МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РЕСПУБЛИКИ КАЗАХСТАН
Некоммерческое акционерное общество
«АЛМАТИНСКИЙ УНИВЕРСИТЕТ ЭНЕРГЕТИКИ И СВЯЗИ»
Кафедра IT-инжиниринг

ДОПУЩЕН К ЗАЩИТЕ
Заведующий кафедрой
PhD, доцент
_______________ Т.С. Картбаев
«____» _______________ 2019 г.

ДИПЛОМНЫЙ ПРОЕКТ

На тему: Разработка интегрированной среды администрирования баз данных

Специальность: 5В070400 – «Вычислительная техника и программное обеспечение»

Выполнил: Баяхметов Р.А. Группа ВТ-15-2
Научный руководитель: ст. преп. Тулегенова Б. А.

Консультанты:
по экономической части: к.э.н., профессор _______________ Ж.Г. Аренбаева
«13» мая 2019 г.
по безопасности
жизнедеятельности: д.т.н., ст. преп. _______________ Ш.Ш. Бекбасаров
«14» мая 2019 г.
по применению
вычислительной техники: ст. преп. _______________ М.Н. Майкотов
«6» мая 2019 г.

Нормоконтролер: ст. преп. _______________ А. А. Айтказина.
«14» мая 2019 г.

Рецензент: д.т.н., профессор _______________ Р. К. Ускенбаева.
«____» _______________ 2019 г.

Алматы 2019

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РЕСПУБЛИКИ КАЗАХСТАН
Некоммерческое акционерное общество
«АЛМАТИНСКИЙ УНИВЕРСИТЕТ ЭНЕРГЕТИКИ И СВЯЗИ»

Институт систем управления и информационных технологий

Кафедра IT-инжиниринг

Специальность 5В070400 – «Вычислительная техника и программное обеспечение»

ЗАДАНИЕ
на выполнение дипломного проекта

Студенту Баяхметову Раису Армановичу

Тема проекта: Разработка интегрированной среды администрирования баз данных

Утверждена приказом по университету № 124 от «26» октября 2018 г.

Срок сдачи законченного проекта «24» мая 2019 г.

Исходные данные к проекту (требуемые параметры результатов исследования (проектирования) и исходные данные объекта): Руководство системы менеджмента качества на предприятии; международные стандарты ИСО-9001, данные преддипломной практики.

Перечень вопросов, подлежащих разработке в дипломном проекте, или краткое содержание дипломного проекта:
- аналитическая часть;
- проектная часть;
- экспериментальная часть;
- экономическая часть;
- безопасность жизнедеятельности;
- приложение А. Техническое задание;
- приложение Б. Листинг программы.

Перечень графического материала (с точным указанием обязательных чертежей): представлены 8 таблиц, 41 иллюстрации.

Основная рекомендуемая литература:
1 Блох Джошуа, «Java. Эффективное программирование», Вильямс, 2019 г. - 464с.

2 Редмонд Эрик., Уилсон Джим Р., «Семь баз данных за семь недель. Введение в современные базы данных и идеологию NoSQL», ДМК Пресс, 2018 г. - 384с;

3 С. Д. Кузнецов, «Введение в модель данных SQL», М.: Национальный Открытый Университет «ИНТУИТ», 2016.

4 Даниэль Барретт, «Linux Pocket Guide, 3rd Edition», O'Reilly Media, 2016г. - 272с.

5 Мартин Фаулер, «UML Distilled», Addison-Wesley Professional, 2018 г. - 208с.

Консультации по проекту с указанием относящихся к ним разделов проекта

Раздел	Консультант	Сроки	Подпись
Экономическая часть	Аренбаева Ж.Г.	04.03.2019 – 13.05.2019	
Безопасность жизнедеятельности	Бекбасаров Ш.Ш.	04.03.2019 13.05.2019	
Программное обеспечение	Майкотов М.Н.	04.03.2019 – 15.03.2019	
Нормоконтролер	Айтказина А. А.	02.04.19 – 15.05.19	

ГРАФИК
подготовки дипломной проекта

Наименование разделов, перечень разрабатываемых вопросов	Сроки представления научному руководителю	Примечание
Аналитическая часть	05.11.2018 - 21.12.2018	
Проектная часть	07.01.2019 - 31.01.2019	
Разработка десктопного приложения	04.02.2019 - 12.04.2019	

Дата выдачи задания «14» января 2018 г.

Заведующий кафедрой ______________________ Т.С. Картбаев

Научный руководитель проекта ________________ Б. А. Тулегенова

Задание принял к исполнению студент ______________________ Р. А. Баяхметов

Аңдатпа

Дипломдық жобаның тақырыбы «Бірыңғай деректер базасын басқару ортасын құру». Жобаның мақсаты - дерекқор әкімшілері мен бағдарламалық жасақтаманы әзірлеушілердің жұмысын талдау, байланыс түрлеріне қарамастан бірыңғай интерфейс пен бірыңғай функционалдылықты пайдалана отырып, дерекқорды басқару жүйелерінің әртүрлі түрлерімен жұмысын жеңілдететін қосымшаны әзірлеу. Жобаны орындау барысында кросс-платформаны, құралдың бірегейлігін, анық интерфейсті және функционалдылықты шешу қажет. Осы мақсатқа қол жеткізу үшін JVM, Kotlin, Gradle, Git, JavaFX, TorandoFX, CSS, FXML технологиялары пайдаланылды. Диссертациялық жұмыста кіріспе, бес тарау, әр тараудың қорытындысы және қорытынды қорытынды бар. Нәтижесінде, бағдарламалық өнімдерді жобалау мен дамытудың заманауи әдістері зерттелді және одан әрі өнім әзірлеуге арналған жоспар жасалды.

Аннотация

Тема дипломного проекта «Разработка интегрированной среды администрирования баз данных». Целью проекта является проведение анализа работы администраторов баз данных и разработчиков программного обеспечения, разработка приложения, упрощающее работу с различными типами систем управления базами данных с помощью унифицированного интерфейса и единой функциональности вне зависимости от типа подключения. Во время выполнения проекта необходимо решить проблему кроссплатформенности, уникальности инструмента, очевидности интерфейса и функционала. Для достижения поставленной цели использовались технологии JVM, Kotlin, Gradle, Git, JavaFX, TorandoFX, CSS, FXML. В дипломный проект входит введение, пять глав, выводы по каждой главе и итоговое заключение. В итоге изучены современные методы проектирования и разработки программных продуктов, составлен план дальнейшего развития продукта.

Annotation

Theme of the thesis project «Development of an integrated database administration environment». The goal of the project is to analyze the work of database administrators and software developers, to develop an application that simplifies working with various types of database management systems using a unified interface and uniform functionality regardless of the type of connection. During the execution of the project, it is necessary to solve the problem of cross-platform accessibility, uniqueness of the tool, obvious interface, and functionality. To achieve this goal, technologies, such as JVM, Kotlin, Gradle, Git, JavaFX, TorandoFX, CSS, FXML were used. The graduation project includes an introduction, five chapters, conclusions for each chapter and a final conclusion. As a result, modern methods of designing and developing software products were studied, and a plan for further product development was drawn up.

Содержание

	Введение	8
1	Описание программного продукта	9
1.1	Назначение программного продукта	10
1.2	Потенциальные пользователи программного продукта	11
1.3	Обзор существующих аналогичных программных продуктов	11
1.4	Постановка цели и задач	13
2	Структура программного продукта	14
2.1	Требования к архитектуре и технологическому стеку	14
2.2	Описание жизненного цикла и архитектуры	16
2.3	Подготовка и настройка рабочего места	21
2.4	Выбор библиотек и фреймворков	22
2.5	Требования к пользовательскому интерфейсу	23
2.6	Прототип интерфейса	25
3	Разработка программного продукта	28
3.1	Создание первоначальной структуры проекта	28
3.2	Создания пакетов архитектуры и подключение зависимостей	30
3.3	Разработка графического интерфейса	32
3.4	Описание пакетов инфраструктуры.	39
3.5	Описание моделей данных	40
3.6	Описание слоя доступа к данным	45
4	Экономическая часть	45
4.1	Трудоёмкость разработки программного продукта	48
4.2	Расчет затрат на разработку ПП	49
4.3	Определение возможной (договорной) цены ПП	55
5	Охрана труда и безопасность жизнедеятельности	56
5.1	Расчет естественного освещения	59
5.2	Расчет искусственного освещения	62
5.3	Итоги	65
	Заключение	67
	Список литературы	68
	Приложение А. Техническое задание	70
	Приложение Б. Листинг программы	75

Введение

Активное развитие современного бизнеса в сторону цифровизации предприятий породило появление на рынке большого количества систем управления базами данных. Каждая из появившихся СУБД имеет свои плюсы и минусы, что позволяет ей быть фаворитом в решении какой-либо определенной задачи. Это позволяет создавать гибкие и надежные информационные инфраструктуры, которые могут включать в себя несколько различных СУБД.

Использование нескольких СУБД делает инфраструктуру более ячеистой и децентрализованной, избавляя от большой нагрузки в одной БД и общей точки отказа. Но такой подход значительно усложняет обслуживание системы. Администраторам и пользователям необходимо иметь множество приложений для доступа к каждой СУБД, что может отрицательно влиять на скорость и качество работы сотрудников. Для решения данной проблемы необходимо избавиться от большого количества разнообразных интерфейсов специализированных программ и централизовать управления всеми СУБД.

От скорости и качества работы всех сотрудников зависит весь коммерческий успех бизнеса. На администраторов и разработчиков возлагается большая ответственность за сохранность данных и правильность их обработки. Любая ошибка может стоить предприятию больших ресурсных затрат. Например, пользователь, который ранее работал только с СУБД Oracle и только через консольный интерфейс, при первом подключении к PostgreSQL через PgAdmin случайно может выполнить запрос не для той СУБД, не отключив при этом автокоммит. Теперь, из-за особенностей работы определенного приложения и человеческого фактора, администраторам придется тратить время на восстановление актуальных данных, вместо выполнения текущих поставленных задач. Подобных ситуаций на производстве может быть много, и их количество прямо пропорционально зависит от количества используемых СУБД и приложений, для управления ими. Следовательно, инструментарий данных сотрудников должен максимально минимизировать вероятность ошибки своего пользователя, что очень важно для бизнеса.

Целью дипломного проекта является разработка унифицированного приложения, которое снизит вероятность ошибки в работе своих пользователей и повысит их продуктивность.

Результатом выполнения проекта будет готовый программный продукт для администраторов и других пользователей СУБД. Данное приложение будет простым в использовании и освоении, не будет содержать функциональность или создавать ситуаций, результаты которых сложно предсказать или он является неопределенным.

1 Описание программного продукта

В настоящее время ни один бизнес не обходится без цифрового хранения и обработки своей информации. Разнообразные цели предпринимателей порождают большое количество решений, что позволяет создавать более гибкие архитектуры программно-аппаратных комплексов. Особенно это заметно в области хранения цифровых данных: обилие решений в среде систем управления базами данных, разнообразие их ориентированности и подходов создает ситуацию, когда использование только одной СУБД на предприятии является не целесообразным и ограничивает развитие информационной инфраструктуры. Следовательно, при развитии IT сектора в рамках одного бизнеса могут использоваться несколько различных СУБД. Например, для ведения учетов транзакций, где потеря данных является катастрофической, бизнес будет использовать надежную и дорогою СУБД «Oracle RDBMS», а для предоставления информации клиентам, во избежание дополнительной нагрузки на основную СУБД, данные будут реплицироваться в открытую документоориентированную СУБД «MongoDB». Аналогичных подходов в рамках одного бизнеса может быть очень много, что значительно усложняет работу администраторам баз данных и разработчикам информационных систем. Для поддержки и выполнения ежедневных задач, сотрудникам необходимо использовать большое количество приложений для каждой из СУБД. Этот список может состоять как из консольных интерфейсов, которые не являются дружелюбными и требуют значительно большего сосредоточения; так и из специализированных приложений, позволяющих комфортно управлять базами данных и понижающих вероятность ошибки и уровень входа. Каждое приложение имеет собственный стиль, интерфейс и подход к решению определенных задач, что создает некоторые трудности для пользователей и может привести к возникновению замедления работы или даже ошибкам во время использования, а любая ошибка при работе с данными, может стоить очень дорого.

Для решения описанной выше проблемы существуют приложения, которые агрегируют в себе доступы и управление различными СУБД. Главное преимущество данных приложений в унифицированном интерфейсе, не зависимом от СУБД, что экономит время пользователей и помогает им освоиться с новой СУБД намного быстрее. Так же данные приложения экономят постоянную и оперативную память на рабочем компьютере пользователя, так как ему необходимо хранить и запускать только один экземпляр приложения для доступа ко всем СУБД. К минусам таких приложений можно отнести большую нагрузку на систему, в сравнении с одним экземпляром специфицированного приложения и отсутствие уникального функционала, свойственных определенным СУБД.

1.1 Назначение программного продукта

Разрабатываемый программный продукт предназначен для использования подключения к системе управления базами данных, выполнению запросов и просмотру результатов этих запросов. Главной целью продукта является предоставление пользователю удобного и единого интерфейса взаимодействия с базами данных для снижения вероятности возникновения ошибок на фоне большого количества разнообразных СУБД и снижения порога входа сотрудников.

Работу с главной функциональной частью приложения можно разделить на три этапа:

– создание подключения к базе данных. На этом этапе пользователю необходимо зарегистрировать в приложении соединение с какой-либо СУБД. В случае первого запуска приложения этот пункт является обязательным для дальнейшего использования основной функциональности. Создание соединения инициируется пользователем путем запуска сценария регистрации соединения. Первым шагом, необходимо выбрать тип СУБД, к которому будет создаваться подключение, так как от выбора зависит необходимые данные для создания подключения. После выбора СУБД пользователю нужно ввести некоторые данные: название для соединения, хост, порт, название базы данных, имя пользователя, пароль. Соединение и дальнейшая работа с СУБД будут происходить от лица указанного аккаунта, все ограничения и доступы будут соответствовать аккаунту. После успешного создания соединения между приложением и СУБД, пользователь может перейти к выполнению запросов;

– написание запросов к базе данных. Большая часть работы с базами данных заключается в создании, а иногда и анализа запросов на SQL или его подобных языках. Для написания запросов в приложении выделена большая часть рабочего пространства, которое возможно настроить под себя. При создании запроса система предоставляет пользователю вспомогательный функционал, который нацелен на облегчение написания и чтения введенного текста. Первым вспомогательным функционалом является нумерация строк, цель которой упростить навигацию по запросу или запросам в случае их большого объема. Не менее важным функционалом, упрощающим навигацию, является поиск по тексту запроса. Поиск поддерживает регулярные выражения, что предоставляет пользователю огромную гибкость в поиске не только по полному совпадению строк, но и по описанным им шаблонам. Так же, функционал поиска может использоваться для замены определенного текста. Самым важным помощником, при создании и чтении запросов будет подсветка синтаксиса SQL стандарта ANSI. Данный функционал создает цветовое выделение ключевых слов в запросе, что упрощает чтение и понимание запроса. По проведенным исследованиям ученых Кембриджского университета, подсветка синтаксиса ускоряет понимание кода на несколько

секунд, при его небольших объемах, а при больших объемах разница увеличивается [1];

– выполнение запроса и просмотр его результатов. У пользователя есть несколько путей к запуску выполнения запроса. Он может запустить весь код, находящийся на рабочей панели нажатием соответствующей кнопки или сочетанием клавиш, при этом, программа разделит весь текст на подзапросы по специальным символам разделителям, которые можно указать в настройках и выполнит отдельные запросы независимо друг от друга. Так же пользователь может выделить какую-то часть запроса в рабочей области и запустить выполнение только выделенной части по тем же правилам. Все запущенные запросы будут выполняться параллельно и не зависят друг от друга. Результаты выполнения каждого запроса будут отображены в нижней панели в соответствующем результату виде, т. е. при неудачном выполнении запроса, будет выведено сообщение об ошибке и показаны рекомендации по её исправлению. В случае удачного выполнения запроса панель будет содержать результирующую таблицу. Пользователю доступны некоторые манипуляции над этой таблицей: сортировка результатов по столбцам, поиск значений в ячейках.

1.2 Потенциальные пользователи программного продукта

Разрабатываемое приложение в основном нацелено на использование его администраторами баз данных и разработчиками информационных систем. Но оно будет полезно всем, кому необходимо подключаться к СУБД для выполнения запросов. В частности, сюда могут относиться бизнес аналитики, топ менеджера крупных финансовых организаций и другие сотрудники.

Так же приложение может использоваться в учебных заведениях, для обучения студентов работе с базами данных в дисциплинах, где работа с СУБД не является основным направлением, но базовые знания необходимы для выполнения основных заданий.

Простота и интуитивная понятность графического интерфейса значительно снижают порог входа и повышают скорость работы в сравнении с классическими консольными интерфейсами. Что, в свою очередь может сэкономить время сотрудников, выступающих в качестве менторов для своих младших коллег. Т. е. уменьшается время адаптации сотрудников, в обязанностях которых есть работа с базами данных, что в свою очередь может положительно сказаться на KPI всех сотрудников.

1.3 Обзор существующих аналогичных программных продуктов

В процессе анализа поставленной задачи и предметной области было найдено несколько существующих аналогов приложения, выполняющих схожие функции, но в реализацию которых, были заложены абсолютно

разные подходы. Был проведен анализ данных приложений для создания более полной картины о предметной области и сбора полезной информации и функционала, который может быть полезен в реализуемом приложении.

Далее приводится список программных продуктов, аналогичных разрабатываемому продукту.

DB Visualizer - продукт компании DbVis Software, выпущенный и поддерживаемый с 1999 года. Продукт представляется как универсальный инструмент для работы с базами данных. Распространяется по условно бесплатной лицензии, т. е. есть бесплатная, ограниченная по функциональности версия, и «Pro» – полная версия. Продукт позволяет выполнять подключение и администрирование любой СУБД из списка поддерживаемых, так же имеет гибкие настройки и удобный интерфейс. Редактор SQL поддерживает подсветку синтаксиса, подсказки и выделение возможного места ошибки, может выполнять несколько запросов одновременно и разделять результаты этих запросов. Есть возможность просмотра структуры базы и построения диаграмм.

Проблемой данного продукта является его политика лицензирования. Многие важные и полезные функции, например, выполнение только выделенного кода, доступны только в платной версии программы. Так же, программа занимает очень много оперативной памяти, и при недостаточном ее количестве, приложение может рушиться из-за переполнения стека.

DBeaver – многоплатформенный инструмент для работы с базами данных. Приложение имеет открытый исходный код и распространяется по лицензии «Apache License v2». Поддерживает все популярные базы данных. Это пользовательское приложение основано на платформе Eclipse IDE и имеет схожий с ним интерфейс. DBeaver имеет стандартный функционал подключения к базе данных и выполнения запросов, имеет множество настроек и большое количество информационных панелей. Так как код данного приложения доступен на платформе GitHub, приложение активно поддерживается сообществом разработчиков со всего мира и имеет множество полезных функций. Но это же является и минусом продукта. Устаревший интерфейс и обилие лишней информации могут отвлечь пользователя от главной задачи, а в обилии информационных панелей сложно ориентироваться, что замедляет понимание происходящего на рабочей поверхности и может стать причиной ошибок.

DataGrip – IDE для администрирования баз данных и SQL от компании JetBrains. JetBrains – компания специализирующаяся на разработке компиляторов и инструментов для разработки, самый известный проект компании интегральная среда разработки Intellij IDEA. На 2018 год у компании более 5 миллионов пользователей, среди клиентов: Google, Salesforce, Twitter, Citibank, HP, Airbnb. DataGrip поддерживает большое количество баз данных и предоставляет много функциональности для пользователей. Самым большим преимуществом данной программы является текстовый редактор, функциональность которого обеспечивает комфортную

работу с кодом. Помимо выделения кода, редактор предоставляет следующие выделяющиеся функции: мультикурсор, интеллектуальное дополнение кода в зависимости от контекста, проверка синтаксиса и наличия объектов в базе данных во время написания запроса. Так же, возможна навигация из запросов к описанию объектов базы данных, генерация кода, автоматическое изменение объектов через графический интерфейс. Изменение данных в таблице и сохранение результата в базу данных. Самым большим недостатком данного приложения является его цена. Существует бесплатная пробная версия сроком на месяц, после чего, необходимо купить полную версию системы.

Проведенный анализ аналогичных продуктов показал, что на рынке существует не много аналогичных продуктов, большая часть из которых является либо условно бесплатными, либо полностью платными.

1.4 Постановка цели и задач

Ниже приведены цель дипломного проекта и задачи, которые необходимо реализовать для достижения поставленной цели.

Целью дипломного проекта является разработка программного продукта, который позволит повысить скорость и качество работы с базами данных.

Основные задачи дипломного проекта:

– изучение предметной области на примере виртуальной IT-компании;
– определение основных требований к программному продукту;
– выбор и обоснование технологий для разработки программного продукта;
– проектирование архитектуры программы;
– создание макета графического интерфейса;
– реализация программного продукта;
– обоснование экономической целесообразности разрабатываемого продукта;
– предложение мероприятий по улучшению условий труда в рамках реализуемого проекта.

2 Структура программного продукта

2.1 Требования к архитектуре и технологическому стеку

Приложение является полностью клиентским приложением, так называемым десктопным, это означает, что оно полностью выполняется на рабочей машине клиента и практически не зависит от глобальной сети Internet.

Приложение должно быть кроссплатформенным, т. е. без проблем запускаться на различных операционных системах и работать везде аналогично.

Приложение должно быть легко расширяемо, для подключения различных типов СУБД и предоставлять доступ к легкому добавлению новых драйверов для новых СУБД или обновлению существующих.

Платформой для приложения в соответствии с указанными требованиями была выбрана JRE версии 11, которая может обеспечить кроссплатформенность разработанного приложения без увеличения кодовой базы. Java Runtime Environment — минимальная реализация виртуальной машины, необходимая для исполнения Java-приложений, без компилятора и других средств разработки. Состоит из виртуальной машины Java Virtual Machine и библиотеки Java-классов. OpenJRE распространяется свободно для большинства платформ. Так же данная платформа открывает доступ для большого количества языков программирования, таких как Java, Scala, Groovy, Kotlin и другие.

