МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ

ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных Технологий

Кафедра Информационных систем и технологий

Специальность 1-40 01 01 «Программное обеспечение информационных

технологий»

Специализация Программирование интернет-приложений

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**К КУРСОВОМУ ПРОЕКТУ НА ТЕМУ:**

«Централизованная система контроля версий Lfk»

Выполнил студент Магдич Алексей Павлович

(Ф.И.О.)

Руководитель проекта к.т.н., доц. Пацей Н.В.

(учен. степень, звание, должность, подпись, Ф.И.О.)

Заведующий кафедрой к.т.н., доц. Смелов В.В.

(учен. степень, звание, должность, подпись, Ф.И.О.)

Консультанты к.т.н., доц. Пацей Н.В.

(учен. степень, звание, должность, подпись, Ф.И.О.)

Нормоконтролер к.т.н., доц. Пацей Н.В.

(учен. степень, звание, должность, подпись, Ф.И.О.)

Курсовой проект защищен с оценкой

Минск 2017

# **Реферат**

Пояснительная записка курсового проекта содержит \_\_ страниц, \_\_ рисунков, \_\_ таблиц, \_\_ источников литературы.

Основной целью курсового проекта является: разработка централизованной системы контроля версий Lfk.

Пояснительная записка состоит из введения, четырёх глав, заключения.

Во введении представлена общая информация, дающая представление о предстоящей работе, определены цели.

В первой главе рассматриваются прототипы приложений и постановка задачи.

Во второй главе представлена архитектура приложения и алгоритмы, реализованные в нем.

В третьей главе представлено руководство пользователя.

В четвёртой главе представлено тестирование приложения.

В заключении представлены итоги курсового проектирования и задачи, которые были решены в ходе разработки программного средства.

# **Введение**

Всё больше и больше людей предпочитают хранить и управлять своими данными с помощью информационных систем. Это подтолкнуло разработчиков программного обеспечения к созданию таких продуктов, как системы контроля версий, позволяющих обезопасить процесс работы с пользовательскими данными.

Система контроля версиями позволяет хранить несколько версий одного и того же документа, при необходимости возвращаться к более ранним версиям, определять, кто и когда сделал то или иное изменение, и многое другое. Такие системы наиболее широко используются при разработке программного обеспечения для хранения исходных кодов разрабатываемой программы. Однако они могут с успехом применяться и в других областях, в которых ведётся работа с большим количеством непрерывно изменяющихся электронных документов.

Существует три основных вида систем контроля версий:

* локальные, работа которых основана на наборе патчей (патч — файл, описывающий различие между файлами), которые хранятся в специальном формате на диске;
* централизованные, в которых есть центральный сервер, на котором хранятся все файлы под версионным контролем, и ряд клиентов, которые получают копии файлов из него;
* распределённые, которые не просто выгружают последние версии файлов, а полностью копируют весь репозиторий. Поэтому в случае, когда возникает сбой на сервере, через который шла работа, любой клиентский репозиторий может быть скопирован обратно на сервер, чтобы восстановить базу данных.

Рассматриваемая система контроля версий Lfk является централизованной, отсюда следует функционал, реализуемый данной системой:

* Иметь возможность отслеживать изменения файлов в рабочем каталоге;
* Индексировать текущее состояние отслеживаемых файлов;
* Сохранять предварительно зафиксированное состояние;
* Отображать историю сохранённых состояний;
* Иметь возможность отката содержимого рабочего каталога в любое из сохранённых состояний;
* Поддержка регистрации и авторизации пользователей;
* Возможность сохранения текущего состояния репозитория на стороне сервера;
* Возможность выгружать и удалять содержащиеся на сервере репозитории на локальную машину пользователя.

Весь вышеописанный функционал обёрнут приятным графическим интерфейсом, поддерживающим интуитивно понятный диалог с пользователем.

# **Глава 1. Постановка задачи**

## **1.1. Обзор и анализ прототипов**

Не секрет, что все современные сложные программные продукты создаются коллективом программистов количеством до нескольких сотен на один проект. Многие серьёзные компании давно уже используют собственные системы управления с таким количеством разработчиков. Рассмотрим некоторые из основных таких систем контроля версий.

1. Subversion

Subversion — централизованная система (в отличие от распределённых систем, таких, как Git или Mercurial), то есть данные хранятся в едином хранилище. Хранилище может располагаться на локальном диске или на сетевом сервере. Работа в Subversion мало отличается от работы в других централизованных системах управления версиями. Клиенты копируют файлы из хранилища, создавая локальные рабочие копии, затем вносят изменения в рабочие копии и фиксируют эти изменения в хранилище. Несколько клиентов могут одновременно обращаться к хранилищу.

1. Mercurial

Mercurial — кроссплатформенная распределённая система управления версиями, разработанная для эффективной работы с очень большими репозиториями кода. Наряду с традиционными возможностями систем контроля версий, Mercurial поддерживает полностью децентрализованную работу (отсутствует понятие основного хранилища кода), ветвление (возможно вести несколько веток одного проекта и копировать изменения между ветками), слияние репозиториев (чем и достигается «распределённость» работы). Поддерживается обмен данными между репозиториями через HTTP/HTTPS, SSH и вручную при помощи упакованных наборов изменений.

1. Git

Главное отличие Git от любых других СКВ — это то, как Git смотрит на свои данные. В принципе, большинство других систем хранит информацию как список изменений (патчей) для файлов. Git не хранит свои данные в таком виде. Вместо этого Git считает хранимые данные набором слепков небольшой файловой системы. Git больше похож на небольшую файловую систему с невероятно мощными инструментами, работающими поверх неё, чем на просто СКВ. На данный момент является самой известной и мощной системой контроля версий.

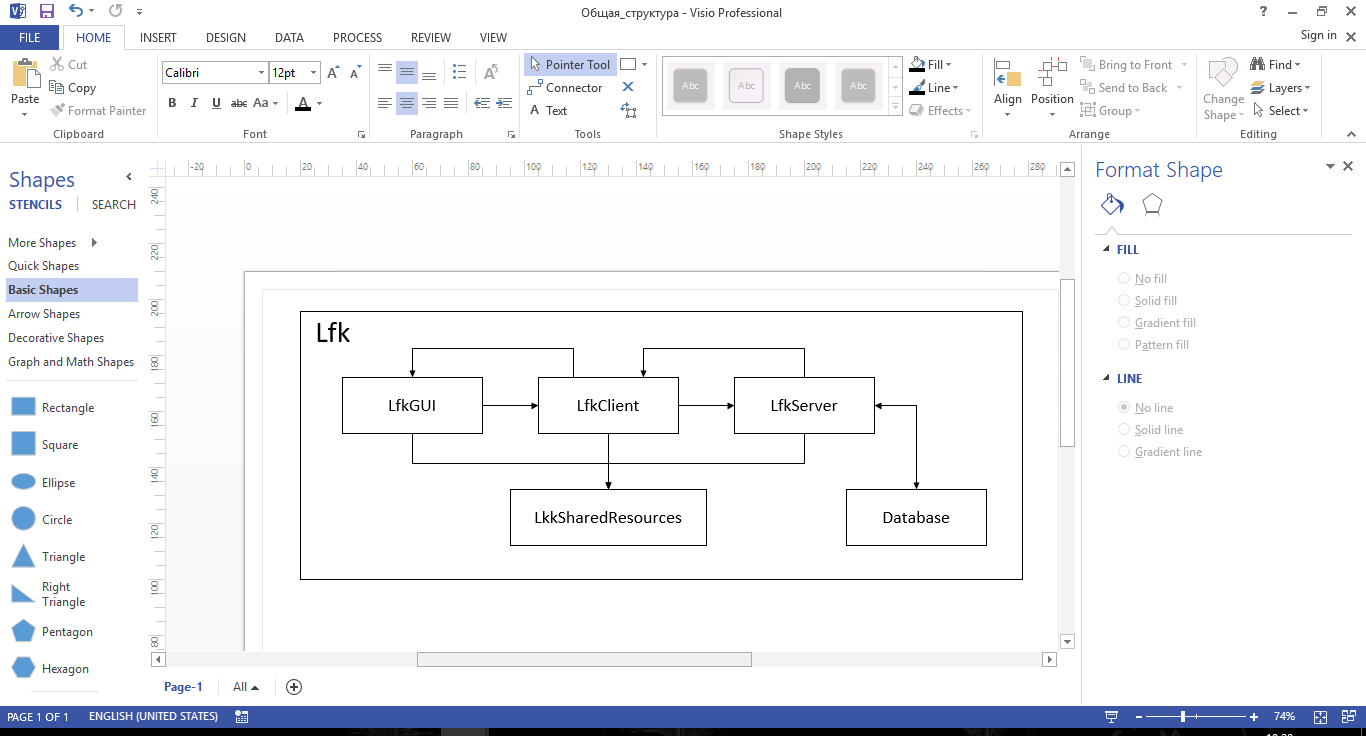
## **1.2. Актуальность решаемой задачи**

Ситуация, в которой электронный документ за время своего существования претерпевает ряд изменений, достаточно типична. При этом часто бывает важно иметь не только последнюю версию, но и несколько предыдущих. В простейшем случае можно просто хранить несколько вариантов документа, нумеруя их соответствующим образом. Такой способ неэффективен (приходится хранить несколько практически идентичных копий), требует повышенного внимания и дисциплины и часто ведёт к ошибкам, поэтому и было принято решение разработать свою собственную систему контроля версий Lfk.

# **Глава 2. Архитектура системы**

## **2.1 Общая структура**

Система контроля версий Lfk состоит из четырёх основных модулей: LfkSharedResources, LfkGUI, LfkClient и LfkServer. Схема их взаимодействия представлена на рисунке 2.1.1.



### Рисунок 2.1.1 – Схема взаимодействия модулей Lfk

Пользователь взаимодействует с программой посредством модуля LfkGUI, основная задача которого – поддержка диалога с пользователем в виде графического интерфейса и перенаправление всех запрашиваемых команд к модулю LfkClient. LfkClient – ядро системы, поскольку именно в нём реализована вся логика контроля за состоянием пользовательских данных. Помимо этого, LfkClient осуществляет связь с центральным сервером, представленным модулем LfkServer. Основная задача LfkServer – ожидание и обработка клиентов, а также формирование запросов к базе данных, которая и выступает единым хранилищем данных всех зарегистрированных пользователей. Каждый из модулей нуждается в некоторых классах, позволяющих разрешить некоторые общие вопросы и избавиться от рутинных алгоритмов. Все эти классы реализованы в модуле LfkSharedResources.

## **2.2 Модуль LfkSharedResources**

**Предназначение модуля**

Данный модуль представляет из себя контейнер классов, которые могут использоваться любым другим компонентом программы. Так называемые «общие ресурсы» призваны решить такие рутинные задачи, как сериализация, формирование и представление сетевых пакетов, сжатие пользовательских данных, а также обеспечить унифицированный способ взаимодействия модулей программы посредством моделей и иерархии исключений.

**Структура модуля LfkSharedResources**

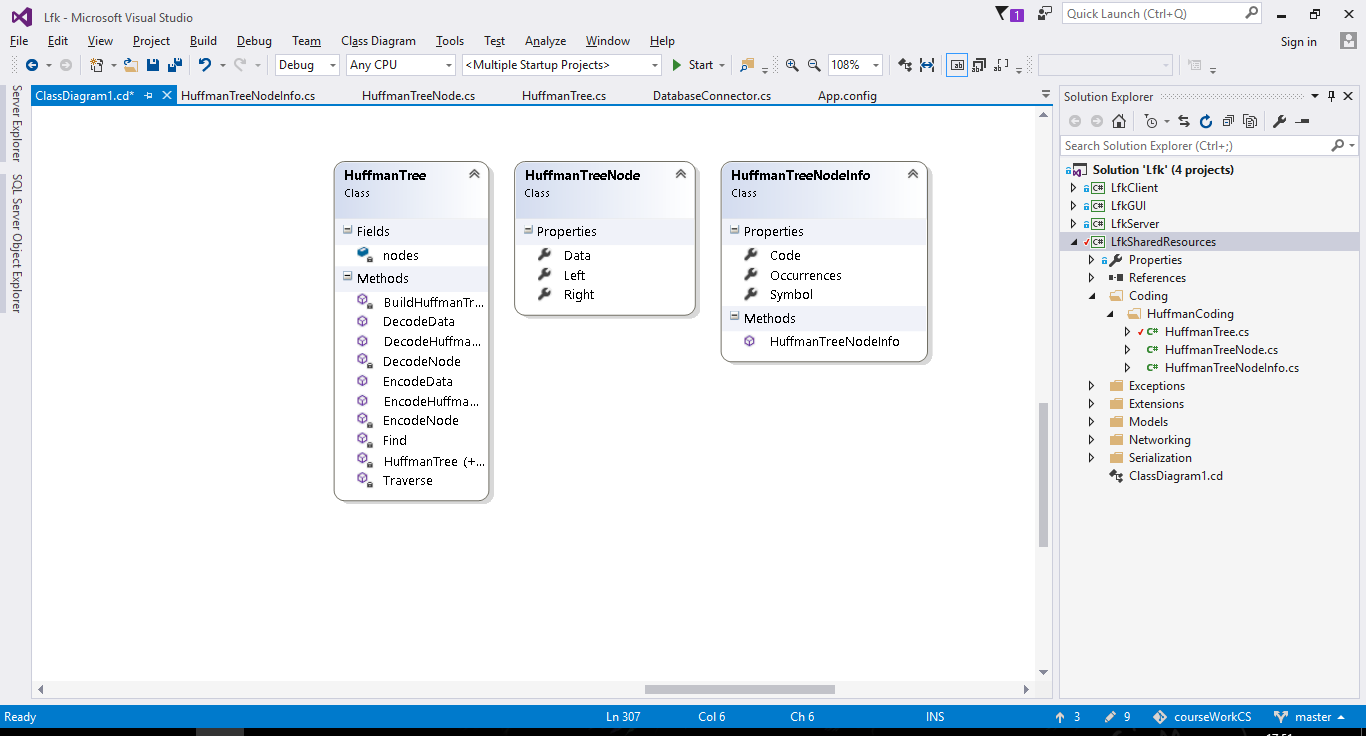
Поскольку идея данного модуля заключается в обеспечении программы независимыми друг от друга ресурсами, то какая-либо связь между компонентами LfkSharedResoures отсутствует. В общем виде данный модуль располагает следующими компонентами:

1. Coding: отвечает за сжатие / разжатие содержимого пользовательских файлов, призванный сократить объём данных, находящихся под управлением системы;
2. Exceptions: представляет из себя иерархию исключений, которые возникают в ходе работы программы, и призванный облегчить взаимодействие основных компонентов системы;
3. Extensions: содержит некоторые методы расширения для основных классов системы;
4. Models: содержит описание всех моделей, использующихся модулями системы;
5. Networking: отвечает за представление и формирование пакетов, передающихся по сети.
6. Serialization: отвечает за сериализацию и десериализацию любого объекта системы.

**Диаграммы, описание и реализация классов модуля**

Для сжатия данных в компоненте Coding было решено использовать популярный и достаточно эффективный алгоритм Хаффмана, в основе которого лежит две основные идеи: построение оптимального кодового дерева и построение отображения код-символ на основе данного дерева.

Диаграмма классов, реализующих данный алгоритм, представлена рисунке 2.2.1.

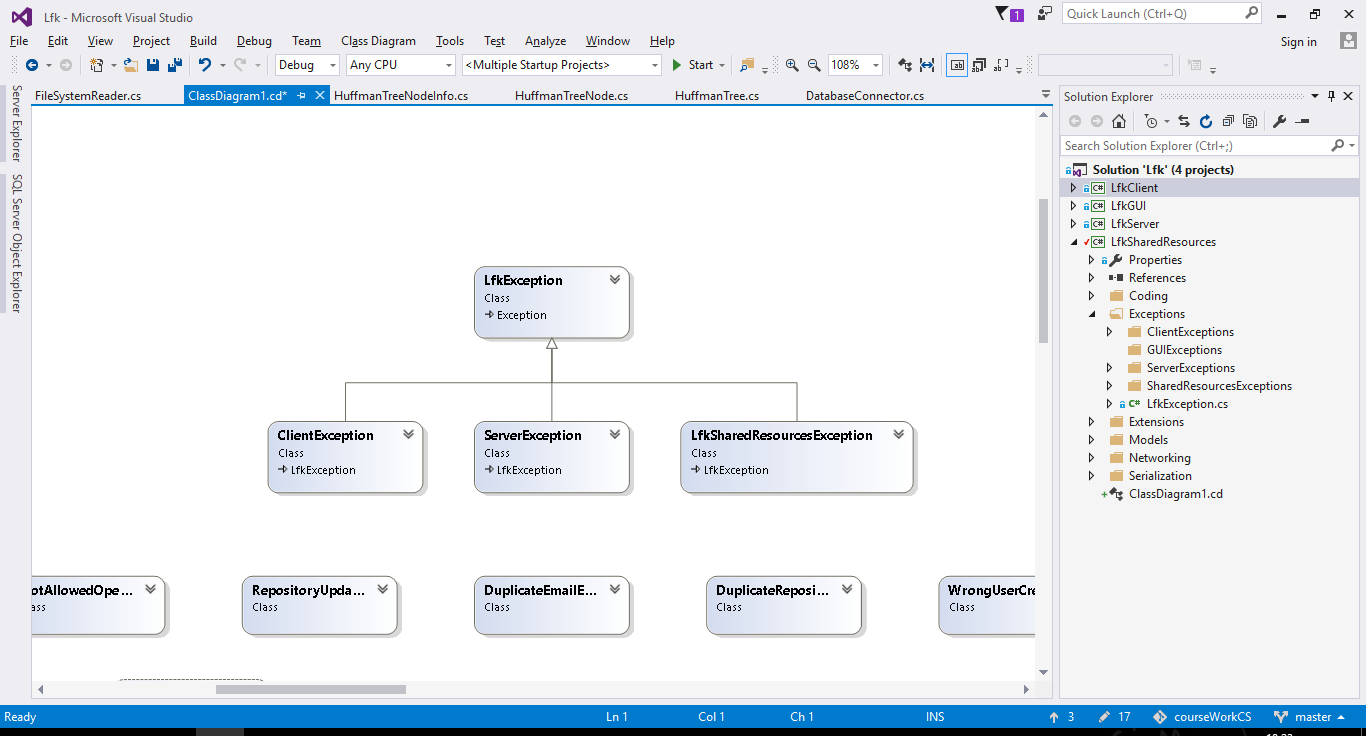


### Рисунок 2.2.1 – Диаграмма классов, реализующих алгоритм Хаффмана

Ключевым звеном в работе алгоритма является класс HuffmanTree, который ответственен за кодирование и раскодирование информации, а также за построение и разбор самого дерева Хаффмана на основе данной информации. Класс HuffmanTreeNode представляет из себя обычный узел бинарного дерева, который содержит некоторые данные, а также указатели на потомков. Класс HuffmanTreeNodeInfo представляет из себя информацию, которую хранит в себе узел дерева Хаффмана, то есть символ, количество его вхождений в исходную строку, а также его код, который формируется после построения дерева.

