# Реферат

# Введение

# Постановка задачи

## Обзор и анализ прототипов

## Актуальность решаемой задачи

# 3. Архитектура проекта

## 3.1 Общая структура

## 3.2 Модуль LfkSharedResources

Реализация : NetworkPackageController, Huffman Encoding

## **3.2 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 3.2.1.

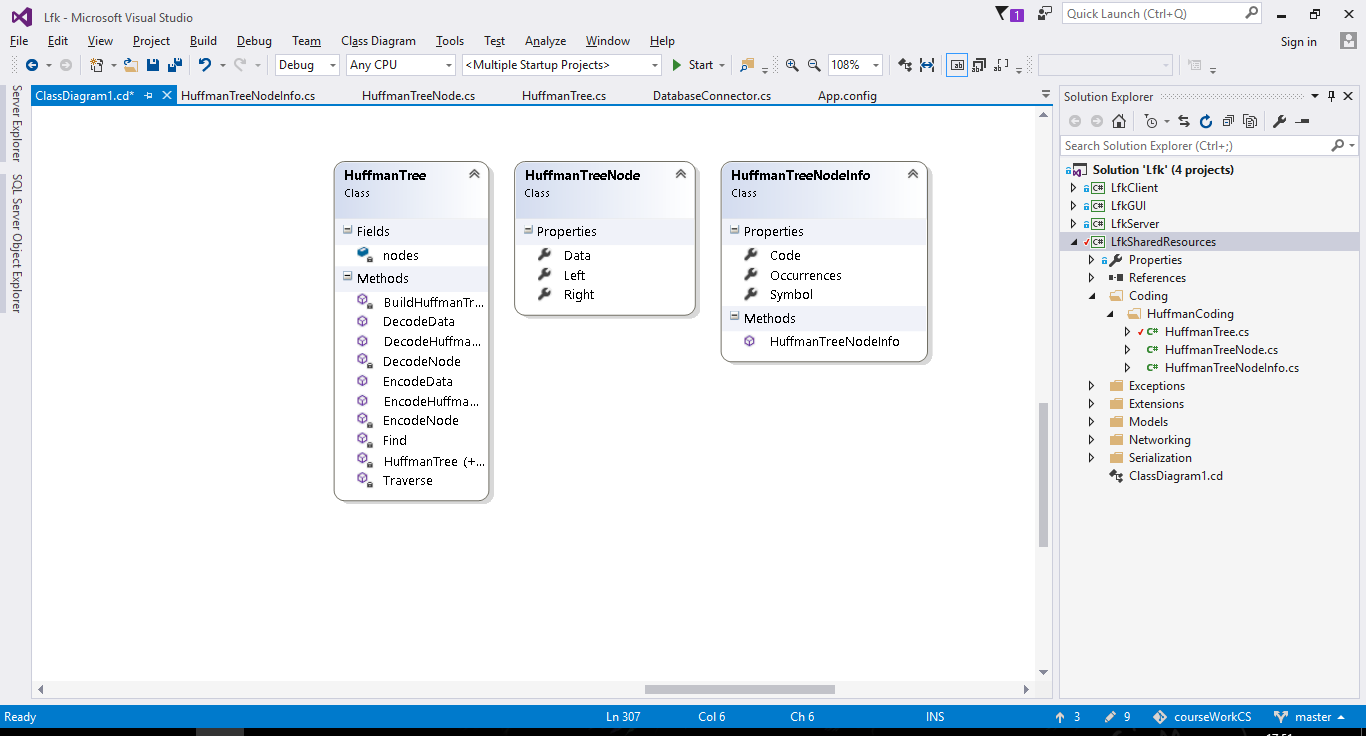


Рисунок 3.2.1 – Диаграмма классов, реализующих алгоритм Хаффмана

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 3.2.2.

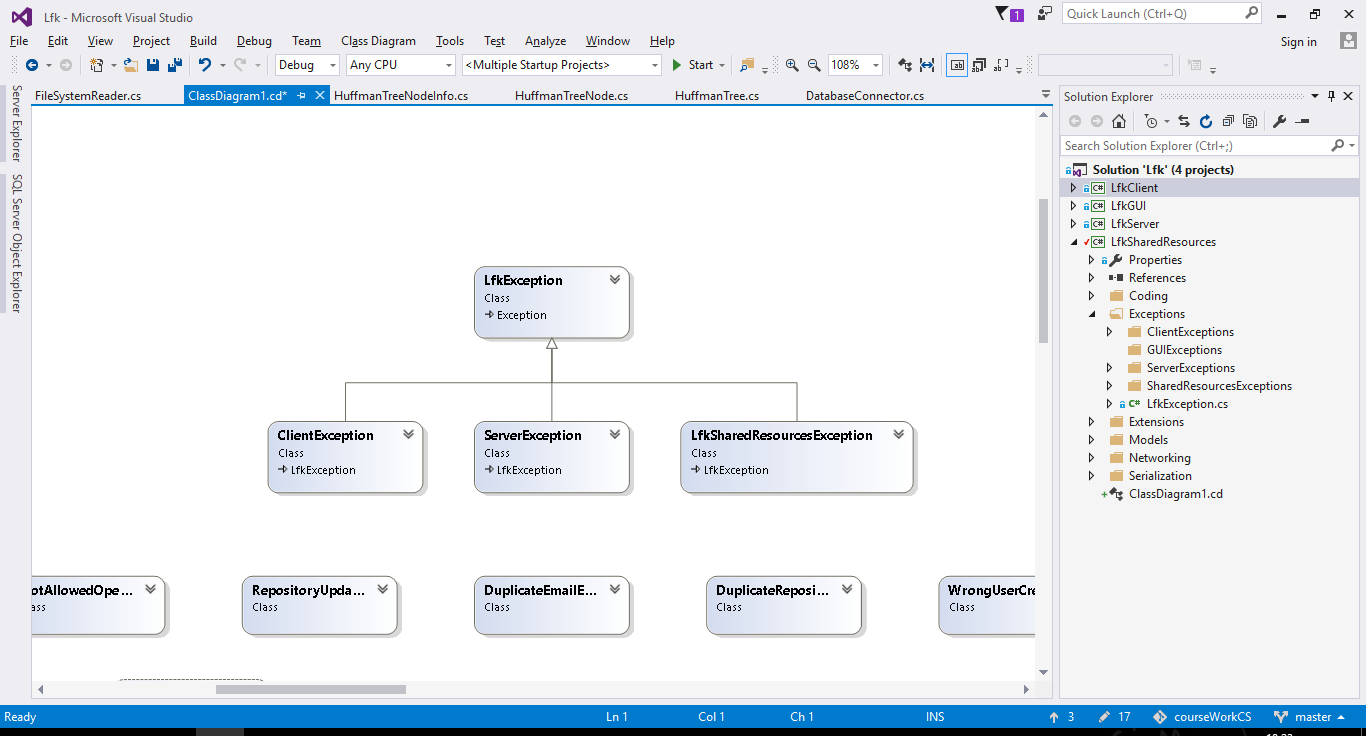


Рисунок 3.2.2 – Общая диаграмма исключений системы Lfk

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 3.2.1.

Таблица 3.2.1 – Описание исключений-потомков ClientException

|  |  |
| --- | --- |
| Исключение | Описание |
| ServerConnectionException |  |
| FolderAlreadyContainsRepositoryException |  |
| NotAllowedOpenRepositoryException |  |
| RepositoryUpdateWithoutCommitsException |  |

Информация об исключениях, являющихся потомками ServerException, представлена в таблице 3.2.2.

Таблица 3.2.2 – Описание исключений-потомков ServerException

|  |  |
| --- | --- |
| Исключение | Описание |
| DuplicateRepositoryTitleException |  |
| WrongUserCredentialsException |  |
| DuplicateEmailException |  |

Информация об исключениях, являющихся потомками SharedResourcesException, представлена в таблице 3.2.3.

Таблица 3.2.3 – Описание исключений-потомков ServerException

|  |  |
| --- | --- |
| Исключение | Описание |
| JsonSerializationException |  |
| JsonSerializerInvalidDataException |  |
| JsonSerializerNullArgumentException |  |

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 3.2.2.

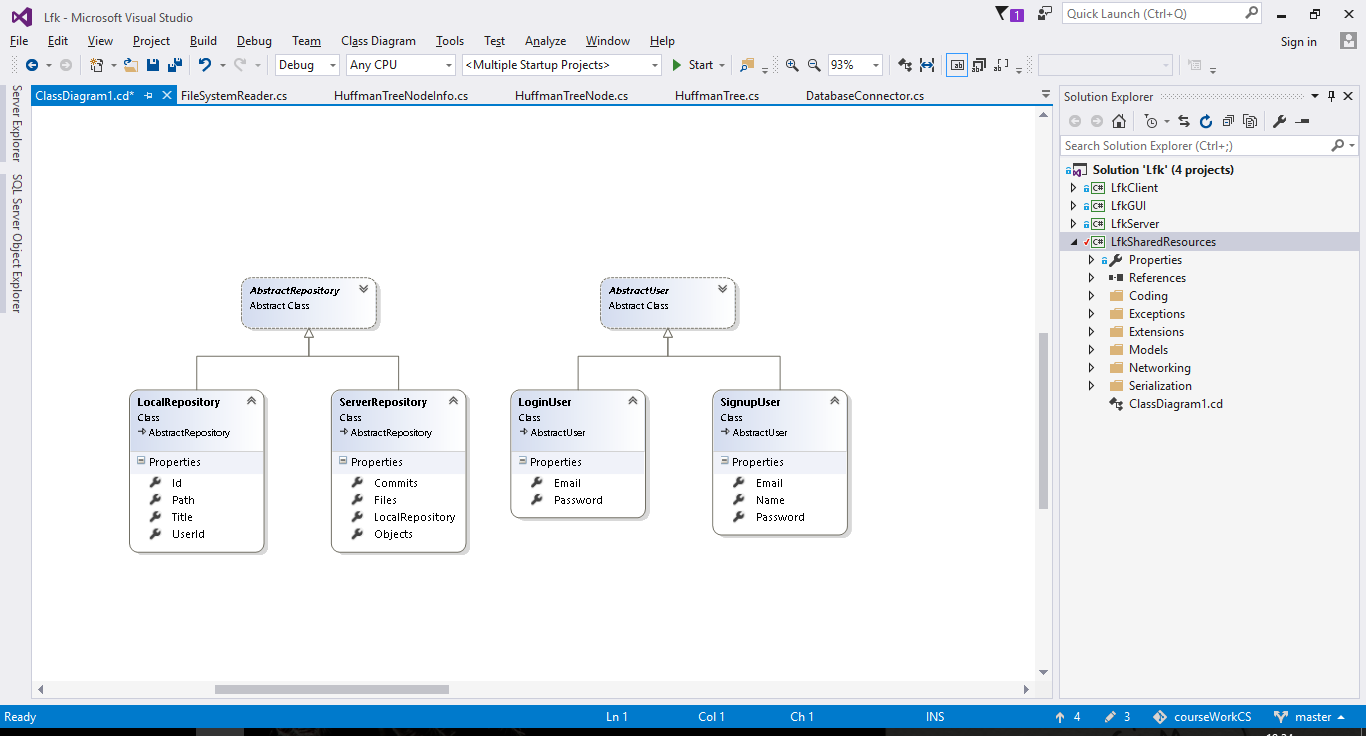


Рисунок 3.2.2 – Основополагающие модели системы

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 3.2.3.