В качестве основного языка разработки выбран Kotlin версии 1.3. Kotlin — это относительно молодой статически типизированный язык от российской компании JetBrains, работающий поверх JVM и компилирующийся в байт-код java. Язык полностью совместим с Java и может работать с библиотеками и фреймворками написанные на java. Но в отличие от java, Kotlin имеет множество преимуществ:

– null-безопасность – в языке нет возможности присвоить любому объекту null если не указать явно, что тип является Nullable, иначе произойдет ошибка на этапе компиляции;

– гибкость и простота синтаксиса. Простые функции и структуры можно объявить одной строкой. Геттеры и сеттеры задаются за кулисами для интероперабельности с Java-кодом. Добавление data-аннотации к классу активирует автоматическую генерацию различных шаблонов;

– функции-расширения. Kotlin позволяет расширять функциональность существующих классов, не прибегая к наследованию. Это делается при помощи функций-расширений. Для объявления такой функции к её имени нужно приписать префикс в виде расширяемого типа;

– функциональное программирование. Важно отметить, что Kotlin заточен под функциональное программирование. Он предоставляет большое количество полезных возможностей, например, функции высшего порядка,

лямбда-выражения, перегрузку операторов и ленивые вычисление логических выражений.

Так же в качестве дополнительного инструментария выбраны следующие продукты:

– Gradle 5.0 - система автоматизации сборки. Вместо традиционной XML-образной формы представления конфигурации проекта, предоставляет DSL на языке Groovy или Kotlin. Gradle позволяет автоматизировать рутинные задачи, которые встречаются нам каждый день. Например, он автоматизирует подключение сторонних библиотек в проект, и теперь нет необходимости вручную скачивать архивы, прописывать их в classpath и включать в сборку. Все, что требует Gradle, это прописать название библиотеки и версию в скрипте сборки и обновиться. Система сама загрузит архивы с репозиториев mavencentral, и настроит окружение для работы с ними. Так же, при сборке с помощью Gradle, он автоматически соберет все модули и зависимости и добавит их в результирующий архив.

– Git 2.20.1 – распределённая система управления версиями, созданная Линусом Торвальдсом для управления разработкой ядра Linux. Git помогает управлять процессом разработки программного обеспечения и управлениями его версий. Git считает хранимые данные набором слепков небольшой файловой системы. Каждый раз, когда вы фиксируете текущую версию проекта, Git, по сути, сохраняет слепок того, как выглядят все файлы проекта на текущий момент. Ради эффективности, если файл не менялся, Git не сохраняет файл снова, а делает ссылку на ранее сохранённый файл. Перед сохранением любого файла Git вычисляет контрольную сумму, и она становится индексом этого файла. Поэтому невозможно изменить содержимое файла или каталога так, чтобы Git не узнал об этом. Эта функциональность встроена в сам фундамент Git'а и является важной составляющей его философии. Если информация потеряется при передаче или повредится на диске, Git всегда это выявит. Механизм, используемый Git'ом для вычисления контрольных сумм, называется SHA-1 хешем. Это строка из 40 шестнадцатеричных символов (0-9 и a-f), вычисляемая в Git'е на основе содержимого файла или структуры каталога. Перечисленные выше особенности делают Git очень быстрой и гибкой системой контроля версий, которая позволяет с легкостью оперировать изменениями в коде в рамках одной или нескольких версий.

– IntelliJIDEA – интегрированная среда разработки программного обеспечения от компании JetBrains для многих языков программирования. Эта ide предоставляет огромный функционал для написания качественного кода. Среди главных преимуществ можно выделить следующие: умное автодополнение кода и подсказки, инструменты для анализа качества кода (так называемые линтеры), удобная навигация по проекту, расширенные возможности рефакторинга и форматирования для большинства языков программирования, интеграция с популярными инструментами, как Git, Gradle, PMD, VisualVM.

– VisualVM – это инструмент, который предоставляет визуальный интерфейс для просмотра подробной информации о приложениях, работающих на виртуальной машине Java (JVM). Позволяет просматривать подробную информацию о используемых ресурсах запущенных приложений. С его помощью можно проанализировать уровень использования оперативной памяти кучей и стеком приложений и определить места, которые необходимо оптимизировать по количеству используемой памяти. Так же, можно проанализировать нагрузку на процессор определенных потоков.

– PMD – статический анализатор кода с открытым исходным кодом. Данное программное обеспечение направлено на обнаружение проблем в коде приложения. Проблемы, о которых сообщает PMD, являются неэффективный код или плохие практики программирования.

2.2 Описание жизненного цикла и архитектуры

Жизненный цикл программного обеспечения — это период времени, который начинается с момента принятия решения о создании программного продукта и заканчивается в момент его полного изъятия из эксплуатации. Этот цикл — процесс построения и развития ПО.

Жизненный цикл можно представить в виде моделей. В настоящее время наиболее распространенными являются: каскадная, инкрементная (поэтапная модель с промежуточным контролем) и спиральная модели жизненного цикла.

Для реализации программного продукта была выбрана каскадная модель жизненного цикла.

Каскадная модель процесса разработки программного обеспечения, напоминает собой поток, последовательно проходящий фазы сбора и анализа требований, проектирования решения, реализации, тестирования, внедрения и поддержки (рисунок 2.1).

Процесс разработки реализуется с помощью упорядоченной последовательности независимых шагов. Модель предусматривает, что каждый последующий шаг начинается после полного завершения выполнения предыдущего шага. На всех шагах модели выполняются вспомогательные и организационные процессы и работы, включающие управление проектом, оценку и управление качеством, верификацию и аттестацию, менеджмент конфигурации, разработку документации. В результате завершения шагов формируются промежуточные продукты, которые не могут изменяться на последующих шагах.

Архитектура системы — принципиальная организация системы, воплощенная в её элементах, их взаимоотношениях друг с другом и со средой, а также принципы, направляющие её проектирование и эволюцию.

Для создания полностью клиентского приложения была выбрана так называемая пятиуровневая архитектура. Данный подход построен на основании трехслойной архитектуры, предложенной компанией Microsoft, но

в отличие от нее подразумевает, что приложение разбивается на пять системных уровней, а именно: инфраструктура, модели данных, сервисы, уровень редактирования, уровень представления (рисунок 2.2). Данная архитектура разделяет код на модули со своей зоной ответственности, что способствует более легким поддержке и расширению продукта.

Рисунок 2.1 – Каскадная модель жизненного цикла ПО

Первый слой архитектуры называется инфраструктурным и содержит в себе библиотеки, модули, классы и методы, которые используются во всем приложении. Данный уровень представляет собой агрегацию из разных полезных утилит и вспомогательных классов, что помогает избавиться от избыточной энтропии в выше расположенных уровнях.

Второй уровень называется модель данных. Модель данных представляет собой объектную модель самосериализуемых классов и слой доступа к данным. Данные объекты либо не содержат никакой логики, либо обладают некоторой минимальной логикой, связанной, например, с верификацией данных. Количество методов, которыми обладают эти объекты, тоже невелико. Как правило, они либо предоставляют упрощенный доступ к другим объектам, связанным с первыми, либо отвечают за сериализацию и десериализацию самого объекта. Слой доступа к данным управляет соединение с СУБД и обменом информацией с ней.

Третий уровень — это уровень сервисов или служб представляет собой набор функциональных модулей. Каждый модуль отвечает за реализацию какой-нибудь одной операции, выполняемой над моделью данных. Модули, ответственные за оказание услуг, располагаются на уровне сервисов. Сервисы не зависят или слабо зависят друг от друга. Их можно поместить в разные модули и избежать излишней связности между ними.

Четвертый уровень отвечает за редактирование. Данный уровень характерен для программ-редакторов. Он содержит типовые компоненты или типовые архитектуры редакторов. Прежде всего, имеется в виду:

– архитектуру Документ/Вид;

– Undo/Redo Management.

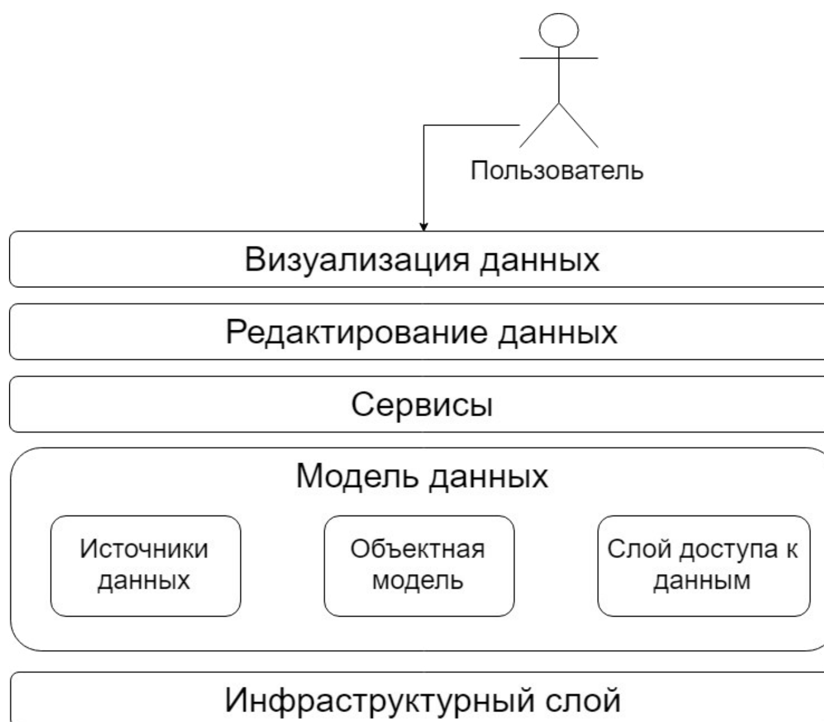

Рисунок 2.2 – Пятиуровневая архитектура.

Архитектура Документ/Вид осуществляет привязку конкретного расширения файла к определенному типу документа, а также привязывает определенный тип документа к определенному виду, ответственному за его визуализацию.

Undo/Redo Management обеспечивает поддержку отмены операции. Благодаря такой возможности пользователь при редактировании сложного документа в любой момент может отменить ошибочно выполненную операцию. В результате редактирование становится устойчивым к ошибкам и не страшным для человека. Такая функциональность стала де-факто стандартом различных программ редактирования.

Уровень редактирования является частью слоя представления классической трехслойной архитектуры. Его выделение в отдельный слой обусловлено тем, что он задает каркас для редактирования. Кроме того, архитектурные концепции, используемые для организации уровня редактирования, не зависят от визуализации данных.

Пятый уровень тоже является ключевым. Без него не обойдется ни одно клиентское приложение. Этот уровень содержит компоненты, отвечающие за визуализацию данных и взаимодействие с пользователем. Исключение из него компонентов, отвечающих за редактирование, позволяет разработчику сконцентрировать свое внимание на визуализации данных и написании разнообразных органов управления.

Разделение между уровнем редактирования и уровнем представления может носить не физический, а ментальный характер. Компоненты обоих слоев могут располагаться в одном модуле.

Для поддержания гибкости и отзывчивости архитектуры, в разработке применяются принципы объектно-ориентированного проектирования SOLID [2]. Данные принципы были названы Робертом Мартином и являются одними из основных принципов поддержания исходного кода приложения в «чистом» виде.

Вот как расшифровывается акроним SOLID:

– Single Responsibility Principle. Принцип единственной ответственности гласит, что каждый объект системы должен иметь только одну ответственность, которая инкапсулируется в класс. «Класс должен иметь только одну причину для изменений», — Роберт Мартин [3].

– Open-Closed Principle. Принцип открытости-закрытости - программные сущности (классы, модули, функции) должны быть открыты для расширения, но не для модификации.

– Liskov Substitution Principle. Принцип подстановки Барбары Лисков - необходимо, чтобы подклассы могли бы служить заменой для своих суперклассов.

– Interface Segregation Principle. Принцип разделения интерфейса - создавайте узкоспециализированные интерфейсы, предназначенные для конкретного клиента. Клиенты не должны зависеть от интерфейсов, которые они не используют.

– Dependency Inversion Principle. Принцип инверсии зависимостей - объектом зависимости должна быть абстракция, а не что-то конкретное.

Большое количество задач, с которыми очень часто приходиться сталкиваться в процессе разработки, уже давно решались другими программистами. Существует множество книг, статей и руководств о т лучших практиках программирования, в которых разработчики делятся хорошими решениями часто встречающихся задач. Очень важно учитывать и использовать опыт других разработчиков при создании собственного продукта, так как это убережет разработчика от создания собственных «велосипедов» с большими затратами времени на тестирование и исправление ошибок.

Одним из лучших следствий передачи опыта обществу разработчиками стали шаблоны проектирования. Шаблоны проектирования – это руководства по решению часто повторяющихся задач. Паттерны проектирования не являются готовыми решениями, которые можно трансформировать непосредственно в код, а представляют общее описание решения проблемы, которое можно использовать в различных ситуациях. Концепцию паттернов впервые описал Кристофер Александер в книге «Язык шаблонов. Города. Здания. Строительство». В книге описан «язык» для проектирования окружающей среды, единицы которого — шаблоны (или паттерны, что ближе к оригинальному термину patterns) — отвечают на архитектурные вопросы:

какой высоты сделать окна, сколько этажей должно быть в здании, какую площадь в микрорайоне отвести под деревья и газоны. Идея показалась заманчивой четвёрке авторов: Эриху Гамме, Ричарду Хелму, Ральфу Джонсону, Джону Влиссидесу. В 1995 году они написали книгу «Приемы объектно-ориентированного проектирования. Паттерны проектирования», в которую вошли 23 паттерна, решающие различные проблемы объектно-ориентированного дизайна. Название книги было слишком длинным, чтобы кто-то смог всерьёз его запомнить. Поэтому вскоре все стали называть её «book by the gang of four», то есть «книга от банды четырёх», а затем и вовсе «GOF book». Паттерны проектирования не зависят от языка или технологии, так как являются только принципами решения задач, поэтому их использование возможно на любой архитектуре. Шаблоны проектирования позволяют упростить код решения, но так же могут сделать его трудным для понимания, поэтому их необходимо использовать с полным пониманием предметной области, целей проекта, задачи и самих паттернов[4].

MVC (model-view-controller) – один из самых популярных составных шаблонов проектирования. Разделяет данные приложения, пользовательской логики и представления данных на три отдельных компонента: модель, представление и контроллер, таким образом, что модификация каждого из них может производиться независимо. Модель представляет данные и реагирует на команды контроллера, изменяя свое состояние. Представление, отвечает за отображение данных модели пользователю, реагируя на изменение модели. Контроллер, интерпретирует действия пользователя, освещая модель о необходимых изменениях. Основная цель шаблона – отделить логику от представления данных. За счет такого разделения повышается возможность повторного использования кода. MVC включает в себя несколько шаблонов проектирования:

– наблюдатель. Паттерн используется для автоматического распространения изменений в источниках данных;

– стратегия. Используется для создания и выбора необходимого котроллера представлению;

– для создания сложносоставного представления используется паттерн компоновщик.

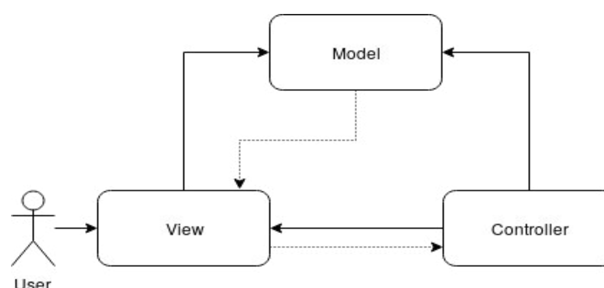

Рисунок 2.3 – Схема работы MVC

Dependency Injection (внедрение зависимостей) – процесс предоставления внешней зависимости программному компоненту. В соответствии с принципом единой зависимости объект отдает заботу построении требуемых ему зависимостей внешнему, специально предназначенному для этого общему механизму.

2.3 Подготовка и настройка рабочего места

Операционной системой для рабочей среды выбран дистрибутив ядра linux, Ubuntu 18.10, обладающей преимуществами: стабильность, легкое изменение переменных сред, простое использование консоли, большое количество доступного ПО и хорошая поддержка. Для удобства работы с различным софтом применяется некоторый принцип установки программного обеспечения: в домашней папке пользователя создается папка с наименованием soft куда разворачиваются все устанавливаемые программы. В переменную среды PATH добавляется путь к папке /home/{username}/bin, в дальнейшем, в этой папке создаются ссылки на исполняемые файлы для всех необходимых программ. Теперь любую из данных программ можно запустить из BASH выполняя команду, соответствующую имени ссылки. Данный подход удобен в случае, если в системе присутствует только один пользователь и необходима работа множества программ с возможностью быстрого переключения между несколькими их версиями.

Перед началом разработки на платформе JVM необходимо установить jdk. Java Development Kit (сокращенно JDK) — бесплатно распространяемый компанией Oracle Corporation комплект разработчика приложений на языке Java, включающий в себя компилятор Java (javac), стандартные библиотеки классов Java, примеры, документацию, различные утилиты и исполнительную систему Java (JRE).

Для установки нужно скачать архив jdk 11.0.2 с официального сайта oracle и распаковать в папку soft. Желательно создать переменную среды JAVA_HOME с указанием ей полного пути к jre[5].

Далее можно скачать и установить необходимые инструменты:

- Intellij IDEA;
- VisualVM;
- Gradle;
- Git.

Скачать данные инструменты можно на их официальных сайтах. После загрузки архивов программ они распаковываются в папку soft и на их исполняемые файлы создаются «мягкие» ссылки в папке bin.

Теперь необходимо настроить инструменты на совместную работу. Запустить Intellij IDEA и в настройках указать пути к JDK, JRE, Gradle, Git. Для интеграции с VisualVM нужно скачать плагин VisualVM Launcher. Для работы с Kotlin, необходимо наличие одноименного плагина.

Для начала работы с git, необходимо предоставить ему своё имя и почту, которые будут использоваться для идентификации коммитов. Для легкого доступа к проекту с другого устройства или другим пользователям git, проекту необходим удалённый репозиторий. В качестве веб хостинга выбран крупнейший среди аналогий веб-сервис – GitHub. На данном сервисе существует возможность создания бесплатного аккаунта и бесплатных публичных или закрытых (с некоторыми ограничениями) репозиториев.

2.4 Выбор библиотек и фреймворков

При решении задач разработки программного обеспечения огромную роль играет возможность использование опыта других разработчиков, столкнувшихся с подобными проблемами.

Библиотека – сборник программ или объектов, используемый для разработки программного обеспечения. Библиотека предоставляет некоторые готовые решения, с возможностью их использования при разработке других программ.

Фреймворк – заготовка, шаблон для программного обеспечения, определяющая структуру ПО; программное обеспечение, облегчающее разработку и объединение разных модулей одного проекта.

«Фреймворк» отличается от понятия библиотеки тем, что библиотека может быть использована в программном продукте просто как набор подпрограмм близкой функциональности, не влияя на архитектуру программного продукта и не накладывая на неё никаких ограничений. «Фреймворк» же диктует правила построения архитектуры приложения, задавая на начальном этапе разработки поведение по умолчанию — «каркас», который нужно будет расширять и изменять согласно указанным требованиям.

При разработке программного продукта будут использоваться следующие библиотеки и фреймворки:

– JDBC — платформенно-независимый промышленный стандарт взаимодействия Java-приложений с различными СУБД, реализованный в виде пакета java.sql, входящего в состав Java SE. JDBC основан на концепции так называемых драйверов, позволяющих получать соединение с базой данных по специально описанному URL. Драйверы могут загружаться динамически (во время работы программы). Загрузившись, драйвер сам регистрирует себя и вызывается автоматически, когда программа требует URL, содержащий протокол, за который драйвер отвечает [6][7];

– JavaFX — платформа на основе Java для создания приложений с насыщенным графическим интерфейсом. Может использоваться как для создания настольных приложений, запускаемых непосредственно из-под операционных систем, так и для интернет-приложений (RIA), работающих в браузерах, и для приложений на мобильных устройствах. JavaFX призвана заменить использовавшуюся ранее библиотеку Swing;

– TornadoFX – модуль, позволяющий минимизировать код для GUI [8];
– ANTLR – анализатор формальных языков;
– yFiles – библиотека, содержащая элементы управления пользовательским интерфейсом для рисования, просмотра и редактирования диаграмм, а также компоновки графиков для автоматического упорядочения сложных графиков и сетей одним нажатием кнопки;
– Apache Commons — это большая коллекция маленьких Java-утилит.

2.5 Требования к пользовательскому интерфейсу

Графический пользовательский интерфейс (GUI)—это последовательный визуальный язык, который используется для представления информации, хранящейся в компьютере. Интерфейс помогает людям, не обладающим специальными навыками работы на компьютере, использовать программное обеспечение.

Первый принцип — это прозрачность интерфейса. Интерфейс должен быть легким для освоения и не создавать перед пользователем преграду, которую он должен будет преодолеть, чтобы приступить к работе. Так как разрабатываемый продукт нацелен не только на продвинутых пользователь SQL, но и на новичков, к интерфейсу добавляется требование в оказании помощи при составлении запроса к СУБД. Например, подсветка синтаксиса в окне ввода запроса, функционал графического построения запроса или результата запроса и т. д.

Второй принцип часто нарушают те авторы программ, которые слишком недооценивают умственные способности пользователей. Это обусловлено разными причинами.

Во-первых, традиционным слегка высокомерным отношением программистов к простым пользователям. Это еще можно было понять в восьмидесятых и начале девяностых годов XX века, когда обычные персональные компьютеры не имели доступных широкой аудитории программных и аппаратных средств для построения привлекательных графических интерфейсов и работы с ними. Самой распространенной операционной системой в то время была MS DOS, основанная на интерфейсе командной строки. Поэтому эффективно работать с персональным компьютером могли люди только с довольно серьезной подготовкой. Кроме того, парк “персоналок” был относительно невелик даже в США, не говоря уже об остальных странах, и, как следствие, число пользователей компьютеров было небольшим.

Сегодня же такой пренебрежительный взгляд на пользователя явно неуместен. Работа с персональным компьютером предполагает относительно не большую начальную подготовку пользователя: интерфейсы компьютерных программ, в первую очередь операционной системы Windows, являющейся законодателем мод в индустрии массового программного обеспечения,

становятся все проще и доступнее для понимания людей. Да и число компьютеров в мире сегодня в несколько раз больше, чем десять лет назад.

