**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ**

**ГОМЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ   
УНИВЕРСИТЕТ ИМЕНИ П. О. СУХОГО**

Факультет автоматизированных и информационных систем

Кафедра «Информатика»

Специальность 1-40 04 01 «Информатика и технологии программирования»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту по дисциплине «МДСУБД»

на тему: **«Расчет потребности в сырье и материалах на плановый объем продукции»**

Исполнитель: студент гр. ИП-31

Коваленко А.И.

Руководитель: преподаватель

Фролова Е.В.

Дата проверки: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата допуска к защите: ­\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата защиты: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Оценка работы: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Подписи членов комиссии

по защите курсовой работы: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Гомель 2022

**СОДЕРЖАНИЕ**

[Введение 3](#_Toc121246882)

[1 Анализ предметной области 4](#_Toc121246883)

[1.1 Обзор реляционных и *NoSql* СУБД 4](#_Toc121246884)

[1.2 Используемые средства разработки и тестирования 5](#_Toc121246885)

[2 Архитектура программного обеспечения 7](#_Toc121246886)

[2.1 Пользовательские функции и роли 7](#_Toc121246887)

[2.2 Реляционная модель данных 10](#_Toc121246888)

[2.3 Анализ общности и различия моделей 15](#_Toc121246889)

[3 Структура и основные алгоритмы создаваемого приложения 17](#_Toc121246890)

[3.1 Описание интерфейсов и классов доступа к данным в СУБД 17](#_Toc121246891)

[3.2 Описание интерфейсов и классов реализации пользовательских функций 18](#_Toc121246892)

[4 Тестирование 24](#_Toc121246893)

[4.1 Unit тестирование 24](#_Toc121246894)

[Заключение 28](#_Toc121246895)

[Список использованных источников 29](#_Toc121246896)

[Приложение А](#_Toc121246897) [(Листинг доступа к данным) 30](#_Toc121246898)

[Приложение Б](#_Toc121246899) [(Листинг тестирования) 44](#_Toc121246900)

# ВВЕДЕНИЕ

С развитием информационных технологий компьютеры с их расширенными функциональными возможностями активно применяются в различных сферах человеческой деятельности, связанных с обработкой информации, предоставлением данных.

В современном обществе, которое функционирует в рыночных условиях, своевременная обработка информации способствует совершенствованию организации производства, оперативному и досрочному планированию, прогнозированию и анализу хозяйственной деятельности, что позволяет успешно конкурировать на рынке. Каждая организация стремится минимизировать затраты времени, материальных, трудовых ресурсов в ходе своей деятельности и упростить процесс обработки информации. Эти задачи можно решить с использованием автоматизированных информационных систем.

Необходимость автоматизации рабочего процесса возникает, в первую очередь, у тех фирм, которые имеют достаточно солидные обороты и объёмы продаж. Таким компаниям нужна оперативность и управляемость, а они, напротив, сталкиваются с ростом разного рода рутинной деятельности. Требуется, насколько возможно, избавить своих сотрудников от нее и более эффективно их использовать [3].

Целью данной курсовой работы является разработка автоматизированной информационной системы по расчету потребности в сырье и материалах на плановый объем продукции.

# 1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

# 1.1 Обзор реляционных и *NoSql* СУБД

СУБД *SQL Server* появилась в 1989 году и с тех пор значительно изменилась. Огромные изменения претерпели масштабируемость продукта, его целостность, удобство администрирования, производительность и функциональные возможности. Краткое введение в систему *SQL Server* 2000. Обзор новых возможностей: новые типы данных, поддержка *XML*, улучшения репликации, поддержка целостности ссылочных данных, улучшения полнотекстового поиска. Обзор реализаций системы *SQL Server*: клиент-серверная и автономная системы. Благодаря новым возможностям облегчается применение и администрирование *SQL Server*, повышается производительность работы *SQL Server*.

