Трудно в учении - легко в отчислении,
Аристотель, VI в. до н.э

2-е издание

ДЕМОЭКЗАМЕН:
наглядное пособие в примерах

Костыльное программирование на C++

Qt

KATZENFLEISH

ПРЕДИСЛОВИЕ

Однажды я уже начинал писать эту методичку, и благополучно ее удалил после некоторых событий. Удалил безвозвратно, о чем чуть позже пожалел, но судьба оказалась такова, что когда я увидел обновленное задание демонстрационного экзамена на грядущий год, то понял, что оно и не зря было удалено, и жалеть-то особо и не о чем. Задание предыдущих годов и нынешнего настолько разнятся, что все, о чем я писал ранее – ныне уже не актуально. Повод начать все с чистого листа, как говорится.

Эти древние свитки будут полностью ориентированы на КОД 09.02.07-2-2024, специальности 09.02.07 Информационные системы и программирование (программист). Допускаю, что студент другой специальности тоже заимеет из этого документа что-то полезное для себя, но реалии таковы, что смежных областей у разных заданий нынче немного, каждое задание заточено под конкретную специальность, так что особо уповать мимо проходящим все же не стоит. Тем более, что ИС специальность будет писать демоэкзамен на 1С, АБД из моего родного корпуса переехали, и текущий выпуск будет последним в этих стенах, а у РВП я вообще ничего не веду, и никогда не буду – веб-разработка мне никогда не была интересна. Такие дела.

Тот вариант, который нам любезно предоставили, был мною прорешен и осмыслен, я постараюсь изложить в этой методичке весь свой опыт, полученный в процессе проектирования и разработке приложения, требуемого по заданию, свои наблюдения, критику и прочие размышления. Сразу отмечу несколько важных моментов. Во-первых, весь мой код можно кратко охарактеризовать как «говнокод», поэтому ни на какую академичность, «правильность» и т.д., он не претендует. Во-вторых, ряд требований по самому заданию я благополучно пропустил, по разным причинам. Само ТЗ насколько качественное, что я просто в какой-то момент устал додумывать за его создателями, ряд требований показались мне столь непроработанными, что затаилась мысль о том, что подобного рода задания верстались либо в невероятной спешке, либо просто «на отвали», лишь бы было, да дополнительной сложности добавляло. Это лишь мое мнение, не знаю, как оно там на самом деле.

Приложение, требуемое к разработке в рамках выполнения демоэкзамена, будет разрабатываться на языке C++ в среде разработки Qt Creator 11.0.1 на версии фреймворка Qt 6.5.2, версии Qt 5/Qt 4 не подойдут, скорее всего. Для всех работ над базой данных будет использоваться СУБД MariaDB, графическая среда проектирования БД – MySQL Workbench 8.

Приступим.

ГЛАВА I. Разбор технического задания к ДЭ

Итак, техническое задание к демоэкзамену. Как уже упомянул выше, крайне ненавистный мне документ, по причине его крайней невнятности. Могу вам сразу порекомендовать несколько простых правил по работе с ним, которые упростят вашу жизнь:

1. Читайте вдумчиво. Не ломитесь вперед сломя голову, читайте внимательно. Требования к проектированию плавно размазаны по всему тексту технического задания. Собрать по крупицам полную картину того, что вам нужно будет сделать и каким образом реализовать, можно только в случае все же уважительного отношения к заданию. Небрежность приведет к неминуемому краху и потере времени, которого у вас и так немного.
2. Используйте записи. На демоэкзамене допускается использование черновиков, вам их предоставят, если они вам будут нужны. А они нужны. Те из вас, кто все же исправно ходил на мои ленты, в курсе относительно того, какие объемы информации на вас свалят, удержать это все в голове сложно, хотя и возможно. А вот бумага все помнит, если возникла идея – не стесняйтесь ее записать, отмечайте важные на ваш взгляд мысли, потому что потом рискуете их забыть, переключив свое внимание.
3. Здраво оценивайте свои силы и возможности. Прочитав ТЗ, отметьте себе те пункты, те требования, которые вы в состоянии выполнить, а те, что для вас показались неподъемными – пропускайте. Не зацикливайтесь на одной идее, если в голову мысли не идут, эта дорога приведет вас в тупик, так еще и время потеряете, оставьте на потом. Задание сформировано таким образом, что выполнить его в полном объеме – задача проблематичная, это нужно иметь какой-никакой, но опыт в разработке, нужно владеть паттернами программирования, и это все не про нас с вами, понятное дело. Не в таких объемах и не за такое время.
4. Основное правило: «если что-то не сказано – не делаем». Это относится ко всему документу, это извечная проблема студента – придумать себе трудности, а потом с ними бороться. Делать нужно только то, что было упомянуто, и только в тех объемах, которые требуются, ни больше, ни меньше. Несмотря на то, что я эту мысль высказал, я сам ей часто не следую, ну душу рвет, когда вижу, как что-то можно было было сделать иначе, чем нужно, и оно работало бы в тысячу раз лучше и практичнее, рука сама так и тянется писать код, который, в общем-то, и даже не будет оценен по итогу, ну, потому что не требовался.

Итак, теперь разберем само задание, я буду использовать краткие выжимки из текста и отдельные тезисы, полностью все перепечатывать не вижу никакого смысла. Полный документ с заданием можно найти на сайте колледжа, думаю, найдете.

Сам документ представляет из себя полный перечень всех тонкостей в моменте проведения демоэкзамена, вещь занимательная, можете почитать на досуге. Помимо прочего, в этом документе представлена таблица в перечнем баллов, присуждаемых на каждый полностью выполненный модуль. В совокупности всех баллов можно заиметь ровно 100, из них 80 баллов – по заданию от самого демоэкзамена, и остальные 20 – по вариативной части, задание которой разрабатывается на базе колледжа. Можно смело утверждать, что ни на намека на сложность в нем представлено не будет, к чему душить своих же студентов. Считайте эти 20 баллов своей «подушкой безопасности», которая

позволит вам перешагнуть за порог нужной вам оценки. Нашего же пристального внимания будет заслуживать та часть документа, которая начинается на 25 странице, пункт 3.6 «Образцы задания». В этом году, кстати говоря, непонятно для чего, основное задание представлено не отдельным документом, а в виде таблицы. Начнем с разбора первого модуля «Разработка модулей программного обеспечения для компьютерных систем».

По сути, это ваш регламент на разработку. В этом году добавилось дополнительное требование к обязательному формированию блок-схемы алгоритма вашего приложения, по этому поводу в тексте задания упоминается ГОСТ 19.701-90, в котором представлены требования по формированию схем алгоритмов, программ, данных и т.д. Почитать его, конечно же стоит, потому что не все студенты в курсе относительно того, что линии в схемах пересекать между собой не рекомендуется, что использование «стрелочек», показывающих направление движения в схеме, строго обязательно, что разделять схему на отдельные ветки выполнения может себе позволить только блок логического условия, а про правила использования блоков-разделителей, блоков-терминаторов и блоков-комментариев – ну, это понятно, наука сложная, ясное дело. Однако, если вы не хотите буквально на ровном месте терять 12 баллов, то придется все же этот ГОСТ разобрать, а в некоторых случаях – и подучить. Отдельного внимания требует таблица, представленная на 149 странице указанного ГОСТ. В ней приведены все символы (т.е. блоки), из которых формируются различные схемы, а также – какие блоки допустимы в использовании относительно разных схем. Вам предстоит разработать блок-схему работы программы, что есть 4-я колонка в таблице, и большая часть блоков, там представленных, в схемах такого типа не используются, будьте внимательны.

Отдельного рассмотрения требует упоминание об использовании в ваших программных кодах соглашения об именовании, другими словами – это выдержка стилистики вашего кода. Примеры в задании представлены для Python и C#, о том, что ВНЕЗАПНО студент волен писать код на любом языке программирования, разрабы благополучно умолчали, так вот, ознакомиться со стилистикой кода, применяемой в разработке на C++, можно ознакомиться по вот этой ссылке: https://gist.github.com/azoyan/b545f7b926f1f7fb40f8c285e3f5c545. Скажу честно, я не шибко сильно ему сам следую, особенно если пишу код сугубо «для себя», поэтому там, сидя, на демоэкзамене, не запаривайтесь относительно того, с какого регистра у вас начинаются наименования ваших объектов, или какие разделители в наименовании вы используете. Главное – придерживайтесь общего стиля, который вы выбрали, чтобы каждое наименование было стилистически выдержано относительно любого другого. А то находятся дикари, которые сперва пишут «int MaxArraySize = 5», а потом строчкой ниже «int count_max_value = 0». Так дела не делаются.

Относительно комментирования кода. Умение грамотно комментировать код, причем, к месту – великий навык, который в обществе разработчиков весьма ценится, однако, я вот не могу утверждать, что тот код, которые получается по итогу, когда программа уже написана, нуждается в дополнительном его комментировании. Можете прокомментировать отдельные функции по своему желанию, чтобы просто получить за это немного баллов, но не переусердствуйте, поскольку комментирование все же отъедает часть времени, которое можно было бы пустить на прямую разработку программы, или же – на ее дебаг. Однако запомните, что полностью закомментированных строчек или блоков кода в готовой к сдаче работе быть не должно, это требование задания. Упустите этот момент – и потеряете свои честно заработанные баллы.

Разбирая текст задания, можно заметить, что там прямо сказано относительно использования любых возможных обработчиков ошибок, дабы ваша программа не сваливалась в необрабатываемые исключения от всякого чиха в ее сторону. Особенно это касается работы с БД, поскольку ваша программа будет постоянно обращаться к БД, и, ВНЕЗАПНО, может не обнаружить там того, что искала, запутаться в дальнейшем выполнении и просто вылететь. Возьмите за правило, что всяких раз, когда ваша программа будет работать с прямым вводом пользователя, или же будет обращаться к БД, в такие участки кода нужно запихать максимум обработчиков ошибок, начиная от банального и небезопасного if...else, заканчивая достопочтеннейшим try...catch.

Помимо того, что программа постоянно должна следить за собственным выполнением, она должна общаться с пользователем, причем излагать свои «мысли» так, чтобы пользователь мог понять их однозначно. Самый простой пример некорректного использования такой возможности: программа пытается авторизировать внутри себя пользователя по его логину и паролю, и видит, что поле ввода пароля не было заполнено. Программа сообщает об этом пользователю сообщением «Неправильный логин или пароль», а потом это же самое сообщение выдает пользователю, который ввел их уже таки правильно, но программа потеряла связь с БД и не может адресовать на сервер запрос. А все потому, что программист решил схалявить и использовать одно сообщение на все типы ошибок, без разбора. Причем, в данном примере, этот программист оказался всюду неправ, ведь то, что поле ввода пустое, вовсе не означает, что набор входных данных ошибочен, там набор в целом неполный, и эта ошибка – совершенно другого типа, которая требует совершенно другого сообщения пользователю. Также необходимо понимать, что если вдруг программа совершит сделать какое-то необратимое действие, допустим, решит удалить данные из БД, или обновить их так, что средствами самой программы это откатить вспять будет невозможно, то программа обязана спросить у пользователя разрешение на это. Понятное дело, что сама программа не может просто так взять и что-то захотеть, скорее всего, это пользователь дал ей такое указание, но он мог сделать это случайно. В таких ситуациях спросить у пользователя дополнительного согласия явно лишним не будет.

Второй же модуль, именуемый как «Разработка, администрирование и защита баз данных», ВНЕЗАПНО, посвящен заданию на проектирование и разработку БД. Если в прошлых годах для выполнения демоэкзамена предоставлялся скрипт восстановления БД, то в этому году базу придется верстать самостоятельно. Если вы способны отличить ключ от простого поля, основную таблицу от простого справочника, и как реализовать связь «многим-ко-многим», то большой проблемы этот модуль вам не представит. Все, что сказано, в этом задании можно уместить в одно предложение: «создайте БД на сервере по техническому заданию на разработку, приведите ее к 3НФ и сохраните получившуюся схему отдельно на жесткий диск, в папку вашего проекта». На этом, собственно говоря, и все. Подробная работа над проектированием и разработкой БД будет представлена в третьей главе этой методички.

Третий же модуль я из своего повествования пропущу, поскольку это самый непонятный для меня образец задания. Я не писец Братства Стали, поэтому как писать ТЗ, особенно в рамках демоэкзамена – понятия не имею, к тому же, разрабы этого задания благополучно не предоставили никаких примеров, лишь кратко сослались на ЕСПД, что просто поразительно, надо сказать. Про добавление дополнительной роли «Менеджер» и внеочередного функционала к нему я тактично промолчу. Мало того, что ТЗ на основную часть программы и так, можно сказать, что неподъемное, так еще и вот это, а такие

требования к нему предъявляются, мама дорогая. Нет, вы не подумайте, там не то чтобы сложно, нет. Там просто никакой конкретики нет, вот и все, буквально – как понял из трех предложений, так и делай. Я считаю, что это несерьезный подход к делу, и это именно эта часть задания явно писалась на скорую руку, поскольку без нее сам модуль выглядел бы как-то неполноценно, и, в общем-то, совершенно не нужным, хотя он не перестал таким быть, честно скажу, лишь только усилил это ощущение.

Про установку и настройку каких-то «необходимых компонентов» и «ПО эксплуатации программного обеспечения» я вообще ничего говорить не буду. Что они там собрались устанавливать, если у нас повсеместно сейчас стоят ОС на базе ядра Linux, где без прав суперпользователя вообще ничего буквально поставить нельзя, и, понятное дело, учетные записи, под которыми вы будете работать, таких прав иметь явно не будут. Студентам групп АБД со своим заданием вообще придется все ставить на виртуальную машину, а потом думать, как скрестить программу на хосте с БД на виртуальной машине. Одним словом, понимание процессов современного образования у составителей заданий к демоэкзамену явно на высоте, ничего не сказать, жаль, что в отрицательном диапазоне.

Следом на очереди идет часть документа под названием «Описание предметной области». Эта часть невероятна важна, поскольку именно в ней находится основной пласт данных, которые потребуются для проектирования как БД, так и приложения. Эту часть документа следует читать особенно внимательно. Однако, следует сразу себе предостеречь от обдуманных поступков. Описание предметной области необходимо больше для ознакомления со сферой деятельности, в рамках которой вам предстоит работать, оно составляется так, чтобы максимально полностью охватить весь диапазон возможных данных, как информации, но при этом, я вам прямо на полном серьезе сейчас говорю, они и даже на половину в вашей работе использоваться и не будут, поскольку в самом техническом задании (речь о котором пойдет чуть позже) вполне вероятно, что даже упоминаний о ряде вещей, которые приводятся в описании предметной области, там просто не будет. Это не более, чем информационный мусор, призванный сбить вас с толку, чтобы начали проектировать избыточную систему, тратить больше времени на то, что даже не будет оцениваться, поскольку отсутствует в техническом задании, но, при этом, усложняет и без того непростой процесс разработки просто своим присутствием.

Итак, представленное описание предметной области говорит о том, что нам предстоит написать приложение для учёта заявок на ремонт оборудования. Далее идет описание основной сущности, вокруг которой и будет происходить основная работа, собственно, сама заявка на ремонт, а чуть ниже – описание пяти модулей, которые будут участвовать в процессе работы всей системы в целом, но фактически, мы будем разрабатывать только три из них: регистрацию заявки, ее обработку и выполнение. На реализацию мониторинга и сбора статистики у нас попросту не хватит времени, не забывайте, что на все мероприятие у вас отведено всего 4,5 часа работы.

Ну и само, техническое задание, конечно же, представленное, кстати говоря, в максимально ублюдском виде – в виде простого маркированного списка, без каких-либо пояснений и дополнений, то есть, буквально так: как понял – так и делаешь, ну об этом я уже упоминал ранее, тут все задание именно на таком принципе и базируется. Что, может быть, для нас только лучше, потому что мы вольны реализовывать требуемый функционал самыми костыльными способами, и, при этом, технически не отклоняясь от требований задания. Настало время непосредственной работы над заданием, в процессе будем опираться на те требования, которые представлены в ТЗ. Начнем.

ГЛАВА II. Работа с СУБД MariaDB. Подготовка БД. Импорт данных.

Первым делом изучаем файлы, которые будут вам переданы для работы с базами данных. По заданию сказано, цитата: «Заказчик системы предоставил файлы с данными (с пометкой import в ресурсах) для переноса в новую систему. Заполните базу данных». Скорее всего, это будут простые файлы Excel формата .xlsx, которые несовместимы с MySQL Workbench, и для импорта их потребуется сконвертивать (пересохранить) в формате .csv, т.е. документа с разделителем-запятой. Пока что пусть останутся как есть. Наличие этих файлов у вас – большой плюс, поскольку из них будет совершенно несложно восстановить структуры исходных таблиц, в которые предписывается импортировать данные. Останется только прокинуть связи между таблицами – и вся недолга. Однако, у меня таких таблиц на импорт нет, поэтому мне придется сверстать БД самостоятельно, опираясь на данные, которые я могу получить из текстов описания предметной области и технического задания.

В первую очередь, я создам основную таблицу заказов «Orders», вокруг которой позже, как в конструкторе, строить свою схему БД. Из пункта 2.1 ТЗ можно выяснить первые семь обязательных полей, которые обязательно должны быть:

- Номер заявки;
- Дата добавления;
- Оборудование, которое требует ремонта;
- Тип неисправности;
- Описание проблемы;
- Клиент, который подал заявку;
- Статус заявки.

Из этого же пункта можно сразу выловить значения, которые я позже буду заносить в справочник «Статус заявки». А что есть справочник и как его распознать? Если говорить кратко и по существу, справочники есть предопределенные наборы данных, значения которых подставляются в другие таблицы, дабы не заполнять однотипные данные для каждой отдельной записи каждый раз «от руки», а просто выбрать уже имеющиеся из списка по уникальному идентификатору, т.е. по ключу. Еще проще: если вам ВНЕЗАПНО захочется что-то выбрать однотипное из выпадающего списка, чем каждый раз вписывать что-то в поле ввода – это значения простого справочника, и должно быть соответствующим образом оформлено в БД. Типовая конструкция справочника состоит из уникального идентификатора, являющегося основным ключом, и поля, которое будет хранить соответствующее идентификатору значение. Пара «ключ-значение» будет представлять собой запись в справочнике. Это если кратко.

Итак, я сразу же опознаю в списке, представленным выше, свои будущие справочники. Это «Тип неисправности», его можно предопределить заранее, а после – просто выбирать нужный тип; «Клиент», хоть и не совсем справочник, но тоже будет работать по внешнему ключу, поскольку 3НФ не подразумевает хранение одних и тех же данных в разных таблицах, поэтому данные о клиенте будут храниться отдельно, и подгружаться в таблицу заказов, ну и, собственно, «Статус заявки». Из пункта 2 описания предметной области можно выдернуть информацию о приоритете заявки и типе устройства, и это те пункты, которые можно легко упустить, но которые стоит включить в структуру таблицы заказов. Разумеется, они также будут являться справочниками.

Полная структура таблицы заказов «Order» представлена в таблице 1.

Таблица 1 – Структура таблицы «Order»

Наименование поля	Тип данных	Тип поля
OrderID	Целый	Уникальный идентификатор записи
OrderStartDate	Дата (без времени)	Дата приема заказа
OrderDeviceName	Строка длиной 128 символов	Наименование устройства
OrderTypeDevice	Целый	Тип устройства, внешний ключ
OrderClient	Целый	Клиент, внешний ключ
OrderDescription	Длинный текст	Описание проблемы
OrderTypeDefect	Целый	Тип неисправности, внешний ключ
OrderPriority	Целый	Приоритет заявки, внешний ключ
OrderStatus	Целый	Статус заявки, внешний ключ
OrderClosedDate	Дата (без времени)	Дата закрытия заказа

Теперь встал вопрос относительно того, а как же реализовать то, что есть на бумаге, непосредственно в БД? Сразу стоит отметить, что вам всем будут выданы логины и пароли пользователя на сервере СУБД. У вашего пользователя в полном доступе будет выделенная БД, и только одна, вы не сможете создавать новые, придется работать только в той, что будет для вас специально создана. Над этой БД у вас будут полные права доступа, что хочешь – то и делай.

База будет пустой, и ее наполнение – это ваша задача, ваш труд. Вы, конечно, можете пользоваться терминалом, и делать все через клавиатуру, но, боюсь, что банально не успеете, поэтому не воспользоваться графической оболочкой на сервером СУБД в виде MySQL Workbench – ну просто усложнить самому себе жизнь. Для того, чтобы создать новую таблицу в БД, необходимо раскрыть в обозревателе объектов сервера СУБД свою БД (в следующем примере – это «user9_db», но далее вся работа будет вестись через БД «user10_db»), выбрать пока пустой каталог «Tables» и через контекстное меню от этого каталога выбрать пункт «Create Table». Этот процесс продемонстрирован на рисунках 1, 2.

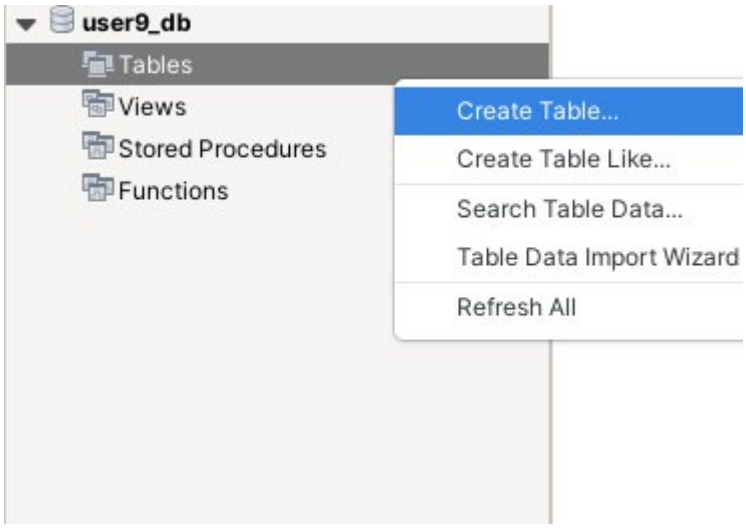

Рисунок 1 – Создание новой таблицы БД

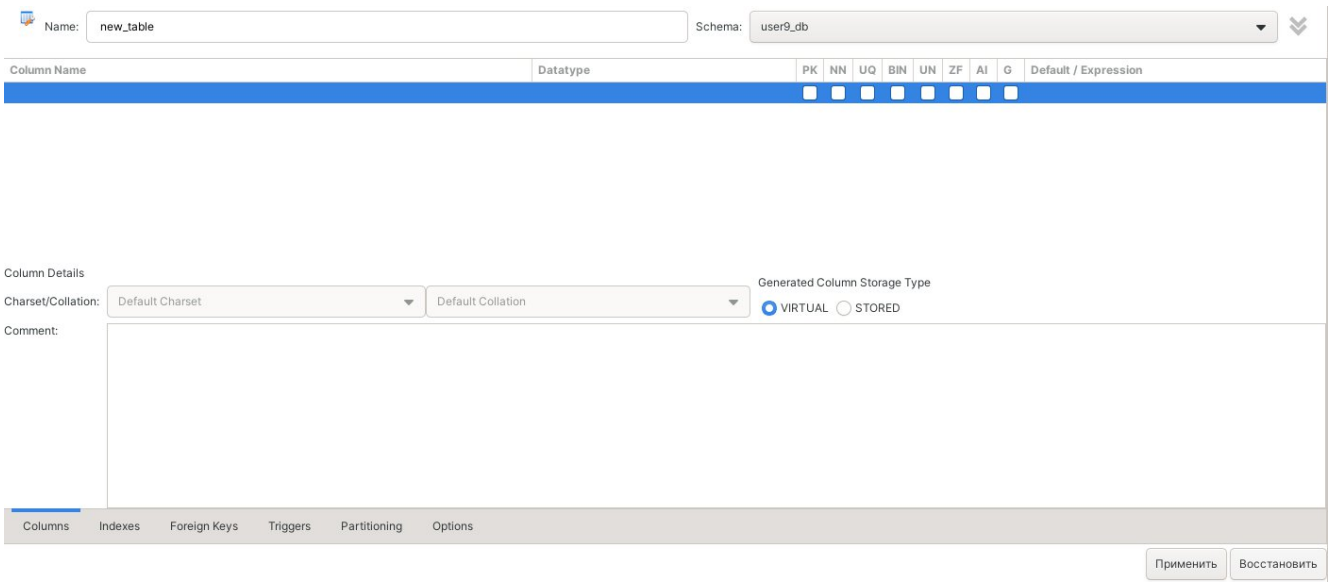

Рисунок 2 – Пустой макет таблицы БД

В поля колонки «Column Name» необходимо внести, собственно-то говоря, название поля таблицы, в колонке «Datatype» из выпадающего списка выбирается значение типа данных для этого поля, а после – можно выставить модификаторы для текущего поля. Остановимся подробнее на этих самых модификаторах. Все из них вам знать наизусть не обязательно, однако, самые основные все же выучить придется:

- ✓ PK – модификатор основного ключа для уникальных идентификаторов (ID);
- ✓ NN – модификатор «not null», не допускающий отсутствия какого-либо значения при создании записи, применяется студентами везде и всюду (что и хорошо), но строго обязателен для полей, содержащих основные и внешние ключи;
- ✓ UQ – модификатор уникального значения, не являющегося при этом основным ключом, это означает, что запись с таким значением может быть только одна на всю таблицу;
- ✓ AI – модификатор автоматической инкрементации значения поля на единицу относительно последнего максимального значения этого же поля, применяется во многом только для полей, содержащих основные ключи.

Этого списка достаточно, использование других модификаторов в рамках выполнения демоэкзамена избыточно. Отдельно хочется отметить, что при создании имен для объектов, неважно, имена таблиц или полей, вы не должны использовать ключевые слова SQL-синтаксиса: user, name, order, role, desc и так далее, полный список всегда можно найти в Интернете. Иначе вы рискуете потерять целостность запроса, поскольку СУБД будет воспринимать название, допустим, таблицы «order» как часть синтаксиса «order by», и непременно свалит ваш запрос в ошибку.

Нижняя часть окна представляет собой панель управления таблицей, в которой можно полностью настроить внутреннюю работу структуры таблицы. Нас будут интересовать всего две вкладки: «Columns», в которой настраивается скелет таблицы и «Foreign Keys», где происходит настройка связей между таблицами. Отдельного упоминания заслуживают постоянные вылеты MySQL Workbench на рабочий стол безо всякой причины и без последующей ошибки, просто программа аварийно закрывается. Рекомендую после каждого поэтапного изменения структур таблиц сохранять изменения кнопкой «Применить», поскольку последствия слета половины БД на демоэкзамене могут быть ужасны.

На текущий момент, у меня отсутствуют какие-либо таблицы, с которыми я мог бы связать таблицу заказов, да и самой таблицы-то, как таковой, у меня тоже нет. Рисунок 3 демонстрирует сформированную структуру таблицы заказов.

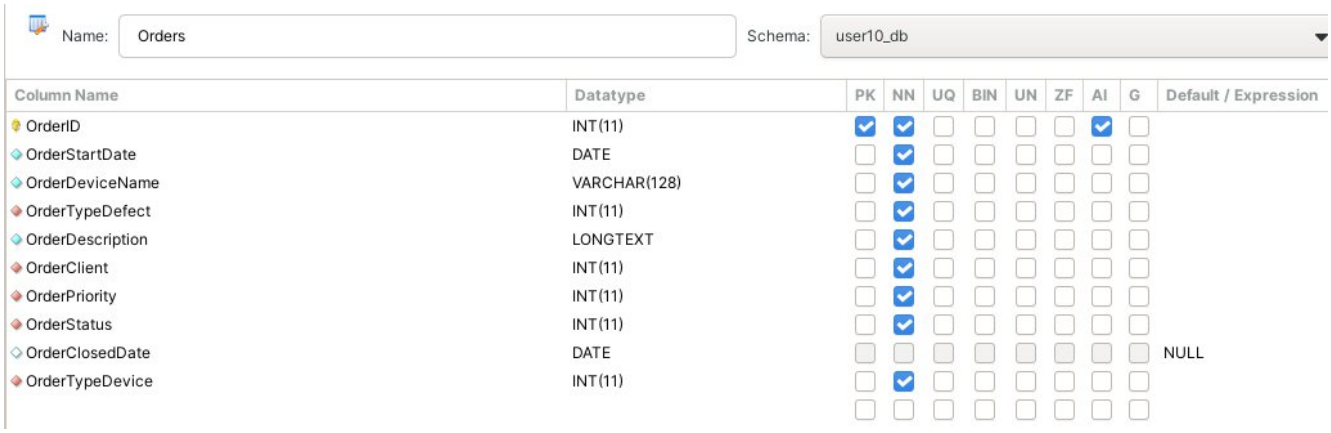

Рисунок 3 – Структура таблицы «Orders»

Далее, не долго думая, можно сразу же сделать нужные мне справочники. Они все будут однотипными, и на два поля: ID и значение. Приведу один скриншот, дабы было общее понимание того, как оно выглядит, остальные же будут созданы по образу и подобию оного.

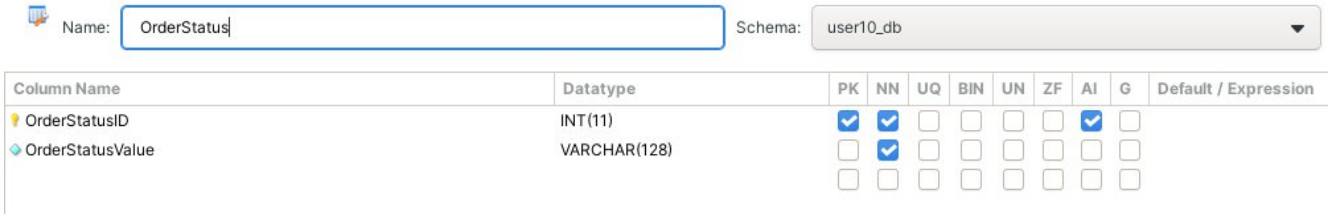

Рисунок 4 – Структура таблицы «OrderStatus»

Последняя таблица, о необходимости которой мне точно известно, это таблица клиентов. Основная загвоздка здесь в том, что мне одновременно нужны и клиенты, и сотрудники, ну ведь, в самом же деле, не клиенты же ведут фактический учет заявок, верно. Набор данных для них будет одинаковым, что для клиента, что для сотрудника: ФИО и телефон. Почему именно такой набор? Ну, в ТЗ об этом ни слова, буквально, ни единого упоминания о том, из чего должна формироваться структура таблицы клиентов, поэтому я волен сам решить этот вопрос, и, конечно же, я сделаю это в свою пользу, обозначив минимально возможное и достаточное количество данных. Однако, третья нормальная форма мне буквально запрещает хранить по сути одни и те же данные в разных таблицах, у студента автоматически зачешется рука просто бахнуть таблицы «Клиент» и «Сотрудник», решив, казалось бы, проблему, но это не наш вариант, в моем случае, я объединю эти две сущности в одну - «Пользователь». Ведь это так и есть, и клиент, и сотрудник – это просто пользователи одной системы с разными ролями в ней. Ролями! Вот чего точно не хватает, и действительно, как система будет определять, кто есть «клиент», а кто «сотрудник», если никаких различий базовых между ними нет? Для придется сформировать дополнительный справочник с возможными ролями в системе, и подкидывать каждую отдельно взятую роль к определенному пользователю. Однако, можно задаться вопросом: а как же авторизация? Простой клиент не должен иметь возможности авторизироваться в системе, у него и не должно быть данных для

авторизации, а если данные о пароле и логине будут храниться внутри структуры таблицы всех пользователей, то это означает, что и для простого клиента придется придумать пароль и логин, что полностью противоречит пункту 3.2 технического задания. Решение этой проблемы я переложу на вспомогательную таблицу, которая будет хранить идентификатор пользователя и его данные для входа. Зная ID сотрудника, я всегда смогу получить пару «логин-пароль» для его авторизации в программе, вот и все. Структура таблицы пользователей представлена на рисунке 5, а структура таблицы авторизации – на рисунке 6.

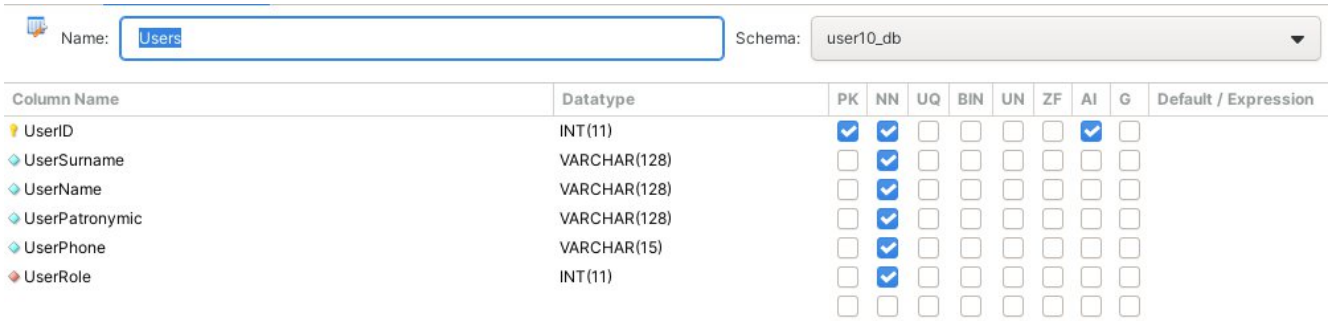

Рисунок 5 – Структура таблицы «Users»

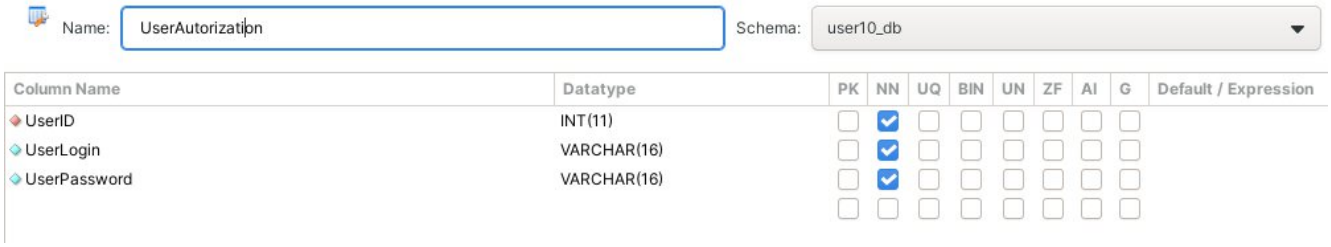

Рисунок 6 – Структура таблицы «UserAutorization»

Теперь – о наболевшем, о связях. Механику и принцип работы основных и внешних ключей дружно идем гуглить, если пробел в знаниях таки имеется. Сейчас у меня висит куча между собой никак не связанных таблиц, которые таки нужно скрестить между собой. Вкладка «Foreign Keys» мне в помощь, она представлена на рисунке 7. Краткая суть механики этого инструмента: рабочее пространство поделено на 4 колонки, в первую необходимо вписать **уникальное** название будущего ключа, MySQL Workbench подставит его самостоятельно, если глаза при этом кровью не текут – можно оставить и такое, далее, во второй колонке необходимо выбрать таблицу В КОТОРОЙ НАХОДИТСЯ **ОСНОВНОЙ** КЛЮЧ. Из этого можно сделать простой, но словно неочевидный вывод, что связи прокидываются из таблиц, в который находится **ВНЕШНИЕ** ключи, поскольку именно **внешний** ключ нужно сопоставить с **основным**, не наоборот! Третья же колонка отвечает за, как раз таки, выбор поля с внешним ключом, к которому будет идти сопоставление. Если все сделано и выбрано правильно, то среда автоматически в четвертую колонку подкинет основной ключ в пару ко внешнему, и такую связь можно сохранять нажатие кнопки «Применить». Замечу, что именно в этом процессе (не сохранения, а работы с ключами в целом) программа стабильно вылетает, порой ей не нравятся слишком длинные имена ключей, иногда после перезапуска она сама себя чинит, иногда нужно сперва прокинуть связи в другой таблице, и тогда в текущей перестанет вылетать. Сложно сказать, почему это происходит, но нужно быть наготове и не иметь несохраненных изменений в БД на момент работы со связями, мало ли – вылетит.

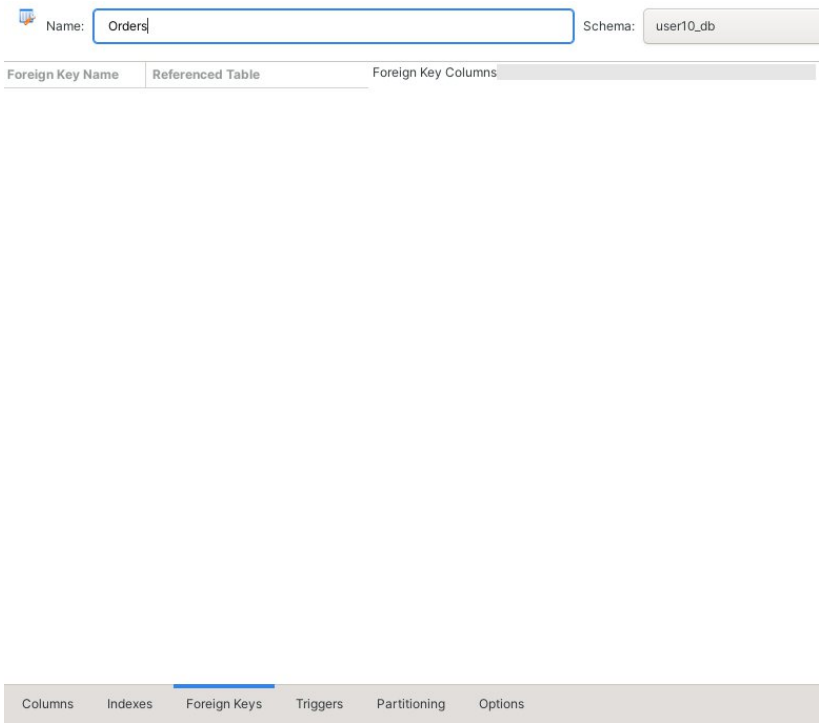

Рисунок 7 – Вкладка «Foreign Keys» внутри таблицы «Orders»

Общий процесс создания связи между таблицей «Orders» и таблицей «OrderStatus» представлен на рисунках 8-10.

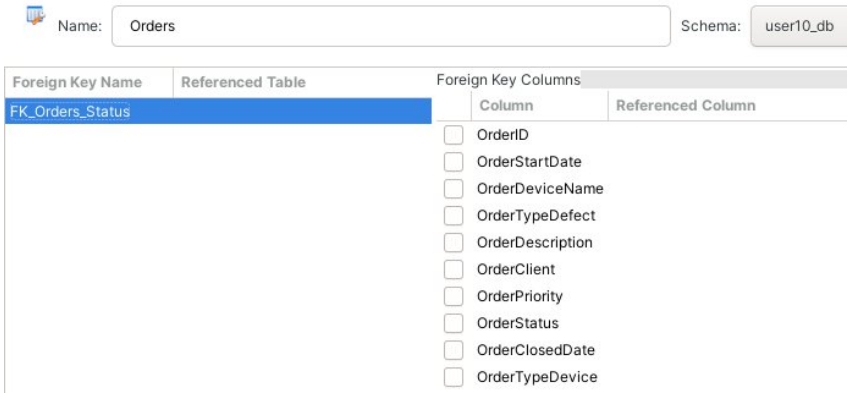

Рисунок 8 – Установка имени для связи с таблицей «OrderStatus»

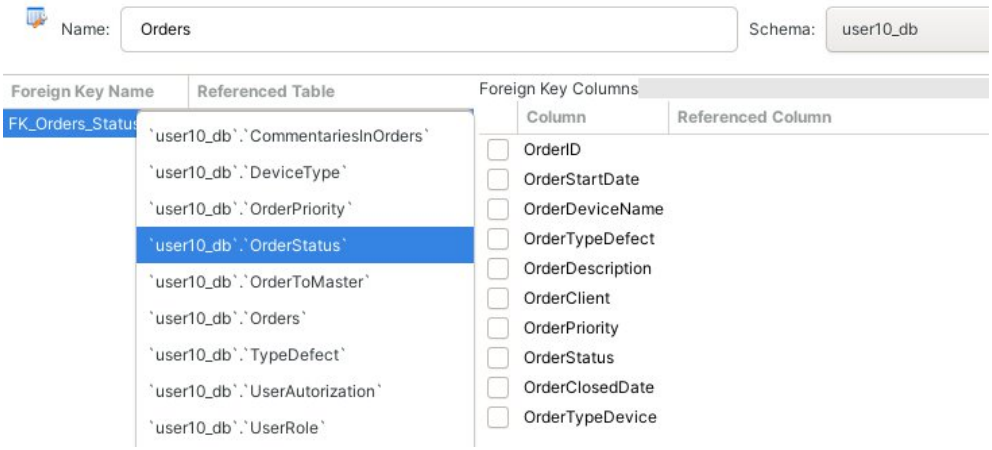

Рисунок 9 – Выбор таблицы «OrderStatus» с основным ключом

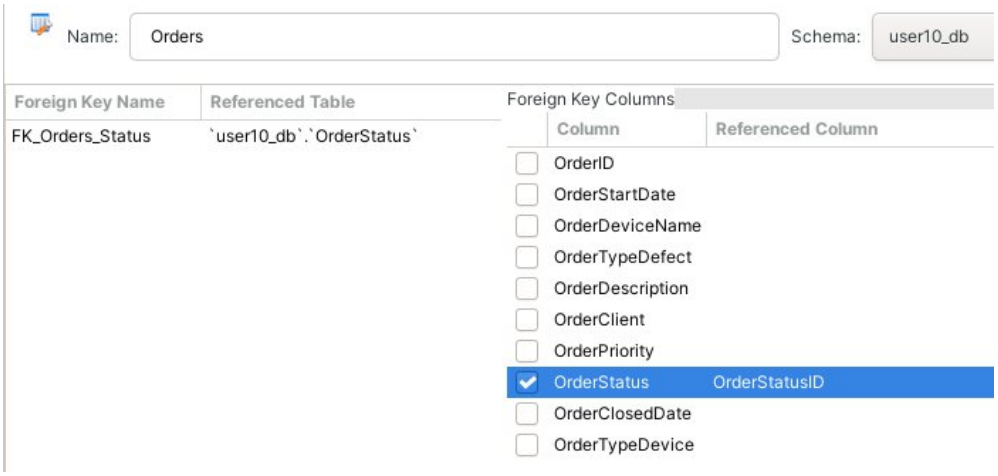

Рисунок 10 – Выбор внешнего ключа внутри таблицы «Orders» с основным из таблицы «OrderStatus»

Вот, в целом, и вся наука. По итогу связи внутри таблицы «Orders» принимают вид, представленный на рисунке 11.