Вторая причина слишком большой недоверчивости программистов к познаниям и квалификации пользователей – чрезмерное увлечение построением встроенной защиты от действий пользователя, не соответствующих правилам эксплуатации. Дело в том, что классические учебные курсы по программированию учат, что большинство ошибок в работе программы вызываются не дефектами исходного кода или программного окружения, а действиями пользователя — например, вводом данных неправильного формата (допустим, текста вместо цифр). Поэтому программист при разработке приложения должен написать функции по проверке результатов как можно большего числа действий пользователя и предусмотреть максимальное количество вариантов развития событий. Это совершенно правильный подход, но многие программисты настолько усложняют защиту от неправильных действий, делают ее такой громоздкой, что работа пользователя с программой начинает напоминать известное «шаг вправо, шаг влево считается побегом». Происходит довольно обычная вещь: то, что задумывалось как решение проблемы, само начинает создавать проблемы.

И, наконец, третья причина во многом обусловлена поведением самих пользователей. Часто при возникновении малейших затруднений при работе с программой пользователь тут же обращается в службу технической поддержки, не удосужившись даже взглянуть на справочную систему продукта, посмотреть секцию «Ответы на частые вопросы» на Web-сайте программы или даже просто чуть-чуть подумать! Отчасти тут вина самих авторов программ. Как говорят опытные разработчики пользовательских интерфейсов: «Если уже на этапе знакомства с программой пользователь вынужден обращаться к справочной системе, над интерфейсом нужно серьезно работать». Поэтому, чтобы соблюсти второй из общих принципов построения интерфейсов и не давать пользователю повода почувствовать, будто его принимают за идиота, не нужно давать разрабатываемой программе слишком большие полномочия и право указывать пользователю, что именно ему делать. Некоторые программисты не знают или не желают осознавать этого и загоняют пользователей своих программных продуктов в тесные рамки, навязывая определенный стиль работы.

Один из примеров такого неправильного отношения к пользователю является отказ программы выполнить вполне естественную с точки зрения пользователя программных продуктов такого рода операцию и вывод диалогового окна, требующего выполнить какую-то другую последовательность действий. Этим «прославился», например, известный текстовый редактор «Блокнот» из состава Windows 95. Если пользователь открывал эту программу и решал перед началом набора текста дать создаваемому «Блокнотом» по умолчанию файлу «Untitled» какое-нибудь имя, выбрав из меню команду Сохранить как, редактор отказывался сделать это,

показывая сообщение: «Вы не ввели какой-либо текст, чтобы его можно было сохранить. Наберите какой-нибудь текст, а затем попытайтесь [сохранить его] снова». Этим создатели «Блокнота» не только отвергли стиль работы очень многих пользователей (перед началом набора текста дать имя файла), но сбили с толку и тех, кто был знаком с «Блокнотом» по предыдущим версиям Windows. Например, в шестнадцатиразрядной Windows 3.1 «Блокнот» позволял сохранять пустые файлы безо всяких проблем. Опытные пользователи, знакомые с принципами работы операционной системы, тоже были в недоумении: если из контекстного меню Проводника Windows в меню «Создать» выбрать пункт Текстовый документ, то получившийся файл длиной 0 байт открывается «Блокнотом» без каких-либо затруднений. К счастью, в последующих версиях Windows «Блокнот» стал более дружественен к пользователю.

Другой пример недооценки возможностей пользователя — вывод информационных сообщений в ситуациях, когда этого не требуется. Многие авторы наделяют свои программы излишней «болтливостью» из благих намерений — например, для того чтобы облегчить освоение продукта или информировать пользователей о полезных функциях программы. Однако вполне может оказаться так, что пользователь уже достаточно уверенно чувствует себя при работе с программой и не нуждается в подсказках, выскакивающих каждую минуту, а некоторые полезные, с точки зрения автора программного продукта, функции для конкретного пользователя таковыми не являются. Поэтому среди разработчиков программного обеспечения хорошим тоном считается предоставление пользователю возможности отключить вывод информационных сообщений. Это позволяет сохранить легкость освоения продукта для начинающих пользователей и одновременно с этим добиться, чтобы информационные сообщения не вызывали у опытных пользователей раздражения.

И, наконец, третий принцип — «Программа должна работать так, чтобы пользователь не считал компьютер глупым».

Несмотря на стремительное развитие информационных технологий, многие компьютерные программы все еще имеют примитивный искусственный интеллект. Они прерывают работу пользователя глупыми вопросами и выводят на экран бессмысленные сообщения, повергая его в недоумение в самых простых ситуациях.

2.6 Прототип интерфейса

Прототипирование интерфейса – это продумывание содержания и расположение важных элементов интерфейса. Составляя прототип заранее, можно облегчить задачу всем, кто занимается созданием и наполнением интерфейса. Кроме того, что упрощается создание продукта, прототипирование помогает сделать жизнь пользователей лучше. То есть добавить что-то полезное или убрать что-то ненужное, чтобы было легче

ориентироваться на странице, найти нужную информацию и решить поставленную задачу. Для создания прототипа был выбран онлайн сервис www.figma.com. Данный инструмент обладает широким функционалом создания макетов интерфейсов на основе векторной графики с возможность. Экспорта отдельных элементов в точечную или векторную графику.

На основе созданных требований разработан прототип пользовательского интерфейса (рисунок 2.4).

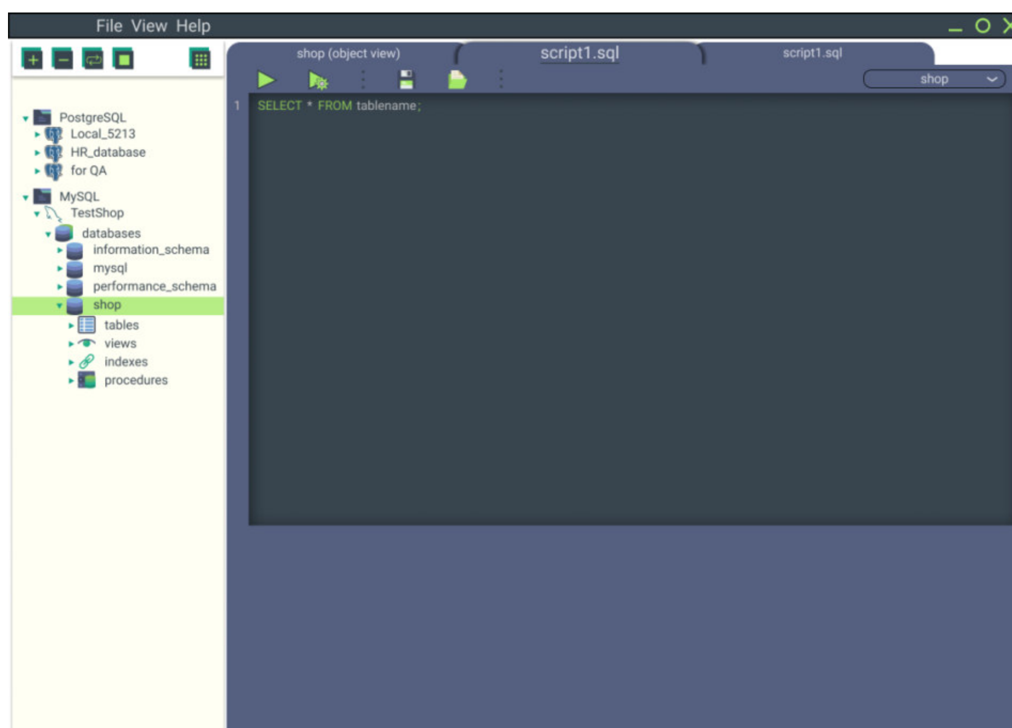

Рисунок 2.4 - Первичный прототип интерфейса

Данный прототип разделяет пользовательский интерфейс на четыре зоны (рисунок 2.5).

– зона №1 отвечает за управление приложением и его настройками. Зона включает в себя меню, агрегирующее в себе полное управление настроек;

– зона №2 является представлением всех созданных подключений к СУБД и списком содержащихся в них объектах. Так же включает в себя функционал управления соединениями с СУБД;

– зона №3 – рабочая область. В данной зоне происходит управление выбранными объектами СУБД или редактирование и выполнение запросов к выбранной схеме;

– зона №4 отвечает за обратную связь приложения. В данной области отображаются все сообщения от задач и результаты запросов в СУБД.

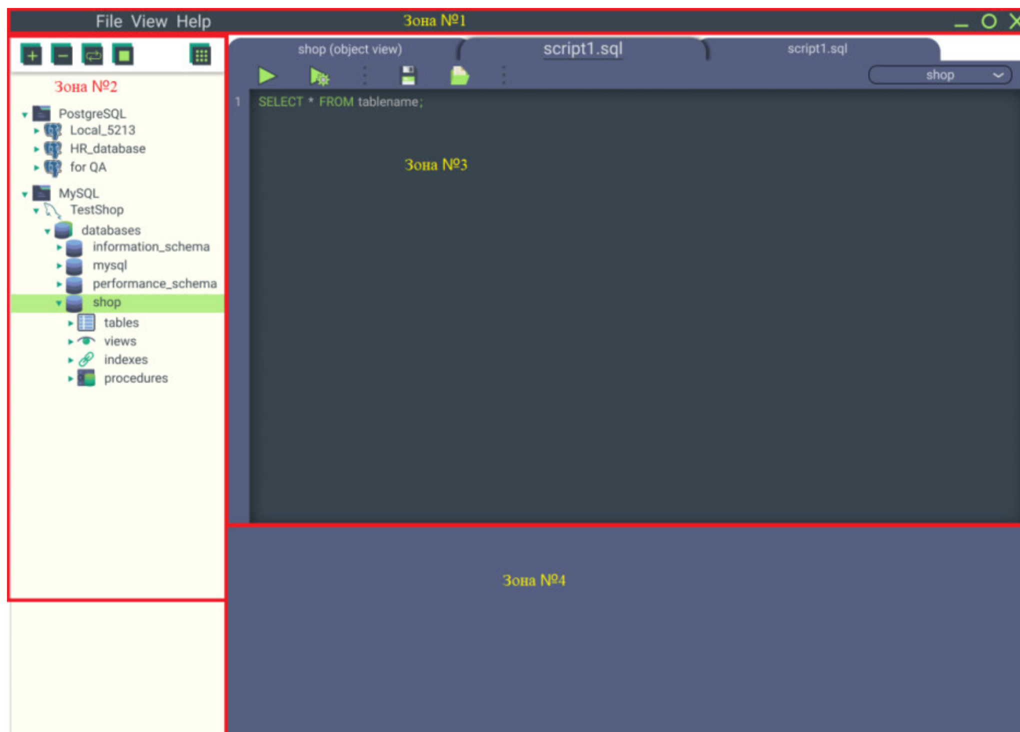

Рисунок 2.5 – Зонировани интерфейса

3 Разработка программного продукта

3.1 Создание первоначальной структуры проекта

Средствами IntellijIdea создается структура пустого Gradle проекта [9][10]. Данная структура представляет собой организацию файлов проекта с выделением исходного кода от других файлов, вроде ресурсов или тестов. При создании проекта указывается основная SDK, в этом случае выбрана oracleJDK 10.0.2. В качестве дополнительных библиотек загружается Kotlin/JVM версии 1.3.11. Так как Gradle основывается на Maven, он перенял основные принципы его структуры и наименований. Таким образом, по конвенции наименования проекта, нам необходимо указать groupID и artifactID и версию для проекта. GroupID – наименование подразделения или организации, обычно использует такие же правила как наименования пакетов в Java – записывают доменное имя, имя организации или подразделения. В данном случае, указываем groupId как “kz.aupet.vt152”. ArtifactID – название проекта: “dbunitool”. Эти данные используются для однозначной идентификации сборок проекта, которые будут иметь вид: goupID:artifactID:version. Далее производится настройка Gradle. Включается автоимпорт для автоматического обновления зависимостей при изменении файлов gradle.build. Выбирается версия Gradle и способ ее подключения, здесь рекомендуемым параметром является использование так называемого gradle wrapper, во избежание проблем с версионностью инструмента на различных машинах. Следующим шагом указывается директория, в которой будет храниться исходный код. Для удобства, все проекты, использующие в качестве системы контроля версий git, хранятся в директории ~/git. После создания проекта в этой папке с названием «dbunitool», получается начальная структура проекта (рисунок 3.1.1).

В данной структуре, весь исходный код и файлы программы содержатся в папке *src* и делится на основные файлы программы, содержащиеся в папке src/main/ эти файлы, войдут в сборку, и тестовые файлы программы, содержащиеся в модуле src/test, предназначенные для тестирования и игнорируются при сборке. Так же в корневой папке программы создались и другие элементы. Папки .gradle и .idea – локальные настройки соответствующих программ. Папка gradle содержит в себе gradle wrapper для использования его на других машинах. Файлы gradlew и gradlew.bat скрипты для запуска gradle wrapper на *nix и windows системах соответственно. Settings.gradle – файл со скриптом на языке groovy, содержащий настройку всего проекта. На данном этапе файл содержит в себе только имя проекта. Gradle.properties – содержит значения флагов командной строки для запуска скриптов. Build.gradle – описание настроек сборки, автоматизация различных задач и указание зависимостей на языке Groovy DSL. Данный файл создается для каждого модуля проекта и содержит описание зависимостей и задач, которые может выполнять gradle.

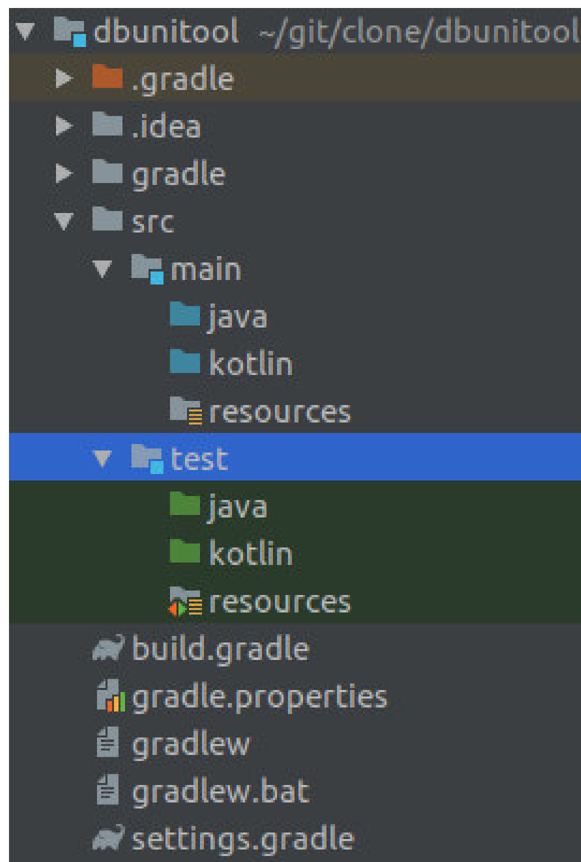

Рисунок 3.1.1 - Первоначальная структура проекта

После создания первичной структуры проекта необходимо создать локальный репозиторий git для ведения управления версиями [11][12]. Для этого в корневой папке проекта необходимо выполнить команду git init. Команда инициализирует пустой git репозиторий в корне проекта, данный репозитория можно увидеть в папке .git/.

```
rias@rias-Aspire-A715-71G:~/git/clone/dbunitool$ git init
Initialized empty Git repository in /home/rias/git/clone/dbunitool/.git/
```

Рисунок 3.1.2 - Создание пустого репозитория

После создания пустого репозиторя создается файл .gitignore для описания паттернов файлов и папок, которые будут игнорироваться системой контроля версий.

Теперь можно добавить все не игнорируемые фалы проекта в систему контроля версий командой git add -A и выполнить первый коммит git commit -m ‘init commit’. После проделанных шагов все файлы проекта буду проиндексированы и добавлены в систему контроля версий, а значит, будет проще следить за историей их изменений.

```
# IDE files #
#############
.idea/
.classpath
.project
*.ids
*.iml
*.ipr
*.iws
```

Рисунок 3.1.3 - Основные элементы файла .gitignore

```
rias@rias-Aspire-A715-71G:~/git/clone/dbunitool$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .gitignore
        README.md
        build.gradle
        gradle.properties
        gradle/
        gradlew
        gradlew.bat
        settings.gradle

nothing added to commit but untracked files present (use "git add" to track)
rias@rias-Aspire-A715-71G:~/git/clone/dbunitool$ git add -A
rias@rias-Aspire-A715-71G:~/git/clone/dbunitool$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   .gitignore
        new file:   README.md
        new file:   build.gradle
        new file:   gradle.properties
        new file:   gradle/wrapper/gradle-wrapper.jar
        new file:   gradle/wrapper/gradle-wrapper.properties
        new file:   gradlew
        new file:   gradlew.bat
        new file:   settings.gradle

rias@rias-Aspire-A715-71G:~/git/clone/dbunitool$ git commit -m 'initialize commit'
[master (root-commit) c6f3d96] initialize commit
 9 files changed, 340 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 README.md
 create mode 100644 build.gradle
 create mode 100644 gradle.properties
 create mode 100644 gradle/wrapper/gradle-wrapper.jar
 create mode 100644 gradle/wrapper/gradle-wrapper.properties
 create mode 100755 gradlew
 create mode 100644 gradlew.bat
 create mode 100644 settings.gradle
```

Рисунок 3.1.4 - Индексация файлов проекта и первый коммит

3.2 Создания пакетов архитектуры и подключение зависимостей

Создается основа архитектуры программного обеспечения. В пакете src/main/kotlin создаются пакеты, которые будут содержать в себе различные слои архитектуры.

– пакет «data» – основной пакет управления. Содержит в себе классы, управляющие доступом к источникам данных. Здесь хранятся классы

управления соединениями к базам данных, классы управления настройками и кэшем приложения;

– пакет «models» – содержит классы, описывающие представления данных в памяти приложения;

– пакет «util» – инфраструктурны пакет. Содержит в себе утильные методы, для решения общих задач, статические данные и специфические функции расширения;

– пакет «view» – пакет представления. Содержит в себе описание и работу с интерфейсом;

– пакет «start» – содержит классы для запуска приложения с различными настройками.

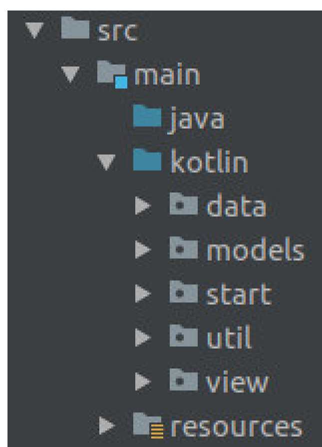

Рисунок 3.2.1 - Создание пакетов архитектуры

Для продолжения необходимо включить некоторые стороние библиотеки и фрейморки. На данном этапе, из mavencentral нужно подключить следующий список зависисимостей:

– org.jetbrains.kotlin:kotlin-stdlib-jdk8 – стандартная библиотека Kotlin для jdk 8 и выше. Включает в себя все стандартные классы и расширения языка;

– group: 'io.reactivex', name: 'rxjava', version: '1.3.8' – библиотека RxJava предоставляет функциональность реактивного программирования на java и jvm языках. Реактивное программирование – парадигма, ориентированная на потоки данных и распространение изменений. В основном, базируется на разлиных реализациях паттерна слушатель, но не ограничивается этим. Данная парадигма очень удобна для разработки пользовательских интерфейсов за счет автоматического распространения изменений. Библиотека RxJava предоставляет хорошее API для создания субъектов и зависимостей от них;

– group: 'org.jetbrains.kotlinx', name: 'kotlinx-coroutines-core', version: '1.2.0' – расширение стандартной библиотеки языка Kotlin, предоставляющее доступ к корутинам. Корутины – облегченные потоки, которые не привязаны к

нативным потокам и не требуют переключения контекста на процессор, следовательно, они быстрее обычных потоков;

– group: 'no.tornado', name: 'tornadofx', version: '1.7.18' – библиотека TornadoFX – предоставляет расширение функционала фреймворка JavaFX для языка Kotlin. Библиотека позволяет описывать пользовательский интерфейс на Kotlin в формате DSL, что делает код минималистичным и легко читаемым;

– group: 'com.google.code.gson', name: 'gson', version: '2.8.5' – библиотека Gson. Разработка компании Google для сериализации и десериализации объектов в формате JSON;

– group: 'commons-dbcp', name: 'commons-dbcp', version: '1.4' – библиотека Apache Software, для организации пула соединений к базам данных.

3.3 Разработка графического интерфейса

Первым этапом разработки идет создание графического пользовательского интерфейса по разработанному макету. Так как разработка интерфейсов ведется на фреймворке JavaFx, есть несколько способов описания GUI. Первый способ – описание через код. В этом способе все элементы графического интерфейса создаются и связываются прямо в классах, описывающих их. Данный способ позволяет создавать гибкие и динамические интерфейсы, но сопутствующий ему код очень быстро набирает объемы и становится трудно читаемым. Второй способ – описание структуры интерфейса в отдельных файлах на языке разметки FXML, который базируется на XML. Данный метод отделяет логику поведения интерфейса от его описания и повышает читабельность кода. Но данный способ применим только для описания статического, не меняющегося интерфейса. Для поддержания равновесия между гибкостью интерфейса и объемом кода в разработке применяются оба метода.

Следуя первому принципу SOLID [3], графический интерфейс разделяется на логические и функциональные части.

– Первым элементом графического интерфейса является контейнер всего рабочего пространство. Для реализации данного контейнера в папке ресурсов создается файл WorkWindow.fxml, который будет содержать разметку интерфейса, и создается класс WorkWindow – контроллер интерфейса. Класс WorkWindow наследуется от класса View библиотеки TornadoFX, тем самым получает доступ к возможностям библиотеки, таким как инъекция зависимостей, DSL описание структуры, привязка свойств. Связь контроллера и fxml файла происходит по именам класса и файла, они должны находиться в одном пакете. Контейнер рабочей поверхности основывается на классе BorderPane, пакета javafx.scene.layout. Данный класс позволяет разбить всю область на пять зон: верхняя, нижняя, левая, правая и центральная. Данное разделение хорошо подходит для макета, и легко добиться зонирования, показанного на рисунке 3.3.1.

Рисунок 3.3.1 - Зонирование интерфейса

– зона 1 — зона управления программой. В данной зоне будет выведено меню управления, в котором будет доступ к управлению всеми настройками и функционалом системы;

– зона 2 — зона управления соединениями. В данной зоне будет отображен список всех доступных соединений, а также функционал управления соединениями: создание удаление;

– зона 3 — рабочая зона. Содержит в себе основную рабочую поверхность и текстовый редактор, для создания запросов к СУБД. Управление запросами, сохранение и чтение из файлов. Так же, в данной секции отображаются результаты запросов;

– зона 4 — зона быстрого доступа. В данную зону будут выводиться доступы к часто вызываемым функциям;

– зона 5 — информационная зона. Здесь будет отображаться общая информация о системе.

