
8       private readonly ICacheStore _cacheStore;
9
10      public ForeignExchange(ICacheStore cacheStore)
11      {
12          _cacheStore = cacheStore;
13      }
14
15      public List<ConversionRate> GetConversionRates()
16      {
17          var ratesCache = this._cacheStore.Get(new ConversionRatesCacheKey());
18
19          if (ratesCache != null)
20          {
21              return ratesCache.Rates;
22          }
23
24          List<ConversionRate> rates = GetConversionRatesFromExternalApi();
25
26          this._cacheStore.Add(new ConversionRatesCache(rates),
27                               new ConversionRatesCacheKey(),
28                               DateTime.Now.Date.AddDays(1));
29
30          return rates;
31      }
32
33      private static List<ConversionRate> GetConversionRatesFromExternalApi()
34      {
35          // Communication with external API. Here is only mock.
36
37          var conversionRates = new List<ConversionRate>();
38
39          conversionRates.Add(new ConversionRate("USD", "EUR", (decimal)0.88));
40          conversionRates.Add(new ConversionRate("EUR", "USD", (decimal)1.13));
41
42          return conversionRates;
43      }
44  }
```

Рисунок 9.13 – Листинг «Чистая выдумка»

Таким образом, мы поддерживаем как высокую связанность (мы только конвертируем валюты), так и слабую зацеплённость (клиентские классы зависят только от интерфейса IForeignExchange). Кроме того, этот класс можно использовать повторно и легко поддерживать.

9. Устойчивость к изменениям (Protected Variations)

Проблема: как спроектировать объекты, подсистемы и системы таким образом, чтобы изменения или нестабильность этих элементов не оказывали нежелательного влияния на другие элементы?

Решение: определите точки прогнозируемого изменения или нестабильности, распределите обязанности по созданию стабильного интерфейса вокруг них.

Один из самых важных принципов, который косвенно связан с остальными принципами GRASP. В настоящее время одним из наиболее важных показателей программного обеспечения является простота изменений. Как архитекторы и программисты, мы должны быть готовы к постоянно меняющимся требованиям. Это не является обязательным и качественным атрибутом «приятно иметь» - это «обязательный» и обязанность программистов и разработчиков.

Вопросы для самоконтроля

21. Что такое ответственность в контексте ПО?
22. Какую проблему решает паттерн «информационный эксперт»?
23. Какую проблему решает паттерн «создатель»?
24. Какую проблему решает паттерн «контроллер»?
25. Какую проблему решает паттерн «слабое зацепление»?
26. Какую проблему решает паттерн «высокая связанность»?
27. Какую проблему решает паттерн «посредник»?
28. Какую проблему решает паттерн «полиморфизм»?
29. Какую проблему решает паттерн «чистая выдумка»?
30. Какую проблему решает паттерн «устойчивость к изменениям»?

Задание

На примере описанных и приведённых паттернов GRASP примените 2 паттерна, решающие проблемы архитектуры Вашего приложения. В отчёт укажите скриншот листинга и дайте краткое описание классов и их ответственностей.

Лабораторная работа №10 - Проектирование диаграмм пакетов.

Если некоторый пакет X в значительной мере зависит от конкретной группы разработчиков, то его нужно сделать максимально устойчивым (не подлежащим изменению в новых версиях системы), поскольку в противном случае повышается зависимость от этих разработчиков. Именно их придётся привлекать для модернизации пакета в случае необходимости.

Это достаточно очевидно, но иногда данному вопросу не уделяется должного внимания, что приводит к неоправданно высоким трудозатратам.

Данный вопрос относится физическому проектированию (physical design) системы, которое выполняется в рамках модели реализации путём разбиения исходного кода на пакеты.

В процессе построения диаграмм на бумаге или с помощью CASE-средств разработчик может разместить типы данных и пакеты достаточно произвольно. Однако в процессе проектирования физической архитектуры, подразумевающей организацию типов данных в физические пакеты на C++, C# или Java, способ разбиения системы на пакеты определяется взаимным влиянием элементов системы.

Вертикальное и горизонтальное зацепление функциональности пакетов.

Основной интуитивный подход к разбиению системы на модули подразумевает группировку элементов по принципу высокого зацепления – тесной взаимосвязи частей пакета в рамках реализации общих задач, служб, политик и функций. Представьте программный комплекс для обеспечения работы POS-систем именуемый AllyPOS. Система спроектирована на языке программирования, реализующий концепцию ООП. Система декомпозирована на функциональные модули, которые в свою очередь делятся внутри модуля на объекты с собственной областью ответственности (работа с базой данных, работа с графическим интерфейсом, работа с веб-сервером и т.д.). Представим, что все типы данных пакета *Pricing* системы AllyPOS связаны с определением стоимости товаров. Уровни и пакеты системы AllyPOS разбиты на группы по функциональному назначению.

Менее очевидным критерием группировки является высокая степень внутреннего связывания, определяющая принадлежность типов некоторому кластеру. Например, класс *Register* тесно связан с классом *Sale*, который, в свою очередь, связан с объектами *SalesLineItem*.

Степень внутреннего связывания пакетов или его *относительное зацепление* можно выразить в числовом представлении, хотя эта характеристика используется достаточно редко. Для общего сведения приведём формулу его вычисления:

RC = ЧислоВнутреннихСвязей / КоличествоТипов

Здесь *ЧислоВнутреннихСвязей* включает связи между параметрами и атрибутами, отношения наследования, реализацию интерфейса среди типов данных пакета.

Для пакета с 6 типами данных и 12 связями коэффициент связывания RC составляет 2. Для пакета с 6 типами и 3 внутренними связями RC = 0.5. Более высокие значения коэффициента означают более высокую степень зацепления элементов данного пакета.

Данная характеристика не очень информативна для пакетов, содержащих в основном интерфейсы. Она гораздо показательнее для пакетов, включающих реализации классов.

Слишком маленькое значение RC означает выполнение одного из следующих условий:

– Пакет содержит не связанные между собой элементы и плохо нормализован.

– Пакет содержит не связанные между собой элементы, но это не должно беспокоить разработчиков. Это случается с пакетами утилит или разрозненных служб. В этом случае RC не играет роли.

– Пакет содержит один или несколько внутренних кластеров с высоким коэффициентом RC, но в целом степень связывания элементов пакета невысока.

Пакет – это базовая структурная единица разработки и программного продукта. Отдельные классы разрабатываются гораздо реже и совсем редко становятся приложениями.

Допустим, существует большой паке P1, содержащий порядка тридцати классов, некоторое подмножество которого (порядка 10 классов: C1..C10) постоянно модифицируется.

В этом случае стоит разделить пакет P1 на 2 отдельных пакета P1-a и P1-b, включив в P1-b десять неустойчивых классов.

Таким образом пакет делится на устойчивую и неустойчивую части либо на группы классов, работа над которым ведётся одновременно. Отсюда следует вывод: если работа над классами пакета ведётся одновременно, значит пакет сгруппирован правильно.

В идеале пакеты нужно сформировать таким образом, чтобы от классов пакета P1-b зависела работа меньшего числа разработчиков, чем от пакета P1-a. Тогда очередная версия пакета P1-b скажется на дальнейшей работе ограниченного круга разработчиков.

Такая перегруппировка является следствием учёта трудоёмкости создания пакета. На ранних итерациях очень сложно сформировать хорошую структуру пакетов приложения. Она постепенно изменяется на стадии развития, и по окончании этой стадии разработки должна быть сформирована устойчивая структура большинства пакетов.

Один из способов повышения устойчивости пакетов является снижение зависимости от конкретных классов других пакетов. Проблемная ситуация изображена на рис. 10.1.

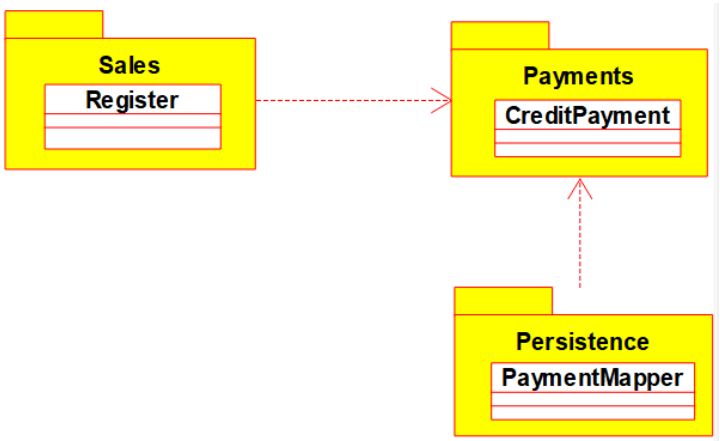


Рисунок 10.1 – Непосредственное связывание с конкретным классов

Предположим, классы *Register* и *PaymentMapper* (класс, осуществляющий преобразование объектов в формат реляционной базы данных и обратно) создают экземпляры объектов *CreditPayment* из пакета *Payments*. Одним из способов повышения устойчивости пакетов *Sales* и *Persistence* является предотвращение создания экземпляров классов, определённых в других пакетах (объектов *CreditPayment* из пакета *Payments*).

Такое связывание можно уменьшить за счёт использования объекта-фабрики, отвечающего за создание экземпляров. Методы этого объекта-фабрики должны возвращать объекты, объявленные в терминах интерфейсов, а не классов (рис. 10.2).