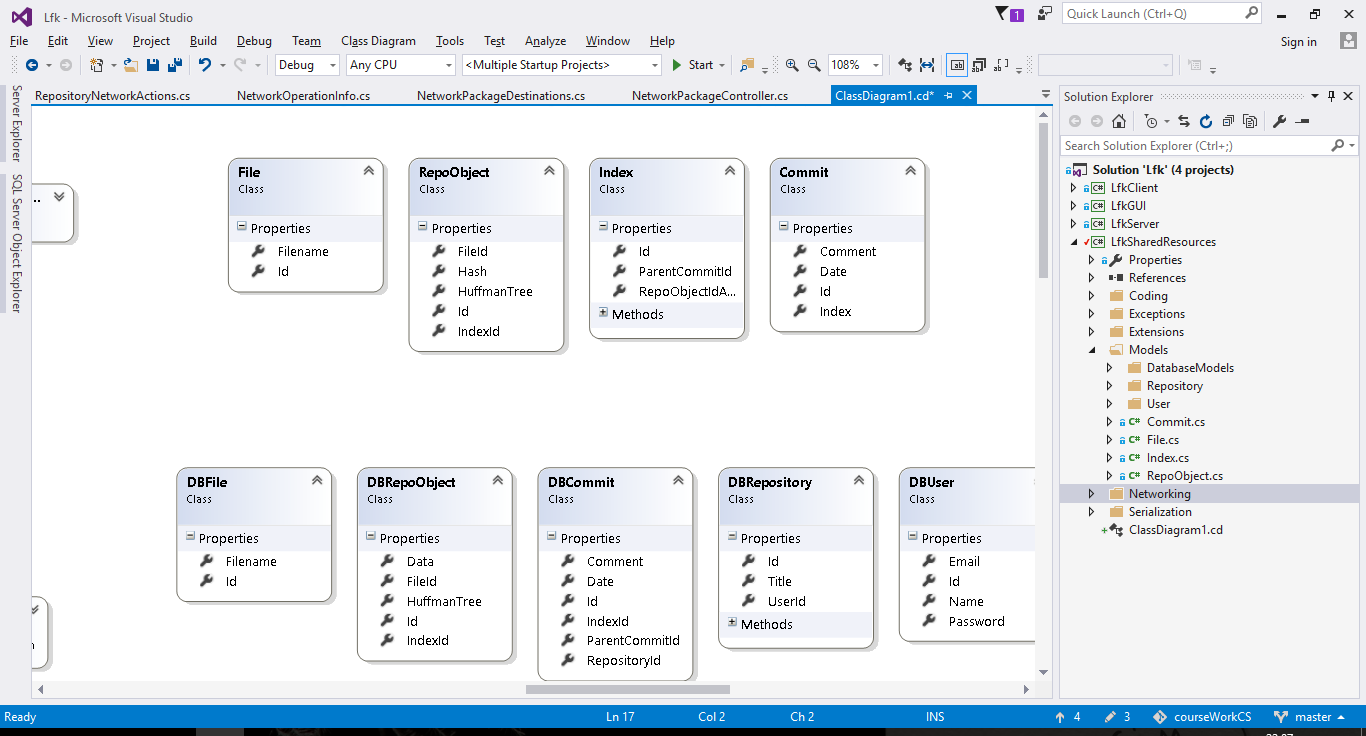


Рисунок 3.2.3 – Диаграмма классов, представляющих модели, используемые в течение всего цикла работы системы Lfk

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 3.2.4.

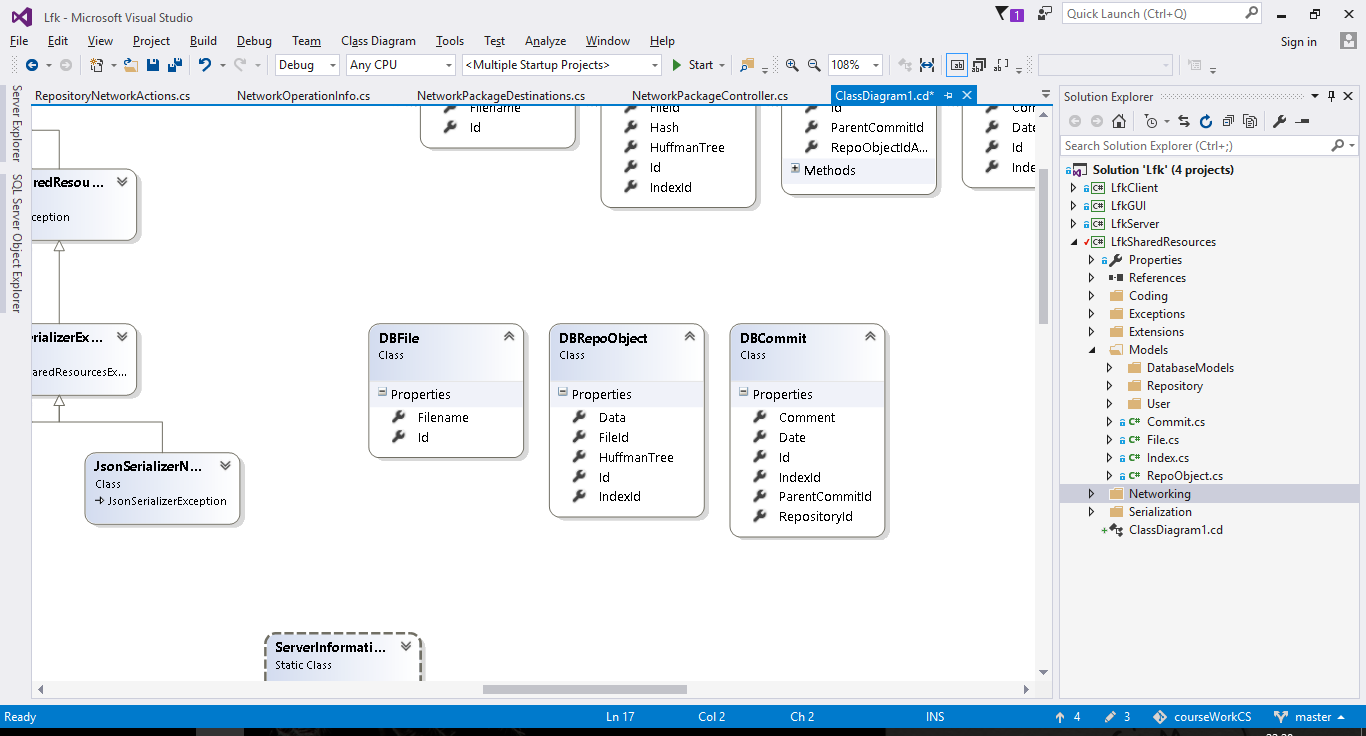


Рисунок 3.2.4 - Диаграмма классов, представляющих модели, используемые для представления содержимого репозитория на стороне сервера

Данные модели существуют в виде свойств класса ServerRepository, представленного на рисунке 3.2.2. 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 3.2.3.

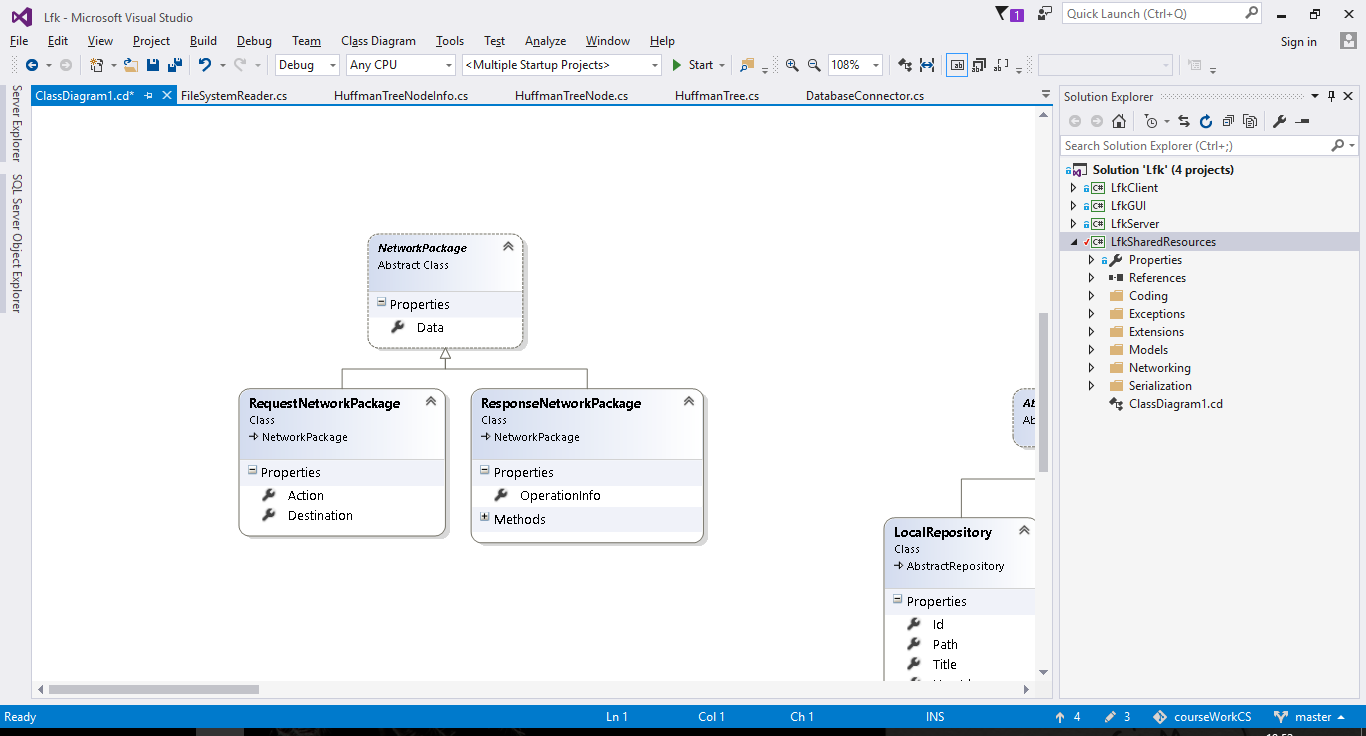


Рисунок 3.2.3 – Диаграмма классов, представляющих сетевые пакеты

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 3.2.4.