Чтобы связать элементы представления с контроллером, необходимо прописать в каждом элементе, который необходимо получить в коде, прописать уникальное значение атрибута fx:id и создать в контроллере поле, тип которого соответствует типу элемента, а имя - значению атрибута fx:id.

```
<center>
    <AnchorPane fx:id="centerPane" prefHeight="200.0" prefWidth="200.0" BorderPane.alignment="CENTER"/>
</center>
<bottom>
    <AnchorPane fx:id="bottomPane" prefHeight="20.0" prefWidth="847.0" BorderPane.alignment="CENTER"/>
</bottom>
<right>
    <AnchorPane fx:id="rightPane" prefHeight="598.0" prefWidth="20.0" BorderPane.alignment="CENTER"/>
</right>
<left>
    <AnchorPane fx:id="leftPane" minWidth="300.0" prefWidth="300.0" BorderPane.alignment="CENTER"/>
</left>
```

Рисунок 3.3.2 - Элементы описание интрефейса в файле WorkWindow.fxml

```
private val leftPane: AnchorPane by fxid()
private val centerPane: AnchorPane by fxid()
private val bottomPane: AnchorPane by fxid()
private val rightPane: AnchorPane by fxid()
```

Рисунок 3.3.3 - Связывание элементов интерфейса и объектами в коде класса WorkWindow

Ключевое слово «by» является зависимым от контекста и используется для делегирования реализации аксессоров свойства. В данном случае происходит делегирование только методов получения свойства объекта, созданного на основе fxml документа.

Далее создаются представления и контроллеры элементов основных зон. Так как количество элементов меню и их расположение может часто меняться, и каждый элемент меню тесно связан со своей функцией, создавать разметку для этого элемента не продуктивно. Для создания меню лучше подойдет принцип DSL, что значительно упрощает чтение и анализ кода. Создаваемое меню состоит их двух пунктов: «File» и «Edit». Пункт «File» включает в себя функционал хранения рабочей среды, а именно методы «Save» и «Load». Метод «Save» сохраняет изменения открытого скрипта на диск, может быть вызван сочетанием клавиш Ctrl+S. Метод «Load», соответственно, загружает файл с диска и открывает его в новом рабочем пространстве, доступен сочетанием клавиш Ctrl+L. Оставшийся метод пункта «File» - вызов справки, функция «Help», доступна по сочетанию клавиш Ctrl+H.

Меню «Edit» отвечает за рабочую среду и предоставляет функции:

– «Preference» - открывает доступ к индивидуальным настройкам приложения;

– «New SQL pane» - создает новое пустое окно редактирования SQL скриптов.

```
override val root = menubar {
    menu("File") {
        item("Save", "Shortcut+S"){
            onAction = EventHandler {
                saveFile()
            }
        }
        item("Load", "Shortcut+L") {
            onAction = EventHandler {
                loadFile()
            }
        }
        separator()
        item("Help", "Shortcut+H"){
            onAction = EventHandler {
                showHelp()
            }
        }
    }
    menu("Edit") {
        item("Preference"){
            onAction = EventHandler {
                openSettings()
            }
        }
        item("New SQL pane") {
            onAction = EventHandler {
                ContextManager.newContext()
            }
        }
    }
}
```

Рисунок 3.3.4 - Описание создания меню с помощью принципа DSL

Для описания панели управления подключениями к базам данных создается пакет navigation. Описание структуры основного контейнера панели навигации находится в файле NavigationPane.fxml, описание его логики и связки структуры с вложенными компонентами — в одноименном классе, далее совокумность представления и контролера элемента будет называться компонентом. Под вложенными элементами компонента понимаются функциональные части, на которые было разбито представление навигации. Это представления панели инструментов для настроек подключений и списка соединений. Для этих элементов создаются компоненты NavigationToolbar и NavigationTree.

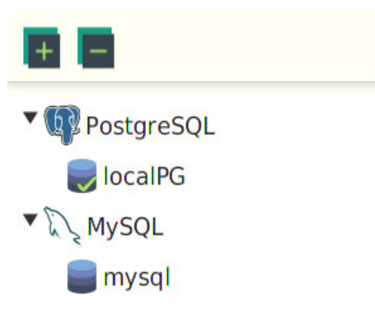

Рисунок 3.3.5 – Элемент навигации и управления соединениями

Так же, к данному функционалу относится окно создания нового соединения. Данное окошко разделено на три компонента: основной контейнер и логика создания соединения NewConnection. Когда открывается окошко создания нового соединение, первым делом, пользователю

отображается список доступных типов СУБД, за этот функционал отвечает компонент DBMSList.

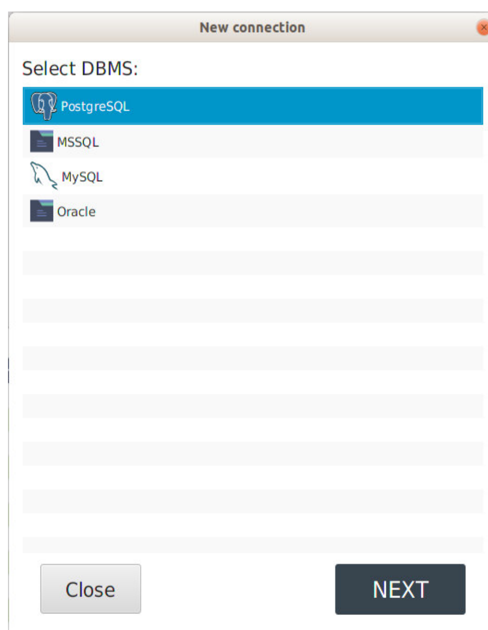

Рисунок 3.3.6 – Выбор типа СУБД для нового соединения

Далее, предлагается указать параметры подключения к СУБД - компонент ConnectionProperties. Общение между внутренними компонентами окна производится через родительский компонент NewConnection, который организует их совместную работу и управляет их результатами, сами же дочерние компоненты отвечают только за предоставление и получение данных и их валидацию.

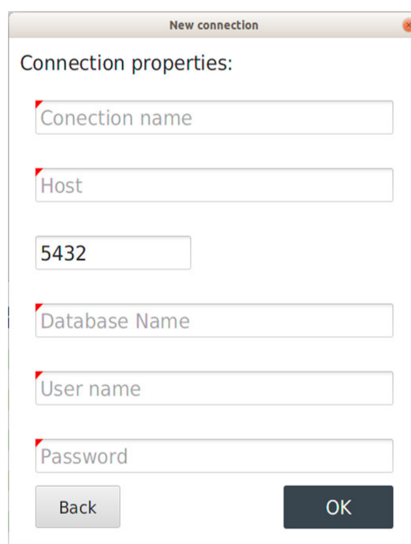

Рисунок 3.3.7 – Указание параметров нового соединения

Представление главной рабочей области системы описывается в пакете work_pane. Эта панель представляет из себя панель рабочих вкладок. Каждая вкладка работает независимо от других, что удобно для параллельной работы над несколькими задачами. Описание данной панели и управление вкладками описывается в компоненте WorkPaneLayout.

Рисунок 3.3.8 - Вкладки рабочего пространства

Внутри каждой вкладки содержатся компонент, управляющий работой с SQL запросами. Компонент SqlPanel содержит панель управления запросами, в которой есть функционал запуска запроса, сохранение запроса и чтение из файла, а также выбор соединения, в котором будет выполняться запрос.

Рисунок 3.3.9 - Панель управления запросами

Следующим элементом компонента является редактор SQL запросов. В данном элементе реализован функционал подсветки ключевых слов, выполнения запроса при определенном сочетании клавиш, поиск и замена по тексту. А так же, есть нумерация строк.

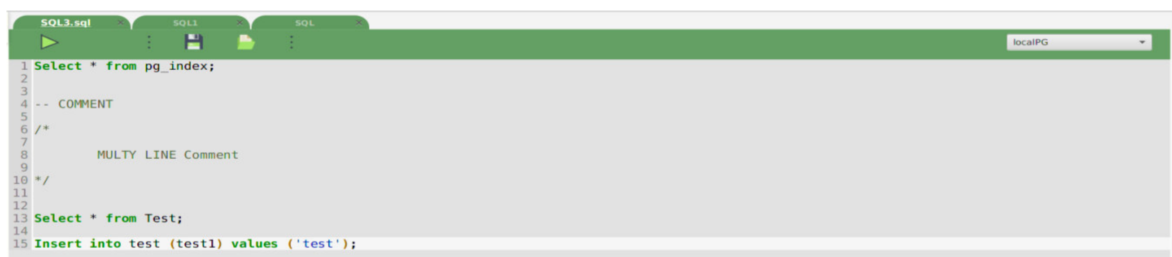

Рисунок 3.3.10 – Панель редактирования скриптов

Выполнение запроса доступно в нескольких видах: полное выполнение всего содержащегося на панели или выполнение только выделенной части. Приоритет всегда имеет выделение, то есть, если в панели есть выделение, будет выполняться только оно.

Рисунок 3.3.11 – Выполнение запроса

При выполнении запросов результат каждого выводится отдельно в таблице. За функционал вывода результатов отвечает компонент OutPane, который создает вкладку для каждого запроса и, в зависимости от текущего состояния запроса отображает соответствующую информацию. За отображение информации о запросе отвечает компонент ResultTable. Во время выполнения запроса, данный компонент показывает анимацию о загрузке, время начала выполнения запроса и возможность остановить выполнение запроса. После выполнения запроса, пользователю отображается либо таблица с результатом в случае удачного выполнения, либо сообщение об ошибке, если выполнение запроса закончилось неудачно.

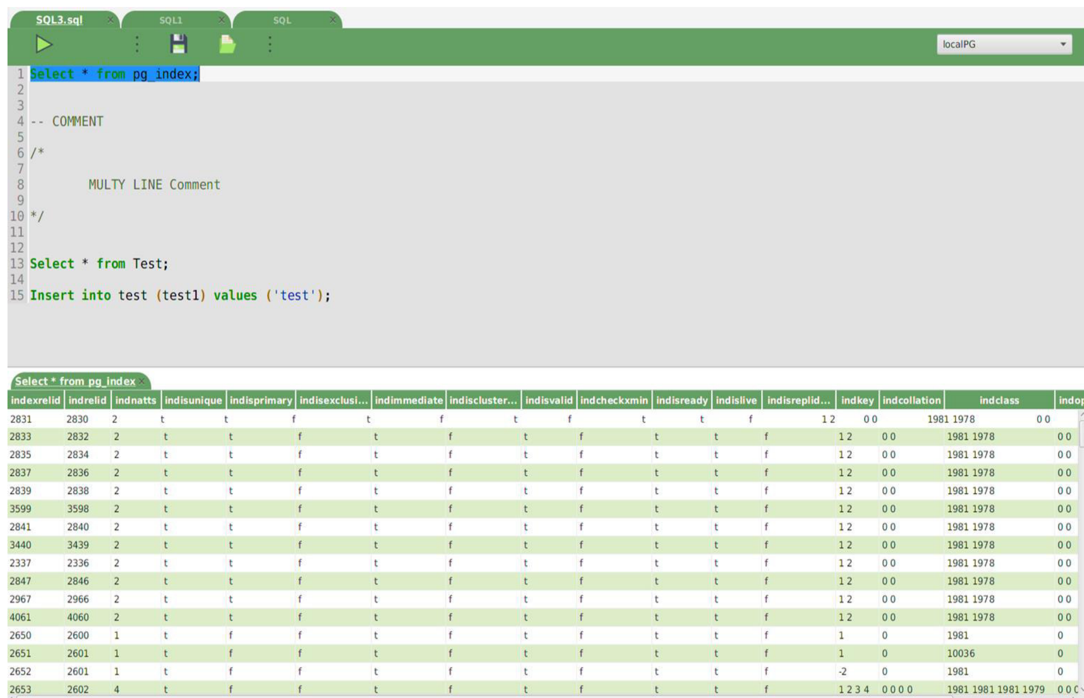

Рисунок 3.3.12 – Результат успешно выполненного запроса

Рисунок 3.3.13 – В результате выполнения запроса произошла ошибка

3.4 Описание пакетов инфраструктуры.

Пакет «util» содержит в себе скриптовые файлы языка Kotlin. Данные файлы предоставляют утильные функции, специализированные функции расширения и статические переменные, которые могут использоваться независимо от контекста и в любом из уровней архитектуры.

Структура пакета «util»:

– Application.kt — содержит в себе константы, которые относятся непосредственно к приложению. Среди них название приложения, расположения папок для настроек и кэша приложения.

– DefaultDb.kt — содержит описание существующих в системе по умолчанию типов СУБД.

– DialogUtil.kt — содержит кастомные переменные для настроек диалоговых окон JavaFX.

– FileUtil.kt — содержит расширяющие функции работы с файлами, такие как чтение и сохранение файла в формате json.

– JsonUtil.kt — вспомогательные функции для сериализации и десериализации объектов в формате json.

– SettingsUtil.kt — методы для создания файлов настроек и кэша.

– ValidatorUtil.kt — функции, расширяющие класс библиотеки ValidatorContext для добаяления кастомных проверок пользовательского ввода.

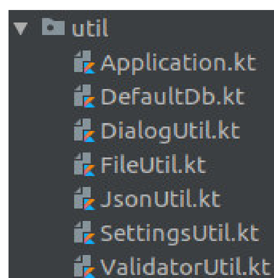

Рисунок 3.4.1 – Пакет «util»

```
class StartWorkWindow : App(WorkWindow::class) {
    override fun start(stage: Stage) {
        stage.title = "DbUniToll"
        stage.icons += Image(resources["/images/main.png"])
        stage.isMaximized = true
        super.start(stage)
    }
}
```

Рисунок 3.4.1 – Класс запуска приложения

Описание пакета «start».

Пакет содержит класс StartWorkWindow, в котором выполняется запуск и настройка приложения, среди которых установка названия, картинки и запуск в режиме на весь экран.

3.5 Описание моделей данных

Пакет «model» содержит в себе классы, описывающие предметную область в памяти приложения.

Многие данные программы, такие как информация о созданных подключениях или контекст работы необходимо хранить не только во временной памяти, но и сохранять их на диск для повторного использования после перезапуска приложения. Для этого были создана архитектура объектов, кэшируемых в постоянной памяти. Для всех объектов, которые необходимо хранить в памяти был реализован интерфейс Saved. Данный интерфейс описывает функционал объекта, который может быть сохранен на диск. По умолчанию, интерфейс сохраняет объекты в формате json но делегирует наследникам выбор директории и названия файла для сохранения. Так же, интерфейс обязывает потомков определить для объекта значение по умолчанию, которое будет возвращено в случае ошибки чтения данных, это необходимо для обеспечения стабильной работы приложения в случае потери файлов кэша.

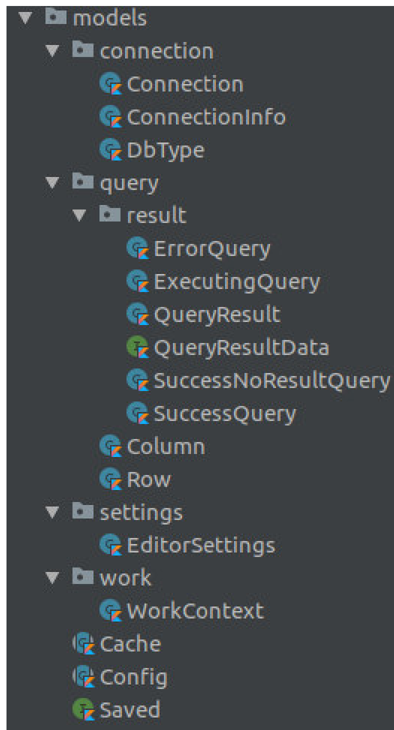

Рисунок 3.5.1 – Структура пакета «model»

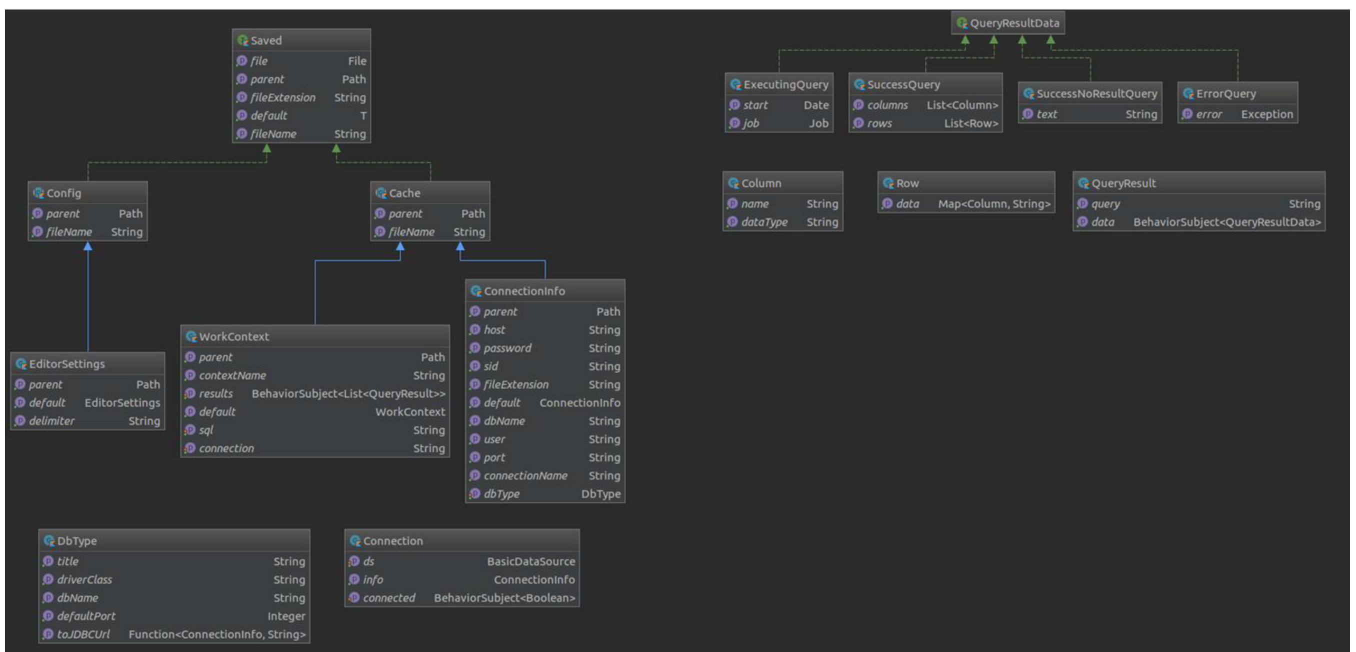

Рисунок 3.5.2 – Диаграмма классов пакета «model»

Многие данные программы, такие как информация о созданных подключениях или контекст работы необходимо хранить не только во временной памяти, но и сохранять их на диск для повторного использования после перезапуска приложения. Для этого были создана архитектура объектов, кэшируемых в постоянной памяти. Для всех объектов, которые необходимо хранить в памяти был реализован интерфейс Saved. Данный

интерфейс описывает функционал объекта, который может быть сохранен на диск. По умолчанию, интерфейс сохраняет объекты в формате json но делегирует наследникам выбор директории и названия файла для сохранения. Так же, интерфейс обязывает потомков определить для объекта значение по умолчанию, которое будет возвращено в случае ошибки чтения данных, это необходимо для обеспечения стабильной работы приложения в случае потери файлов кэша.

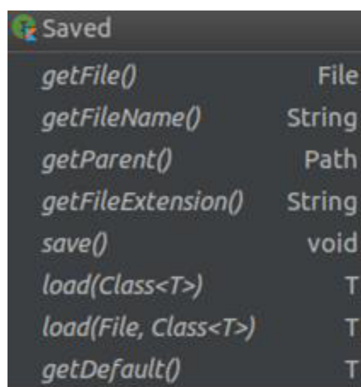

Рисунок 3.5.3 – Класс Saved

Классы Cache и Config — два абстрактных класса, реализующие интерфейс Saved, разделяющие сохраняемые объекты программы на две группы: кэш — данные о соединениях, рабочий контекст и настройки приложения соответственно. Данные классы не меняют стандартной реализации интерфейса Saved, кроме определения директории, для сохранения объектов и добавления конструктора, в который необходимо передать название для файла.

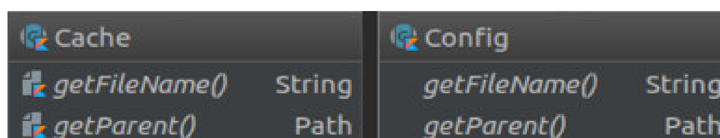

Рисунок 3.5.4 – Абстрактные классы Cache и Config

В пакете «connection» лежат классы, представляющие модели соединения с СУБД.

Класс DbType — описывает тип СУБД, к которому выполняется подключение. Так же, класс предоставляет функциональность генерации URL, на основе данных о соединении, получении изображения СУБД, если такое имеется в ресурсах.

Методы формата componentN() на диаграмме класса — функции деструктурирующего присваивания, генерируемые автоматически языком Kotlin. Данные методы используются для удобного разбития объекта на переменные.

```
DbType
image()                                                          URL
equals(Object)                                                   boolean
hashCode()                                                       int
toString()                                                       String
component1()                                                     String
component2()                                                     String
component3()                                                     String
component4()                                                     Integer
component5()                                    Function<ConnectionInfo, String>
copy(String, String, String, Integer, Function<ConnectionInfo, String>)   DbType
title                                                            String
driverClass                                                      String
dbName                                                           String
defaultPort                                                      Integer
toJDBCUrl                                       Function<ConnectionInfo, String>
```

Рисунок 3.5.5 – Класс DbType

```kotlin
val POSTGRE_SQL: DbType =
    DbType("postgresql", "PostgreSQL", "org.postgresql.Driver", 5432)
fun main() {
    val (name, title) = POSTGRE_SQL
    println("$name -|- $title")// output: postgresql -|- PostgreSQL
}
```

Рисунок 3.5.6 – Деструктурирующее присваивание

Класс ConnectionInfo содержит в себе данные об определенном соединении с СУБД, такие как хост, порт, название базы данных, имя и пароль пользователя. Так же, в классе содержится пользовательское название соединения и тип СУБД. Данный класс является наследником абстрактного класса Cache, а значит, он может быть сохранен на диск.