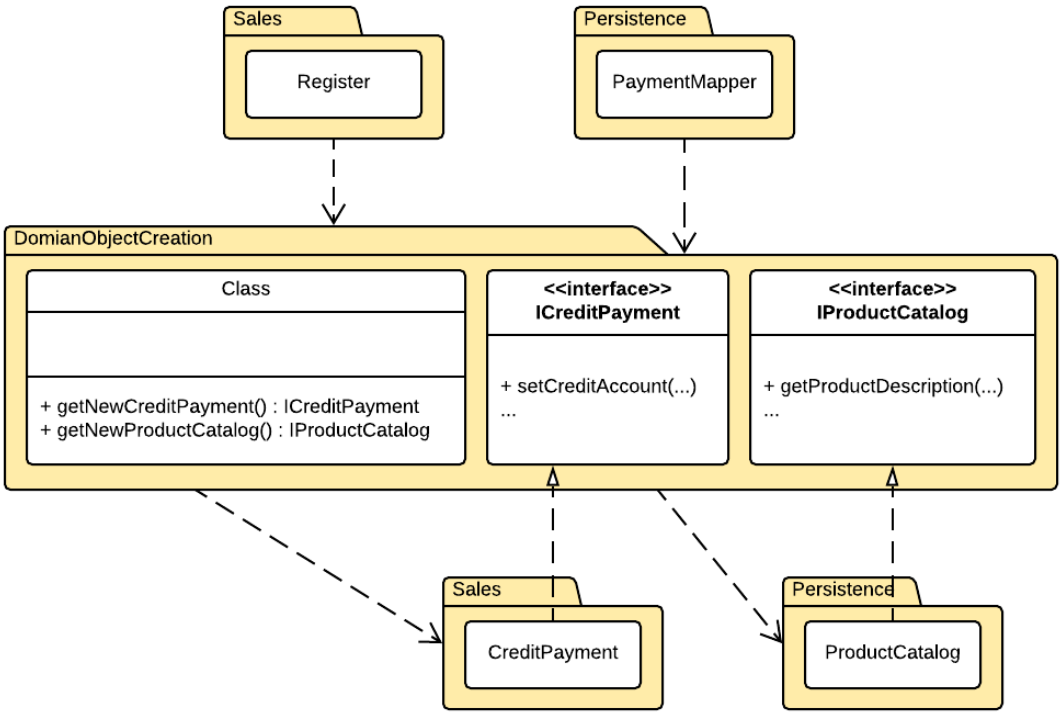


Рисунок 10.2 – Снижение зависимости за счёт объекта-фабрики

Теперь рассмотрим структуру диаграмм пакетов клиентской и серверной логики приложения Sound Room, рис 3 и 4

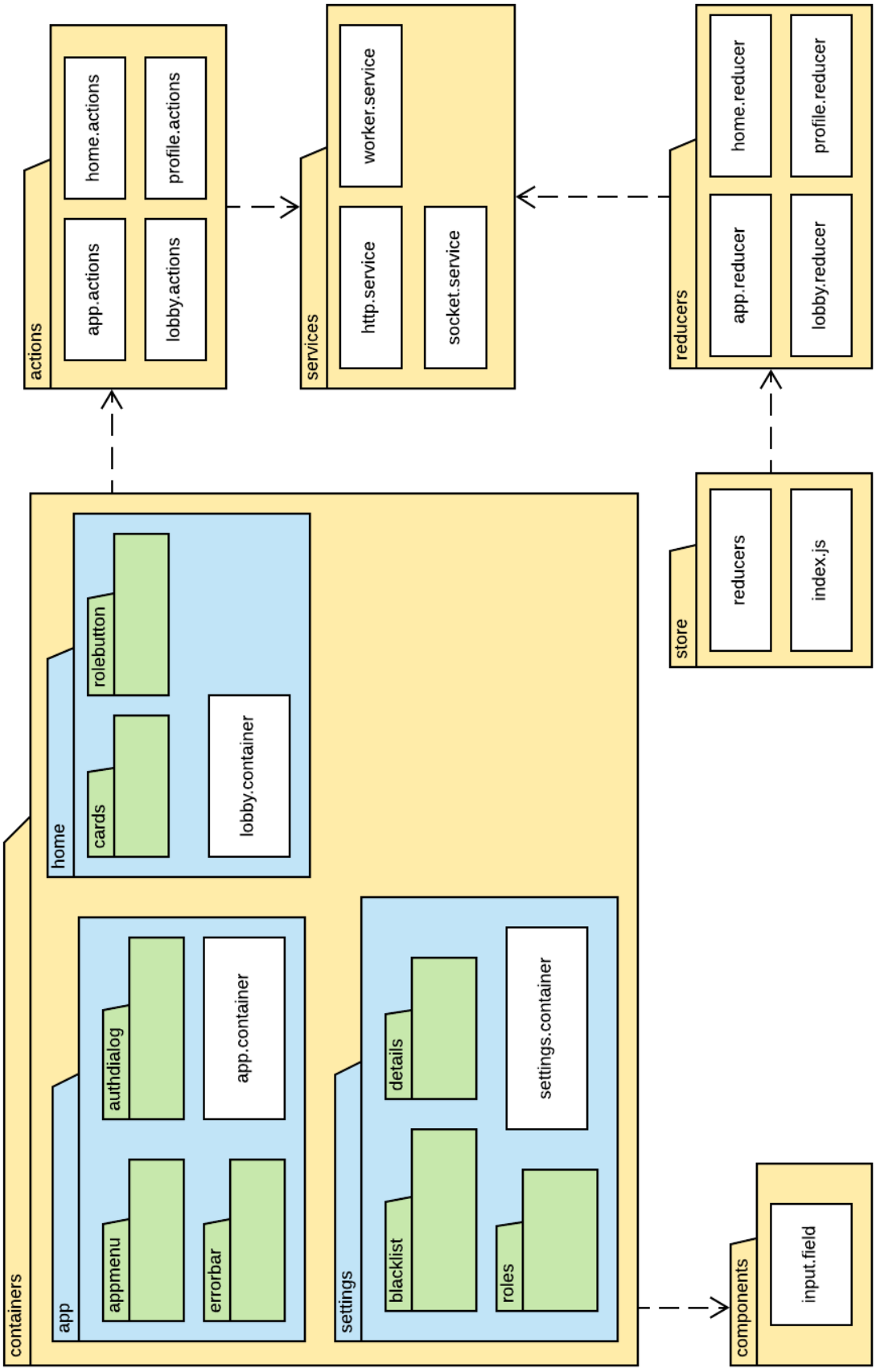


Рисунок 10.3 – Диаграмма пакетов клиентской логики

Как представлено на рисунке 10.3, компоненты клиентской логики разбиты на различные пакеты, каждый из который отвечает за собственную область функционирования. Обратите внимание на пакет *containers*. Данный пакет имеет тройную вложенность, что может быть применимо только для описания графических элементов. При организации пакетов логического взаимодействия, вложенность более двух уровней сильно усложняет архитектуру проекта. Если проект подразумевает большое количество графических элементов, а их точное определение не влияет на общий смысл описания функционирования системы, описания уровней можно сократить до одного или двух, пренебрегая указанием нижних уровней.

containers – в данном пакете хранятся другие пакеты, который в свою очередь имеют под-пакеты, содержащие код описания графических элементов приложения. Все функциональные компоненты данной области содержат по графическому описанию элементов, а также код, содержащий свойства элементов и типы действий, которые может совершать компонент.

actions – пакет, содержащий описание действий, которые могу совершать компоненты клиентской логики.

services – пакет, содержащий логику описания сервисов, с помощью которых клиентская система может отправлять и получать запросы от веб-сервера, сервисы обработки фоновых событий на странице, а также организация программных интерфейсов.

reducers – пакет, содержащий набор объектов, обрабатывающих события компонентов, и передающих их единое хранилище для их управления.

store – пакет, содержащий программный объект, обрабатывающий и управляющий всеми состояниями графических компонентов в приложении.

components – пакет, содержащий реализацию базовых графических компонентов, используемых в других графических компонентах.

Диаграмму из рисунка 3 можно описать следующим образом. Клиентское приложение содержит 6 пакетов реализации программных объектов и 5 связей. Значение RC = 0.8, что свидетельствует о высоком зацеплении.

Пакет *containers* ссылается на пакет *components* для реализации графических компонентов, а также ссылается на пакет *actions*, который содержит логику взаимодействий компонентов. Пакет *actions* имеет единственную ссылку на пакет *services,* который содержит объекты для связи клиентского приложения с внешними сервисами. Пакет *services* является самостоятельным. Пакет *reducers* содержит реализацию состояний графических компонентов, ссылается на пакет *services* для использования сервисов в логике реализации состояния. Пакет *store* является объектом-хранилищем для управления состояниями компонентов.