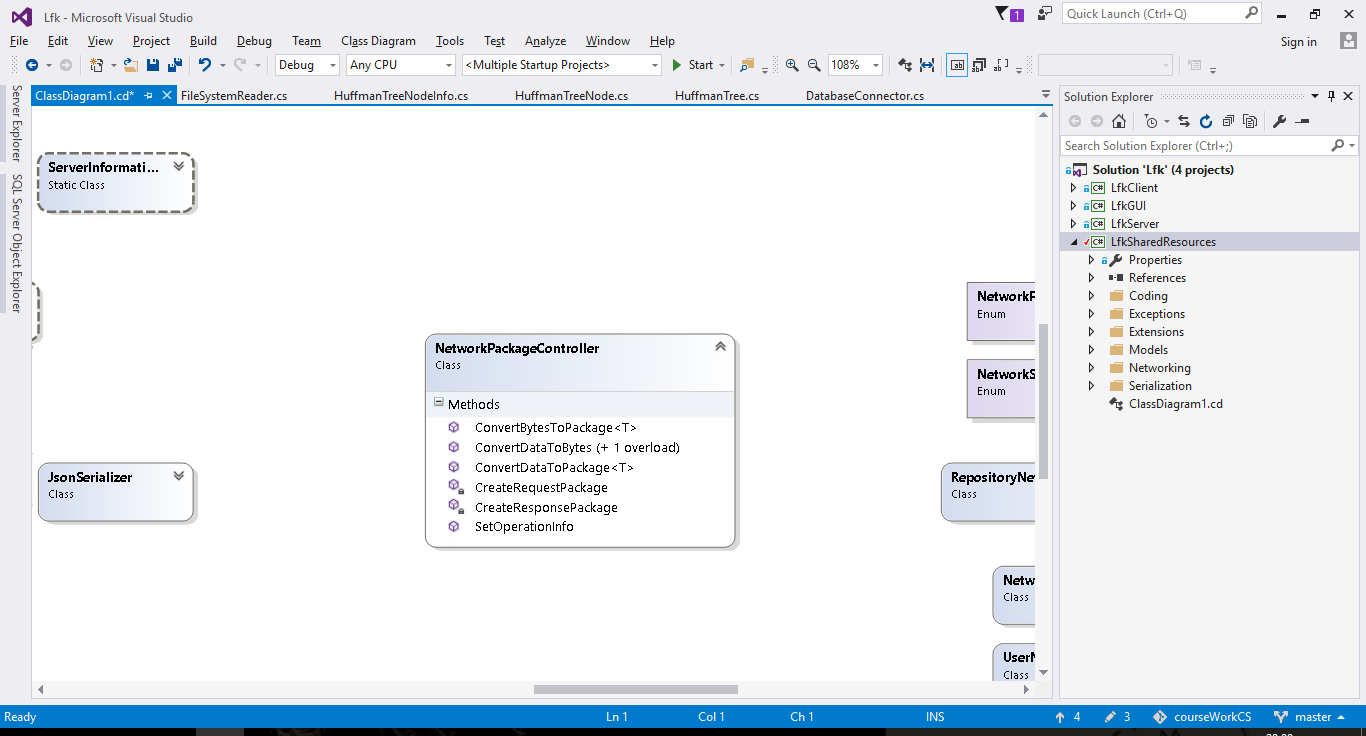


Рисунок 3.2.4 – Структура класса NetworkPackageController

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 3.2.5.

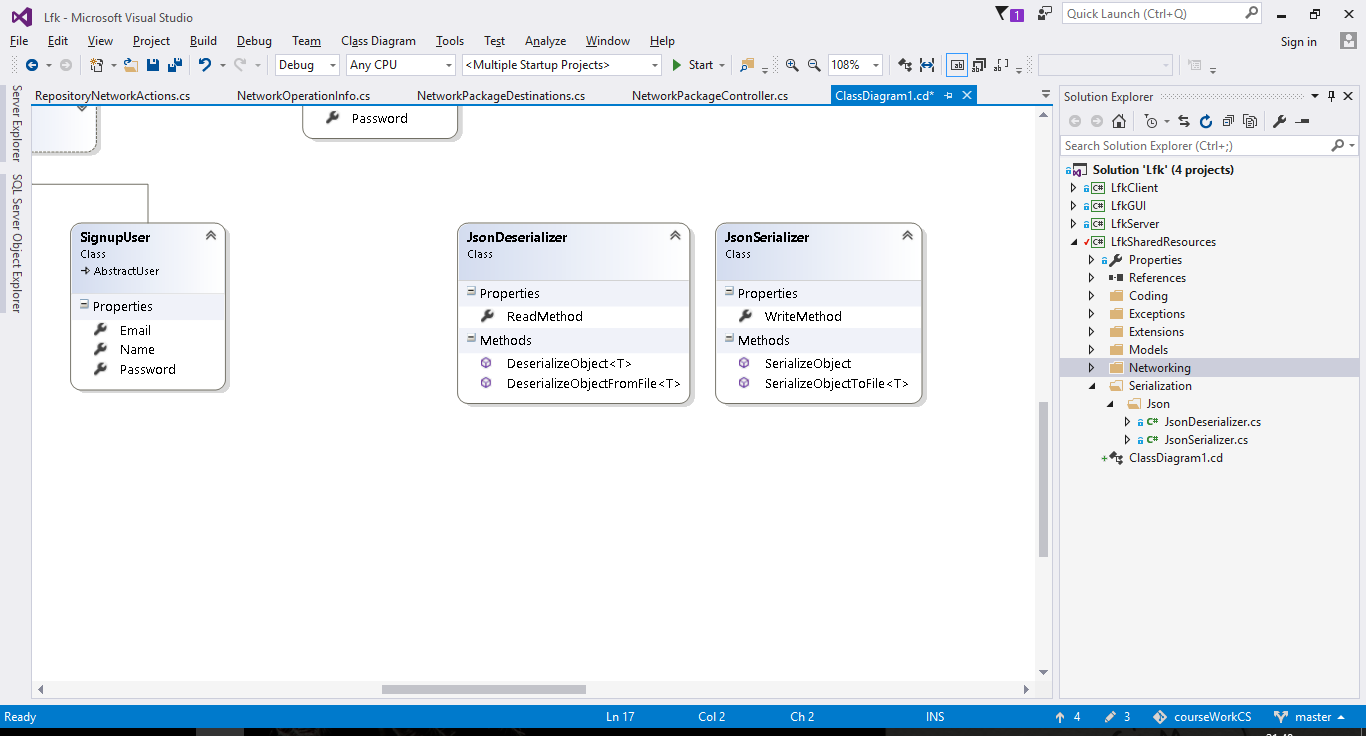


Рисунок 3.2.5 – Классы для десериализации и сериализации

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.

Вынести в раздел о БД

|  |
| --- |
| В типичном варианте работы программы, пользователь производит авторизацию, выбирает или создаёт репозиторий, а затем с помощью него осуществляет контроль за своей рабочей директорией. После сеанса работы, пользователь производит выгрузку локального репозитория на сервер. |

## **3.3 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 3.3.2.1 представлена схема взаимодействия компонентов модуля LfkGUI.



Рисунок 3.3.1 – 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 3.3.2 представлена диаграмма классов, отвечающих за представление моделей(View). Связанные с ними классы моделей представлений показаны далее на рисунке 3.3.5.

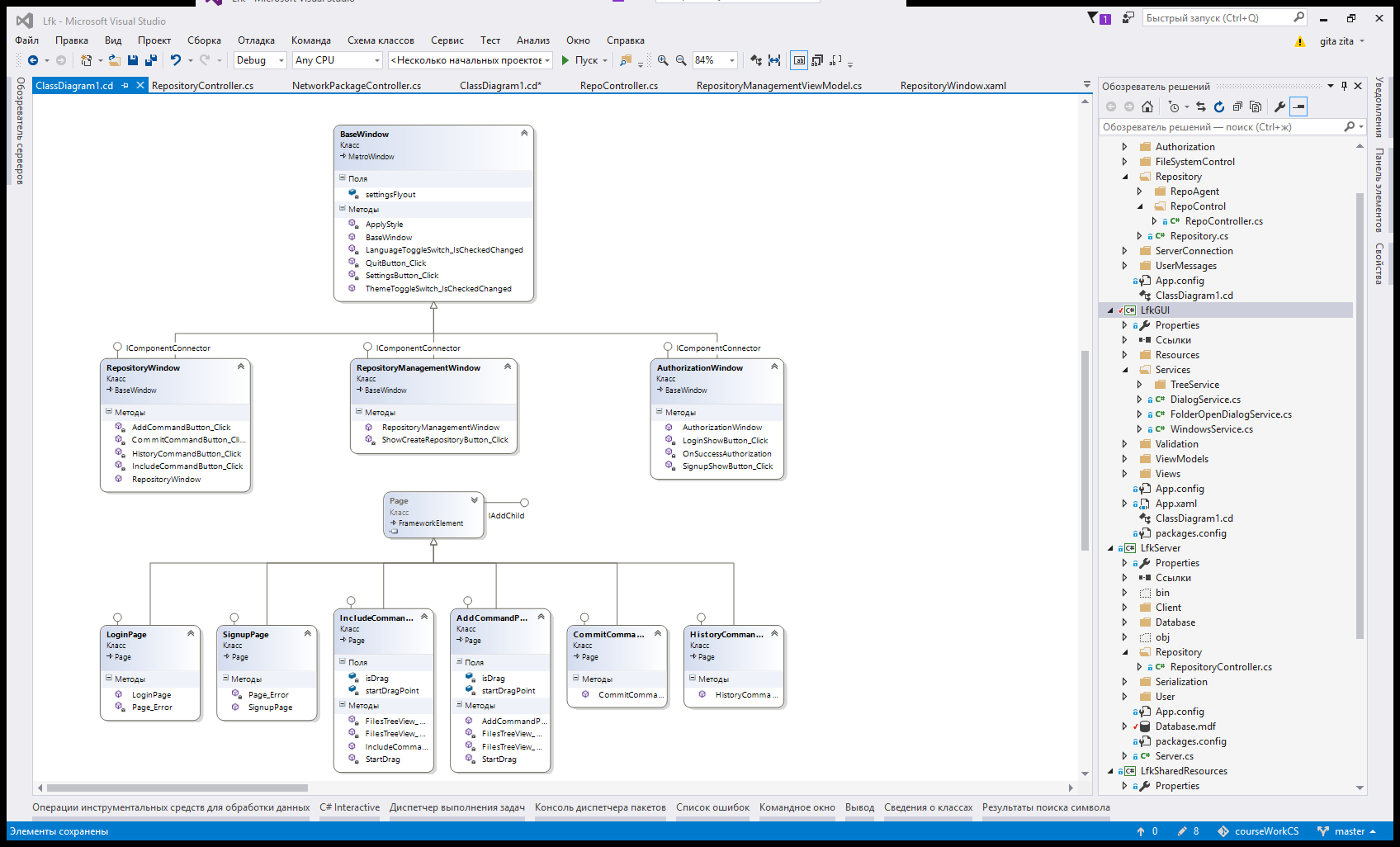


Рисунок 3.3.2 – Диаграмма классов представлений

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 3.3.3 представлена структура этих классов.