Класс Connection, описывает определенное соединение к СУБД в памяти приложения. Содержит в себе информацию о подключении к СУБД, источник данных о текущем состоянии данного подключения и пул соединений. Под источником информации понимается объект, на изменение данных которого можно выполнить подписку и получать о них «уведомления». Так как в данном классе хранится пул соединений, он предоставляет функционал создания нового соединения к СУБД.

В пакете «query» содержатся классы, описывающие результаты запроса к СУБД. Основной класс — QueryResult. Данный класс хранит в себе запрос и источник данных о текущем состоянии данного запроса. Состояние запроса описывается интерфейсом QueryResultData и его реализациями.

ConnectionInfo	
parent	Path
host	String
password	String
sid	String
fileExtension	String
default	ConnectionInfo
dbName	String
user	String
port	String
connectionName	String
dbType	DbType

Рисунок 3.5.7 – Класс ConnectionInfo

Connection	
isConnected()	BehaviorSubject<Boolean>
connect()	void
ds	BasicDataSource
info	ConnectionInfo
connected	BehaviorSubject<Boolean>

Рисунок 3.5.8 – Класс Connection

Класс ExecutingQuery — реализация интерфейса QueryResultData. Данная реализация описывает исполняющийся запрос, предоставляя информацию о времени начала выполнения запроса и объект типа Job, который представляет из себя фоновую задачу, выполняющую запрос.

Класс ErrorQuery — реализация интерфейса QueryResultData. Применяется при неудачном выполнении запроса и содержит в себе информацию об ошибке.

Класс SuccessNoResultQuery – реализация интерфейса QueryResultData. Содержит в себе сообщение об успешном завершении запроса. Применяется, когда запрос не возвращает результат, например при выполнении обновления записи.

Класс SuccessQuery – реализация интерфейса QueryResultData. Используется, когда успешный запрос возвращает данные. Содержит в себе список колонок и строи которые вернул запрос.

Класс Column — описывает одну колонку, результата запроса.

Класс Row — описывает строку результата запроса.

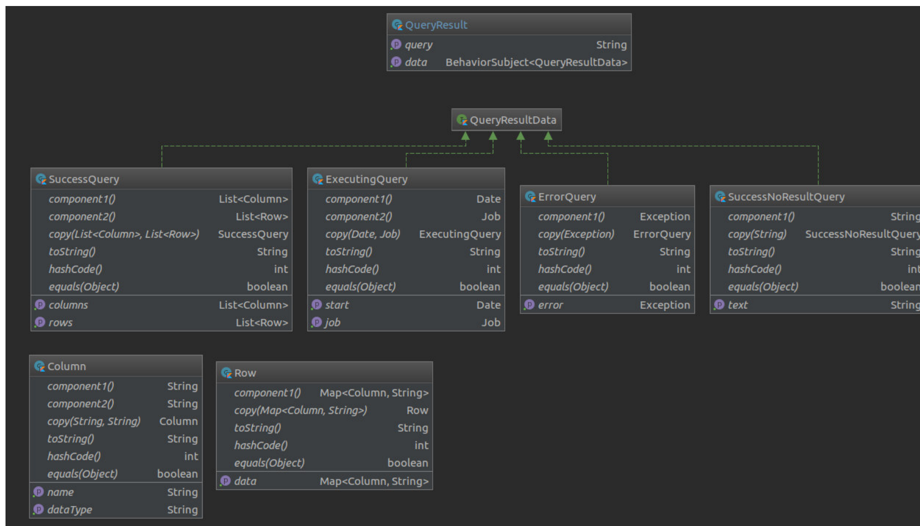

Рисунок 3.5.9 – Диаграмма классов пакета «query»

Пакет «settings» хранит в себе объекты, описывающие настройки программы. Все классы, содержащиеся в данном пакете, реализуют абстрактный класс Config и сохраняются в постоянную память.

Пакет «work» хранит объекты для представления рабочего контекста программы. В данном пакете содержится класс WorkContext, который отвечает за хранение данных о рабочей вкладке пользователя. Для каждой вкладки создается свой WorkContext. Класс содержит в себе содержимое панели редактирования кода и название выбранного подключения к СУБД. Так же, в классе находится источник данных для результатов последних запросов. Так как WorkContext реализует Cache, все данные, кроме результатов запросов, сохраняются на диск.

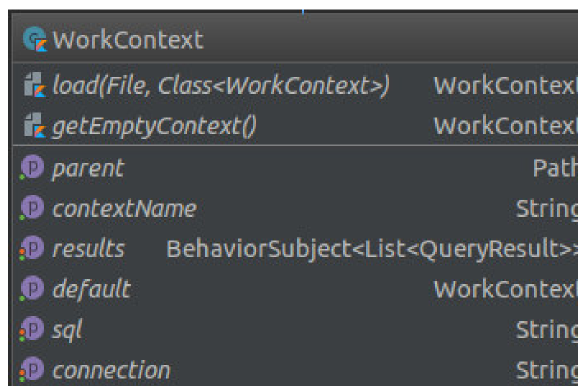

Рисунок 3.5.10 – Класс WorkContext

3.6 Описание слоя доступа к данным

Пакет «data» содержит в себе объекты управления данными. Объекты в Kotlin — одиночки, встроенные в функциональность языка. То есть это классы, которые будут созданы в единственном экземпляре.

Объект ContextManaget выполняет управление всеми рабочими контекстами. Объект хранит в себе источник данных всех контекстов, который при старте приложения считывает сохраненные контексты с диска. Менеджер предоставляет функции добавления, удаления и создание нового контекста. При этом, после любого изменения списка контекстов, все, кто используют его, получат обновленные данные.

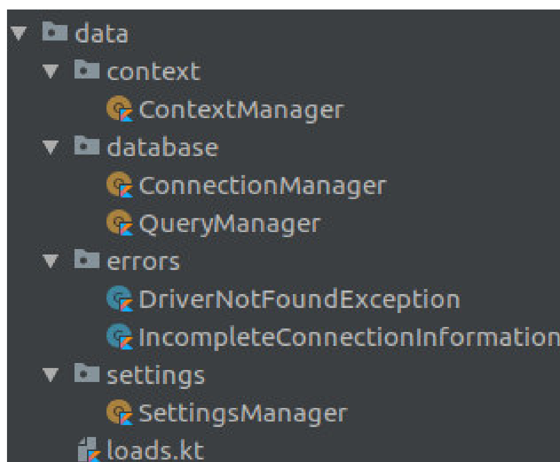

Рисунок 3.6.1 – Структура пакета «data»

ContextManager	
INSTANCE	ContextManager
addContext(WorkContext)	void
removeContext(WorkContext)	void
newContext(String, String)	WorkContext
contexts	BehaviorSubject<List<WorkContext>>

Рисунок 3.6.2 – Класс ContextManager

Объект ConnectionManager обеспечивает аналогичный предыдущему методу функционал, но работающий с соединениями к базам данных. Объект так же имеет источник данных, в котором находятся информация о всех созданных подключениях. Предоставляемый функционал: добавление, удаление соединений, валидация имен для новых соединений и метод загрузки драйвера для определенного типа СУБД и проверки подключения.

ConnectionManager	
INSTANCE	ConnectionManager
addConnection(ConnectionInfo)	void
removeConnection(ConnectionInfo)	void
checkDriverAndConnect(Connection)	void
validateConnectionName(String)	boolean
connections	BehaviorSubject<List<Connection>>

Рисунок 3.6.3 – Класс ContextManager

Объект QueryManager отвечает за выполнение запросов. Объект предоставляет функционал выполнения запросов в выбранном соединении.

Объект SettingsManager отвечает за управление настройками. Выполняет загрузку настроек с диска или создание значений по умолчанию, сохранение настроек на диск при их изменении.

QueryManager	
INSTANCE	QueryManager
executeQuery(WorkContext, String)	void

Рисунок 3.6.4 – Класс QueryManager

Пакет «errors» содержит пользовательские реализации класса Exception для генерации их в исключительных ситуациях.

4 Экономическая часть

4.1 Трудоёмкость разработки программного продукта

Реализация программного продукта включает в себя следующие этапы:

- анализ требований к проекту;
- проектирование;
- реализация программы;
- тестирование;
- документирование.

Оценка трудоёмкости и описание работ представлены в таблице 4.1.

Таблица 4.1 - Распределение работ по этапам и видам и оценка их трудоемкости

Этап разработки ПП	Вид работы на данном этапе	Трудоемкость разработки ПП, чел.× ч.
Анализ требований к проекту	Сбор и обработка требований; Формулировка целей и задач; Выделение базовых сущностей; Определение этапов, сроков и стоимости разработки.	32
Проектирование	Объектная декомпозиция; Построение моделей системы; Выбор аппаратной платформы; Получение технических заданий и разработка спецификаций.	48
Реализация программы	Создание дизайна – разработка стиля, прототипирование, реализация интерфейса; Написание программного кода на выбранной платформе.	332

Продолжение таблицы 4.1

Этап разработки ПП	Вид работы на данном этапе	Трудоемкость разработки ПП, чел.× ч.
Тестирование	Проверка корректности работы ПО; Проверка соответствия требованиям.	80
Документирование	Оформление инструкций пользователя и пояснительной записки.	24
ИТОГО		516

4.2 Расчет затрат на разработку ПП

Расчет затрат на разработку программного продукта производится по следующим пунктам:

– материальные;
– амортизационные отчисления;
– оплата труда;
– социальный налог;
– прочее.

Материальные затраты – затраты на все материалы и комплектующие, используемые во время разработки программного продукта. Расчет данной формы затрат производится в таблице 4.2. Общая сумма затрат этой категории определяется по формуле 4.1:

$$З_{м} = \sum_{i}^{n} З_{м_i}\,;\; З_{м_i} = P_i * C_i, \tag{4.1}$$

где $З_м$ – общая сумма материальных затрат;
$З_{мi}$ – сумма затрат i-го ресурса;
i – вид ресурса;
n – общее количество ресурсов;
P_i – расход i-го ресурса;
C_i - стоимость единицы i-го ресурса.

Таким образом:

$$З_{м} = 32 + 48 + 332 + 80 + 24 = 516\ (\text{чел} \times \text{ч}). \tag{4.2}$$

Таблица 4.2 – Затраты на материальные ресурсы

Наименование материального ресурса	Единица измерения	Кол-во израсходованного материала	Цена за единицу, тг	Сумма, тг
Бумага	Блок	2	900	1800
Батарейки AAA 1.5 V	Шт	12	305	3660
ИТОГО				5460

Так как при разработке используется электрооборудование, необходим расчет затрат электроэнергии.

Сумма затрат на электроэнергию для каждого оборудования рассчитывается по формуле 4.3:

$$З_э = M * K * T * C, \quad (4.3)$$

где $З_э$ – затраты на электроэнергию оборудования;

М – мощность оборудования;

К – коэффициент использования мощности;

Т – время работы оборудования чч;

С – цена электроэнергии тг/кВт*ч (на 2019г. для юридических лиц в г. Алматы составляет 26,71тг/кВт*ч).

$$З_{э1} = 0{,}135 * 0{,}8 * 516 * 26{,}71 = 1488{,}5\ (тг). \quad (4.4)$$

$$З_{э2} = 0{,}009 * 0{,}8 * 516 * 26{,}71 = 99{,}233\ (тг). \quad (4.5)$$

Общая сумма затрат на электроэнергию высчитывается по формуле 4.6:

$$З_{э_о} = \sum_i^n З_{э_i}. \quad (4.6)$$

$$З_{э_о} = 1488{,}5 + 99{,}233 = 1587{,}733\ (тг), \quad (4.7)$$

где $З_{э_о}$- итоговая сумма затрат на электроэнергию;

i - номер оборудования;

n – количество оборудования;

$З_{э_i}$- сумма затрат на электричество i-го оборудования.

Таблица 4.3 – Затраты на электроэнергию

Наименование оборудования	Паспортная мощность, кВт	Коэффициент использования мощности	Время работы оборудования для разработки ПП, ч	Цена электроэнергии, тг/(кВт*ч)	Сумма, тг
Ноутбук	0,135	0,8	516	26,71	1488,5
Модем	0,009	0,8	516		99,233
ИТОГО					1587,733

Амортизационные отчисления строятся на основе стоимости основного оборудования и программного обеспечения, задействованных в разработке продукта.

Амортизационная стоимость оборудования рассчитывается по формуле 4.8:

$$З_{ам} = \frac{Ф * Н_А * Т_{нир}}{100 * Т_{эф}}, \qquad (4.8)$$

где $З_{ам}$ – сумма амортизационных отчислений фонда;

Ф – стоимость фонда;

$Т_{нир}$ – время работы фонда за период разработки продукта;

$Т_{эф}$ – эффективное время работы фонда за год;

$Н_А$ – годовая норма амортизации.

Годовая норма амортизации вычисляется по формуле 4.9:

$$Н_А = \frac{100}{Т_N}, \qquad (4.9)$$

где $Т_N$ – возможный срок использования ОФ.

Для персонального компьютера (ноутбука) возможный срок использования ОФ нужно выбрать равным 4 годам, тогда

$$Н_А = \frac{100}{4} = 25\ (\%). \qquad (4.10)$$

$$З_{ам} = \frac{260000 * 25 * 516}{100 * 3024} = 11091\ (тг). \qquad (4.11)$$

Общая сумма амортизационных отчислений определяется по формуле 4.12:

$$З_{АМ_о} = \sum_i^n З_{АМ_i}, \quad (4.12)$$

где $З_{АМ_о}$ – общая сумма амортизации;

i – номер ОФ;

n – общее количество ОФ;

$З_{АМ_о}$- сумма амортизации i-го ОФ.

Таблица 4.4 – Амортизация основных фондов (ОФ)

Наименование оборудования и ПО	Стоимость оборудования и ПО, тг	Годовая норма амортизации, %	Эффективный фонд времени работы оборудования и ПО, ч/год	Время работы оборудования и ПО для разработки ПП, ч	Сумма, тг
Ноутбук Acer Aspire 7 A715-71G-56BD	260000	25	3024	516	11091
ИТОГО					11091

Оплата труда включает в себя расходы на оплату труда сотрудников, участвующих в разработке программного продукта.

Сумма оплаты труда работника высчитывается по формуле 4.13:

$$З_{ст} = ЧС * Т, \quad (4.13)$$

где $З_{ст}$ – сумма оплаты труда сотрудника;

ЧС – часовая ставка сотрудника;

Т – трудоемкость разработки ПП.

Часовая ставка работника рассчитывается по формуле 4.14:

$$ЧС = \frac{ЗП}{ФРВ}, \quad (4.14)$$

где ЗП – месячная заработная плата сотрудника;

ФРВ – количество рабочих часов сотрудника в месяц.

Средняя месячная заработная плата программиста в городе Алматы составляет 185000 тг.

$$ЧС = \frac{185000}{21*8} = 1100 \text{ (тг)}. \quad (4.15)$$

$$З_{ст} = 1100 * 516 = 567600 \text{ (тг)}. \quad (4.16)$$

Общая сумма заработной платы высчитывается по формуле 4.17:

$$З_{ЗП} = \sum_{i}^{n} З_{СТ_i}, \quad (4.17)$$

где $З_{ЗП}$ – общая сумма затрат на заработную плату;
i – номер сотрудника;
n – общее количество сотрудников;
$З_{СТ_i}$ – сумма заработной платы i-го сотрудника.

Таблица 4.5 – Затраты на оплату труда

Категория работника	Квалификация	Трудоемкость разработки ПП, чел.×ч	Часовая ставка, тг/ч	Сумма, тг
Сотрудник	Программист	516	1100	567600
ИТОГО				567600

Социальный налог рассчитывается как 9,5% от затрат на оплату труда после вычета пенсионных отчислений. Пенсионные отчисления составляют 10% от полных затрат на заработную плату.

Таким образом, пенсионные отчисления рассчитываются по формуле 4.18:

$$З_{п} = \frac{З_{ЗП} * 10}{100}, \quad (4.18)$$

где $З_{П}$ – сумма пенсионных отчислений;
$З_{ЗП}$ – общая сумма затрат на заработную плату.

$$З_{П} = \frac{567600 * 10}{100} = 56760 \text{ (тг)}. \quad (4.19)$$

Социальный налог вычисляется по формуле 4.20:

$$З_{соц} = \frac{(З_{ЗП} - З_{П}) * 9{,}5}{100}. \quad (4.20)$$

Тогда:

$$З_{соц} = \frac{(567600 - 56760) * 9{,}5}{100} = 48529 \text{ (тг)}. \quad (4.21)$$

Итого затраты на пенсионные отчисления составляют 56760 тг., а затраты на социальный налог – 48529 тг.

Прочие затраты включают в себя затраты на арендную плату с коммунальными услугами и прочие хозяйственные расходы. Смета прочих расходов представлена в таблице 4.6:

Таблица 4.6 – Прочие затраты

Наименование ресурса	Единица измерения	Кол-во	Цена за единицу, тг	Сумма, тг
Коворкинг	час	516	600	309600
Программное обеспечение IntellijIDEA	Копия	1	189,000	189000
ИТОГО затраты на прочее				498600

На основе проделанных вычислений составляется общая смета затрат на реализацию программного продукта. Результирующая смета представлена в таблице 4.7:

Таблица 4.7 – Смета затрат на разработку ПП

Статьи затрат	Сумма, тг
1. Материальные затраты, в том числе: - материалы - электроэнергия	7047,733
2. Затраты на оплату труда.	567600
3. Отчисления на социальные нужды.	48529
4. Амортизация основных фондов.	11091
5. Прочие затраты.	498600
ИТОГО по смете	1132867,733

4.3 Определение возможной (договорной) цены ПП

Договорная цена прикладного программного обеспечения вычисляется по формуле 4.22:

$$Ц_{д} = З_{НИР} * \left(1 + \frac{Р}{100}\right), \quad (4.22)$$

где $Ц_д$ – договорная цена, тг;

$З_{нир}$ – затраты на разработку программы, тг;

Р – средний уровень рентабельности ПП, %.

Таким образом, получается:

$$Ц_д = 1132867,733 * \left(1 + \frac{26}{100}\right) = 1427413,34 \text{ (тг).} \quad (4.23)$$

Цена реализации высчитывается с учетом НДС, который равен 12% (формула 4.24):

$$Ц_р = Ц_д + Ц_д * НДС. \quad (4.24)$$

Следовательно:

$$Ц_р = 1427413,34 + 1427413,34 * 0,12 = 1598702,94 \text{ (тг).} \quad (4.25)$$

Таким образом, можно рассчитать возможную прибыль от реализации программного продукта по формуле 4.26:

$$П = Ц_р - З_{НИР}. \quad (4.26)$$

Подставив значения, получим:

$$П = 1598702,94 - 1427413,34 = 171289 \text{ (тг).} \quad (4.27)$$

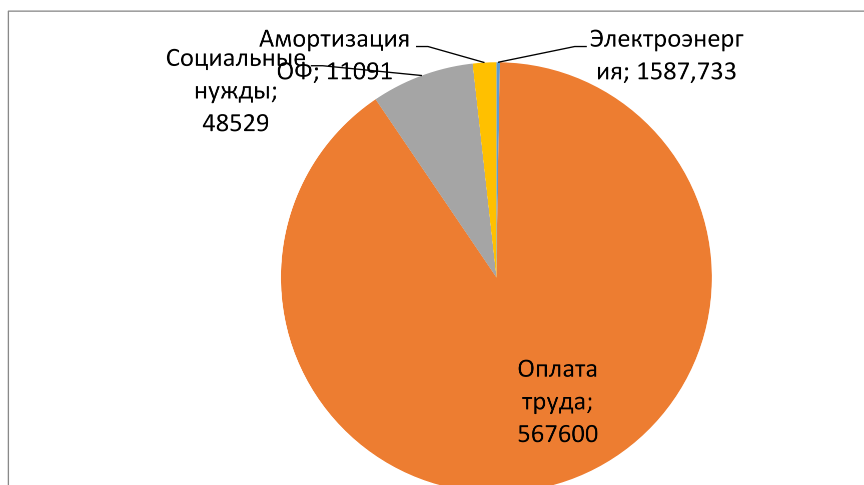

Рисунок 4.1 - Диаграмма затрат на разработку ПП

5 Охрана труда и безопасность жизнедеятельности

Разрабатываемое программное обеспечение предназначено для системных администраторов, администраторов баз данных и программистов. Продукт обеспечивает легкий доступ к выполнению запросов в различных базах данных и их администрировании.

Так как данный продукт будет использоваться в офисных помещениях, необходимо рассмотреть условия труда в офисе и, при необходимости, рассчитать необходимые мероприятия по обеспечению безопасности жизнедеятельности и охране труда.

Для тестирования и внедрения продукта предоставлен офис разработчика прикладных приложений. Площадь помещения: $S = 30$ м^2(6x5), высота потолков – 3 метра. Источники шума в помещении отсутствуют, а контроль над микроклиматом осуществляется с помощью многофункционального кондиционера, что позволяет постоянно поддерживать комфортную температуру и влажность среды. В комнате имеется окно, площадью 2 м^2. Искусственное освещение обеспечивается 8 светодиодными лампами СТ-15. Рабочее место сотрудника не предоставляет необходимого уровня комфорта: отсутствует возможность регулирования высоты кресла и монитора, из-за чего монитор оказывается сильно ниже глаз сотрудника и вызывает дискомфорт при продолжительном времени работы, под столом недостаточно мест, для комфортного расположения ног сотрудника.

Таким образом, на основании приведенного выше анализа можно заключить, что предоставленный офис нуждается в проверке достаточности освещенности и качества эргономики рабочего места, для обеспечения комфортной и безопасной работы.

Работа за персональным компьютером требует значительного умственного сосредоточения, высокой напряженности зрительной работы. Для поддержания эффективности и работоспособности большое значение имеет эргономичность рабочего места и отсутствие внешних раздражителей. Так же, важно правильное сочетание труда и отдыха.

Рабочее место за компьютером требует серьезного подхода к его эргономическому проектированию. Взаимное расположение всех предметов и поза сотрудника должны соответствовать его физическим и психологическим требованиям. Например, при организации рабочего места необходимо учесть такие условия, как: оптимальное размещение технического оборудования, при котором все элементы не ограничивают пространство и находятся на комфортном расстоянии, достаточное пространство рабочего места для возможности свободного перемещения и расположения дополнительных элементов.