Идея алгоритма кодирования заключается в обходе каждого символа кодируемой строки и поиске количества его вхождений в данную строку. На основе этой информации строится само дерево, листьями которого и будут являться эти символы с метаданными в виде количества их вхождения в строку. Затем осуществляется обход построенного дерева Хаффмана с целью закодировать каждый лист дерева в соответствии с позицией обхода. Сжатие достигается засчёт того, что в результате обхода дерева самым встречающимся символам будут приписаны самые короткие коды, а наименее встречающимся – самые длинные. Такой подход, в зависимости от сжимаемых данных, может сократить их объём на 20-60%. Более подробно с принципами работы алгоритма Хаффмана можно ознакомиться в приложении таком-то.

Для осуществления беспрерывной работы программы при возникновении ошибки на любом этапе её выполнения, была разработана иерархия исключений, общая диаграмма которой представлена на рисунке 2.2.2.



### Рисунок 2.2.2 – Общая диаграмма исключений системы Lfk

Из диаграммы видно, что базовым для всех исключений является LfkExcetion, а его прямыми потомками – исключения для модулей LfkClient, LfkServer и LfkSharedResources. Рассмотрим более подробно исключения, присущие каждому из этих модулей.

Информация об исключениях, являющихся потомками ClientExcetion, представлена в таблице 2.2.1.

Таблица 2.2.1 – Описание исключений-потомков ClientException

|  |  |
| --- | --- |
| Исключение | Описание |
| ServerConnectionException | Ошибка соединения с сервером |
| FolderAlreadyContainsRepositoryException | Ошибка в результате попытки создать репозиторий в каталоге, уже содержащем репозиторий |
| NotAllowedOpenRepositoryException | Ошибка в результате попытки открыть репозиторий, принадлежащий другому пользователю |
| RepositoryUpdateWithoutCommitsException | Ошибка в результате попытки загрузить на сервер репозиторий, не содержащий коммитов |

Информация об исключениях, являющихся потомками ServerException, представлена в таблице 2.2.2.

Таблица 2.2.2 – Описание исключений-потомков ServerException

|  |  |
| --- | --- |
| Исключение | Описание |
| DuplicateRepositoryTitleException | Дублирование названия репозитория |
| WrongUserCredentialsException | Неверные пользовательские данные при авторизации |
| DuplicateEmailException | Дублирование e-mail пользователя при регистрации |

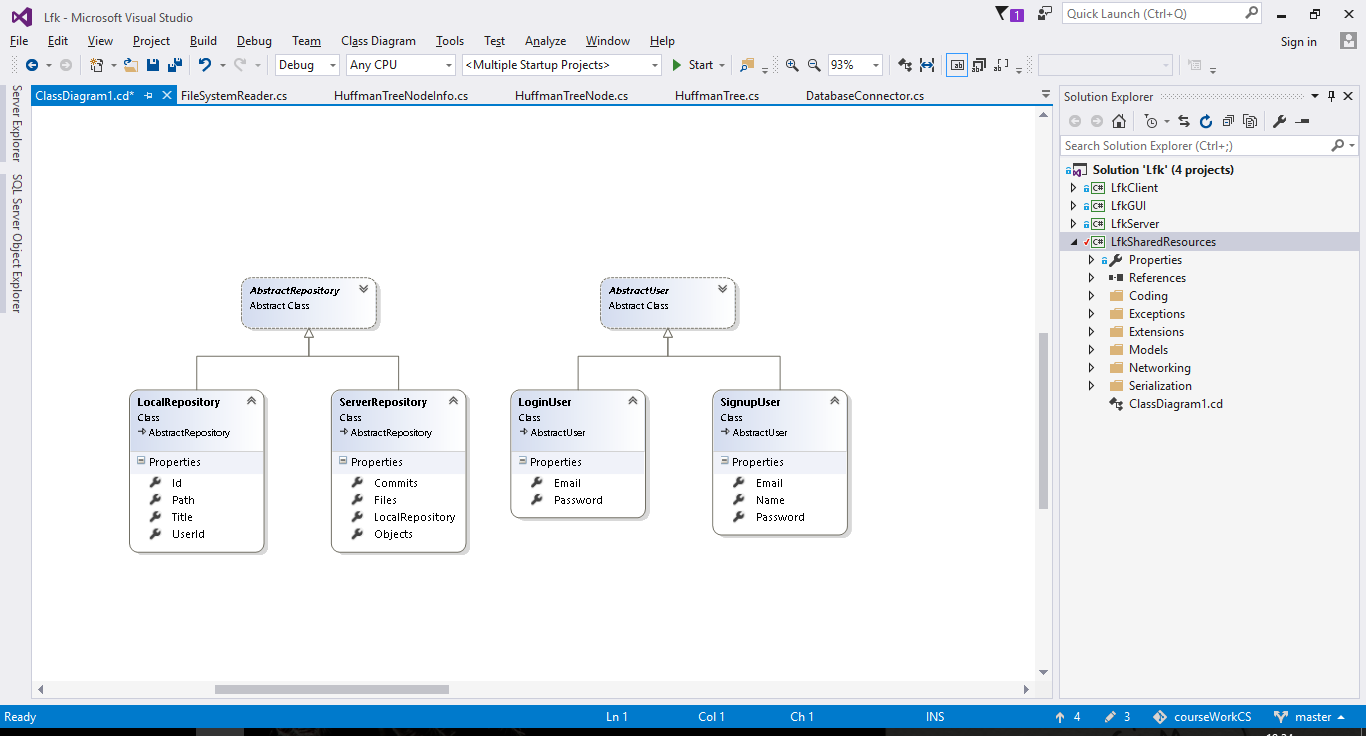
Информация об исключениях, являющихся потомками SharedResourcesException, представлена в таблице 2.2.3.

Таблица 2.2.3 – Описание исключений-потомков ServerException

|  |  |
| --- | --- |
| Исключение | Описание |
| JsonSerializationException | Ошибка сериализации в JSON |
| JsonSerializerInvalidDataException | Ошибка сериализации в JSON из-за некорректных данных |
| JsonSerializerNullArgumentException | Ошибка при получении получении входного параметра, не ссылающегося на объект в памяти |

Технически компонент Models представляет из себя набор классов, представляющих модели, которые используются во всех модулях системы. Все модели можно условно разделить на два вида: модели, используемые на стороне клиента (модуль LfkClient) и модели, используемые на стороне сервера (модуль LfkServer). Это сделано с той целью, чтобы не хранить на соответствующей стороне ненужную ей информацию. Совокупность всех моделей отражает две важнейшие сущности: пользователь и репозиторий.

Рассмотрим две основополагающих абстрактных модели пользователя и репозитория, диаграмма которых представлена на рисунке 2.2.3.

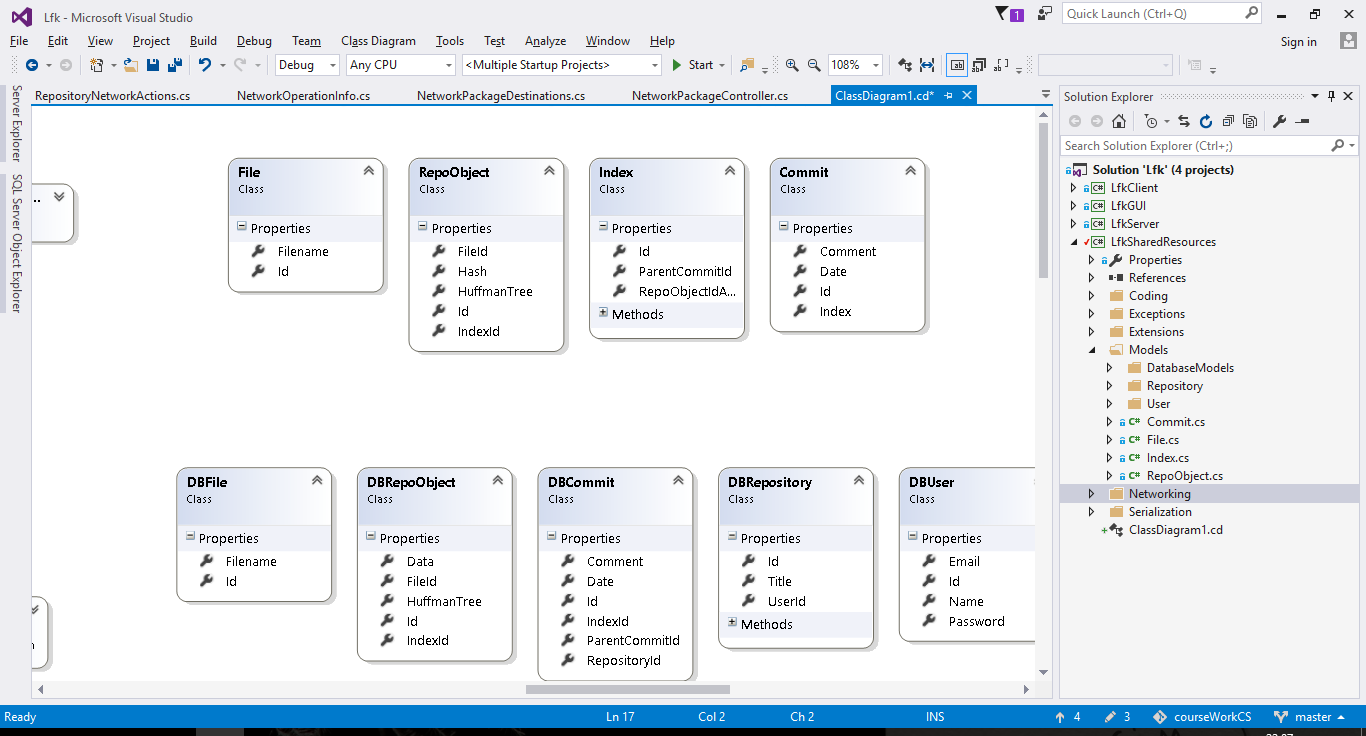


### Рисунок 2.2.3 – Основополагающие модели системы

Видно, что как для модели пользователя, так и для модели репозитория, существуют локальные и серверные аналоги. Локальные модели формируются в течение работы пользователя над своей рабочей директорией, однако при использовании команд, связанных с работой сервера, происходит трансформация локальных моделей в свои серверные аналоги.

AbstractUser представляет из себя модель для представления пользователя, в частности, такой информации, как его e-mail, пароль и имя.

С точки зрения модуля LfkClient, репозиторий представляет из себя класс LocalRepository, хранящей в себе идентификатор, получаемый со стороны сервера при создании репозитория, абсолютный путь на машине пользователя к данному репозиторию, его название и идентификатор пользователя, владеющего данным репозиторием. В течение сеанса работы, пользователь неявно проходит по всему циклу работы системы контроля версий, каждый из этапов представлен соответствующим классом, диаграмма которых представлена на рисунке 2.2.4.



### Рисунок 2.2.4 – Диаграмма классов, представляющих модели, используемые в течение всего цикла работы системы Lfk

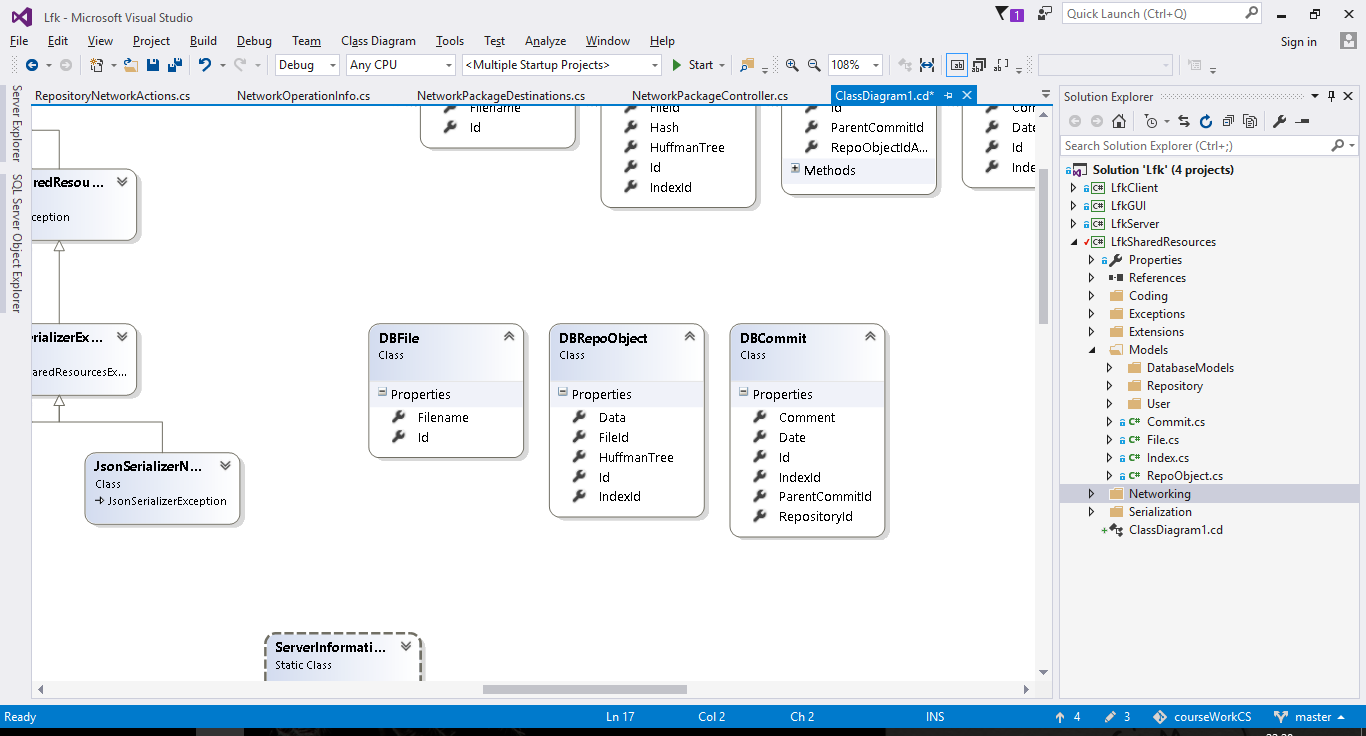
Как только файл попадает под контроль системы Lfk в результате выполнения команды include, для него вычисляется 128-битный глобальный уникальный идентификатор (GUID), после чего имя данного файла и вычисленный идентификатор сохраняются в системный файл files.json. Данная модель необходима для того, чтобы в ходе выполнения цикла работы не терялась информация о названиях файлах.

После выполнения команды add, в системную директорию objects/ сохраняется файл, содержащий в себе сериализованное содержимое объекта класса RepoObject (блоб-объект), представляющий снимок состояния файла в зафиксированный момент времени. Блоб-объект хранит в себе идентификатор файла, содержимое которого будет сохранено, сам идентификатор блоба-объекта, идентификатор текущего индекса, а непосредственно само содержимое, сжатое в результате работы алгоритма Хаффмана, вместе с соответствующим деревом.

Связующим звеном между командами add и commit служит модель Index, которая хранит в себе идентификатор индекса (вычисляющийся после каждой успешной команды commit), идентификатор родительского коммита, а также список пар, в которых связаны идентификатор блоба-объекта и соответствующие им имена файлов.

В результате выполнения команды commit, в системную директорию commits/ сохраняет файл, содержащий в себе сериализованное содержимое объекта класса Commit (коммит), представляющий снимок состояния рабочего каталога пользователя в зафиксированный момент времени. Коммит хранит соответствующей ему идентификатор, дату и время, пользовательский комментарий, а также текущий на данный момент индекс, который, по сути, и отражает состояние рабочего каталога.