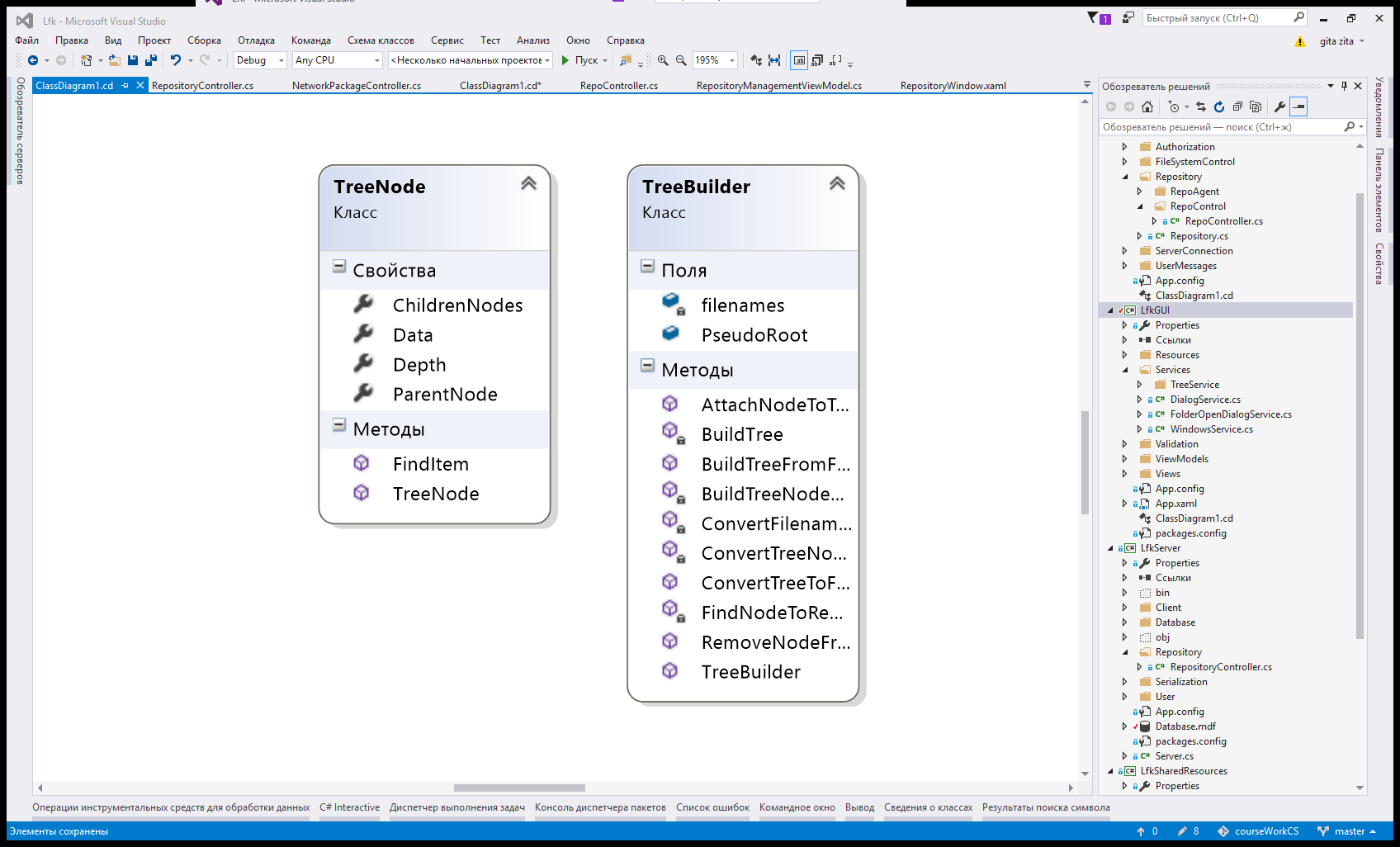


Рисунок 3.3.3 – Диаграмма классов для представления пользовательской директории

Механизмы построения, добавления, и удаления элементов, реализованы на основе рекурсивных алгоритмов обхода дерева. На рисунке 3.3.4 представлен код рекурсивной функции, которая позволяет преобразовать узел в список путей файлов.

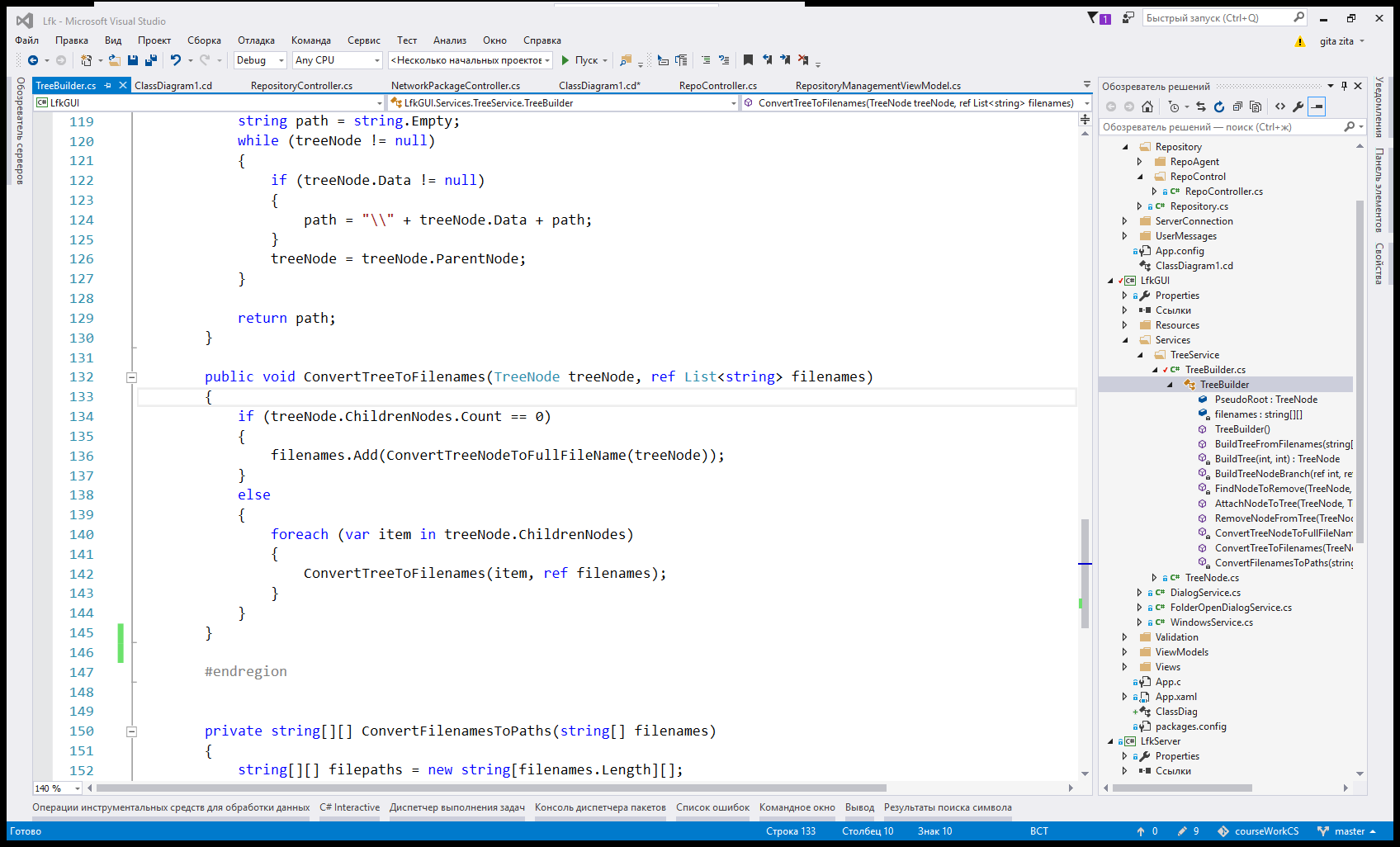


Рисунок 3.3.4 – Функция построения путей файлов из узла дерева

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.