*Microsoft SQL Server*2019 – это реляционная система управления базой данных (СУБД). В реляционных базах данных данные хранятся в таблицах. Взаимосвязанные данные могут группироваться в таблицы, кроме того, могут быть установлены также и взаимоотношения между таблицами. Отсюда и произошло название реляционные – от английского слова *relational* (родственный, связанный отношениями, взаимозависимый). Пользователи получают доступ к данным на сервере через приложения, а администраторы, выполняя задачи конфигурирования, администрирования и поддержки базы данных, производят непосредственный доступ к серверу. *SQL**Server* является масштабируемой базой данных, это значит, что она может хранить значительные объемы данных и поддерживать работу многих пользователей, осуществляющих одновременный доступ к базе данных.

СУБД *SQL**Server* появилась в 1989 году и с тех пор значительно изменилась. Огромные изменения претерпели масштабируемость продукта, его целостность, удобство администрирования, производительность и функциональные возможности.

*MongoDB* реализует новый подход к построению баз данных, где нет таблиц, схем, запросов *SQL*, внешних ключей и многих других вещей, которые присущи объектно-реляционным базам данных.

В отличие от реляционных баз данных *MongoDB* предлагает документно-ориентированную модель данных, благодаря чему *MongoDB* работает быстрее, обладает лучшей масштабируемостью, ее легче использовать.

Но, даже учитывая все недостатки традиционных баз данных и достоинства *MongoDB*, важно понимать, что задачи бывают разные и методы их решения бывают разные. В какой-то ситуации *MongoDB* действительно улучшит производительность вашего приложения, например, если надо хранить сложные по структуре данные. В другой же ситуации лучше будет использовать традиционные реляционные базы данных. Кроме того, можно использовать смешенный подход: хранить один тип данных в *MongoDB*, а другой тип данных – в традиционных базах данных.

Вся система *MongoDB* может представлять не только одну базу данных, находящуюся на одном физическом сервере. Функциональность *MongoDB* позволяет расположить несколько баз данных на нескольких физических серверах, и эти базы данных смогут легко обмениваться данными и сохранять целостность.

*MongoDB* написана на *C++*, поэтому ее легко портировать на самые разные платформы. *MongoDB* может быть развернута на платформах *Windows, Linux, MacOS, Solaris*. Можно также загрузить исходный код и самому скомпилировать *MongoDB*, но рекомендуется использовать библиотеки с официального сайта.

**Если в традиционном мире *SQL* есть таблицы, то в мире *MongoDB* есть коллекции. И если в реляционных БД таблицы хранят однотипные жестко структурированные объекты, то в коллекции могут содержать самые разные объекты, имеющие различную структуру и различный набор свойств.**

**Система хранения данных в *MongoDB* представляет набор реплик. В этом наборе есть основной узел, а также может быть набор вторичных узлов. Все вторичные узлы сохраняют целостность и автоматически обновляются вместе с обновлением главного узла. И если основной узел по каким-то причинам выходит из строя, то один из вторичных узлов становится главным.**

# 1.2 Используемые средства разработки и тестирования

Разработка данного программного обеспечения выполнялось при помощи языка *С#*.

*C#* является объектно-ориентированным языком, но поддерживает также и компонентно-ориентированное программирование. Разработка современных приложений все больше тяготеет к созданию программных компонентов в форме автономных и самоописательных пакетов, реализующих отдельные функциональные возможности. Важная особенность таких компонентов – это модель программирования на основе свойств, методов и событий. Каждый компонент имеет атрибуты, предоставляющие декларативные сведения о компоненте, а также встроенные элементы документации. *C#* предоставляет языковые конструкции, непосредственно поддерживающие такую концепцию работы. Благодаря этому *C#* отлично подходит для создания и применения программных компонентов.

Язык имеет статическую типизацию, поддерживает полиморфизм, перегрузку операторов (в том числе операторов явного и неявного приведения типа), делегаты, атрибуты, события, переменные, свойства, обобщённые типы и методы, итераторы, анонимные функции с поддержкой замыканий, *LINQ*, исключения, комментарии в формате *XML*.