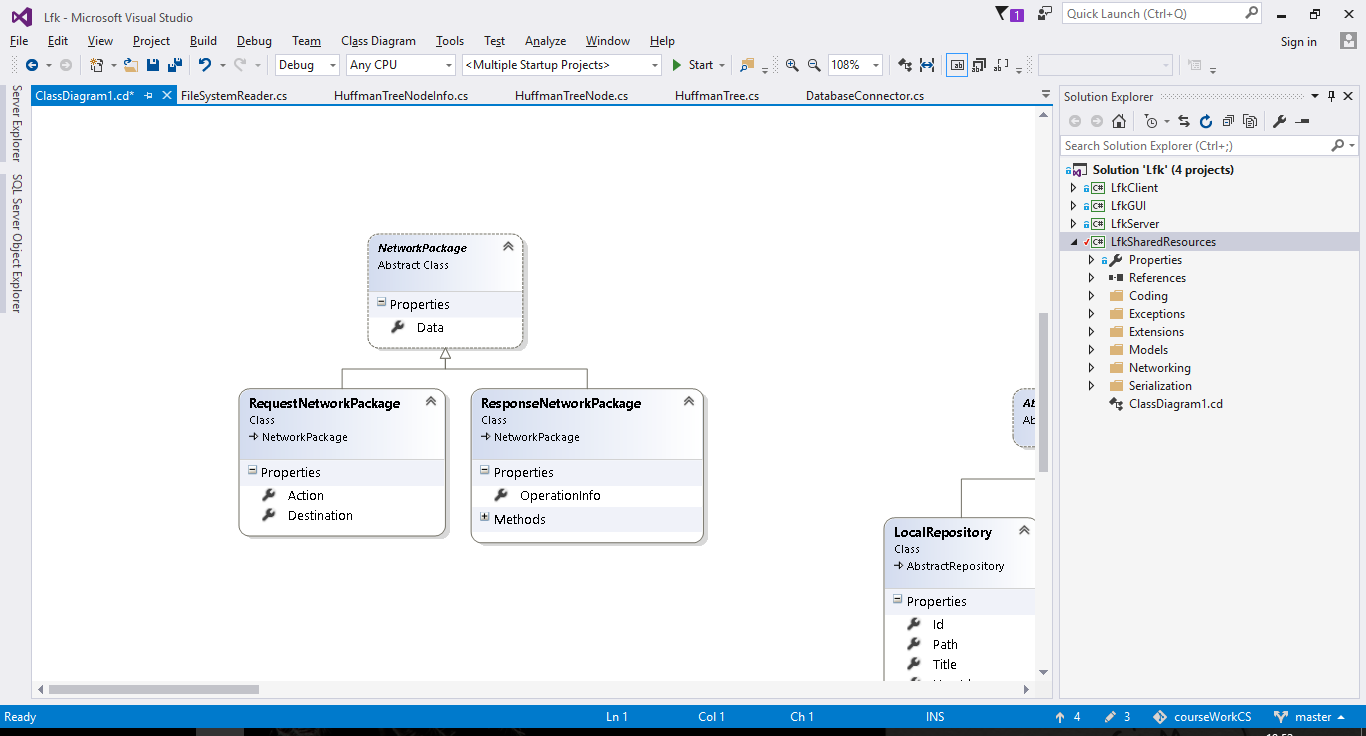
Все вышеописанные классы работают на локальном уровне. Как только выполняется команда update, все эти модели трансформируются в соответствующие им серверные аналоги, классы которых представлены на рисунке 2.2.5.



### Рисунок 2.2.5 - Диаграмма классов, представляющих модели, используемые для представления содержимого репозитория на стороне сервера

Данные модели существуют в виде свойств класса ServerRepository, представленного на рисунке 2.2.3. Как видно, необходимость в модели Index пропала, теперь блоб-объект и коммит связаны только идентификатором индекса. После передачи репозитория на сервер, всё его содержимое сохраняет в базе данных.

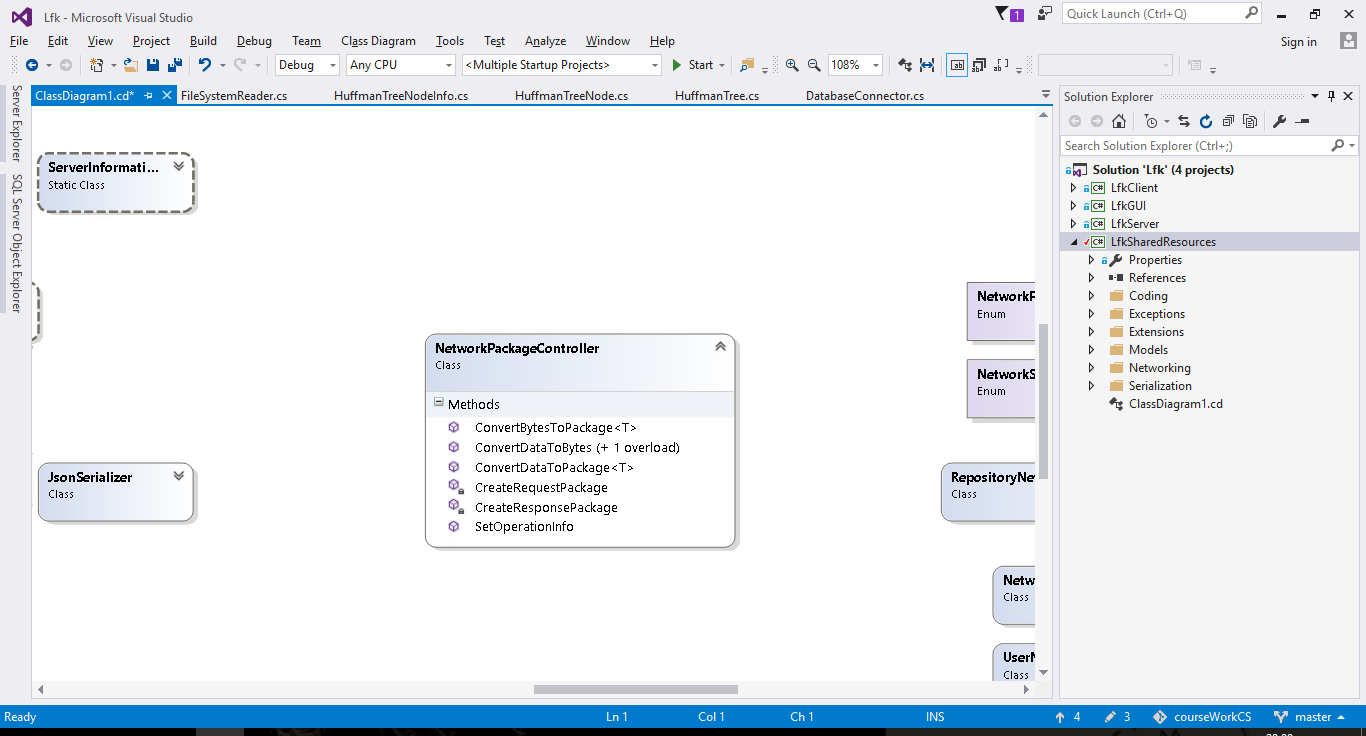
Компонент Networking является необходимым при передачи данных со стороны клиента к стороне сервера. Основу данного компонента составляет иерархия классов, представляющих собою сетевые пакеты, в которых содержится передаваемая информация, а также некоторые метаданные. Диаграмма классов, представляющих сетевые пакеты, представлена на рисунке 2.2.6.



### Рисунок 2.2.6 – Диаграмма классов, представляющих сетевые пакеты

Каждый сетевой пакет содержит в себе данные для передачи, которые технически представляют из себя набор байт. В случае запроса к серверу, к пакету (RequestNetworkPackage) добавляются метаданные в виде пункта назначения пакета, что говорит о характере передаваемых данных (информация о репозитории или пользователе), и действия, которые необходимо сделать по прибытию в данный пункт назначения (создать, прочитать, обновить или удалить). После успешной обработки запроса, серверная сторона формирует пакет с ответом (ResponseNetworkPackage), содержащий метаданные в виде отчёта по выполнению данного запроса (статус-код и сообщение).

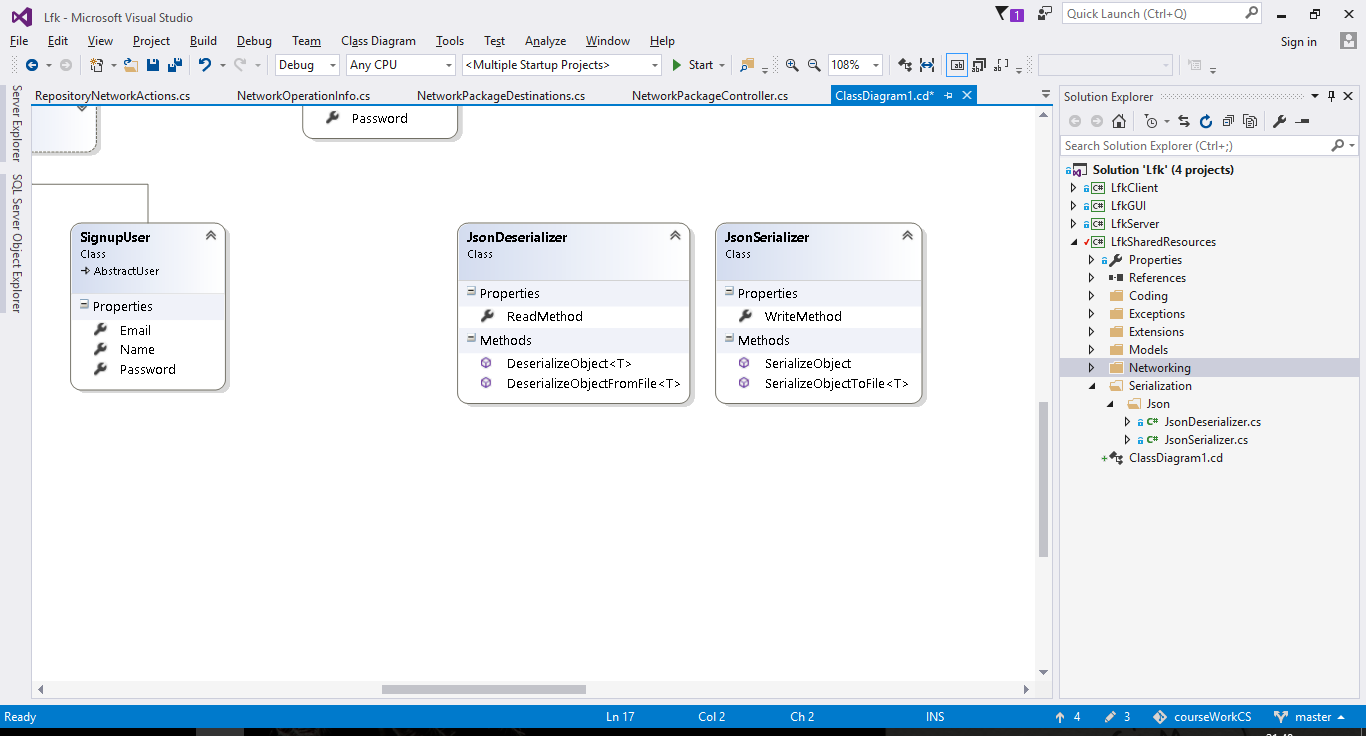
Формированием вышеописанных пакетов занимается специальный класс NetworkPackageController, структура которого представлена на рисунке 2.2.7.



### Рисунок 2.2.7 – Структура класса NetworkPackageController

Необходимость в наличии NetworkPackageController заключается в том, что данные между сервером и клиентом передаются в виде потока байт, и необходим некоторый посредник, способный корректно преобразовывать сформированные в ходе работы объекты в байты, и наоборот, превращать поток байт в объекты, которыми оперируют клиентская и серверная стороны.

Компонент Serialization берёт на себя ответственность за сериализацию и десериализацию любых данных, которыми оперирует система. На данный момент компонент Serialization поддерживает только JSON формат. Диаграмма классов, реализующих логику десериализации и сериализации представлена на рисунке 2.2.8.



### Рисунок 2.2.8 – Классы для десериализации и сериализации

Оба класса имеют по два метода для самой сериализации и десериализации данных, а также их аналоги, работающие непосредственно с файловой системой. Свойства ReadMethod и WriteMethod представляют из себя делегаты и служат своеобразной настройкой сериализаторов, указывающей методам, работающим с файловой системой способ чтения и записи файлов.

## **2.3 Модуль LfkGUI**

**Предназначение модуля**

Данный программный модуль предназначен для визуализации рабочего процесса системы контроля версий Lfk. LfkGUI частично реализует фильтрацию и валидацию вводимых пользователем данных. Также он предоставляет возможность ведения диалога с пользователем, и визуализацию сообщений об ошибках, возникающих в других модулях.

**Структура модуля LfkGUI**

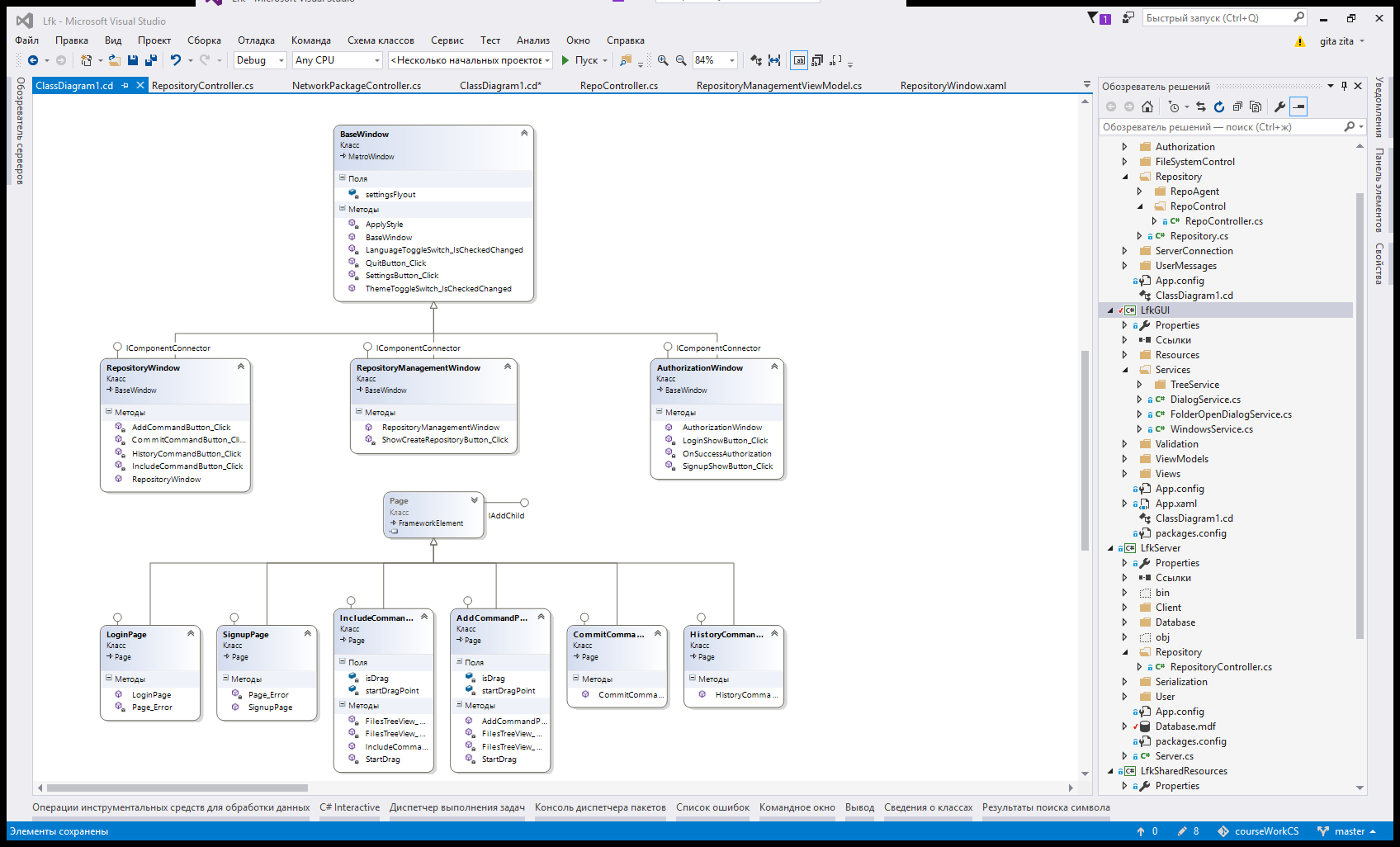
Данный модуль реализован на основе популярного шаблона проектирования MVVM(Model-View-ViewModel). В его состав входят представления, модели представлений, а также валидационные правила и дополнительные сервисы, отвечающие за диалоговые окна и навигацию: WindowService – отвечает за навигацию между окнами, DialogService – отвечает за вывод сообщений пользователю, FolderOpenDialogService – отвечает за диалоговое окно для выбора директорий. LfkGUI также содержит ряд ресурсов, которые содержат стили, словари локализации. На рисунке 2.3.1 представлена схема взаимодействия компонентов модуля LfkGUI.



### Рисунок 2.3.1 – Схема взаимодействия компонентов модуля LfkGUI

**Диаграммы, описание и реализация классов модуля**

При разработке данного проекта использовался паттерн проектирования MVVM, позволяющий отделить представление данных от их обработки. На рисунке 2.3.2 представлена диаграмма классов, отвечающих за представление моделей(View). Связанные с ними классы моделей представлений показаны далее на рисунке 2.3.5.