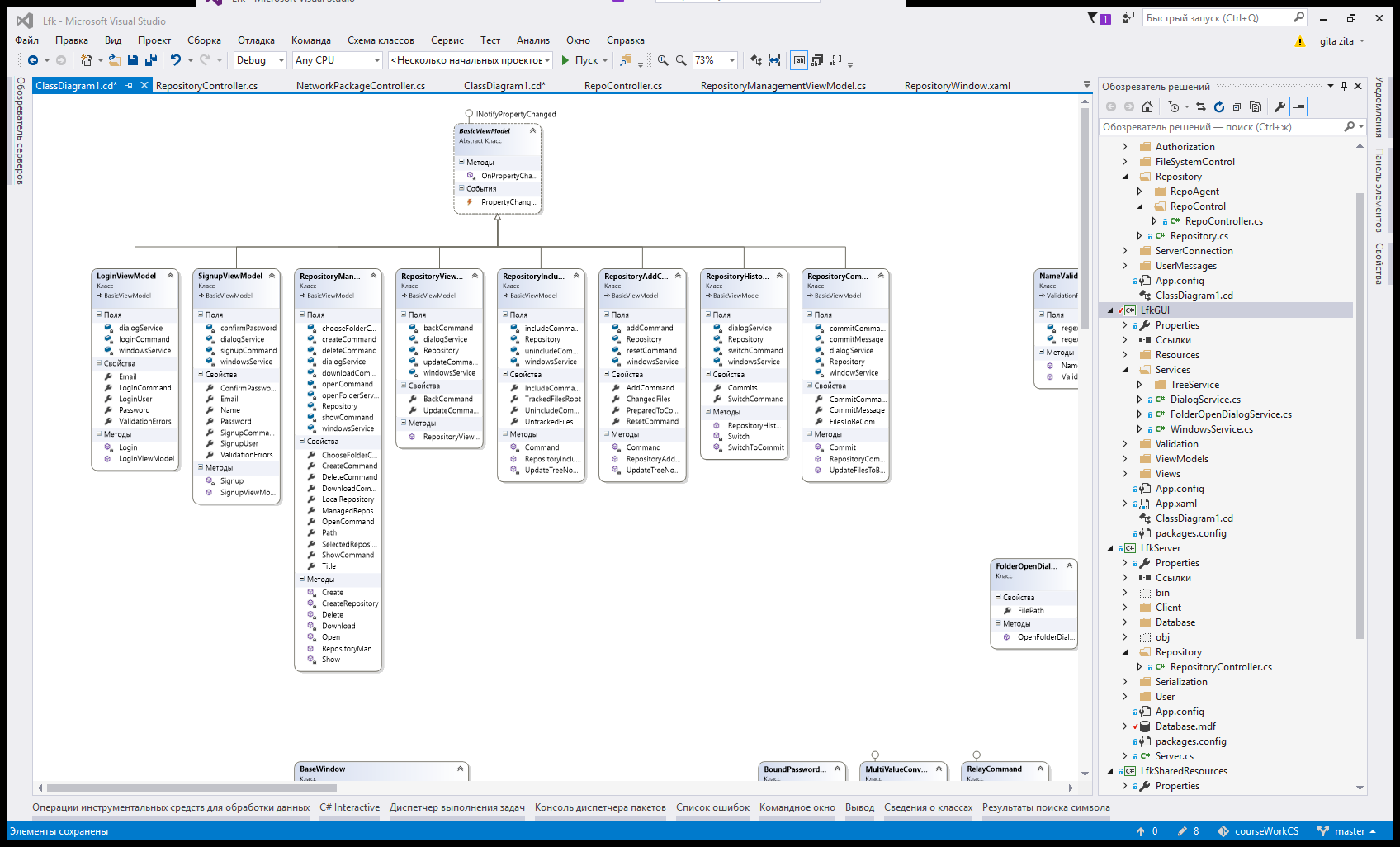


Рисунок 3.3.5 – Диаграмма классов моделей представления

На рисунке 3.3.6 представлен код реализации модели представления LoginViewModel, который отражает основные принципы построения ViewModel в проекте LfkGUI.

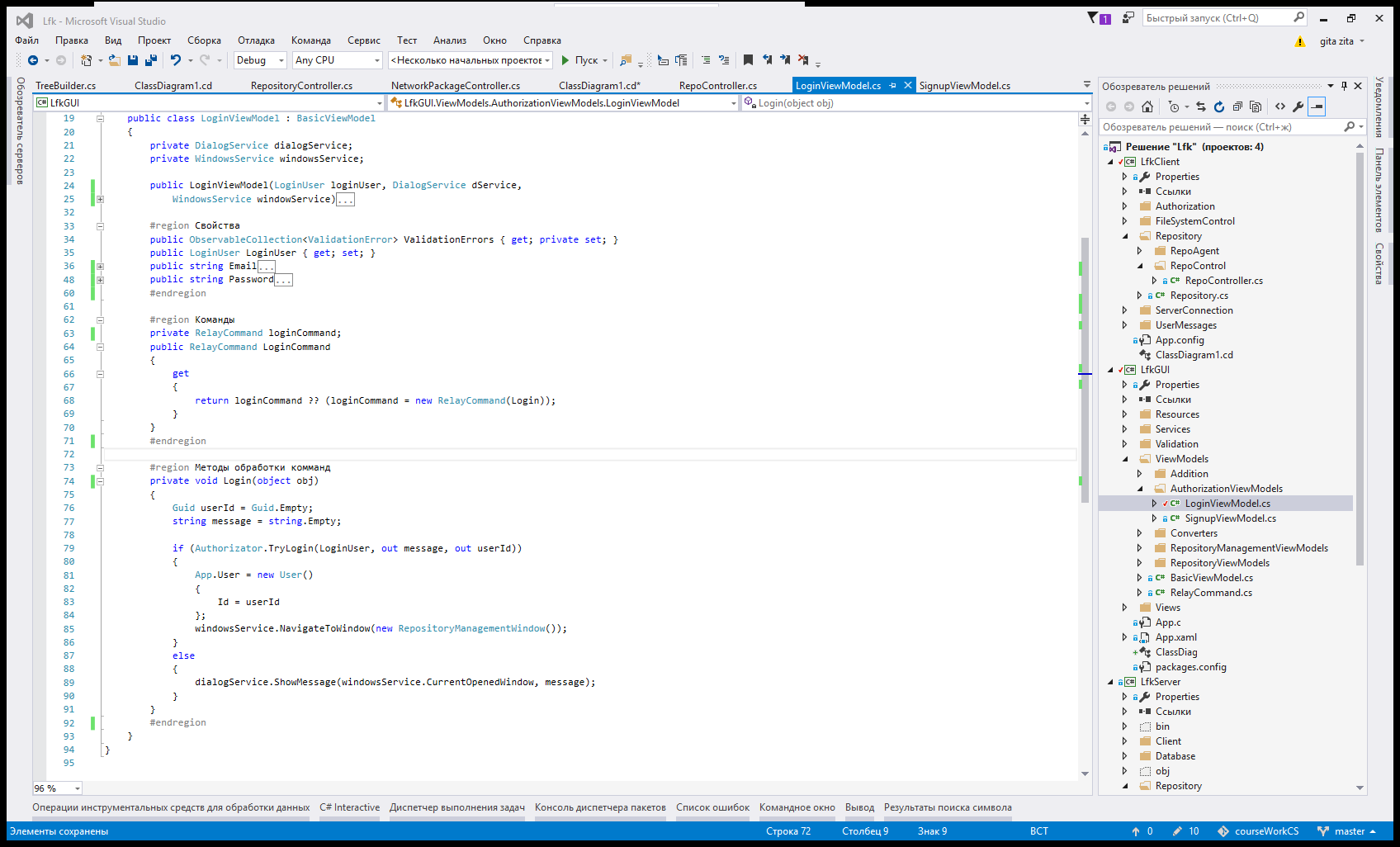


Рисунок 3.3.6 – Код реализации модели представления

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.

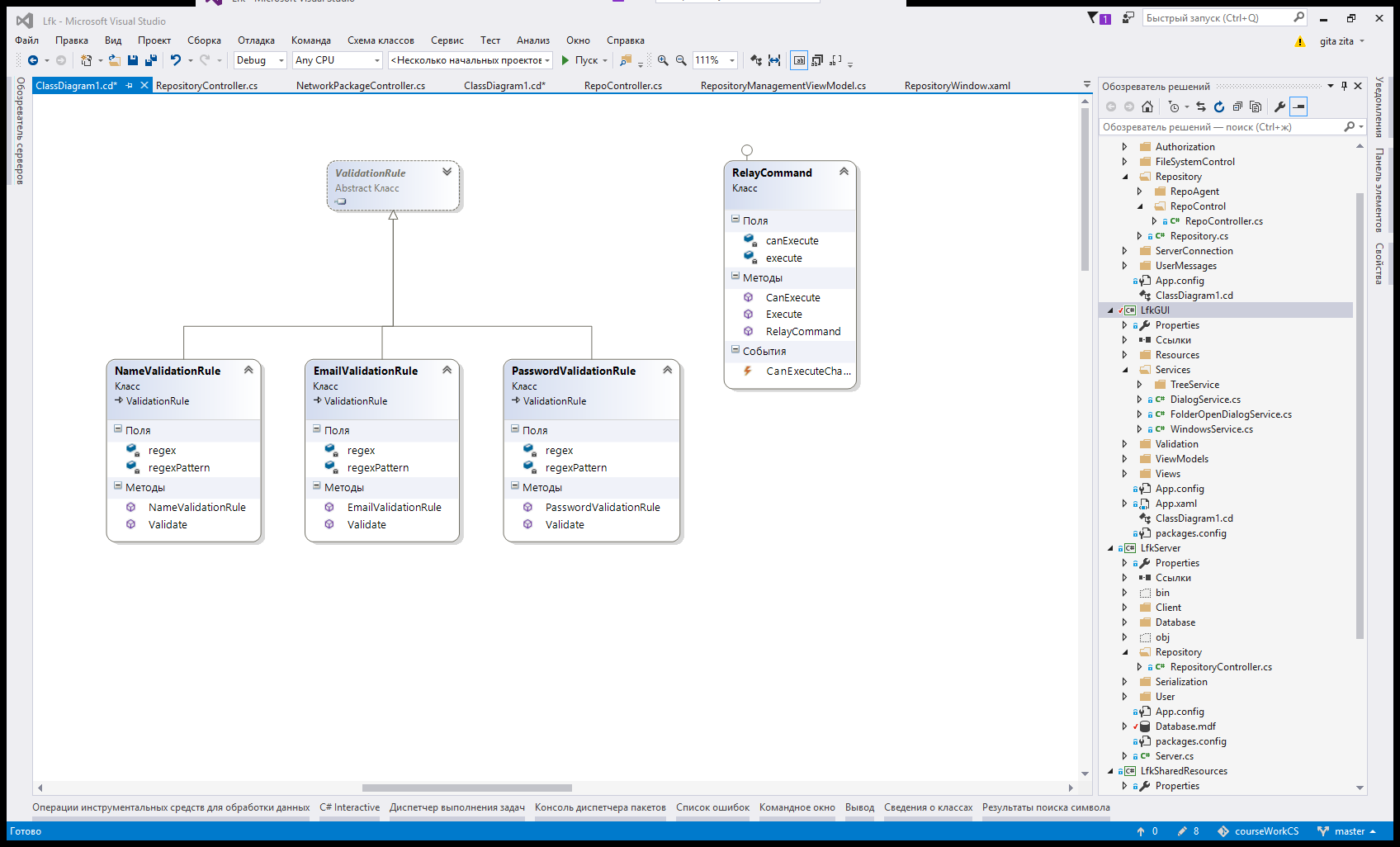


Рисунок 3.3.7 – Диаграмма классов валидации и класса команды

Программная реализация одного из валидационного правила представлена на рисунке 3.3.8