Для разработки графического интерфейса использовалась графическая подсистема, входящая в состав *.NET Framework*, использующая язык *XAML*: *WPF (Windows Presentation Foundation).* Это система следующего поколения для построения клиентских приложений *Windows* с визуально привлекательными возможностями взаимодействия с пользователем.

В основе *WPF* лежит векторная система отрисовки, не зависящая от разрешения и созданная с расчетом на возможности современного графического оборудования. *WPF* расширяет базовую систему полным набором функций разработки приложений, в том числе язык *XAML (Extensible Application Markup Language)*, элементами управления, привязкой данных, макетом, двухмерный- и трехмерный - графикой, анимацией, стилями, шаблонами, документами, мультимедиа, текстом и оформлением. *WPF* входит в состав *Microsoft .NET Framework* и позволяет создавать приложения, включающие другие элементы библиотеки классов *.NET Framework*.

Для хранения данных использовалась база данных *MongoDb.*

*MongoDB* – это *NoSQL* хранилище данных, крайне удобное для хранения информации, которая не может быть нормально структурирована в рамках реляционных баз данных.

*MongoDB* – это СУБД с открытым исходным кодом, не требующая описания схемы таблиц и написанная на языке *C++*. Благодаря специализации базы данных удалось отойти от принципа «один размер подо всё», а за счёт минимизации методов работы с транзакциями появилась возможность решения целого ряда проблем, связанных с недостатком производительности, что также способствовало улучшению горизонтального масштабирования.

Для тестирования был выбран фреймворк *MsTest.* Это фреймворк юнит-тестирования от компании *Microsoft*, который по умолчанию включен в *Visual* *Studio* и который также можно использовать с *.NET Core*. Данный фреймворк предоставляют несложный *API*, который позволяет быстро написать и автоматически проверить тесты.

# 2 АРХИТЕКТУРА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

# 2.1 Пользовательские функции и роли

Предметная область должна описывать все объекты, которые являются частью предметной области, а также любые взаимоотношения между ними. Следовательно, необходимо описать все зависимости между сущностями. Важно знать какие именно объекты попадают в предметную область и какие связями между ними существуют и необходимо помнить цель проектирования данного приложения.

Для формирования представления о предметной области используют *UML* диаграммы, которые представлены на рисунках 2.1, 2.2, 2.3, 2.4.

Актер – множество логически связанных ролей в *UML*, исполняемых при взаимодействии с прецедентами или сущностями (система, подсистема или класс). Актером может быть человек или другая система, подсистема или класс, которые представляют нечто вне сущности.

Любые (в том числе и программные) системы проектируются с учётом того, что в процессе своей работы они будут использоваться людьми и/или взаимодействовать с другими системами. Сущности, с которыми взаимодействует система в процессе своей работы, называются актерами, причём каждый актер ожидает, что система будет вести себя строго определённым, предсказуемым образом.

Графически актeр изображается символом класса с соответствующим стереотипом. Эта форма представления имеет один и тот же смысл и могут использоваться в диаграммах. «Стереотипированная» форма чаще применяется для представления системных актеров или в случаях, когда актер имеет свойства и их нужно отобразить.

Прецеденты представляют действия, выполняемые системой в интересах актеров. Проще говоря, прецедент – это описание последовательности действий (или нескольких последовательностей), которые выполняются системой и производят для отдельного актера видимый результат. Один актер может использовать несколько элементов прецедентов, и наоборот, один прецедент может иметь несколько актеров, использующих его. Каждый прецедент задает определенный путь использования системы. Набор всех прецедентов определяет полные функциональные возможности системы.