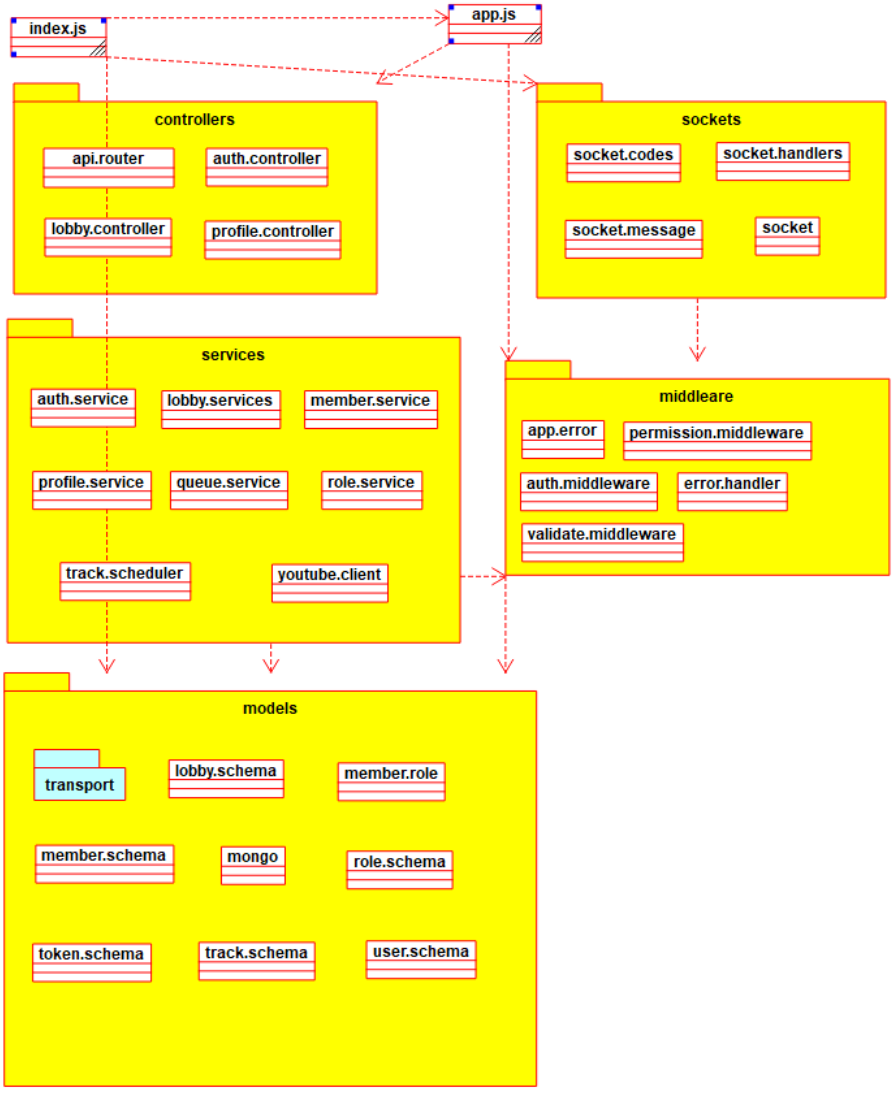


Рисунок 10.4 – Диаграмма пакетов серверной логики

Рассмотрим организацию серверной логики приложения Sound Room. Среда Umbrello имеет существенный недостаток при построении диаграмм, содержащих многослойную структуру. Недостаток связан с неконтролируемым поведением перекрывания одних элементов другими.

Система содержит 5 пакетов, 2 выделенных программных объекта и 9 связей. Значение RC = 1.125, что свидетельствует о высоком зацеплении. Объект *index.js* является главным исполняемым файлом приложения, ссылается на объект *app.js*, который отвечает за организацию внутренней логики функционирования приложения, ссылается на пакет *models*, который ответственен за организацию доступа к сущностям базы данных, ссылается на объект *sockets*, реализующий внешние интерфейсы приложения. Объект *app.js* ссылается на пакет *controllers*, отвечающий за переход пользователя на разные разделы приложения, ссылается на пакет *middleware*, реализующий промежуточное ПО для отслеживания ошибок, аутентификацию и валидацию. Пакет *controllers* является самостоятельным. Пакет *sockets*

ссылается на пакет *middleware* для реализации обработки ошибок при реализации интерфейсов. Пакет *services* реализует основную логику работы приложения (создание комнат, добавление треков, модерация), ссылается на пакет *middleware* для использования объектов обработки ошибок, а также инструментов валидации и авторизации, ссылается на объект *models* для записи данных, возникающих за время функционирования приложения. Пакеты *middleware* и *models* являются самостоятельными.

Следует понимать, что пакет это ни что иное как отдельный каталог внутри проекта. При помощи встроенных инструментов разных языков программирования директории могут оборачиваться в программные пространства имён и иметь глобальную ссылку для доступа из разных мест проекта.

Чтобы начать проектирование диаграммы пакетов в среде Umbrello, необходимо в логическом разделе вашего проекта добавить новую диаграмму классов, дав ей соответствующее название. Следующим шагом добавьте пакет, нажав правой кнопкой мыши по рабочей области и выбрав соответствующий пункт контекстного меню (Package…) (рис. 10.5).

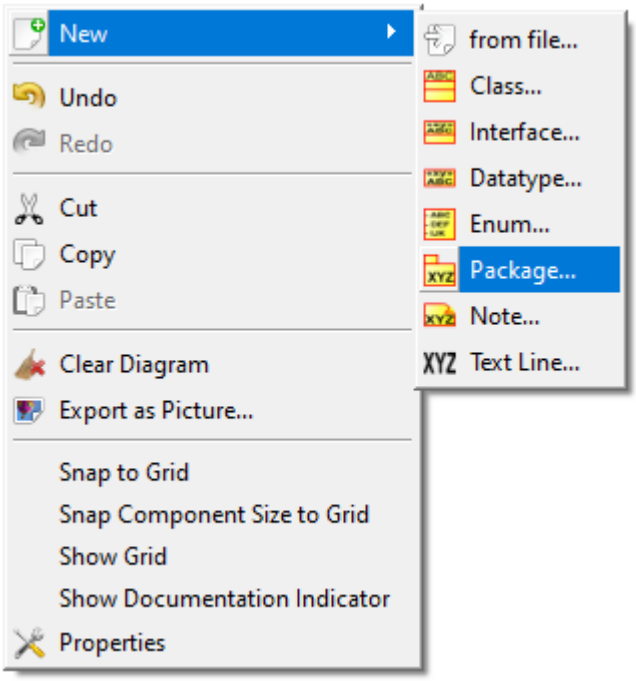


Рисунок 10.5 – Добавление пакета в область диаграммы

Для добавления программных объектов (классов) в область пакет, нажмите на иконку класса в верхней части окна и нажатием левой кнопкой мыши добавьте его в область отображения пакета.

Чтобы провести связь между двумя объектами, нажмите левой кнопкой мыши по иконки зависимости (Dependency) и проведите связь между пакетами нажав сначала на пакет, который использует зависимости

другого пакета, после на пакет, который реализует логики зависимых объектов.

Вопросы для самоконтроля.

1. Для чего необходимо проектировать диаграммы пакетов?
2. Что такое принцип высокой зацепления?
3. Как вычисляется относительное зацепление?
4. Что означает высокое значение коэффициента относительного зацепления?
5. По какой причине может быть малое значение коэффициента относительного зацепления?
6. Что такое пакет?
7. Для чего необходимо дробить пакет 2 отдельные?
8. Всегда ли диаграмма пакетов является полностью сформированной структурой с 1-й итерации её создания?
9. Что такое устойчивость пакета?
10. Какой смысл у объекта-фабрики?

Задание.

Спроектируйте диаграмму пакетов для информационной системы Вышей предметной области. Опишите назначение каждого из пакетов и смысл создания связей между пакетами.

Лабораторная работа №11 - Проектирование диаграммы развёртывания.

Где и как? Именно на эти проектные вопросы даёт ответы диаграмма развёртывания. Ни одна информационная система не работает в вакууме. Она должна где-то находиться и выполняться, принимать определённые данные и куда-нибудь отправлять результат своей деятельности. Для каждого типа ПО существуют свои принципы разворачивания и содержания, но в каждом сценарии присутствуют исполняемый (построенный) код и база данных. Об организации данных компонентов обычно заботятся на финальных этапах проекта. В некоторых случаях, данный вопрос рассматривают в начале разработки проекта, исходя из уже имеющихся ресурсов.

Диаграммы развёртывания отражают соответствие конкретных программных артефактов (например, выполняемых файлов) вычислительным узлам (выполняющим обработку). Они показывают размещение программных элементов в физической архитектуре системы и взаимодействие (обычно сетевое) между физическими элементами. Представленная на рисунке 11.1 диаграмма развёртывания приложения Sound Room позволяет лучше понять физическую архитектуру (или архитектуру развёртывания).

Проектируя диаграмму развёртывания, архитектор работает с наивысшими представлениями программных абстракций. Из данной диаграммы можно узнать основной принцип взаимодействия между информационной системой и пользователем. Данная диаграмма нацелена на системных администраторов и DevOps-инженеров, для которых данное описание является самым быстрым способом внедрения в проект.

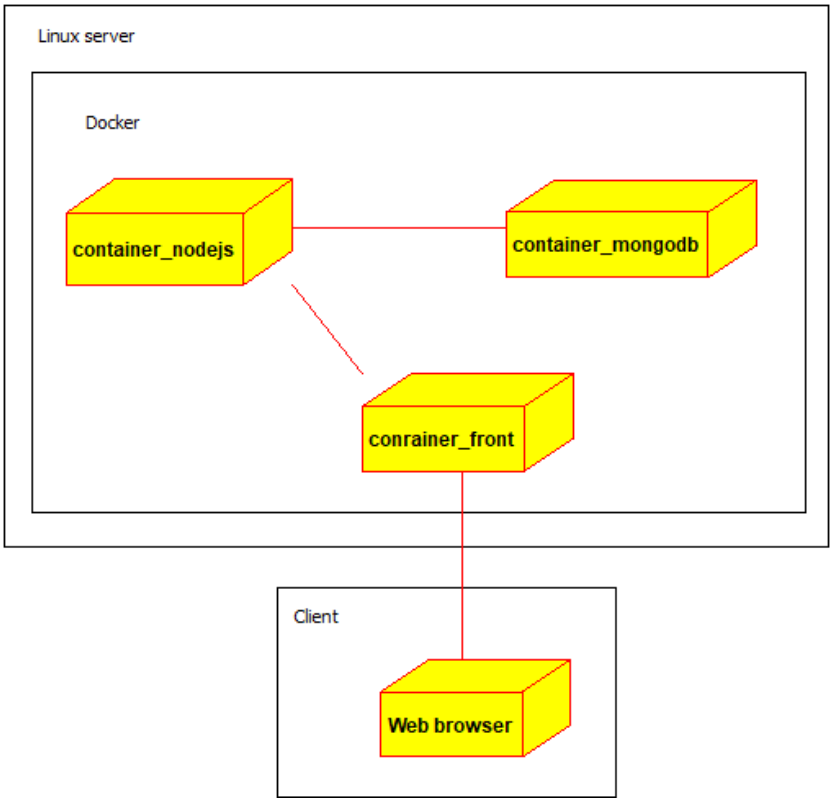


Рисунок 11.1 – Диаграмма развёртывания Sound Room

Основным элементом диаграммы развёртывания является узел (node), относящийся к одному из двух типов.