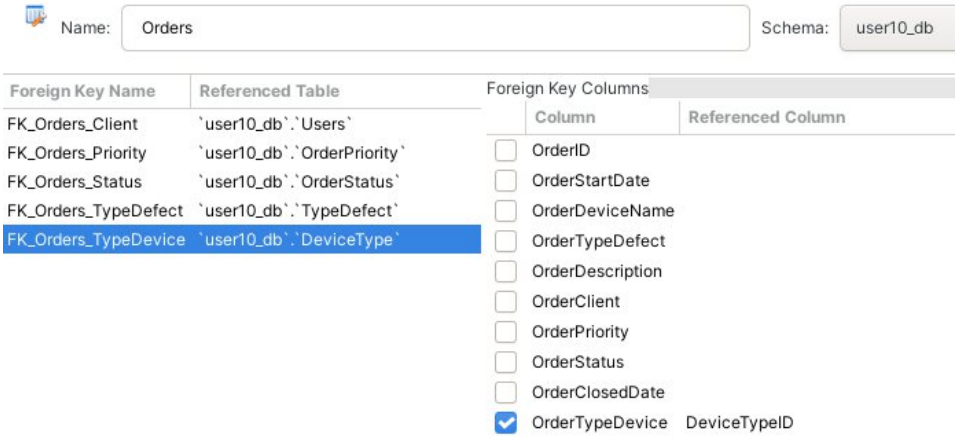

Рисунок 11 – Структура связей таблицы «Order»

Далее идет моя любимая часть Марлезонского балета, а именно – додумывание. Больше мне ничего в ТЗ явно не дано, однако сказано, что информация о исполнителе заявки определяется на этапе обработки заявки, а не в момент ее создания, это видно из пункта 3 описания предметной области и пункта 2.2 технического задания. Помимо прочего, пункт 2.4 ТЗ гласит, что «Исполнитель может добавлять комментарии на форме заявки», т.е. эти комментарии должны относиться к самой заявки и существовать параллельно с ней. Задачка та еще, но попробуем ее решить простыми методами. Для хранения информации об исполнителе заявки мною будет создана вспомогательная таблица «многим-ко-многим», что позволит соотнести множество пользователей со множеством заявок. Вы можете заметить, что в таком случае и клиент, чисто технически, будет иметь возможность быть исполнителем у заявки, и с точки зрения БД это действительно так, однако, такую проблему мы решим программным способом, на уровне кода. Комментарии будут реализованы так же на базе таблицы БД, которая будет хранить информацию о самой заявке, т.е. к какой заявке будет относиться комментарий, ID сотрудника, который оставил этот коммент, собственно, сам текст комментария и время, когда он был оставлен, причем не просто дата, но и время. Структуры этих таблиц представлены на рисунках 12, 13.

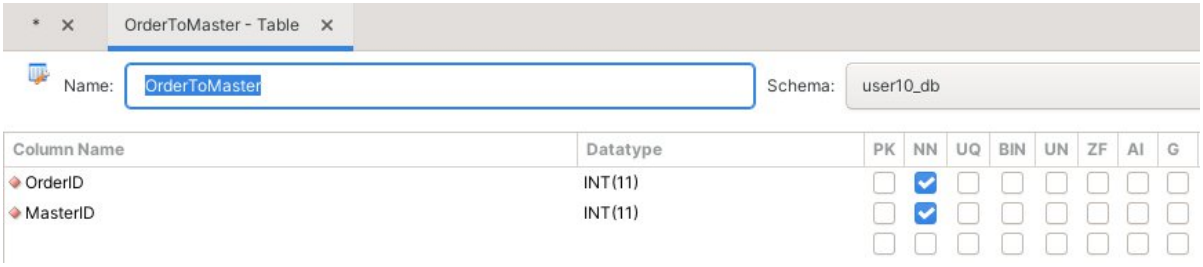

Рисунок 12 – Структура таблицы «OrderToMaster»

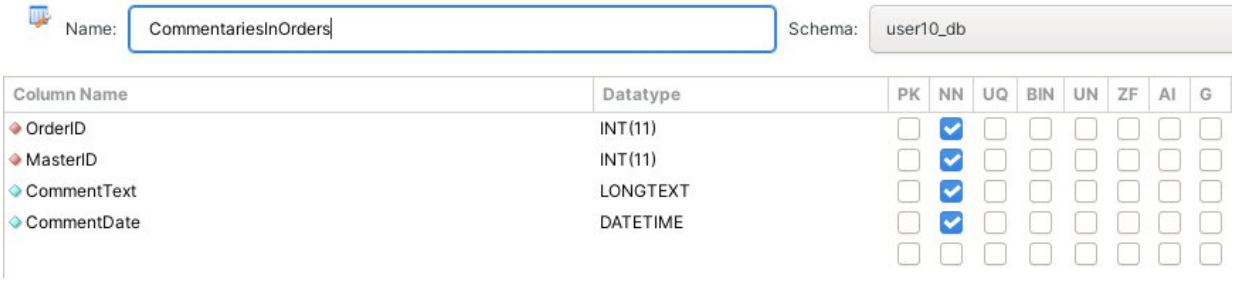

Рисунок 13 – Структура таблицы «CommentariesInOrders»

После определения всех связей внутри таблиц БД, общий вид схемы БД «user10_db» будет иметь следующий вид, представленный на рисунке 14.

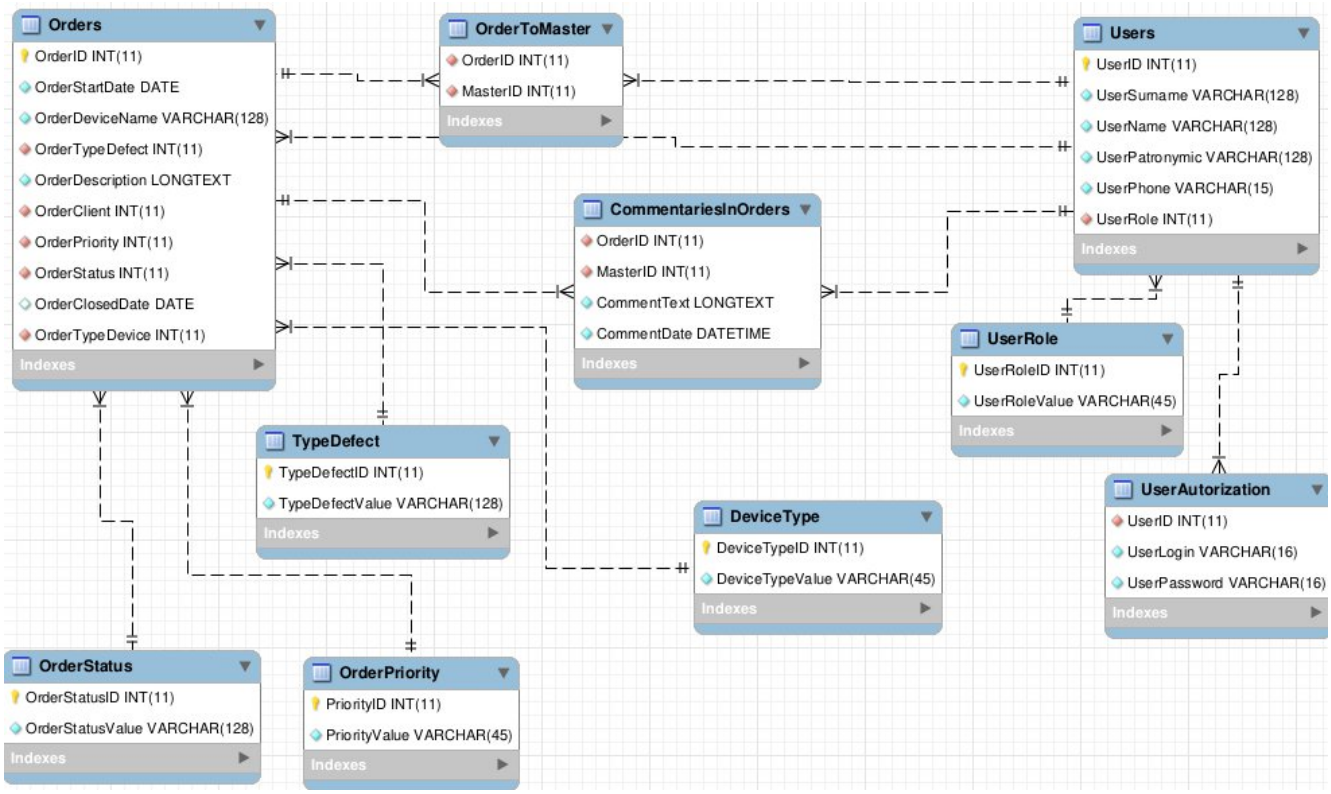

Рисунок 14 – Схема данных БД «user10_db»

Получить такую схему можно, выполнив комбинацию клавиш «Ctrl+R», в появившемся окне необходимо указать текущее подключение к БД, далее выбрать саму БД, для которой будет происходить построение схемы, и просто прокликивать кнопки «Next» и «Execute», пока схема не отобразится в рабочем пространстве среды MySQL Workbench. К слову говоря, именно эту схему нужно сохранить в папку вашего проекта. Общий процесс построения схемы данных представлен на рисунках 15-20.

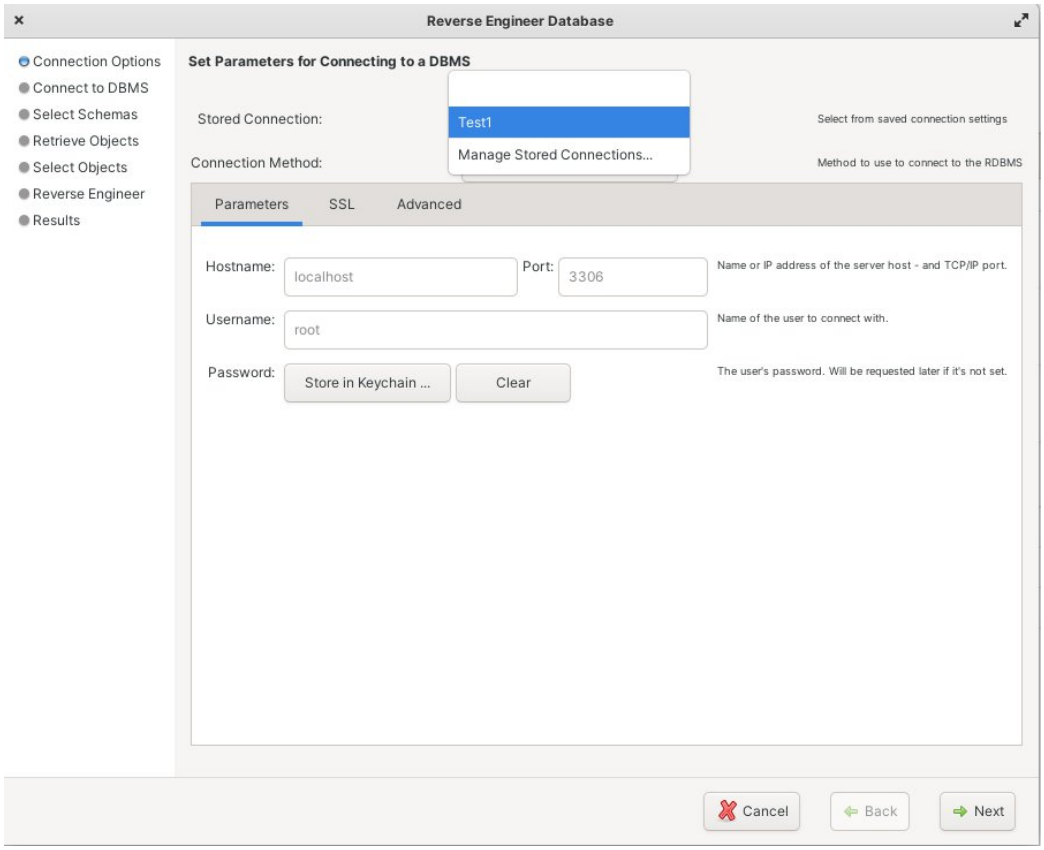

Рисунок 15 – Выбор соединения к серверу СУБД

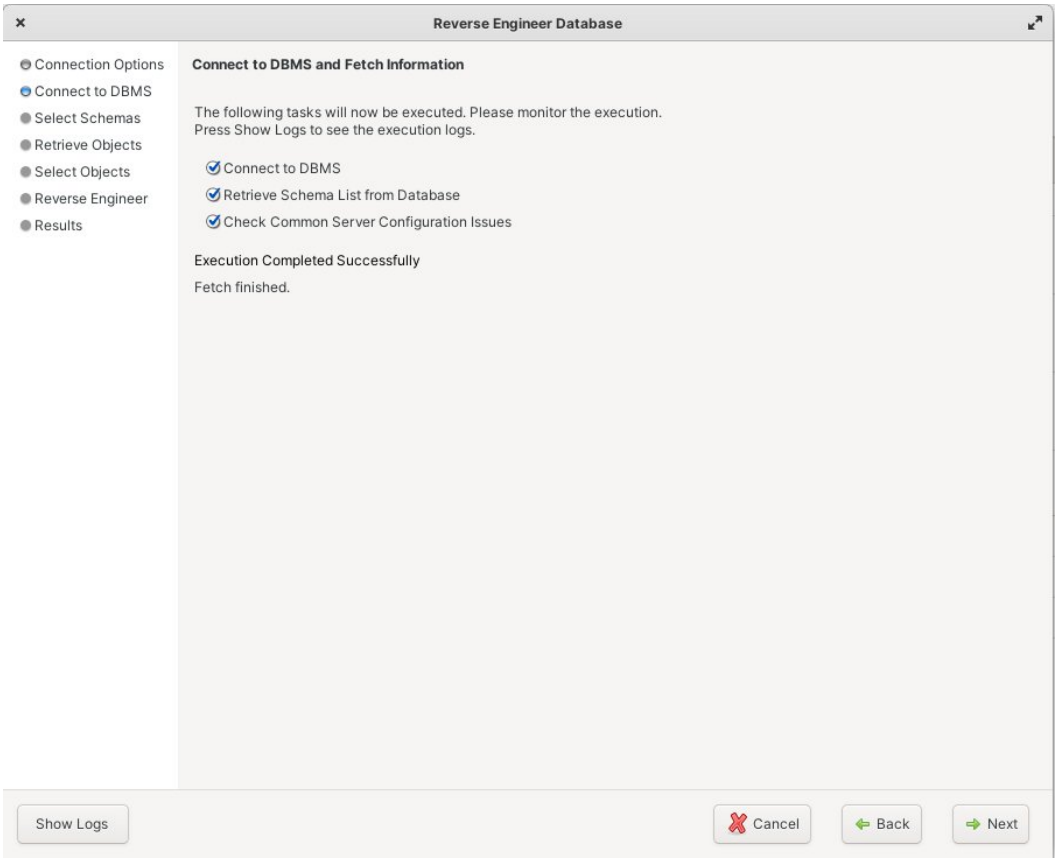

Рисунок 16 – Подтверждение успешного подключения к серверу СУБД

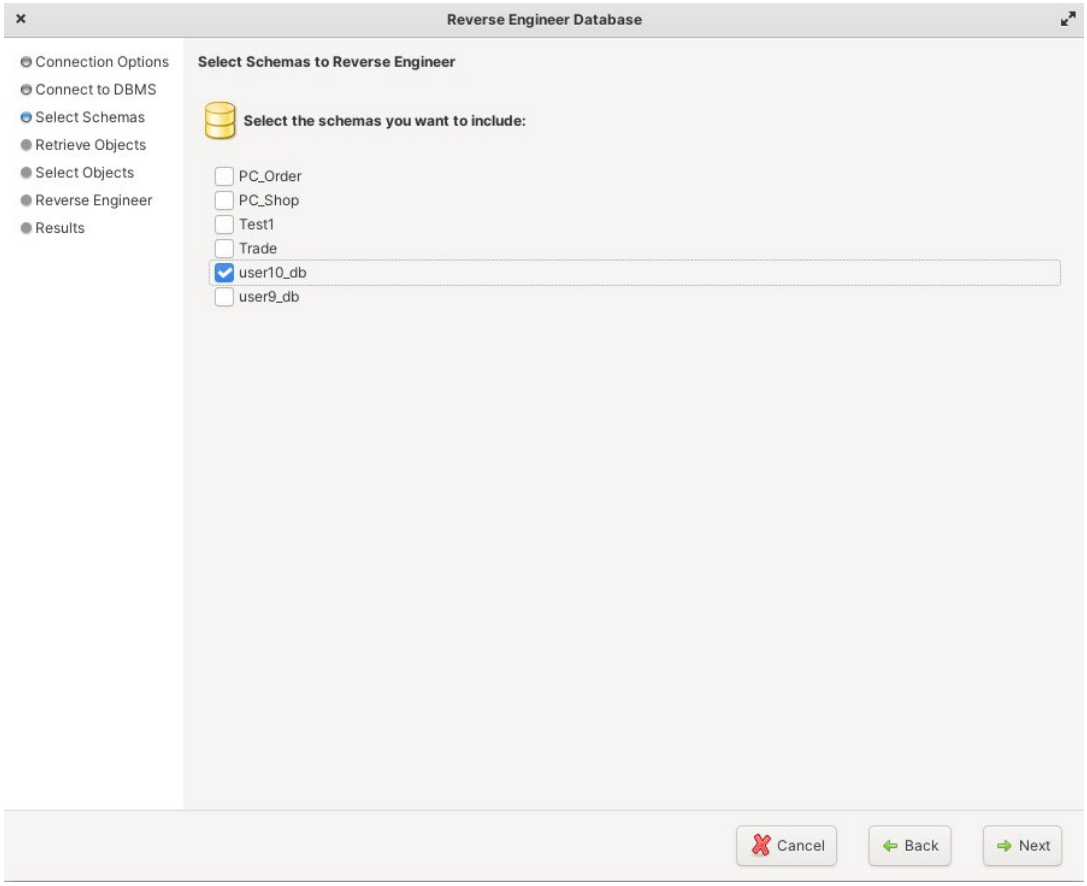

Рисунок 17 – Выбор БД, для которой будет проводиться построение схемы

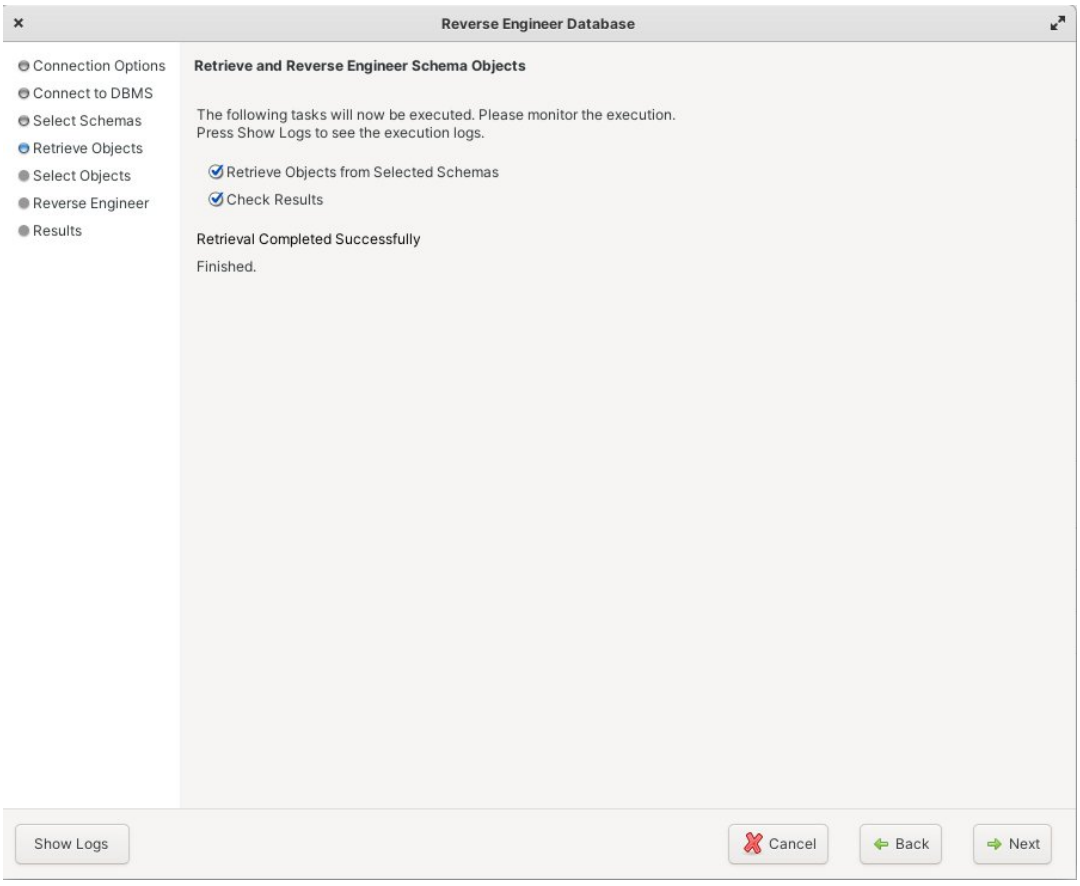

Рисунок 18 – Подтверждение успешного выбора БД

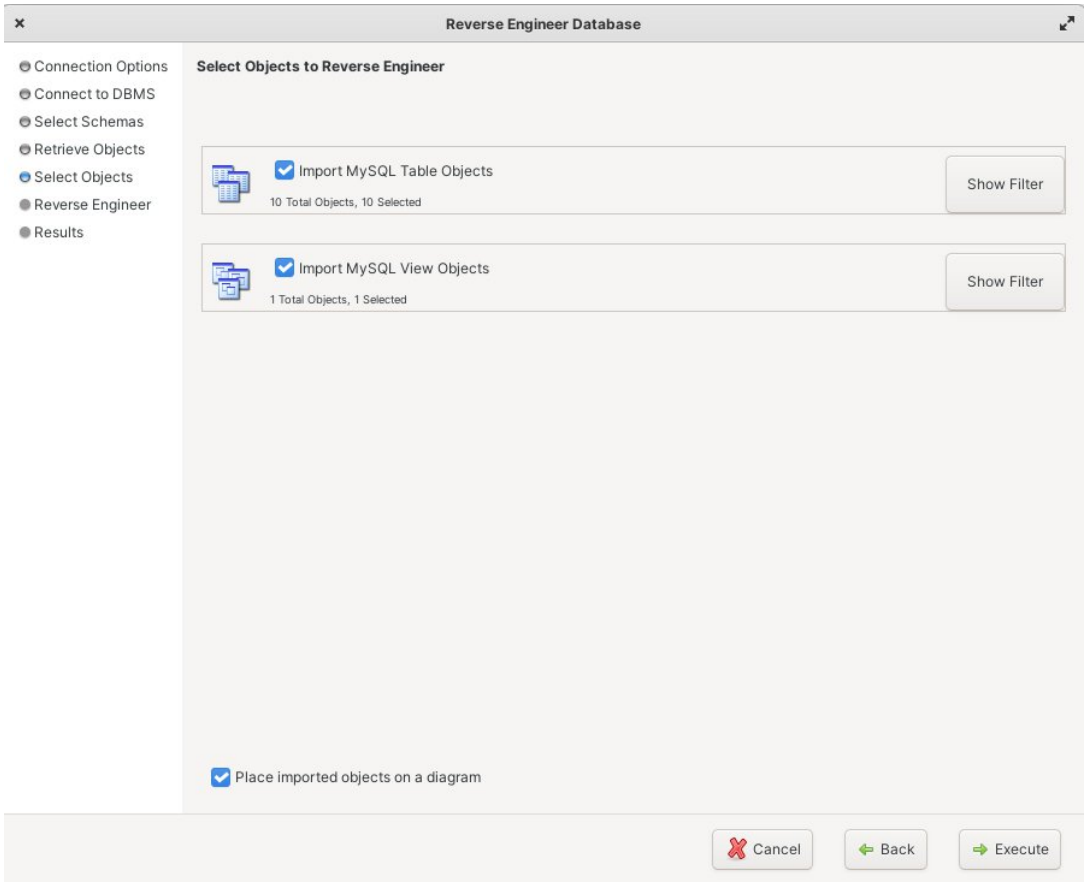

Рисунок 19 – Выбор объектов выбранной БД для построения схемы данных

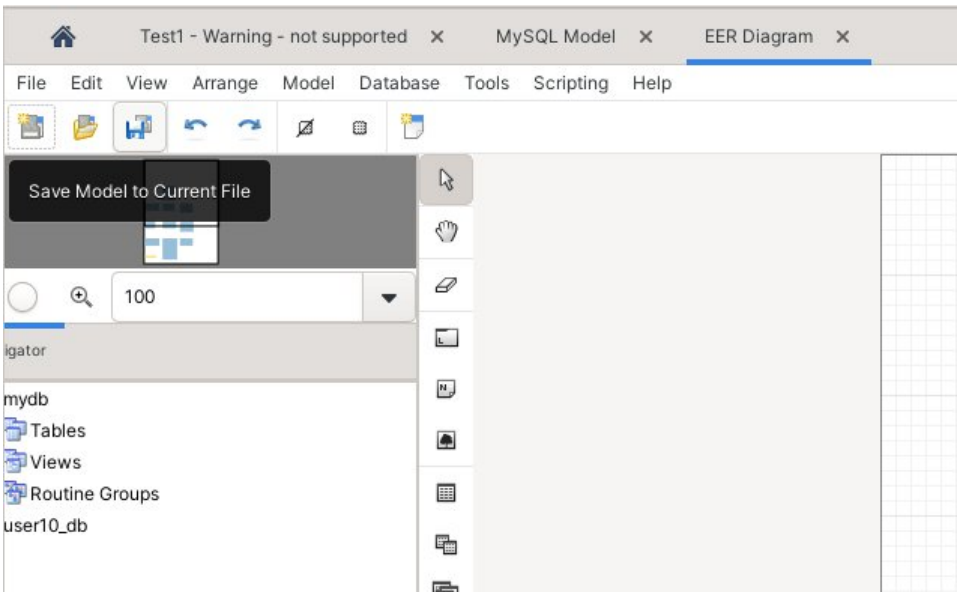

Рисунок 20 – Сохранение построенной схемы в отдельном файле

По сути, подготовительная часть готова, осталось заполнить справочники значениями, добавить несколько пользователей с разными ролями, и разными данными для авторизации, соответственно, и можно приступать к основному импорту данных.

Я не знаю, в каком количестве и в каком качестве будут предоставлены данные для импорта, поэтому я покажу на одном примере импорта заявок, каким образом это делается, но сперва – справочники. Как работает команда «insert into %table% (%column1%, %column2%, ...) values (%value1%, %value2%, ...);» я не буду, за этим – в гугл,

единственное, что я скажу, что поля с модификатором AI не перечисляются, и данные в такие поля не направляются, СУБД сама управляет значениями этих полей. Если у каких-то из ваших полей не стоит модификатор NN, и вам не нужно прямо сейчас добавлять туда значение, то такое поле тоже пропускается из перечня, при создании новой записи такому полю будет автоматически присвоено значение NULL.

Больших сложностей с наполнением справочников быть не должно, для общего удобства я перечислю те значения для следующих таблиц, которые счел для себя приемлемыми:

– UserRole: оператор, мастер, клиент;
– DeviceType: монитор, системный блок, ноутбук, нетбук, смартфон, периферия, видеокарта, материнская плата;
– TypeDefect: «не включается, не работает», «включается, не работает», «включается, работает со сбоями», «плохо заряжается/не держит зарядку», «шумно/медленно работает», «механические повреждения», «не определено»;
– OrderPriority: низкая, обычная, высокая;
– OrderStatus: принята, на выполнении, выполнена.

Для того, чтобы просмотреть первые 1000 записей, хранящихся в какой-нибудь таблице, необходимо выбрать нужную таблицу в обозревателе объектов, и, вызвав контекстное меню, выбрать пункт «Select Rows – Limit 1000». Если же в процессе работы потребуется изменить структуру уже существующей таблицы, то пункт все того же контекстного меню «Alter Table» – к вашим услугам, эти пункты меню продемонстрированы на рисунке 21.

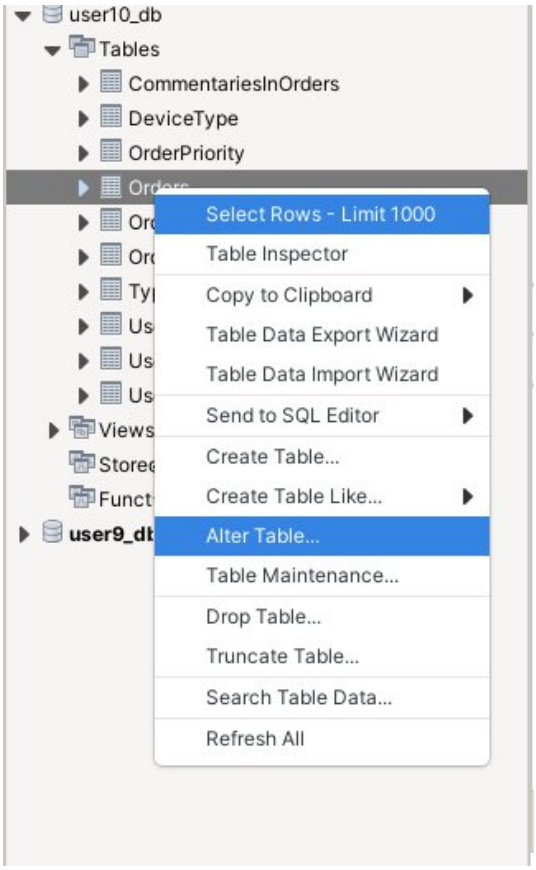

Рисунок 21 – Пункты контекстного меню от вызова с таблицы «Orders»

Теперь – импорт данных, завершающая часть в этой главе. Я понятия не имею, как будут выглядеть настоящие файлы на импорт в реального задания для демоэкзамена, поэтому придется немного сымпровизировать. Пример файла .xlsx на импорт представлен на рисунке 22.

A	B	C	D	E	F	G	H	I	J
Номер заявки	Дата приема заявки	Устройство	Клиент	Тип устройства	Описание	Тип поломки	Приоритет	Статус	Дата закрытия заявки
34	10.11.2023	Dell Inspire 470X-44eDa	Слопырев Анатолий Иванович	Системный блок	Не включается, запах гари из блока питания	Не включается, не работает	Высокий	Принята	
5	10.11.2023	Gigabyte GTX 690	Калашников Антон Эдуардович	Видеокарта	Отвал gpu, артефакты	Включается, работает со сбоями	Обычный	Принята	

Рисунок 22 – Исходный файл для импорта в таблицу «Orders»

Как можно видеть, в файле используются данные фактические, а структура таблицы «Orders» подразумевает использование внешних ключей на эти значение, которые хранятся в других таблицах. Итак, первым делом, мне необходимо сопоставить значения из таблицы Excel с ключами таблиц БД, допустим, начнем с первого попавшегося – с клиентов. На рисунке 23 продемонстрированы текущие данные записей таблицы Users.

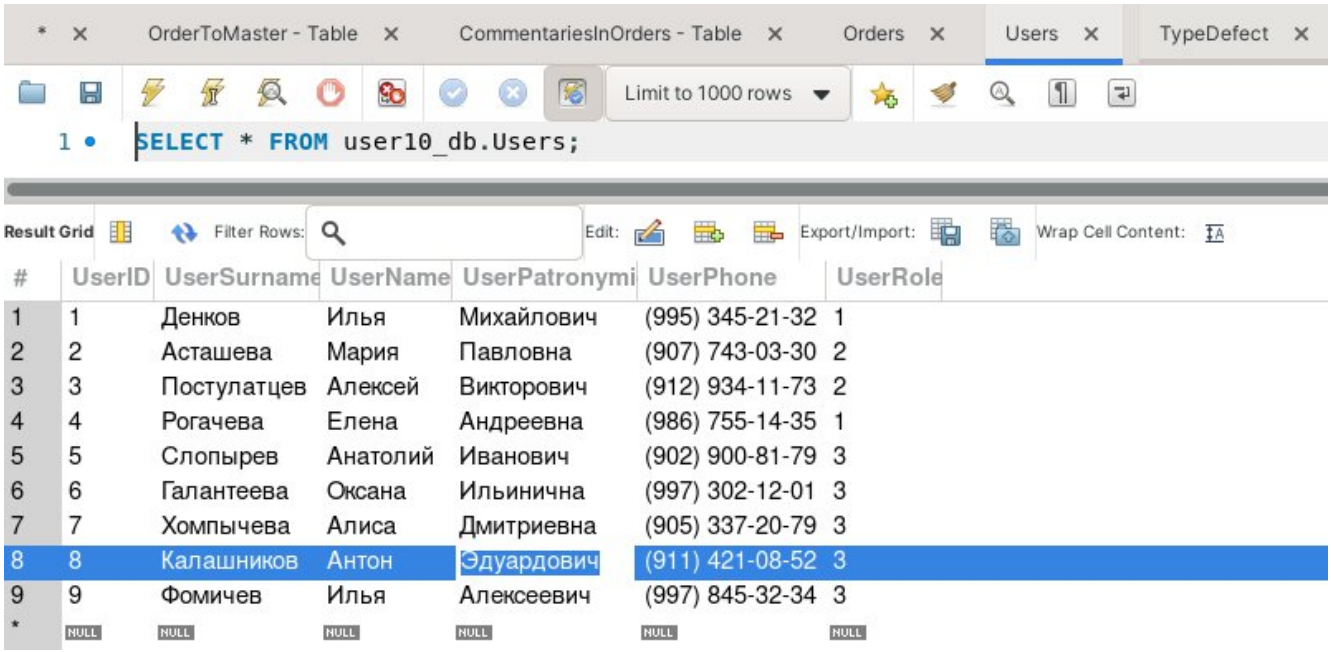

Рисунок 23 – Содержимое таблицы «Users»

У клиентов роль пользователя обособляется значением «3», и, ВНЕЗАПНО, фамилии Слопырева и Калашникова идентифицируются в БД как клиенты, а значения уникальных идентификаторов по этим записям равняются «5» и «8». Так я возьму эти значения, и заменю ими полные ФИО в таблице Excel. И так – для каждого справочника. А вот поле «Номер заявки» из этой таблицы меня не устраивает полностью, поскольку на это поле в таблице БД «Orders» установлен модификатор AI, поэтому этот столбец я полностью удалю. Также необходимо помнить, что формат даты для SQL-синтаксиса есть «гггг-ММ-дд», т.е. 4 цифры года, 2 цифры месяца и 2 цифры дня. И именно в таком порядке, и нет, не «какая разница». Есть разница, и ее нужно соблюдать. Подготовленная к импорту таблица Excel представлена на рисунке 24.

A	B	C	D	E	F	G	H	I
Дата приема заявки	Устройство	Клиент	Тип устройства	Описание	Тип поломки	Приоритет	Статус	Дата закрытия заявки
2023-11-10	Dell Inspire 470X-44eDa	5	2	Не включается, запах гари из блока питания	1	3	1	
2023-11-10	Gigabyte GTX 690	8	7	Отвал gpu, артефакты	3	2	1	

Рисунок 24 – Подготовленный файл для импорта в таблицу «Orders»

Дело осталось за малым. Сохраняем подготовленный файл в формате .csv, т.е. файл с разделителем-запятой, простые .xlsx файлы MySQL Workbench, увы, не обрабатывает. В обозревателе объектов БД выбираем таблицу «Orders», вызываем контекстное меню и выбираем пункт «Table Data Import Wizard», как показано на рисунке 25.

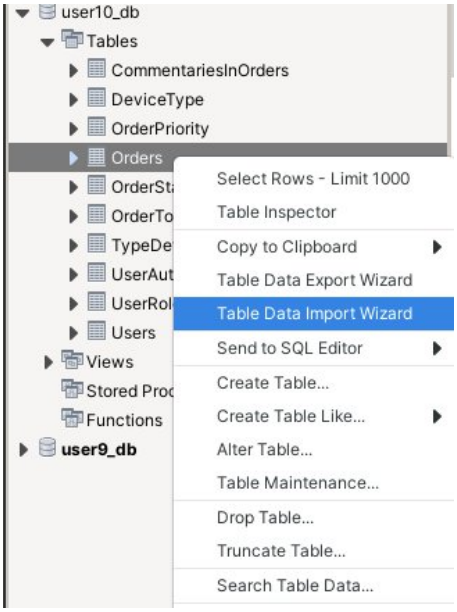

Рисунок 25 – Вызов мастера импорта от таблицы «Orders»

Выбираем нужный файл для импорта (см. рисунок 26), и жмем «Next». Поскольку мы через обозреватель объектов явно указали, в какую таблицу будем импортировать, то кнопку «Next» нужно спамить до тех пор, пока вы не перейдете на форму отношений колонок к полям таблицы. В этом окне вам необходимо установить соответствие полям таблицы «Orders» колонкам из файла на импорт. Одновременно можно ограничить импорт данных в некоторые поля, например, отвечающие за основные ключи, которые управляются самой СУБД. Интерфейс вполне понятный, слева – колонки .csv файла, справа – выпадающие списки, включающие в себя наименования поле выбранной таблицы. Устанавливаем нужные соответствия и получаем следующий набор импорта, представленный на рисунке 27.