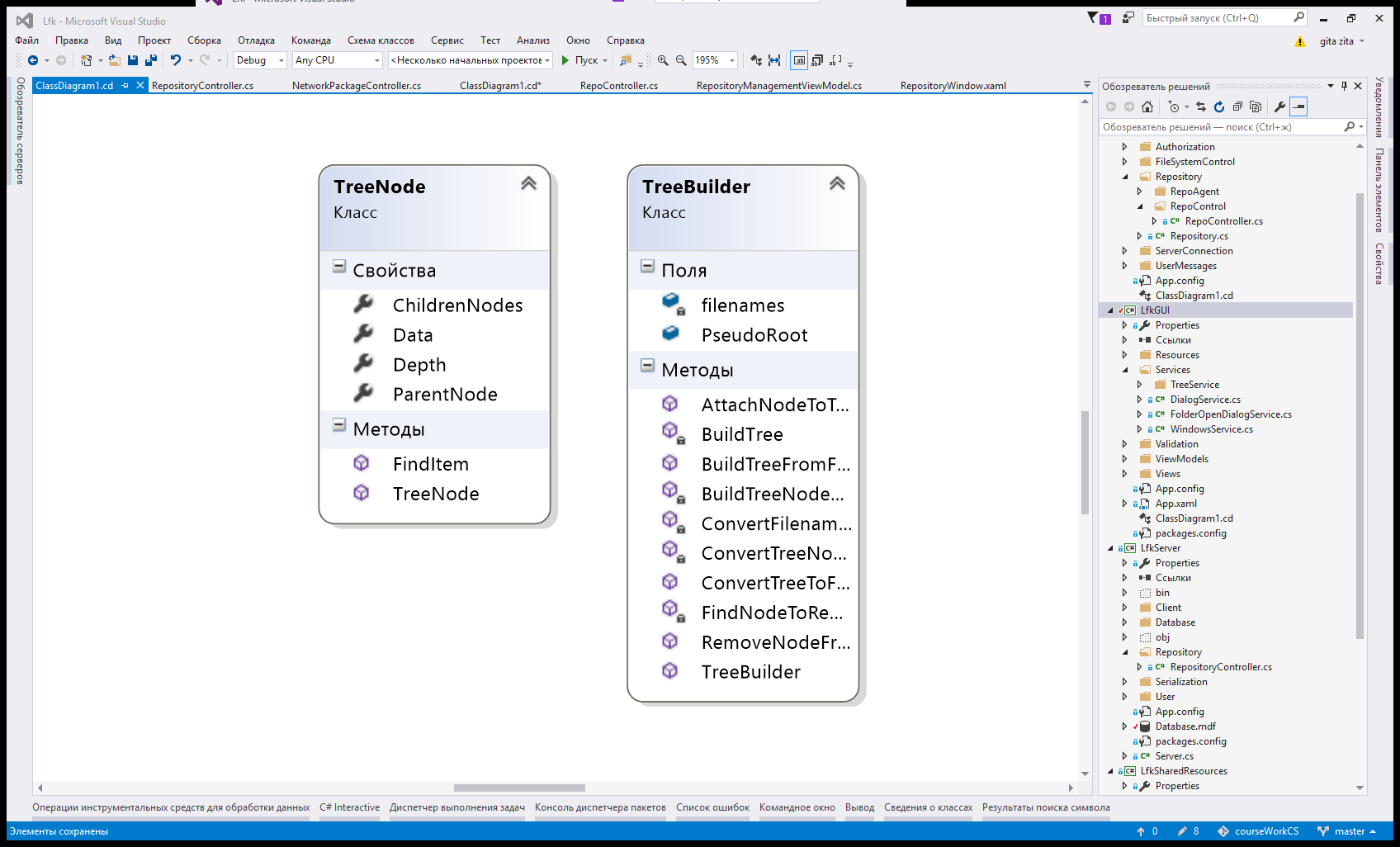


### Рисунок 2.3.2 – Диаграмма классов представлений

Класс BaseWindow является базовым классом для используемых в проекте окон, данный класс описывает основной стиль графического интерфейса, также в нем реализованы функции меню общие для всех окон программы, такие как смена стиля программы, смена языка программы и возможность сменить пользователя в любом окне программы. Необходимо отметить также, что данный класс в силу особенностей наследования классов WPF, не содержит в себе XAML разметки, и все базовые стили и функционал прописаны программно. На диаграмме классов видно, что от класса BaseWindow наследуются три класса, которые и являются главными окнами взаимодействия с сущностью LfkClient.

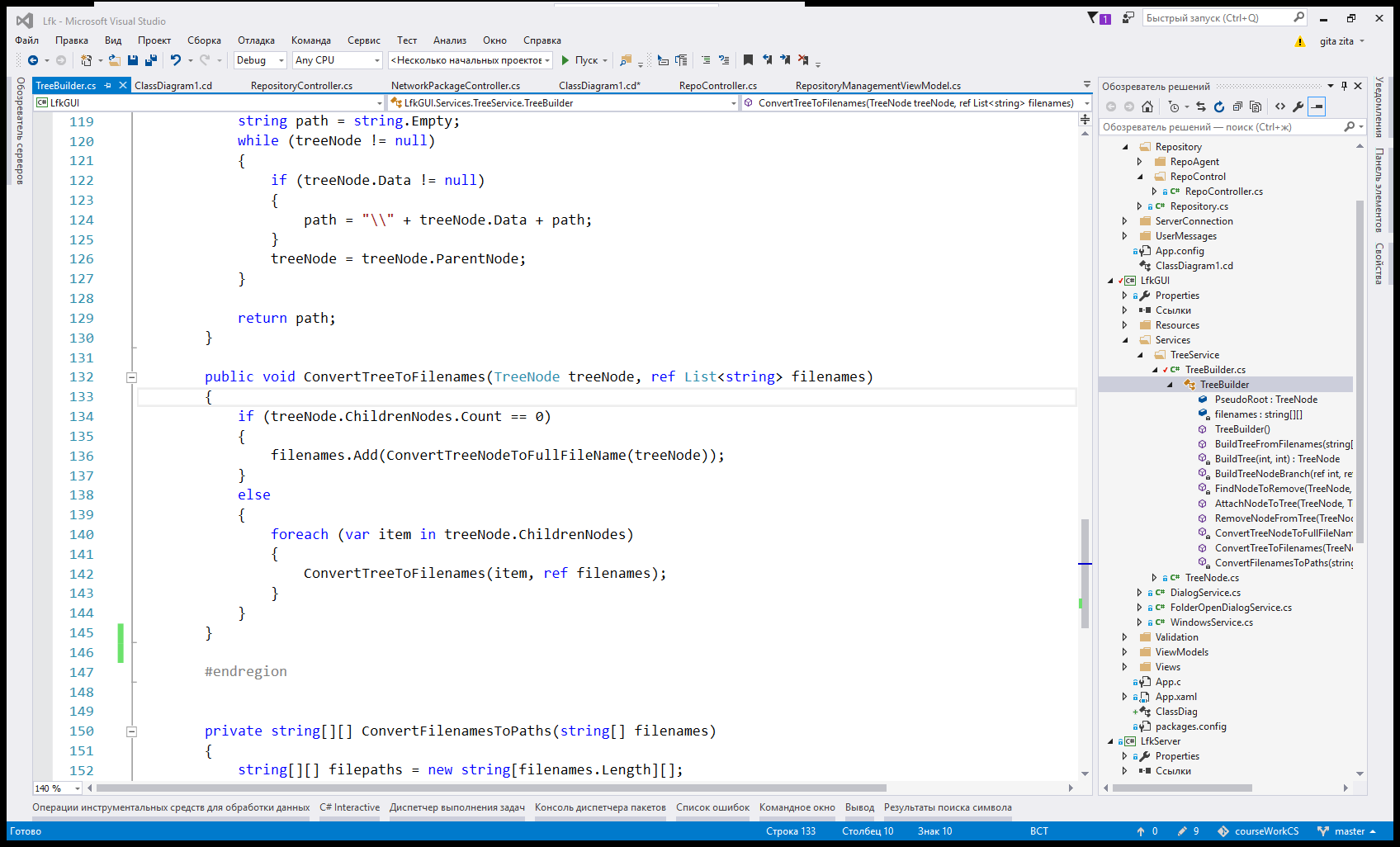
Класс AuthorizationWindow отвечает за авторизацию пользователя, и содержит в себе две страницы LoginPage и SignupPage. LoginPage отвечает за авторизацию пользователя и двусторонне связан с моделью представления LoginViewModel, которая в свою очередь отвечает за вход и идентификацию пользователя. В свою очередь SignupPage визуализирует процесс регистрации пользователя и связан с SignupViewModel. Класс RepositoryManagementWindow представляет пользователю возможность управления пользовательскими репозиториями. Класс RepositoryWindow отвечает за визуализацию действий производимых с пользовательскими файлами, и основных комманд системы контроля версий. Каждой комманде соответствует свой класс Page, который связан с моделью представлений.

Хотелось бы отметить что в визуальном представлении рабочей директории используется элемент TreeView, который позволяет представлять элементы иерархически, в виде раскрывающихся списков. Специально для этого была создана модель являющаяся узлом дерева, а для формирования дерева был написан класс TreeBuilder, который из списка файлов формирует дерево, также в процессе управления файлами пользователь может динамически добавлять и удалять узлы дерева. На рисунке 2.3.3 представлена структура этих классов.



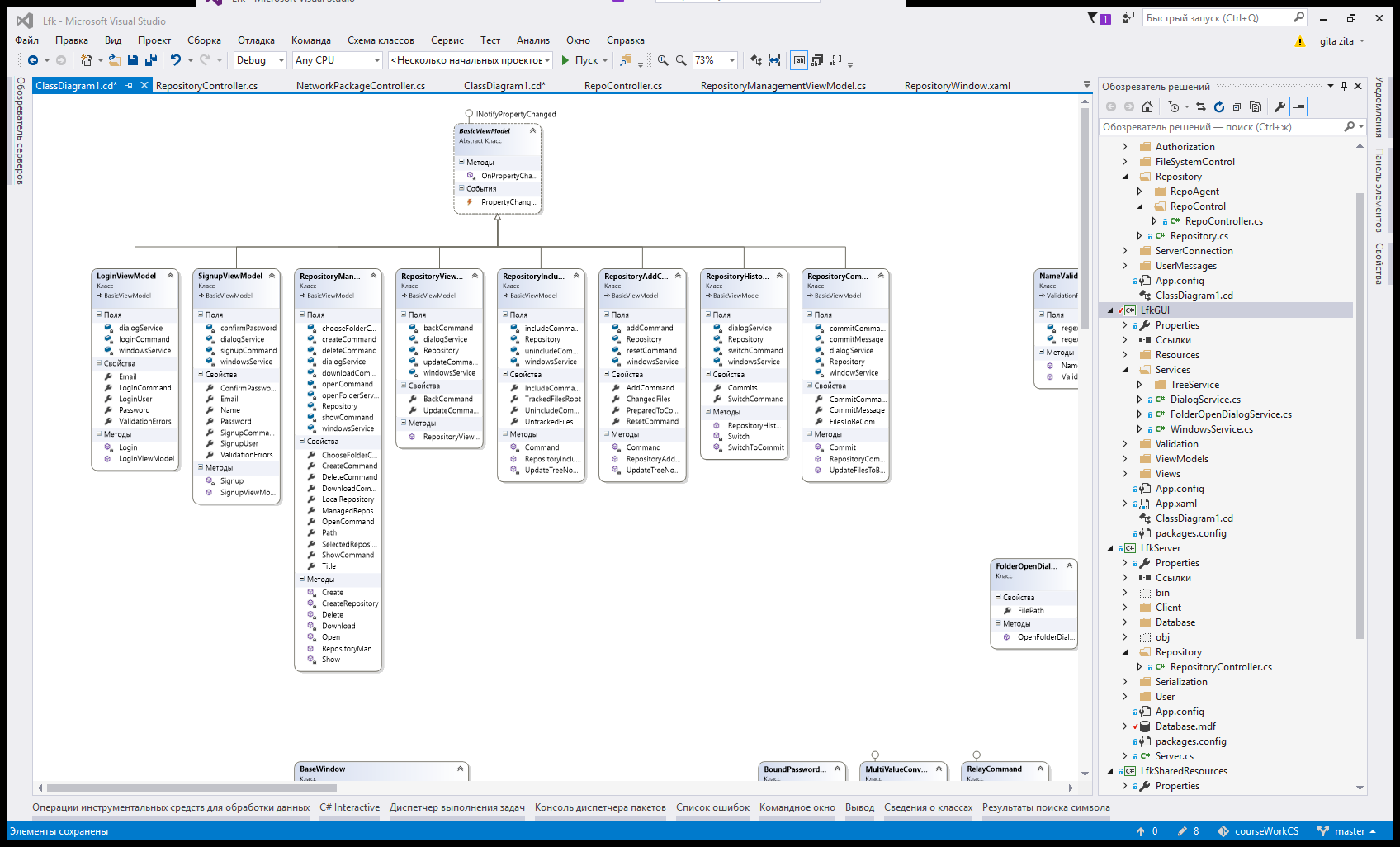
### Рисунок 2.3.3 – Диаграмма классов для представления пользовательской директории

Механизмы построения, добавления, и удаления элементов, реализованы на основе рекурсивных алгоритмов обхода дерева. На рисунке 2.3.4 представлен код рекурсивной функции, которая позволяет преобразовать узел в список путей файлов.



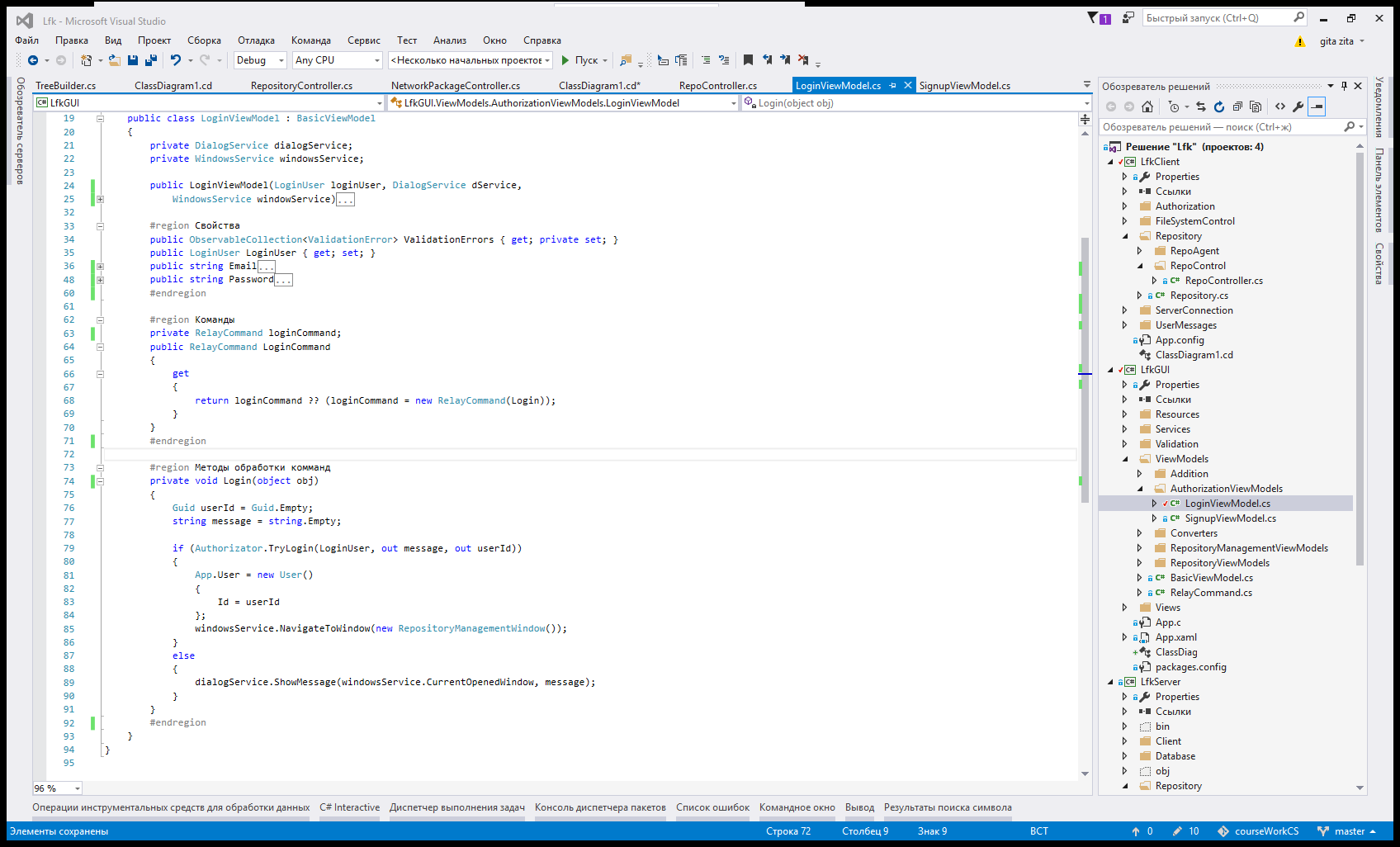
### Рисунок 2.3.4 – Функция построения путей файлов из узла дерева

Ниже представлена диаграмма классов моделей представления, тоесть классов, которые отвечают за обработку пользовательских действий, и последующий вызов функций остальных модулей. Необходим отметить что базовый для всех моделей представления класс реализует интерфейс INotifyPropertyChanged, что позволяет оповещать привязанные к свойствам контролы, об изменении свойств. Также активно используется специальный класс ObservableCollection. Он по функциональности похож на список List за тем исключением, что позволяет известить внешние объекты о том, что коллекция была изменена.