- *Узел устройства* (device node) (или просто устройство (node)). Это физический (например, цифровой или электронный) вычислительный ресурс с памятью и процессорным элементом, на котором работает программное обеспечение. В роли устройства может выступать обычный компьютер или смартфон.
- *Исполняющий узел окружения* (execution environment node - EEN) – это программный вычислительный ресурс, работающий в рамках другого узла (например, компьютера) и обеспечивающий выполнение других выполняемых программных элементов. К EEN относятся:
  - *операционная система* (operating system) – это программное обеспечение, выполняющее другие прикладные программы;
  - *виртуальная машина* (virtual machine), например Java или .NET, отвечающее за выполнение программ;
  - *система управлениями базами данных* (database engine), например PostgreSQL. Она получает и выполняет SQL-запросы, а также хранимые процедуры;
  - *Web-браузер,* отвечающий за выполнение сценариев JavaScript, аплетов Java и других активных элементов;
  - *механизм управления процессом выполнения задач;*
  - *сервлет-контейнер* или *EJB-контейнер.*

Согласно спецификации UML многие типы узлов отображаются с помощью стереотипов, таких как <<server>>, <<OS>>, <<database>> и <<browser>>.

Из представленного выше описания и рис. 11.1 опишем каждый элемент по соответствующей ему роли.

- *Узел устройства*. Из рис. 11.1 можно выделить 2 устройства, участвующие в жизненном цикле ИС: сервер, на котором выполняется операционная система вместе с развёрнутой программной средой для проекта Sound Room, а также персональный компьютер или смартфон, который обращается к серверу для получения необходимой информации пользователю.
- *Исполняющий узел окружения*. В примере из рис. 1 можно выделить несколько исполняющих узлов в соответствии с иерархией. Объединяющим исполняющим узлом является Linux-сервер, который содержит необходимое ПО для выполнения и управления «контейнерами». Далее по иерархии следует контейнеры, которые являются минималистичными изолированными операционными системами (Linux), содержащие необходимые программные компоненты под свою роль, связанные через программные интерфейсы среды Docker. На конечном уровне вложенности находятся программные компоненты. Среди можно выделить среду Node.js для исполнения серверного кода, СУБД MongoDB для хранения записей приложения, а также клиентский веб-браузер, который через запросы получает и записывает данные.

Главной задачей архитектора информационной системы является оптимизация существующих ресурсов и минимизация расходов на обслуживание системы. К расходам на обслуживание можно отнести:

- оплата труда системному администратору;
- оплата электроэнергии;
- оплата ресурсов выделенного сервера;
- замена вышедшего из строя оборудования;
- докупка оборудования для масштабирования.

На сегодняшний день существует несколько способов размещения вашей системы для доступа пользователям из вне.

Собственный сервер. В данном сценарии всё необходимое оборудование и программное обеспечение находится на территории вашей компании. Все затраты на обслуживание серверов и работоспособности вашей системы являются вашими обязанностями.

Выделенный сервер. В этом случае вы получаете выделенное виртуальное пространство на сервере хостинг-компании. Вашей обязанностью является оплата использования этого пространства в соответствии с тарифами компании. Все обязанности по организации работ

для обеспечения круглосуточной работы вашего виртуального пространства – обязанности компании.

Облачные технологии. С недавнего времени является крайне популярным способом для размещения всей логики приложения через интерфейс облачных технологий на сервера крупных компаний. К такому типу технологии относятся такие продукты как:

- Amazon Web Services;
- Azure;
- Google Cloud Platform;
- Heroku;
- IBM Cloud.

Принцип схож с сценарием выделенного сервера, но ключевым отличием является то, что у Вас нет необходимости работать на прямик с операционной системы для настойки среды под Ваши нужды. Вместо этого вы используйте программные интерфейсы соответствующих сервисов, через которые вы отправляете Ваш код, который Вы будете разворачивать на соответствующих программных оболочках. В данном случае, компания по предоставлению облачного сервиса обязывается взять на себя ответственность по оптимизации нагрузки и стабильной работоспособности Ваших сервисов. Вашей задачей является своевременная оплата выбранного плана.

Каждый из способов имеет плюсы и недостатки. Выбор способа зависит от требований к реализации системы. Ниже приведён пример (рис. 2) диаграммы размещения при сценарии использования сервиса одной из облачных компаний.

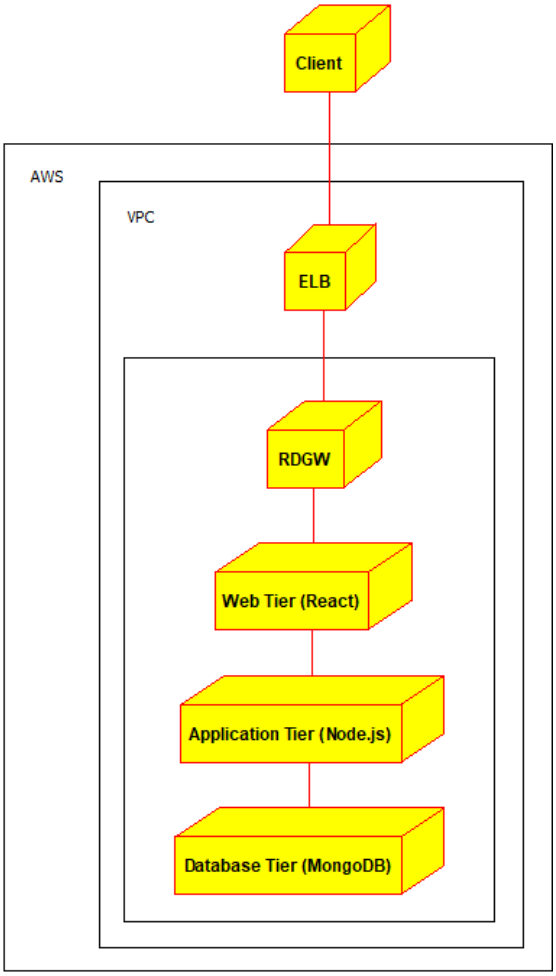


Рисунок 11.2 – Диаграмма размещения системы на облачном сервисе

Диаграмма размещения используется не только для демонстрации способа размещения программных компонентов, но также для документирования внутренней инфраструктуры компании. Например, можно указать минимально необходимую среду для разворачивания разработчику (рис. 11.3).

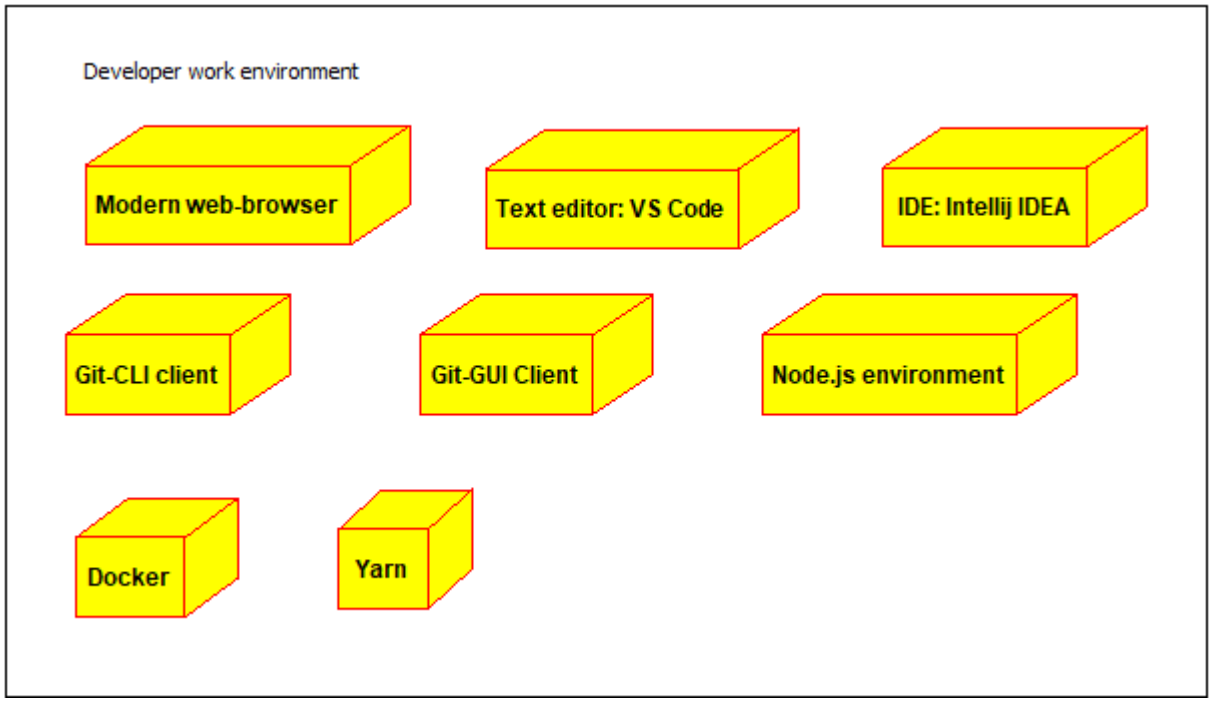


Рисунок 11.3 – Разворачивание компонентов для разработчиков

Диаграмма размещения – хороший и простой инструмент для документирования как отдельного проекта, так и внутренней инфраструктуры.

Вопросы для самоконтроля

1. Что отображает диаграмма размещения?
2. Что такое узел устройства?
3. Что такое исполняющий узел окружения?
4. Для кого нацелена диаграмма развёртывания?