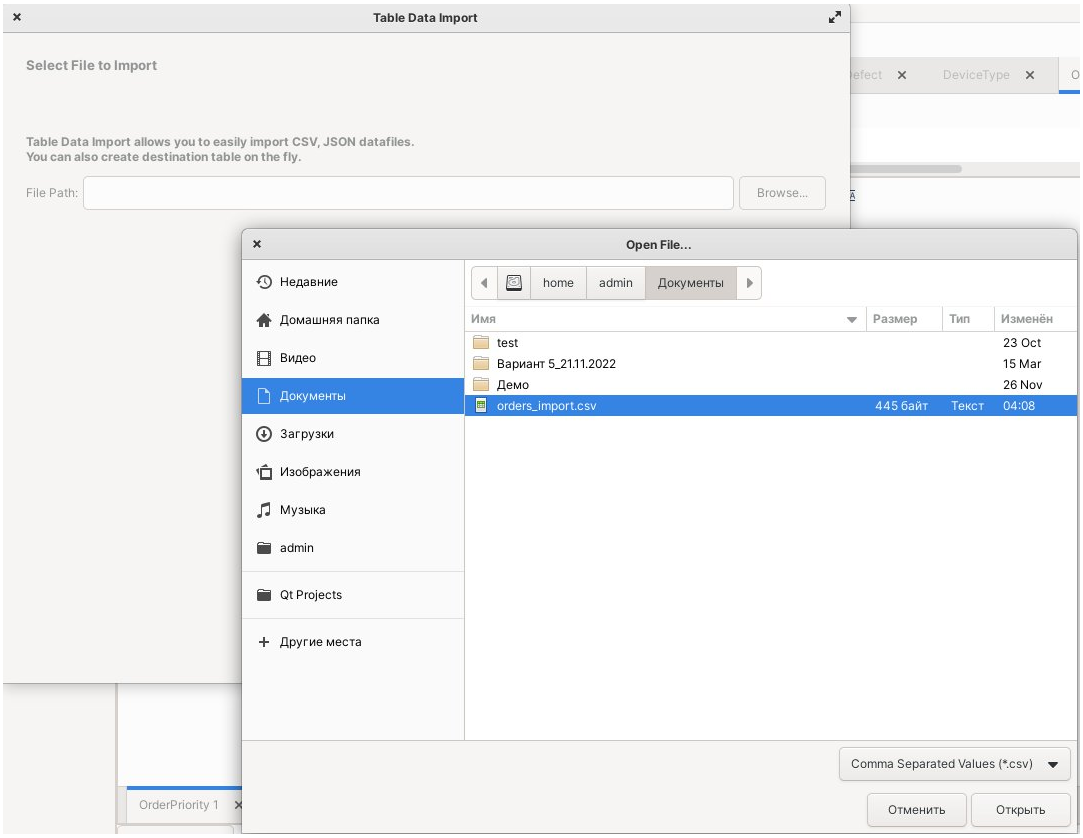

Рисунок 26 – Выбор .csv файла для импорта в таблицу «Orders»

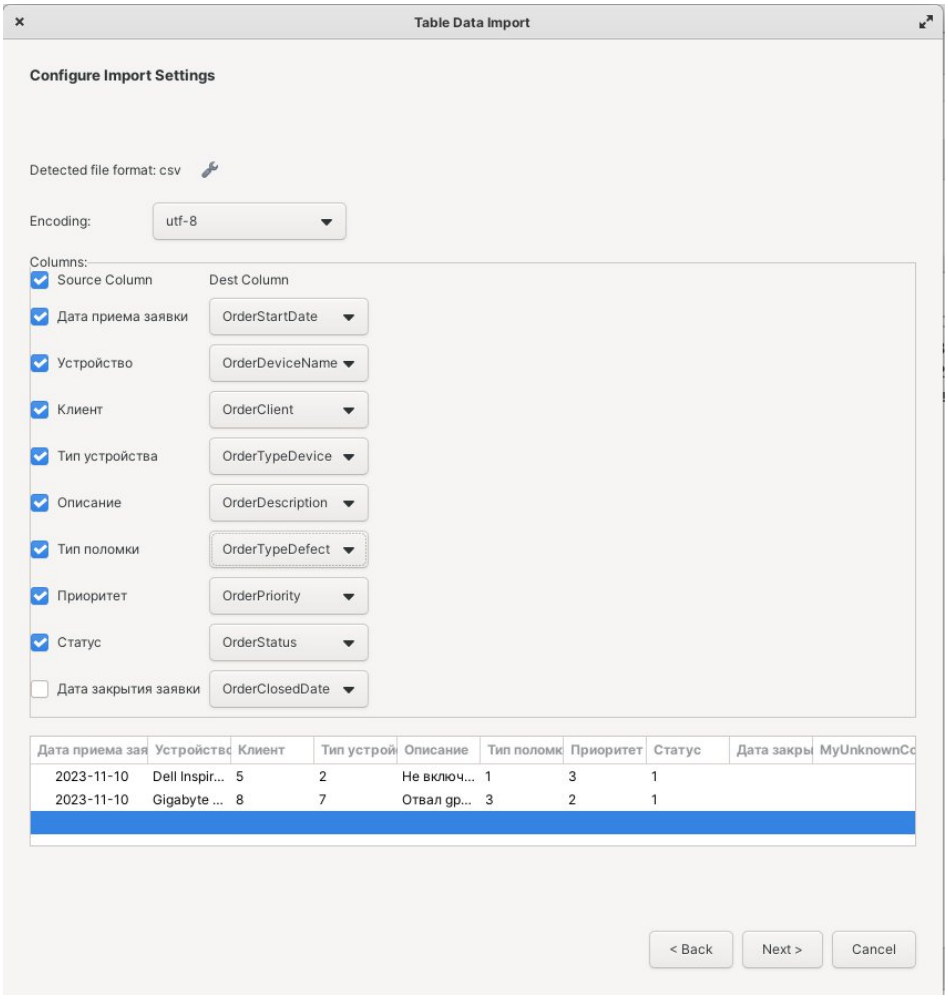

Рисунок 27 – Импорт данных из .csv файла в таблицу «Orders»

Если вы все сделали правильно, то мастер импорта данных отобразит вам следующее окно, представленное на рисунке 28. Импорт данных в таблицу «Orders» завершен и продемонстрирован на рисунок 29.

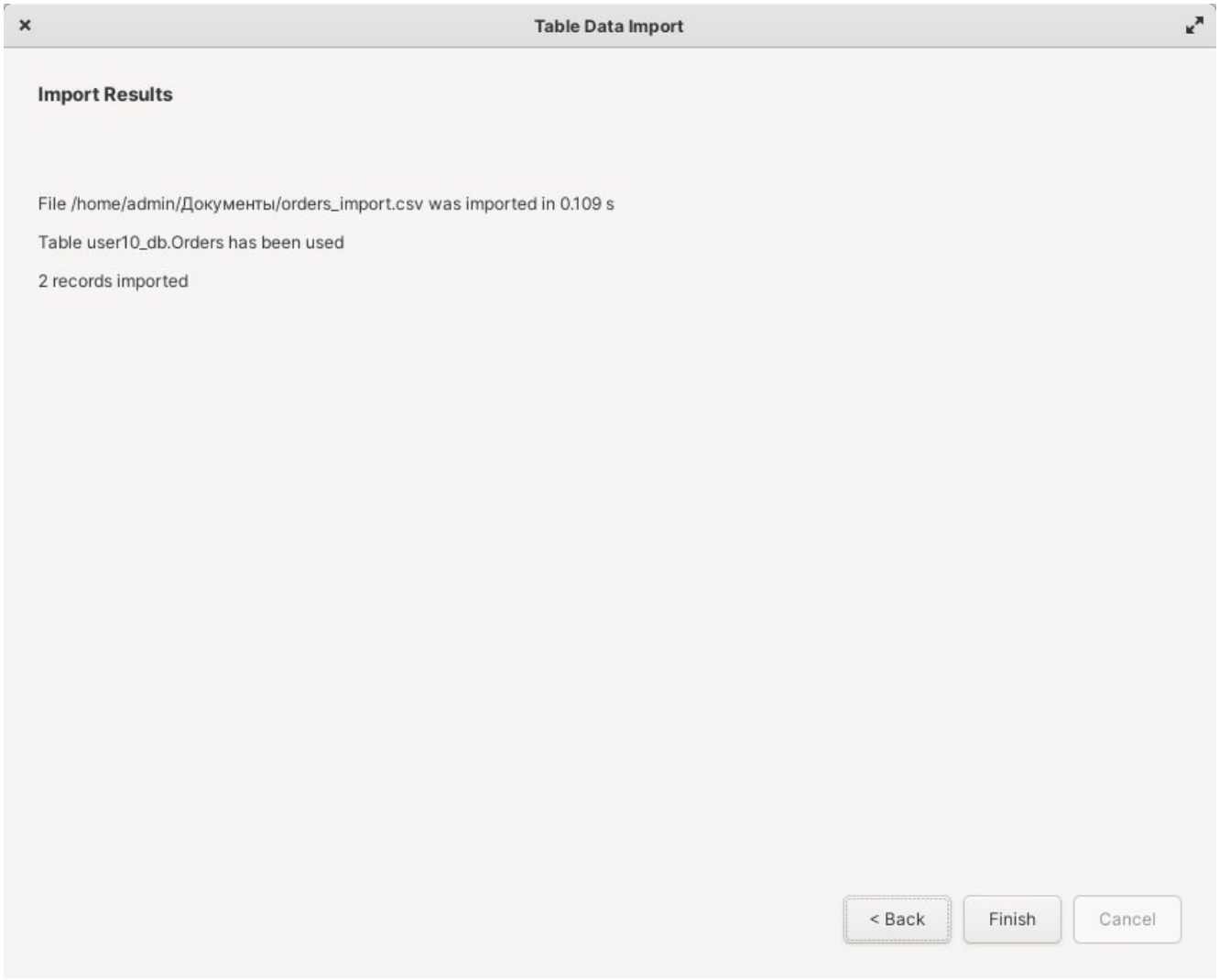

Рисунок 28 – Импорт данных из .csv файла в таблицу «Orders»

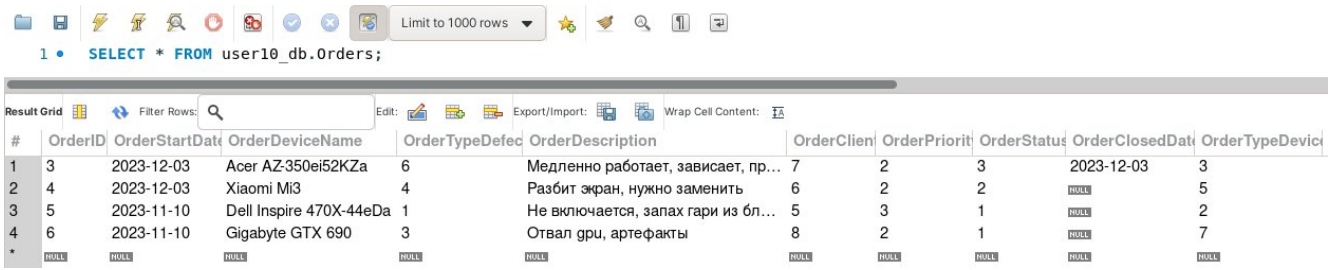

Рисунок 29 – Содержимое таблицы «Orders» после импорта

Сразу хочется сделать небольшую ремарку. Я терпеть ненавижу синтаксис MySQL, по сравнению с божественным T-SQL – это просто небо и земля, как пирожок с капустой из столовки и «Двойная пепперони». Отдельно хочу отметить полное отсутствие нормального дизайнера представлений, какой имеется в SSMS. Не воодушевляет сидеть вручную набивать запросы, когда у тебя нет особо времени, и это даже никак не поощряется. Жаль, что у Microsoft нет своих решений SQL Server 2019 и Management Studio под Linux, очень жаль, но тут хотя бы объективно понятно, почему.

ГЛАВА III. Проектирование и разработка приложения

Сложная для разбора тема. Пожалуй, стоит сразу обозначить те аспекты программирования, без знания которых хотя бы минимальном уровне дальнейшее чтение этой методички попросту бессмысленно, вы просто ничего не поймете, откуда что берется, зачем оно происходит, и по каким принципам работает. Бессмысленно просто механически заучивать большие объемы программного кода без искреннего понимания того, как он устроен и как работает, по сути, если вы могете в C++, то вам и заучивать ничего и не нужно, руки сами код напишут, а мозг придумает, как заставить его работать. Итак, пойдем по списку крайне необходимых для знания тем:

1. Общее построение программ на языке C++. Директивы препроцессора #include, #define, #ifndef, #endif, #pragma once. Сборка программы от исходного кода в исполняемый файл. Стандартное пространство имен std.
2. Стандартные типы данных C++, объявление и инициализация переменных.
3. Унарные, бинарные и тернарный операторы.
4. Условные операторы, оператор безусловного перехода.
5. Оператор множественного выбора.
6. Циклы. Операторы break и continue. Бесконечные циклы. Цикл перебора элементов коллекций.
7. Обработчик исключений try...catch. Основные типы исключений, оператор throw;
8. Одномерные и двумерные статические массивы.
9. Указатели в массивах. Динамические массивы.
10. Векторы. Основные возможности векторов C++. Итераторы.
11. Строки C++. Работа со строками.
12. Функции. Прототипы и реализации. Термины «аргумент» и «параметр». Определение функций в программном коду C++.
13. Области видимости переменных.
14. Многомодульность в C++. Работа с заголовочными файлами и файлами исходного кода. Ключевое слово static.
15. Отладка программ. Точки остановки. Понятия «шаг с заходом» и «шаг с обходом». Просмотр значений локальных переменных. Стек вызовов.
16. Классы в C++. Конструкторы и деструкторы класса. Статический и динамические экземпляры классов. Понятия «поле класса», «метод класса». Указатель this.
17. Полиморфизм. Перегрузка конструкторов, методов класса.
18. Инкапсуляция. Модификаторы доступа. Реализация аксессоров.
19. Наследование. Полное, неполное, частичное наследование. Виртуальные функции.
20. Многомодульность в ООП. Обращение к базовому классу при реализации многомодульности.

Итак, всего 20 тем для качественной зубрежки. По сути, это все темы из курса основ программирования и алгоритмизации за 1-2 курс, смотря на базе скольких классов образования вы поступали в колледж. Прошу заметить, что список тем для изучения крайне поверхностный, и во многом применим вообще к любому языку программирования, но, так получилось, что я говорю только на C-подобных языках, поэтому я могу ошибаться в этой догадке. Если вы поднапряжете свою пятую точку дабы она сидела ровно, при этом

действительно начнете прорабатывать свои пробелы в знаниях, то покрыв их, вы практически будете готовы к написанию демоэкзамена, поскольку все основные, базовые возможности языка C++ вы будете знать и уметь ими пользоваться.

Однако же, я не зря упомянул слово «практически». Осталось познакомиться с фреймворком, который позволит писать не просто консольные программки для лабораторных работ, а вполне себе имеющие вид приложения с графическим интерфейсом пользователя. Таким фреймворком для вас будет являться Qt. Во-первых, те из вас, кто еще застал эпоху C# и Windows Forms, достаточно легко на него перейдут, ну поскольку граней подобия слишком много, нужно будет только привыкнуть к синтаксису C++, и к работе с Qt. А тем, кому этот фреймворк вообще впервые откроет дверь в мир GUI, будет достаточно просто освоить редактор форм, который работает по принципу конструктора, и данная методичка позволит вам быстрее понять принципы работы с объектами Qt, и общие правила построения программного кода на Qt в целом.

Приступим.

Думаю, что чайную церемонию и воспевание всех плюсов сего фреймворка можно смело пропустить, этим займутся другие. Я же сразу приступлю к делу, все повествование будет вестись только по существу. Напомню, что это самоучитель по Qt, а лишь методичка по сдаче демоэкзамена, за более подробными сведениями и академической теории – в учебники, дамы и господа. Поговорим немного про среду разработки Qt Creator.

Открыв среду разработки, создайте новый проект, нажатием на кнопку «Create Project». Мастер создания проектов предложит создать новый по имеющемуся шаблону, вам необходимо выбрать «Прилижение Qt Widgets».

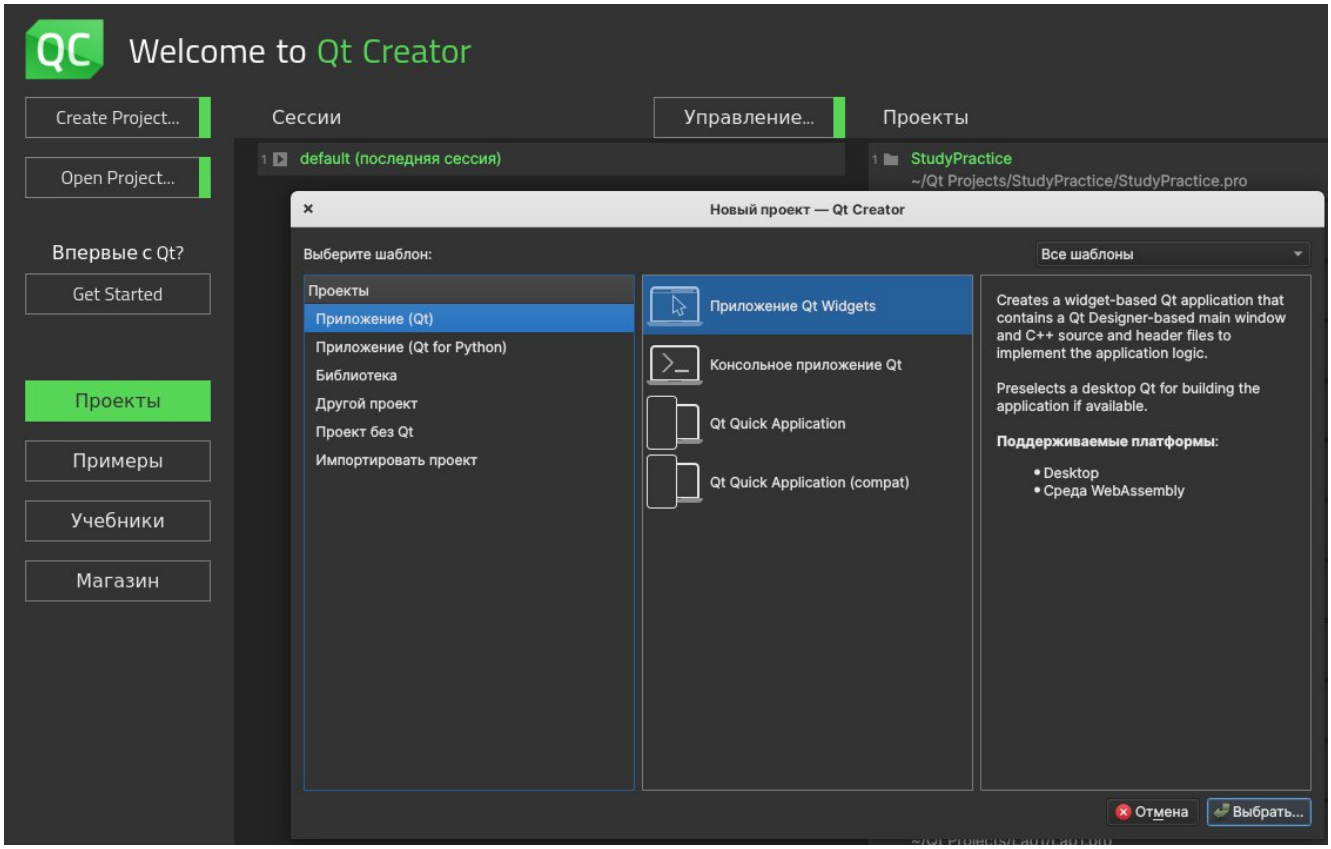

Рисунок 30 – Создание нового проекта

Дайте название вашему проекту, допустим, «XX_AAAAA», где XX – это номер рабочего места, за которым вам предстоит восседать, а AAAAA – это ваша фамилия, на латинице, понятное дело. Систему сборки проекта оставьте без изменений (по умолчанию – qmake). Далее мастер предложит создать новый класс, который будет являться первой формой в вашем проекте, и той формой, которая будет запускаться при старте вашей программы.

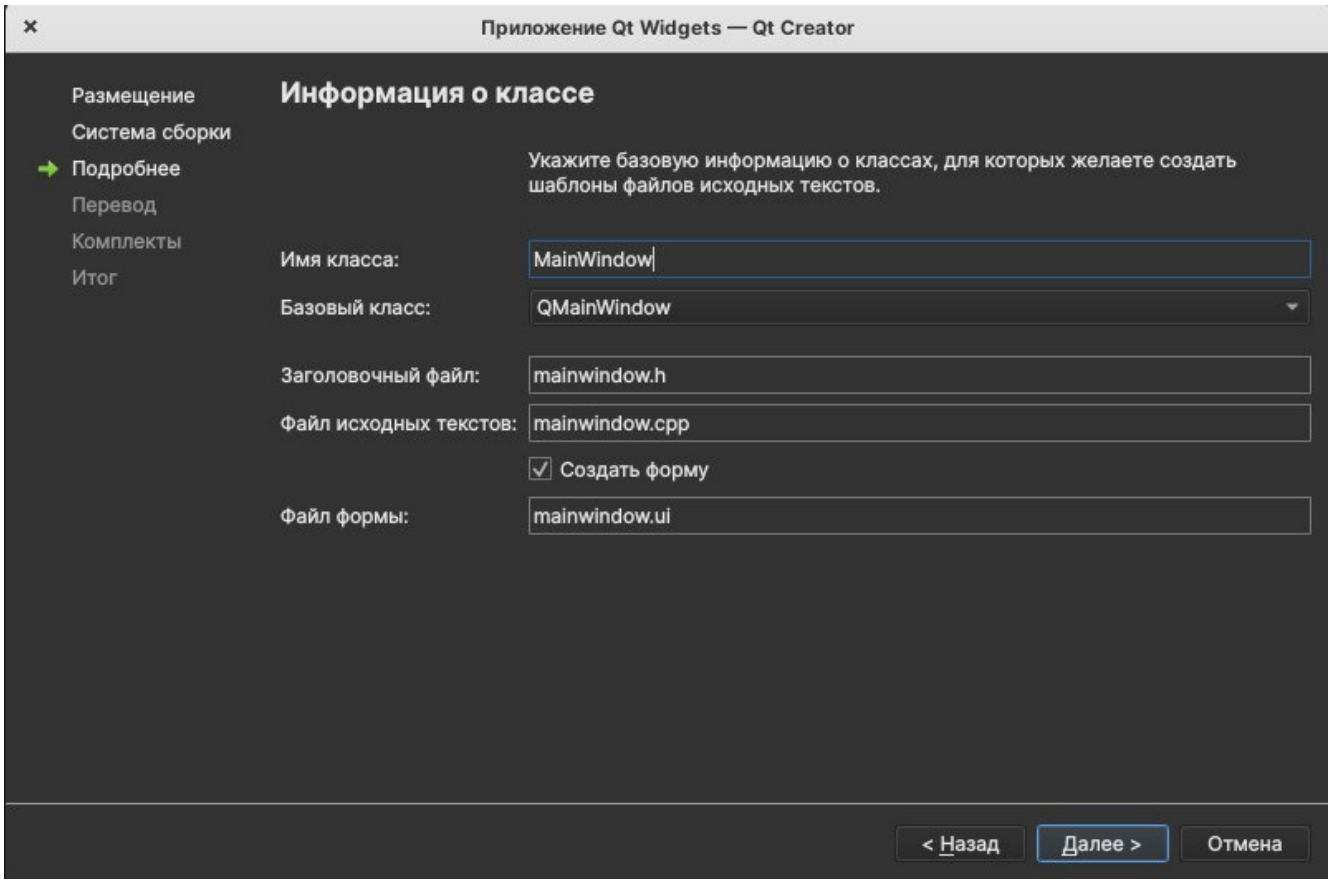

Рисунок 31 – Создание первого класса внутри проекта

Наименования классов необходимо указывать в стиле CamelCase, само название должно кратко характеризовать суть класса. По заданию, пользователь должен авторизироваться для работы с системой, вами спроектированной, поэтому первой формой будет как раз форма авторизации, так ее и назовите – AutorizationForm. Требования к разработке ПО демоэкзамена гласят, что стандартные наименования классов и объектов форм не должны иметь стандартных значений, наподобие Form1, Form2, MainWindow, MainForm, и так далее, подойдите к этому вопросу с умом. Предложение мастера добавить файл переводов вежливо отклоните, а систему сборки не трогайте, она настраивается отдельно, не вами, и не вам ее менять, оставьте как есть. После успешного создания проекта вашему взору предстанет основное рабочее пространство среды разработки, изображенное на рисунке 32, остановимся на нем подробнее.

Для удобства пояснений я сделал на рисунке отметки нумерации, ссылаясь на которые, мне будет проще вам объяснить, для чего каждое отдельное окно или кнопка предназначена. Если так посмотреть, то те из вас, кто работал в среде Visual Studio, и так быстро разберутся в интерфейсе Qt Creator, поскольку ничего сложного в нем совершенно нет.

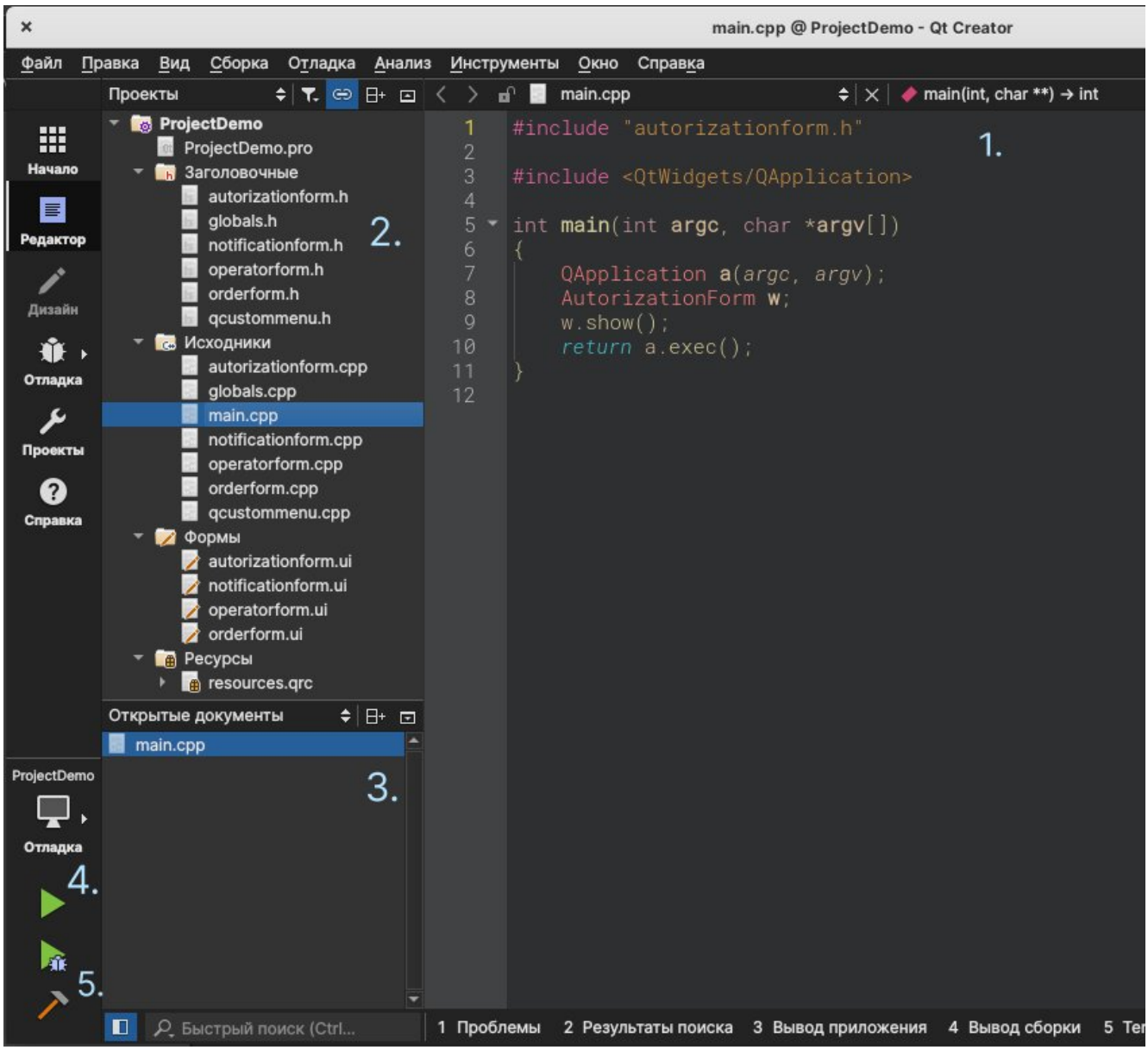

Рисунок 32 – Рабочее пространство среды разработки Qt Creator

1. Рабочее пространство кода. ВНЕЗАПНО, в нем пишут код. Слева отображается нумерация строк, и, ВНЕЗАПНО, тоже слева от нумерации есть немного пустого, казалось бы, пространства. Так вот, напротив номера строки с левой стороны по нажатию левой кнопки мыши устанавливаются точки остановки программы для отладки.

2. Обозреватель проекта. Обозреватель представляет собой иерархичную структуру каталогов и файлов, которые относятся к одному проекту. Добавление новых элементов в проект происходит также через обозреватель, вызовом контекстного меню от нужного каталога, и выбором пункта меню «Add New».

3. Как можно заметить по названию – список открытых документов в текущей сессии работы, позволяет быстро переключаться между текущими файлами проекта без необходимости искать их в обозревателе. Между заголовочным файлом и файлом исходного кода одного модуля можно быстро переключаться с помощью клавиши F4.

4. Кнопка запуска программы. Можно использовать комбинацию клавиш Ctrl+R, либо F5, но она запускает именно отладку.

5. Кнопка сборки программы.

Теперь ознакомимся с редактором форм. Опять же, я отмечу основные модули окна соответствующей нумерацией.

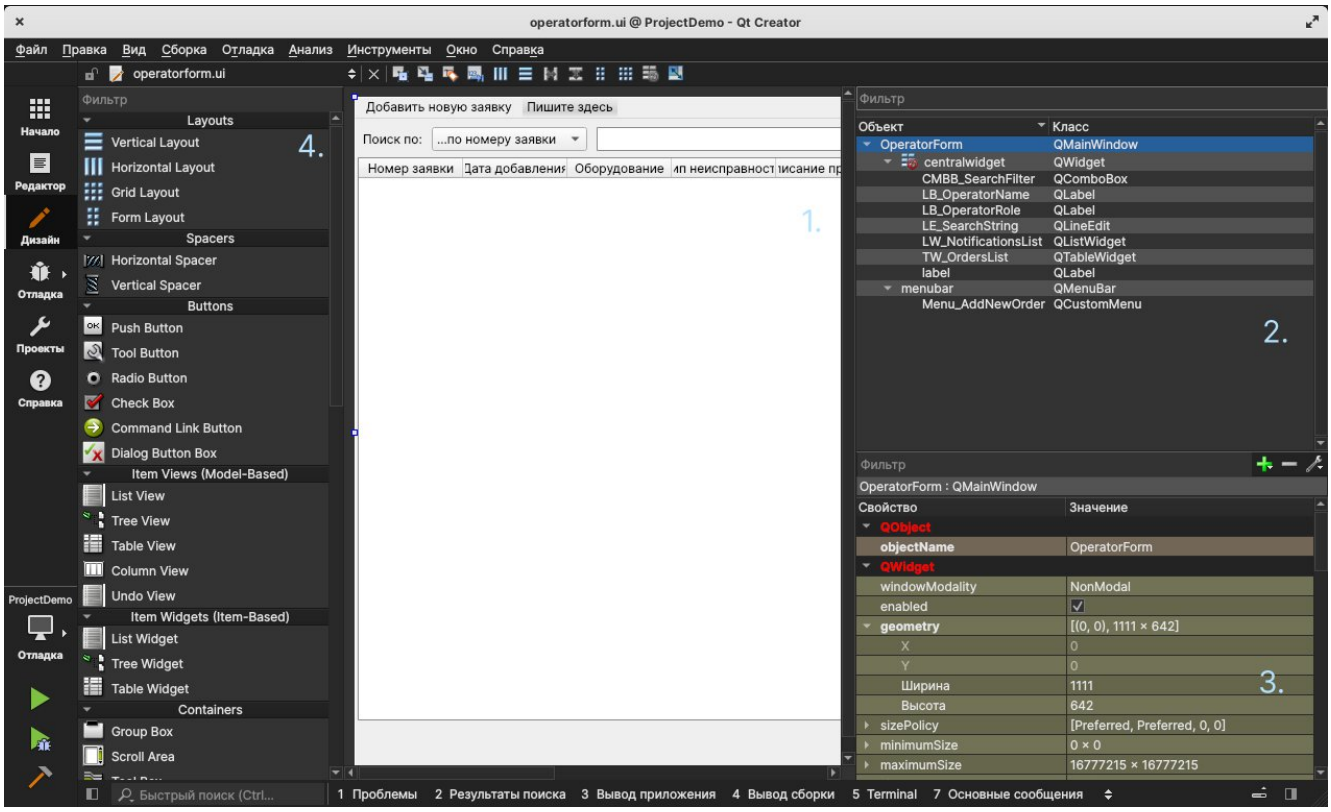

Рисунок 32 – Рабочее пространство дизайнера форм Qt Creator

Теперь ознакомимся с редактором форм. Опять же, я отмечу основные модули окна соответствующей нумерацией.

1. Основное пространство макета формы. Позволяется редактировать форму по своему желанию.
2. Обозреватель объектов управления на текущем макете. Представлен в виде иерархического списка, каждый уровень представляет собой вложенные в определенный контейнер элементы управления. Любой элемент управления либо сам является контейнером, либо вложен в него. При этом, сам контейнер также может быть вложен в другой.
3. Обозреватель свойств элемента управления. К сожалению, большая часть свойств, которые можно изменить для элемента, недоступна из этого обозревателя и настраивается через вызовы аксессоров в программном коде. Все свойства вложены в выпадающие списки, порядок списков имеет значение, поскольку представляет собой структуру наследования классов для выбранного элемента управления.
4. Панель элементов управления. В ней расположены все те виджеты, которые вы можете размещать на своей форме, моделирование дизайна формы представляет собой конструктор: объекты из этой панели перетаскиваются мышью на форму, и далее – видоизменяются по желанию.

В дизайнере форм нет ничего сложного, и те из вас, кто имел опыт в работе с Visual Studio, быстро найдут подобия и легко адаптируются как к новому интерфейсу, так и возможностям самого дизайнера. Краткий экскурс закончен, теперь же нужно перейти к изучению самих элементов управления и механике взаимодействия между ними.

Все элементы управления (далее – виджеты, ибо они именно так и называются) наследуются от основного базового класса – QObject, т.е. виджеты – это объекты, в первую очередь. Наследование этого класса дает виджетам основное свойство – objectName, которое является псевдонимом объекта, по которому к нему можно будет обратиться из программного кода. Нейминг (наименование) объектов есть крайне важная вещь, поскольку чем больше виджетов на форме, тем сложнее держать в голове все их псевдонимы, а постоянно переключаться в дизайнер, чтобы вспомнить «как я там назвал эту кнопку» – не круто, поскольку это просто трата времени на ровном месте. Для нейминга виджетов я использую префиксную форму записи псевдонима, т.е. каждый объект будет иметь такой префикс, который соответствует классу этого объекта, чтобы однозначно идентифицировать его в коде как «кнопка», «поле ввода», «выпадающий список» и т.д., к тому же, основное свойство префиксов – уникальность: два разных класса не должны использовать один и тот же префикс, поскольку это будет вносить двойственность в программный код, и дополнительно усложнит работе над ним, именно это и гарантирует однозначность идентификации объектов в коде. Помимо прочего, Qt Creator умеет «подсказывать» наименования, так что просто написав префикс определенного класса, вам всплывет подсказка в виде выпадающего списка, в котором будут представлены все виджеты, использующие этот префикс, а уже по самому псевдониму можно будет легко определить и выбрать нужный в текущий момент. Сплошные плюсы, одним словом. В таблице ниже будут представлены все элементы управления, которые вам нужно освоить, а их свойства – выучить, поскольку они будут активно использоваться в работе, остальные прочие же вовсе не будут задействованы, поэтому их заучивать для сдачи демоэкзамена и не нужно, просто лишняя информация.

Виджет	Базовый класс виджета	Описание	Свойства		Префикс
			Некоторые повторяющиеся свойства (не общие абсолютно для всех):		
			enable	Определяет, доступен ли элемент для взаимодействия	
			visible	Определяет видимость элемента в контейнере	
			objectName	Псевдоним, имя элемента	
			windowTitle	Задает название модуля	
			windowIcon	Задает иконку модуля	
			palette	Определяет цветовую гамму элемента	
			text	Определяет текст, находящийся внутри элемента	
			font	Определяет шрифт текста внутри элемента и его начертание	
			frameShape	Определяет тип обводки контура вокруг элемента	
			geometry	Определяет размер элемента и его	

				расположение по X, Y координатам внутри контейнера	
			maxLength	Задает максимальную длину возможного текста, вводимого в элемент	
			readOnly	Задает параметр «только чтение» для элемента	
			placeholder Text	Устанавливает замещающий текст внутри элемента, заглушку текста в пустом элементе	
Label	QLabel	Простая надпись	pixmap	Определяет изображение внутри надписи	LB_
			alignment	Определяет выравнивание текста внутри надписи	
Line Edit	QLineEdit	Поле ввода текста в одну строку	inputMask	Задает маску ввода для текста	LE_
			echoMode	Задает скрытие символов текста на точки: Normal – без скрытия, Password – скрывает символы	
			readOnly	Задает параметр «только чтение» для элемента	
Push Button	QPushButton	Простая кнопка	icon	Задает иконку для кнопки	BTN_
			flat	Определяет объемность кнопки, делая ее похожей на надпись	
Check Box	QCheckBox	Флажок, простая «галочка»	checked	Определяет, поставлена ли галочка внутри элемента, или нет	CHKB_
Combo Box	QComboBox	Выпадающий список	editable	Определяет, можно ли вручную вписывать текст внутрь элемента	CMBB_
			currentText	Определяет текст, отображаемый в выпадающем списке	
			currentIndex	Определяет индекс выбранного элемента в в выпадающем списке	
Text Edit	QTextEdit	Поле ввода большого текста	plainText	Определяет текст, находящийся внутри элемента, этот элемент не имеет свойства text	TE_
Table Widget	QTableWidget	Табличный вывод данных	selection Mode	Определяет возможность выбора нескольких строк в	TW_

				таблице: NoSelection – выбор строк недоступен, SingleSelection – выбор только одной строки, MultiSelection - выбор нескольких строк	
			selection Behavior	Определяет тип выделения при выборе в таблице: SelectItems – подсветка только одной ячейки, SelectRows – подсветка всей строки	
			horizontal HeaderStretch LastSection	Определяет, можно ли растянуть ширину последнего столбца под свободное пространство в шапке таблицы	
			verticalHeader Visible	Задает видимость заголовков строк	
			horizontal HeaderVisible	Задает видимость заголовков столбцов	
			rowCount	Кол-во строк в таблице	
			columnCount	Кол-во столбцов в таблице	
Frame	QFrame	Простой контейнер элементов	Определяется базовым набором свойств		FR_
Tab Widget	QTabWidget	Элемент групповой сортировки элементов во вкладках	currentTab Text	Определяет надпись текущей вкладки	TAB_
			currentTab Name	Определяет псевдоним текущей вкладки	
Date Edit	QDateEdit	Элемент выбора даты	date	Определяет отображаемую дату внутри элемента	DE_
			minimumDate	Определяет минимальную дату для ввода	
			maximumDate	Определяет максимальную дату для ввода	
Menu	QMenu	Пункт панели меню приложения	title	Определяет надпись пункта меню	MENU_
			icon	Задает иконку для пункта меню	

Разумеется, это далеко не весь список тех виджетов, с помощью которых вы можете настраивать GUI вашего приложения, однако, для сдачи демоэкзамена большего не потребуется. Приведенный список свойств также является неполным, более подробно свойства будут рассматриваться уже на конкретных примерах. Для работы с каждым свойством внутри программного кода предусмотрены свои аксессоры get (в некоторых

случая - is) и set, они вызываются от объекта в виде простых функций, поскольку простых свойств, как в C#, классы в C++ не имеют.

Как было сказано ранее, форма, которая должна быть отображена при запуске приложения, есть форма авторизации, ее класс нами был ранее создан, поэтому сразу приступаем к дизайну макета.

В представленном техническом задании ничего не сказано про внешний вид любого из требуемых модулей, поэтому не нужно ничего подумывать и заниматься лишними «украшательствами», реализовывайте макет в минимальном и достаточном виде, так, чтобы требуемый функционал мог быть реализован, и не более того. Для формы авторизации потребуется всего два поля ввода (для логина и пароля), а также кнопка самой авторизации. На рисунке 33 представлен внешний вид макета формы авторизации.

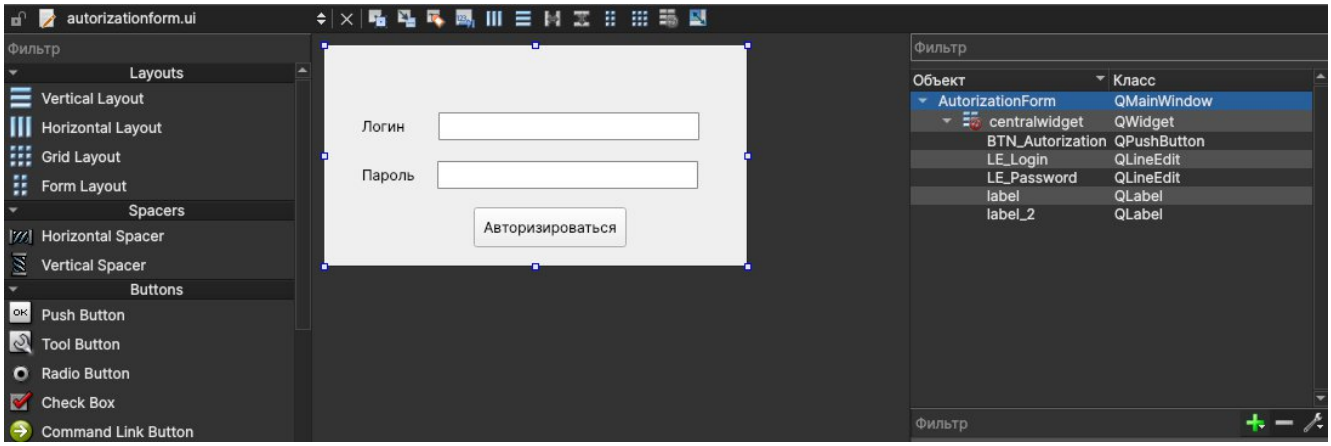

Рисунок 33 – Макет формы авторизации

Осталось сообразить, каким образом настроить работу этого модуля. Исходим из того, что пользователь должен ввести данные в поля ввода и нажать кнопку «Авторизироваться», после чего, программа должна проверить поля на заполненность, а после этого – обратиться к БД, дабы получить данные о пользователе относительно его логина и пароля и обработать результат запроса. Основная проблема здесь – вопрос взаимодействия пользователя и программы. Пользователь постоянно взаимодействует с элементами управления на форме, и эти события должны обрабатываться в программе, реализуя ее программную логику. В Qt для этих нужд предусмотрен механизм сигналов и слотов, с которым мы сейчас и познакомимся.