На рисунке 2.1 отображена диаграмма, отображающая возможные ветки управления данным приложением на уровень доступа «Гость»

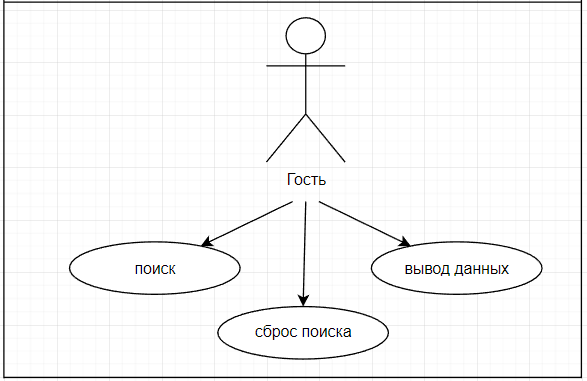


Рисунок 2.1 – Диаграмма актера «Гость»

На рисунке 2.2 изображена диаграмма, отображающая возможные ветки управления данным приложением на уровень доступа «Клиент»

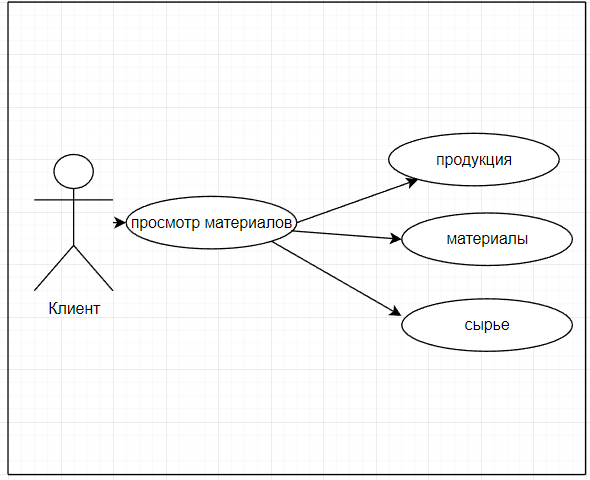


Рисунок 2.2 – Диаграмма прецедентов роли «Клиент»

На рисунке 2.3 изображена диаграмма, отображающая возможные ветки управления данным приложением на уровень доступа «Менеджер»

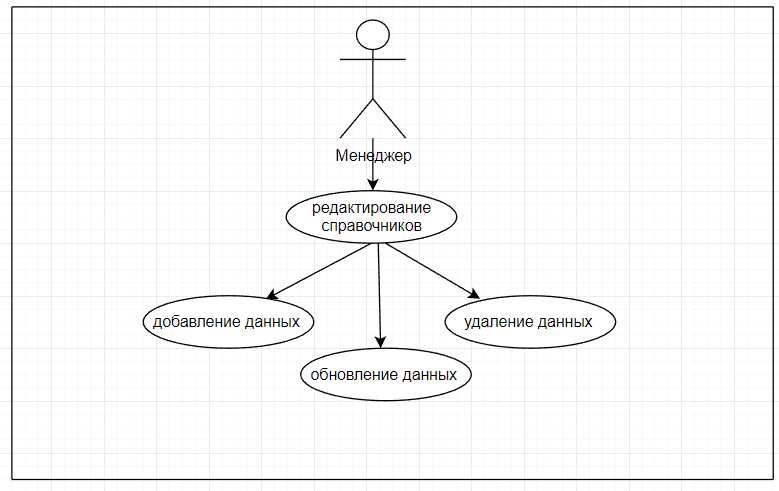


Рисунок 2.3 – Диаграмма прецедентов роли «Менеджер»

На рисунке 2.4 изображена диаграмма, отображающая возможные ветки управления данным приложением на уровень доступа «Администратор»

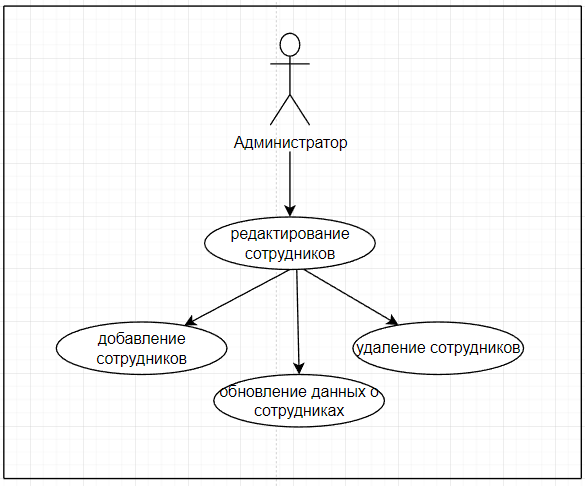


Рисунок 2.4 – Диаграмма прецедентов роли «Администратор»

Далее приведено описание методов и ролей.