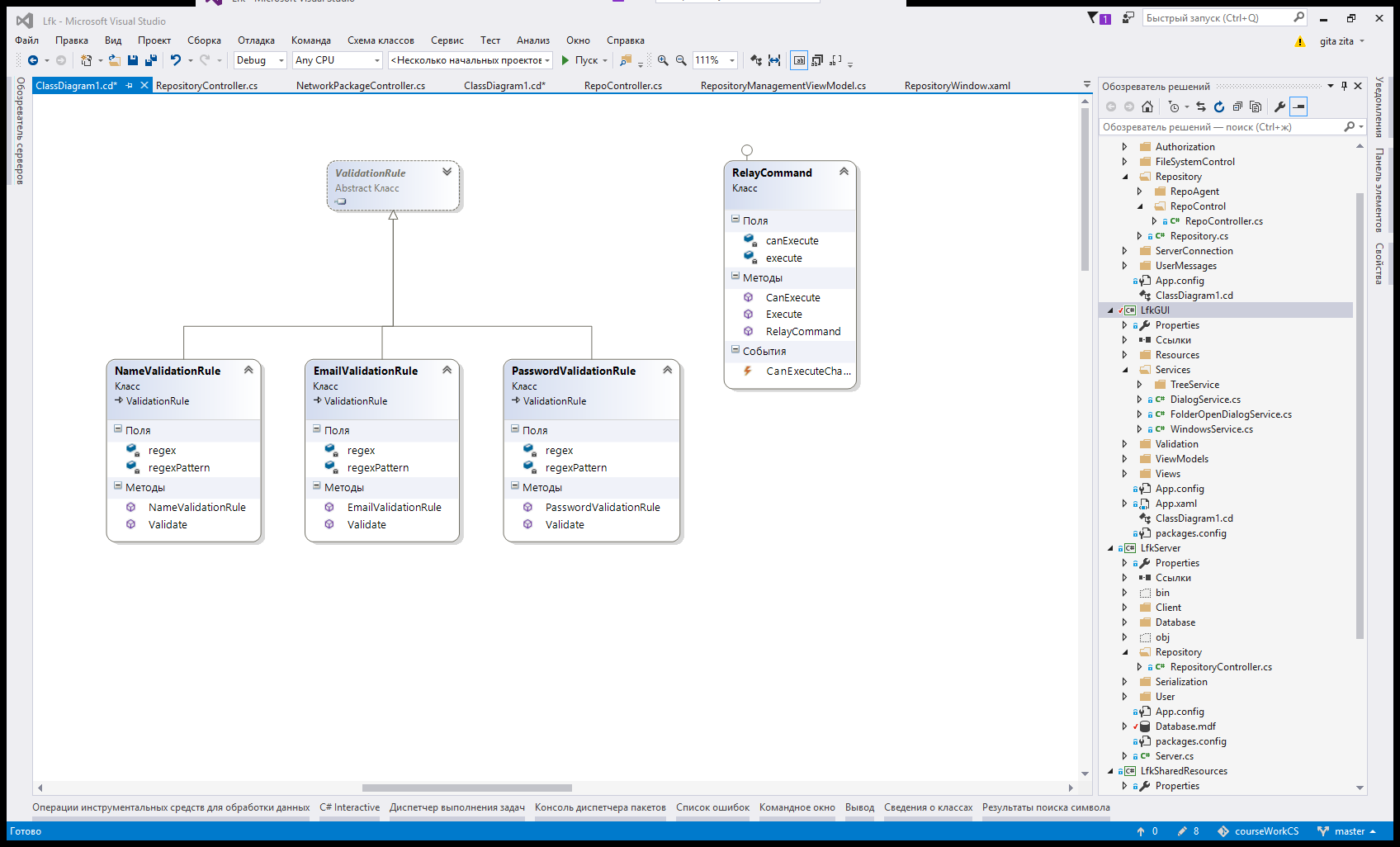
### Рисунок 2.3.5 – Диаграмма классов моделей представления

На рисунке 2.3.6 представлен код реализации модели представления LoginViewModel, который отражает основные принципы построения ViewModel в проекте LfkGUI.



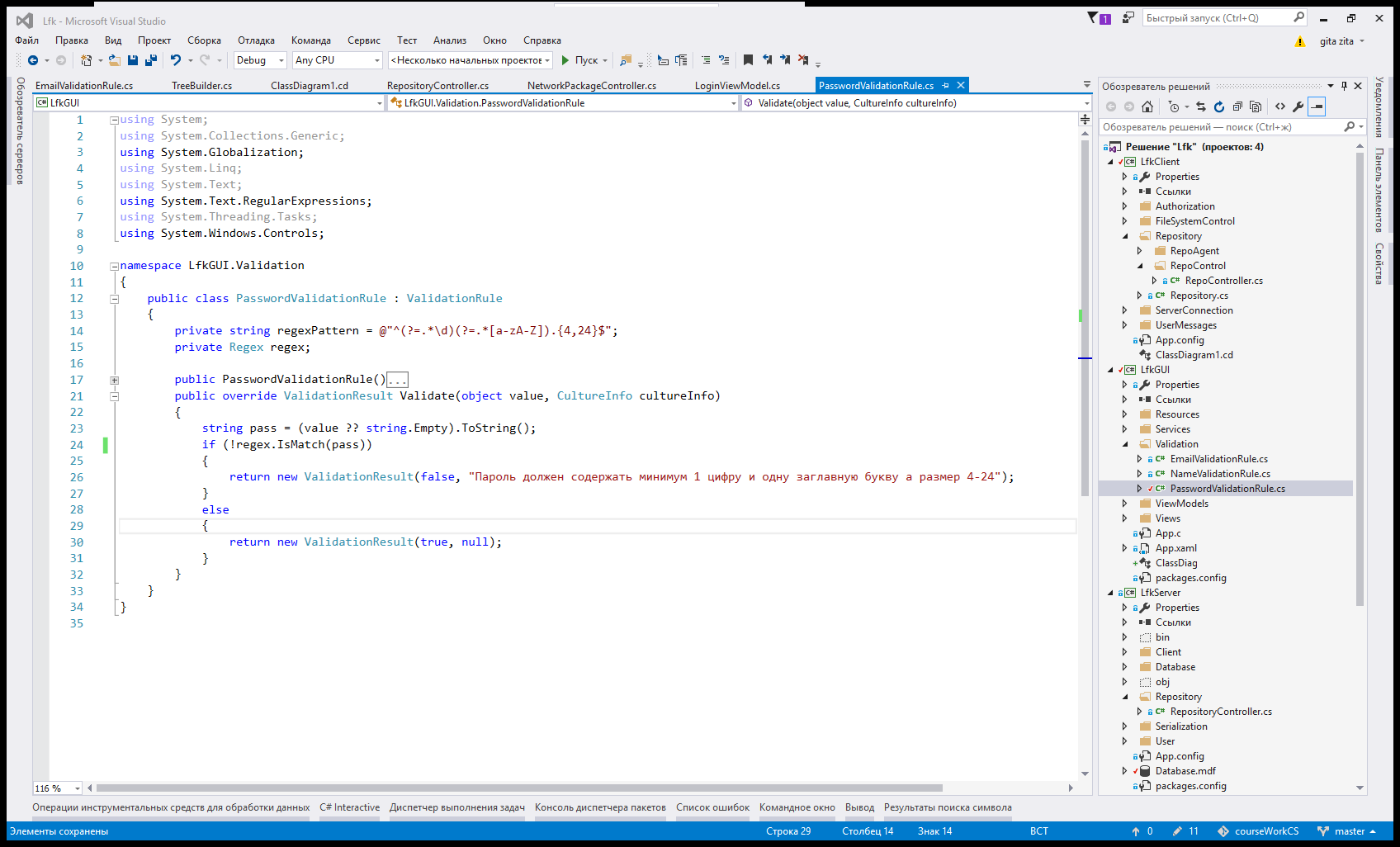
### Рисунок 2.3.6 – Код реализации модели представления

Каждая модель представления содержит в себе свойства, привязанные к элементам интерфейса, также, существует система валидации вводимых пользователем данных, которая основана на том, что пользователю не позволяется выполнять команды, пока введенные данные не удовлетворят необходимым условиям. Данная система основана на валидационных правилах и командах. Ошибки отлавливаются на уровне представления, и при наличии хоть одной, во избежание непредвиденных последствий выполнение основных команд блокируется, и элемент управления отвечающий за исполнение команды становится неактивным. На рисунке 3.3.4 представлена диаграмма классов, отвечающих за валидацию, которая в основана на механизме регулярных выражений. Также ниже отображен класс, представляющий собой команду.



### Рисунок 2.3.7 – Диаграмма классов валидации и класса команды

Программная реализация одного из валидационного правила представлена на рисунке 2.3.8



### Рисунок 2.3.8 – Код реализации класса, отвечающего за валидацию

Необходимо отметить, что для стилизации графического интерфейса использовалась библиотека MaterialDesign, которая включает в себя огромный набор стилей, для всех видов элементов управления. Также для расширения функционала стандартного класса Window была подключена библиотека MahApps.Metro, которая позволила добавить меню, общее для всех окон, и систему диалоговых окон.

## **2.4 Модуль LfkClient**

**Предназначение модуля**

Данный программный модуль предназначен для управления авторизацией, работы с основным функционалом системы контроля версий, и взаимодействия с серверной частью приложения.

**Структура модуля LfkClient**

LfkClient можно разделить на пять компонент, три основных и две вспомогательных. Основные компоненты:

1. Authorizator – компонента отвечающая за отправку идентификационных данных пользователя на сервер, и возврат ответа графическому интерфейсу.
2. RepoController – компонента отвечающая за управление пользовательскими репозиториями. Содержит функционал инициализации, создания, скачивания и удаления репозиториев.
3. RepoAgent - компонента, отвечающая за основной функционал работы с локальным репозиторием. Отвечает за такие команды как, включение пользовательских файлов в список отслеживаемых, фиксация изменений, переключение между версиями, загрузка изменений на удаленный сервер.

Дополнительные компоненты:

1. FileSystemControl – класс для работы с файловой системой.
2. ServerConnector – класс для взаимодействия с сервером.

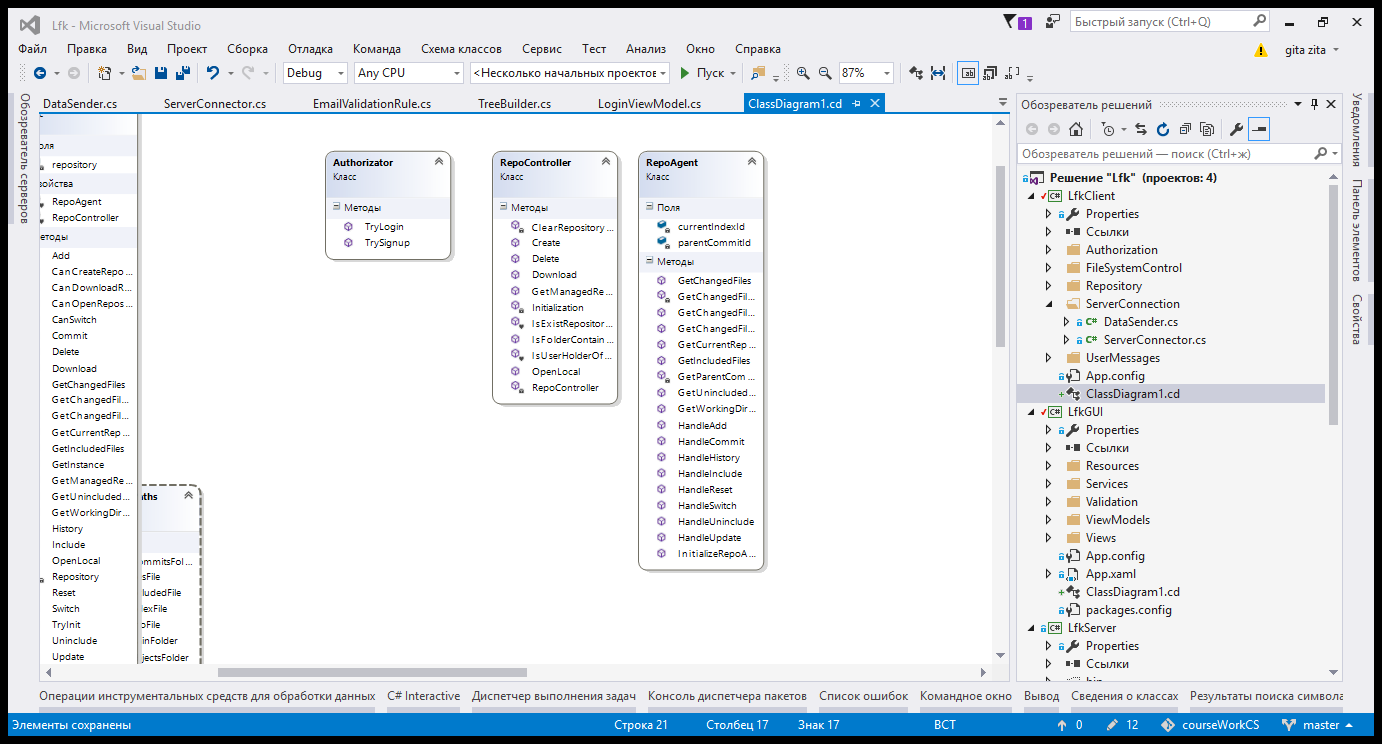
При разработке данного модуля активно использовался шаблон проектирования фасад - структурный шаблон, позволяющий скрыть сложность системы путём сведения всех возможных внешних вызовов к одному объекту, делегирующему их соответствующим объектам системы. На рисунке 2.4.1 представлена схема взаимодействия модуля LfkClient.



### Рисунок 2.4.1 – Схема взаимодействия компонент модуля LfkClient

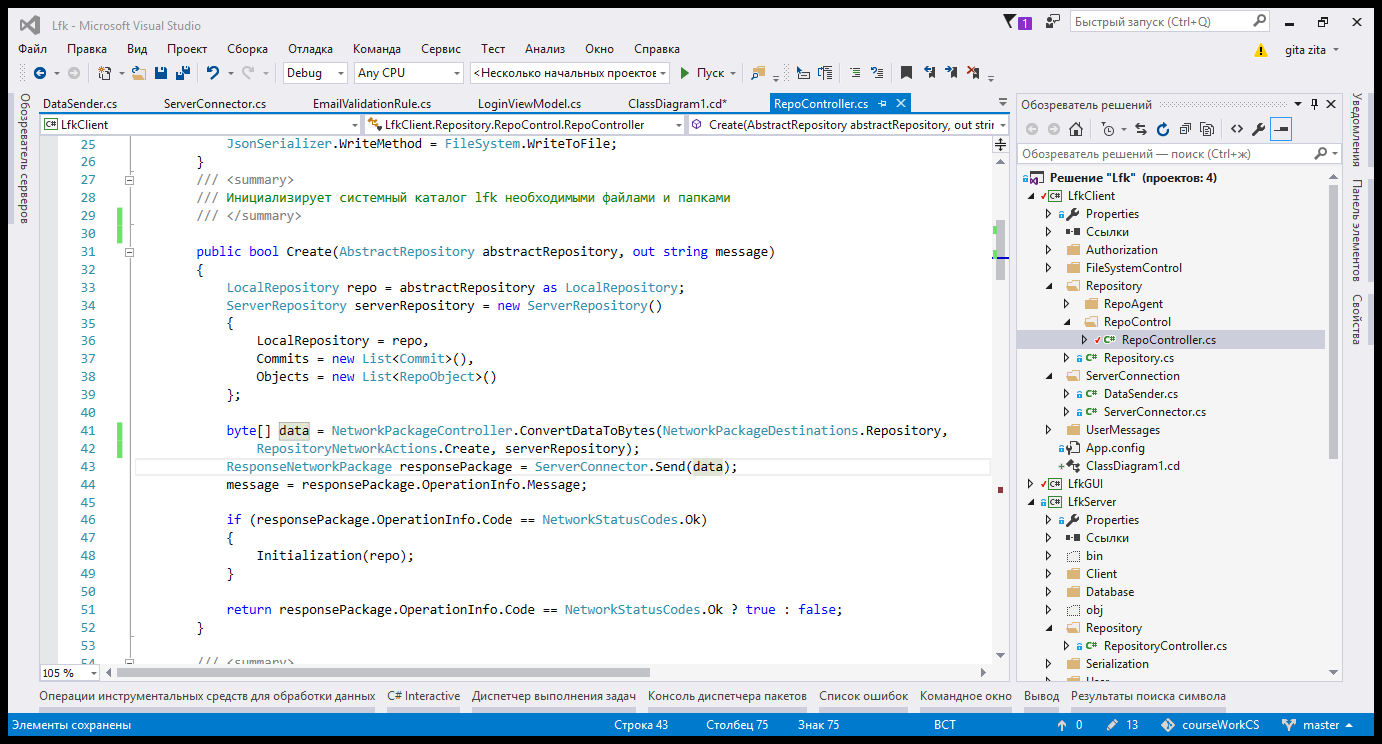
**Диаграммы, описание и реализация классов модуля**

При разработке данного модуля использовался паттерн проектирования «Фасад». Фасадными объектами являются классы: Repository, FileSystem. Класс Repository помимо функций RepoAgent и RepoController также содержит некоторые дополнительные функции управления репозиторием, что обеспечивает использование всего функционала системы контроля версий через один класс. На рисунке 2.4.2 изображена диаграмма основных классов.