5. Чем является операционная система в диаграмме размещения?
6. В каких целях можно использовать диаграмму размещения?
7. Какие существуют способы размещения информационной системы?
8. Чем отличается выделенный сервер от облачного сервиса?
9. Чем является смартфон в диаграмме размещения?
10. Что такое Docker?

Задание

Спроектируйте диаграмму размещения для выбранного Вами проекта. Реализуйте 2 версии размещения вашего проекта:

1. На выделенном сервере.
2. На облачном сервере.

Лабораторная работа №12 - Проектирование ER-диаграммы.

Модель – искусственный объект, представляющий собой отображение (образ) системы и её компонентов**.**

Модель данных **(Data Model)** – это графическое или текстовое представление анализа, который выявляет данные, необходимые организации с целью достижения ее миссии, функций, целей, стратегий, для управления и оценки деятельности организации. **Модель данных** выявляет ***сущности,*** **домены (атрибуты)** и **связи** с другими данными, а также предоставляет концептуальное представление данных и связи между данными.

Цель создания модели данных состоит в обеспечении разработчика ИС концептуальной схемой базы данных в форме одной модели или нескольких локальных моделей, которые относительно легко могут быть интегрированы в любую базу данных.

При создании моделей данных используется метод семантического моделирования. Семантическое моделирование основывается на значении структурных компонентов или характеристик данных, что способствует правильности их интерпретации (понимания, разъяснения). В качестве инструмента семантического моделирования используются различные варианты диаграмм сущность-связь (ER — Entity-Relationship) — ERD.

Существуют различные варианты отображения ERD**,** но все варианты **диаграмм сущность-связь** исходят из одной идеи — рисунок всегда нагляднее текстового описания. **ER-диаграммы** используют графическое изображение сущностей предметной области, их свойств (атрибутов), и взаимосвязей между сущностями.

Сущность (таблица, отношение) — это представление набора реальных или абстрактных объектов (людей, вещей, мест, событий, идей, комбинаций и т. д.), которые можно выделить в одну группу, потому что они имеют одинаковые характеристики и могут принимать участие в похожих связях. Каждая сущность должна иметь наименование, выраженное существительным в единственном числе. Каждая сущность в модели изображается в виде прямоугольника с наименованием.

Можно сказать, что **сущности** представляют собой множество реальных или абстрактных вещей (людей, объектов, событий, идей и т. д.), которые имеют общие **атрибуты** или характеристики.

Экземпляр сущности (запись, кортеж) – это конкретный представитель данной сущности.

Атрибут сущности (поле, домен) — это именованная характеристика, являющаяся некоторым свойством сущности.

Связь — это некоторая ассоциация между двумя сущностями. Одна сущность может быть связана с другой сущностью или сама с собою. Связи позволяют по одной сущности находить другие сущности, связанные с ней.

Связь типа *один-к-одному* означает, что **один экземпляр первой сущности** связан с одним экземпляром второй сущности. Связь один-к-одному чаще всего свидетельствует о том, что на самом деле мы имеем всего одну сущность, неправильно разделенную на две.

Связь типа *один-ко-многим* означает, что один экземпляр первой сущности связан с несколькими экземплярами второй сущности. Это наиболее часто используемый тип связи. Сущность со стороны «один» называется родительской, со стороны «много» — дочерней.

Для обозначения связей на диаграмме используют несколько обозначений типов связи.

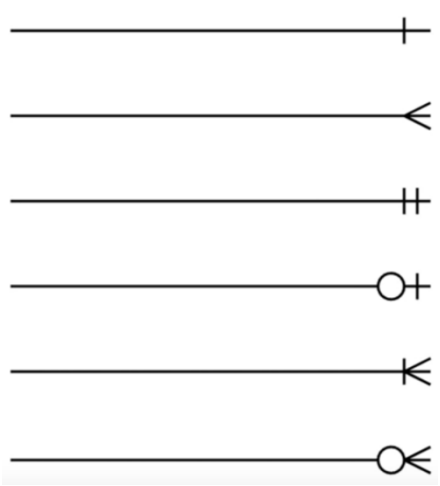


Рисунок 12.1 – Типы связей между таблицами

Рассмотрим подробнее ER-диаграмму приложения Sound Room (рис. 12.2).

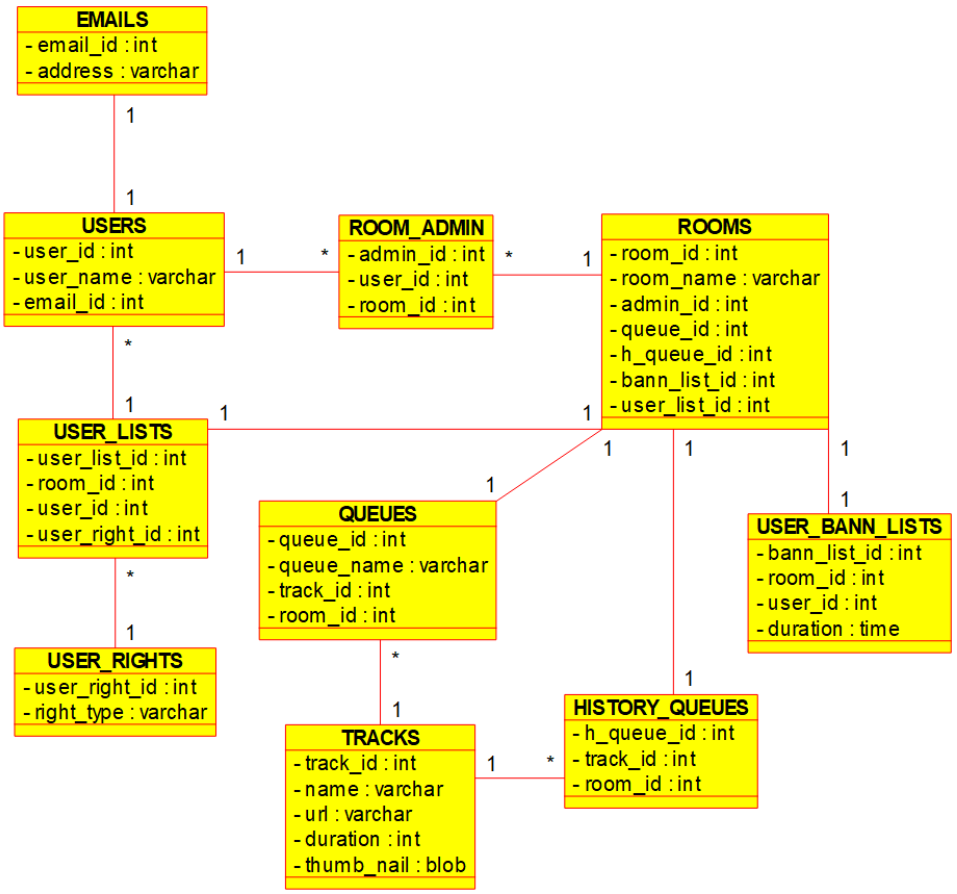


Рисунок 12.2 – ER-диаграмма реляционной БД Sound Room

Из представленных сущностей основными в данной модели данных являются USERS и ROOMS. USERS содержит данные о пользователях системы. ROOMS содержит данные о комнатах (лобби). Данные сущностей разделяются на подтаблицы для оптимизации хранения данных и безопасности.

Таблица 1 – Структура данных БД Sound Room

Название таблицы	**Описание**
EMAILS	Содержит данные об электронных адресах пользователей. Хранятся в отдельной таблице для предоставления базовой безопасности.
USERS	Содержит данные о пользователе приложения Sound Room.
ROOM_ADMIN	Содержит записи пользователей с правами администратора комнат.
ROOMS	Содержит данные об определённой комнате (лобби), ссылается на таблицу с данными о пользователях, присутствующих в комнате, на таблицу администраторов лобби, на таблицу с «чёрным списком» пользователей, на таблицу с информацией об актуальных треках в очереди, на таблицу с историей добавленных треков за всё время.
USER_LISTS	Содержит данные о пользователях для каждой комнаты, ссылается на таблицу с правами пользователя.
USER_RIGHTS	Содержит данные о правах пользователей в комнате (только

	прослушивание, добавление новых треков в очередь или возможность добавлять и удалять треки).
USER_BANN_LIST	Содержит данные о пользователях, добавленных в «чёрный список» определённой комнаты.
QUEUES	Содержит данные о списке треков на очередь для воспроизведения в лобби.
HISTORY_QUEUES	Содержит данные истории воспроизведения треков в лобби.
TRACKS	Содержит данные о записи трека, добавленного в плейлист комнаты.

Прежде чем приступить проектированию, обратимся к одному из соглашений именования объектов в реляционных базах данных.

Названия. Задаётся при создании объекта (таблицы). Чтобы отделять названия таблиц от иных объектов реляционной БД, все буквы названия должны быть в верхнем регистре, а слова должны быть разделены нижним прочерком. Пример: USER_LISTS. При проектировании БД с однотипными таблицами, но относящихся разным доменным областям, в качестве знака соответствия указывают префикс доменной области в названии таблиц. Например, в базе данных содержатся 2 таблицы: с пользователями сервиса и системные пользователи, их названия будут USERS и SYS_USERS соответственно.

Атрибуты (столбцы). Применяя к диаграммам классов, указываются вместо обычных атрибутов вместе с типом, который подразумевается в соответствующем параметре. Атрибуты именуются полностью в нижнем регистре с разделением слов в виде нижнего прочерка. Поэтому имена атрибутов идентификатора, имени пользователя, а также типов доступа таблицы USERS будут следующими:

— user_id;
— user_name;
— user_access.

Стоит также отметить, что в имена таблиц и их атрибутов должны использоваться только существительные, передающие смысл сущности. Слов должно быть не более двух.