Для начала разберемся с терминологией. Сигналом в Qt обозначают событие, произошедшее над элементом управления. Допустим, пользователь нажал на кнопку. Произошло ***событие***, и элемент управления, который был нажат, «понимает», что над ним было совершено действие, и посылает ***сигнал*** о том, какое именно событие произошло. Этот сигнал может быть «перехвачен» программой, и адресован в определенный участок программы, который будет отвечать за то, чтобы обработать этот сигнал, т.е. произвести некоторые действия, которые должны быть выполнены только при получении определенного сигнала. Такая область программы называется ***слотом***, и оформляется как простой метод класса со своей маркировкой, по которой программа как раз и понимает, что это не просто рядовой метод, а именно ***обработчик события***. Механика проста и понятна, на самом деле, те из вас, кто работал с Visual Studio, найдут явные параллели, однако, за создание нужных обработчиков событий и присоединение элемента

управления, создающего нужное событие к нему, была ответственна сама среда разработки, а в Qt реализация этого механизма полностью возложена на программиста.

Для сдачи демоэкзамена достаточно выучить базовые возможность этого механизма, и мы не будем слишком сильно углубляться в теорию, а лишь отметим важные тезисы, которым вам придется следовать. Сперва определим, какие события программе необходимо регистрировать в модуле авторизации, и это только одно событие – нажатие на кнопку авторизации. Эта кнопка будет посылать сигнал о том, что была нажата, а программа будет должна вызвать соответствующий обработчик события, т.е. слот. Как определить слот для модуля? Для этого необходимо перейти в заголовочный файл класса формы, и в теле класса определить метку «slots:» с модификатором доступа public. Все прототипы методов, которые будут предшествовать этой метке, будут определяться классом как обработчики события, так что формируйте список слотов отдельно от объявления прочих объектов, дабы не поломать структурную логику класса.

Сигналы и слоты соединяются между собой с помощью функции connect(...), которая в ваших работах ПРАКТИЧЕСКИ ВСЕГДА должна вызываться из конструктора класса. Это обеспечит подключение всех элементов к обработчикам в момент создания экземпляра класса через вызов, собственно говоря, конструктора, и при отображении формы на экране пользователя все элементы будут полностью готовы к реализации программной логики. Единственное исключение в ваших работах будет составлять подключение всплывающих уведомлений к слотам класса, но это совсем другая история, которая будет рассматриваться позже, да и я не рассчитываю на то, что вы вообще будете реализовывать этот механизм. Функция connect является перегруженной, и мы будем активно использовать три из них, остановимся на них поподробнее.

Функция connect всегда требует определения основных параметров: объект-адресант, который будет посылать какой-то сигнал, сам сигнал, который программа будет перехватывать, объект-получатель, который будет ответственен за обработку поступившего сигнала, а также слот, находящийся в определении класса объекта-получателя, который, собственно, и будет этот сигнал обрабатывать. Конструкция несколько замысловата, но это позволяет настроить взаимодействие нескольких объектов между собой, которые, при этом, могут быть расположены совершенно в разных модулях, и, получается, в разных классах.

Как я уже сказал ранее, мы будем использовать три перегрузки от этого метода:

```
connect (указатель_на_объект_адресант, SIGNAL(сигнал_от_объекта),
указатель_на_объект_адресат, SLOT(псевдоним_обработчика(аргумент1, ... , аргументN));
connect (указатель_на_объект_адресант, &базовый_класс_объекта::сигнал_от_объекта,
указатель_на_объект_адресат, &базовый_класс_объекта::псевдоним_обработчика);
connect (указатель_на_объект_адресант, &базовый_класс_объекта::сигнал_от_объекта,
[указатель_на_объект_адресат] { псевдоним_обработчика(аргумент1, ... , аргументN ); });
```

Первый вариант вызова функции connect существует со стародавних времен, и ныне считается устаревшим, и программисты им практически не пользуются, предпочитая ему второй или третий варианты перегрузки функции connect, однако, он все еще работает и по сей день, даже в версии Qt 6.5.2. На сегодняшний день используется вызов функции connect через параметры-указатели, что считается более предпочтительным методом, поскольку при использовании указателей у вас не получится «привязать» объект с сигналом, которым он не обладает, а сигнал – к слоту, которого не существует в определении класса

объекта-адресата. Любая попытка бессвязного перечисления параметров функции connect будет считаться как синтаксическая ошибка, и такая программа не будет скомпилирована, а использование первого варианта перегрузки через ключевые слова SIGNAL и SLOT, наоборот, позволяет привязывать что угодно и куда угодно, при этом не вызывая синтаксических ошибок. Код скомпилируется, программа запустится, однако же, элементы управления не будут подавать никаких признаков жизни, и понять, что же тут происходит, будет крайне непросто, поскольку критических ошибок и вылетов программы при таком методе соединения тоже не будет. Третий метод перегрузки повторяет второй, но при этом использует лямбда-функцию для передачи объекта-адресата и слота в функцию connect. Лямбда-функции в connect – необходимость, поскольку при обращении к области класса для определения слота, в этот самый слот нельзя передать параметры, если слот их требует, а такие слоты с параметрами будут нужны по ходу работы, и без них никак не обойтись, это 100% информация. Лямбда-функция позволяет от объекта-адресата вызывать слот, а не сослаться на него, передавая нужные аргументы при вызове, этот способ решает массу проблем, однако, поскольку сей документ не является учебной литературой, то я не буду углубляться в теорию лямбд в C++, вы просто будете ими пользоваться, выучив их конструкцию, вот и все.

Теперь разберемся с терминами «объект-адресант» и «объект-адресат», и как эти объекты представляются в программном коде. Объектами-адресантами будут элементы управления Qt, ведь именно они и генерируют нужные сигналы. Но функция connect требует перечисления объектов через указатели, что внешне создает нам дополнительную проблему: откуда взять указатель на объект, если мы не создавали никаких указателей, а просто перетащили с панели виджет и разместили его на форме. В этот момент в дело вступают ***пространства имен***. Если вы чуть поднаторели в C++, то пространство имен std вам должно быть очень хорошо знакомо. Это пространство имен предоставляет доступ ко всем псевдонимам функций, которые находятся в стандартных библиотеках C++, без прямого или глобального объявления std вы буквально не сможете ничего сделать, даже в консоль «Hello, World!» вывести. Не буду ходить вокруг да около, да метаться полунамеками. При создании элемента управления на макете формы Qt, в пространстве имен Ui создается динмачиеский экземпляр класса выбранного элемента с набором свойств, унаследованных от базового класса. Обратившись к указателю на пространство имен Ui, можно получить доступ ко всем псевдонимам элементов управления, которые сейчас размещены на форме, а поскольку экземпляры создаются через выделение динамической памяти, то на эту область памяти будет ссылаться указатель с именем объекта, задаваемым через свойство objectName.

Такие дела. Это, конечно, если не вдаваться в тонкости, и не очень-то и точно, если так посмотреть, но если использовать те термины, и те механики, которыми вы должны были овладеть, то это почти что на самом деле так. Пространство имен Ui, как и указатель на него, создается в момент создания самого класса, через среду разработки, так что нам ничего не стоит воспользоваться инструментами, данными нам свыше. Что касается объекта-адресата, то в 99,9% в ваших проектах таким объектом будет сам экземпляр того класса, в котором находятся объекты-адресанты, который будет вызываться в функцию connect через указатель this. Далее, я приведу в таблице все необходимые для зубрежки сигналы от элементов, с которыми пользователь будет взаимодействовать.

Элемент управления	Сигнал	Описание сигнала
Line Edit	textChanged	Изменение текста внутри поля ввода

Push Button	clicked	Нажатие на кнопку (в том числе, пометка «галочкой»
Check Box		
Combo Box	currentIndexChanged	Выбор нового элемента из выпадающего списка
Text Edit	textChanged	Изменение текста внутри поля ввода
Table Widget	cellClicked	Нажатие на ячейку какой-либо строки в таблице
	cellDoubleClicked	Двойное нажатие на ячейку какой-либо строки в таблице
Menu	aboutToHide	Сигнал возникает, когда подсветка пункта меню после нажатия на него пропадает

Самый нелогичный сигнал в этом списке – aboutToHide, казалось бы, но объекты класса QMenu не имеют простого сигнала clicked, поэтому придется импровизировать. Разумеется, далее в работе я покажу, как этот сигнал можно «прикрутить» к объектам пунктов меню через переопределение базового класса, однако, маловероятно, что вы будете такой ерундой на демке заниматься, этому без некоторого «прикостыливания» все же не обойтись.

Последнее, что осталось разобрать перед тем, как начать усиленно разбирать код, который я подготовил – это механизмы взаимодействия с БД. Программа, созданная на Qt, общается с СУБД через ***драйверы***. Драйвер определяет тип СУБД, наборы инструкций для взаимодействия, ну, и много чего еще, что нас сейчас мало интересует. А вот то, что если вы пахнете слабостью, и поставили Qt 5 из терминала, а не из онлайн-установщика, то все необходимые драйвера для работ с любыми СУБД вам придется собирать вручную, из исходников. То еще удовольствие, честно скажу, так еще и шанс их успешной сборки прямо пропорционален вашему скиллу в Linux-системах. Онлайн-установщик же соберет базовые драйвера самостоятельно. Какие драйвера понадобятся вам? Один – «QMYSQL», поскольку для работы с базами данных вам будет доступна только СУБД MySQL, и не более.

Для работы с БД через программу на Qt, необходимо знать поверхностные механики работы двух классов, это QSqlDatabase и QSqlQuery. Объекты класса QSqlDatabase в ваших программах будут отвечать за настройку соединения с той БД, которая будет в вашем распоряжении, а объекты класса QSqlQuery – за адресацию запросов в эту БД и получению результатов этих запросов для их последующей программной обработки. Существует одна неочевидная механика в работе подключений от объектов QSqlDatabase. По-хорошему, подключение к БД не должно быть открыто постоянно в процессе работы программы, это должно обуславливаться принципами безопасности и устойчивости системы, поэтому программа должна открывать подключение всякий раз, когда хочет адресоваться к БД, и закрывать текущее подключение, дабы предотвратить возможные утечки данных, или же их порчу, несанкционированный доступ к ним, и так далее. Однако, если из одного модуля программы, открыть другой модуль, в котором будет произведен цикл открытия-закрытия подключения к БД, то при закрытии этого дочернего модуля, в родительском подключение не просто закроется, а отключится, то есть, его нельзя будет открыть заново без перезапуска модуля, поскольку подключения к БД в Qt реализованы на глобальном уровне, и используют общие стеки вызовов. Закрытие стека одного модуля от глобального подключения закрывает все прочие стеки от других модулей по этому подключению, и это крайне неочевидная вещь. Ладно, я понял, простыми словами: вы авторизировались под учеткой пользователя с ролью «только чтение», программа запустила модуль с выводом данных, обратилась к БД, отобразила данные в окне, а вы осознали, что роль пользователя не та, и вышли из модуля.

При уничтожении экземпляра формы уничтожилось и глобальное подключение к БД, и модуль авторизации больше не может подключиться к базе, он буквально не сможет настроить соединение, пока вы не перезапустите модуль авторизации.

Выглядит ситуация странно, но так и есть, совершенно рядовая ситуация, которая решается простыми методами. Начнем по порядку – с подключения к БД. Как было сказано ранее, за это будут отвечать объекты класса QSqlDatabase. Они объявляются в заголовочном классе формы, для их объявления требуется подключить библиотеку <QSqlDatabase>. Такие объекты являются приватными членами класса, и объявляются с помощью метода addDatabase(...) от класса QSqlDatabase. Метод addDatabase() принимает два параметра: название драйвера СУБД в виде строки и метку подключения, тоже в виде строки. Метка подключения может иметь любое содержимое, и по этой самой метке программа будет использовать локальные подключения к БД, от метки, а не глобальное, что решит проблему перекрытия подключений между модулями. Каждый модуль будет обращаться только к своему экземпляру QSqlDatabase, и будет создавать подключение по метке, и закрывать его тоже по этой же самой метке.

```
#include <QSqlDatabase>
....
class AutorizationForm : public QMainWindow
{
    ...
private:
    QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL", "autorization");
                                                        драйвер        метка
    ...
};
```

Однако, просто наличие объекта подключения не дает самого подключения, его необходимо настроить. Достигается это вызовом функций-акссессоров от экземпляра класса QSqlDatabase. Последовательность вызовов аксессоров не так важна, но я привык придерживаться следующего порядка:

1. Определение IP адреса сервера СУБД, на котором крутится ваша БД. Вызывается через метод setHostName(), в который необходимо передать **строку** с IP адресом сервера: db.setHostName("127.0.0.1");
2. Определение порта для подключению к серверу СУБД. Вызывается через метод setPort(), в который необходимо передать **целое значение** с портом сервера: db.setPort(3306);
3. Определение наименования самой БД, к которой будет происходить подключение. Вызывается через метод setDatabaseName(), в который необходимо передать **строку** с названием: db.setDatabaseName("user10_db");
4. Определение имени пользователя на сервере СУБД для авторизации на самом сервере и получения доступа к БД. Вызывается через метод setUserName(), в который необходимо передать **строку** с именем юзера: db.setUserName("root");
5. Определение пароля пользователя на сервере СУБД для авторизации на сервере. Вызывается через метод setPassword(), в который необходимо передать **строку** с именем юзера: db.setPassword("0000");
6. Вызов функции open(), которая вернет true, если программа смогла настроить подключение с указанными параметрами, или false, если таки не смогла: db.open();

Сразу же возьмем за внимание, что задание на демоэкзамен подразумевает разработку трех отдельных модулей: авторизации, основное окно и модуль заявки, в которых будут объявлены свои экземпляры класса QSqlDatabase со своими метками, и для каждого модуля при каждой попытке подключения к БД, это подключение нужно будет настраивать. Один и тот же механизм будет многократно раз вызываться из трех разных модулей. Можно определять параметры подключения каждый раз при возникновении необходимости, т.е. буквально в каждой функции модуля дублировать по шесть строчек кода, что явно выглядит как говнокод, не иначе. Можно для каждого отдельного модуля создать свою функцию, которая будет настраивать подключение и возвращать результат по этому поводу, но дублирование трех (и это минимум) блоков одного и того же кода – тоже не прелесть какое занятие. Воспользуемся определением глобальной функции, которая не будет привязана к какому-либо классу, и может быть вызвана из любого из них, поскольку будет описана в собственном модуле. Кстати говоря, это не единственный раз, когда вам потребуется создавать глобальные функции, будет и еще парочка.

Итак, добавьте к проекту один файл исходного кода C++ и один заголовочный класс C++. Внимание: НЕ КЛАСС C++, статичных классов, как в C#, в C++ нет, поэтому нет смысла объявлять глобальные функции через классы в C++ нет никакого, можно это сделать просто модулями и подключением заголовочных файлов к классам форм. Сразу отмечу, что названия этих файлов должны полностью совпадать, это важно, ну, за исключением расширения файла, понятное дело. Я назвал их как «globals.cpp» и «globals.h».

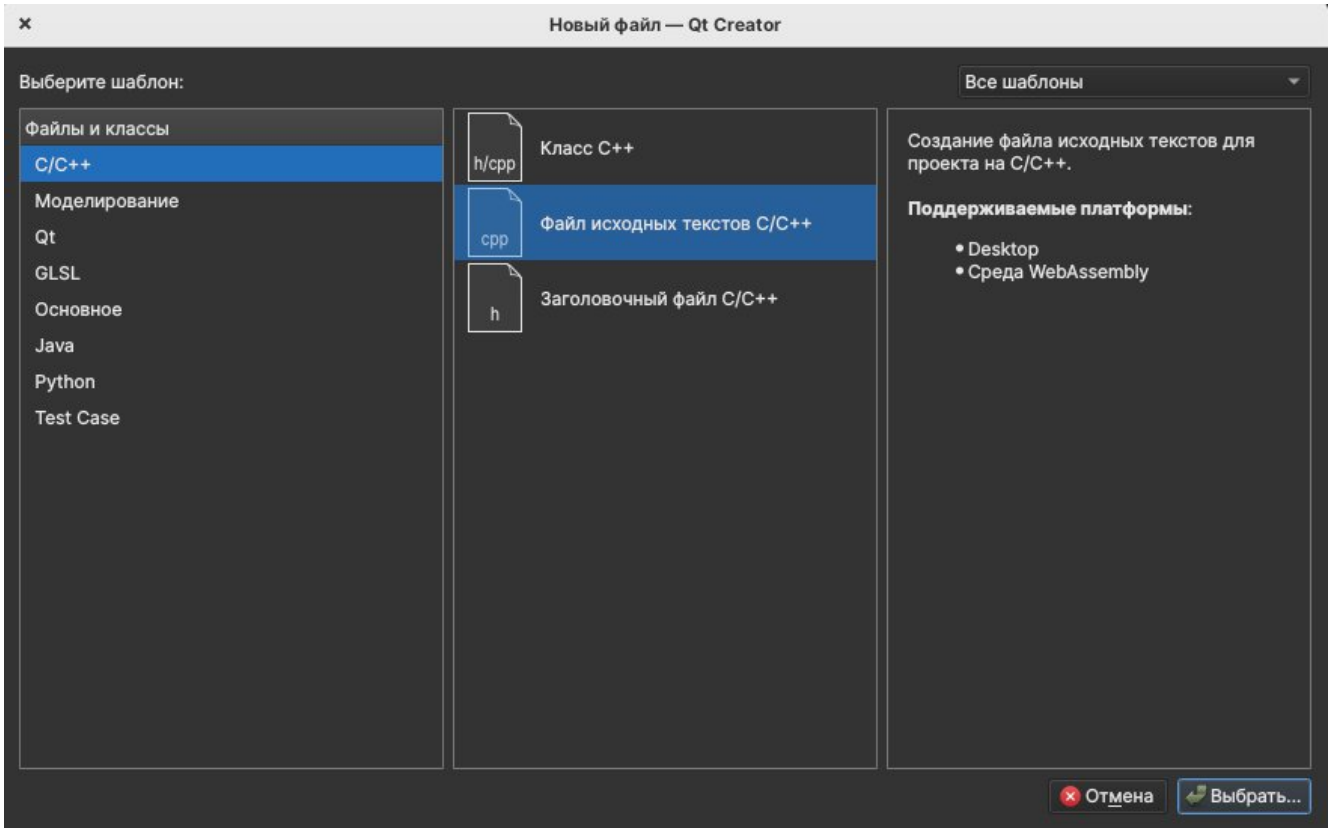

Рисунок 34 – Добавление нового файла .cpp в проект

В заголовочном файле необходимо определить прототип глобальной функции настройки подключения к БД. Функция будет возвращать булевое значение, а принимать аргумент экземпляра класса QSqlDatabase, для которого будут настраиваться параметры подключения к БД.

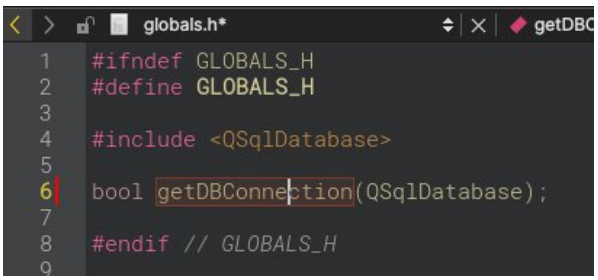

Рисунок 35 – Добавление прототипа в заголовочный файл

Добавление реализаций для прототипов функций в Qt автоматизировано за счет вызова функций рефакторинга. Наведите курсор мыши на ваш прототип, вызовите контекстное меню, выберете пункт «Рефакторинг», а уже внутри него – «Добавить реализацию в globals.cpp». Если у вас не отображается этот пункт, то тут есть два варианта: реализация для этого прототипа уже создана в этом файле исходного кода, или же название заголовочного файла не совпадает с файлом исходного кода, и среда разработки не воспринимает их как составные части одного модуля.

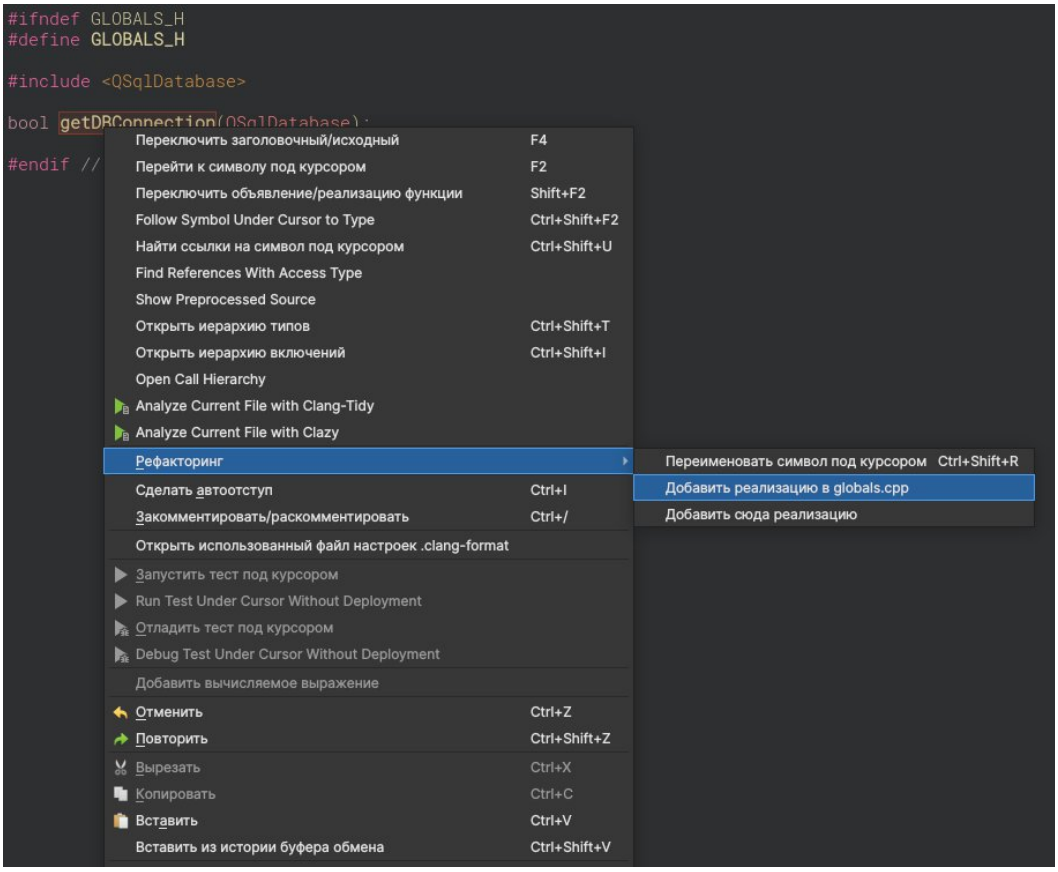

Рисунок 36 – Добавление реализации для прототипа функции

Среда разработки сама перенаправит вас в файл исходного кода, в котором будет представлена пустая реализация вашей функции без псевдонима аргумента. Добавьте его, также добавьте подключение заголовочного файла #include "globals.h", если среда разработки не сделала это самостоятельно при добавлении реализации через рефакторинг. Далее – ничего сложного, оформляем вызовы вышеописанных аксессоров от экземпляра QSqlDatabase, пришедшего на вход как аргумент, и возвращаем значение глобальной функции настройки подключения к БД от вызова метода .open от объекта QSqlDatabase.

Рисунок 37 – Определение реализации глобальной функции

Закончим с определением глобальных функций. Так получилось, что вызов диалоговых окон сообщений пользователю в Qt устроен сложнее, чем в Windows Forms. Там это умещалось в вызове одной строчки кода от одной функции, а здесь это – целый квест, разумеется, с определением параметров через аксессоры, но уже от другого класса – класса QMessageBox.

Вернитесь в заголовочный файл модуля глобальных функций, это можно сделать не только через обозреватель объектов проекта, но через горячую клавишу F4, которая переключает .cpp/.h файлы между собой в рамках одного модуля. Добавьте два прототипа в одним и тем же названием функции, т.е. сделайте перегрузку функции. Эта функция будет настраивать и отображать пользователю некоторое сообщение на экране. Первый аргумент будет просто строкой непосредственно самого сообщения, второй аргумент – это заголовок для окна этого сообщения, тоже представляемый как строка. А вот третий и четвертый аргументы должны представлять собой кнопку, задаваемую внутри окна, которую пользователь должен нажать, и пиктограмму, которая характеризует смысл сообщения пользователю. Как же их объявить в виде аргументов? Ответ: по определению нужного перечисляемого типа от базового класса QMessageBox.

«Ничего не понял. Но очень интересно.» – примерно такой реакции я ожидаю сейчас от вас, читающих сей документ, и попытаюсь на простых примера объяснить, что это такое, как используется и для чего нужно. Перечисляемые типы определяются по ключевому слову enum, и представляют собой множество константных элементов, которые никак нельзя изменять, и каждый из которых можно представить в виде двух равносильных и взаимозаменяемых параметров: по имени элемента и по значению, которое определяется целым типом данных, в частности, int. Это означает, что прямое сравнение имени элемента внутри типа enum с его значением будет идентичным, поскольку программа всегда будет подразумевать именно целое значение от элемента, но программисту работать проще именно с символическим представлением значения, которое может иметь осознанный вид, чем с набором чисел, которые могут означать все, что угодно. Так вот, Qt имеет множество перечисляемых типов, одни относятся ко всему фреймворку в целом, как например, выравнивание текста или стандартные цвета палитры, а другие относятся к разным классам, вот перечисления пиктограмм и кнопок для окон сообщений находятся в классе сообщений QMessageBox.

Перечисляемый тип кнопок называется StandardButton, значение, хранящиеся в нем, определяют, ВНЕЗАПНО, кнопки, которые могут быть размещены внутри диалогового окна сообщения. В Qt не предусмотрено сдвоенных, или даже строенных определений кнопок, как WF: OkCancel, YesNo, AbortRetryIgnore и т.д. Список значений для кнопок, которые необходимо бы выучить, по-хорошему, представлен в таблице ниже.

Имя элемента	Значение int	Описание
QMessageBox::Ok	0x00000400	Кнопка «OK»
QMessageBox::Open	0x00002000	Кнопка «Открыть»
QMessageBox::Save	0x00000800	Кнопка «Сохранить»
QMessageBox::Cancel	0x00400000	Кнопка «Отменить»
QMessageBox::Close	0x00200000	Кнопка «Закрыть»
QMessageBox::Discard	0x00800000	Кнопка «Отменить изменения» или «Не сохранять»
QMessageBox::Apply	0x02000000	Кнопка «Применить»
QMessageBox::Reset	0x04000000	Кнопка «Сброс»
QMessageBox::RestoreDefaults	0x08000000	Кнопка «Восстановить по умолчанию»
QMessageBox::Help	0x01000000	Кнопка «Помощь»
QMessageBox::SaveAll	0x00001000	Кнопка «Сохранить все»
QMessageBox::Yes	0x00004000	Кнопка «Да»
QMessageBox::YesToAll	0x00008000	Кнопка «Да для всех»
QMessageBox::No	0x00010000	Кнопка «Нет»
QMessageBox::NoToAll	0x00020000	Кнопка «Нет для всех»
QMessageBox::Abort	0x00040000	Кнопка «Прерывать»
QMessageBox::Retry	0x00080000	Кнопка «Повторить попытку»
QMessageBox::Ignore	0x00100000	Кнопка «Игнорировать»

Перечисляемый тип пиктограмм называется Icon. Список всех пиктограмм, которые используются для подчеркивания контекста сообщения, представлен в таблице ниже.

Имя элемента	Значок	Описание
QMessageBox::NoIcon		В окне сообщения нет значка.
QMessageBox::Question		Значок, указывающий, что сообщение содержит вопрос.
QMessageBox::Information		Значок, указывающий, что в сообщении нет негативного контекста.
QMessageBox::Warning		Значок, указывающий, что сообщение является предупреждением о некритической ситуации.
QMessageBox::Critical		Значок, указывающий, что сообщение представляет собой критическую проблему.

Теперь непосредственно о самом диалоговом окне. Для того, чтобы сообщить пользователю о чем-нибудь, необходимо объявить статичный экземпляр класса QMessageBox, и переопределить некоторые значения полей класса этого экземпляра по следующим акссессорам:

1. setWindowTitle() определяет заголовок окна, требует явного указания параметра QString как значения заголовка;
2. setText() определяет сообщение, которое будет отображено пользователю, требует явного указания параметра QString как значения сообщения;
3. setDefaultButton() определяет кнопку «по умолчанию», т.е. та, которая будет находиться в фокусе при отображении окна на экране, требует явного указания параметра QMessageBox::StardardButton как значения кнопки;
4. setIcon() определяет пиктограмму, которое будет соответствовать контексту сообщения, требует явного указания параметра QMessageBox::Icon как значения пиктограмм;
5. addButton() добавляет кнопку на окне сообщения, значение кнопки необходимо передать в метод как параметр QMessageBox::StardardButton.

Вызов метода exec() от экземпляра класса QMessageBox отображает диалоговое окно на экран пользователя с заданными параметрами, причем нажатие пользователем любой из представленных кнопок вернет из этого метода целое значение, соответствующее нажатой кнопке, что позволит нам в будущем определять выбор пользователя из нескольких вариантов, и ветвить выполнение программы. Разумеется, эти списки с аксессорами не полные, и не раскрывают всех возможностей классов, таких как QMessageBox, допустим, но этого вполне достаточно, чтобы сделать **К Р А С И В О**, и получить n-ное количество баллов за оформление вывода сообщений по заданию. Теперь можно вернуться к прототипам. Их недаром потребуется два, поскольку для 99% всех сообщений в вашей работе потребуется всего одна кнопка: OK, и ее даже передавать аргументом необязательно, но параметры расширяют вариативность использования функций, так что – лишним не будет. Но вот тот самый 1% однажды заставит пользователя спросить выбор о двух или более вариантах исхода, и тогда как передать в функцию значения из enum StandardButton, в случае, если их больше чем одно? Ну, то есть, их может быть и 2, и 3, или даже 4, никто не запретит, но обработать такое диалоговое окно необходимо в любом случае, так что же делать? Правильно, воспользоваться массивами. Передадим как параметр не одну кнопку, а массив значений тех кнопок, которые нужно разместить в окне. Базовыми массивами пользоваться уже попросту неудобно, векторами из STL библиотеки в рамках программирования на Qt – крайне вопросительно, поэтому воспользуемся местной реализацией списков – QList.

По итогу прототипы функций будут иметь следующий вид:

```
#include <QSqlDatabase>
#include <QMessageBox>
bool getSqlConnection(QSqlDatabase);
int showMessageBox(QString, QString, QMessageBox::Button, QMessageBox::Icon);
int showMessageBox(QString, QString, QList<QMessageBox::Button>, QMessageBox::Icon);
```

Про реализацию QList вы не найдете информации далее, потому что мне тогда вообще придется пересказать весь учебник по Qt, а это вы можете и самостоятельно сделать, так что – дерзайте. Мне неиронично придется пропустить очень многое, но только

потому, что этот документ – не учебник таки, он рассчитывает на то, что хотя бы какие-то знания и навыки у вас уже есть, я и так расписываю все, что на мой взгляд кажется неочевидным или непонятным.

Реализации же этих функций представлены ниже.

```
#include "globals.h"

bool getSqlConnection(QSqlDatabase db)
{
    db.setHostName("127.0.0.1");
    db.setPort(3306);
    db.setDatabaseName("user10_db");
    db.setUserName("root");
    db.setPassword("0000");
    return db.open();
}

int showMessageBox(QString message, QString title, QMessageBox::Button button, QMessageBox::Icon icon)
{
    QMessageBox msg;
    msg.setWindowTitle(title);
    msg.setText(message);
    msg.setDefaultButton(button);
    msg.setIcon(icon);
    return msg.exec();
}

int showMessageBox(QString message, QString title, QList<QMessageBox::Button> buttons, QMessageBox::Icon icon)
{
    QMessageBox msg;
    msg.setWindowTitle(title);
    msg.setText(message);
    for(QMessageBox::Button button : buttons)
    {
        msg.addButton(button);
    }
    msg.setDefaultButton(buttons.at(buttons.length() - 1));
    msg.setIcon(icon);
    return msg.exec();
}
```

Теперь перейдем к работе над модулем авторизации. Подключим основные библиотеки по работе с БД в заголовочный файл, а в теле класса формы определим один слот, который будет обрабатывать нажатие на кнопку авторизации, и экземпляр QSqlDatabase для настройки подключения к БД из этого модуля:

```
#include <QSqlDatabase>
#include <QSqlQuery>
class AutorizationForm : public QMainWindow
{
    ...
public slots:
    void tryAutorization(QString, QString);
private:
    QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL", "autorization");
};
```

Слот tryAutorization() будет принимать 2 параметра: это логин и пароль, которые пользователь введет в поля ввода на форме. Добавьте реализацию этого слота в файл исходного кода класса.

```
void AutorizationForm::tryAutorization(QString login, QString password)
{

}
```

Алгоритм этого метода сводится к выполнению следующих этапов:

1. Настраиваем и проверяем подключение к БД от текущего экземпляра QSqlDatabase;
2. Если подключение к БД установлено, то:
    a. Если длина строки с логином, после удаления из нее всех символов пробела, больше 0, то:
        i. Если длина строки с паролем, после удаления из нее всех символов пробела, больше 0, то:
            A. Адресуем запрос в БД выборки ФИО пользователя и его ролью с параметрами логина и пароля, поступившими на вход в слот;
            B. Если возвращаемые данные из БД были получены, т.е. сопоставление найдено, и найден такой пользователь с такими логином и паролем, то запускаем форму основного окна, передаем в нее значения из БД, замораживаем выполнение текущей формы, пока не закроется новое окно, а после – очищаем поля ввода данных;
            C. Иначе показываем сообщение об ошибке по неудачной авторизации;
        ii. Иначе показываем сообщение об ошибке по незаполненному полю «Пароль»;
    b. Иначе показываем сообщение об ошибке по незаполненному полю «Логин»;
3. Иначе показываем сообщение об ошибке по отсутствию подключению к БД.

Полный код слота tryAutorization() представлен ниже:

```
void AutorizationForm::tryAutorization(QString login, QString password)
{
    db = QSqlDatabase::database("autorization");
    if(getSqlConnection(db))
    {
        if(login.remove(" ").length() > 0)
        {
            if(password.remove(" ").length() > 0)
            {
                QSqlQuery query(db);
                query.prepare("select Users.UserID, concat(Users.UserSurname, ' ', Users.UserName, ' ', Users.UserPatronymic) as FIO, UserRole.UserRoleValue from Users inner join UserRole on UserRole.UserRoleID = Users.UserRole inner join UserAutorization on UserAutorization.UserID = Users.UserID where UserAutorization.UserLogin = ? and UserAutorization.UserPassword = ?;");
                query.addBindValue(login);
                query.addBindValue(password);
                query.exec();
                if(query.next())
                {
                    OperatorForm *mainForm = new OperatorForm();
```

```
                    mainForm->setFormState(query.value(0).toString(), query.value(1).toString(),
query.value(2).toString());
                    this->hide();
                    mainForm->show();
                    while(mainForm->isVisible())
                    {
                        QApplication::processEvents();
                    }
                    ui->LE_Login->clear();
                    ui->LE_Password->clear();
                    ui->LE_Login->setFocus();
                    this->show();
                }
                else
                {
                    showMessageBox("Неверный логин или пароль.\nПользователь не обнаружен.",
"Неудачно", QMessageBox::Ok, QMessageBox::Warning);
                }
            }
            else
            {
                showMessageBox("Поле 'Пароль' должно быть заполнено.", "Действие прервано",
QMessageBox::Ok, QMessageBox::Warning);
            }
        }
        else
        {
            showMessageBox("Поле 'Логин' должно быть заполнено.", "Действие прервано",
QMessageBox::Ok, QMessageBox::Warning);
        }
    }
    else
    {
      showMessageBox("Отсутствует подключение к БД.", "Действие прервано", QMessageBox::Ok,
QMessageBox::Critical);
    }
}
```

Для работы глобальных функций showMessageBox() и getSqlConnection() необходимо в текущий .cpp файл подключить заголовочный файл #include "globals.h", и так нужно будет делать для всех файлов исходного кода классов форм, с которыми вы будете работать.

Здесь необходимо объяснить механику «заморозки» состояния формы. На WF есть два способа вызова модальных окон: Show() и ShowDialog(). Вывоз формы на экран через ShowDialog() перехватывает управление на дочернюю форму, и возвращает его только в случае закрытия этого окна, соответственно, выполнение кода внутри родительской формы приостанавливается, что позволяет, допустим, скрыть эту форму до вызова дочерней, а после – отобразить обратно, и для пользователя это выглядит бесшовно, моментально, словно так и задумано. Вызов же окна через метод Show() просто отображает дочернее окно, однако, с родительским по-прежнему можно взаимодействовать параллельно, что не всегда подходит под задуманную программную логику. Так вот, метода вызова ShowDialog() в Qt нет, и родительская форма всегда будет доступна пользователю для взаимодействия, что позволит сколь угодно как взаимодействовать с ней в момент, когда по логике программы пользователь не должен иметь такой возможности. Для этого к коду программы будет прикручен костыль, который буквально будет приостанавливать работу формы через цикл, который будет постоянно опрашивать состояние дочерней формы, ее отображение на экране, и, как только эта

форма будет закрыта, то выполнение кода внутри родительской формы будет продолжено. Это буквально костыль, поскольку выполнение цикла будет нагружать процессор, вплоть до 10-15%, но это быстрый и простой способ удовлетворить нужды программной логики.

Теперь поработаем над основным модулем, который будет предоставлять доступ к основному функционалу программы, а именно: поиск и просмотр данных по заявкам, а также добавление новой заявки или просмотр/редактирование сведений по текущей заявке. Добавьте новый класс формы Qt, назовите его OperatorForm, и смоделируйте следующий GUI для этой формы:

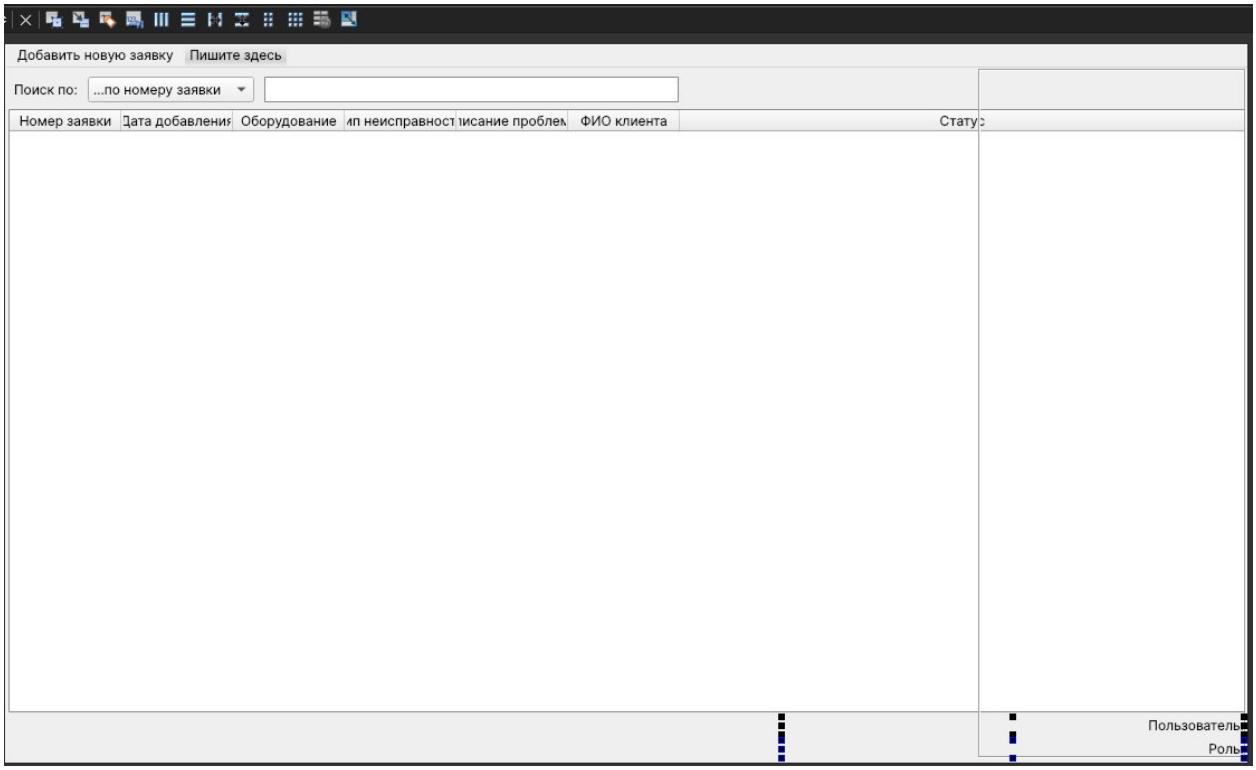

Рисунок 38 – Макет формы основного окна

Для изменения колонок таблицы QTableWidget щелкните по ней, вызовите контекстное меню, и выберете в нем пункт меню «Изменить элементы».