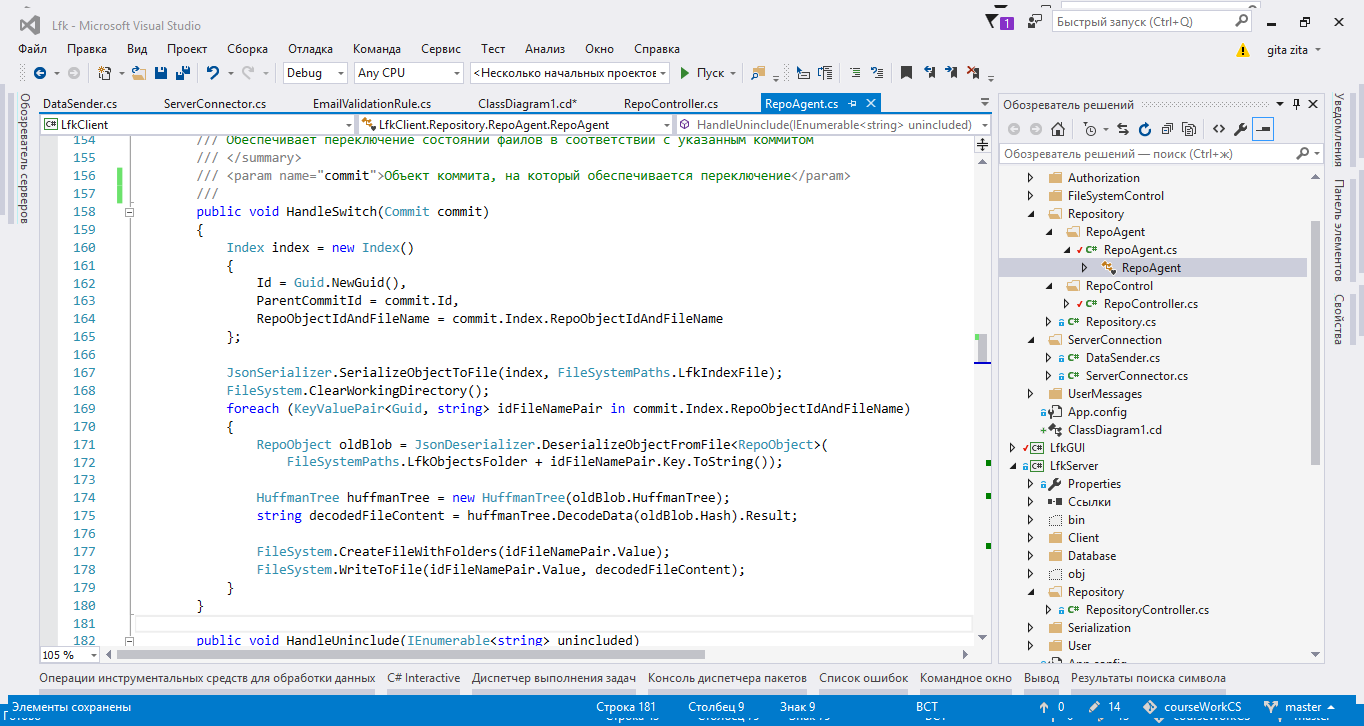
### Рисунок 2.4.2 – Диаграмма основных классов модуля

Класс RepoController предоставляет функции работы с пользовательскими репозиториями, помимо этого у него есть ряд методов, проверяющих возможность выполнения той или иной команды. Результатом работы этих функция являются специально определенные перечисления, позволяющие точнее выявить причину невозможности выполнения действия. На рисунке 2.4.3 приведен пример функции создания пользовательского репозитория.



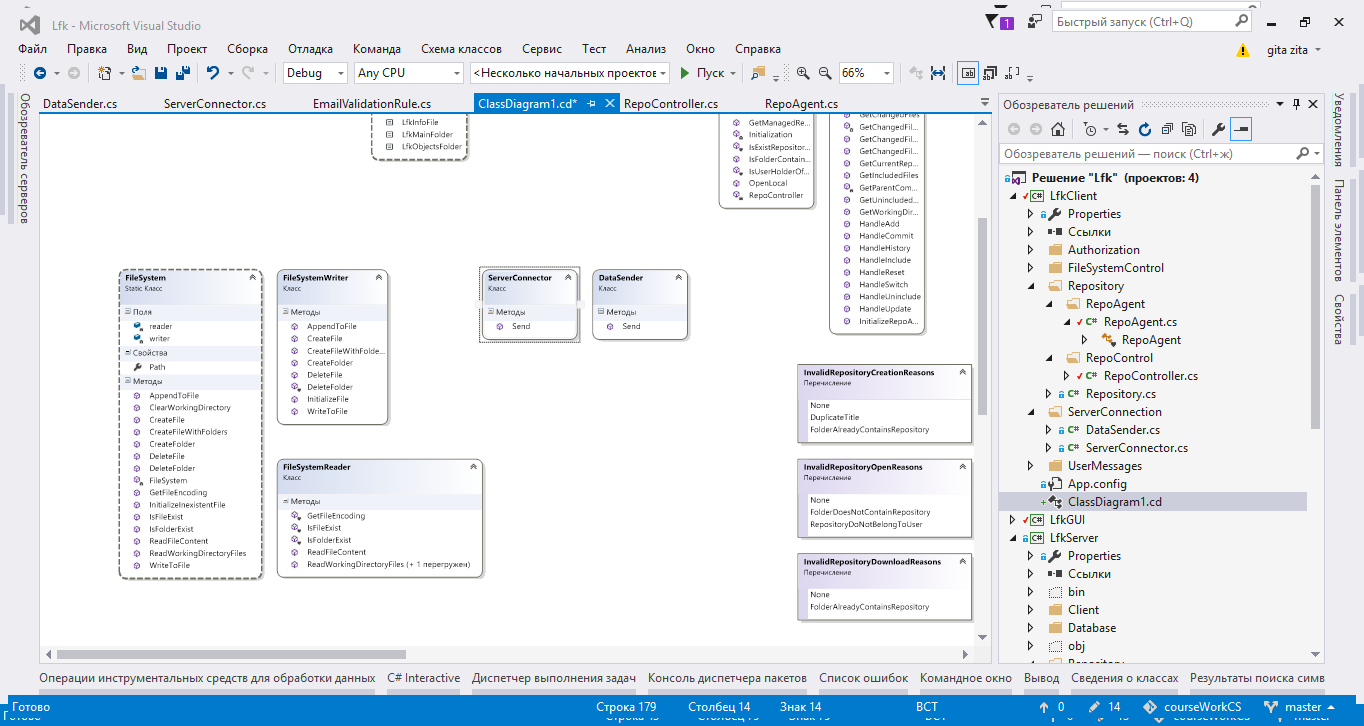
### Рисунок 2.4.3 – Код реализации функции создания репозитория

Класс RepoAgent содержит в себе основной функционал системы контроля версий, также в нем определены методы, для получения информации о статусе пользовательских файлов и самого репозитория. На рисунке 2.4.4 приведен код реализации функции switch.



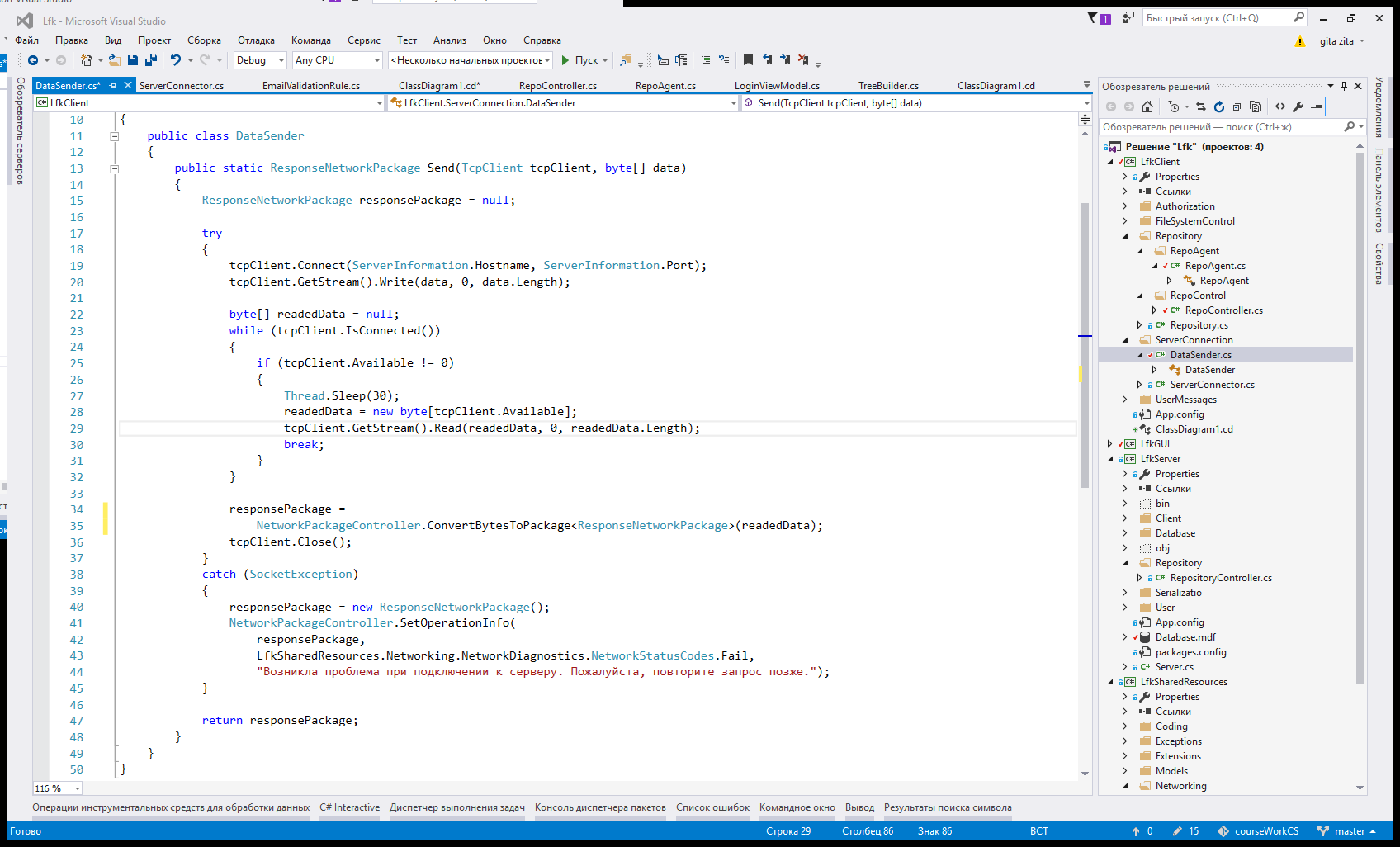
### Рисунок 2.4.4 – Программная реализация команды switch

LfkClient также включает в себя ряд вспомогательных классов, диаграмма которых представлена на рисунке 2.4.5.



### Рисунок 2.4.5 – Диаграмма вспомогательных классов

Класс FileSystem инкапсулирует в себе всю работу с пользовательскими файлами и директориями, такой подход позволяет обрабатывать исключения ввода-вывода в одном месте и определить свою логику их обработки. Обязанности работы с файловой системой распределены между двумя классами FileSystemWriter и FileSystemReader, для записи и чтения соответственно. Класс DataSender содержит в себе один единственный метод, отправки пакета на сервер, и последующего считывания ответа. В случае возникновения неполадок на стороне сервера, вызывающим модулям будет отправлено соответствующее сообщение. На рисунке 2.4.6 представлена реализация этого класса.



### Рисунок 2.4.6 – Реализация метода соединения с сервером

Взаимодействие с сервером осуществляется посредством использования класса NetworkPackage определенного в модуле LfkSharedResources. Соединение же устанавливается стандартным классом TcpClient.

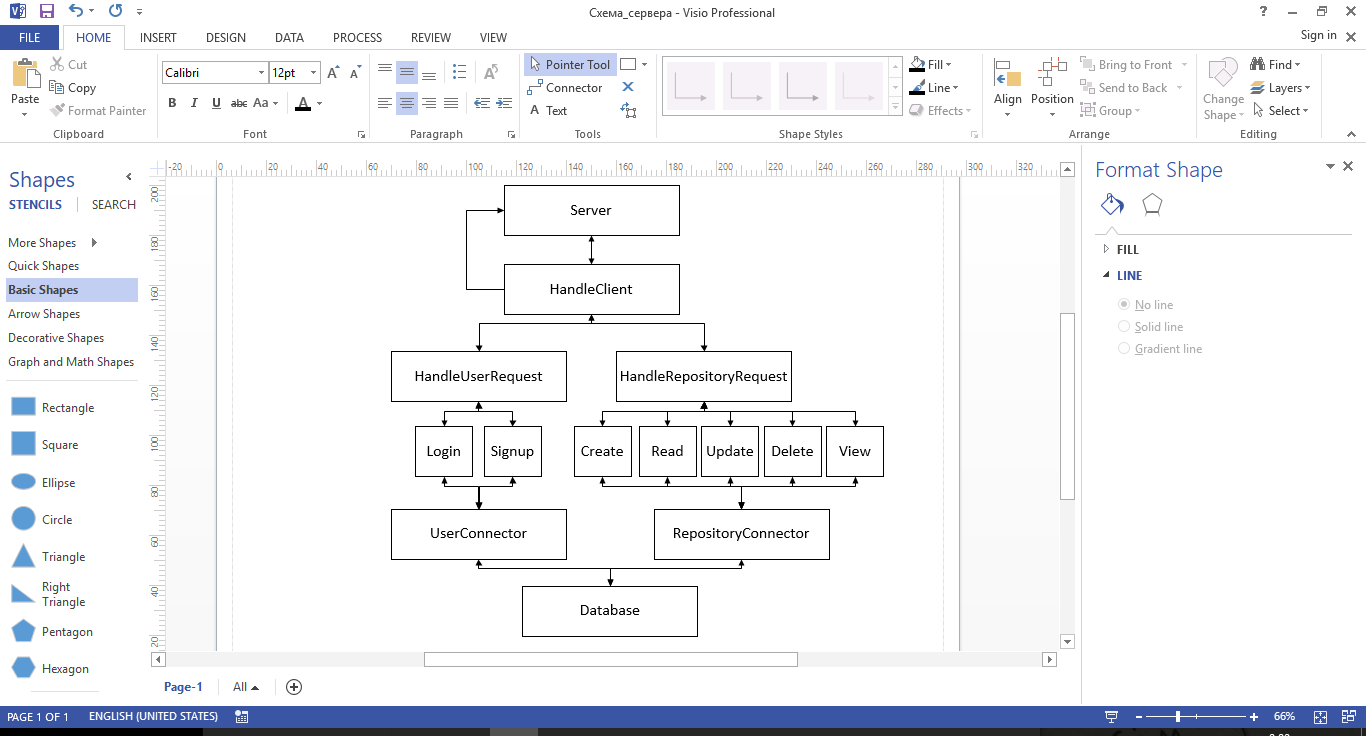
## **2.5. Модуль LfkServer**

**Предназначение модуля**

Система контроля версий Lfk является централизованной, то есть основным хранилищем пользовательских данных выступает сервер, куда можно сохранять и откуда можно выгружать необходимые репозитории. Реализацией такого сервера выступает модуль LfkServer.

**Структура модуля LfkServer**

Общая схема работы модуля LfkServer представлена на рисунке 2.5.1.



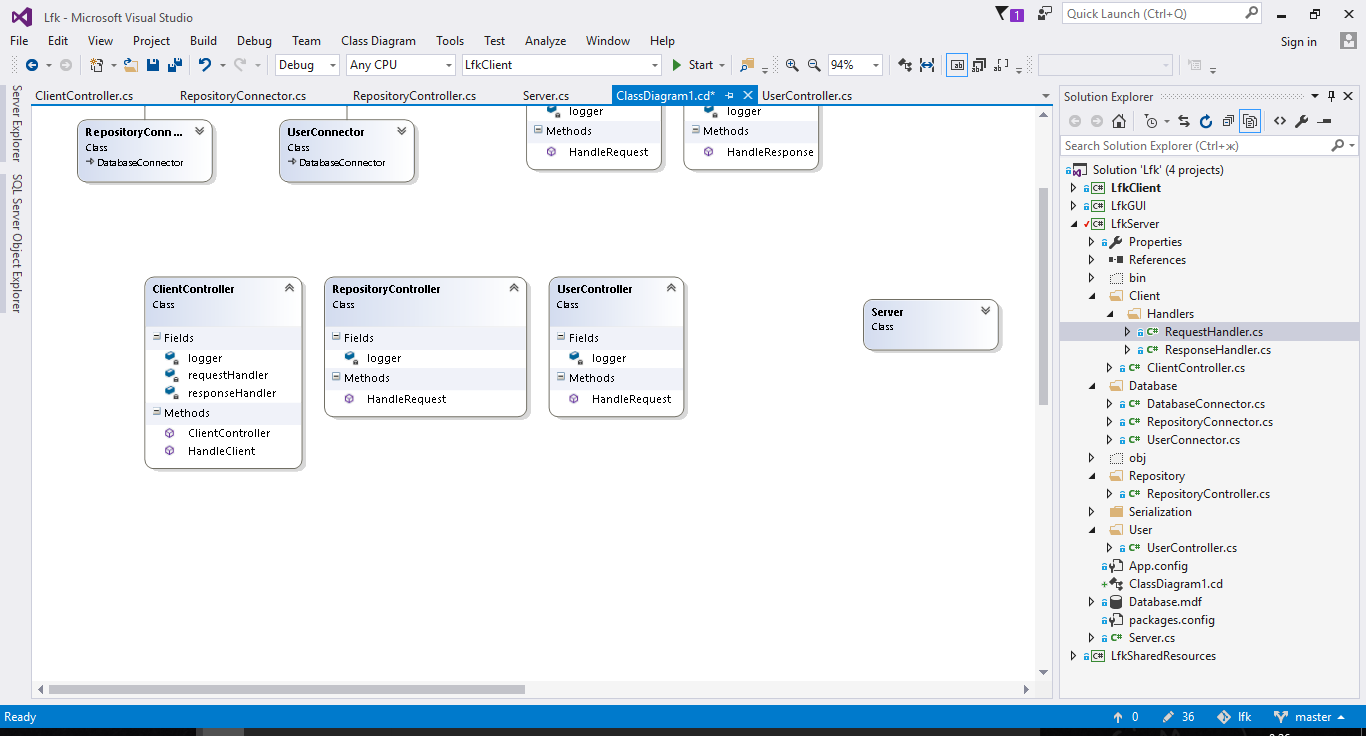
### Рисунок 2.5.1 – Схема работы модуля LfkServer

В течение работы сервера происходит ожидание клиентов (Server). Как только на сервер поступает запрос, он отправляется на обработку (HandleClinet), в этот же момент возобновляется ожидание клиентов, а полученный запрос перенаправляется к соответствующему обработчику, указанному в пункте назначения сетевого пакета (HandlerUserRequest, если пунктом назначения является работа с пользователями и HandleRepositoryRequest, если пунктом назначения является работа с репозиториями). Далее, данные, содержащиеся в сетевом пакете, отправляются на уровень ниже к обработчикам конкретных действий (авторизация, создание, удаление, обновление и так далее). Эти данные разбираются соответствующим обработчиком, который на их основе конструирует серверные модели, которые впоследствии отправляются к необходимым подключателям (UserConnector, RepositoryConnector), формирующих непосредственно запросы к базе данных. СУБД обрабатывает входной запрос и возвращает результаты своей работы подключателям, которые передают ответ всем вышестоящим слоям. В конечном итоге формируется сетевой пакет, содержащий ответ на клиентский запрос, который передаётся в вызывающий модуль LfkClient.