Рассмотрим подробнее связи в диаграмме (рис. 2). Более современные системы UML проектирования (Lucidchart, StarUML, Visio, …) поддерживают способ определения связей между таблицами по обозначениям указанные в рис. 12.2, но данный способ на данный момент невозможен в Umbrello. Указание типа связи указывается в свойстве связи между таблицами. Данный способ менее эстетичен, но достаточен для точного определения типа связи.

Таблица 2 – Связи таблиц БД

Связь	**Тип**	**Пояснение**
EMAILS → USERS	1 - 1	Реализовано в виду более безопасного

		хранения данных, а также для возможности использования данного объекта данных в иных процессах бизнес-логики.
USERS → ROOM_ADMIN	1 - *	1 пользователь приложения может иметь права администратора в разных лобби.
ROOMS → ROOM_ADMIN	1 - *	У 1 комнаты может быть много администраторов
USER_LISTS → USERS	1 - *	1 пользователь может быть в разных списках пользователей лобби.
USER_LISTS → ROOMS	1 - 1	1 комната (лобби) может иметь только 1 список пользователей.
USER_RIGHTS → USER_LISTS	1 - *	Пользователи сервиса могут иметь разные типы прав в разных лобби.
ROOMS → QUEUES	1 - 1	В 1 комнате (лобби) может быть только 1 очередь треков.
ROOMS → HISTORY_QUEUES	1 - 1	К 1 комнате (лобби) может относиться только 1 список истории воспроизведения.
ROOMS → USER_BANN_LISTS	1 - 1	К 1 комнате (лобби) может относиться только 1 список заблокированных пользователей.
TRACKS → QUEUES	1 - *	1 и тот же трек может быть в разных очередях на воспроизведение.
TRACKS → HISTORY_QUEUES	1 - *	1 тот и же трек может быть одновременно в разных списках истории воспроизведения.

Обратите внимание, что объект из таблицы TRACKS является целостным и персистентным. За добавление его в реляционную базу данных отвечает отдельный сервис, который индексирует каждый новый добавленный трек из сервисов YouTube, SoundCloud и подобным.

Теперь рассмотрим способ построения ER-диаграммы в среде Umbrello. Первым шагом необходим добавить новую диаграмму в разделе Entity Relationship Model (рис. 12.3).

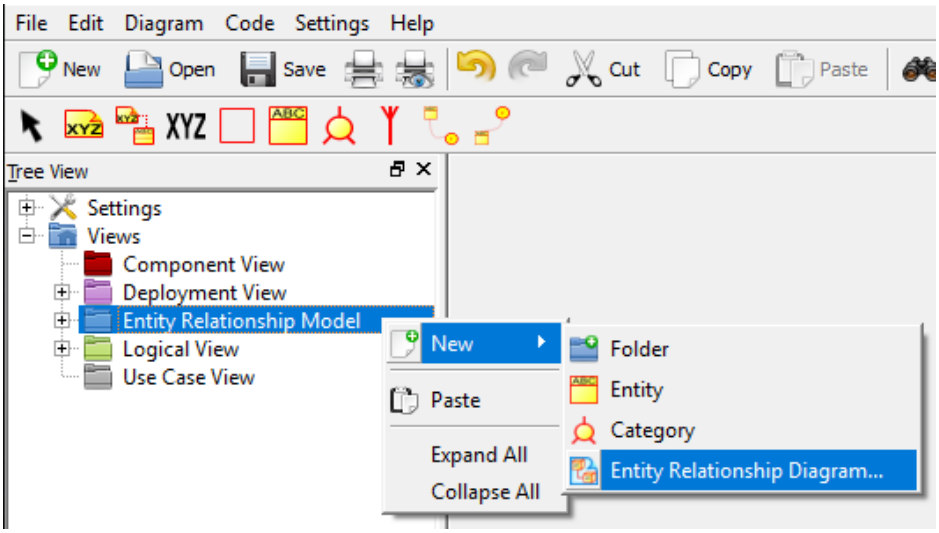


Рисунок 12.3 – Добавление ER- диаграммы

Следующим шагом необходимо нажать правой кнопкой мыши в области проектирования диаграммы и добавить новую сущность и дать ей название (рис. 12.4).

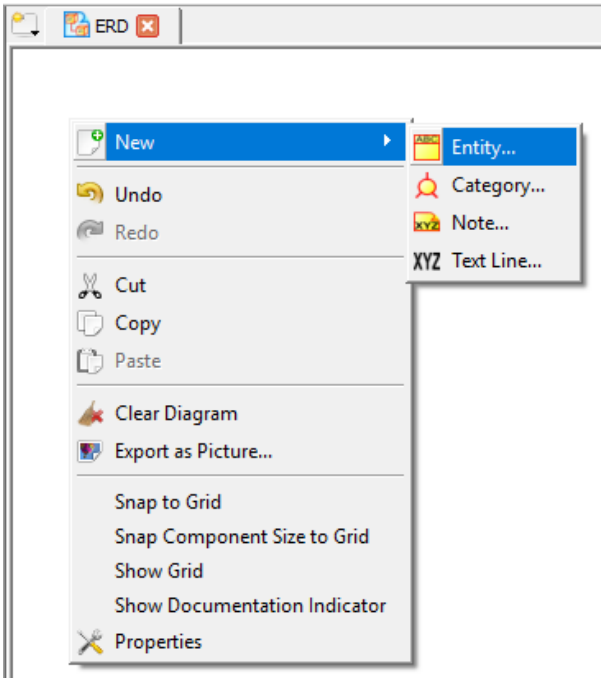


Рисунок 12.4 – Добавление новой сущности

У новой добавленной сущности необходим определить её атрибуты. Производится это нажатием правой кнопкой мыши по сущности и выбором пункта добавления нового атрибута (Entity Atribute…) из контекстного меню (рис. 12.5).

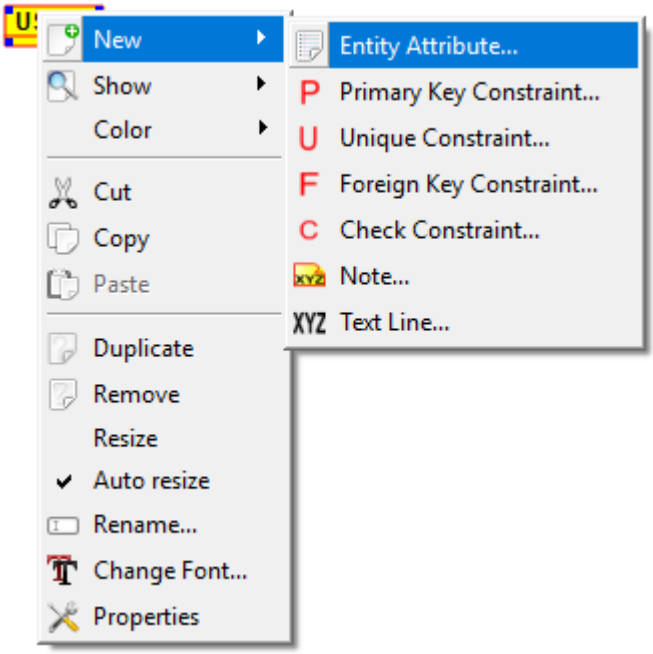


Рисунок 12.5 – Добавление нового атрибута у сущности

После данного действия откроется меню, где необходим определить имя атрибута (Name) и его тип (Type), а также при необходимости задать дополнительные параметры (рис. 12.6)

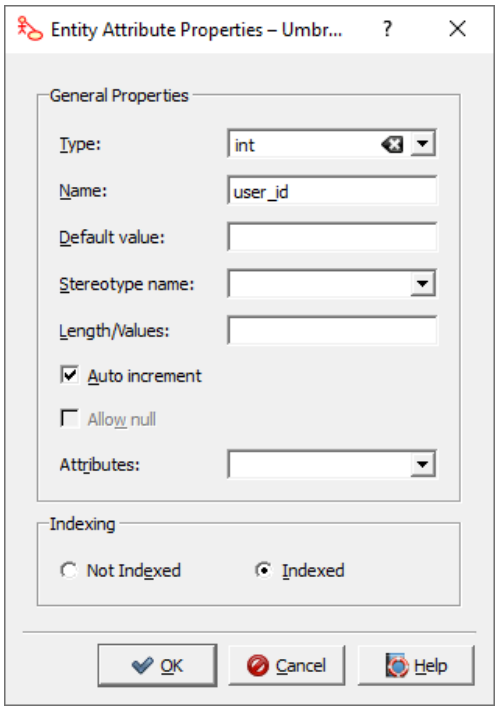


Рисунок 12.6 – Настройка атрибутов сущности

Для того чтобы у атрибута отображался его тип, в меню настройки (ПКМ по сущности → Properties). В открывшемся меню перейти на вкладку Display и поставить галочку рядом со свойством Attribute Signatures (рис. 12.7).

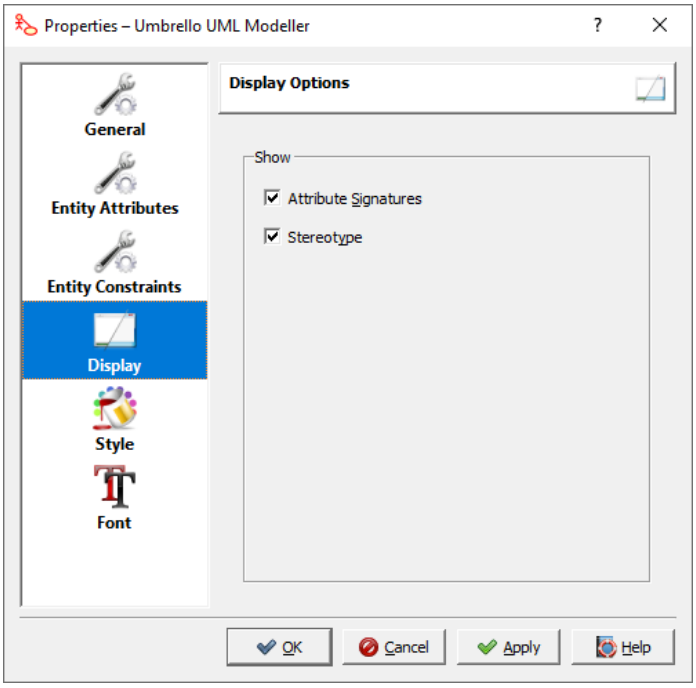


Рисунок 12.7 – Настройка отображения типа атрибутов

После данного действия сущность приобретёт следующий вид (рис. 12.8). Данную процедуру необходим провести для каждой сущности отдельно.