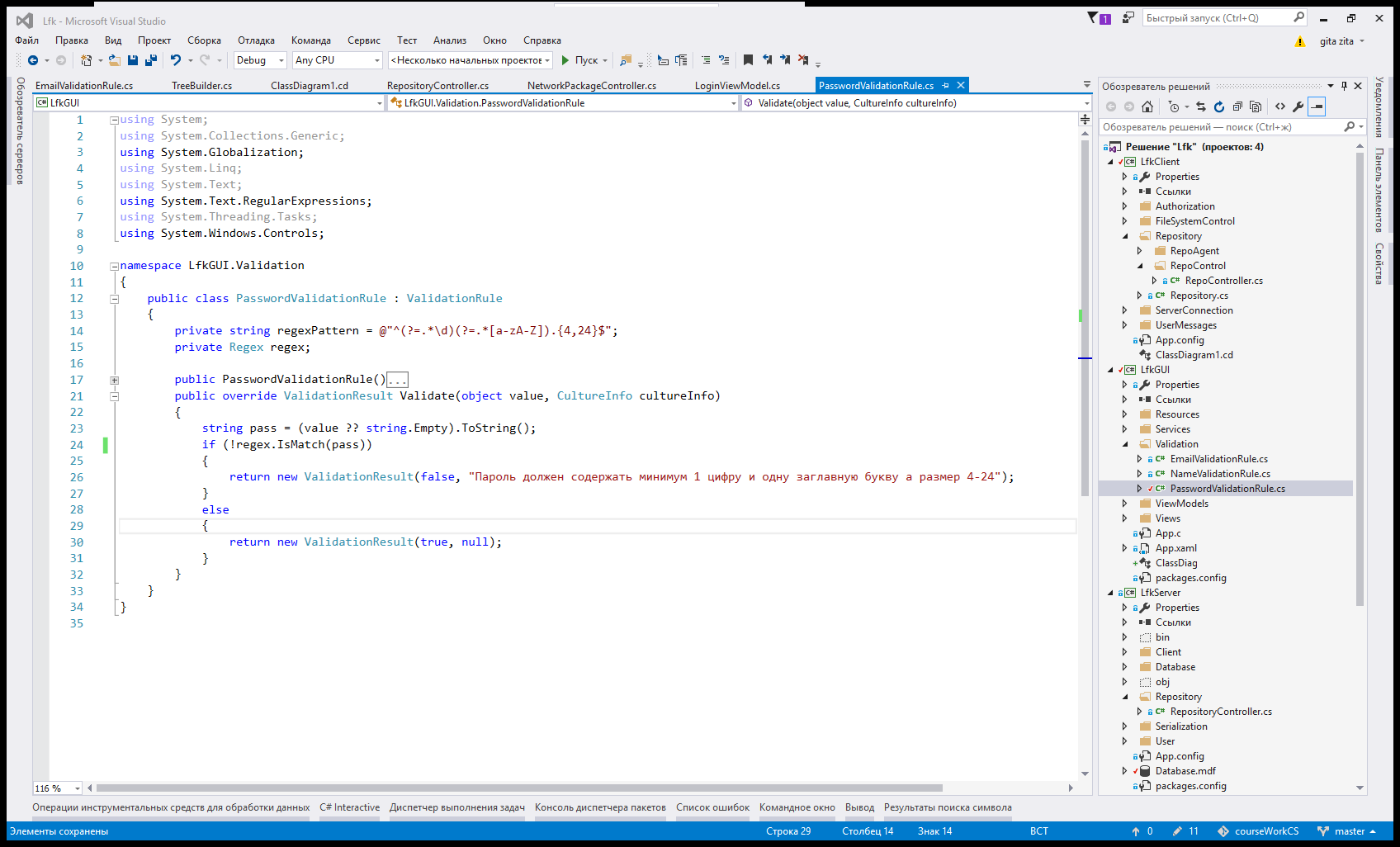


Рисунок 3.3.8 – Код реализации класса отвечающего за валидацию

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.

## **3.4 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 3.4.1 представлена схема взаимодействия модуля LfkClient.



Рисунок 3.4.1 – 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 3.4.2 изображена диаграмма основных классов.

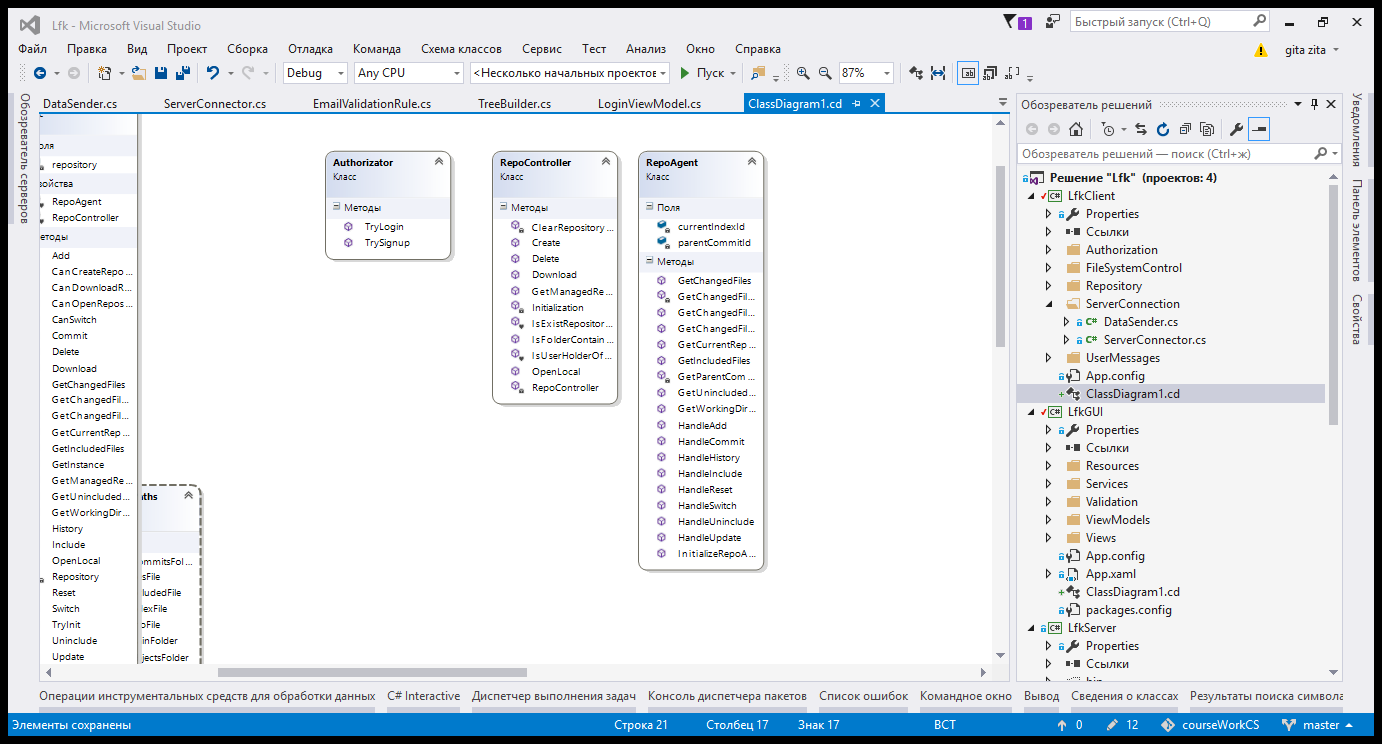


Рисунок 3.4.2 – Диаграмма основных классов модуля

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 3.4.3 приведен пример функции создания пользовательского репозитория.

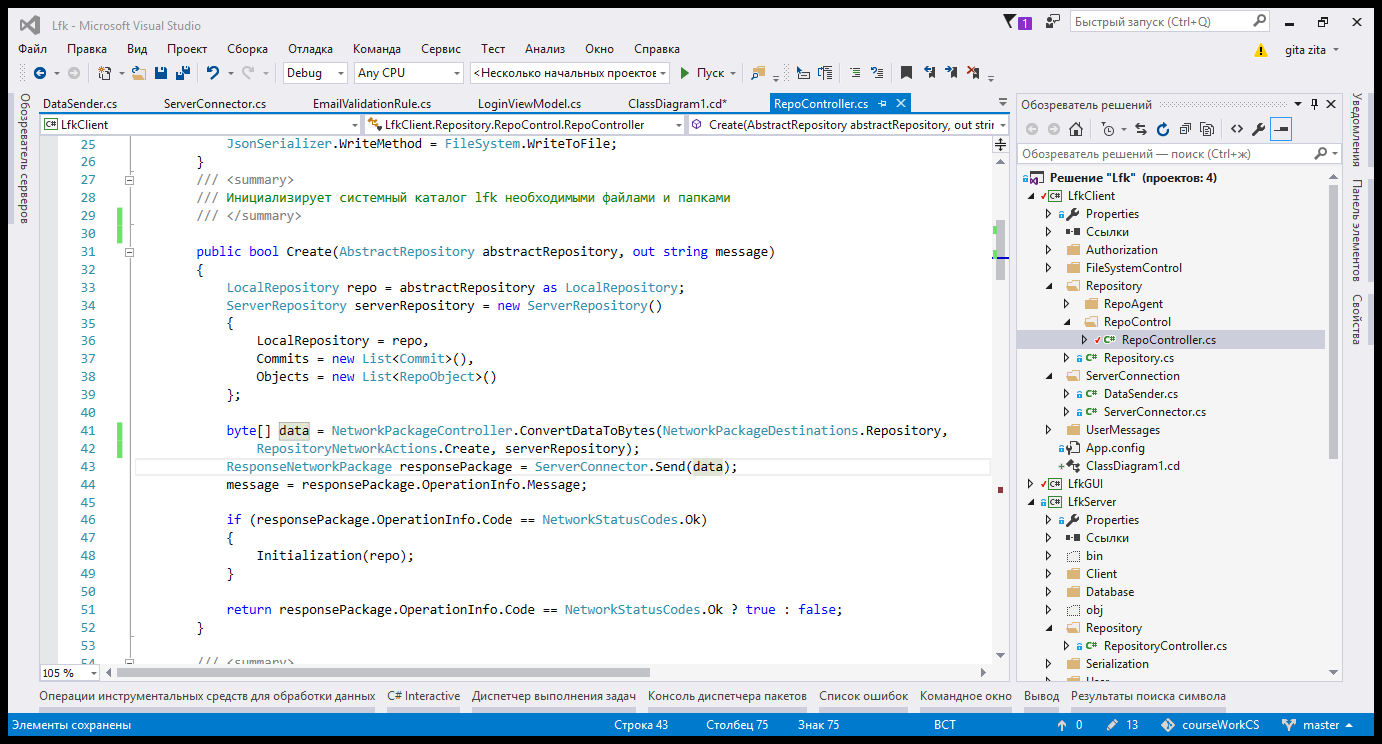


Рисунок 3.4.3 – Код реализации функции создания репозитория

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 3.4.4 приведен код реализации функции switch.