К важным составляющим эргономики рабочего места, можно отнести следующие аспекты: высота поверхности стола и высота кресла, если такой имеется, пространство для ног, место для хранения документов и работы с

ними, характеристики рабочего кресла, поверхность рабочего стола, возможность регулировать характеристики рабочего места (изменять высоту кресла, наклон монитора и т.д.).

Главными элементами рабочего места программиста являются стол и кресло. Основное время работы, сотрудник проводит в положении сидя, в том случае, если сотрудник не использует стол для работы стоя. Рабочая поза сидя вызывает минимальное утомление программиста, рабочая поза стоя позволяет снизить напряжение на мышцы кора и положительно влияет на кровообращение. Рациональная планировка рабочего места предусматривает четкий порядок и постоянство размещения предметов, инструментов и документации. То, что требуется для выполнения работ чаще, расположено в зоне легкой досягаемости рабочего пространства. Моторное поле - пространство рабочего места, в котором могут осуществляться двигательные действия человека. Максимальная зона досягаемости рук — это часть моторного поля рабочего места, ограниченного дугами, описываемыми максимально вытянутыми руками при движении их в плечевом суставе. Оптимальная зона - часть моторного поля рабочего места, ограниченного дугами, описываемыми предплечьями при движении в локтевых суставах с опорой в точке локтя и с относительно неподвижным плечом.

Для рабочего стола можно выделить следующие характеристики:

– высота стола должна обеспечивать возможность свободного расположения за ним, в удобной позе;

– нижняя часть стола не должна сковывать и ограничивать сотрудника в удобном расположении ног;

– поверхность стола должна обладать свойствами, исключающими появление бликов в поле зрения программиста;

– конструкция стола должна предусматривать наличие выдвижных ящиков (не менее 3 для хранения документации, листингов, канцелярских принадлежностей).

– высота рабочей поверхности рекомендуется в пределах 680-760мм. Высота поверхности, на которую устанавливается клавиатура, должна быть около 650мм.

Большое значение придается характеристикам рабочего кресла. Так, рекомендуемая высота сиденья над уровнем пола находится в пределах 420-550мм. Поверхность сиденья мягкая, передний край закругленный, а угол наклона спинки - регулируемый.

Необходимо предусматривать при проектировании возможность различного размещения документов: сбоку от монитора, между монитором и клавиатурой и т. п.

Положение экрана определяется:

– расстоянием считывания (0,6…0,7м);

– углом считывания, направлением взгляда на 20 (ниже горизонтали к центру экрана, причем экран перпендикулярен этому направлению.

Должна также предусматриваться возможность регулирования экрана:

– по высоте +3 см;
– по наклону от -10º до +20 º относительно вертикали;
– в левом и правом направлениях.

Большое значение также придается правильной рабочей позе пользователя. При неудобной рабочей позе могут появиться боли в мышцах, суставах и сухожилиях. Требования к рабочей позе пользователя видеотерминала следующие:

– голова не должна быть наклонена более чем на 20 º;
– плечи должны быть расслаблены;
– локти - под углом 80 º - 100 º;
– предплечья и кисти рук - в горизонтальном положении.

Причина неправильной позы пользователей обусловлена следующими факторами: нет хорошей подставки для документов, клавиатура находится слишком высоко, а документы - низко, некуда положить руки и кисти, недостаточно пространство для ног. В целях преодоления указанных недостатков даются общие рекомендации: лучше передвижная клавиатура; должны быть предусмотрены специальные приспособления для регулирования высоты стола, клавиатуры и экрана, а также подставка для рук.

Существенное значение для производительной и качественной работы на компьютере имеют размеры знаков, плотность их размещения, контраст и соотношение яркостей символов и фона экрана. Если расстояние от глаз оператора до экрана дисплея составляет 60…80 см, то высота знака должна быть не менее 3мм, оптимальное соотношение ширины и высоты знака составляет 3:4, а расстояние между знаками – 15…20% их высоты. Соотношение яркости фона экрана и символов - от 1:2 до 1:15.

Во время пользования компьютером медики советуют устанавливать монитор на расстоянии 50-60 см от глаз. Специалисты также считают, что верхняя часть видеодисплея должна быть на уровне глаз или чуть ниже. Когда человек смотрит прямо перед собой, его глаза открываются шире, чем когда он смотрит вниз. За счет этого площадь обзора значительно увеличивается, вызывая обезвоживание глаз. К тому же если экран установлен высоко, а глаза широко открыты, нарушается функция моргания. Это значит, что глаза не закрываются полностью, не омываются слезной жидкостью, не получают достаточного увлажнения, что приводит к их быстрой утомляемости.

Создание благоприятных условий труда и правильное эстетическое оформление рабочих мест на производстве имеет большое значение как для облегчения труда, так и для повышения его привлекательности, положительно влияющей на производительность труда.

Правильно выбранное освещение рабочего места уменьшает нагрузку на глаза, снижает их утомляемость, что положительно влияет на качество труда. Недостаточность освещения приводит к высокому контрасту монитора и

окружающей среды, что вызывает сильную утомляемость глаз, ослабляет внимание. Чрезмерное освещение может вызвать ослепление и резь в глазах, вызывая сильный дискомфорт. Резкие тени и блики на рабочей поверхности могут вызвать дезориентацию работника. Все вышеперечисленные причины могут снизить качество труда или привести к заболеванию сотрудника, поэтому важно правильно подобрать освещение.

Возможно использование трех видов освещения – естественное, искусственное или совмещенное.

Освещение помещения дневным светом через световые проемы называется естественным освещением.

Искусственное освещение применяется в случаях, когда невозможно обеспечить достаточного естественного освещения.

В помещениях, где проводится работа с вычислительными машинами необходимо обеспечить совмещенное освещение общей освещенностью 300лк. Кроме того, вся рабочая поверхность должна быть освещена равномерно, то есть не должно быть резкого контраста яркости помещения и экрана, иначе зрение будет сильно напрягаться и быстро наступит утомление.

5.1 Расчет естественного освещения

Методика заключается в предварительном расчете площади световых проёмов. В случае бокового освещения используется формула:

$$100\frac{S_0}{S_n} = \frac{e_N K_з n_0}{t_0 r_1} K_{зд}, \qquad (5.1)$$

где: S_0 - площадь световых проемов при боковом освещении, м^2;

S_n - площадь пола помещения, м^2;

e_N – нормируемое значение КЕО;

$K_з$ – коэффициент запаса;

n_0 – световая характеристика окон;

t_0 - общий коэффициент светопропускания;

r_1 - коэффициент, учитывающий повышение КЕО при боковом освещении, благодаря свету, отраженному от поверхности помещения и подстилающего слоя, примыкающего к зданию;

$К_{зд}$ - коэффициент, учитывающий затемнение окон противостоящими зданиями.

Общий коэффициент светопропускания рассчитывается по следующей формуле:

$$t_0 = t_1 t_2 t_3 t_4 t_5, \qquad (5.2)$$

где t_1 – коэффициент светопропускания;

t_2 – коэффициент потери света в переплетах проема;

t_3 – коэффициент потери света в несущих конструкциях, равен 1 при боковом освещении;

t_4 – коэффициент потери света в солнцезащитных устройствах;

t_5 – коэффициент потери света в защитной сетке, под фонарями, равен 0,9.

Так как световым проемом будет окно с двойным стеклом, то

$$t_1 = 0{,}8. \tag{5.3}$$

Спаренные оконные переплеты, по таблице означают, что:

$$t_2 = 0{,}7. \tag{5.4}$$

Коэффициент потери света в регулируемых жалюзи равен:

$$t_4 = 1. \tag{5.5}$$

На основании собранных данных общий коэффициент светопропускания равен:

$$t_0 = 0{,}8 * 0{,}7 * 1 * 1 * 0{,}9 = 0{,}504. \tag{5.6}$$

Коэффициент запаса для офисов с вертикальным светопропускным материалом, равен:

$$K_з = 1{,}2. \tag{5.7}$$

Отношение длины помещения к глубине равно:

$$o1 = \frac{6}{5} = 1{,}2 \sim 1. \tag{5.8}$$

Отношение глубины помещения, к разности высоты верхнего края окна и высоты рабочей поверхности:

$$o2 = \frac{5}{2{,}5-1{,}2} = 3{,}8 \sim 4. \tag{5.9}$$

С помощью найденных значений находим значение световой характеристики окон при боковом освещении:

$$n_0 = 21. \tag{5.10}$$

Коэффициент r_1, с учетом определенных значений, равен:

$$\mathrm{r}_1 = 7{,}3. \tag{5.11}$$

Ввиду отсутствия зданий напротив световых проемов, коэффициент затемнения близко стоящими зданиями принимается равным:

$$К_{зд} = 1. \quad (5.12)$$

Нормируемое значение КЕО рассчитывается по формуле:

$$е_{N} = e_{н} * m_{N}, \quad (5.13)$$

где N – номер группы обеспеченности естественным светом;

$е_н$ – значение КЕО;

m_N – коэффициент светового климата.

Алматы относится к четвертой группе обеспеченности естественным светом,

$$N = 4. \quad (5.14)$$

Световые проемы ориентированы на восток, следовательно:

$$m_{N} = 0,65. \quad (5.15)$$

Значение КЕО, при высокой точности зрительной работы:

$$e_{н} = 1,2. \quad (5.16)$$

Теперь выполним расчет нормируемого значения КЕО:

$$е_{N} = 1,2 * 0,65 = 0,78. \quad (5.17)$$

Подставив полученные значения в формулу площади светового проёма, получим:

$$100\frac{S_0}{30} = \frac{0,78*1,2*21}{0,504*7,3} * 1 = 5,34. \quad (5.19)$$

Из полученного уравнения найдем площадь светового проёма:

$$S_0 = \frac{5,34*30}{100} = 1,6\ (м^2). \quad (5.20)$$

Таким образом, для достаточного освещения выделенного помещения достаточно окна площадью 1,6 м^2, следовательно, комната обеспечивается достаточным количеством естественного освещения, так как площадь его окна составляет 2м^2.

5.2 Расчет искусственного освещения

Метод коэффициента использования светового потока применяется для расчета освещённости в помещениях с общим освещением горизонтальных поверхностей.

Данный метод учитывает световые потоки непосредственно от светильников и отраженные от стен, потолков и самой поверхности.

Чтобы найти необходимое количество светильников, нужно определить световой поток, вычисляемы по формуле:

$$F = \frac{E*K*S*z}{n}, \tag{5.21}$$

где F – световой поток, лм;

E – нормированная минимальная освещенность, лк;

S – площадь освещаемого помещения;

Z – отношение средней освещенности к минимальной (Z = 1,02);

К – коэффициент запаса, учитывающий уменьшение светового потока лампы в результате загрязнения светильников в процессе эксплуатации (его значение зависит от типа помещения и характера проводимых в нем работ);

n – коэффициент использования, (выражается отношением светового потока, падающего на расчетную поверхность, к суммарному потоку всех ламп и исчисляется в долях единицы; зависит от характеристик светильника, размеров помещения, окраски стен и потолка, характеризуемых коэффициентами отражения от стен (РС) и потолка (РП)).

Работу программиста, в соответствии с этой таблицей, можно отнести к разряду точных работ, следовательно, минимальная освещенность будет:

$$\text{E} = 300, (\text{лк}). \tag{5.22}$$

Индекс помещения рассчитывается по формуле:

$$i = \frac{S}{h*(A+B)}. \tag{5.23}$$

Отсюда получаем:

$$i = \frac{30}{3*(6+5)} = 0{,}91. \tag{5.24}$$

Коэффициенты отражения потолка, стен и пола соответственно равны: 70%, 50%, 30%.

Используя полученные данные, можно определить коэффициент использования (n) по таблице (рисунок 5.1).

Типовая кривая	Равномерная								Косинусная Д								Глубокая Г							
$\rho_{п}$, %	70				50		30	0	70				50		30	0	0				50		30	0
$\rho_{с}$, %	50		30		50	30	10	0	50		30		50	30	10	0	50		30		50	30	10	0
$\rho_{р}$, %	30	10	30	10	10		10	0	30	10	30	10	10		10	0	30	10	30	10	10		10	0
i	Коэффициент использования, %																							
0,5	28	28	21	21	25	19	15	13	36	35	30	30	34	28	25	22	58	57	55	53	57	53	49	47
0,6	35	34	27	26	31	24	18	17	43	42	35	34	40	33	28	27	68	65	62	60	64	60	57	56
0,7	44	39	32	31	39	31	25	24	48	47	41	38	45	38	33	31	74	69	68	64	69	64	61	60
0,8	49	46	38	36	43	36	29	28	54	51	45	43	49	43	37	36	78	73	72	69	72	69	66	64
0,9	51	48	40	39	46	39	31	30	57	55	48	46	52	46	41	39	81	76	75	72	75	72	70	67
1,0	54	50	43	41	48	41	34	32	60	57	52	50	55	49	45	42	84	78	78	75	77	74	72	70
1,1	56	52	46	43	50	43	35	33	64	60	55	52	58	51	47	44	87	81	80	77	79	76	74	72
1,25	59	55	49	46	53	45	38	35	69	63	60	56	61	55	50	48	90	83	84	79	82	79	76	75
1,5	64	59	53	50	56	49	42	39	75	69	67	62	67	61	55	53	94	86	88	83	85	82	79	78
1,75	68	62	57	53	60	53	45	42	79	72	71	66	70	65	60	57	97	88	92	85	86	85	82	80
2,0	73	65	61	56	63	56	48	45	83	75	75	69	73	68	64	61	99	90	95	88	88	87	84	82
2,25	76	68	65	60	66	59	51	48	86	77	79	73	76	71	66	64	101	92	97	90	90	88	85	83
2,5	79	70	68	63	68	61	54	51	89	80	82	75	78	73	69	66	103	93	99	91	91	89	87	85
3,0	83	75	73	67	72	65	58	55	93	83	86	79	81	77	73	71	105	94	102	92	93	91	89	86
3,5	87	78	77	70	75	68	61	59	96	86	90	82	83	80	76	73	107	95	104	94	94	93	90	88
4,0	91	80	81	73	78	72	65	62	99	88	93	84	85	83	79	76	109	96	105	94	94	94	91	89
5,0	95	83	86	77	80	75	69	65	105	90	98	88	88	85	81	79	111	97	108	96	96	95	92	90

Рисунок 5.1 – Коэффициент использования светового потока

По данным таблицы, получается:

$$n = 0{,}51. \quad (5.25)$$

Коэффициент запаса для светодиодных ламп в офисных помещениях:

$$K = 1. \quad (5.26)$$

Теперь можно рассчитать необходимый световой поток:

$$F = \frac{300*1*30*1{,}02}{0{,}51} = 18000 \text{ (лм)}. \quad (5.27)$$

В качестве освещения, в офисе установлены светодиодные лампы СТ-15 в количестве 8 штук. Световой поток каждой лампы равен $F_л$ = 1 500 (лм).

Таким образом, световой поток данного освещения:

$$F_п = N * F_Л = 8 * 1500 = 12000 \text{ (лм)}. \quad (5.28)$$

Полученное значение светового потока меньше необходимого, что указывает о недостаточности данной системы искусственного освещения для комфортной работы в выделенном помещении. Для достижения достаточной освещенности необходимо увеличить количество ламп.

Рассчитаем необходимое количество ламп по формуле:

$$N = \frac{E*K*S*z}{F_л*n}, \quad (5.29)$$

где N – число ламп;

F – световой поток;

$F_л$ – световой поток лампы.

Подставив значения, получаем:

$$N = \frac{18000}{1\,500} = 12\ (шт). \quad (5.30)$$

Таким образом, для достижения достаточного уровня искусственного освещения в помещении необходимо увеличить количество ламп до 12 штук. Для установки освещения будут использованы светодиодные светильники типа «СТ» в количестве 4 штук. Расположение светильников будет производиться по вершине прямоугольника.

Высота светильников над рабочим местом определяется по следующей формуле:

$$h = H_п - \left(H_{св} + H_{р.п.}\right), \quad (5.31)$$

где $H_{св}$ – высота свеса ламп ($H_{св} = 0$ м);

$H_{р.п.}$ – расстояние рабочей поверхности над полом ($H_{р.п.} = 1м$);

$H_п$ – высота потолков помещения.

Подставив значения, получаем:

$$h = 3 - (0 + 1) = 2\ (м). \quad (5.32)$$

Необходимое расстояние между светильниками определяется по формуле:

$$L = \lambda * h, \quad (5.33)$$

где L – расстояние между светильниками, (м);

h – высота подвеса светильников над рабочей поверхностью;

λ– относительное наивыгоднейшее расстояние между светильниками ($\lambda = 2м$).

Тогда получается:

$$L = 2 * 2 = 4, (м). \quad (5.34)$$

Расстояние между рядами светильника:

$$L_b = \lambda * H_{р.п.} = 2 * 1 = 2, (м); \quad (5.35)$$

$$L_a = L_b * 1{,}5 = 2 * 1{,}5 = 3, (м), \quad (5.36)$$

где L_b – меньшая сторона прямоугольника расположения светильников;

L_a – большая сторона.

Расстояние между светильниками и стеной принимают равным в пределах (0,3 … 0,5) $L_{b,a}$.

Отсюда получаем:

$$l_a = 0{,}4 * L_a = 0{,}4 * 3 = 1{,}2, (\text{м}); \quad (5.37)$$

$$l_b = 0{,}4 * L_b = 0{,}4 * 2 = 0{,}8, (\text{м}), \quad (5.38)$$

где l_a – расстояние от стены до светильника для большей стороны прямоугольника;

l_b – расстояние от стены до светильника для меньшей стороны прямоугольника.

Теперь можно построить схему расположения светильников помещения (рисунок 2).

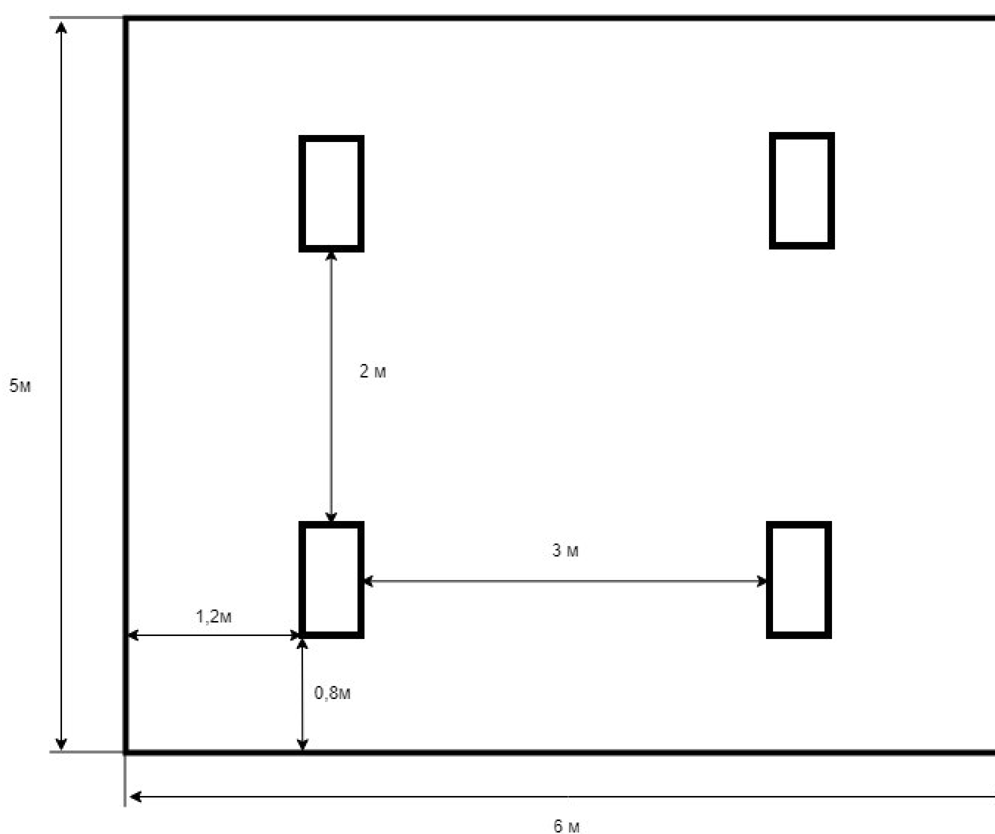

Рисунок 5.4 – Схема расположения светильников.

5.3 Итоги

По итогам проведенного анализа соответствия офиса требуемым нормам, было выявлено несколько проблем:

– эргономика рабочего места не предоставляет необходимого уровня комфорта проведения большого количества времени за персональным компьютером;

– искусственное освещение недостаточно для работы при отсутствии естественного света.

– Для устранения данных проблем было предложено внести следующие изменения:

– обеспечить эргономику рабочего пространства, предоставив сотруднику регулируемое кресло и более просторную площадь рабочего стола;

– обеспечить достаточную освещенность офиса, путем увеличения количества источников искусственного света по предоставленным расчётам.

Заключение

Во время реализации дипломного проекта были выполнены следующие задачи:

– локализованы и проанализированы проблемы современного управления базами данных;

– предложены пути решения проблем, путем разработки приложения, являющегося уникальной точкой доступа к различным СУБД;

– проведен анализ рынка на наличие аналогичных готовых решений и выполнено сравнение;

– проведен анализ предметной области;

– описаны требования к архитектуре программного обеспечения;

– выбраны технологии, архитектура и жизненный цикл программного обеспечения;

– созданы требования к пользовательскому интерфейсу и его прототип;

– выполнена реализация программного продукта;

– проведено экономическое обоснование целесообразности разработки продукта;

– рассчитаны необходимые требования к рабочему месту пользователя и предложены меры по их улучшению.

Итогом выполнения дипломного проекта стало приложение, цель которого упростить работу с большим количеством СУБД и сделать этот процесс более продуктивным. Данное приложение имеет классический макет интерфейса, что поможет потенциальным пользователям быстрее освоиться. Так же, оно имеет некоторые уникальные функции, которые помогут пользователям выполнять повседневные задачи продуктивнее.

Получившееся приложение занимает меньшее количество постоянной памяти устройства, за счет избавления от множества специализированных приложений и освобождает дополнительное место в оперативной памяти.

Список литературы

1 Адвайт Саркар, статья «The impact of syntax colouring on program comprehension», компьютерная лаборатория, Кембриджский университет, Кембридж, Великобритания, сайт https://pdfs.semanticscholar.org/d14b/edf3f58080ecf7a92f60746371b894a7bc08.pdf.