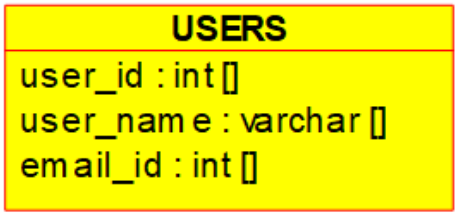


Рисунок 12.8 – Отображение сущности с типами

После определения всех атрибутов необходим определить первичный ключ сущности. Для это из контекстного меню по нажатию правой кнопки мыши необходим выбрать соответствующий пункт (Primary Key Constraint…) (рис. 12.9).

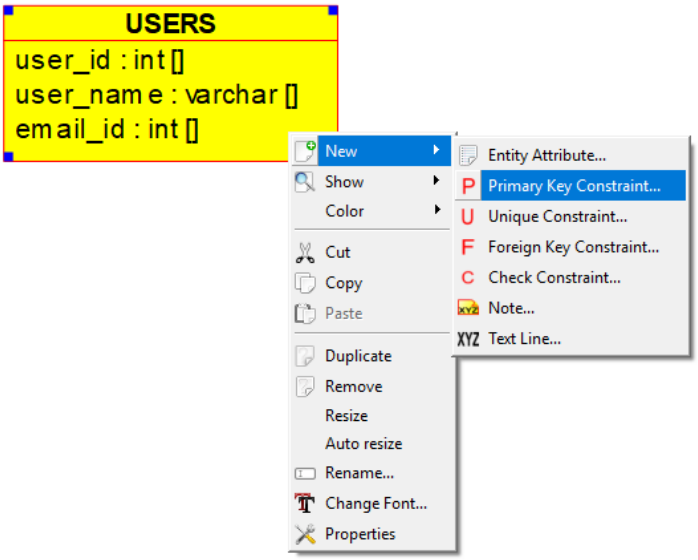


Рисунок 12.9 – Определение первичного ключа сущности

В открывшемся окне (рис. 12.10) необходим определить названия правила определения первичного ключа, а также выбрать атрибут, который будет им являться. После чего нажать кнопку Add.

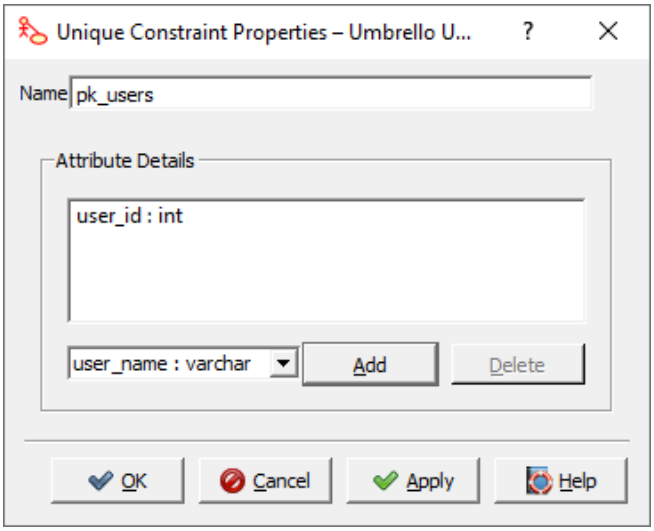


Рисунок 12.11 – Настройка первичного ключа

После определения первичного ключа, его наличие отображается в древовидном меню в верхней левой части экрана (рис. 12.12).

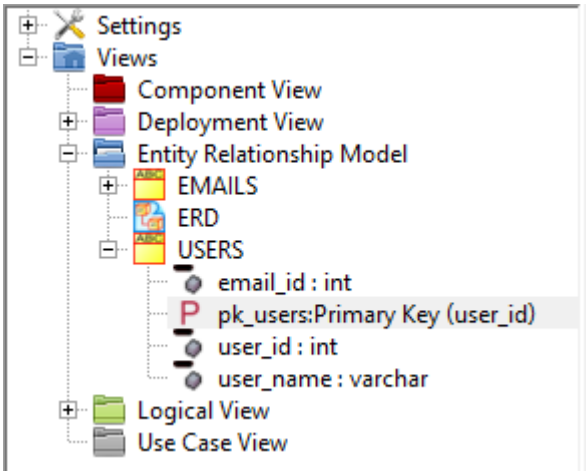


Рисунок 12.12 – Отображение первичного ключа в проекте

Для определения связи и внешнего ключа у сущностей, необходимо у сущности, которая содержит атрибут внешнего ключа, нажатием правой кнопкой мыши выбрать пункт добавления внешнего ключа (Foreign Key Constraint…) (рис. 12.13).

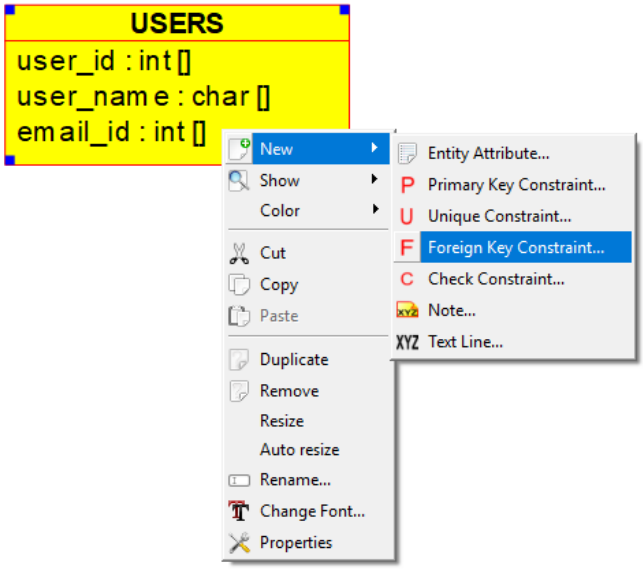


Рисунок 12.13 – Определение внешнего ключа у сущности

В открывшемся окне необходим дать название новой связи, а также таблицу, к первичному ключу которой производится обращение (рис. 12.14). Далее в разделе Columns выбрать название атрибута, к которому производится обращение (рис. 12.15).

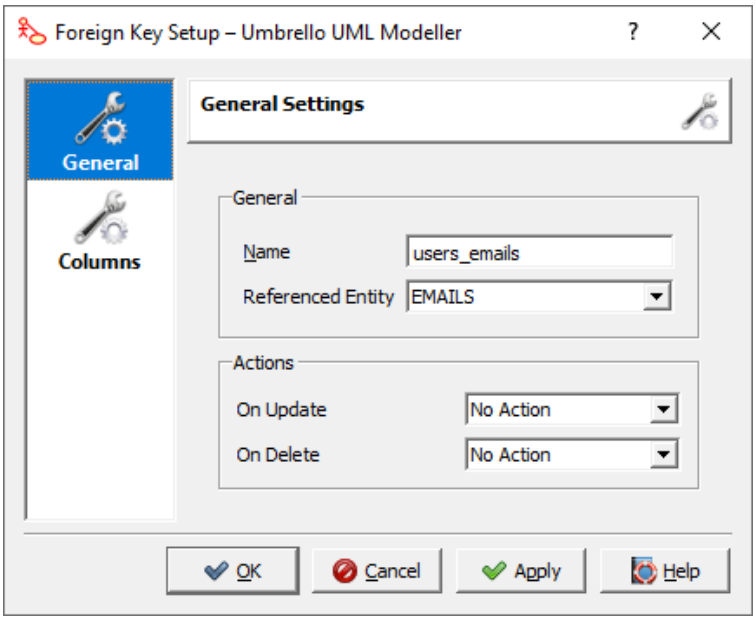


Рисунок 12.14 – Выбор таблицы для определения внешнего ключа

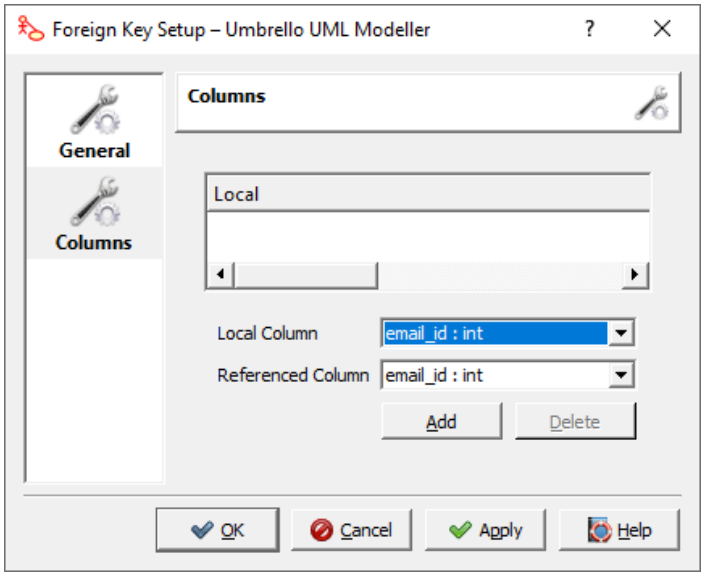


Рисунок 12.15 – Выбор атрибута для определения внешнего ключа

После данной процедуры образуется связь между таблицами (рис. 12.16).