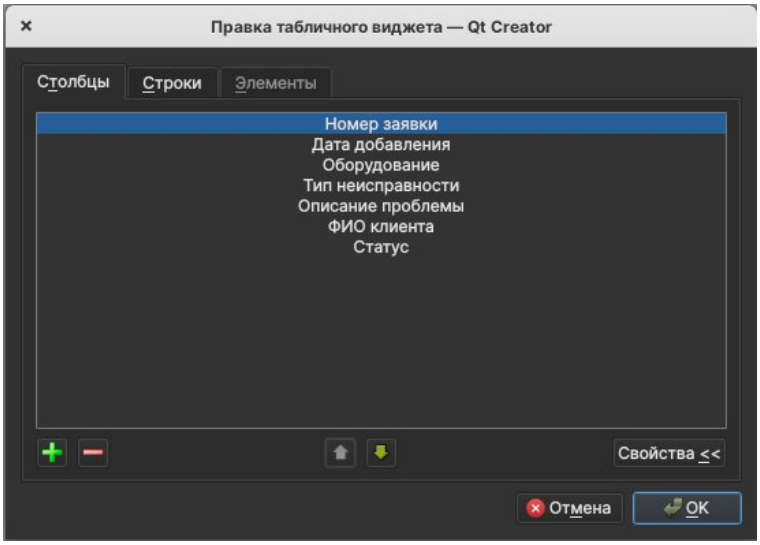

Рисунок 39 – Изменение элементов виджета QTableWidget

Этот же способ работает и для элементов выпадающего списка QComboBox. Что касается колонок таблиц QTableWidget, то дизайнер форм не позволяет редактировать их ширину, это делается программно.

Справа можно заменить некоторую рамку, которая перекрывает собой таблицу. Это не просто рамка, которая могла бы попасть на макет формы случайно, это элемент QListWidget. Переопределением цветовой палитры свойства palette я сделал его задний фон прозрачным, а рамку выставил специально, чтобы вы могли его заметить на форме. Этот элемент будет содержать всплывающие уведомления об изменениях состояний статусов заявок, это требуется по заданию демоэкзамена. Сразу скажу, что именно реализация уведомлений в ваших работах, скорее всего, будет пропущена, поскольку работать с кастомным выводом через макеты элементов вам будет сложно, если у вас нет хоть какого-либо опыта, просто будете время терять на демке, но разобрать эту механику, точнее, то, как я ее сам понял и как смог реализовать – весьма полезно, лишним не будет.

Однако, я это рассмотрю не сейчас. В первую очередь необходимо реализовать базовый функционал. По сути дела, эта форма не годится сейчас для дальнейшей работы, поскольку у нас нет никакой выборки данных для вывода. Мы сможем реализовать программные коды для вывода данных в таблицу, и поиск, и уведомления, но не сможем их проверить, потому что программа не предоставляет возможностей для создания заявок. Мы вернемся к работе над этим классом позже, а пока что создайте новый класс формы Qt, назовите его OrderForm. Это будет макет формы работы над заявкой, используя возможности этого модуля, пользователь сможет как создавать новые заявки, так и редактировать данные уже существующей заявки.

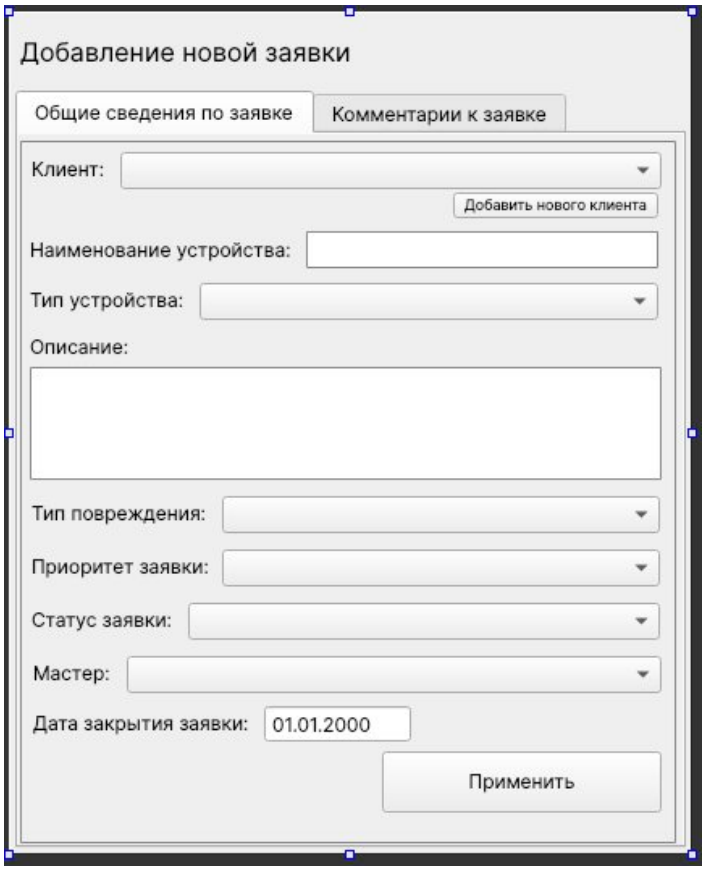

Рисунок 40 – Макет формы заявки

Техническое задание гласит, что на форме заявки исполнитель должен иметь возможность добавлять комментарии к заявке. Стоит разбить функционал с помощью элемента групповой сортировки по вкладкам, QTabWidget, дабы не перегружать интерфейс формы. Сперва поработаем над вкладкой «Общие сведения по заявке», как можно видеть на рисунке 40, большая часть элементов управления представляют собой элементы QComboBox, данные в которые будут предзагружаться из справочников БД. Надпись «Дата закрытия заявки» и элемент QDateEdit будут невидимы для пользователя, если заявка не закрыта, и недоступны для ручного изменения данных в них, поскольку за управление значением даты будет отвечать сама программа. Также обратите внимание на кнопку «Добавить нового клиента». Система должна предусматривать ситуацию, при которой пользователь должен будет иметь возможность добавить нового клиента в систему, прямо в момент создания нового заказа, причем с сохранением ранее введенных данных в поля ввода. Это удобнее и практичнее, чем реализовывать отдельный модуль только для добавления нового клиента, реализуем эту идею на базе уже существующего модуля.

Правда, для этого придется пойти на некоторую хитрость. Внутри контейнера вкладки «Общие сведения по заявке» можно заметить вложенную рамку, внутри которой размещены все остальные прочие элементы управления. Как можно догадаться, это не просто случайная рамка, а элемент QFrame, который является в данном случае вложенным контейнером. Идея предельно простая: элементы управления, размещенные внутри контейнера, принимают некоторые базовые свойства самого контейнера, внутри которого они располагаются, для нас это особенно интересно, поскольку изменение свойства visible на false только для контейнера скроет не только сам контейнер, но и все элементы, которые находятся внутри него, разом и автоматически. Это правило относится ко многих свойствам: visible, enabled, palette и т.д.

Моей идеей стало размещение двух контейнеров QFrame, которые будут менять свои свойства visible, в зависимости от того, какое действие сейчас нужно будет реализовать пользователю: создавать заявку или добавлять пользователя. Один из контейнеров будет накладываться поверх другого, а другой будет скрываться, для пользователя это будет совершенно не заметно, а нам позволит выполнить вот такой финт ушами. Самое главное, чтобы объекты QFrame не находились внутри друг друга, но были расположены рядом, внутри контейнера вкладки, этого можно достичь, выставив чуть разные координаты X и Y свойства geometry: для контейнера общей информации по заявке пусть они будут как [1;4], а для контейнера добавления нового пользователя [2;4]. Разница в 1 пиксель относительно X координаты позволит наложить один контейнер на другой, при этом так, чтобы они не считались вложенными относительно друг друга.

Панель добавления нового клиента изображена на рисунке 41, а целиком форма со вкладкой с наложенными друг на друга полями представлена на рисунке 42.

По сути дела, именно эта форма будет представлять собой самый нагруженный функционалом макет, с точки зрения программного кода это примерно 700 строк, поскольку здесь идет совмещение функционала: тут и создание новой заявки, и просмотр/редактирование уже имеющейся, и просмотр/добавление комментариев. Одним словом, достаточное количество функционала для одной формы. Начнем с заявок, а возможность комментирования отложим «на потом». Сейчас стоит разобраться с наименованием объектов внутри вкладки «Общие сведения по заявке», а также с обработчиками событий и функциями, которые будут реализовывать набор программной логики внутри нее.

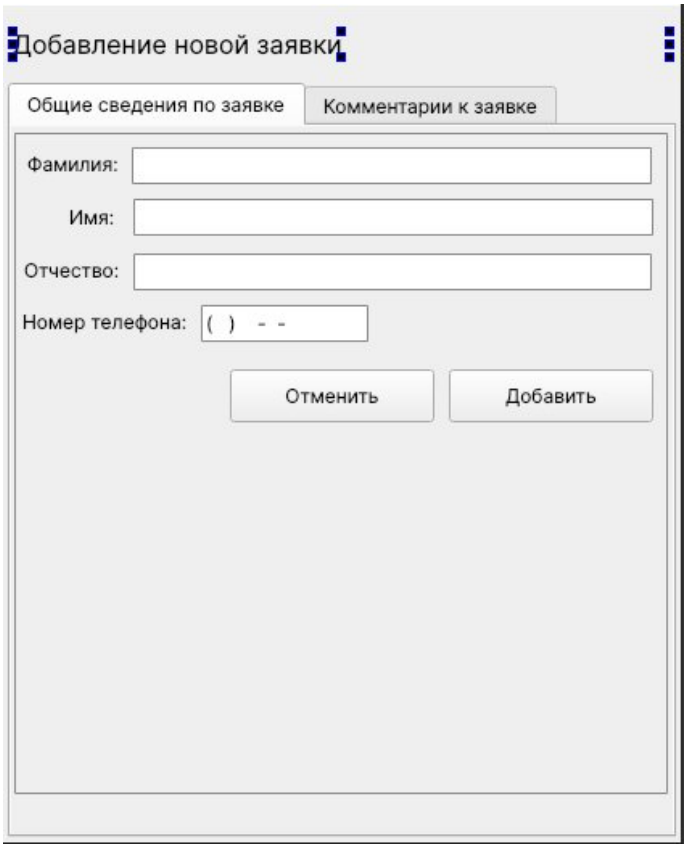

Рисунок 41 – Макет формы заявки (панель добавления нового клиента)

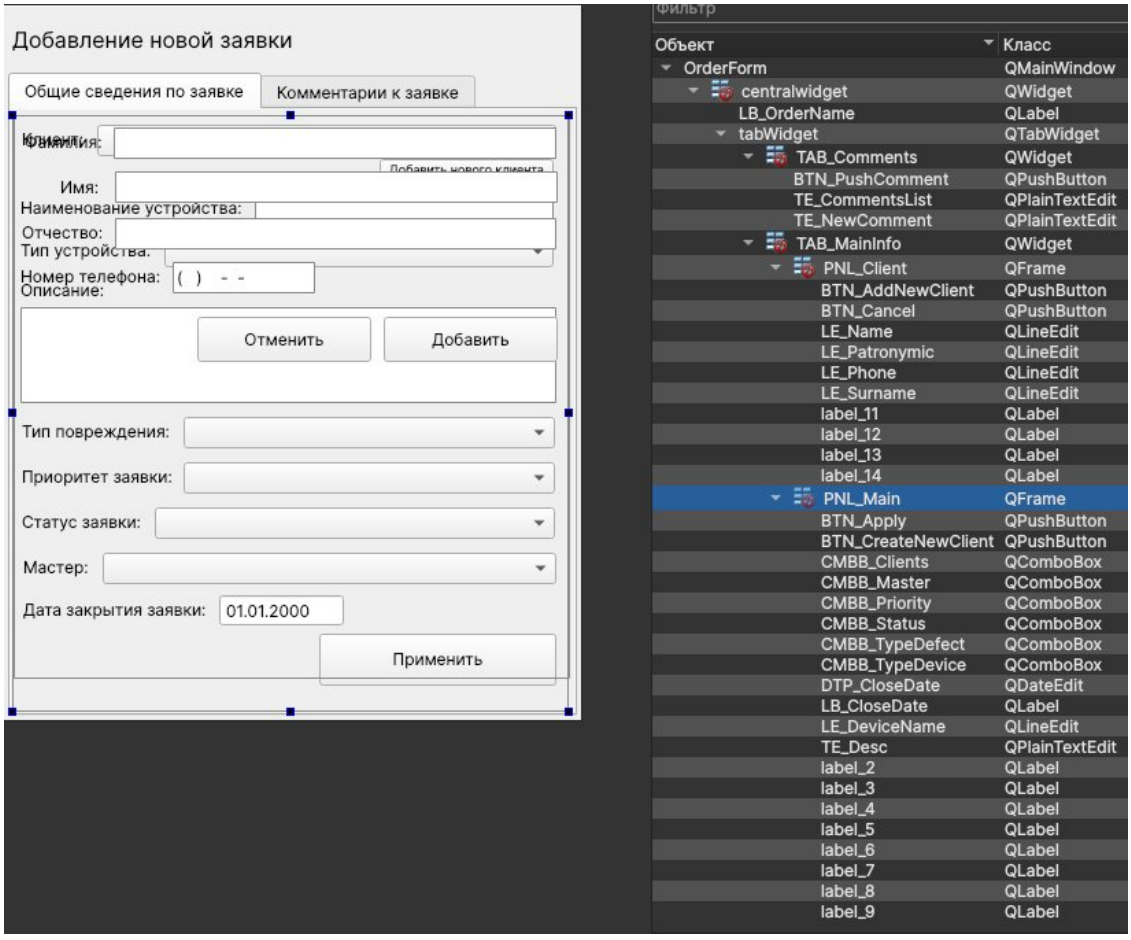

Рисунок 42 – Макет формы заявки (панель добавления нового клиента)

Для удобства работы я буду использовать ту же иерархичную структуру вложения в контейнеры, которая используется в обозревателе объектов на форме в Qt Creator.

LB_OrderName – Основная надпись на форме, когда происходит создание новой заявки, то пользователю в ней будет отображено «Добавление новой заявки», а когда происходит просмотр/изменение текущей заявки, то надпись будет менять свое значение на «Заявка № XX от XXXX-XX-XX»;

TAB_Comments – Вкладка с комментариями к заявке;

BTN_PushComment – Кнопка «Прикрепить комментарий»;

TE_CommentsList – Элемент просмотра текста всех комментариев в заявке;

TE_NewComment – Поле ввода нового комментария;

TAB_MainInfo – Вкладка «Основные сведения по заявке»;

PNL_Client – Панель с элементами управления для добавления нового клиента;

BTN_AddNewClient – Кнопка «Добавить», при ее нажатии программа проверит содержимое полей ввода, добавит новые сведения о клиенте в систему и обновит выпадающий список с перечнем доступных клиентов;

BTN_Cancel – Кнопка «Отменить», при ее нажатии программа очистит поля ввода информации о клиенте и скроет панель «PNL_Client», отобразив при этом панель «PNL_Main»;

LE_Name – Поле ввода имени клиента;

LE_Patronymic – Поле ввода отчества клиента;

LE_Phone – Поле ввода номера телефона клиента;

LE_Surname – Поле ввода фамилии клиента;

PNL_Main – Панель с элементами управления для работы над заявкой;

BTN_Apply – Кнопка «Добавить», в режиме работы формы «Создание» проверяет все поля ввода, создает новую заявку в системе и закрывает текущую форму; в режиме работы формы «Изменение» проверяет необходимые поля ввода, только те, которые могут быть изменены, и применяет изменения по заявке на сервере, а после – закрывает текущую форму;

BTN_CreateNewClient – Кнопка «Добавить нового клиента», при ее нажатии программа скрывает панель «PNL_Main» и отображает на экран панель «PNL_Client», скрыта в режиме «Изменение»;

CMBB_Clients – выпадающий список с перечнем всех клиентов, доступных в системе, недоступен для изменения в режиме «Изменение»;

CMBB_Master – выпадающий список с перечнем всех мастеров, доступных в системе;

CMBB_Priority – выпадающий список с перечнем всех приоритетов из справочника «Приоритеты заявки» в БД;

CMBB_Status – выпадающий список с перечнем всех приоритетов из справочника «Статусы заявки» в БД;

CMBB_TypeDefect – выпадающий список с перечнем всех приоритетов из справочника «Типы дефектов устройства» в БД, недоступен для изменения в режиме «Изменение»;

CMBB_TypeDevice – выпадающий список с перечнем всех приоритетов из справочника «Типы устройств» в БД, недоступен для изменения в режиме «Изменение»;

DTP_CloseDate – элемент просмотра даты закрытия заявки, недоступен для прямого изменения, и скрыт, если заявка не закрыта;

LB_CloseDate – надпись «Дата закрытия заявки:», скрыта, если заявка не закрыта;

LE_DeviceName – поле ввода наименования устройства;

TE_Desc – поле ввода описания заявки.

Теперь поговорим про обработчики событий, про слоты, которые будут реализовывать программную логику. Очевидно, что нам будут необходимы обработчики на все кнопки, которые пользователь будет иметь волен нажимать, их пять штук. Помимо этого, необходимо предусмотреть две неочевидные возможности модуля:

1. При выборе любого мастера из выпадающего списка, значение поля «Статус заявки» должно быть автоматически изменено на «На выполнении», поскольку, ну, так не бывает просто по логике, что исполнитель назначен на заявку, но ее статус по-прежнему висит «На выполнении», это неправильно;
2. При выборе значения «Выполнена» из выпадающего списка статусов заявок, программа должна спросить разрешения у пользователя для закрытия заявки, и, в случае положительного ответа, программа должна изменить статус заявки и заблокировать все поля ввода, поскольку их изменение более недопустимо для закрытой заявки.

Это еще 2 обработчика событий, итого на весь модуль их набежало аж 7 штук:

```
class OrderForm : public QMainWindow
{
    ...
public slots:
    void acceptOrderChanges(); //обработчик на нажатие кнопки BTN_Apply
    void startCreateNewClient(); //обработчик на нажатие кнопки BTN_CreateNewClient
    void cancelCreateNewClient(); //обработчик на нажатие кнопки BTN_Cancel
    void addNewClient(); //обработчик на нажатие кнопки BTN_AddNewClient
    void pushNewCommentToOrder(); //обработчик на нажатие кнопки BTN_PushComment
    void updateStatusByMaster(); //обработчик на выбор нового значения из CMBB_Master
    void updateStatusByClosed(); //обработчик на выбор нового значения из CMBB_Status
    ...
};
```

Помимо этого, необходимо подумать о базовых функциях класса, поскольку одними обработчиками событий сыт не будешь, они не покроют все необходимые требования к функционалу формы. Начнем с публичных методов. В класс, т.е. в саму форму, необходимо передавать стартовые значения для некоторых полей класса, таких как:

– текущий ID пользователя, поскольку на его анализе происходит доступ к возможности комментирования заказа;

– ID текущей записи, которая изменятся пользователем, это необходимо для корректной работы запроса UPDATE ... WHERE, программа должна знать ID заявки, чтобы иметь возможность изменять данные по ее уникальному идентификатору, а не всех заявок подряд;

– флаг работы модуля, т.е. в каком режиме сейчас будет происходить работа в модуле: на изменение имеющихся данных, или же на добавление новой заявки. Это важно, поскольку модуль, в случае его работы в режиме «Изменение», должен обращаться к БД, выгружать оттуда данные по заявке, по ее ID, блокировать доступ к полям, которые по заданию не должны изменяться, а после – настраивать работы кнопки BTN_Apply, потому что она должна будет работать с командами запроса UPDATE ... WHERE, а не INSERT INTO.

Добавим эти поля как приватные члены класса OrderForm, а также создадим публичные аксессоры для работы с ними:

```
class OrderForm : public QMainWindow
{
    ...
public:
    ...
    void setCurrentIDUser (QString value) { currentIDUser = value; }
    void setCheckState (bool value) { checkState = value; }
    void setCurrentIDOrder (QString value) { currentIDOrder = value; }
    ...
private:
    QString currentIDUser = "", currentIDOrder = "";
    bool checkState = false;
    ...
};
```

Также не забудем про внутренние вспомогательные методы этого класса:

1. Нам строго необходим метод, который будет обновлять содержимое полей ввода после определения базовых параметров этой самой формы, и который нельзя вызывать из конструктора класса, поскольку в момент создания экземпляра класса эти самые параметры еще не будут загружены, и форму нельзя настраивать под их значения. За это будет отвечать публичный метод void updateForm();
2. Необходим метод предзагрузки данных из справочников БД в выпадающие списки QComboBox. Этот функционал можно повесить и на метод updateForm(), однако, это и без того перегрузит этот метод функционалом, проще развести функционал на отдельные функции, чтобы с ними было проще и быстрее работать, а потом вызывать их по необходимости. За это будет отвечать приватный метод void fillDataToCMBB();
3. Отдельно потребуется метод, который будет обновлять данные **только** в выпадающем списке клиентов, потому что у нас есть функционал, позволяющий добавить нового клиента, и совершенно идиотской идеей будет перезагружать ВСЕ выпадающие списки только ради того, чтобы обновить только один, в котором данные заведомо устаревшие после добавления нового клиента, при этом, перезагружая все комбобоксы, мы рискуем потерять введенные в них ранее данные, и пользователю придется делать это по-новой, это непрактично, и так GUI никто не верстает, никто не будет работать с такой программой, это недопустимо. За работу этого функционала будет отвечать приватный метод void updateClientsCMBB();
4. По ходу работы потребуется фактически выбирать данные из выпадающих списков относительно тех данных, которые есть по заявке в БД. Это выгоднее, чем выставлять свойство editable как true у всех выпадающих списков, чтобы позже обносить свойство currentText у них же на те значение, которые присущи текущей заявке, это выглядит как костыль, и это не наш вариант. За это будет отвечать приватный метод void setCurrentText(QComboBox *, QString) с двумя параметрами: первым будет указатель на выпадающий список, в котором нужно будет выбирать значение, а второй – само значение, с которым будет сравниваться содержимое переданного выпадающего списка для последующего выбора в нем;
5. Отдельным методом я вынес функцию формирования комментария для поля TE_CommentsList. За работу этого функционала будет отвечать приватный метод QString

createMessage(QSqlQuery), который будет возвращать готовую строку комментария в элемент управления, и будет принимать параметр экземпляра QSqlQuery, который будет спозиционирован на следующий комментарий из выборки SQL-запроса.

Помимо этого, программа должна хранить в своей памяти ID предыдущего исполнителя, который мог быть назначен для заявки, а также индекс последнего выбранного статуса заявки, ведь, как было сказано ранее, программа должна подтверждать выбор пользователя при попытке изменения статуса заявки на «Выполнена», однако, она может получить и отрицательный ответ, при котором должна будет вернуть то значение, которое стояло прежде, до выбора отмененного, однако, позвольте спросить, как она собирается это сделать, если не будет держать в памяти предыдущее? Верно, что никак, и мы, как программисты, обязаны дать программе такую возможность.

Итак, много было сказано, но мало сделано, будем это исправлять. Полный код заголовочного файла класса OrderForm представлен ниже.

```
#ifndef ORDERFORM_H
#define ORDERFORM_H
#include <QtWidgets/QMainWindow>
#include <QSqlDatabase>
#include <QSqlQuery>
#include <QDate>
#include <QDateTime>
#include <QComboBox>

namespace Ui { class OrderForm; }

class OrderForm : public QMainWindow
{
    Q_OBJECT

public:
    explicit OrderForm(QWidget *parent = nullptr);
    ~OrderForm();
    void setCurrentIDUser(QString value) { currentIDUser = value; }
    void setCheckState (bool value) { checkState = value; }
    void setCurrentIDOrder(QString value) { currentIDOrder = value; }
    void updateForm();
public slots:
    void acceptOrderChanges();
    void startCreateNewClient();
    void cancelCreateNewClient();
    void addNewClient();
    void pushNewCommentToOrder();
    void updateStatusByMaster();
    void updateStatusByClosed();
private:
    Ui::OrderForm *ui;
    QString currentIDUser = "", currentIDOrder = "", previousIDMaster = "";
    int previousStatusIndex = -1;
    bool checkState = false;
    QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL", "order");
```

```
    void setCurrentText(QComboBox *, QString);
    void updateClientsCMBB();
    void fillDataToCMBB();
    void loadOrderComments();
    QString createMessage(QSqlQuery);
};
#endif // ORDERFORM_H
```

Теперь по тихой грусти начнем потихоньку работать над реализацией всех идей и надежд, возложенных на этот модуль. Начнем с малого – с выпадающих списков.

Итак, стартанем с метода fillDataToCMBB(). Его суть предельно проста: объявляем подключение к БД по метке модуля, проверяем подключение, и, если оно присутствует, то для каждого выпадающего списка на панели PNL_Main формируем запрос на выборку значений из справочников, и добавляем эти значения в них, а после – вручную устанавливаем значение «-1» для свойства currentIndex, чтобы ни одно значение не было выбрано из выпадающих списков, поскольку по-умолчанию программа всегда установит это значение на «0», и выберет первый из возможных.

```
void OrderForm::fillDataToCMBB()
{
    db = QSqlDatabase::database("order");
    if(getSqlConnection(db))
    {
        QSqlQuery query(db);
        query.exec("select DeviceTypeValue from DeviceType;");
        while(query.next())
        {
            ui->CMBB_TypeDevice->addItem(query.value(0).toString());
        }
        query.exec("select PriorityValue from OrderPriority;");
        while(query.next())
        {
            ui->CMBB_Priority->addItem(query.value(0).toString());
        }
        query.exec("select TypeDefectValue from TypeDefect;");
        while(query.next())
        {
            ui->CMBB_TypeDefect->addItem(query.value(0).toString());
        }
        query.exec("select OrderStatusValue from OrderStatus;");
        while(query.next())
        {
            ui->CMBB_Status->addItem(query.value(0).toString());
        }
      query.exec("select concat(UserSurname, ' ', UserName, ' ', UserPatronymic) from Users where UserRole
= 2;");
        while(query.next())
        {
            ui->CMBB_Master->addItem(query.value(0).toString());
        }
        ui->CMBB_TypeDevice->setCurrentIndex(-1);
        ui->CMBB_Priority->setCurrentIndex(-1);
```

```
        ui->CMBB_TypeDefect->setCurrentIndex(-1);
        ui->CMBB_Status->setCurrentIndex(-1);
        ui->CMBB_Master->setCurrentIndex(-1);
    }
    else
    {
        return;
    }
}
```

Далее проделываем все то же самое, но для метода updateClientsCMBB(). Единственная разница заключается в том, что выпадающий список с клиентами должен быть принудительно очищен от предыдущих значений перед добавлением новых, дабы не возникло задублирования одних и тех же записей клиентов.

```
void OrderForm::updateClientsCMBB()
{
    db = QSqlDatabase::database("order");
    if(getSqlConnection(db))
    {
        ui->CMBB_Clients->clear();
        QSqlQuery query(db);
      query.exec("select concat(UserSurname, ' ', UserName, ' ', UserPatronymic) from Users where UserRole
= 3;");
        while(query.next())
        {
            ui->CMBB_Clients->addItem(query.value(0).toString());
        }
        ui->CMBB_Clients->setCurrentIndex(-1);
    }
    else
    {
        return;
    }
}
```

После того, как в выпадающих списках появились хоть какие-то данные, следует настроить работу панелей так, чтобы пользователю было удобно работать, а выполнение работы модуля не скатилась в чехарду. Сперва настроим отображение панели добавления нового клиента через метод startCreateNewClient(), а после – отмены добавления, используя метод cancelCreateNewClient().

```
void OrderForm::startCreateNewClient()
{
    ui->PNL_Client->setVisible(true);
    ui->PNL_Main->setVisible(false);
}

void OrderForm::cancelCreateNewClient()
{
    ui->PNL_Client->setVisible(false);
    ui->PNL_Main->setVisible(true);
    ui->LE_Surname->clear();
    ui->LE_Name->clear();
```

```
    ui->LE_Patronymic->clear();
    ui->LE_Phone->clear();
}
```

Отлично, работаем поступательно, коль мы начали работать над панелью добавления нового клиента, стоит сразу же прикрутить к модулю обработчик на нажатие кнопки «Добавить», который называется addNewClient(). Его алгоритм работы можно свести к следующему поведению:

1. Проверяем все поля ввода на заполненность данными, для этого удалим все символы пробелов из каждой строки, и ожидаем, что длина получившихся строк внутри полей ввода больше 0 (а для поля «Номер телефона» равным 15 с пробелами, поскольку на поле LE_Phone должна стоять маска ввода формата (XXX) XXX-XX-XX, изменяемая через свойство inputMask);
2. Если это правда, то все поля ввода заполнены, программа должна настроить подключение к БД от текущего экземпляра подключения QSqlDatabase и проверить подключение;
    a. Если подключение установлено, то программа должна добавить новую запись о клиенте в БД, оповестить об этом пользователя, а после – перезагрузить выпадающий список со всеми клиентами и вернуть пользователя в панель работы с заявкой;
    b. Если подключение не установлено, то программа должна выдать ошибку по отсутствию подключения к БД;
3. Если неправда, то программа должна определить поле, которое не было заполнено, и сообщить пользователю ошибку, оповещающую о том, что это поле не должно быть пустым.

Алгоритм предельно простой, посмотрим на то, как он реализуется в программном коде:

```
void OrderForm::addNewClient()
{
    if(ui->LE_Surname->text().remove(" ").length() > 0)
    {
        if(ui->LE_Name->text().remove(" ").length() > 0)
        {
            if(ui->LE_Patronymic->text().remove(" ").length() > 0)
            {
                if(ui->LE_Phone->text().length() == 15)
                {
                    db = QSqlDatabase::database("order");
                    if(getSqlConnection(db))
                    {
                        QSqlQuery query(db);
                  query.prepare("insert into Users (UserSurname, UserName, UserPatronymic, UserPhone,
UserRole) values (?,?,?,?,'3');");
                        query.addBindValue(ui->LE_Surname->text());
                        query.addBindValue(ui->LE_Name->text());
                        query.addBindValue(ui->LE_Patronymic->text());
                        query.addBindValue(ui->LE_Phone->text());
                        query.exec();
```

```
                showMessageBox("Новый клиент успешно добавлен!", "Успешно", QMessageBox::Ok,
QMessageBox::Information);
                    updateClientsCMBB();
                    cancelCreateNewClient();
                }
                else
                {
                    showMessageBox("Отсутствует подключение к БД!", "Ошибка модуля",
QMessageBox::Ok, QMessageBox::Critical);
                    return;
                }
            }
            else
            {
              showMessageBox("Поле 'Телефон' должно быть заполнено!", "Действие прервано",
QMessageBox::Ok, QMessageBox::Warning);
                return;
            }
        }
        else
        {
            showMessageBox("Поле 'Отчество' должно быть заполнено!", "Действие прервано",
QMessageBox::Ok, QMessageBox::Warning);
            return;
        }
    }
    else
    {
        showMessageBox("Поле 'Имя' должно быть заполнено!", "Действие прервано",
QMessageBox::Ok, QMessageBox::Warning);
        return;
    }
}
else
{
    showMessageBox("Поле 'Фамилия' должно быть заполнено!", "Действие прервано",
QMessageBox::Ok, QMessageBox::Warning);
    return;
}
}
```

Поработаем над изменением значений в выпадающих списках, у нас таких два, как было сказано ранее, начнем с исполнителя, за который отвечает метод updateStatusByMaster(). Программа должна проверить, что если значение свойства currentIndex для этого выпадающего списка больше «-1», то это значит, что пользователь выбрал нового исполнителя для заявки, причем совершенно неважно какого, и значение в выпадающем списке статуса заявки необходимо изменить на «На выполнении».

Смена значений в выпадающих списках по значениям строк, а не по индексу мы все дружно повесили на метод setCurrentText(QComboBox *, QString), поэтому стоит сперва добавить его реализацию, базирующейся на следующем алгоритме:

1. Перебираем все значения, хранящиеся внутри выбранного выпадающего списка;

2. Если текущий перебираемый элемент по своему значению равен строке, пришедшей на вход аргументом, то:

a. Устанавливаем свойство currentIndex от текущего значения переменной цикла перебора элементов выпадающего списка;

b. Если псевдоним текущего выпадающего списка равен псевдониму выпадающего списка приоритетов заявки, то программа должна запомнить выбранный индекс текущего элемента внутри него, чтобы позднее это значение можно было вернуть, иначе просто ничего не делаем и идем далее по коду;

c. Завершаем работу всей функции, поскольку результаты достигнуты.

Программный код метода setCurrentText(QComboBox *, QString) представлен ниже.

```
void OrderForm::setCurrentText(QComboBox *cmb, QString value)
{
    for(int i = 0; i < cmb->count(); i++)
    {
        if(cmb->itemText(i) == value)
        {
            if(cmb->objectName() == "CMBB_Status")
            {
                previousStatusIndex = i;
            }
            cmb->setCurrentIndex(i);
            return;
        }
    }
}
```

Вот теперь можно рассмотреть программный код метода updateStatusByMaster().

```
void OrderForm::updateStatusByMaster()
{
    if(ui->CMBB_Master->currentIndex() > -1)
    {
        setCurrentText(ui->CMBB_Status, "На выполнении");
    }
}
```

Следующий на очереди – обработчик на изменения значений в выпадающем списке статусов заявок, его алгоритм уже куда забористее, чем предыдущий:

1. Если выбранное значение из выпадающего списка статусов заявок равняется «Выполнена», и при этом программа знает ID текущей заявки, а последний запомненный индекс из этого же выпадающего списка не равен тому, который только что был выбран, т.е. тот, что не указывает на элемент «Выполнена», то:

a. Программа спрашивает пользователя о том, нужно ли действительно менять значение статуса заявки на «Выполнена», поскольку пользователь мог сделать это случайно, и, если результат ответа пользователя положительный, то:

I. Программа настраивает подключение к БД, и если у нее получается это сделать, то она получает из БД ID статуса заявки, который

соответствует значению «Выполнена», и обновляет данные по самой заявке в БД, обновляя и дату закрытия заявки, а после – блокирует поля ввода информации для панели работы с заявкой, и полностью блокирует возможность добавления новых комментариев к ней, и оповещает пользователя об успешной закрытии заявки;

II. Иначе программа выдает пользователю сообщение об ошибке, оповещающее о проблемах подключением к БД;

b. Если ответ пользователя отрицательный, то программа меняет свойство currentIndex для текущего выпадающего списка на последний запомненный до текущего выбора.

Выглядит уже чуть сложнее, посмотрим на то, как оно реализовывается в программном коде:

```
void OrderForm::updateStatusByClosed()
{
    if(ui->CMBB_Status->currentText() == "Выполнена" && currentIDOrder != "" && previousStatusIndex != ui->CMBB_Status->currentIndex())
    {
        QList<QMessageBox::Button> buttons;
        buttons.append(QMessageBox::Cancel);
        buttons.append(QMessageBox::Ok);
        if(showMessageBox("Вы действительно хотите закрыть заявку?", "Подтвердите ввод", buttons, QMessageBox::Question) == QMessageBox::Ok)
        {
            db = QSqlDatabase::database("order");
            if(getSqlConnection(db))
            {
                QString ID_Status = "";
                QSqlQuery query(db);
                query.prepare("SELECT OrderStatusID FROM OrderStatus where OrderStatusValue = ?;");
                query.addBindValue(ui->CMBB_Status->currentText());
                query.exec();
                if(query.next())
                {
                    ID_Status = query.value(0).toString();
                }
                else
                {
                    showMessageBox("Ошибка получения ID статуса заявки!", "Ошибка модуля", QMessageBox::Ok, QMessageBox::Critical);
                    return;
                }

                query.prepare("update Orders set OrderStatus = ?, OrderClosedDate = ? where OrderID = ?;");
                query.addBindValue(ID_Status);
                query.addBindValue(QDate::currentDate().toString("yyyy-MM-dd"));
                query.addBindValue(currentIDOrder);
                if(query.exec())
                {
                    ui->TAB_Comments->setEnabled(false);
```

```
                ui->TAB_Comments->setVisible(false);

                ui->DTP_CloseDate->setDate(QDate::currentDate());
                ui->DTP_CloseDate->setVisible(true);
                ui->LB_CloseDate->setVisible(true);
                ui->CMBB_Master->setEnabled(false);
                ui->TE_Desc->setReadOnly(true);
                ui->CMBB_Priority->setEnabled(false);
                ui->CMBB_Status->setEnabled(false);

                showMessageBox("Заявка успешно закрыта!", "Успешно", QMessageBox::Ok,
QMessageBox::Information);
                return;
            }
            else
            {
                showMessageBox("Возникла ошибка при закрытии заявки в БД.\nПроверьте
ввод данных и повторите попытку.", "Неудачно", QMessageBox::Ok, QMessageBox::Critical);
                return;
            }
        }
        else
        {
            showMessageBox("Отсутствует подключение к БД!", "Ошибка модуля",
QMessageBox::Ok, QMessageBox::Critical);
            return;
        }
    }
    else
    {
        ui->CMBB_Status->setCurrentIndex(previousStatusIndex);
    }
  }
}
```

Теперь на очереди просто галактических масштабов обработчик acceptOrderChanges(), который будет висеть на кнопке BTN_Apply. Да, это там самая кнопка, которая добавляет новую заявку или редактирует данные уже существующей. Без малого 300 строчек кода на один метод, и алгоритм для будет страницы на 2-3, но это необходимость, это очень важная кнопка, которая и совершает основной объем работы, попробуем разобрать, что же именно она должна делать и каким именно образом. Во-первых, программа должна настроить подключение к БД, и, если таковое получено, то программа разделяет свое выполнение относительно флага работы модуля: «Добавление» или «Изменение».

Проанализируем, что программа должна делать в случае работы модуля в режиме «Добавление»:

1. Программа проверяет, чтобы все поля ввода были заполнены, включая выпадающие списки. Если это так, то:
    a. Программа определяет временные переменные для хранения ID всех значений выпадающих списков, которые имеются на форме, а затем последовательно обращается к БД для их получения, и если у нее получается

получить ID какой-либо записи из требуемого справочника, то она сохраняет их в переменные, а иначе – возвращает сообщение об ошибке и принудительно завершает работу всей функции;

b. Программа формирует запрос INSERT INTO для добавления новой заявки, и пытается провести этот запрос на сервере, если у нее не получается, то она возвращает сообщение об ошибке и принудительно завершает работу всей функции, иначе:

I. Программа анализирует поле ввода исполнителя, если оно заполнено, то она вызывает значение ID выбранного мастера из БД, сохраняет его, а после – добавляет запись об этом в БД, выдает сообщение об успешно добавленной записи по новой заявке, и завершает работу модуля;

II. Иначе программа понимает, что исполнитель на этой стадии еще не назначен, и выдает сообщение об успешно добавленной записи по новой заявке, и завершает работу модуля;

2. Если программа обнаруживает незаполненное поле, то она уведомляет об этом пользователя, и принудительно завершает работу всей функции.