**Диаграммы, описание и реализация классов модуля**

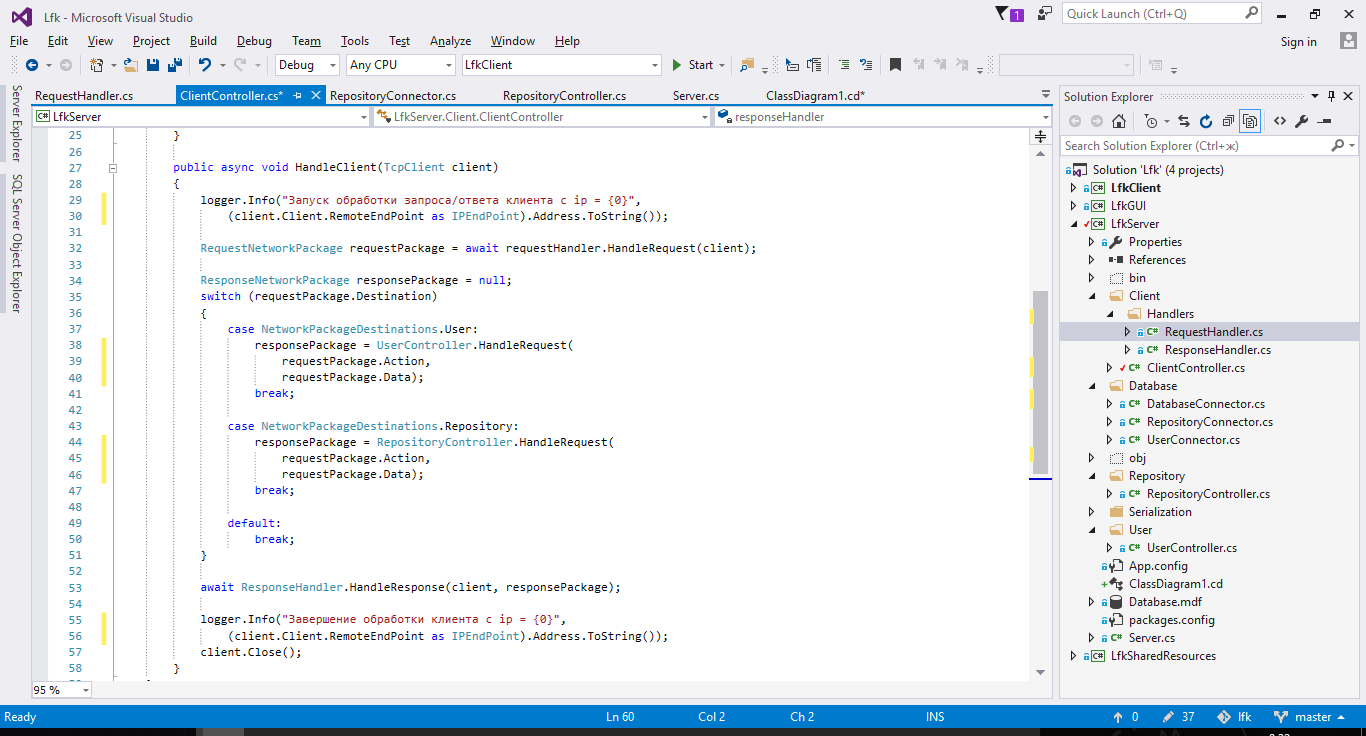
Главным достоинством сервера является его асинхронность, что позволяет принимать запросы следующего клиента сразу после получения предыдущего, не дожидаясь обработки последнего. Данная особенность реализована с помощью асинхронного программирования C# 5.0 на основе async / await методов. Помимо этого, для выявления ошибок в случае аварийного завершения работы сервера, используется тщательное логгирование работы каждого его компонента. Ведение журнала работы сервера реализовано с помощью библиотеки NLog и поддерживает все основные уровни логгирования.

Логика обработки запроса и формирования ответа заключена в классе ClientController, который опирается на работу классов RepositoryContoller и UserColtroller. Диаграмма данных классов представлена на рисунке 2.5.2.



### Рисунок 2.5.2 – Основные обработчики запросов

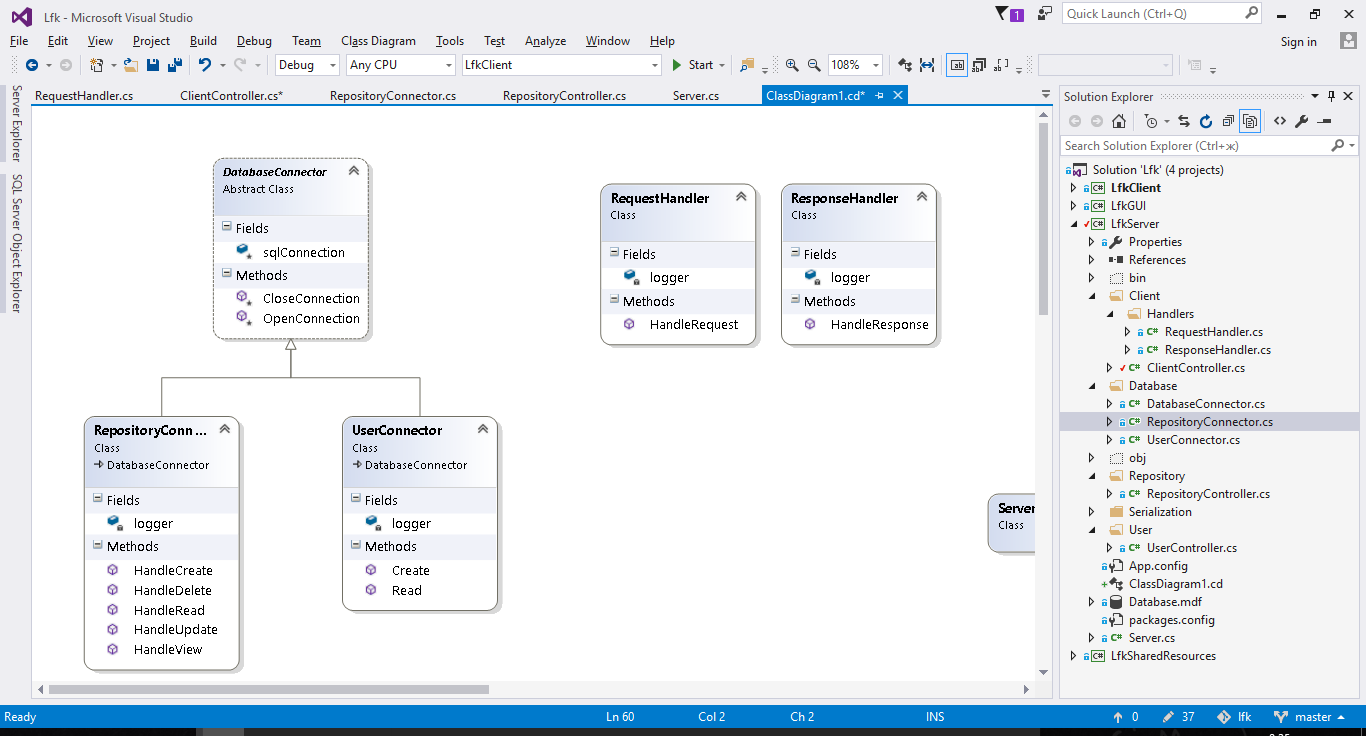
ClientController считывает содержимое сетевого пакета, определяет пункт назначения, запускает соответствующий обработчик, дожидается результата и формирует пакет с ответом. Реализация данных действий показана на рисунке 2.5.3.



### Рисунок 2.5.3 – Реализация обработки клиентского запроса

Классы-обработчики UserController и RepositoryController приводят входные данные к нужному виду и перенаправляют их к подключателям в соответствии с необходимым действием для непосредственного контакта с базой данных.

Подключатели UserConnector и RepositoryConnector базируются на абстрактном классе DatabaseConnector, хранящего в себе подключение к базе данных, а также методы его открытия и закрытия. Сами же UserConnector и RepositoryConnector имеют методы для обработки всех возможных действий в пределах соответствующего им пункта назначения. Эти методы производят трансформацию моделей и осуществляют запросы к базе данных. Диаграмма этих классов представлена на рисунке 2.5.4.

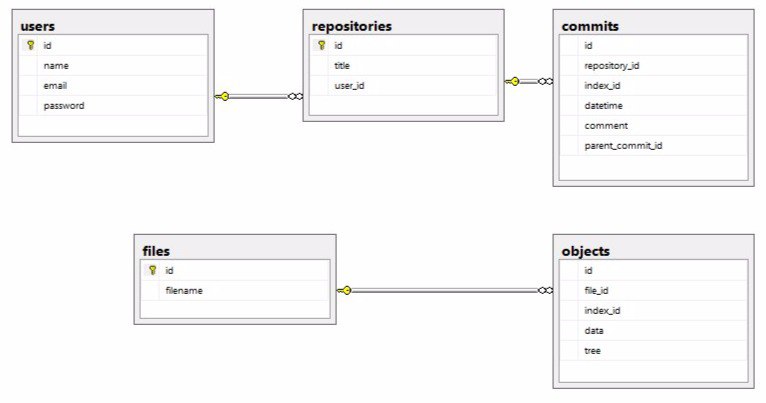


### Рисунок 2.5.4 – Диаграмма классов, реализующих взаимодействие с базой данных

Результат запроса к базе данных возвращается классам UserConnector и RepositoryConnector, которые отправляют его выше по стеку вызовов, где данный результат наращивается и превращается в полноценный сетевой пакет для отправки обратно по сети в классе ClientController.

**2.6 Проектирование базы данных**

База данных системы Lfk представлена 5 таблицами, диаграмма которых представлена на рисунке 2.6.1.



### Рисунок 2.6.1 – Диаграммы базы данных системы Lfk

Таблица users отвечает за хранение информации о зарегистрированных пользователях. Описание данной таблицы предоставлено в таблице 2.6.1.

Таблица 2.6.1 – Описание таблицы users

|  |  |
| --- | --- |
| Столбец | Описание |
| id | Уникальный идентификатор пользователя |
| name | Имя |
| e-mail | Почтовый ящик |
| password | Пароль |

Таблица repositories отвечает за хранение информации о созданных репозиториях. Описание данной таблицы предоставлено в таблице 2.6.2.

Таблица 2.6.2 – Описание таблицы repositories

|  |  |
| --- | --- |
| Столбец | Описание |
| id | Уникальный идентификатор репозитория |
| title | Название |
| user\_id | Уникальный идентификатор владельца репозитория |

Таблица commits отвечает за хранение информации о коммитах в определённом репозитории. Описание данной таблицы предоставлено в таблице 2.6.3.

Таблица 2.6.3 – Описание таблицы commits

|  |  |
| --- | --- |
| Столбец | Описание |
| id | Уникальный идентификатор коммита |
| repository\_id | Уникальный идентификатор соответствующего репозитория |
| index\_id | Уникальный идентификатор соответствующего индекса |
| datetime | Дата и время создания коммита |
| comment | Пользовательский комментарий |
| parent\_commit\_id | Уникальный идентификатор родительского коммита |

Таблица objects отвечает за хранение информации о блоб-объектах. Описание данной таблицы предоставлено в таблице 2.6.4.

Таблица 2.6.4 – Описание таблицы objects

|  |  |
| --- | --- |
| Столбец | Описание |
| id | Уникальный идентификатор блоб-объекта |
| file\_id | Уникальный идентификатор соответствующего файла |
| index\_id | Уникальный идентификатор соответствующего индекса |
| data | Сжатое содержимое данных |
| tree | Дерево Хаффмана |

Таблица files отвечает за хранение информации об отслеживаемых файлах. Описание данной таблицы предоставлено в таблице 2.6.5.

Таблица 2.6.5 – Описание таблицы files

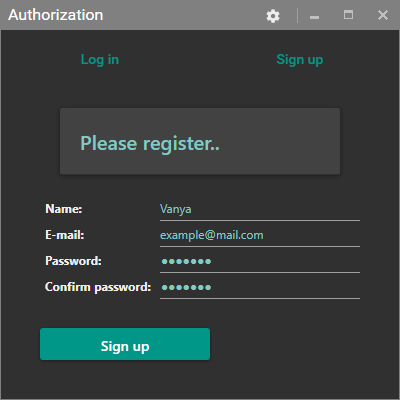
|  |  |
| --- | --- |
| Столбец | Описание |
| id | Уникальный идентификатор файла |
| filename | Имя файла |

Все вышеописанные таблицы приведены к нормализированным формам, наделены необходимыми ограничениями целостности и объединены между собою внешними связями.

# **Глава 3. Руководство пользователя**

**Авторизация**

При запуске приложения пользователю предоставляется возможность войти с помощью e-mail и пароля, либо зарегистрировать новую учетную запись. На рисунке 3.1 представлено окно регистрации нового пользователя.



### Рисунок 3.1 – Окно регистрации пользователя

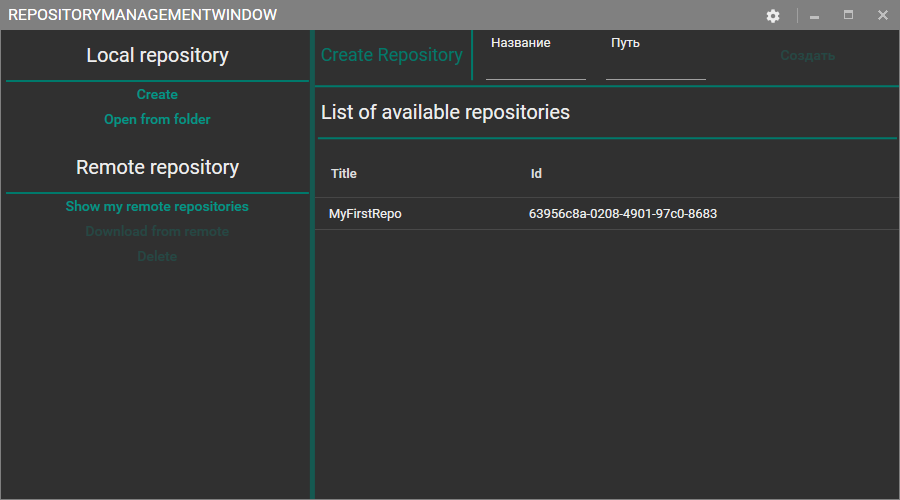
При регистрации новой учетной записи существует ряд правил, которым пользователь должен следовать:

1. Имя должно состоять только из латинских символов, включая верхний и нижний регистр. Минимальная длина имени 2 символа, максимальная 24.
2. E-mail адрес должен соответствовать принятым стандартам, что означает состоять из двух частей разделенных символом «@», причем в правой части в части доменного имени сервера должно быть минимум два символа.
3. Пароль должен содержать минимум один символ латиницы верхнего регистра, и одну или более цифр. Минимальная длина пароля 6 символа, максимальная - 24.

При отсутствии соединения с сервером, пользователь получит соответствующее сообщение, но уже не сможет использовать основной функционал приложения.

**Управление пользовательскими репозиториями**

В случае же успешной авторизации, пользователь перейдет к окну управления пользовательскими репозиториями. Все управляемые пользователем репозитории будут автоматически привязываться к текущему пользователю. На рисунке 3.2 изображено главное окно управления пользовательскими репозиториями.



### Рисунок 3.2 – Окно управления пользовательскими репозиториями

При открытии пользователь видит в левой части приложения два меню, одно из которых предназначено для работы с локальными репозиториями, а другое позволяет управлять репозиториями которые находятся на сервере.

**Создание репозитория**

Для создания репозитория пользователь должен развернуть меню создания репозитория нажатием на пункт меню Создать. Далее необходимо ввести название и путь к существующей локальной директории, в которой будет располагаться репозиторий. Путь можно прописать в поле для ввода вручную, либо двойным щелчком левой клавиши мыши открыть диалоговое окно выбора директории.

При попытке создания репозитория в папке уже содержащей репозиторий, пользователю представиться выбор перезаписать репозиторий, либо выбрать другой каталог для создания репозитория.

**Открытие существующего репозитория**

Для открытия репозитория пользователь должен выбрать каталог, содержащий инициализационные файлы. В случае отсутствия в каталоге системных файлов приложение не может открыть каталог и возвращает управление окну управления пользовательскими репозиториями.

**Скачивание удаленного репозитория**

Для скачивания репозитория с сервера необходимо выбрать репозиторий из списка, и нажать кнопку скачать, после, приложение предложит пользователю выбрать каталог для размещения репозитория. В случае успешного скачивания, пользователь может отрыть данный репозиторий и начать с ним работать.