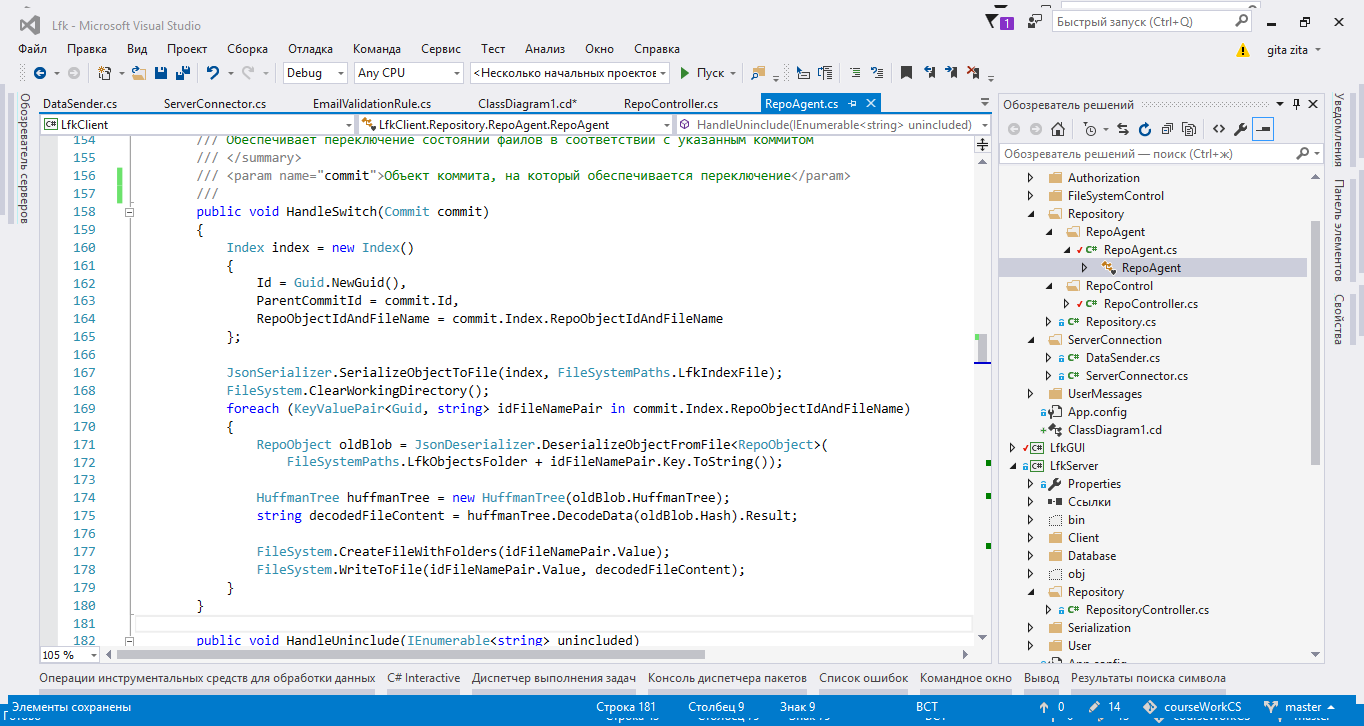


Рисунок 3.4.4 – Программная реализация команды switch.

LfkClient также включает в себя ряд вспомогательных классов, диаграмма которых представлена на рисунке 3.4.5.

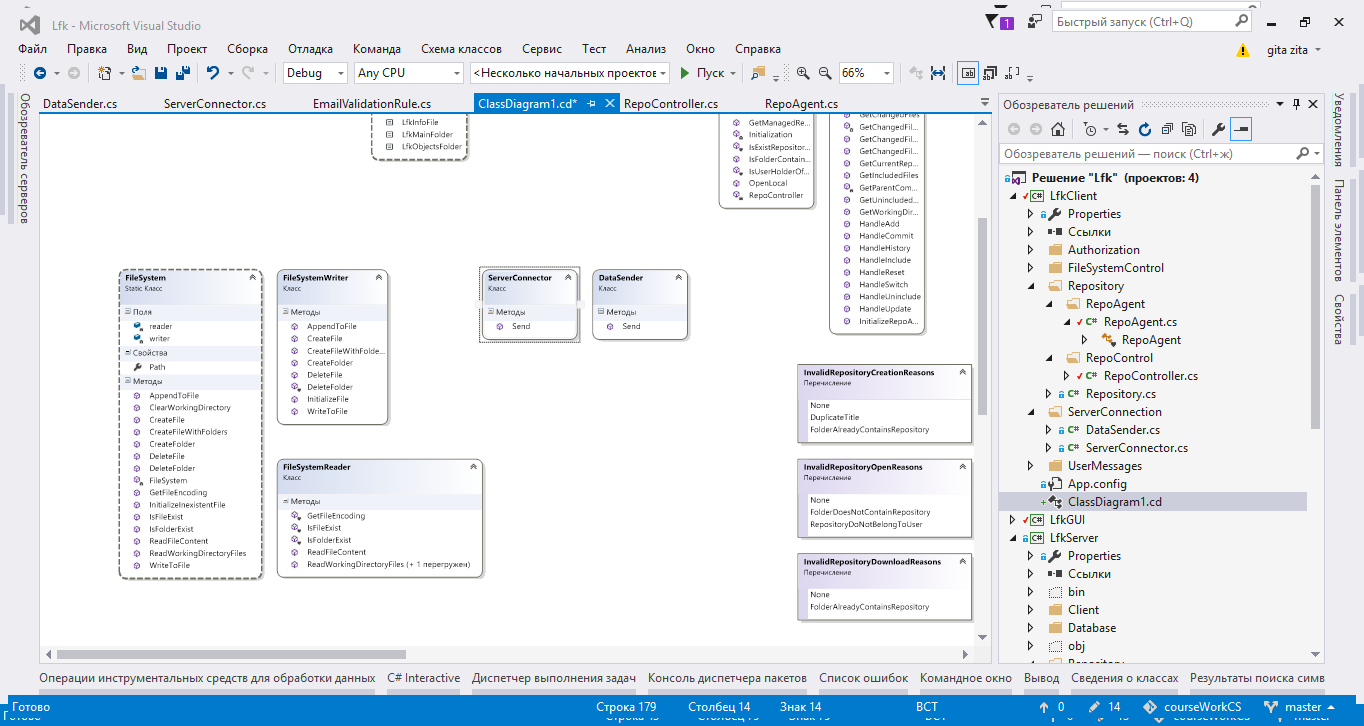


Рисунок 3.4.5 – Диаграмма вспомогательных классов.

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. Стоит отметить, что вся система Lfk заточена на работу с кодировкой Unicode: любой пользовательский файл, попадающий под управление Lfk, в первую очередь адаптируется под Unicode. Это достигается помощью метода, реализация которого представлена на рисунке 3.4.6.

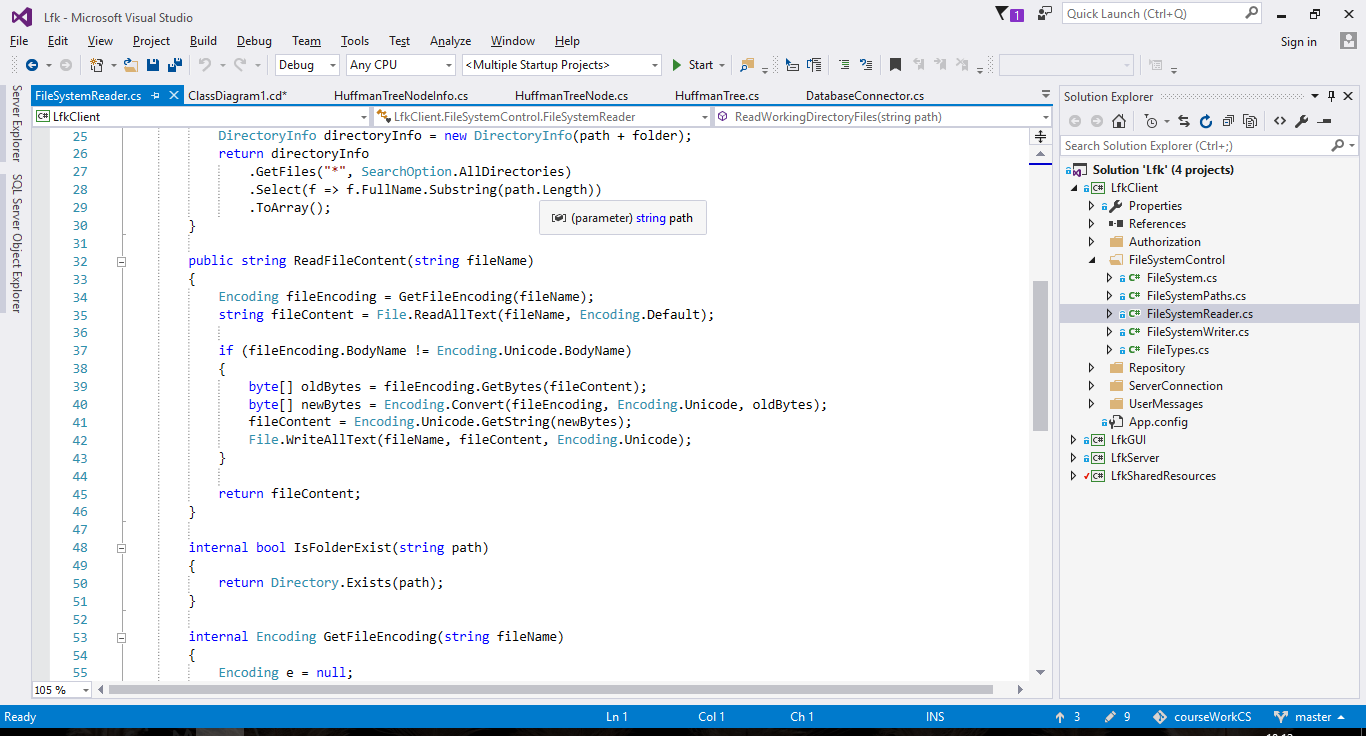


Рисунок 3.4.6 – Адаптация файла под кодировку Unicode при считывании его содержимого

Прежде чем считать содержимое файла, сперва вычисляется его кодировка. Если же она отлична от Unicode, то считанное содержимое перезаписывается в исходный файл в соответствии с данной кодировкой. Такой подход был необходим в первую очередь для стабильной работы алгоритма Хаффмана, описанного в 3.2, поскольку классическая реализация данного алгоритма подразумевает использование кодировки ASCII.

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 3.4.7 представлена реализация этого класса.