Теперь рассмотрим алгоритм работы функции acceptOrderChanges() в случае работы модуля в режиме «Изменение»:

1. Программа определяет временные переменные для хранения ID требуемых для изменений значений выпадающих списков, которые имеются на форме, а затем последовательно обращается к БД для их получения, и если у нее получается получить ID какой-либо записи из требуемого справочника, то она сохраняет их в переменные, а иначе – возвращает сообщение об ошибке и принудительно завершает работу всей функции;

2. Программа формирует запрос UPDATE table SET...WHERE для изменения текущей заявки по ее ID, и пытается провести этот запрос на сервере, если у нее не получается, то она возвращает сообщение об ошибке и принудительно завершает работу всей функции, иначе:

a. Программа анализирует поле ввода исполнителя, если оно заполнено, и при этом значение последнего измененного исполнителя не равно текущему, то она вызывает значение ID выбранного мастера из БД, сохраняет его, а после – удаляет предыдущую запись об исполнителе заявки по ее ID, добавляет новую запись в БД, выдает сообщение об успешно добавленной записи по новой заявке, и завершает работу модуля;

b. Иначе программа понимает, что исполнитель не был изменен в процессе работы модуля, и выдает сообщение об успешно добавленной записи по новой заявке, и завершает работу всего модуля.

Выглядит не слишком сложно, однако, программа учитывает множество исключений, которые могут возникнуть в процессе обработки действия, поэтому ее код постоянно исследует состояние значений, получаемых как от БД, так и от самого пользователя, чтобы данные были однозначно изменены в БД, неважно, что происходит при этом, добавление или изменение данных. Теперь посмотрим на полный код этого обработчика:

```
void OrderForm::acceptOrderChanges()
{
    db = QSqlDatabase::database("order");
    if(getSqlConnection(db))
    {
        if(!checkState)
        {
            if(ui->CMBB_Clients->currentIndex() != -1)
            {
                if(ui->LE_DeviceName->text().remove(" ").length() > 0)
                {
                    if(ui->CMBB_TypeDevice->currentIndex() != -1)
                    {
                        if(ui->TE_Desc->toPlainText().remove(" ").length() > 0)
                        {
                            if(ui->CMBB_TypeDefect->currentIndex() != -1)
                            {
                                if(ui->CMBB_Priority->currentIndex() != -1)
                                {
                                    if(ui->CMBB_Status->currentIndex() != -1)
                                    {
                                        QSqlQuery query(db);
                                        QString ID_TypeDevice = "",
                                                ID_TypeDefect = "",
                                                ID_Priority = "",
                                                ID_Status = "",
                                                ID_Client = "",
                                                ID_Master = "";

                                        query.prepare("select UserID from Users where concat(UserSurname,
' ', UserName, ' ', UserPatronymic) = ?;");
                                        query.addBindValue(ui->CMBB_Clients->currentText());
                                        query.exec();
                                        if(query.next())
                                        {
                                            ID_Client = query.value(0).toString();
                                        }
                                        else
                                        {
                                            showMessageBox("Ошибка получения ID клиента!", "Ошибка
модуля", QMessageBox::Ok, QMessageBox::Critical);
                                            return;
                                        }

                                        query.prepare("select DeviceTypeID from DeviceType where
DeviceTypeValue = ?;");
                                        query.addBindValue(ui->CMBB_TypeDevice->currentText());
                                        query.exec();
                                        if(query.next())
                                        {
                                            ID_TypeDevice = query.value(0).toString();
                                        }
```

```
                                    else
                                    {
                                        showMessageBox("Ошибка получения ID типа устройства!",
"Ошибка модуля", QMessageBox::Ok, QMessageBox::Critical);
                                        return;
                                    }

                                    query.prepare("select PriorityID from OrderPriority where PriorityValue
= ?;");
                                    query.addBindValue(ui->CMBB_Priority->currentText());
                                    query.exec();
                                    if(query.next())
                                    {
                                        ID_Priority = query.value(0).toString();
                                    }
                                    else
                                    {
                                        showMessageBox("Ошибка получения ID приоритета заявки!",
"Ошибка модуля", QMessageBox::Ok, QMessageBox::Critical);
                                        return;
                                    }

                                    query.prepare("select OrderStatusID from OrderStatus where
OrderStatusValue = ?;");
                                    query.addBindValue(ui->CMBB_Status->currentText());
                                    query.exec();
                                    if(query.next())
                                    {
                                        ID_Status = query.value(0).toString();
                                    }
                                    else
                                    {
                                        showMessageBox("Ошибка получения ID статуса заявки!",
"Ошибка модуля", QMessageBox::Ok, QMessageBox::Critical);
                                        return;
                                    }

                                    query.prepare("select TypeDefectID from TypeDefect where
TypeDefectValue = ?;");
                                    query.addBindValue(ui->CMBB_TypeDefect->currentText());
                                    query.exec();
                                    if(query.next())
                                    {
                                        ID_TypeDefect = query.value(0).toString();
                                    }
                                    else
                                    {
                                        showMessageBox("Ошибка получения ID типа дефекта
устройства!", "Ошибка модуля", QMessageBox::Ok, QMessageBox::Critical);
                                        return;
                                    }
```

```
                query.prepare("insert into Orders (OrderStartDate,
OrderDeviceName, OrderTypeDefect, OrderDescription, OrderClient, OrderPriority, OrderStatus,
OrderTypeDevice) values (?, ?, ?, ?, ?, ?, ?, ?);");
                query.addBindValue(QDate::currentDate().toString("yyyy-MM-dd"));
                query.addBindValue(ui->LE_DeviceName->text());
                query.addBindValue(ID_TypeDefect);
                query.addBindValue(ui->TE_Desc->toPlainText());
                query.addBindValue(ID_Client);
                query.addBindValue(ID_Priority);
                query.addBindValue(ID_Status);
                query.addBindValue(ID_TypeDevice);
                if(query.exec())
                {
                    if(ui->CMBB_Master->currentIndex() != -1)
                    {
                        query.exec("select max(OrderID) from Orders;");
                        if(query.next())
                        {
                            currentIDOrder = query.value(0).toString();
                        }
                        else
                        {
                            showMessageBox("Ошибка получения ID последней
добавленной заявки!", "Ошибка модуля", QMessageBox::Ok, QMessageBox::Critical);
                            return;
                        }

                        query.prepare("select UserID from Users where
concat(UserSurname, ' ', UserName, ' ', UserPatronymic) = ?;");
                        query.addBindValue(ui->CMBB_Master->currentText());
                        if(query.next())
                        {
                            ID_Master = query.value(0).toString();
                        }
                        else
                        {
                            showMessageBox("Ошибка получения ID мастера!",
"Ошибка модуля", QMessageBox::Ok, QMessageBox::Critical);
                            return;
                        }

                        query.prepare("insert into OrderToMaster (OrderID, MasterID)
values (?,?);");
                        query.addBindValue(currentIDOrder);
                        query.addBindValue(ID_Master);
                        if(query.exec())
                        {
                            showMessageBox("Новая заявка успешно создана!",
"Успешно", QMessageBox::Ok, QMessageBox::Information);
                            this->close();
                        }
                        else
```

```
                                        {
                                            showMessageBox("Возникла ошибка обновления данных о мастере в БД.\nПроверьте ввод данных и повторите попытку.", "Неудачно", QMessageBox::Ok, QMessageBox::Critical);
                                            return;
                                        }
                                    }
                                    else
                                    {
                                        showMessageBox("Новая заявка успешно создана!", "Успешно", QMessageBox::Ok, QMessageBox::Information);
                                        this->close();
                                    }
                                }
                                else
                                {
                                    showMessageBox("Возникла ошибка добавления данных о заявке в БД.\nПроверьте ввод данных и повторите попытку.", "Неудачно", QMessageBox::Ok, QMessageBox::Critical);
                                    return;
                                }
                            }
                            else
                            {
                                showMessageBox("Поле 'Статус заявки' должно быть заполнено!", "Действие прервано", QMessageBox::Ok, QMessageBox::Warning);
                                return;
                            }
                        }
                        else
                        {
                            showMessageBox("Поле 'Приоритет заявки' должно быть заполнено!", "Действие прервано", QMessageBox::Ok, QMessageBox::Warning);
                            return;
                        }
                    }
                    else
                    {
                        showMessageBox("Поле 'Тип дефекта' должно быть заполнено!", "Действие прервано", QMessageBox::Ok, QMessageBox::Warning);
                        return;
                    }
                }
                else
                {
                    showMessageBox("Поле 'Описание' должно быть заполнено!", "Действие прервано", QMessageBox::Ok, QMessageBox::Warning);
                    return;
                }
            }
            else
            {
```

```
                    showMessageBox("Поле 'Тип устройства' должно быть заполнено!", "Действие прервано", QMessageBox::Ok, QMessageBox::Warning);
                    return;
                }
            }
            else
            {
                showMessageBox("Поле 'Наименование устройства' должно быть заполнено!", "Действие прервано", QMessageBox::Ok, QMessageBox::Warning);
                return;
            }
        }
        else
        {
            showMessageBox("Поле 'Клиент' должно быть заполнено!", "Действие прервано", QMessageBox::Ok, QMessageBox::Warning);
            return;
        }
    }
    else
    {
        QSqlQuery query(db);

        QString ID_Priority = "",
                ID_Status = "",
                ID_Master = "",
                ID_PrevMaster = "";

        query.prepare("select PriorityID from OrderPriority where PriorityValue = ?;");
        query.addBindValue(ui->CMBB_Priority->currentText());
        query.exec();
        if(query.next())
        {
            ID_Priority = query.value(0).toString();
        }
        else
        {
            showMessageBox("Ошибка получения ID приоритета заявки!", "Ошибка модуля", QMessageBox::Ok, QMessageBox::Critical);
            return;
        }

        query.prepare("select OrderStatusID from OrderStatus where OrderStatusValue = ?;");
        query.addBindValue(ui->CMBB_Status->currentText());
        query.exec();
        if(query.next())
        {
            ID_Status = query.value(0).toString();
        }
        else
        {
```

```
                showMessageBox("Ошибка получения ID статуса заявки!", "Ошибка модуля",
QMessageBox::Ok, QMessageBox::Critical);
                return;
            }

            if(ui->CMBB_Master->currentIndex() > -1)
            {
                query.prepare("select UserID from Users where concat(UserSurname, ' ', UserName, ' ',
UserPatronymic) = ?;");
                query.addBindValue(ui->CMBB_Master->currentText());
                query.exec();
                if(query.next())
                {
                    ID_Master = query.value(0).toString();
                }
                else
                {
                    showMessageBox("Ошибка получения ID мастера!", "Ошибка модуля",
QMessageBox::Ok, QMessageBox::Critical);
                    return;
                }
            }

            query.prepare("update Orders set OrderDescription = ?, OrderPriority = ?, OrderStatus = ?
where OrderID = ?;");
            query.addBindValue(ui->TE_Desc->toPlainText());
            query.addBindValue(ID_Priority);
            query.addBindValue(ID_Status);
            query.addBindValue(currentIDOrder);
            if(query.exec())
            {
                if(ID_Master != previousIDMaster && ui->CMBB_Master->currentIndex() > -1)
                {
                    query.prepare("delete from OrderToMaster where OrderID = ?;");
                    query.addBindValue(currentIDOrder);
                    query.exec();

                    query.prepare("insert into OrderToMaster (OrderID, MasterID) values (?,?);");
                    query.addBindValue(currentIDOrder);
                    query.addBindValue(ID_Master);
                    if(query.exec())
                    {
                        showMessageBox("Заявка № "+ currentIDOrder +" успешно обновлена!",
"Успешно", QMessageBox::Ok, QMessageBox::Information);
                        this->close();
                    }
                    else
                    {
                        showMessageBox("Возникла ошибка обновления данных о мастере в
БД.\nПроверьте ввод данных и повторите попытку.", "Неудачно", QMessageBox::Ok,
QMessageBox::Critical);
                    }
```

```
                }
                else
                {
                    showMessageBox("Заявка № "+ currentIDOrder +" успешно обновлена!",
"Успешно", QMessageBox::Ok, QMessageBox::Information);
                    this->close();
                }
            }
            else
            {
                showMessageBox("Возникла ошибка обновления основных данных в БД.\nПроверьте
ввод данных и повторите попытку.", "Неудачно", QMessageBox::Ok, QMessageBox::Critical);
            }
        }
    }
    else
    {
        showMessageBox("Отсутствует подключение к БД!", "Ошибка модуля", QMessageBox::Ok,
QMessageBox::Critical);
        return;
    }
}
```

Вернемся немного назад и вспомним про возможность комментирования. После последнего рассмотренного метода работы кнопки «Применить» тут сейчас будет происходить по истине отдых. Сперва рассмотрим вспомогательный метод QString createMessage(QSqlQuery), который формирует готовое сообщение для его последующего вывода. В нем нет ничего сложного, он просто формирует строку QString по определенному шаблону, «подкидывая» в нужные места данные, полученные из текущей строки виртуальной таблицы БД результата запроса на выборку всех сообщений по заявке.

```
QString OrderForm::createMessage(QSqlQuery query)
{
    return query.value(0).toString() + " в " + query.value(2).toDateTime().toString("dd.MM.yyyy hh:mm:ss") +
":\n" + query.value(1).toString() + "\n_____\n";
}
```

По итогу, мы получим сообщение следующего характера, которое отражено на рисунке 43.

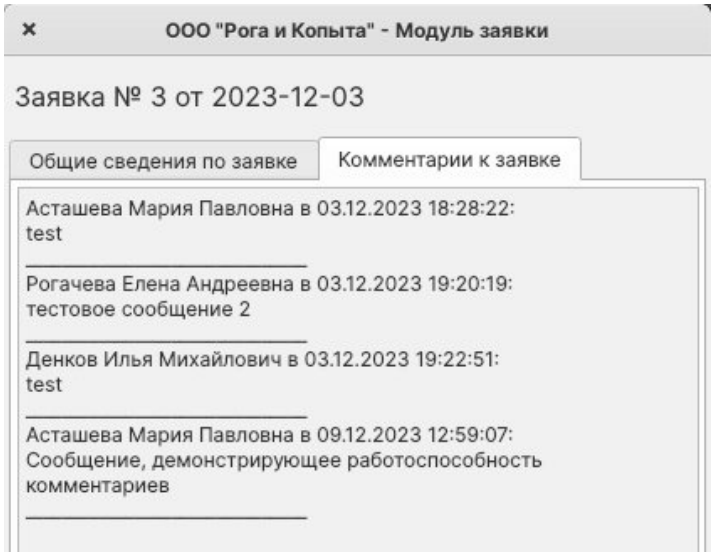

Рисунок 43 – Список комментариев к заявке № 3

Функция loadOrderComments() будет загружать из БД данные по всем комментариям, оставленным к заявке и добавлять их в поле вывода данных TE_CommentsList, предварительно очищая это поле от предыдущих комментариев, чтобы они не наслаивались друг на друга.

```
void OrderForm::loadOrderComments()
{
    ui->TE_CommentsList->clear();
    db = QSqlDatabase::database("order");
    if(getSqlConnection(db))
    {
        QSqlQuery query(db);
        query.prepare("select concat(Users.UserSurname, ' ', Users.UserName, ' ', Users.UserPatronymic) as
FIO, CommentariesInOrders.CommentText, CommentariesInOrders.CommentDate,
CommentariesInOrders.OrderID from CommentariesInOrders inner join Users on Users.UserID =
CommentariesInOrders.MasterID where OrderID = ?;");
        query.addBindValue(currentIDOrder);
        query.exec();
        while(query.next())
        {
            ui->TE_CommentsList->setPlainText(ui->TE_CommentsList->toPlainText()+createMessage(query));
        }
    }
    else
    {
        showMessageBox("Отсутствует подключение к БД!", "Ошибка модуля", QMessageBox::Ok,
QMessageBox::Critical);
        return;
    }
}
```

И последний обработчик этой вкладки – обработчик pushNewCommentToOrder() на нажатие кнопки BTN_PushComment, который проверяет заполненность поля ввода нового комментария, и в случае такового, добавляет новый коммент к заявке в БД, а после –

очищает поле ввода нового комментария и перезагружает элемент вывода всех комментариев, заполняя его новыми данными.

```
void OrderForm::pushNewCommentToOrder()
{
    if(ui->TE_NewComment->toPlainText().remove(" ").length() > 0)
    {
        db = QSqlDatabase::database("order");
        if(getSqlConnection(db))
        {
            QSqlQuery query(db);
            query.prepare("insert into CommentariesInOrders (OrderID, MasterID, CommentText,
CommentDate) values (?,?,?,?);");
            query.addBindValue(currentIDOrder);
            query.addBindValue(currentIDUser);
            query.addBindValue(ui->TE_NewComment->toPlainText());
            query.addBindValue(QDateTime::currentDateTime().toString("yyyy-MM-dd hh:mm:ss"));
            if(query.exec())
            {
                ui->TE_NewComment->clear();
                loadOrderComments();
            }
            else
            {
                showMessageBox("Возникла необрабатываемая ошибка при отравке
сообщения.\nПроверьте данные и повторите ввод", "Ошибка модуля", QMessageBox::Ok,
QMessageBox::Critical);
                return;
            }
        }
        else
        {
            showMessageBox("Отсутствует подключение к БД!", "Ошибка модуля", QMessageBox::Ok,
QMessageBox::Critical);
            return;
        }
    }
    else
    {
        showMessageBox("Поле 'Комментарий' должно быть заполнено!", "Действие прервано",
QMessageBox::Ok, QMessageBox::Warning);
        return;
    }
}
```

Последний метод в этом классе, но не по значению – это метод updateForm(), который занимается преднастройкой всей формы, перед отображением ее пользователю на экран. Метод работает по следующему алгоритму:

1. Программа предзагружает данные во все выпадающие списки на форме;
2. Если модуль будет работать в режиме «Добавление», то программа отключает вкладку «Комментарии в заявке», и скрывает элемент вывода даты и соответствующей ему надписи с формы;

3. Если модуль будет работать в режиме «Изменение», то программа настраивает подключение к БД, и, если подключение работает, то:

a. Программа получает данные по всей основной информации по заявке из БД и заполняет ими поля ввода данных;

b. Программа запрашивает данные по исполнителю заявки, и если получает их от БД, то заполняет поле ввода «Мастер:» и запоминает ID последнего измененного исполнителя в своей памяти;

c. Программа анализирует переменную, хранящую ID последнего измененного исполнителя, если он не равен ID текущего пользователя, работающего с модулем, или он вообще пуст, то программа отключает возможность комментирования заявки, скрывая нужные элементы управления с формы;

d. Программа отключает стандартные поля ввода данных, которые не должны изменяться по техническому заданию, причем если статус заявки равен «Выполнена», то программа отключит вообще все поля ввода, включая комментарии;

e. После всех вышеописанных манипуляций, программа загрузит комментарии по заявке.

Теперь же посмотрим, как этот метод выглядит в программных кодах:

```
void OrderForm::updateForm()
{
    updateClientsCMBB();
    fillDataToCMBB();

    if(!checkState)
    {
        ui->TAB_Comments->setEnabled(false);
        ui->TAB_Comments->setVisible(false);
        ui->LB_CloseDate->setVisible(false);
        ui->DTP_CloseDate->setVisible(false);
    }
    else
    {
        db = QSqlDatabase::database("order");
        if(getSqlConnection(db))
        {
            QSqlQuery query(db);
            query.prepare("select * from showShortOrderInfo where OrderID = ?;");
            query.addBindValue(currentIDOrder);
            query.exec();
            if(query.next())
            {
                ui->LB_OrderName->setText("Заявка № " + query.value(0).toString() + " от " +
query.value(1).toString());
                ui->LE_DeviceName->setText(query.value(2).toString());
                setCurrentText(ui->CMBB_TypeDefect, query.value(3).toString());
                ui->TE_Desc->setPlainText(query.value(4).toString());
                setCurrentText(ui->CMBB_Clients, query.value(5).toString());
                setCurrentText(ui->CMBB_Priority, query.value(6).toString());
                setCurrentText(ui->CMBB_Status, query.value(7).toString());
                if(query.value(8).toString() != "")
```

```
            {
                ui->DTP_CloseDate->setDate(query.value(8).toDate());
            }
            else
            {
                ui->LB_CloseDate->setVisible(false);
                ui->DTP_CloseDate->setVisible(false);
            }
            setCurrentText(ui->CMBB_TypeDevice, query.value(9).toString());
        }

        query.prepare("select MasterID from OrderToMaster where OrderID = ?;");
        query.addBindValue(currentIDOrder);
        query.exec();
        if(query.next())
        {
            previousIDMaster = query.value(0).toString();
        }

        if(previousIDMaster != "")
        {
            query.prepare("select concat(UserSurname, ' ', UserName, ' ', UserPatronymic) from Users
where UserID = ?;");
            query.addBindValue(previousIDMaster);
            query.exec();
            if(query.next())
            {
                setCurrentText(ui->CMBB_Master, query.value(0).toString());
            }
        }
        if(previousIDMaster != currentIDUser || previousIDMaster == "")
        {
            ui->BTN_PushComment->setVisible(false);
            ui->TE_NewComment->setVisible(false);
        }
        ui->CMBB_Clients->setEnabled(false);
        ui->CMBB_TypeDefect->setEnabled(false);
        ui->CMBB_TypeDevice->setEnabled(false);
        ui->LE_DeviceName->setReadOnly(true);
        ui->BTN_CreateNewClient->setVisible(false);
        if(ui->CMBB_Status->currentText() == "Выполнена")
        {
            ui->TAB_Comments->setEnabled(false);
            ui->TAB_Comments->setVisible(false);

            ui->DTP_CloseDate->setVisible(true);
            ui->LB_CloseDate->setVisible(true);
            ui->CMBB_Master->setEnabled(false);
            ui->TE_Desc->setReadOnly(true);
            ui->CMBB_Priority->setEnabled(false);
            ui->CMBB_Status->setEnabled(false);
        }
```

```
            loadOrderComments();
        }
    }
}
```

Осталось изменить работу конструктора класса, соединив все объекты на форме со своими обработчиками сигналов, т.е. со слотами. Помимо этого, мы запретим возможность изменения размеров формы, потому что мы не занимались тем, чтобы сделать масштабируемую верстку, и будет лучше, если пользователь вообще лишится возможности изменять размеры окна, они останутся такими, какими их задумали мы, программисты. Помимо этого, на будет необходимо скрыть панель добавления нового клиента с формы, поскольку на этом этапе, т.е. в момент, когда пользователю будет отображена форма, ее не было видно, за ее отображение будет отвечать отдельная кнопка.

```
OrderForm::OrderForm(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::OrderForm)
{
    ui->setupUi(this);

    this->setFixedSize(QSize(this->width(), this->height()));

    ui->PNL_Client->setVisible(false);

    connect(ui->BTN_CreateNewClient, &QPushButton::clicked, this, &OrderForm::startCreateNewClient);
    connect(ui->BTN_AddNewClient, &QPushButton::clicked, this, &OrderForm::addNewClient);
    connect(ui->BTN_Cancel, &QPushButton::clicked, this, &OrderForm::cancelCreateNewClient);
    connect(ui->BTN_Apply, &QPushButton::clicked, this, &OrderForm::acceptOrderChanges);
    connect(ui->BTN_PushComment, &QPushButton::clicked, this, &OrderForm::pushNewCommentToOrder);
    connect(ui->CMBB_Master, &QComboBox::currentIndexChanged, this, &OrderForm::updateStatusByMaster);
    connect(ui->CMBB_Status, &QComboBox::currentIndexChanged, this, &OrderForm::updateStatusByClosed);
}
```

Ну вот и все, работа над модулем завершена. Сделаем небольшой контрольный пример, показывающий, что вся программная логика, которую мы запланировали, работает корректно.

Начну демонстрацию возможностей модуля с ее работы в режиме «Добавление». Пользователю при запуске будет отображено окно, продемонстрированное на рисунке 44.

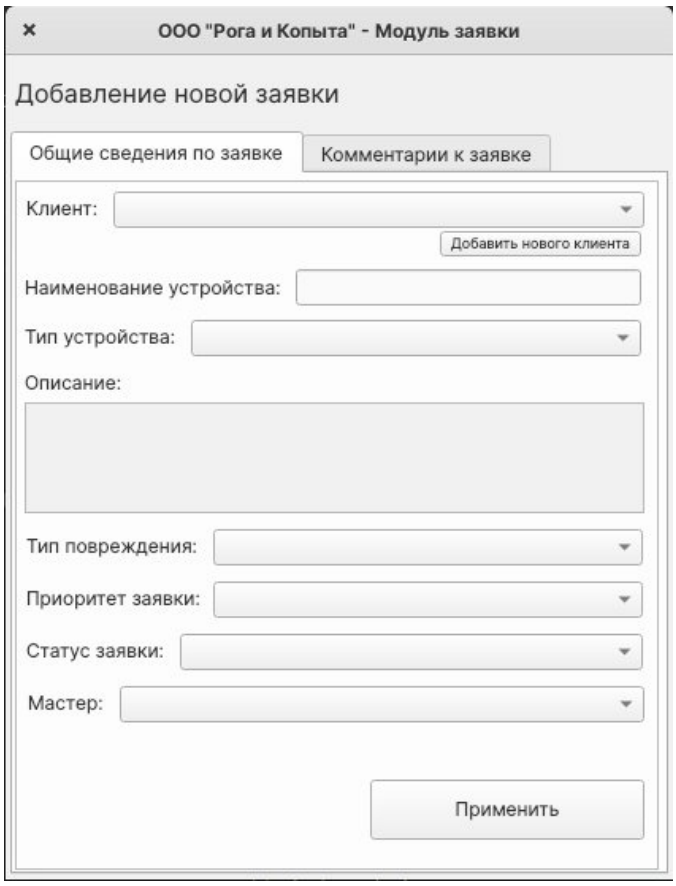

Рисунок 44 – Окно модуля заявки

Как можно видеть, все поля ввода доступны для изменений, и при этом они изначально пустые, т.е. это ровно то, что там и необходимо. Проверим содержимое какого-либо выпадающего списка, пусть это будет «Тип повреждения», которое отображено на рисунке 45.

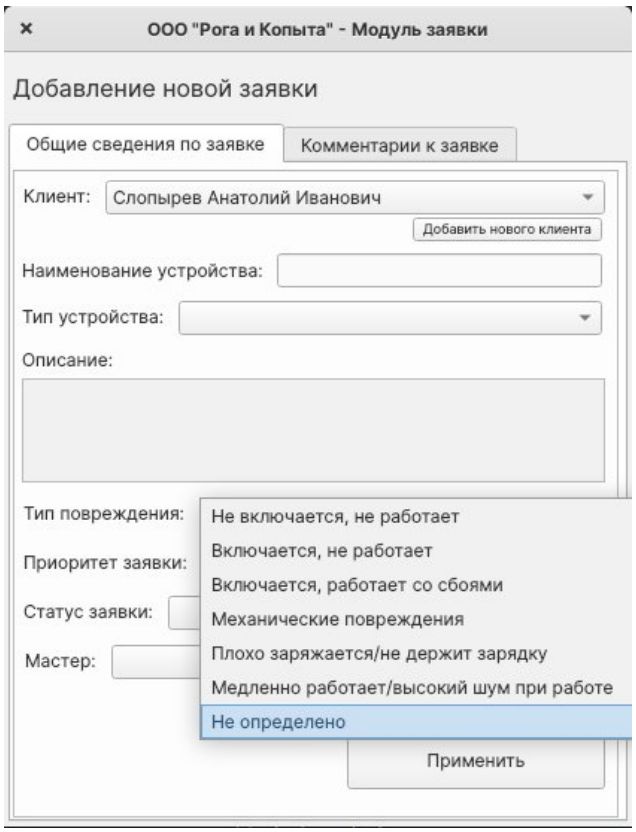

Рисунок 45 – Просмотр содержимого выпадающего списка «Тип повреждения»

Попробуем добавить нового клиента, для этого нажмем на кнопку «Добавить нового клиента». Модуль добавления нового клиента проиллюстрирован на рисунке 46.

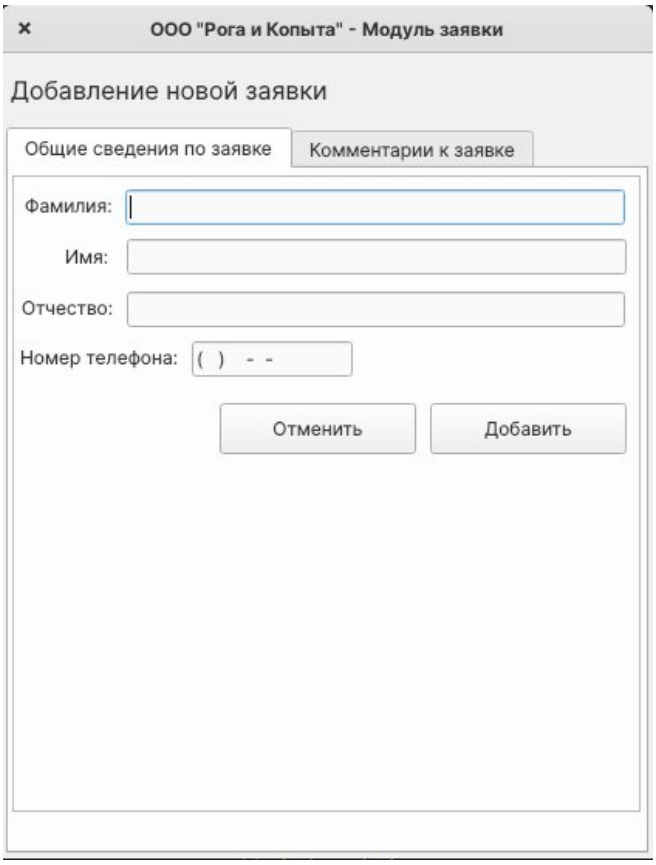

Рисунок 46 – Модуль добавления нового клиента

При попытке добавить нового клиента с какими-то недостоющими данными, программа выведет пользователю сообщение об ошибке, представленное на рисунке 47.

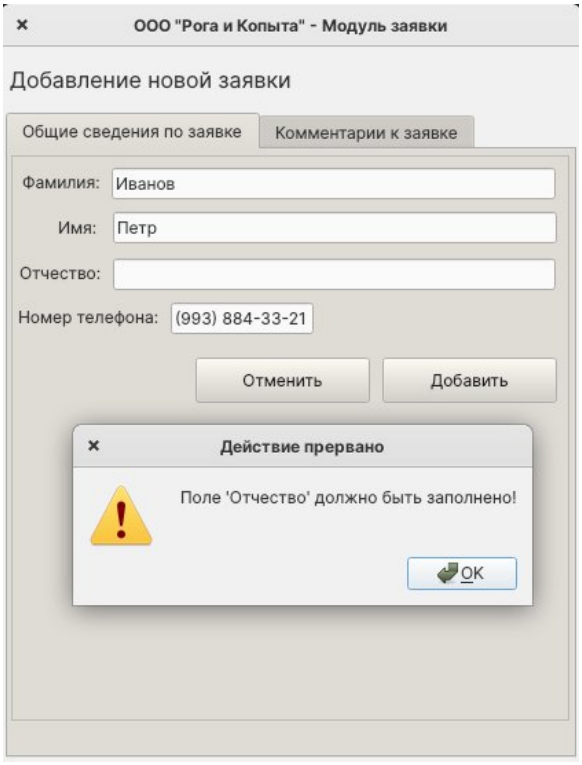

Рисунок 46 – Сообщение об ошибке при добавлении нового клиента

Если же пользователь ввел данные корректно, то программа сообщит пользователю об этом, а в выпадающем списке клиентов появится новое ФИО, что продемонстрировано на рисунках 47, 48.

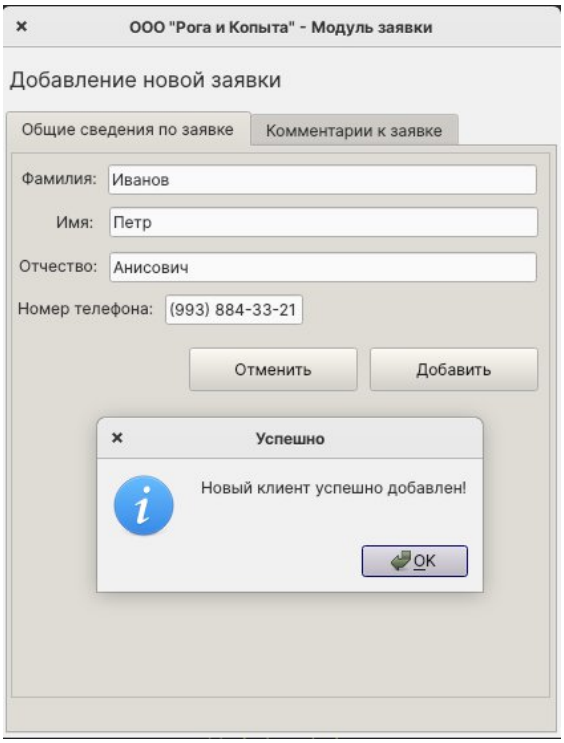

Рисунок 47 – Сообщение об успехе при добавлении нового клиента

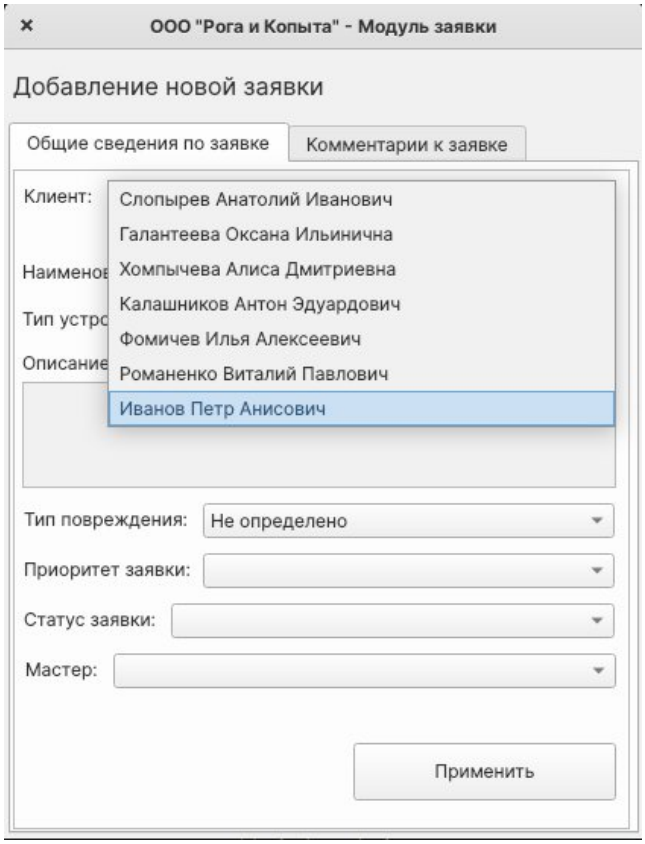

Рисунок 47 – Выпадающий список с клиентами после добавлении нового клиента

Попробуем применить изменения по заявке прямо сейчас, без заполенения всех данных. Программа обнаружит некорректный ввод данных, и сообщит о том, что мы катострафически не правы с своем намерении, это показано на рисунке 48.

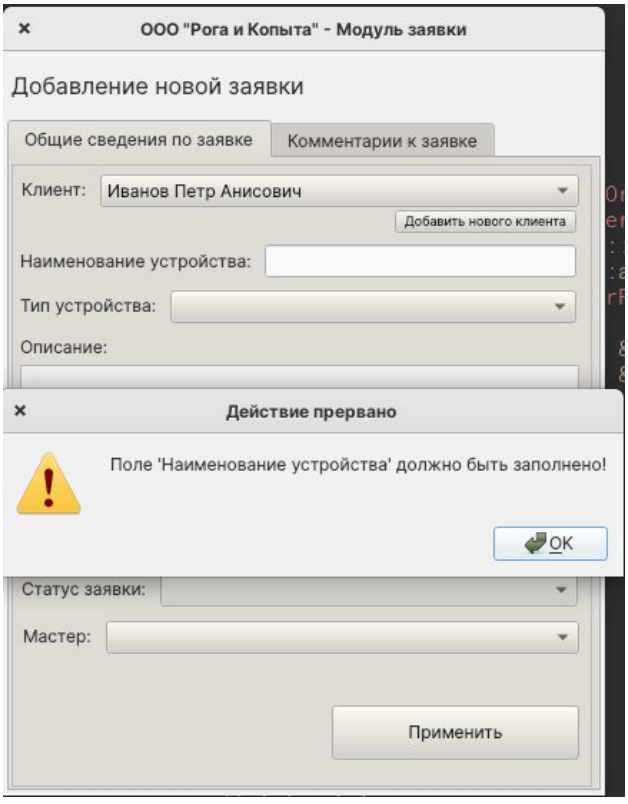

Рисунок 48 – Сообщение об ошибке при добавлении новой заявки

Действительно, не получается. Что же, давайте заполним поля ввода тестовыми значениями и посмотрим, как будет вести себя программа. Кстати, в текущий момент можно посмотреть на вкладку «Комментарии по заявке», и обнаружить, что элементы взаимодействия недоступны, а данные в поле вывода комментариев программа вводить не дает, все, как мы и задумывали.

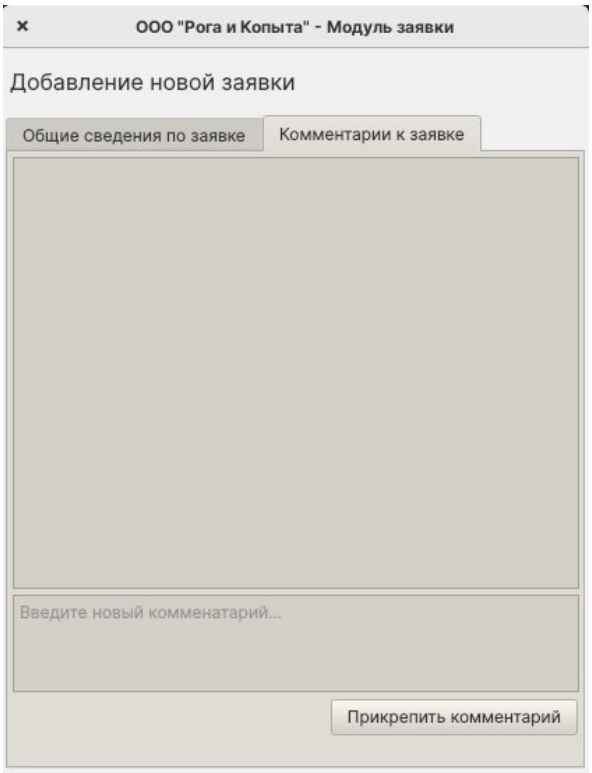

Рисунок 49 – Заблокированная вкладка «Комментарии по заявке»

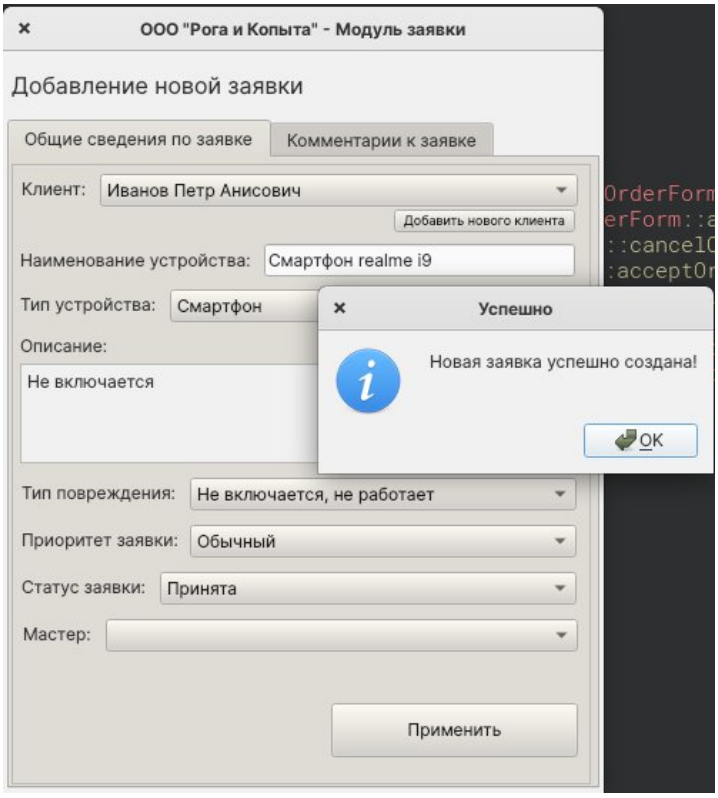

Рисунок 50 – Добавленная запись о новой заявке в систему

Теперь запустим форму в режиме «Изменение». Пока что мы не рассмотрели, каким образом это можно сделать, но мы это так надолго не оставим, поверьте, пока что просто смотрим, как оно будет работать, когда программный код основного модуля будет полностью написан и отлажен, эта форма изображена на рисунке 51.