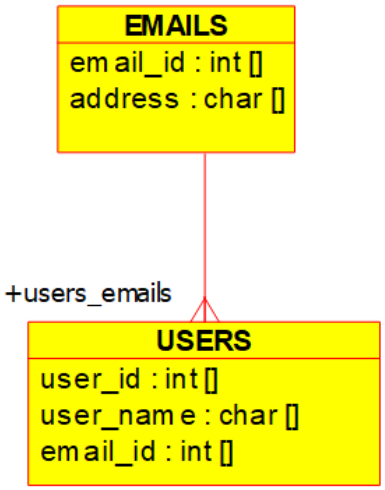


Рисунок 12.16 – Связь между таблицами

Среда Umbrello по умолчанию имеет только отображение связи типа один-ко-многим, что является большим недостатком, когда необходим отобразить связи типа один-ко-одному или многие-ко-многим. Для решения этой проблемы с свойствах связи необходим явно указать к какому типу относится данная связь. Для этого необходим нажатием правой кнопки мыши по связи выбрать свойства (Properties) и перейти в раздел ролей (Roles) (рис. 12.17).

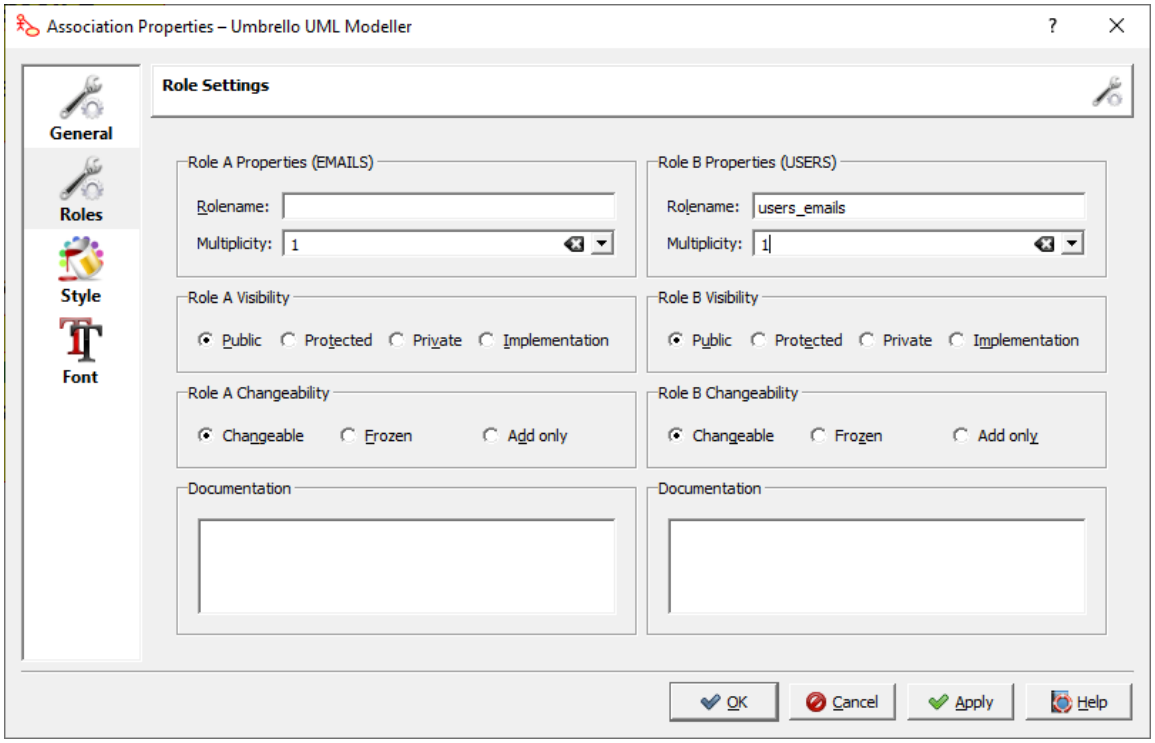


Рисунок 12.17 – Явное определение типа связей

Тип связи указывается в поле Multiplicity соответственно сущностям. После определение связь будет иметь следующий вид (рис. 12.18).

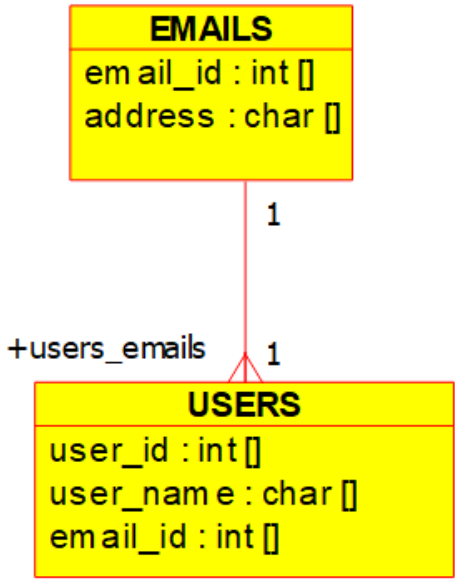


Рисунок 12.18 – Вид связи после определения явного определения типа.

Вопросы для самоконтроля.

1. Дайте определение модели.
2. Дайте определение модели данных.
3. Дайте расшифровку акрониму ERD.
4. Дайте определение сущности.
5. Дайте определение атрибуту сущности.
6. Дайте определение связи.
7. Назовите существующие типы связей.
8. Согласно правилам, каким образом должна именоваться сущность?
9. Согласно правилам, каким образом должны именоваться атрибуты сущности?
10. Какова цель создания модели данных?

Задание.

Постройте ER-диаграмму в соответствии с предметной областью вашего проекта. Произведите моделирование в среде Umbrello, а также опишите назначение сущностей и связей в отчёте к лабораторной работе. Описание производить по примеру Таблицы 1 и Таблицы 2.

Список рекомендованной литературы

1. Григорьев, М. В. Проектирование информационных систем : учебное пособие для вузов / М. В. Григорьев, И. И. Григорьева. — Москва : Издательство Юрайт, 2019 ; Тюмень : Тюменский государственный университет. — 318 с. — (Высшее образование). — ISBN 978-5-534-01305-4 (Издательство Юрайт). — ISBN 978-5-400-01099-6 (Тюменский государственный университет). — Текст : электронный // ЭБС Юрайт [сайт]. — URL: https://biblio-online.ru/bcode/434436 (дата обращения: 16.09.2019).

2. Рыбальченко, М. В. Архитектура информационных систем : учебное пособие для вузов / М. В. Рыбальченко. — Москва : Издательство Юрайт, 2019. — 91 с. — (Университеты России). — ISBN 978-5-534-01159-3. — Текст : электронный // ЭБС Юрайт [сайт]. — URL: https://biblio-online.ru/bcode/437686 (дата обращения: 16.09.2019).

3. Лаврищева, Е. М. Программная инженерия и технологии программирования сложных систем : учебник для вузов / Е. М. Лаврищева. — 2-е изд., испр. и доп. — Москва : Издательство Юрайт, 2019. — 432 с. — (Бакалавр. Академический курс). — ISBN 978-5-534-07604-2. — Текст : электронный // ЭБС Юрайт [сайт]. — URL: https://biblio-online.ru/bcode/436514 (дата обращения: 16.09.2019).

4. Коваленко В. В. Проектирование информационных систем: учеб. пособие. - М.: Форум: НИЦ ИНФРА-М, 2014. - 320 с. Режим доступа: http://znanium.com/bookread2.php?book=473097.

5. Голицына О. Л. Информационные системы: учеб. пособие. - М.: Форум: ИНФРА-М, 2007. - 496 с. Режим доступа: http://znanium.com/bookread2.php?book=129184.

6. Бахтизин, В.В. Стандартизация и сертификация программного обеспечения : учеб. пособие / В. В. Бахтизин, Л. А. Глухова. – Минск : ГУИР,2006.

7. Буч Г., Рамбо Д., Якобсон А. Язык UML. Руководство пользователя. Второе издание. — ДМК, 2008, 496 с.

8. Новиков Ф.А, Иванов Д.Ю. Моделирование на UML. Теория, практика, видеокурс. — СПб, Профессиональная литература, Наука и Техника, 2010, 640 с.

9. Буч Г., Якобсон А., Рамбо Д. UML. 2-е издание Классика CS. — Спб., Питер, 2009, 736 с.

10. Буч Г., Якобсон А., Рамбо Д. Унифицированный процесс разработки программного обеспечения. Питер, 2012, 496 с.

11. Крэг Л. Применение UML 2.0 и шаблонов проектирования, 3-е издание. Вильямс, 2007, 736 с.

12. Марка Д.А., МакГоуэн К. Методология структурного анализа и проектирования. М., "МетаТехнология", 2006.

13. Рамбо Д., Блаха М. UML 2.0. Объектно-ориентированное моделирование и разработка. Питер, 2007, 540 с.

14. Фаулер М. UML. Основы. 3-е издание. — Символ-Плюс, 2005, 192 с.

15. Путилин А.Б., Юрагов Е.А. Компонентное моделирование и программирование на языке UML: Практическое руководство по проектированию информационно-измерительных систем / А.Б. Путилин, Е.А. Юрагов. – М.: НТ Пресс, 2010. – 664с.: ил. – (Проектирование и моделирование).

16. Майер Г. Надежность программного обеспечения. – М: Мир, 2010. – 280 с.

ПРОЕКТИРОВАНИЕ ПРОГРАММНЫХ КОМПЛЕКСОВ

Методические указания к лабораторным работам

Внутри кафедральное издание

Составитель Янаева Марина Викторовна

Компьютерная верстка М.В. Янаева