**Удаление репозитория**

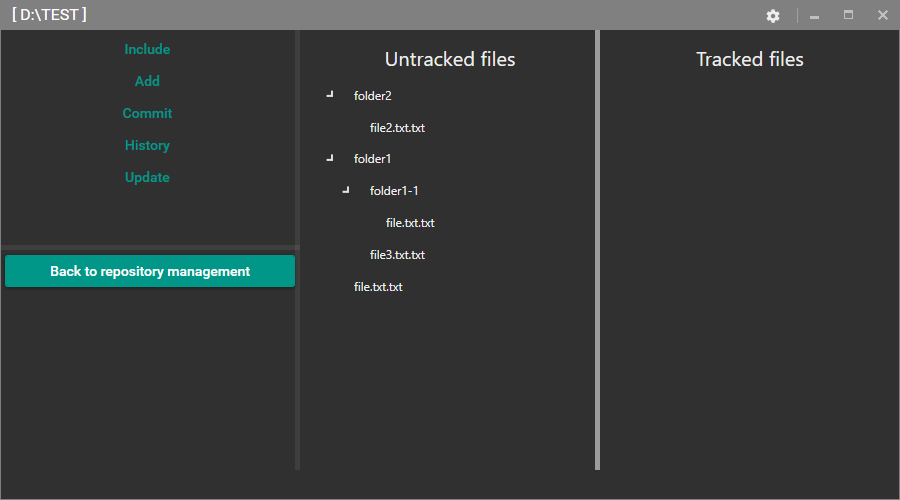
Для удаления репозитория с сервера необходимо выбрать репозиторий из списка, и нажать кнопку удалить, после чего на сервере уничтожатся все данные связанные с этим репозиторием. Удалить локальный репозиторий можно просто удалив системную папку, созданную в рабочем каталоге.

**Работа с репозиторием**

При работе с репозиторием пользовательские файлы могу находится в одном из нескольких состояний:

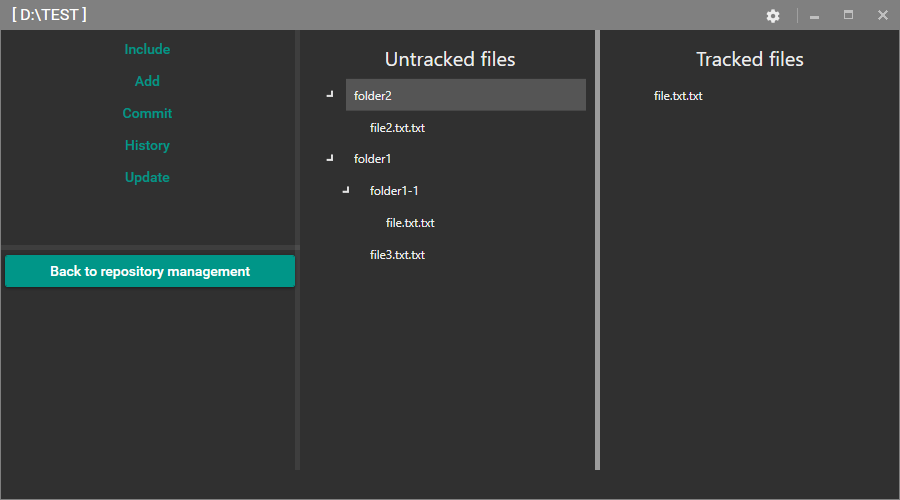
* Не отслеживаемые файлы – файлы, которые являются новыми для системы контроля версий;
* Отслеживаемые файлы – файлы, изменения которых отслеживаются системой контроля версий;
* Добавленные для включения в следующий коммит файлы.

Изначально при создании репозитория, все пользовательские файлы являются не отслеживаемыми. Для включения файлов в список отслеживаемых необходимо использовать команду Include. Рабочая директория на странице команды Include представлена в виде иерархического дерева файлов и директорий. На рисунке 3.3 представлена страница включения файлов в список отслеживаемых.



### Рисунок 3.3 – Страница команды Include

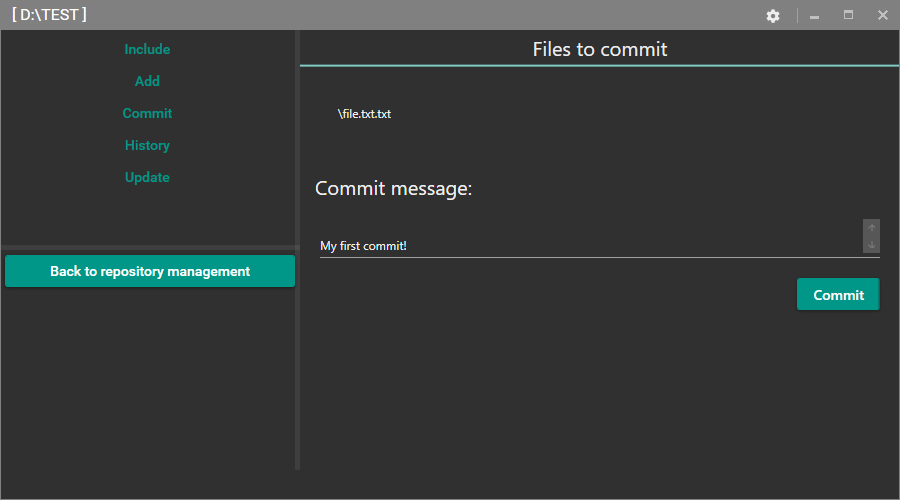
Добавление файлов в список отслеживаемых производится перетаскиванием любого узла в область страницы соответствующую отслеживаемым файлам. На рисунке 3.4 отображена страница команды Include после включения файла file.txt.txt.



### Рисунок 3.4 – Страница команды Include

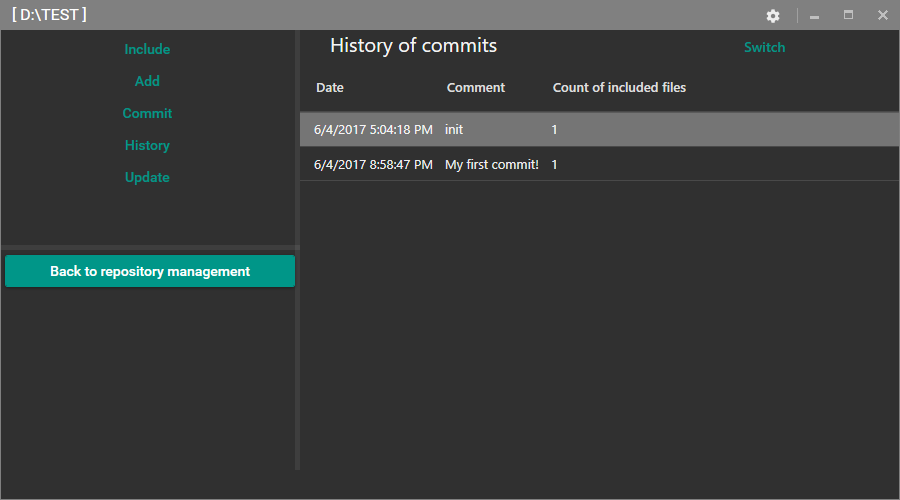
Для отмены отслеживания файла, необходимо перетащить его обратно в область не отслеживаемых файлов. Аналогичным образом производится выполнение команды Add.

Создание коммита происходит на вкладке Commit, которая изображена на рисунке 3.5. При создании коммита пользователю выводится список файлов, которые войдут в текущий коммит, также необходимо ввести сообщение коммита, отражающее суть произведенных изменений.



### Рисунок 3.5 – Страница команды Commit

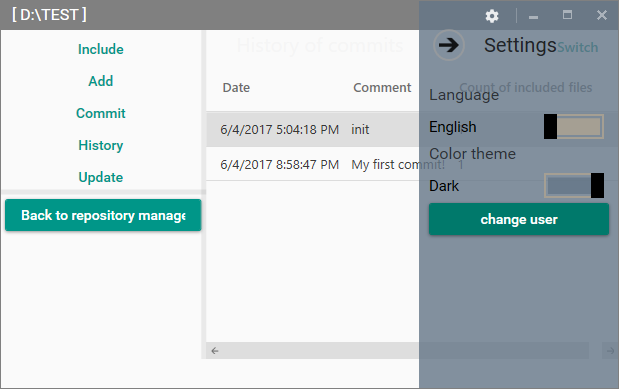
Вкладка History позволяет пользователю просматривать историю коммитов. Также на вкладке History присутствует кнопка, позволяющая переключиться на любой коммит, в результате чего изменится состояние пользовательского рабочего каталога. На рисунке 3.6 изображена страница комманды History.



### Рисунок 3.6 – Страница команды History

Команда Update позволяет загрузить текущее состояние репозитория на сервер. Выполнение команды Update возможно лишь в случае если в репозитории создан хотя бы один коммит.

Также в приложении Lfk присутствует общее для всех окон меню настроек приложения, в котором можно переключить настройки стиля, и язык приложения. Данное меню продемонстрировано на рисунке 3.7.



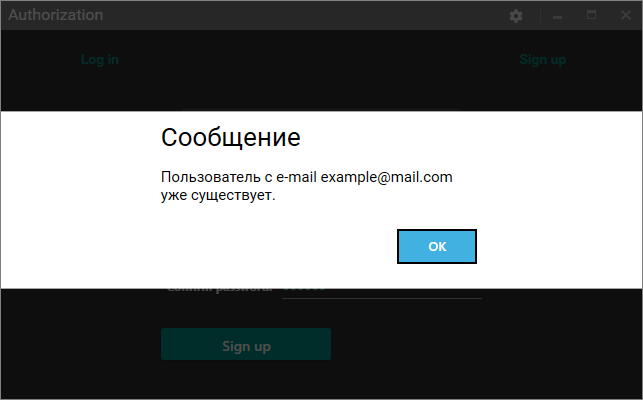
### Рисунок 3.7 – Меню приложения Lfk

Кнопка change user позволяет сменить пользователя, что реализовано путем перенаправления пользователя на страницу авторизации.

# **Глава 4. Тестирование**

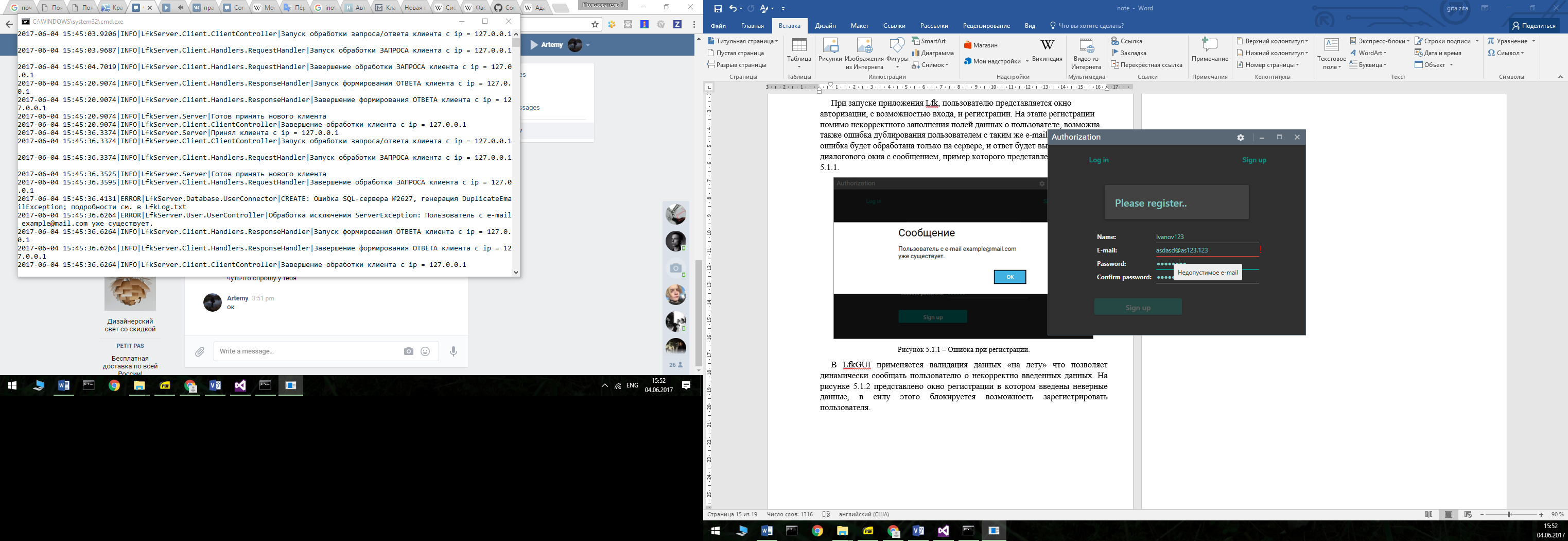
## **4.1 Авторизация**

При запуске приложения Lfk, пользователю представляется окно авторизации, с возможностью входа, и регистрации. На этапе регистрации помимо некорректного заполнения полей данных о пользователе, возможна также ошибка дублирования пользователем с таким же e-mail. Данная ошибка будет обработана только на сервере, и ответ будет выведен в виде диалогового окна с сообщением, пример которого представлен на рисунке 4.1.1.



### Рисунок 4.1.1 – Ошибка при регистрации

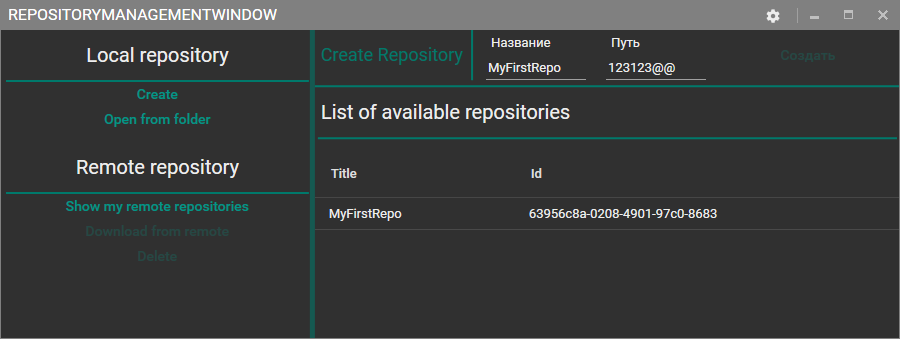
В LfkGUI применяется валидация данных «на лету» что позволяет динамически сообщать пользователю о некорректно введенных данных. На рисунке 4.1.2 представлено окно регистрации в котором введены неверные данные, в силу этого блокируется возможность зарегистрировать пользователя.



### Рисунок 4.1.2 – Введение некорректных данных

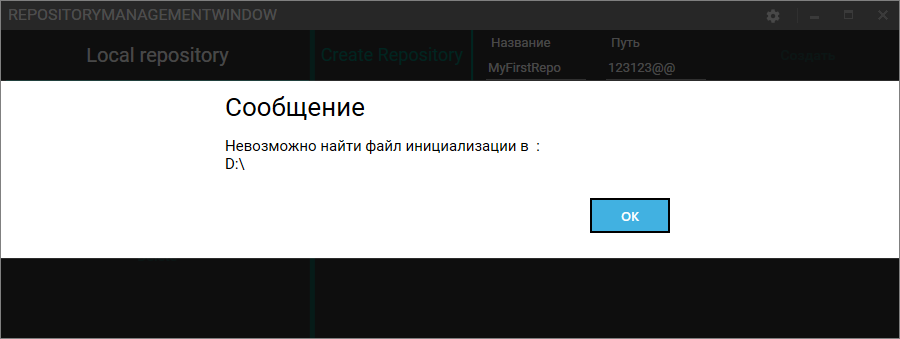
## **4.2 Управление пользовательскими репозиториями**

Управление пользовательскими репозиториями осуществляется с помощью команд: открыть, скачать, показать все. Как видно на рисунке 4.2.1 возможность создать репозиторий будет блокирована, до тех пор, пока пользователь не введет корректные данные, также возможность удаления и скачивания блокируется пока пользователь не выберет репозиторий из списка.



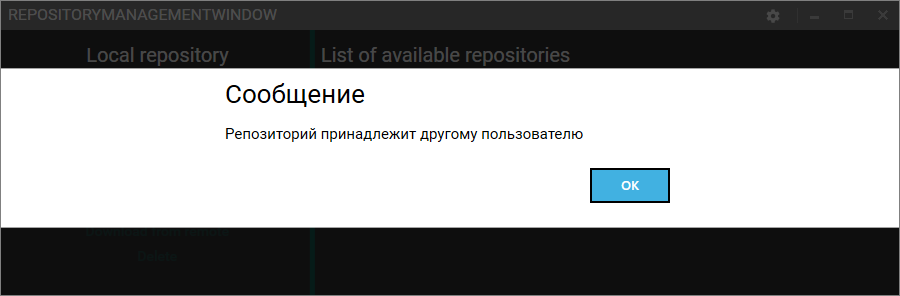
### Рисунок 4.2.1 – Ввод некорректных данных

При попытке открыть репозиторий из папки которая не содержит скрытого каталога с системными и идентификационными файлами также будет выдано соответствующее сообщение, что показано на рисунке 4.2.2.



### Рисунок 4.2.2 – Попытка открыть репозиторий из каталога, не содержащего репозиторий

Также при попытке открыть репозиторий другого пользователя будет выведено сообщение о том, что репозиторий недоступен, данное сообщение представлено на рисунке 4.2.3.



### Рисунок 4.2.3 – Попытка открыть чужой репозиторий.

## **4.3 Контроль версий рабочей директории пользователя**

## 4.4 Сервер

# **Заключение**

# **Список литераторы**