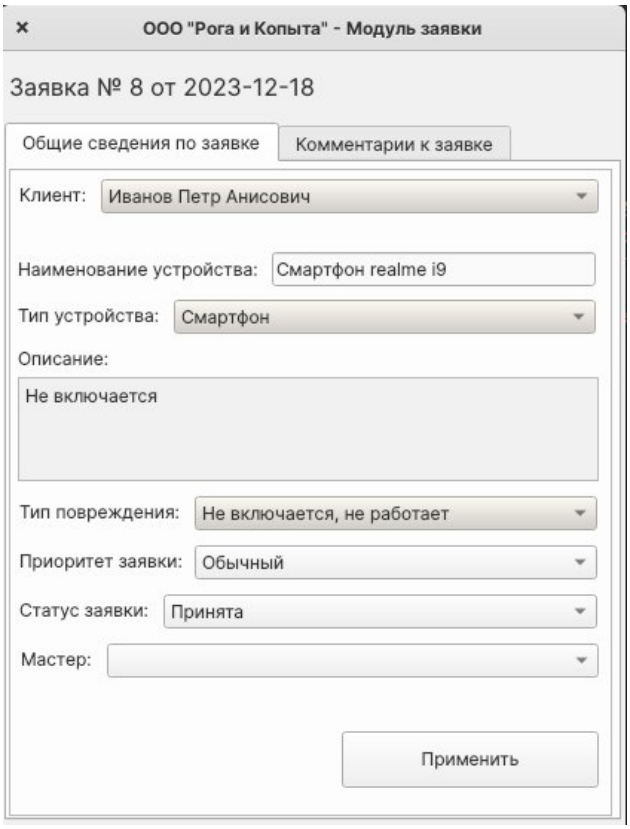

Рисунок 51 – Просмотр текущей заявки в модуле заявки

Как можно видеть из рисунка, у нас поменялась основная надпись на форме, что очень хорошо, также оказались заблокированными поля «Клиент», «Тип устройства» и «Тип повреждения». Для поля «Наименование устройства» включен параметр «только чтение». Остальные поля доступны для изменений. Попробуем сменить статус заявки на «Выполнена», но откажемся от действия, чтобы просто посмотреть на сообщение от программы, представленное на рисунке 52.

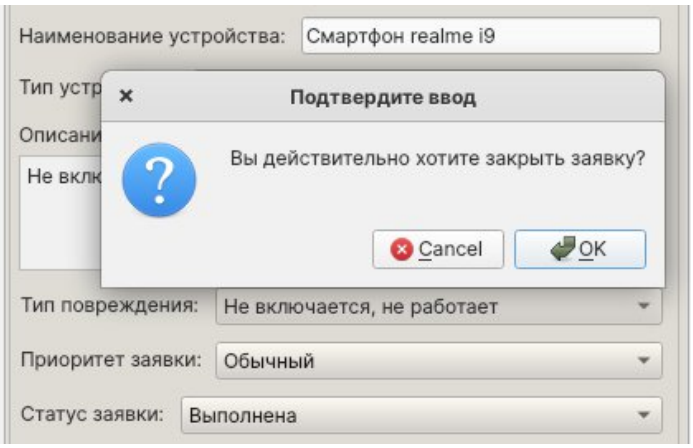

Рисунок 52 – Подтверждение изменения статуса текущей заявки на «Выполнена»

Как можно видеть, действительно сообщение имеется, с двумя выборами соответственно. Перейдем на вкладку «Комментарии к заявке», если ранее, при добавлении новой заявки, элементы были просто выключены, то теперь они попросту скрыты, потому что текущий пользователь, авторизированный в системе, не является исполнителем этой заявки, и не может оставлять к ней комментарии, только просматривать.

Рисунок 53 – Вкладка «Комментарии к заявке» текущей заявки

Попробуем изменить исполнителя для заявки, при этом статус заявки будет автоматически изменен на «На выполнении», что продемонстрировано на рисунке 54.

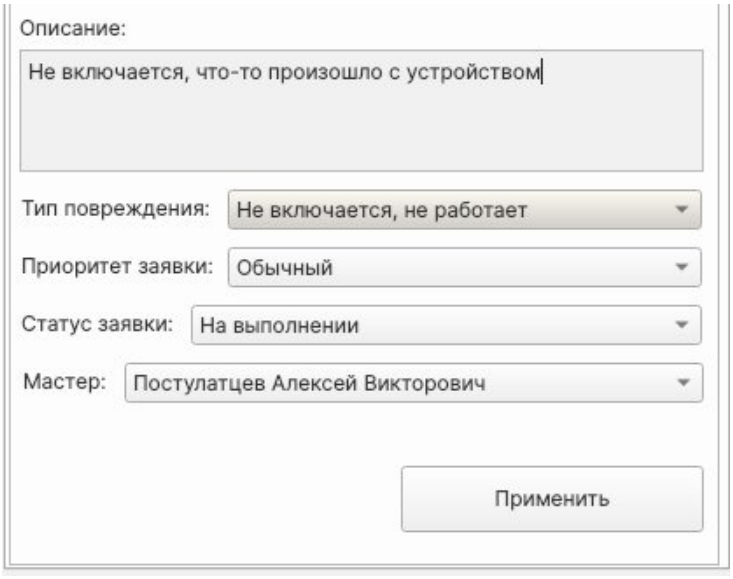

Рисунок 54 – Вкладка «Комментарии к заявке» текущей заявки

Применим изменения в заявке, сообщение об успешном изменении отображено на рисунке 55.

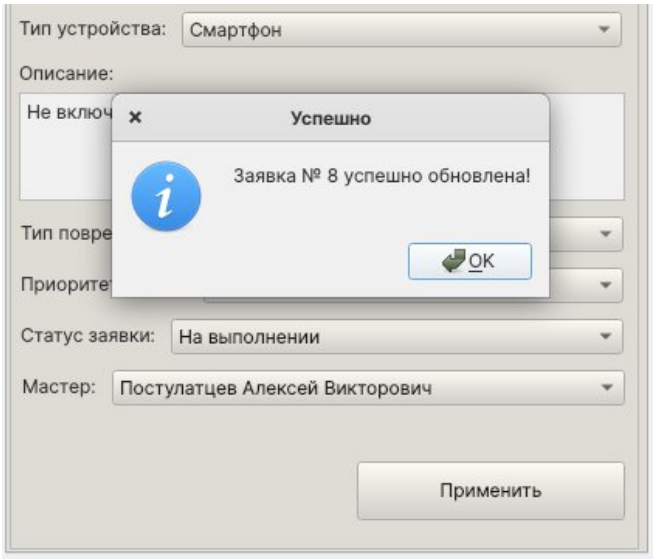

Рисунок 55 – Сообщение об успешном изменении текущей заявки

Если мы проверим заявку от лица исполнителя, то ему будет доступен функционал по добавлению комментариев, что отображено на рисунке 56.

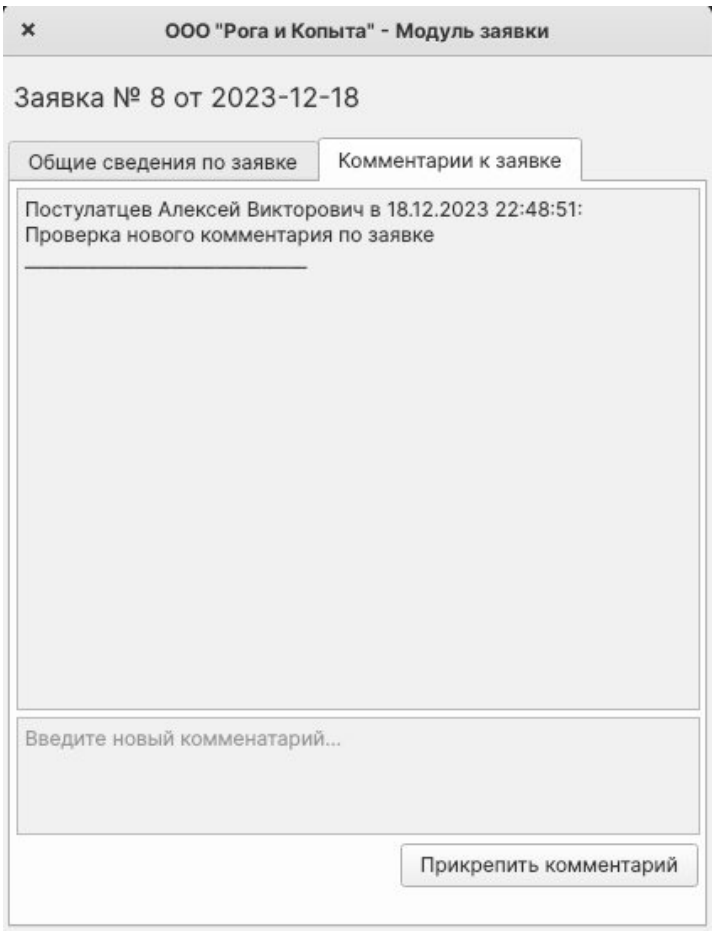

Рисунок 56 – Добавление комментариев к текущей заявке

Осталось проверить закрытие заявки. Вызовем сообщение об изменении статуса заявки и согласимся на него. После этого пользователю будет продемонстрировано сообщение об успешном закрытии заявки, и отображена дата закрытия, которую ранее не было видно, что продемонстрировано на рисунке 57.

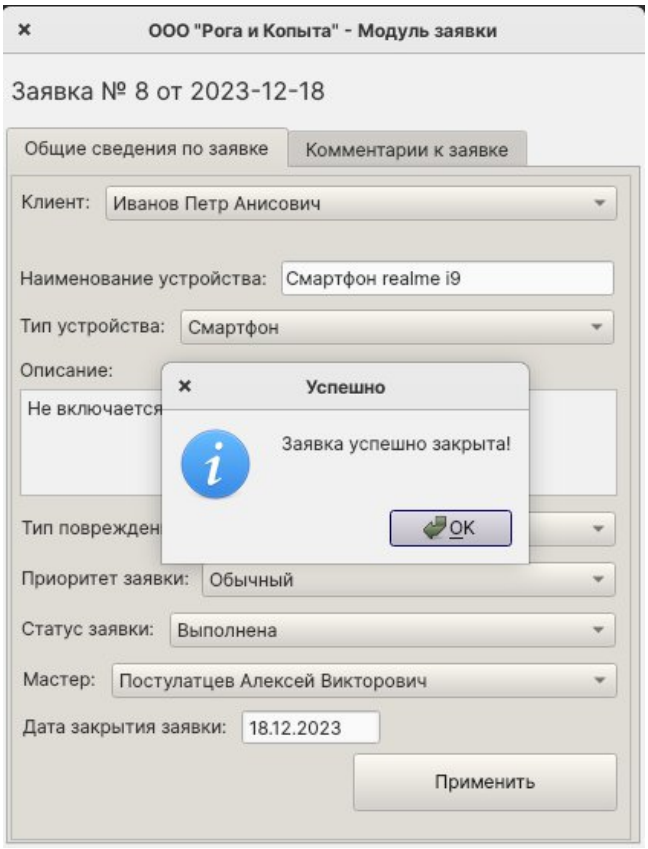

Рисунок 57 – Сообщение об успешном закрытии текущей заявки

Итак, модуль отлично работает, и полностью покрывает требования технического задания. Настало время доработать модуль основного окна, и программа для демоэкзамена будет успешно написана.

Макет формы у нас готов, осталось заняться программным кодом. Итак, подумаем над обработчиками событий. Элементов на форме здесь куда меньше, чем в предыдущем модуле, и сообразить, что должно выполняться и при выполнении каких действий – куда проще.

1. Ну сразу можно понять, что нам нужны обработчики на добавление новой заявки и на просмотр текущей заявки. Для вызова модуля заявки в режиме «Добавление» у нас предусмотрен пункт меню «Добавить новую заявку». Для вызова модуля заявки в режиме «Изменение» мы будем двойным кликом выбирать нужную заявку из таблицы QTableWidget.
2. Нам необходим поиск данных по параметрам. Не будем ничего здесь изобретать и совместим вывод данных по всем заявкам из БД с фильтрацией, что позволит объединить это в один метод.
3. Работа с уведомлениями. Суть их должна заключаться в том, что программа должна проверять состояние текущих заявок **постоянно**, и в случае, если статус заявки в программе у какой-либо заявки не совпадает с данными из БД, программа должна обновить эти данные, и оповестить пользователя об измененном статусе этой заявки.

Забегая вперед, сразу скажу, что в модуле основного окна проверка по статусам заявок будет висеть на глобальном таймере, который будет запускаться с начала открытия формы и работать постоянно, пока эта форма существует в памяти ОС. Подробнее об этом – позже, скажу лишь, что нам потребуется обработчик на «тик» таймера, т.е. на выполнение им подсчета определенного количества времени, после которого программа будет должна сделать какое-то действие, в нашем случае – обратиться к БД и проверить данные по заявкам.

4. Помимо этого, пользователь должен иметь возможность закрывать появившиеся на его экране уведомления. За их отображение будет отвечать элемент QListWidget, но сами уведомления будут представлять собой особый, «кастомный» виджет, который будет посылать сигналы о том, что пользователь хочет его закрыть. Программа должна будет перехватывать эти сигналы и убирать выбранное пользователем уведомление с экрана, а в для нас, программистов, это означает, что программа должна будет удалять элементы из виджета QListWidget, за это должен будет отвечать отдельный слот.

Список обработчиков событий не слишком велик. Теперь разберемся с рядовыми функциями. В модуль основного окна из формы авторизации мы должны подгружать сведения по авторизировавшемуся в систему пользователю: ФИО, роль и его ID, этот же ID будет позднее подгружаться в модуль заявки. Далее нам понадобиться метод для отрисовки на экран нового уведомления, а также метод, который будет добавлять данные непосредственно из БД в таблицу. Функционал, который будет реализован в этом методе, можно было бы совместить с общим методом поиска/вывода, однако, это не лучший вариант, поскольку в том методе будет много ветвлений, и дублирование одного и того же кода по разным веткам выглядит буквально как говнокод, мы не будем заниматься такими низменными вещами, о которых мы, тем более, что в курсе, и вынесем такой код в отдельную функцию.

Посмотрим, как будет выглядеть заголовочный файл модуля основного окна.

```
#ifndef OPERATORFORM_H
#define OPERATORFORM_H
#include <QtWidgets/QMainWindow>
#include <QSqlDatabase>
#include <QSqlQuery>
#include <QTimer>

namespace Ui { class OperatorForm; }

class OperatorForm : public QMainWindow
{
    Q_OBJECT

public:
    explicit OperatorForm(QWidget *parent = nullptr);
    ~OperatorForm();
    void setFormState(QString, QString, QString); //метод добавления данных из авторизации

public slots:
    void closeNotification(int); //слот закрытия выбранного уведомления
    void addNewOrder(); //слот открытия формы заявки в режиме «Добавление»
    void loadDataFromDB(); //слот чтения данных по заявкам из БД аля поиск/фильтрация
```

```
    void checkOrdersStatusInDB(); //слот проверки данных по статусу заявки от «тика» таймера
    void showFullInfoAboutOrder(); //слот открытия формы заявки в режиме «Изменение»

private:
    Ui::OperatorForm *ui;
    QTimer *timer;
    QString currentIDUser = "";
    QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL", "main");
    void createNewNotification(QString, QString); //метод создания нового уведомления
    void insertDataToTW(QSqlQuery); //метод добавления записи о заявке из БД в таблицу
};
#endif // OPERATORFORM_H
```

Наберем воздуха в грудь, выдохнем на все легкие и начнем верстать программную логику для нашего модуля. Начнем с вывода данных по заявкам, поскольку функционал, позволяющий эти заявки добавлять у нас уже предусмотрен в программе. Алгоритм этого слота, который называется loadDataFromDB(), представлен ниже:

1. Программа должна очистить предыдущий вывод данных в таблице. Замечу, что от экземпляра QTableWidget можно вызвать метод clear(), который успешно работает для выпадающих списков, очищая их содержимое, но для таблиц он работает несколько иначе. Да, метод clear() очистит строки, которые были добавлены в таблицу со всем содержимым этих строк, но более того, он еще очистит и содержимое столбцов, т.е. шапки таблицы, и она станет буквально *пустой*. Это, конечно, хорошо, но не совсем то, что нам надо, поэтому мы через цикл будем удалять каждую строку по очереди, пока их количество не станет 0.
2. Программа подключается к БД, и с случае успешного подключения выполняет следующие действия:
    a. Программа совершает выборку из представления со всеми данными, которые понадобятся для вывода в таблицу. Как верстать представления SQL вы должны были на дисциплинах МДК 07.01 и МДК 11.01, так что разбирать это дополнительно я не буду, кто не могет – добро пожаловать в гугл.
    b. Если длина строки, заключенной внутри поля ввода строки поиска, больше 0, то:
        I. Программа анализирует выпадающий список с параметрами поиска, и выбирает одно поле из строки виртуальной таблицы БД в соответствии с тем, по какому полю будет происходить поиск;
        II. Если это значение, выбранное ранее, приведенное к нижнему регистру, содержит как часть своей строки ту строку, которая находится внутри поля ввода поиска, также приведенной к нижнему регистру, то программа добавляет всю строку из виртуальной таблицы БД с записью о заявке в таблицу вывода данных. Приведение к нижнему регистру каждого из элементов сравнения критически важно, поскольку поиск должен быть нечуствительным к регистру, никто из пользователей не будет писать ФИО в строку поиска с прописных, все будут печатать строчными, и если поиск не будет нечуствительным к регистру, то пользователь не получит никаких результатов поиска, хотя должен;
        III. Иначе программа пропускает текущую строку в виртуальной таблицы БД, и приступает к анализу следующей;

c. Иначе, если строка поиска пустая, то проверять на соответствие поиску ничего не нужно, и программа просто добавит всю строку из виртуальной таблицы БД с записью о заявке в таблицу вывода данных;

3. Иначе программа выдаст сообщение об ошибке по отсутствию подключения к БД.

Исходный код этого обработчика события представлен ниже:

```
void OperatorForm::loadDataFromDB()
{
    while(ui->TW_OrdersList->rowCount() != 0)
    {
        ui->TW_OrdersList->removeRow(0);
    }

    db = QSqlDatabase::database("main");
    if(getSqlConnection(db))
    {
        QSqlQuery query(db);
        query.exec("select * from showShortOrderInfo order by OrderClient asc");
        while(query.next())
        {
            if(ui->LE_SearchString->text().length() > 0)
            {
                QString search = "";
                switch(ui->CMBB_SearchFilter->currentIndex())
                {
                case 0:
                    search = query.value(0).toString();
                    break;
                case 1:
                    search = query.value(5).toString();
                    break;
                case 2:
                    search = query.value(4).toString();
                    break;
                case 3:
                    search = query.value(2).toString();
                    break;
                }

                if(search.toLower().contains(ui->LE_SearchString->text().toLower()))
                {
                    insertDataToTW(query);
                }
                else
                {
                    continue;
                }
            }
            else
            {
                insertDataToTW(query);
```

```
            }
        }
        ui->TW_OrdersList->resizeRowsToContents();
    }
    else
    {
        showMessageBox("Отсутствует подключение к БД", "Ошибка модуля", QMessageBox::Ok, QMessageBox::Critical);
    }
}
```

Сразу же рассмотрим процесс добавления данных из БД в таблицу, который вынесен в отдельную функцию insertDataToTW(QSqlQuery). Как можно видеть, он принимает один аргумент QSqlQuery, который содержит всю виртуальную таблицу БД, полученную в результате выборки из представления, и в которой указатель на текущую строку является спозиционированным.

Так вот, добавление данных в таблицу QTableWidget. Для добавления новой строки в таблицу используется метод insertRow(int), который вызывается от экземпляра этого класса, ну то есть, от самого объекта, и в который обязательно нужно передать параметр порядка добавления новой строки. Ключевое слово здесь – порядок. То есть, тут указывается порядок новой строки, ее фактический индекс расположения внутри двумерной матрицы. Если новая строка будет являться первой, то ее индекс будет равен 0, поскольку индексация элементов двумерного массива (а таблица QTableWidget является представлением такового) начинается с нуля, и именно этот индекс нужно передать в метод insertRow(). Как получить этот индекс программно, ведь мы не можем знать, какой индекс должна занимать новая строка в таблице, и действовать наугад мы также не можем, какое же это программирование тогда. Однако, если мы вызовем значение свойства всех имеющихся строк в таблице через метод rowCount(), то мы как раз получим индекс для расположения новой строки в таблице. Подумайте сами, ведь если мы добавляем самую первую строку в таблицу, то возвращаемое значение функции rowCount() будет равняться 0, ведь строк-то еще никаких и нет, все правильно. Если будем добавлять вторую строку, то функция rowCount() вернет значение 1, и это верно, поскольку мы добавили одну строку, которая имеет индекс расположения 0, и сейчас нужно добавлять новую, с индексом 1.

Однако, просто добавить новую строку в таблицу недостаточно. При этом добавляется просто пустой скелет строки, в котором нет никаких данных, есть просто пустые ячейки, не более. Итак, начнем с того, что таблица QTableWidget состоит не просто из строк и столбцов, или ячеек. Она состоит из объектов QTableWidgetItem, а вот уже эти элементы предоставляют доступ к информации, которая может располагаться внутри ячейки. То, что я скажу – это не совсем верно, но объект QTableWidgetItem – и есть сама ячейка, так вам будет проще понять и запомнить структурное наполнение таблиц QTableWidget. Для добавления содержимого ячейки, ну то есть буквально самой ячейки, как структурное единицы, необходимо вызвать метод setItem() от экземпляра таблицы, в который нужно передать три обязательных параметра: индекс строки, индекс столбца и новый указатель на объект QTableWidgetItem, который будет находиться внутри ячейки, расположенной на пересечении указанных индексов. При этом будет создан не просто указатель, а динамический экземпляр класса QTableWidgetItem, в конструктор которого передается содержимое ячейки в виде строки, так-то. А для получения доступа к уже имеющейся ячейке, необходимо обратиться к методу item(), в который надо передать

индексы строки и столбца, по которым располагается требуемая ячейка. Нам это потребуется для настройки выравнивания текста внутри ячейки, потому что объективно говоря, ряд данных, типа наименования устройства, ФИО клиента или описания заявки может не помещаться одной строкой в ячейку, и тогда строки придется разбивать, но по-умолчанию таблицы QTableWidget так делать не умеют, это придется обрабатывать программно.

Обобщим все, что было сказано выше, в програмный код. Добавим новую строку в таблицу:

```
ui->TW_OrdersList->insertRow(ui->TW_OrdersList->rowCount());
```

Теперь в добавленную строку через вызов функции rowCount()-1 обратимся к 0 строке (только что добавленной), и 0 (т.е. первой для нас) колонке, чтобы добавить в нее содержимое ячейки QTableWidgetItem, которое мы будем брать из 1 колонки в строке виртуальной таблицы БД:

```
ui->TW_OrdersList->setItem(ui->TW_OrdersList->rowCount()-1, 0, new QTableWidgetItem
(query.value(0).toString()));
```

Отлично, теперь поговорим про выравнивание текста в таблицах. За перенос текста внутри ячейки по строкам отвечает значение одного из enum, т.е. из перечисляемого типа Qt «Qt::TextWordWrap», однако, оно не является типом именно *выравнивания* строк, для этого это значение из enum придется явно привести к перечисляемому типу Qt::Alignment, которое хранит значения всех возможных выравниваний текста, как по горизонтали, так и по вертикали. Для удобного представления данных о возможных выравниваниях перечисляемого типа Qt::Alignment я приведу их в единой таблице:

Значение	Описание
Qt::AlignLeft	Выравнивается по левому краю.
Qt::AlignRight	Выравнивается по правому краю.
Qt::AlignHCenter	Центрирует горизонтально в доступном пространстве.
Qt::AlignJustify	Выравнивает текст по доступному пространству.
Qt::AlignTop	Совпадает с верхом.
Qt::AlignBottom	Выравнивается по низу.
Qt::AlignVCenter	Центрирует вертикально в доступном пространстве.
Qt::AlignBaseline	Выравнивается по базовой линии.

Итак, установим выравнивание по центру относительно вертикали, слева отновистельно горизонтали и перенос по строкам для содержимого ячейки [0;0]:

```
ui->TW_OrdersList->item(ui->TW_OrdersList->rowCount()-1, 0)->setTextAlignment(Qt::AlignVCenter | Qt::AlignLeft
| Qt::Alignment(Qt::TextWordWrap));
```

А теперь можно продемонстрировать полный код функции insertDataToTW(QSqlQuery).

```
void OperatorForm::insertDataToTW(QSqlQuery query)
{
    ui->TW_OrdersList->insertRow(ui->TW_OrdersList->rowCount());
    ui->TW_OrdersList->setItem(ui->TW_OrdersList->rowCount()-1, 0, new
QTableWidgetItem(query.value(0).toString()));
```

```
    ui->TW_OrdersList->item(ui->TW_OrdersList->rowCount()-1, 0)->setTextAlignment(Qt::AlignVCenter |
Qt::AlignCenter);
    ui->TW_OrdersList->setItem(ui->TW_OrdersList->rowCount()-1, 1, new QTableWidgetItem
(query.value(1).toString()));
    ui->TW_OrdersList->setItem(ui->TW_OrdersList->rowCount()-1, 2, new QTableWidgetItem
(query.value(2).toString()));
    ui->TW_OrdersList->setItem(ui->TW_OrdersList->rowCount()-1, 3, new QTableWidgetItem
(query.value(3).toString()));
    ui->TW_OrdersList->item(ui->TW_OrdersList->rowCount()-1, 3)->setTextAlignment(Qt::AlignVCenter |
Qt::AlignLeft | Qt::Alignment(Qt::TextWordWrap));
    ui->TW_OrdersList->setItem(ui->TW_OrdersList->rowCount()-1, 4, new QTableWidgetItem
(query.value(4).toString()));
    ui->TW_OrdersList->item(ui->TW_OrdersList->rowCount()-1, 4)->setTextAlignment(Qt::AlignVCenter |
Qt::AlignLeft | Qt::Alignment(Qt::TextWordWrap));
    ui->TW_OrdersList->setItem(ui->TW_OrdersList->rowCount()-1, 5, new QTableWidgetItem
(query.value(5).toString()));
    ui->TW_OrdersList->item(ui->TW_OrdersList->rowCount()-1, 5)->setTextAlignment(Qt::AlignVCenter |
Qt::AlignLeft | Qt::Alignment(Qt::TextWordWrap));
    ui->TW_OrdersList->setItem(ui->TW_OrdersList->rowCount()-1, 6, new QTableWidgetItem
(query.value(7).toString()));
    ui->TW_OrdersList->item(ui->TW_OrdersList->rowCount()-1, 6)->setTextAlignment(Qt::AlignVCenter |
Qt::AlignCenter);
}
```

Теперь прикрутим обработчики событий на создание новой заявки и для редактирования текущей заявки. За это будут отвечать слоты addNewOrder() и showFullInfoAboutOrder() соответственно. Первый прост в исполнении: просто создаем экземпляр класса OperatorForm, подключив к .cpp файлу текущего модуля заголовочный файл класса OperatorForm, далее через публичные методы этого класса setCheckState() и setCurrentIDUser() передаем данные о режиме работы формы заявки и ID текущего пользователя в форму, после – вызываем метод updateForm(), который преднастроит работы формы заявки, а далее программе необходимо скрыть текущую форму, отобразить форму модуля заявки, заморозить выполнение кода внутри текущей формы через цикл, а после – отобразить текущую форму и перезагрузить основную таблицу, поскольку пользователь мог добавить новую заявку, и эту возможность необходимо обработать. Полный код метода addNewOrder() представлен ниже:

```
void OperatorForm::addNewOrder()
{
    OrderForm *newOrder = new OrderForm();
    newOrder->setCheckState(false);
    newOrder->setCurrentIDUser(currentIDUser);
    newOrder->updateForm();
    this->hide();
    newOrder->show();
    while(newOrder->isVisible())
    {
        QApplication::processEvents();
    }
    this->show();
    loadDataFromDB();
}
```

Метод showFullInfoAboutOrder() отличается, по сути, только одной строчкой: вызовом публичного метода setCurrentIDOrder(), который передает в форму ID текущей заявки, который расположен в первой колонке текущей выбранной строки внутри таблицы QTableWidget. Программный код этого метода представлен ниже:

```
void OperatorForm::showFullInfoAboutOrder()
{
    OrderForm *showOrder = new OrderForm();
    showOrder->setCheckState(true);
    showOrder->setCurrentIDOrder(ui->TW_OrdersList->item(ui->TW_OrdersList->currentRow(), 0)->text());
    showOrder->setCurrentIDUser(currentIDUser);
    showOrder->updateForm();
    this->hide();
    showOrder->show();
    while(showOrder->isVisible())
    {
        QApplication::processEvents();
    }
    this->show();
    loadDataFromDB();
}
```

Теперь поговорим про последний простой, и надеюсь, понятный метод setFormState(QString, QString, QString), который загружает данные об авторизировавшемся пользователе из формы авторизации в текущую форму. Параметрами для этого метода будут являться ID текущего пользователя, его ФИО и роль, при этом, если роль пользователя не равна «Оператор», то такой пользователь не будет иметь доступа к пункту меню «Добавить новую заявку», этот пункт будет програмнно заблокирован.

```
void OperatorForm::setFormState(QString id, QString name, QString role)
{
    currentIDUser = id;
    ui->LB_OperatorName->setText("Пользователь: " + name);
    ui->LB_OperatorRole->setText("Роль пользователя: " + role);

    if(role != "Оператор")
    {
        ui->Menu_AddNewOrder->setEnabled(false);
    }
}
```

Сейчас же пойдет крайне абстрактная информация, поскольку нам предстоит сделать последнее – это прикрутить каким-то чудо-образом уведомления к нашему модулю. Для этих нужд нам придется создать кастомный элемент управления, т.е. пользовательский виджет, экземпляры класса которого мы будем помещать в элемент QListWidget. Да-да, это та самая прозначная рамка, которую вы могли видеть на рисунке 38, и которая перекрывает собой часть таблицы вывода. Постараюсь объяснить этот момент предельно подробно и понятно, поскольку это, на самом деле, есть занимательная механика, умение реализаций которой вам может в дальнейшем понадобиться в жизни, ну, допустим, когда будете верстать свои дипломные проекты.

Итак, начнем.

Во-первых, нам необходимо создать новый класс формы Qt Designer, что продемонстрировано на рисунке 58.

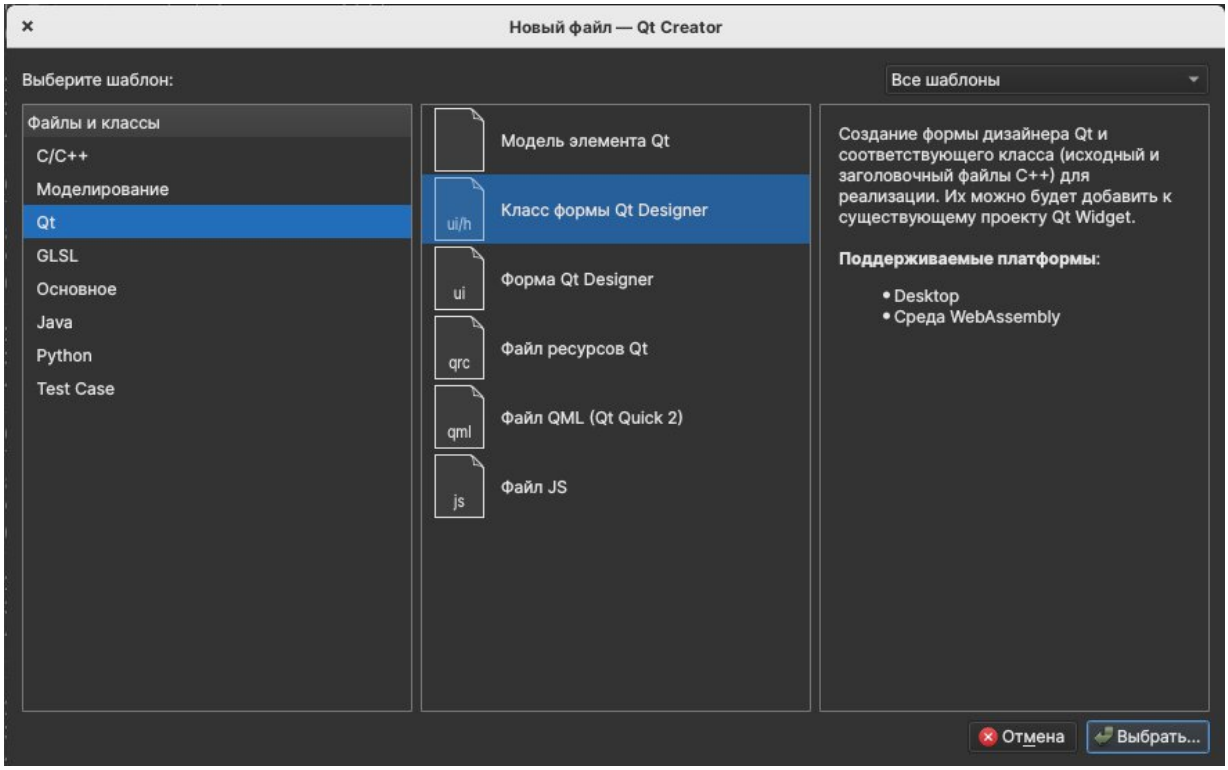

Рисунок 58 – Создание новой формы Qt Designer

Однако, в момент выбора шаблона формы, представленном на рисунке 59, вам необходимо вырать шаблон «Widget», а не «Main Window», поскольку мы будем верстать именно пользовательский виджет, а не новую форму, как до этого прежде.

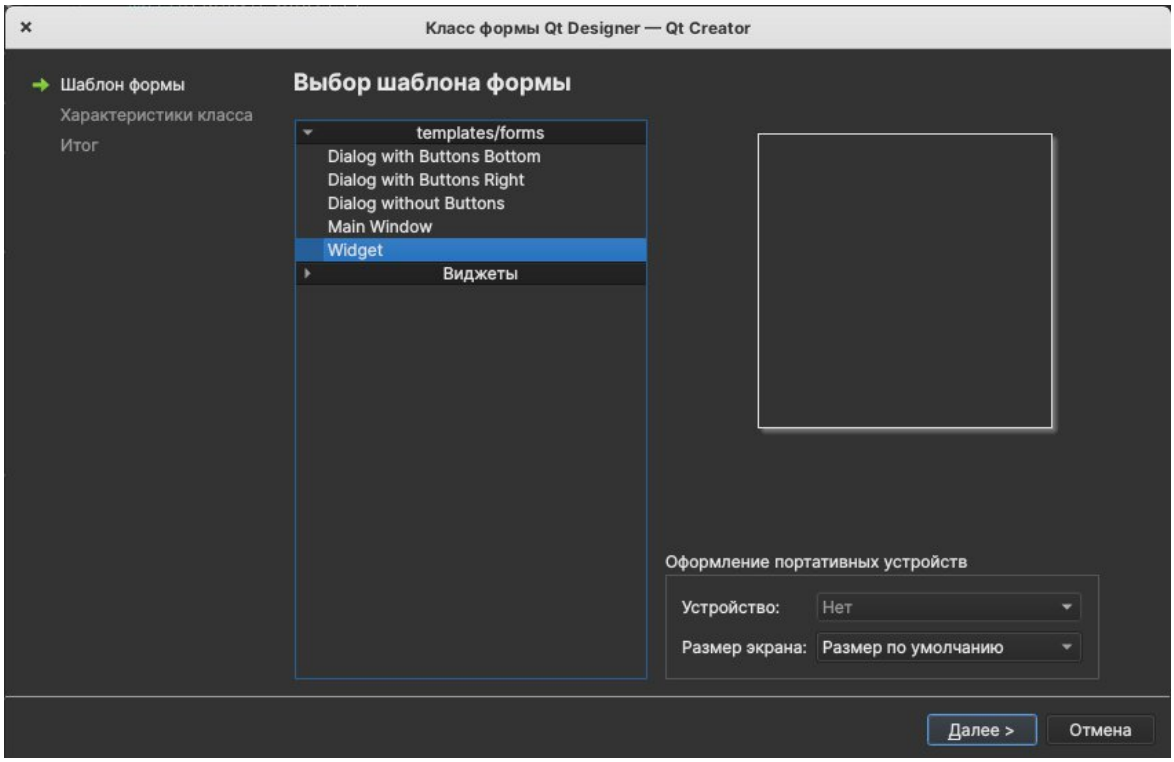

Рисунок 59 – Создание новой формы Qt Designer

Назовем этот класс этого виджета NotificationForm, и добавим новый класс в проект. Вам будет отображен пустой макет виджета, представленный на рисунке 60.

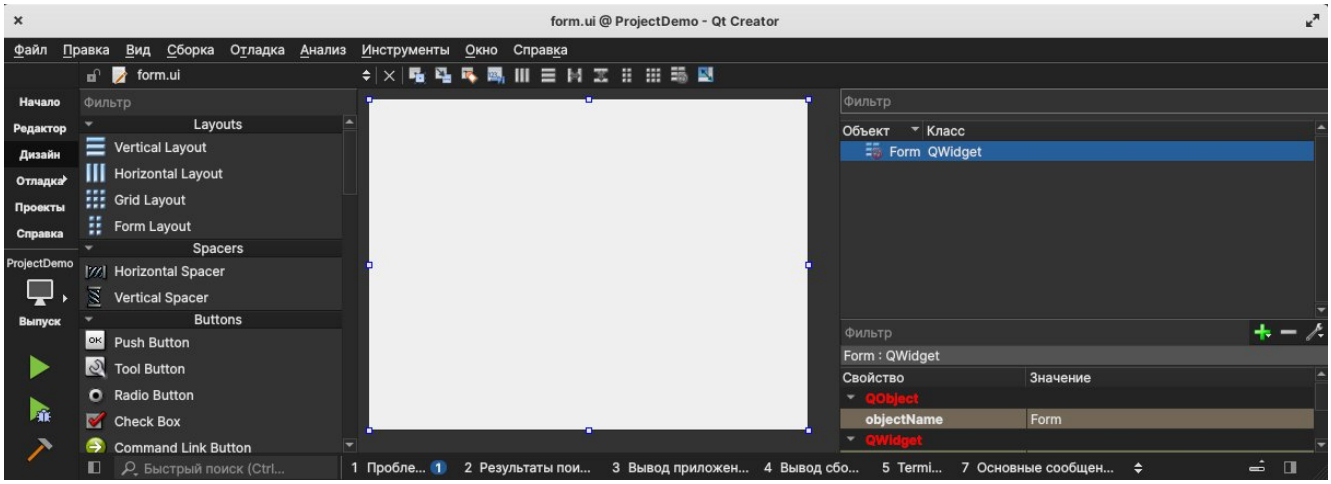

Рисунок 60 – Пустой макет виджета внутри Qt Creator

Смоделируйте свой виджет по примеру, представленному на рисунке 61. Сформируйте внутри объекта QFrame необходимые элементы управления, не забудьте выставить свойство frameShape для объекта QFrame как StyledPanel. А для объекта QTextEdit, наоборот, нужно убрать рамку вокруг него, выставив значение NoFrame для его свойства frameShape.

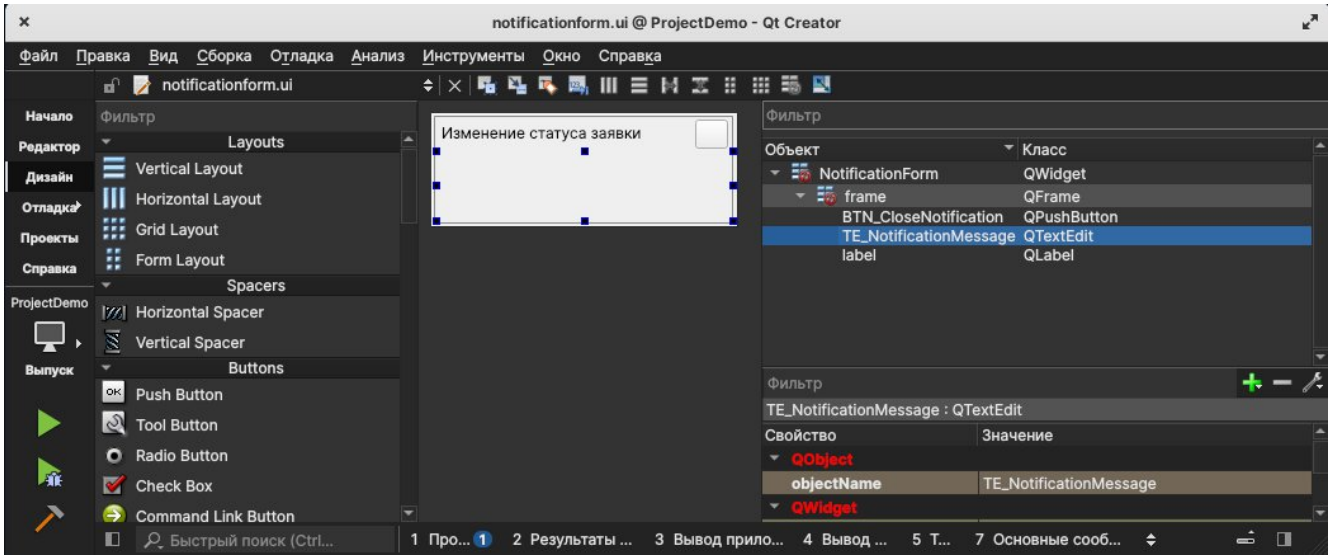

Рисунок 61 – Сформированный макет виджета внутри Qt Creator

Теперь поработаем над заголовочным файлом класса этого виджета. Для начала подумаем, что нам вообще нужно от этого элемента, как от структурной единицы программы. Этот виджет должен отображать информацию по изменению статуса заявки для пользователя, это раз. Пользователь должен иметь возможность закрыть это уведомление, для чего нами предусмотрен единственный интерактивный элемент на форме – это кнопка QPushButton, это два. В момент, когда пользователь будет нажимать на эту кнопку, программа должна перехватывать сигнал о том, что именно это уведомление должно быть закрыто, а не какое-то там еще, ведь уведомлений может быть несколько. Нельзя допустить ситуации, при которой попытка закрыть уведомление №1 приведет к тому, что закроются разом одновременно все 5 уведомлений, условно, пусть их будет 5.