2 Мартин Роберт, «Чистый код: создание, анализ и рефакторинг», Издательский дом "Питер", 2013 г. - 464с.

3 Мартин Роберт, «Чистая архитектура. Искусство разработки программного обеспечения», Питер, 2018г. - 352с.

4 Фримен Эрик, Робсон Элизабет, Сьерра Кэти, Бейтс Берт, «Head First. Паттерны проектирования», Издательский дом "Питер", 2018 г. - 656с;

5 Ник Самойлов, Мохамед Санаулла, «Java 11 Cookbook: A definitive guide to learning the key concepts of modern application development, 2nd Edition», Packt Publishing Ltd, 2018 г. – 802 с.

6 Джордж Риз, «Database programming with JDBC and Java», Кембридж, Массачусетс: О'Рейли, 2000 г. - 328с.

7 Сайт https://docs.oracle.com/javase/8/docs/api, официальная документация языка Java.

8 Сайт https://tornadofx.io/dokka/tornadofx/tornadofx/index.html, официальная документация TornadoFX.

9 Бенджамин Мущко, «Gradle in Action», Мэннинг, 2014г. - 456с.

10 Сайт: https://docs.gradle.org, официальная документация по Gradle.

11 Скотт Чакон, «Pro Git», Апресс, 2014 г. - 288с.

12 Чакон Скотт, Штрауб Бен, «Git для профессионального программиста», Издательский дом "Питер", 2015 г. - 496с.

13 Сайт http://groovy-lang.org/documentation.html, официальная документация по языку Groovy.

14 Сайт https://kotlinlang.org/docs/reference, официальная документация языка Kotlin.

15 Хавьер Фернандес Гонсалес, «Java 9 Concurrency Cookbook», Packt Publishing Ltd, 2017г. - 594с.

16 Сайт https://github.com/akullpp/awesome-java, сборник библиотек и решений платформы JVM.

17 Чарли Хант, Моника Беквит, Пунам Пархар, Бенгт Рутиссон,«Java Performance Companion», Addison-Wesley Professional, 2016 г. - 192с.

18 Сайт https://ru.coursera.org/learn/kotlin-for-java-developers, курс «Kotlin for java developers».

19 Блох Джошуа, «Java. Эффективное программирование», Вильямс, 2019 г. - 464с.

20 Редмонд Эрик , Уилсон Джим Р., «Семь баз данных за семь недель. Введение в современные базы данных и идеологию NoSQL», ДМК Пресс, 2018 г. - 384с;

21 С. Д. Кузнецов, «Введение в модель данных SQL», М.: Национальный Открытый Университет «ИНТУИТ», 2016.

22 Даниэль Барретт, «Linux Pocket Guide, 3rd Edition», O'Reilly Media, 2016г. - 272с.

23 Мартин Фаулер, «UML Distilled», Addison-Wesley Professional, 2018 г. - 208с.

24 Елена Самойлова, статья «Бессмертная классика Waterfall», сайт: https://worksection.com/blog/waterfall.html, 2017г.

Приложение А
(обязательное)

Техническое задание

Общие требования

Наименование программного продукта (ПП): «DbUniTool».

Цель разработки ПО

Целью разработки является разработка приложения с единым унифицированным интерфейсом доступа к различным СУБД, которое будет обладать функциональностью удобного создания и чтения запросов к СУБД с поддержкой подсветки синтаксиса, поиска и нумерации строк; функциональностью выполнения запроса или запросов и просмотра результатов каждого запроса.

Идеология программного обеспечения

Данное программное обеспечение будет способствовать повышению продуктивности работы администраторов баз данных и разработчиков. Так же, за счет унификации рабочего интерфейса способствует снижению вероятности возникновения ошибок в рабочем процессе.

Используемые технология создания ПО

– JVM – виртуальная машина исполнения кода;
– Kotlin – язык программирования, имеющий возможность запускаться на JVM;
– JavaFX – фреймворк для разработки пользовательских интерфейсов;
– TornadoFX – библиотека, позволяющая минифицировать количество написанного кодв для пользовательского интерфейса;
– JavaRx – библиотека, предоставляющая API реактивного программирования.

Выбор модели ПО. Обоснование выбранной модели

Для реализации программного продукта была выбрана каскадная модель жизненного цикла.

Каскадная модель процесса разработки программного обеспечения, напоминает собой поток, последовательно проходящий фазы сбора и анализа требований, проектирования решения, реализации, тестирования, внедрения и поддержки.

Процесс разработки реализуется с помощью упорядоченной последовательности независимых шагов. Модель предусматривает, что каждый последующий шаг начинается после полного завершения выполнения предыдущего шага. В результате завершения шагов формируются промежуточные продукты, которые не могут изменяться на последующих шагах.

Продолжение приложения А

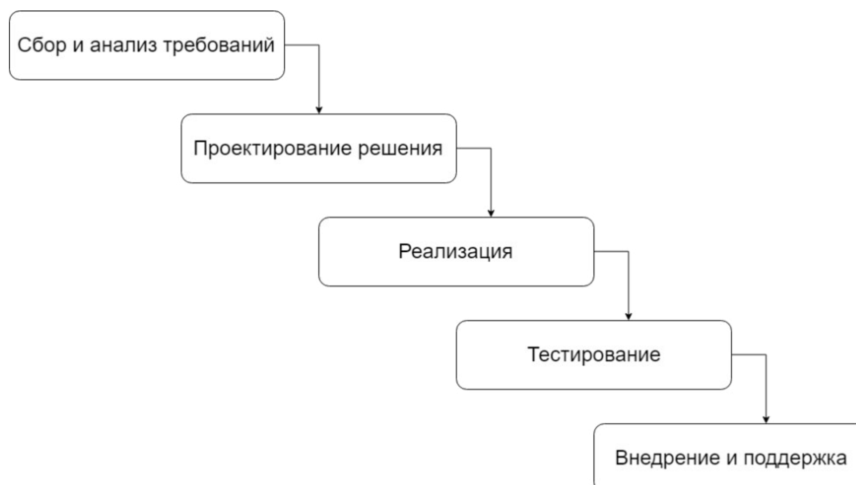

Рисунок А.1 – Каскадная модель

Выбор архитектуры построения ПП

В данном приложении должна быть применена однопользовательская архитектура ПП, так как настройки приложения будут относиться индивидуально к каждому пользователю. Кроме того, в рамках однопользовательской архитектуры данное приложение будет принадлежать к одной общей области применения.

Выбор метода разработки ПП

Степень автоматизации обозначается как «неавтоматизированное проектирование программ», так как оно наиболее подходящее для данного случая, когда разработкой ПО занимается очень ограниченный круг лиц.

Объектно-ориентированный подход к проектированию программных продуктов основан на следующих принципах:

– выделение классов объектов;
– установление характерных свойств объектов и методов их обработки;
– создание иерархии классов, наследовании свойств объектов и методов их обработки.

Каждый объект объединяет как данные, так и программу обработки этих данных и относится к определенному классу. С помощью класса один и тот же программный код можно использовать для относящихся к нему различных объектов.

Построение общей модели разрабатываемого программного продукта

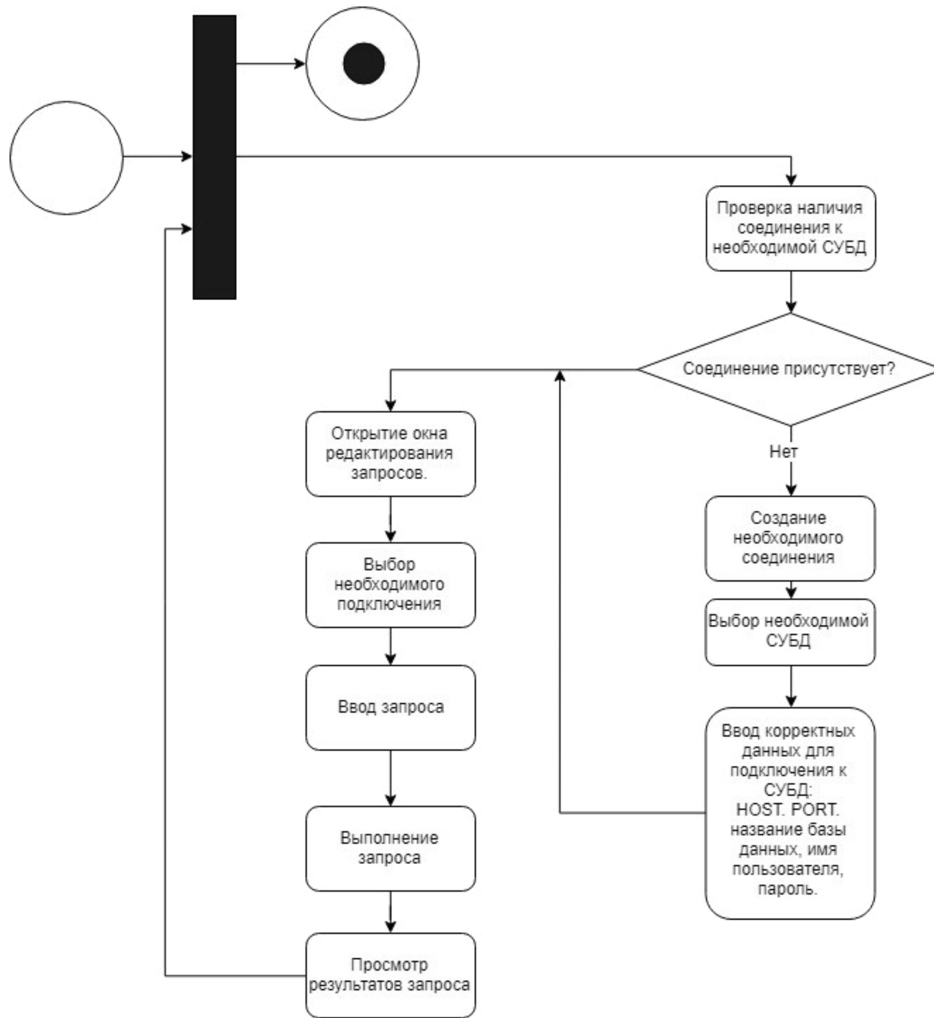

Рисунок А.2 – Модель разрабатываемого продукта на основе диаграммы деятельности

Выбор языка программирования

В качестве основного языка программирования выбраны Kotlin – статически типизированный язык программирования, работающий поверх JVM.

Предполагаемая аудитория

Целевая аудитория программного продукта – администраторы баз данных и разработчики программных продуктов, а так же другие люди, которым необходимо выполнять подключение к СУБД и выполнять запросы.

Общий объем ПП, Мб

Предположительный объем программного продукта – 10 Мб.

Технические требования

Основной диапазон разрешения мониторов, на которых будет просматриваться программный продукт:

– Наиболее удобным использование данного ПП будет на экранах с разрешением от 1600x900 до 3840x2160 дюймов.

Минимальное разрешение монитора, в котором будет просматриваться программный продукт: 1366x768 дюйма.

Операционные системы, на которых работает программное обеспечение:

Программное обеспечение должно на всех операционных система, для которых существует поддержка JVM.

- Минимальные требования к персональному компьютеру:
- CPU Intel Celeron;
- ОЗУ 4 Гб;
- HDD 500 Гб или SSD 32 Гб;
- Встроенная видеокарта.

Обеспечение защищенности программого продукта.

Безопасность программного продукта обеспечнивается отсутствием хранимых данных.

Тестирование и отладка программого продукта

Тестирование и отладка продукта должны выполняться изначально в ходе создания проекта и далее после окончания его разработки. В экономическом обосновании должны быть предусмотрены дополнительные часы на разработку программного обеспечения.

Специфические требования:

– Универсальность ПП. Программный продукт должен гарантировать пользователям единый интерфейс и поведение для всех доступных СУБД.

– Полнофункциональность ПП. Программный продукт будет предоставлять полноценный функционал управления той или иной СУБД не вводя никаких ограничений со своей стороны.

– Целостность программного продукта. Программный продукт будет представлять собой совершенно самостоятельное приложение без требований к установке какого-либо программного обеспечения.

Дизайн программного продукта (эстетика, стиль, цветовое решение)

Оформление приложения будет простым и инстуитивно понятным. Количество цветов должно быть ограниченным, чтобы цветовое восприятие было значительно упрощенно.

- Расположение элементов интерфейса

В верхней части интерфейса должен располагаться блок с навигацией, а основная часть окна предназначена для непосредственной работы пользователя.

В основной часть должны отображаться элементы управления соединениями и запросами к СУБД.

Продолжение приложения А

Эргономика (дружественность, последовательность и т.д.)

Пользовательский интерфейс должен быть максимально простым и удобным. Не должно быть перегруженности в меню выбора действий, а также максимально просто и понятно составлены настраиваемые параметры. Это обеспечит позитивный настрой пользователя при использовании приложения, т.к. не будет возникать раздражености от того, что не удается правильно использовать какой-либо пункт приложения.

Приложение Б
(обязательное)

Листинг программы

StartWorkWindow.kt
```
package start

class StartWorkWindow : App(WorkWindow::class) {
    override fun start(stage: Stage) {
        stage.title = "DbUniToll"
        stage.icons += Image(resources["/images/main.png"])
        stage.isMaximized = true
        super.start(stage)
    }
}
```
WorkWindow.kt
```
package view
class WorkWindow : View("DbUniTool") {
    override val root: BorderPane by fxml()
    private val menu = find(WorkMenu::class)
    private val navigation = find(NavigationPane::class)
    private val work = find(WorkPaneLayout::class)
    private val leftPane: AnchorPane by fxid()
    private val centerPane: AnchorPane by fxid()
    private val bottomPane: AnchorPane by fxid()
    private val rightPane: AnchorPane by fxid()
    init {
        root.top = menu.root
        leftPane.add(navigation)
        AnchorPane.setTopAnchor(navigation.root, 0.0)
        AnchorPane.setBottomAnchor(navigation.root, 0.0)
        AnchorPane.setLeftAnchor(navigation.root, 0.0)
        AnchorPane.setRightAnchor(navigation.root, 0.0)
        centerPane.add(work)
        AnchorPane.setTopAnchor(work.root, 0.0)
        AnchorPane.setBottomAnchor(work.root, 0.0)
        AnchorPane.setLeftAnchor(work.root, 0.0)
        AnchorPane.setRightAnchor(work.root, 0.0)
        loadFont(resources["/fonts/Roboto-Light.ttf"], 18)
    }
}
```
WorkMenu.kt
```
package view.components

class WorkMenu : Fragment("DbUniTool") {
    override val root = menubar {
        menu("File") {
            item("Save", "Shortcut+S"){
                onAction = EventHandler {
```

```
                saveFile()
            }
        }
        item("Load", "Shortcut+L") {
            onAction = EventHandler {
                loadFile()
            }
        }
        separator()
        item("Help", "Shortcut+H"){
            onAction = EventHandler {
                showHelp()
            }
        }
    }
    menu("Edit") {
        item("Preference"){
            onAction = EventHandler {
                openSettings()
            }
        }
        item("New SQL pane") {
            onAction = EventHandler {
                ContextManager.newContext()
            }
        }
    }
}

private fun loadFile() {
    val fileChooser = FileChooser()
    val file = fileChooser.showOpenDialog(currentWindow)
    if (file != null) {
        ContextManager.newContext(file.name, file.readText())
    }
}
}
```

NavigationPane.kt

```
package view.components.navigation

class NavigationPane : Fragment("DbUniTool") {
    override val root: AnchorPane by fxml()
    private val toolbar = find(NavigationToolbar::class)
    private val tree = find(NavigationTree::class)
    private val toolbarPane: AnchorPane by fxid()
    private val treePane: AnchorPane by fxid()

    init {
        toolbarPane.add(toolbar)
        AnchorPane.setTopAnchor(toolbar.root, 0.0)
```

```
    AnchorPane.setBottomAnchor(toolbar.root, 0.0)
    AnchorPane.setLeftAnchor(toolbar.root, 0.0)
    AnchorPane.setRightAnchor(toolbar.root, 0.0)
    treePane.add(tree)
    AnchorPane.setTopAnchor(tree.root, 0.0)
    AnchorPane.setBottomAnchor(tree.root, 0.0)
    AnchorPane.setLeftAnchor(tree.root, 0.0)
    AnchorPane.setRightAnchor(tree.root, 0.0)

    toolbar.removeEvent.subscribe {
      val info = tree.getSelected() ?: return@subscribe
      ConnectionManager.removeConnection(info)
    }
  }
}
```

NavigationToolbar.kt

```
package view.components.navigation

class NavigationToolbar : Fragment("DbUniTool") {
  override val root: AnchorPane by fxml()
  private val newConnection = find<NewConnection>()
  private val add: Label by fxid()
  private val remove: Label by fxid()

  val removeEvent = PublishSubject.create<Nothing>()
  init {
    root.addClass("navigationToolbar")
    add.onMouseClicked = EventHandler { startCreateNewConnection() }

    remove.onMouseClicked = EventHandler { startRemoveConnection() }
  }

  private fun startRemoveConnection() {
    removeEvent.onNext(null)
  }

  private fun startCreateNewConnection() {
    newConnection.openModal()
  }
}
```

NavigationTree.kt

```
package view.components.navigation

class NavigationTree : Fragment("DbUniTool") {
  override val root: AnchorPane by fxml()
  val tree =
    treeview<Any> {
      addClass("navigationTree")
      addClass("noFocus")
      root = TreeItem("")
```

```
        isShowRoot = false
        cellFormat {
            graphic = when (it) {
                is DbType -> ImageView(Image(it.image().toString(), 30.0, 30.0, true, true))
                is Connection -> DatabaseImage(it).root
                else -> null
            }
            text = when (it) {
                is String -> it
                is Connection -> it.info.connectionName
                is DbType -> it.title
                else -> it.toString()
            }
        }
    }

  init {
    root.add(tree)
    AnchorPane.setTopAnchor(tree, 10.0)
    AnchorPane.setRightAnchor(tree, 0.0)
    AnchorPane.setBottomAnchor(tree, 0.0)
    AnchorPane.setLeftAnchor(tree, 0.0)
    ConnectionManager.connections.subscribe { connections ->
        val c = connections.groupBy { it.info.dbType }
        tree.populate { parent ->
            when {
                parent == tree.root -> c.keys
                parent.value is DbType -> c[parent.value as DbType]
                else -> null
            }
        }
    }

  }
  fun getSelected(): ConnectionInfo? {
    val res = tree.selectedValue
    return if (res is Connection) res.info else null
  }
}
```

DatabaseImage.kt

```
package view.components.navigation

class DatabaseImage(connectionInfo: Connection) : Fragment(connectionInfo.info.connectionName) {
  private val tick = ImageView(Image(resources["/images/tick.png"]))
  override val root = stackpane {
    add(ImageView(Image(resources["/images/database.png"])))
    StackPane.setAlignment(tick, Pos.BOTTOM_RIGHT)
    add(tick)
```

```
    connectionInfo.isConnected.subscribe {
      if (it) {
        tick.show()
      } else {
        tick.hide()
      }
    }
  }
}
```

NewConnection.kt

```
package view.components.new_connection

class NewConnection : View("New connection") {
  override val root: AnchorPane by fxml()
  private val dbmsList = find(DBMSList::class)
  private var connectionProperties: ConnectionProperties? = null

  init {
    dbmsList.onDbmsSelected.subscribe { t -> toConnectionProperties(t) }
  }

  private fun toConnectionProperties(type: DbType) {
    root.children.remove(dbmsList.root)
    connectionProperties = find(ConnectionProperties::class, Pair("dbType", type))

    if (connectionProperties == null) {
      close()
      return
    }

    initializeConnectionProperties(connectionProperties!!)
  }

  private fun initializeConnectionProperties(propertiesElement: ConnectionProperties) {
    root.add(propertiesElement)
    propertiesElement.onBack.subscribe {
      toDbmsList()
    }
    propertiesElement.connectionInfo.subscribe { info ->
      ConnectionManager.addConnection(info)
    }
  }

  override fun onBeforeShow() {
    toDbmsList()
  }

  private fun toDbmsList() {
    if (connectionProperties != null) {
      root.children.remove(connectionProperties!!.root)
```

```
        }

        if (!root.children.contains(dbmsList.root)) {
            root.add(dbmsList)
        }
    }
}
```

DBMSList.kt

```
package view.components.new_connection


class DBMSList : Fragment("New connection") {
    override val root: AnchorPane by fxml()
    private val closeBtn: Button by fxid()
    private val nextBtn: Button by fxid()
    private val list: ListView<DbType> by fxid()
    val onDbmsSelected: PublishSubject<DbType> = PublishSubject.create()

    init {
        closeBtn.onMouseClicked = EventHandler { close() }
        nextBtn.onMouseClicked = EventHandler { selectDBMS() }
        with(list) {
            addClass("noFocus")
            cellFormat {
                text = it.title

                val img = it::class.java.getResource("/images/${it.dbName}.png")
                graphic =
                    if (img == null) {
                        ImageView(Image(resources["/images/folder.png"]))
                    } else {
                        ImageView(Image(img.toString(), 30.0, 30.0, true, true))
                    }
            }
            this.items = FXCollections.observableArrayList(allDbType)
            this.onDoubleClick {
                selectDBMS()
            }
        }
    }

    private fun selectDBMS() {
        if (list.selectedItem == null) {
            return
        }
        onDbmsSelected.onNext(list.selectedItem)
    }

}
```

ConnectionProperties.kt

```
package view.components.new_connection
class ConnectionProperties : Fragment("New connection") {
    override val root: AnchorPane by fxml()
    private val dbType: DbType by param(EMPTY_TYPE)
    private val connectionName: TextField by fxid()
    private val host: TextField by fxid()
    private val port: TextField by fxid()
    private val dbName: TextField by fxid()
    private val userName: TextField by fxid()
    private val password: PasswordField by fxid()
    private val backBtn: Button by fxid()
    private val okBtn: Button by fxid()

    private val context = ValidationContext()

    val connectionInfo: PublishSubject<ConnectionInfo> = PublishSubject.create()
    val onBack: PublishSubject<Void> = PublishSubject.create()

    init {
        initializeValidators()
        okBtn.onMouseClicked = EventHandler { createInfo() }
        backBtn.onMouseClicked = EventHandler { onBack.onNext(null) }
```