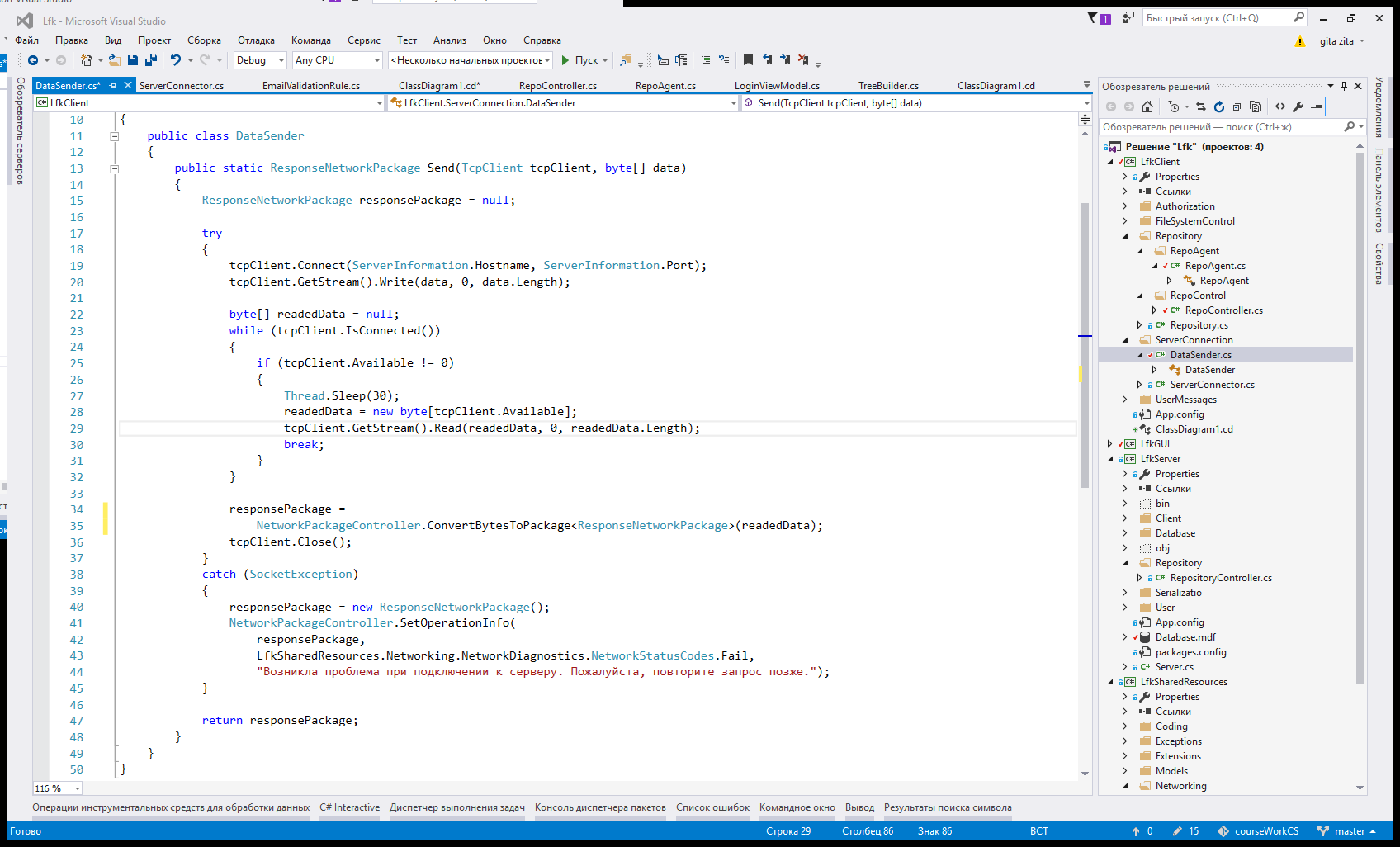


Рисунок 3.4.7 – Реализация метода соединения с сервером.

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

Реализация : RepoController, RepoAgent

## 3.5 Модуль LfkServer

Реализация : Контроллер

## 3.6 Проектирование базы данных

# 4. Руководство пользователя

# 5. Тестирование

## **5.1 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 5.1.1.

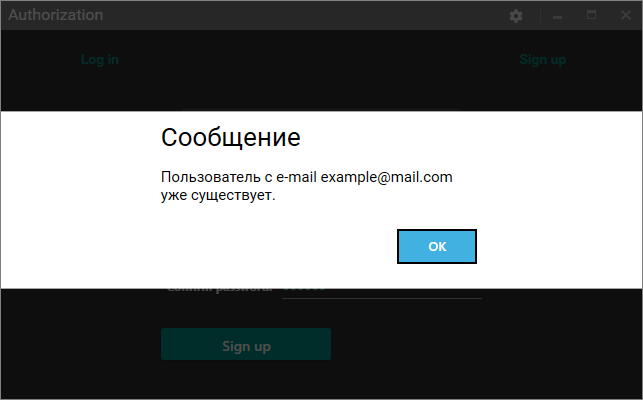


Рисунок 5.1.1 – Ошибка при регистрации.

В LfkGUI применяется валидация данных «на лету» что позволяет динамически сообщать пользователю о некорректно введенных данных. На рисунке 5.1.2 представлено окно регистрации в котором введены неверные данные, в силу этого блокируется возможность зарегистрировать пользователя.

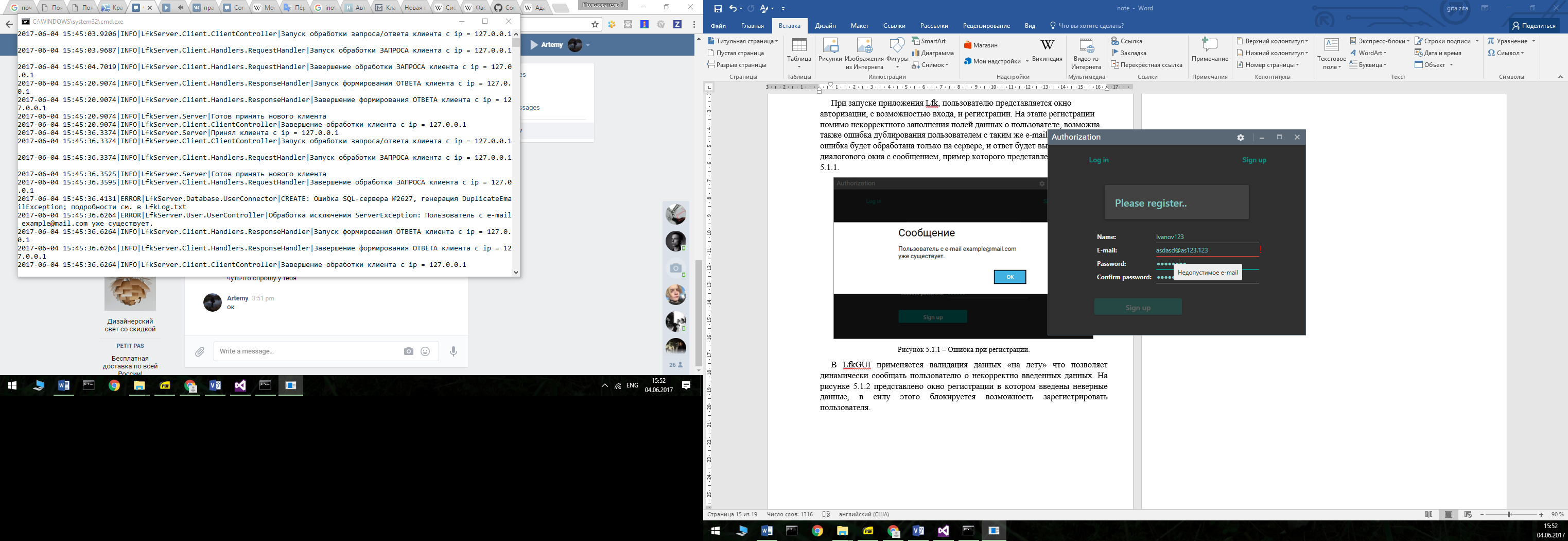


Рисунок 5.1.2 – Введение некорректных данных.

**5.2 Управление пользовательскими репозиториями**

Управление пользовательскими репозиториями осуществляется с помощью команд: открыть, скачать, показать все. Как видно на рисунке 5.2.1 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.

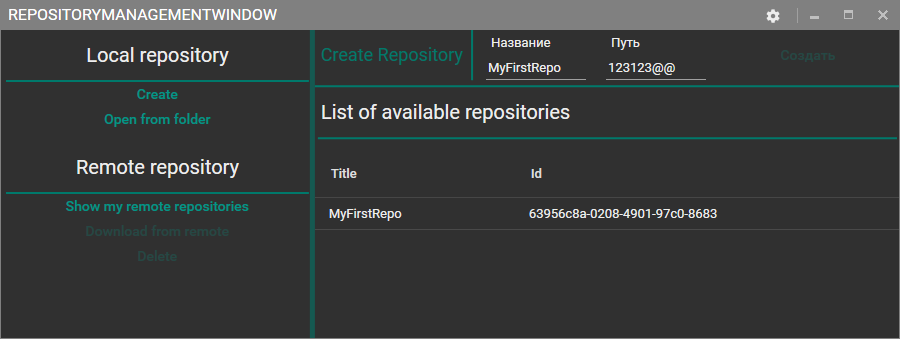


Рисунок 5.2.1 – Введение некорректных данных.

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 5.2.2.

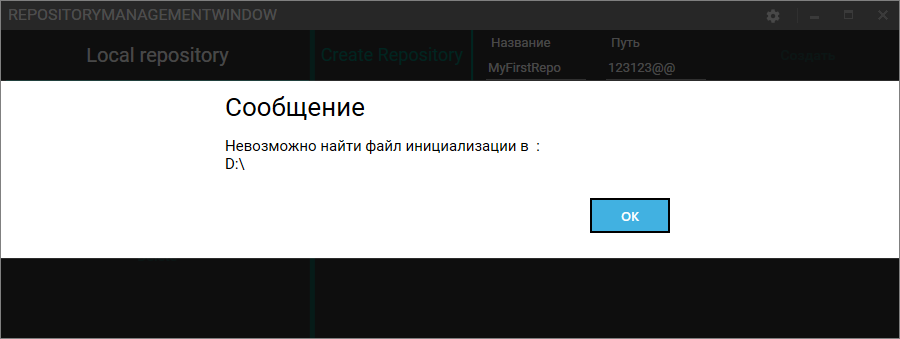


Рисунок 5.2.2 – Попытка открыть репозиторий из каталога, не содержащего репозиторий.

Также при попытке открыть репозиторий другого пользователя будет выведено сообщение о том, что репозиторий недоступен, данное сообщение представлено на рисунке 5.2.3.

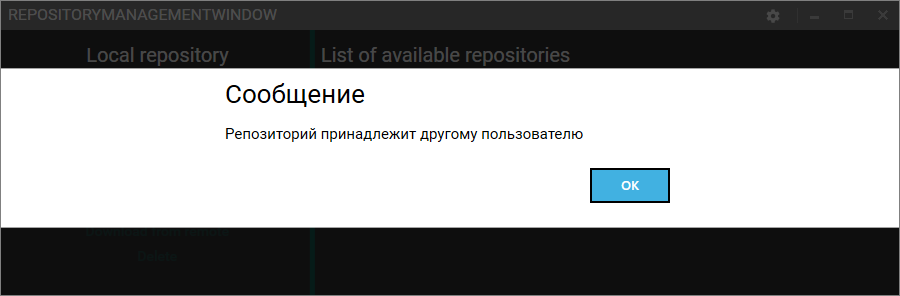


Рисунок 5.2.2 – Попытка открыть чужой репозиторий.

## **5.3 Контроль версий рабочей директории пользователя**

## 5.4 Сервер

# 6. Заключение

# Список литераторы