Это несерьезный подход к программированию, получается, что программа должна отличать один экземпляр класса виджета уведомления от другого, это три, а уже, на самом деле, и четыре. Давайте посмотрим, что мы можем, как программисты, сделать для решения поставленных перед нами проблем.

Во-первых, если мы можем внутри класса определять слоты, т.е. обработчики событий, то нам чисто технически никто и ничто не может запретить определять новые сигналы, *пользовательские сигналы*. Мы можем это сделать, объявив нужные там прототипы функций под модификатором «signals:». Получается, что и сигналы в Qt – это не более, чем просто методы класса, и да, это правда, так и есть. И мы создадим свой сигнал, который будет говорить всей программе о том, что текущий экземпляр уведомления пользователь хочет закрыть.

Во-вторых, экземпляр класса виджета уведомлений должен предельно четко понимать когда этот сигнал нужно посылать. А когда? А при нажатии пользователем на кнопку QPushButton. Иными словами, класс должен обрабатывать сигнал от кнопки о том, что ее нажали, и посылать другой сигнал, о том, что уведомление нужно закрыть, поскольку событие, которому этому сигналу предшествует, было совершено пользователем. Получается, что нам нужен обработчик события на нажатие кнопки.

Далее нам нужен уникальный идентификатор уведомления. Каждый экземпляр класса виджета уведомлений будет хранить в себе целое значение, на достаточно большой диапазоне, чтобы однозначно отличаться по этому значению от других, прочих, экземпляров уведомлений. Сигнал, который будет посылать форма, будет нести с собой и уникальный идентификатор, который, как и сам сигнал, программа может (и будет) перехватить, и определить, какой из имеющихся в текущий на форме уведомлений послало его, чтобы найти среди всех прочих, и закрыть.

Проблема заключается в том, что «а как найти среди всех прочих?». Для этого мы будем перебирать все имеющиеся экземпляры уведомлений внутри элемента QListWidget, который будет просто контейнером для их хранения и отображения на экране, и сравнивать значения уникальных идентификаторов объектов внутри этого контейнера. Если программа обнаружит такой элемент, значение уникального идентификатора которого равно тому, что было перехвачено вместе с сигналом, то программа поймет, что он и является отправителем, и будет точно знать его расположение внутри контейнера, чтобы удалить его оттуда, и при этом – только его.

Ну и нужен метод, который будет загружать само сообщения внутрь элемента QTextEdit, как контейнера текста, и сам уникальный идентификатор, который будет генерироваться вне этого класса, будет приходить извне.

Вот и все. Исходный код заголовочного файла класса виджета уведомлений представлен ниже:

```
#ifndef NOTIFICATIONFORM_H
#define NOTIFICATIONFORM_H
#include <QtWidgets/QWidget>

namespace Ui { class NotificationForm; }

class NotificationForm : public QWidget
{
    Q_OBJECT
public:
    explicit NotificationForm(QWidget *parent = nullptr);
```

```
    ~NotificationForm();
    void loadMessage(QString, int); //метод загрузки информации об уведомлении
    int getIndex() { return notificationIndex; } //аксессор для получения ID уведомления
signals:
    void closed(int index); //сигнал закрытия текущего уведомления
public slots:
    void closeNotification(); //обработчик на нажатие кнопки закрытия уведомления
private:
    Ui::NotificationForm *ui;
    int notificationIndex = -1; //уникальный ID уведомления
};
#endif // NOTIFICATIONFORM_H
```

Сигналы не требуют собственной реализации внутри файла исходного кода класса, это некоторое исключение из правила «прототип неразрывен от его реализации», поскольку у нас не совсем-то и прототип функции, у нас сигнал, хоть и выглядят идентично. Аргументы сигнала должны являться полями класса и именно они (ну точнее, их значения) будут отсылаться вместе со своим сигналом. Это важное правило работы с сигналами Qt, которое не должно нарушаться.

Ну а тут дальше ничего сложного, на самом-то деле. Создаем реализацию метода loadMessage(QString, int), который будет направлять строку уведомления в контейнер текста QTextEdit, а пришедшее целое значение сохранять в переменную ID уведомления:

```
void NotificationForm::loadMessage(QString message, int index)
{
    ui->TE_NotificationMessage->setText(message);
    notificationIndex = index;
}
```

Затем создаем обработчик события с единственным оператором emit, который отсылает указанный после сигнал в программу, и только.

```
void NotificationForm::closeNotification()
{
    emit closed(notificationIndex);
}
```

А после – подключаем обработчик события closeNotification() к сигналу clicked() от элемента управления QPushButton внутри макета через конструктор класса NotificationForm:

```
NotificationForm::NotificationForm(QWidget *parent) :
    QWidget(parent),
    ui(new Ui::NotificationForm)
{
    ui->setupUi(this);
    connect(ui->BTN_CloseNotification, &QPushButton::clicked, this, &NotificationForm::closeNotification);
}
```

Вот и все, работа над созданием макета уведомления готова, возвращаемся к работе над модулем основного окна.

Здесь нам необходимо определить обработчик события на «тик» глобального таймера, его работа еще не настроена, но что должно произойти, когда таймер отсчитает нужное количество времени нам нужно определить уже сейчас, за это будет отвечать

слот checkOrdersStatusInDB(). Его суть предельно проста: подключаемся к БД, если это получается, то делаем выборку из всех данных по заявкам в БД из использованного нам ранее представления. Помимо этого, функции понадобиться флаг добавленного уведомления, который понадобиться чуть позже. Итак, далее перебираем все строки в нашей таблице на выводе дабы опрелить ту строку, которую можно сопоставить текущей перебираемой строке в виртуальной таблице БД. При обнаружении такой строки по ID заявкам программа сверяет данные по статусу заявки из таблицы QTableWidget и со значением из таблицы БД, и если они не совпадают, то программа создает новое уведомление, а если флаг создания такового ранее не был выдан, то выдает его. После перебора всех строк в виртуальной таблице БД программа опрашивает значение флага, и, если оно положительно, то перезагружает данные в таблице QTableWidget, ну, поскольку выяснилось что хотя бы одна строка содержит неактульаные данные.

Полный код этого метода представлен ниже:

```
void OperatorForm::checkOrdersStatusInDB()
{
    db = QSqlDatabase::database("main");
    if(getSqlConnection(db))
    {
        bool isStatusChanged = false;
        QSqlQuery query(db);
        query.exec("select * from showShortOrderInfo order by OrderClient asc");
        while(query.next())
        {
            for(int i = 0; i < ui->TW_OrdersList->rowCount(); i++)
            {
                if(query.value(0).toString() == ui->TW_OrdersList->item(i, 0)->text())
                {
                    if(query.value(7).toString() != ui->TW_OrdersList->item(i, 6)->text())
                    {
                        createNewNotification(query.value(0).toString(), query.value(7).toString());

                        if(!isStatusChanged)
                        {
                            isStatusChanged = true;
                        }
                    }
                }
            }
        }

        if(isStatusChanged)
        {
            loadDataFromDB();
        }
    }
}
```

А как же создать новое уведомление? Ну, это целая наука (хотя и несложная), за реализацию которой будет отвечать метод createNewNotification(QString, QString), который

будет принимать параметры ID заявки, статус которой был изменен, а также значение измененного статуса.

Для начала определим внутри этого метода новый динамический экземпляр класса NotificationForm. Для работы с этим классом в текущий .cpp файл нужно подключить соответствующий заголовочный файл #include "notificationform.h", конечно же. Динамический экземпляр класса позволит хранить в памяти программы указатель на этот экземпляр, на его область памяти, и пока сам экземпляр будет существовать, будет существовать и память на него, даже если у нас пропадет прямой доступ к этой области по указателю, мы этот момент обыграем несколько позже.

Итак, в созданный экземпляр класса уведомлений загружаем через публичный метод loadMessage два значения: сообщение, которое должно отображаться на форме уведомления, а также уникальный идентификатор, который мы будет генерировать с помощью генератора псевдослучайных чисел от текущего системного времени по следующей формуле: «максимальное значение – rand() % (максимальное значение – минимальное значение)», а после этого – подключим через устаревшую форму записи оператора connect текущий экземпляр уведомления с обработчиком события, который и будет фактически закрывать, т.е. удалять экземпляр из контейнера QListWidget.

Далее создаем новый экземпляр класса QListWidgetItem. ВНЕЗАПНО, как и таблица QTableWidget состоит структурно из совокупности элементов QTableWidgetList, расположенных по индексам строк и столбцов, так и элемент QListWidget также структурно состоит из элементов QListWidgetItem. От текущего, только что добавленного элемента QListWidgetItem, нужно вызвать аксессор setSizeHint(QSize), который установит размер элемента для отрисовки под размеры нашего макета уведомления. Размер элемента считается по длине диагонали прямоугольника, ограниченным размером M x N, где нас будет интересовать именно значение N, которое должно быть равно высоте макета на вывод, в моем случае, это 88 пикселей в высоту, и такой же размер я задам для подложки, внутри которой и будет располагаться мой макет, а это именно экземпляр класса QListWidgetItem.

Теперь мне ничего не стоит, чтобы добавить внутрь контейнера QListWidget новый элемент, внутри которого будет находиться подготовленный мной заранее элемент QListWidgetItem. Далее, я вызову от последнего добавленного элемента в QListWidget аксессор setBackground(), который переопределит цвет заднего фона элемента QListWidgetItem, поскольку я переопределил цвет заднего фона родительского контейнера QListWidget на прозрачный, и все элементы внутри него также унаследуют значение этого цвета, точнее, его альфа-канала, что мне совершенно не подходит, к чему мне прозрачное уведомление, текст внутри него будет сливаться с надписями на форме, это неудобно и просто дико бессмысленно.

После все манипуляций над объектом QListWidgetItem внутри последнего добаленного элемента QListWidget, я переопределю свойство itemWidget через его сеттер, setItemWidget(), что позволит мне встроить **ЛЮБОЙ** виджет внутрь объекта QListWidgetItem. Таким образом, внутри элемента QListWidget будет отображаться добавленный элемент класса уведомления, который позже может быть вызван, и более того, он совершенно работоспособен и подключен к нужному обработчику события.

Итак, теории много, и ее достаточно, если что-то непонятно, то перечитываем еще раз, или идем закрывать пробелы в знаниях в гугле, а мы, тем временем, переходим к практике. Исходный код метода createNewNotification(QString, QString) представлен ниже.

```
void OperatorForm::createNewNotification(QString order, QString status)
{
```

```
    NotificationForm* notice = new NotificationForm();
    notice->loadMessage("Статус заявки " + order + " изменился на статус: "' + status +'"", 9999-
rand()%(9999-0));
    connect(notice, SIGNAL(closed(int)), this, SLOT(closeNotification(int)));

    QListWidgetItem *item = new QListWidgetItem();
    item->setSizeHint(QSize(0, 88));

    ui->LW_NotificationsList->addItem(item);
    ui->LW_NotificationsList->item(ui->LW_NotificationsList->count()-1)->setBackground(QColor::fromRgb(202,
201, 201));
    ui->LW_NotificationsList->setItemWidget(item, notice);
}
```

Последний обработчик события, который осталось написать, это, собственно, слот для удаления экземпляра уведомления из контейнера QListWidget, за это будет отвечать слот closeNotification(int), который будет перехватывать вместе с сигналом ID уведомления, который этот сигнал и отослал, именно поэтому я использовал старый вариант записи оператора connect, поскольку он безболезненно позволяет это делать, без лишних танцев с бубном. Итак, суть обработчика предельно проста, и уже была истолкована как-то выше. Мы будем перебирать все элементы, хранящиеся внутри контейнера QListWidget и смотреть, какой из них отослал этот сигнал по пришедшему на вход ID. Однако, всплыл один интересный вопрос: мы только что в методе выше встроили внутрь элемента QListWidgetItem экземпляр класса NotificationForm. И по сути своей, мы будем перебирать не прямо их, а только подложки, на которых они размещены. Как получить экземпляры класса NotificationForm обратно из QListWidgetItem, чтобы получить доступ к публичному слоту getIndex() внутри этих экземпяров?

Ответ – с помощью динамического преобразования типов QObject, которое называется qobject_cast.

Тут следует объяснить важную суть всего и сразу в Qt. Все элементы управления, которые используются в Qt, наследуются от основного и базового для всех класса QObject, т.е. все сущее в Qt является прежде всего объектами фреймворка, а уже потом – все остальное. Именно по этой причине в обозревателе свойств объектов в дизайнере форм Qt Creator свойство objectName всегда располагается самым первым, поскольку унаследовано для объекта от самого примитивного, базового класса. Если один объект был унаследован от QObject, то его можно привести, т.е. буквально переделать в другой элемент, но который тоже должен быть унаследован он QObject, и более того, приводимый элемент должен иметь механики преобразования к производный, и наоборот. И это та ситуация, при которой мы можем привести тип QListWidgetItem к типу NotificationForm, потому что оба из них унаследованы от QObject, и являются объектами Qt, и более того, NotificationForm был насильно загружен в элемент QListWidgetItem, и поэтому может быть получен из него обратно. Разумеется, с помощью qobject_cast. Разумеется, бывают ситуации, когда один из элементов, или оба из них, не являются унаследоваными от QObject, и тогда приходится использовать static_cast для статичных объектов, или dynamic_cast для указателей, но это не наш случай, о спасибо.

Синтаксис qobject_cast весьма прост:

```
qobject_cast <преобразовываемый_тип_данных> (объект_который_имеет_этот_тип_но_не_в_
данный_момент);
```

Это, конечно, крайне вольный перевод, за тонкостями и подробностями обращайтесь в гугл, нам важно знать то, что нам нужно откуда то получить элемент, который представляет собой нужный нам тип данных, и мы буквально получим его прямой экземпляр с помощью преобразования объектов.

Для того, чтобы поместить экземпляр класса NotificationForm внутрь QListWidgetItem, я напрямую изменял свойство itemWidget через его сеттер. Так если я вызову это свойство напрямую, контейнер QListWidget вернет мне мой экземпляр класса уведомлений, который сейчас в нем хранится как QListWidgetItem, но я знаю его настоящий тип данных, это класс NotificationForm, и я легко смогу получить указатель на него обратно с помощью преобразования. В момент, когда это будет сделано, я смогу обратиться к нему как к простому указателю и вызвать публичный метод getIndex(), который вернет ID уведомления и сравнить его с тем, которое пришло вместе с сигналом, и если оно совпадает, то программа сохранит порядок расположения этого элемента в контейнере, прервет цикл перебора всех элементов в контейнере QListWidget, потому что желаемый уже обнаружен, а после – удалит указатель на объект QListWidgetItem внутри контейнера QListWidget, по его найденному порядку внутри него.

Сложна, сложна, сложна, ничего не понятно, в который раз. Понимаю, но если вам необходимо больше сведений, то я настоятельно рекомендую почитать вот эту статью на Хабре: https://habr.com/ru/articles/106294/, она закроет часть пробелов в ваших знаниях по преобразванию типов.

Полный код обработчика события closeNotification(int) представлен ниже:

```
void OperatorForm::closeNotification(int index)
{
    int removeIndex = -1;

    for(int i = 0; i < ui->LW_NotificationsList->count(); i++)
    {
        if(qobject_cast<NotificationForm*>(ui->LW_NotificationsList->itemWidget
(ui->LW_NotificationsList->item(i)))->getIndex() == index)
        {
            removeIndex = i;
            break;
        }
    }
    delete ui->LW_NotificationsList->item(removeIndex);
}
```

Последний незакрытый гештальт – это конструктор класса модуля основного окна. В нем мы выполним следующий набор действий:

1. Зафиксируем размеры окна, чтобы пользователь не мог их изменять вручную, поскольку у нас не предусмотрена масштабируемая верстка макета;
2. Определим генератор псевдослучайных чисел от текущего системного времени на момент вызова текущей формы;
3. Настроим работу таймера, выставив его интервал для срабатывания «тика», и запустим этот таймер с функцией автоматического перезапуска, чтобы он работал всегда, пока сама форма существует;

4. Переопределим горизонтальный заголовок таблицы QTableWidget, т.е. шапку таблицы, чтобы внутри нее также работал перенос слов по строкам, а также установим минимальную высоту для заголовка таблицы;

5. Настроим ширину для каждой из колонок таблицы QTableWidget, поскольку из редактора форм Qt Creator это сделать невозможно;

6. Соединим все элементы на форме со своими обработчиками событий;

7. И загрузим данные из БД в таблицу на вывод.

Начнем с таймера. Для установки интервала срабатывания таймера, при котором происходит его «тик», вызывается метод setInterval(), в котором задается парамерт времени в миллисекундах. 1000 миллисекунд - 1 секунда, если что. Для запуска таймера необходимо вызвать метод start() от его экземпляра.

Для изменения горизонтального заголовка таблицы, необходимо от текущей таблицы QTableWidget вызвать геттер horizontalHeader(), который и возвращает текущую шапку таблицы, а у этого объекта нас будут интересовать два сеттера: setDefaultAlignment(), в который передаются параметры выравнивания из перечисляемого типа Qt::Alighment, а также setMinimumHeight(), в который передается значение высоты заголовка в пикселях. Тоже ничего сложного.

Для редактирования ширины колонок таблицы, необходимо вызвать сеттер setColumnWidth() от экземпляра таблицы QTableWidget, в который передаются два параметра: порядок колонки (которые индексируются также с 0), и длину колонки в пикселях. В общем-то, на этом все. Полный код конструктора класса модуля основного окна представлен ниже:

```
OperatorForm::OperatorForm(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::OperatorForm)
{
    ui->setupUi(this);

    this->setFixedSize(QSize(this->width(), this->height()));

    srand(time(NULL));

    timer = new QTimer();
    timer->setInterval(5000);
    timer->start();

    ui->TW_OrdersList->horizontalHeader()->setDefaultAlignment(Qt::AlignCenter | Qt::Alignment
(Qt::TextWordWrap));
    ui->TW_OrdersList->horizontalHeader()->setMinimumHeight(40);

    ui->TW_OrdersList->setColumnWidth(0, 60);
    ui->TW_OrdersList->setColumnWidth(1, 80);
    ui->TW_OrdersList->setColumnWidth(2, 150);
    ui->TW_OrdersList->setColumnWidth(3, 230);
    ui->TW_OrdersList->setColumnWidth(4, 250);
    ui->TW_OrdersList->setColumnWidth(5, 230);
```

```
    loadDataFromDB();

    connect(ui->CMBB_SearchFilter, &QComboBox::currentIndexChanged, this,
&OperatorForm::loadDataFromDB);
    connect(ui->LE_SearchString, &QLineEdit::textChanged, this, &OperatorForm::loadDataFromDB);
    connect(ui->TW_OrdersList, &QTableWidget::cellDoubleClicked, this,
&OperatorForm::showFullInfoAboutOrder);
    connect(timer, &QTimer::timeout, this, &OperatorForm::checkOrdersStatusInDB);
    connect(ui->Menu_AddNewOrder, &QMenu::aboutToHide, this, &OperatorForm::addNewOrder);
}
```

Вуаля, хспаде, отмучались. Я вас поздравляю, демоэкзамен написан, и мы можем приступать к демонстрации получившегося результата.

При запуске программы нас встретит окно авторизации пользвателя, представленное на рисунке 62.

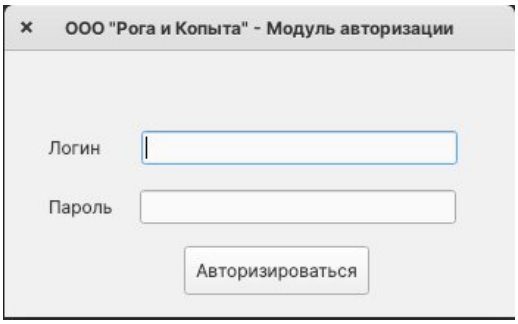

Рисунок 62 – Окно авторизации

Попытка авторизироваться с пустыми полями приведет к неутешающим результатам, представленным на рисунке 63. Авторизация под учетными данными, которые не в внесены в систему, также приведет к неудаче, что отражено на рисунке 64.

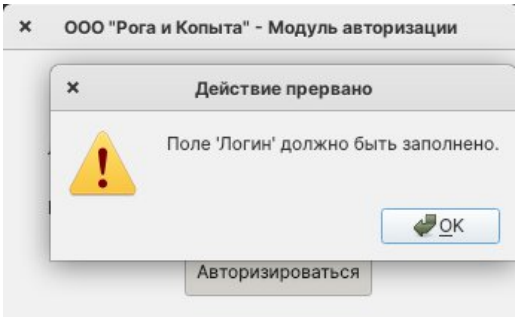

Рисунок 63 – Сообщение об ошибке при незаполненных полях ввода

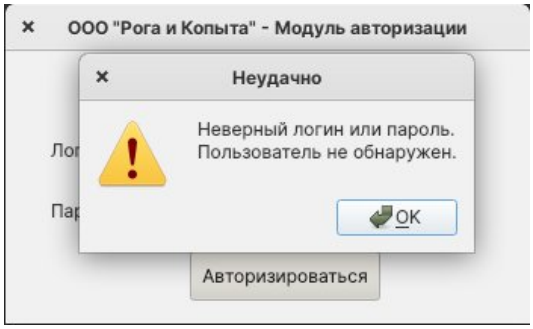

Рисунок 64 – Сообщение об ошибке при неудачной авторизации

Если пользователь не обладает повышенным кретинизмом, то он успешно авторизируется в системе, и его взору предстанет форма основного окна, представленное на рисунке 65.

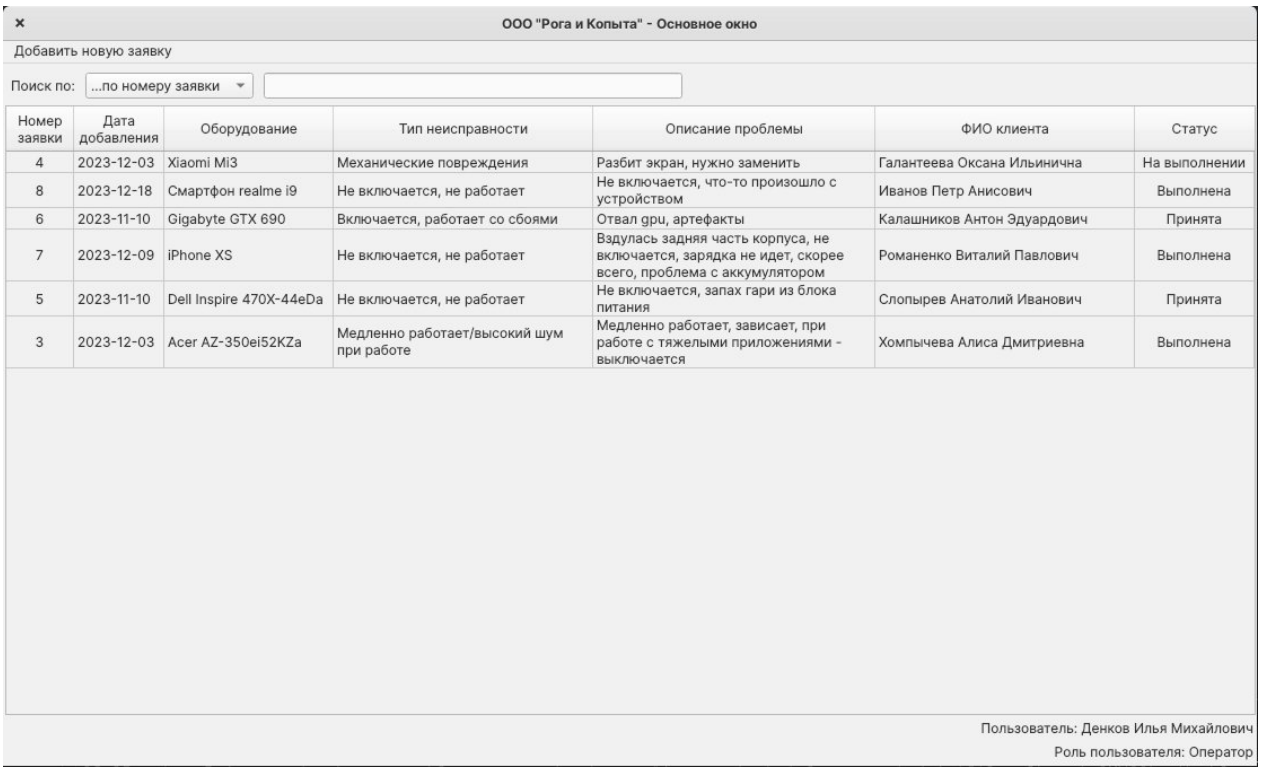

Рисунок 65 – Модуль основного окна программы

Проверим как работает поиск. Выберем из выпадающего списка параметров поиска значение «...по ФИО клиента» и забьем в строку поиска некоторое значение, что продемонстрировано на рисунке 66.

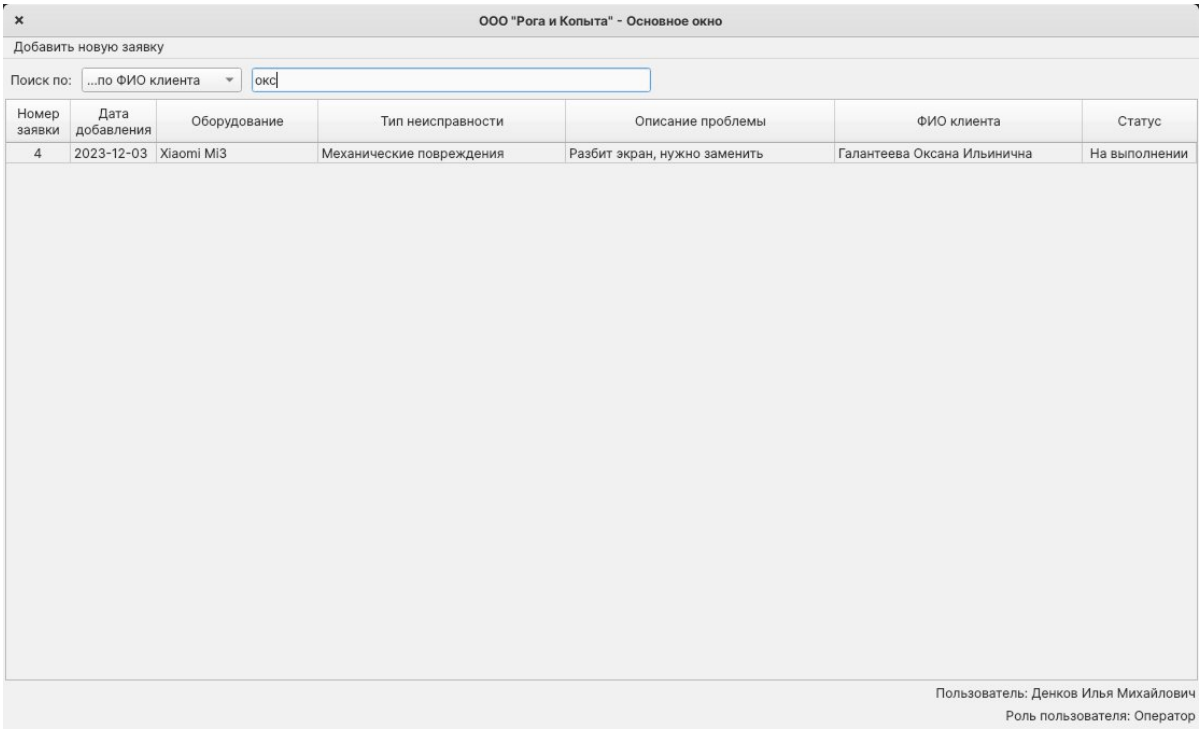

Рисунок 66 – Модуль основного окна программы с работающим поиском

Как можно видеть, работает как часы. Двойное нажатие на строку в таблице приведет к открытию формы заявки в режиме «Изменение», что проиллюстрировано на рисунке 67.

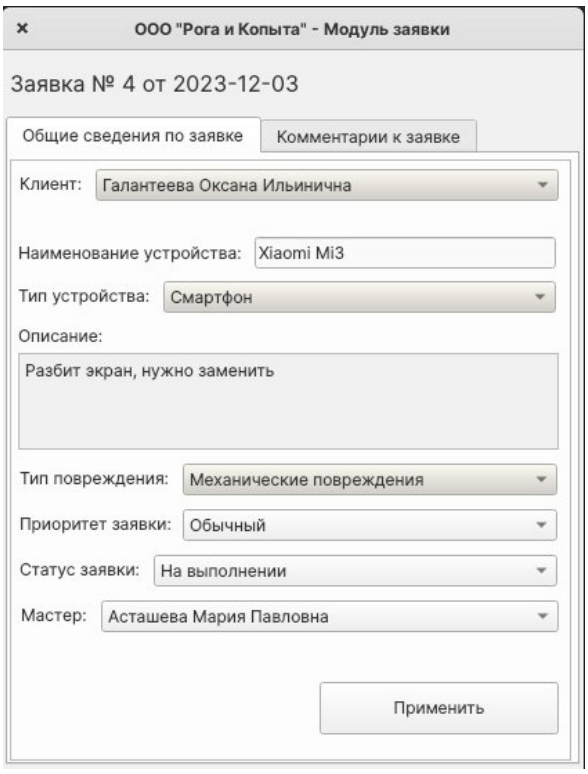

Рисунок 66 – Просмотр заявки № 4 из модуля заявки

Изменим статус для текущей заявки, и сохраним результат.

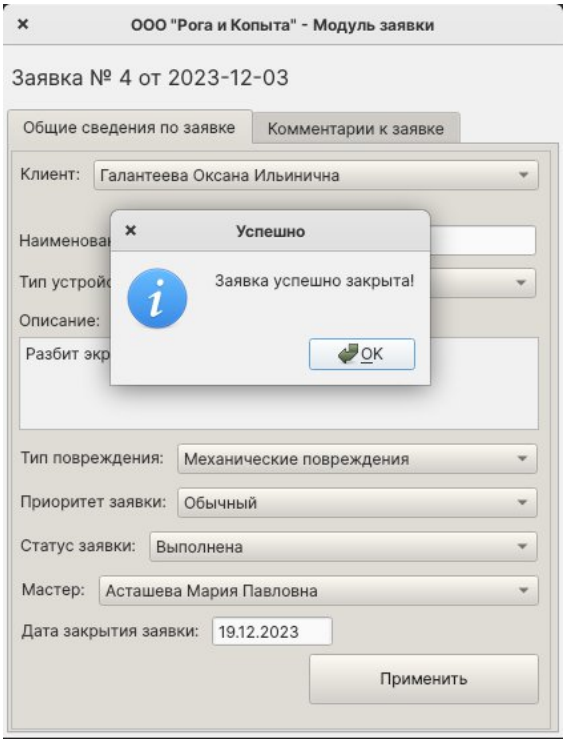

Рисунок 67 – Изменение статуса заявки № 4

Таймер внутри программы на проверку данных по статусам заявки установлен на 5000 миллисекунд, т.е. на 5 секунд. Выждем необходимое время и закроем текущее окно, чтобы вернуться в модуль основного окна, и, о чудеса, обнаружим уведомление об изменении статуса заявки для этой самой заявки, что продемонстрировано на рисунке 68.

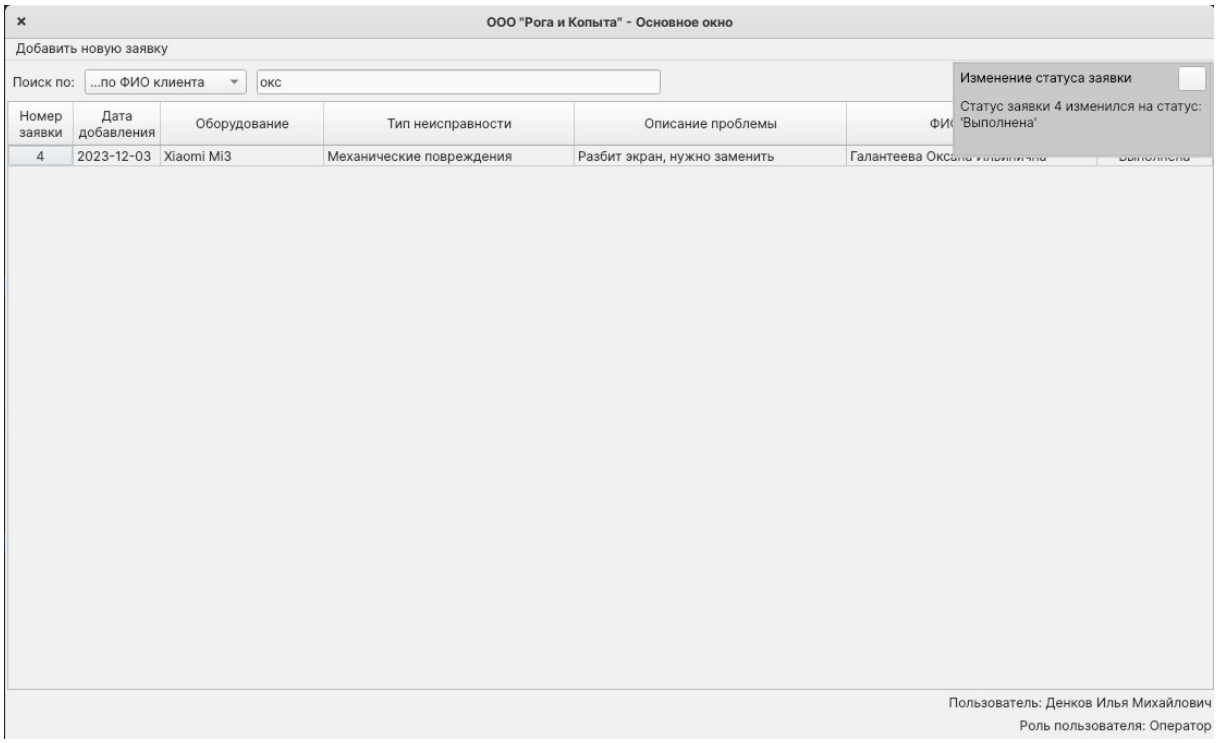

Рисунок 68 – Модуль основного окна с уведомлением об изменении статуса заявки

При нажатии на кнопку в правом верхнем углу уведомления оно будет закрыто. Если в систему авторизируется пользователь не с ролью «Оператор», то пункт меню «Добавить новую заявку» для него будет недоступен, что показано на рисунке 69.

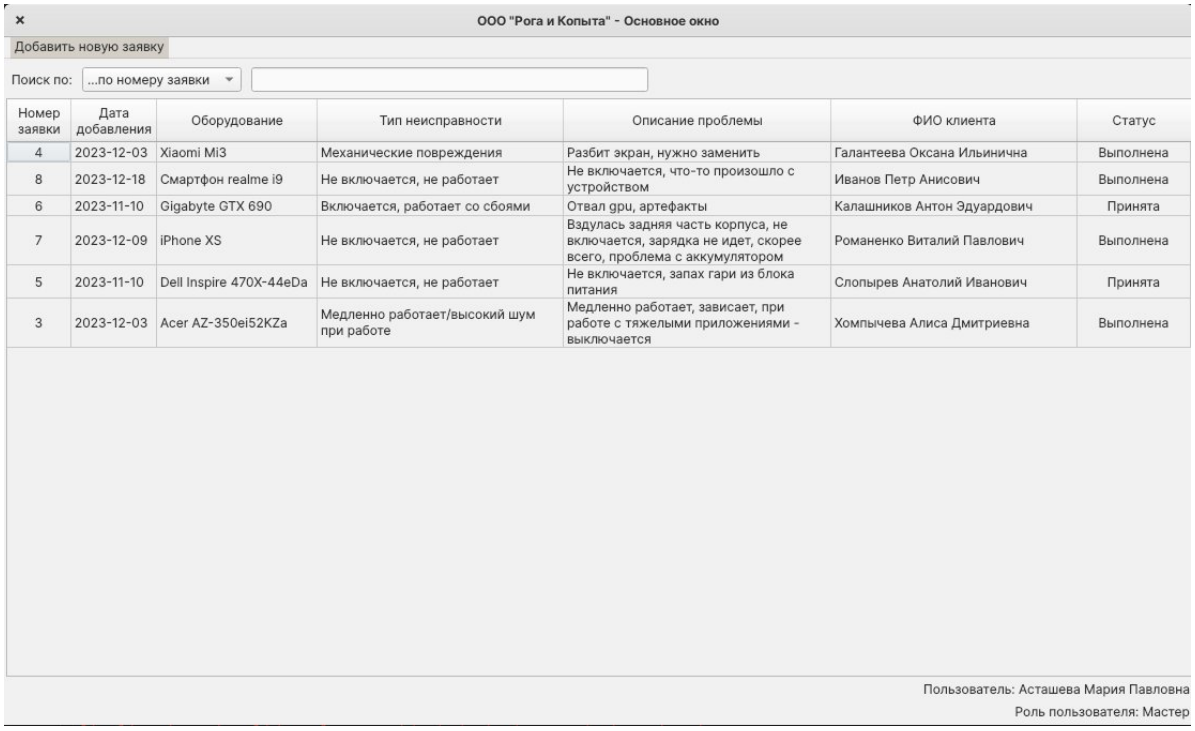

Рисунок 68 – Модуль основного окна пользователя с ролью «Мастер»

ГЛАВА IV. ЗАКЛЮЧЕНИЕ

Ну что же, вот мы и дошли до конца. Знаете, в момент, когда я писал это заключение, я одновременно сидел на онлайн-конференции колледжа, считайте, что на педсовете. Услышал мысль о том, что ничего сложного в демоэкзамене нет. Вот и думайте, ничего сложно нет. Всего-то 100 страниц вызубрить.

Ну да ладно, будем посмотреть. Я попытался впихнуть невпихуемое в 100 страниц, естественно, что охватить все-все тонкости, всю теоритическую базу я чисто технически бы не смог, это мне тогда было бы проще попросту перепечатать пару-тройку учебников по C++ и Qt 6. Если в процессе работы с этой методичкой у вас возникло осознание в недостатке знаний по программированию, по базам данных, то не закапывайте эти пробелы, не убирайте их в долгий ящик. Выписывайте себе, отмечайте те темы, с которыми возникают трудности, и прорабатывайте их отдельно. Поверьте, эти проблемы всплывут обязательно, в кратном объеме и в самых неподходящий момент.

Та программа, что была мною написана для этой методички, она в рамках демоэкзамена неосуществима, я так считаю. Ее нельзя будет написать за 4.5 часа, ну, я бы, честно, не смог, скорее всего, просто бы не успел. Моей задачей было показать, что это в принципе возможно, а ваша задача – смоделировать процесс выполнения демоэкзамена у себя дома и посмотреть, а какой объем работ именно вы сможете повторить, что вы точно сможете сделать, что придется вызубрить на механическом уровне, а что придется попросту слить из своей работы, потому что не справитесь, или же банально не хватит времени на реализацию.

За полгода подготовиться вполне реально, считайте, что вам нужно взять мировой рекорд по скоростному прохождению видеоигры, и для этого необходимо плотно закреплять участки маршрута, постоянно улучшать свое время, и проводить работу над ошибками, анализировать, где и на что вы потратили больше всего времени, как эти временные затраты можно оптимизировать, как стать лучше.

От себя я могу пожелать лишь удачи. Я сам был когда-то на вашем месте, причем дважды: как рядовой студент выпускного курса, так и как преподаватель. Ничего невыполнимого нет в демке, вопрос стоит лишь в масштабах работ, на какой кусок пирога вы замахнетесь, и сможете ли вы его прожевать.

За сим – буду прощаться, сам не люблю долгие проводы. Надеюсь, что вы осилите этот этап в своей жизни, и я со всеми вами встречусь на вашем же выпускном.

Увидиимся на защитах ваших дипломов.

Удачи.