Продолжение приложения Б

```
        port.text = dbType.defaultPort.toString()
    }

    private fun createInfo() {
        if (!context.validate()) {
            return
        }
        val newCon = ConnectionInfo(
            connectionName.text,
            host.text,
            port.text,
            dbName.text,
            userName.text,
            password.text,
            dbType
        )
        connectionInfo.onNext(newCon)

        close()
    }

    private fun initializeValidators() {
        context.addValidator(connectionName, connectionName.textProperty()) {
            if (ConnectionManager.validateConnectionName(it)) {
                null
            } else {
                error("Incorrect contextName")’
```

```
        }
      }

      context.addRequireValidator(host)
      context.addRequireValidator(dbName)
      context.addRequireValidator(userName)
      context.addRequireValidator(password)

      context.addNumberValidator(port)
    }

}
```

WorkPaneLayout.kt

```
package view.components.work_pane

class WorkPaneLayout : View("WorkPane") {
    override val root: AnchorPane by fxml()
    private val workPane: TabPane by fxid()

    init {
        workPane.tabClosingPolicy = TabPane.TabClosingPolicy.ALL_TABS
        ContextManager.contexts.subscribe { contexts ->
            updateTabs(contexts)
        }
    }

    private fun updateTabs(contexts: List<WorkContext>) {
        for (context in contexts) {
            if (unknownContext(context)) {
                addNewTab(context)
            }
        }
    }

    private fun addNewTab(context: WorkContext) {
        workPane.tab(context.contextName) {
            val sqlEditor = SqlPanel(context)
            add(sqlEditor.root)
            onCloseRequest = EventHandler { sqlEditor.closeRequest() }
        }
    }

    private fun unknownContext(context: WorkContext) =
        workPane.tabs.filtered { t -> t.text == context.contextName }.isEmpty()
}
```

SqlPanel.kt

```
package view.components.work_pane

class SqlPanel(private val context: WorkContext) : Fragment(context.contextName) {
    override val root: AnchorPane by fxml()
```

```
    private val codePane: AnchorPane by fxid()
```

Продолжение приложения Б

```
    private val codeArea: CodeArea = CodeArea()

    private val conSelect: ComboBox<String> by fxid()
    private val run: Label by fxid()
    private val save: Label by fxid()
    private val open: Label by fxid()

    private val outContainer: AnchorPane by fxid()
    private val out: OutPane = find(OutPane::class, Pair("source", context.results))

    init {
        outContainer.add(out)
```

Продолжение приложения Б

```
        codeArea.paragraphGraphicFactory = LineNumberFactory.get(codeArea)
        codeArea
            .multiPlainChanges()
            .successionEnds(Duration.ofMillis(150))
            .subscribe { codeArea.setStyleSpans(0, computeHighlighting(codeArea.text)) }

        codeArea.addEventHandler(KEY_PRESSED) {
            if (it.code == KeyCode.ENTER && it.isControlDown) {
                executeSelected()
            }
        }
        codeArea.appendText(context.sql)
        codeArea.textProperty().onChange { context.sql = it ?: "" }
        codePane.add(codeArea)

        open.onMouseClicked = EventHandler { loadFile() }
        run.onMouseClicked = EventHandler { executeSelected() }
        save.onMouseClicked = EventHandler { context.save() }

        AnchorPane.setTopAnchor(codeArea, 0.0)
        AnchorPane.setBottomAnchor(codeArea, 0.0)
        AnchorPane.setLeftAnchor(codeArea, 0.0)
        AnchorPane.setRightAnchor(codeArea, 0.0)

        ConnectionManager.connections.subscribe { t -> updateConnectionsList(t) }
        conSelect.valueProperty().onChange { context.connection = it ?: "" }
    }

    private fun updateConnectionsList(t: List<Connection>) {
        with(conSelect) {
            items.clear()
            items.addAll(t.map { it.info.connectionName })
            if (items.contains(context.connection)) {
                value = context.connection
            } else if (items.size > 0) {
```

```
                value = items[0]
                context.connection = items[0]
            }
        }
    }

    private fun executeSelected() {
        var code = codeArea.selectedText
        code = when {
            code.isNullOrEmpty() -> codeArea.text
            else -> code
}

        code = code.replace(Regex("--[^\n]*" + "|" + "/\\*(.|\\R)*?\\*/"), "")

        QueryManager.executeQuery(context, code)
    }
    fun closeRequest() {
        alert(Alert.AlertType.WARNING,
            "Close request",
            "Save content in ${context.contextName} ?",
            ButtonType.YES,
            ButtonType.NO,
            owner = this.currentWindow,
            actionFn = {
                if (it == ButtonType.YES) {
                    val fileChooser = FileChooser()
                    fileChooser.initialFileName = context.contextName + ".sql"
                    fileChooser.showSaveDialog(currentWindow)?.writeText(context.sql)
                }
                ContextManager.removeContext(context)
            })
    }
    private fun loadFile() {
        val fileChooser = FileChooser()
        val file = fileChooser.showOpenDialog(this.currentWindow)
        if (file != null) {
            alert(
                Alert.AlertType.WARNING,
                "Where to open the file?",
                null,
                THIS_PANE,
                OTHER_PANE,
                ButtonType.CANCEL,
                owner = currentWindow,
                actionFn = {
                    if (it == THIS_PANE) {
                        codeArea.clear()
                        codeArea.appendText(file.readText())
                    } else if (it == OTHER_PANE) {
```

Продолжение приложения Б

```
                ContextManager.newContext(file.name, file.readText())
            }
        }
      )
    }
  }
}
```

Продолжение приложения Б

OutPane.kt

```
package view.components.work_pane

class OutPane() : Fragment("My View") {
  override val root = tabpane { }
  private val source: Subject<List<QueryResult>, List<QueryResult>> by param()

  init {
    source.subscribe { results ->
      clearResults()
      for (result in results) {
        addResult(result)
      }
    }

    AnchorPane.setLeftAnchor(root, 0.0)
    AnchorPane.setRightAnchor(root, 0.0)
    AnchorPane.setBottomAnchor(root, 0.0)
    AnchorPane.setTopAnchor(root, 0.0)
  }

  private fun addResult(result: QueryResult) {
    with(root) {
      tab(result.query) {
        val table = ResultTable(result.query, result)
        add(table)
        AnchorPane.setTopAnchor(table.root, -0.9)
        AnchorPane.setBottomAnchor(table.root, 0.0)
        AnchorPane.setLeftAnchor(table.root, 0.0)
        AnchorPane.setRightAnchor(table.root, 0.0)
      }
    }
  }

  private fun clearResults() {
    root.tabs.removeAll(root.tabs)
  }
}
```

ResultTable.kt

```
package view.components.work_pane

class ResultTable(text: String, result: QueryResult) : Fragment(text) {
```

```
override val root = anchorpane()

init {
   result.data.subscribe { setResult(it) }
}

private fun setResult(it: QueryResultData) {
   when (it) {
      is ErrorQuery -> {
         errorMessage(it)
      }
      is SuccessQuery -> {
         resultTable(it)
      }
      is SuccessNoResultQuery -> {
         successMessage(it)
      }
      is ExecutingQuery -> {
         loading(it)
      }
   }
}

private fun successMessage(query: SuccessNoResultQuery) {
   Platform.runLater {
      root.textarea("Success : " + query.text) {
         style {
            fill = c(0, 150, 0)
         }
         AnchorPane.setTopAnchor(this, 0.0)
         AnchorPane.setBottomAnchor(this, 0.0)
         AnchorPane.setLeftAnchor(this, 0.0)
         AnchorPane.setRightAnchor(this, 0.0)
         isEditable = false
      }
   }
}

private fun loading(query: ExecutingQuery) {
   Platform.runLater {
      root.vbox {
         AnchorPane.setTopAnchor(this, 0.0)
         AnchorPane.setBottomAnchor(this, 0.0)
         AnchorPane.setLeftAnchor(this, 0.0)
         AnchorPane.setRightAnchor(this, 0.0)
         alignment = Pos.TOP_CENTER
         imageview(Image("/images/loading.gif", 100.0, 100.0, false, false))
         button("STOP") {
            onMouseClicked = EventHandler {
               query.job.cancel()
```

```
            }
        }
        separator(Orientation.HORIZONTAL) { }
        text("From : " + SimpleDateFormat("yyyy/MM/dd hh24:mm:ss.sss").format(query.start))

      }
    }
  }

  private fun errorMessage(query: ErrorQuery) {
    Platform.runLater {
      root.textarea("ERROR : " + query.error) {
        style {
          fill = c(150, 0, 0)
        }
        AnchorPane.setTopAnchor(this, 0.0)
        AnchorPane.setBottomAnchor(this, 0.0)
        AnchorPane.setLeftAnchor(this, 0.0)
        AnchorPane.setRightAnchor(this, 0.0)
        isEditable = false
      }
    }
  }

  private fun resultTable(query: SuccessQuery) {
    Platform.runLater {
      root.tableview(query.rows.observable()) {
        isEditable = false
        multiSelect(true)

        AnchorPane.setTopAnchor(this, 0.0)
        AnchorPane.setBottomAnchor(this, 0.0)
        AnchorPane.setLeftAnchor(this, 0.0)
        AnchorPane.setRightAnchor(this, 0.0)

        for (c in query.columns) {
          val column = TableColumn<Row, String>(c.name)
          column.setCellValueFactory {
            return@setCellValueFactory SimpleStringProperty(it.value.data[c])
          }

          this.addColumnInternal(column)
        }
      }
    }
```

ApplicationUtil.kt

```
package util

const val appName = "DbUniTool"
```

```
val appConfigDir: Path = Paths.get(System.getProperty("user.home")).resolve(".config").resolve(appName)

val appCacheDir: Path = Paths.get(System.getProperty("user.home")).resolve(".cache").resolve(appName)
```

DefaultDb.kt

```
package util

import models.connection.DbType
import java.util.function.Function

val EMPTY_TYPE: DbType = DbType("", "", "", 0)
val POSTGRE_SQL: DbType =
    DbType("postgresql", "PostgreSQL", "org.postgresql.Driver", 5432)

val MY_SQL: DbType = DbType("mysql", "MySQL", "com.mysql.jdbc.Driver", 3306)

val MS_SQL: DbType =
    DbType("sqlserver", "MSSQL", "com.microsoft.sqlserver.jdbc.SQLServerDriver", 1433)

val ORACLE: DbType =
    DbType("oracle", "Oracle", "oracle.jdbc.OracleDriver", 1521, Function { info ->
        val url = info.host + if (info.port == null) "" else ":${info.port}"
        if (info.sid == null) {
            "jdbc:${info.dbType.dbName}thin:@//$url/${info.dbName}"
        } else {
            "jdbc:${info.dbType.dbName}thin:@$url:${info.sid}"
        }
    })

val allDbType = arrayListOf(POSTGRE_SQL, MS_SQL, MY_SQL, ORACLE)
```

FileUtil.kt

```
package util

fun File.createOrClean(): File {
    val parent = this.parentFile

    if (!parent.exists()) {
        parent.mkdirs()
    }
    if (!this.exists()) {
        this.createNewFile()
    }

    this.writeText("")
    return this
}
fun saveToJson(file: File, value: Serializable) {
```

```
    file.createOrClean()
        .writeText(toJson(value))
}

fun <T : Serializable> loadFromJson(src: File, clazz: Class<T>): T? {
    return try {
        fromJson(src.readText(), clazz)
    } catch (e: JsonSyntaxException) {
        null
    } catch (e: FileNotFoundException) {
        null
    }
}
```

ValidatorUtil.kt

```
package util

import javafx.scene.control.TextInputControl
import tornadofx.ValidationContext

fun ValidationContext.addRequireValidator(node: TextInputControl) {
    this.addValidator(node, node.textProperty()) {
        if (it.isNullOrEmpty()) error("Must be fill") else null
    }
}

fun ValidationContext.addNumberValidator(node: TextInputControl) {
    this.addValidator(node, node.textProperty()) {
        if (it!!.matches(Regex("^[0-9]*$"))) {
            null
        } else {
            error("Only number")
        }
    }
}
```

Saved.kt

```
package models

import util.loadFromJson
import util.saveToJson
import java.io.File
import java.io.Serializable
import java.nio.file.Path

interface Saved<T : Serializable> : Serializable {
    fun getFile(): File {
        return getParent().resolve(getFileName() + getFileExtension()).toFile()
    }

    fun getFileName(): String
    fun getParent(): Path
```

```
    fun getFileExtension(): String {
        return ".json"
    }

    fun save() {
        saveToJson(getFile(), this)
    }

    fun load(clazz: Class<T>): T {
        return load(getFile(), clazz)
    }

    fun load(from: File, clazz: Class<T>): T {
        return loadFromJson(from, clazz) ?: return getDefault()
    }

    fun getDefault(): T
}
```

Config.kt

```
package models

import util.appConfigDir
import java.io.Serializable

abstract class Config<T : Serializable>(private val name: String) : Saved<T> {
    override fun getFileName(): String {
        return name
    }

    override fun getParent() = appConfigDir
}
```

Cache.kt

```
package models
import util.appCacheDir
import java.io.Serializable

abstract class Cache<T : Serializable>(private val name: String) : Saved<T> {
    override fun getFileName(): String {
        return name
    }

    override fun getParent() = appCacheDir
}
```

WorkContext.kt

```
package models.work

class WorkContext(val contextName: String, var sql: String, var connection: String = "") :
    Cache<WorkContext>(contextName) {
    @Transient
```

```
    var results: BehaviorSubject<List<QueryResult>> =
BehaviorSubject.create(emptyList())

    companion object {
        @JvmStatic
        val emptyContext = WorkContext("", "", "")
    }

    override fun getDefault(): WorkContext {
        return emptyContext
    }

    override fun getParent(): Path {
        return super.getParent().resolve("contexts")
    }

    override fun load(from: File, clazz: Class<WorkContext>): WorkContext {
        val load = super.load(from, clazz)
        load.results = BehaviorSubject.create(emptyList())
        return load
    }
}
```

DbType.kt

```
package models.connection

import java.io.Serializable
import java.net.URL
import java.util.function.Function

data class DbType(
    val dbName: String,
    val title: String,
    val driverClass: String,
    val defaultPort: Int?,
    @Transient val toJDBCUrl: Function<ConnectionInfo, String> = Function { info ->
        val url = info.host + if (info.port == null) "" else ":${info.port}"
        "jdbc:${info.dbType.dbName}://$url/${info.dbName}"
    }
) : Serializable {

    fun image(): URL {
        return this::class.java.getResource("/images/$dbName.png")
            ?: return this::class.java.getResource("/images/folder.png")
    }

    override fun equals(other: Any?): Boolean {
        if (this === other) return true
        if (javaClass != other?.javaClass) return false

        other as DbType
```

```
    if (dbName != other.dbName) return false
    if (title != other.title) return false
    if (driverClass != other.driverClass) return false
    if (defaultPort != other.defaultPort) return false

    return true
  }

  override fun hashCode(): Int {
    var result = dbName.hashCode()
    result = 31 * result + title.hashCode()
    result = 31 * result + driverClass.hashCode()
    result = 31 * result + (defaultPort ?: 0)
    return result
  }

  override fun toString(): String {
    return "DbType(dbName='$dbName', title='$title', driverClass='$driverClass', defaultPort=$defaultPort)"
  }
}
```

ConnectionInfo.kt

```
package models.connection

data class ConnectionInfo(
  val connectionName: String,
  val host: String,
  val port: String?,
  val dbName: String,
  val user: String,
  val password: String,
  var dbType: DbType,
  val sid: String? = null
) : Serializable, Cache<ConnectionInfo>(connectionName) {
  companion object {
    @JvmStatic
    val emptyConnectionInfo = ConnectionInfo("", "", null, "", "", "", EMPTY_TYPE)
  }

  override fun getDefault(): ConnectionInfo {
    return emptyConnectionInfo
  }

  override fun getParent(): Path {
    return super.getParent().resolve("connections")
  }

  override fun getFileExtension(): String {
    return ".connection"
  }
```

```
    override fun load(from: File, clazz: Class<ConnectionInfo>): ConnectionInfo {
        val load = super.load(from, clazz)
        val db = allDbType.findLast { it == load.dbType }

        if (db != null) {
            load.dbType = db
        }

        return load
    }
}
```

Connection.kt

```
package models.connection

import org.apache.commons.dbcp.BasicDataSource
import rx.subjects.BehaviorSubject

class Connection(val info: ConnectionInfo) {
    var isConnected = BehaviorSubject.create<Boolean>(false)
    var ds = BasicDataSource()
    fun connect() {
        ds.url = info.dbType.toJDBCUrl.apply(info)
        ds.username = info.user
        ds.password = info.password
        ds.minIdle = 5
        ds.maxIdle = 10
        ds.maxOpenPreparedStatements = 100
        isConnected.onNext(true)
    }
}
```

QueryResult.kt

```
package models.query.result

import rx.subjects.BehaviorSubject

class QueryResult(val query: String, val data: BehaviorSubject<QueryResultData>)
```

loads.kt

```
package data

fun loadAllConnections(): List<Connection> {
    return try {
        Files.walk(emptyConnectionInfo.getParent())
            .filter { t ->
                t.toFile().isFile                                              &&
t.toFile().name.endsWith(emptyConnectionInfo.getFileExtension())
            }.map { t ->
                val load = emptyConnectionInfo.load(t.toFile(), ConnectionInfo::class.java)
                Connection(load)
```

```
        }.toList()
    } catch (e: NoSuchFileException) {
        emptyList()
    }
}
fun loadAllContext(): List<WorkContext> {
    return try {
        Files.walk(WorkContext.emptyContext.getParent()).filter { t ->
            t.toFile().isFile                                                  &&
t.toFile().name.endsWith(WorkContext.emptyContext.getFileExtension())
        }.map { t ->
            WorkContext.emptyContext.load(t.toFile(), WorkContext::class.java)
        }.toList()
    } catch (e: NoSuchFileException) {
        emptyList()
    }
}
```

ContextManager.kt

```
package data.context

object ContextManager {
    private const val DEFAULT_CONTEXT_NAME = "SQL"
    val          contexts:          BehaviorSubject<List<WorkContext>>          =
BehaviorSubject.create(loadAllContext())
    fun addContext(t: WorkContext) {
        t.save()
        contexts.onNext(contexts.value.plus(t))
    }

    fun removeContext(context: WorkContext) {
        Files.delete(context.getFile().toPath())
        contexts.onNext(contexts.value.filter { it.contextName != context.contextName })
    }

    private fun nameToContext(requestedName: String?): String {
        val names = contexts.value.map { it.contextName }
        var name: String = requestedName ?: DEFAULT_CONTEXT_NAME
        var i = 1
        while (names.contains(name)) {
            name = requestedName ?: DEFAULT_CONTEXT_NAME + i++
        }

        return name
    }

    fun newContext(name: String? = null, sql: String? = null): WorkContext {
        val workContext = WorkContext(nameToContext(name), sql ?: "")
        addContext(workContext)
        return workContext
    }
```

```
}
```

ConnectionManager.kt

```
package data.database
object ConnectionManager {
    var connections: BehaviorSubject<List<Connection>> =
BehaviorSubject.create(loadAllConnections())

    fun addConnection(t: ConnectionInfo) {
        val connection = Connection(t)
        checkDriverAndConnect(connection)
        t.save()
        connections.onNext(connections.value.plus(connection))
    }

    fun removeConnection(info: ConnectionInfo) {
        Files.delete(info.getFile().toPath())
        connections.onNext(connections.value.filter { it.info.connectionName !=
info.connectionName })
    }

    fun checkDriverAndConnect(connection: Connection) {
        if (connection.info == ConnectionInfo.emptyConnectionInfo ||
connection.info.dbType == EMPTY_TYPE) {
            throw IncompleteConnectionInformation()
        }

        loadDriver(connection)
        connection.connect()

        connection.ds.connection.use {
        }
    }
    private fun loadDriver(connection: Connection) {
        try {

            Class.forName(connection.info.dbType.driverClass)

        } catch (e: ClassNotFoundException) {
            throw DriverNotFoundException(connection.info.dbType.driverClass)
        }
    }

    fun validateConnectionName(name: String?): Boolean {
        if (name.isNullOrEmpty()) return false

        if (connections.value.find { it.info.connectionName == name } != null) {
            return false
        }

        return true
```

```
    }
}
```

QueryManager.kt

```
package data.database


object QueryManager {
    fun executeQuery(context: WorkContext, code: String) {
        val connectionInfo =
            ConnectionManager.connections.value.first { con -> context.connection ==
con.info.connectionName }

        val queries = code.split(SettingsManager.editorSettings.value.delimiter)

        ConnectionManager.checkDriverAndConnect(connectionInfo)

        val results = executeAll(queries, connectionInfo)

        context.results.onNext(results)
    }

    private fun executeAll(
        queries: List<String>,
        connectionInfo: Connection
    ): List<QueryResult> {
        val results = mutableListOf<QueryResult>()

        for (query in queries.map { it.trim() }.filter { !it.isEmpty() }) {
            val requestData = BehaviorSubject.create<QueryResultData>()
            requestData.onNext(ExecutingQuery(Date(), GlobalScope.launch {
                startExecuteQuery(connectionInfo, query, requestData)
            }))
            results += QueryResult(query, requestData)
        }
        return results
    }

    private fun startExecuteQuery(
        connectionInfo: Connection,
        query: String,
        requestData: BehaviorSubject<QueryResultData>
    ) {
        try {
            connectionInfo.ds.connection.use { connection ->
                connection.prepareStatement(query).use { ps ->
                    if (ps.execute()) {
                        ps.resultSet.use { rs ->
                            val columns = getColumns(rs)
                            val rows = getRows(rs, columns)
                            requestData.onNext(SuccessQuery(columns, rows))
```

```
                }
            } else {
                requestData.onNext(SuccessNoResultQuery("Updated
${ps.updateCount}"))
            }
        }
    }
} catch (e: Exception) {
    requestData.onNext(ErrorQuery(e))
}
}
private fun getRows(
    rs: ResultSet,
    columns: List<Column>
): List<Row> {
    val rows = mutableListOf<Row>()
    while (rs.next()) {
        rows += Row(columns.map { it to rs.getString(it.name) }.toMap())
    }
    return rows
}

private fun getColumns(rs: ResultSet): List<Column> {
    val columns = mutableListOf<Column>()
    val columnCount = rs.metaData.columnCount
    for (i in 1..columnCount) {
        columns                    +=                    Column(rs.metaData.getColumnName(i),
rs.metaData.getColumnTypeName(i))
    }
    return columns
}
}
```