«Гость» – пользователь, которому не доступны большинство функции. Единственное что он может – это просмотр справочника товаров, о нас, контакты и отзывы наших клиентов.

«Клиент» – это пользователь, задачей которого является оформление заказа. Для удобства просмотра истории заказов данному уровню доступа доступен личный кабинет, где он может посмотреть подробности заказа.

«Менеджер» – это работник, предназначенный для согласования заказа с клиентом, то есть основная задача которого является отказ или принятие заказа.

«Каталог товаров» метод предназначен для ознакомления с материалами, располагающими у фирмы.

«Сделать заказ» служит для оформления заказа клиентом.

«Изменение статуса заказа» предназначен для предоставления менеджеру возможности изменения состояния заказа: отказать в услуге или принять.

«Статистика» – это окно, предназначенное для отображения статистики, генерируемой автоматически по следующий полям:

* количество заказов за всё время работы;
* количество клиентов за всё время работы;
* количество сотрудников на данный момент времени;
* количество материалов, имеющихся на складе на данный момент

времени;

* средняя стоимость всех имеющихся материалов.

Прецедент «Личный кабинет» служит для просмотра информации о заказах, которые выполняет сотрудник с данным *ID*.

# 2.2 Реляционная модель данных

Реляционная модель – совокупность данных, состоящая из набора двумерных таблиц. В теории множеств таблице соответствует термин отношение (*relation*), физическим представлением которого является таблица, отсюда и название модели – реляционная.

Соответственно теория построения баз данных, которая является приложением к задачам обработки данных таких разделов математики, как теория множеств и логика первого порядка.

В сравнении с иерархической и сетевой моделью данных, реляционная модель отличается более высоким уровнем абстракции данных. Реляционная модель является удобной и наиболее привычной формой представления данных, так в настоящее время эта модель является фактическим стандартом, на который ориентируются практически все современные коммерческие СУБД.

На реляционной модели данных строятся реляционные базы данных Модель данных на рисунке 2.5 представляет набор таблиц, связанных между собой.

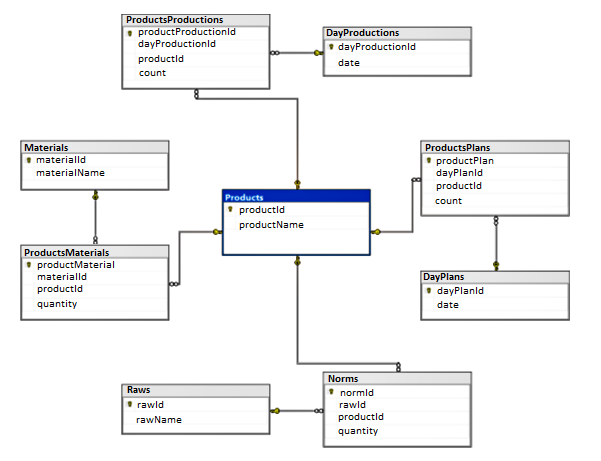


Рисунок 2.5 – Реляционная модель данных

Рассмотрим подробно назначения таблиц реляционной модели данных, показанных на рисунке 2.5.

Таблица «*Materials»* предназначена для хранения дополнительной информации о таблице «*ProductsMateruals»*.

Таблица «*Products»* предназначена для хранения в себе всех видов продуктов, имеющихся на складе магазина.

Таблица «*Raws*» имеет схожее назначение с таблицей «*Materials»* т.к. они оба являются справочниками нашей информационной системы.

Таблица «*Norms»* предназначена для хранения уникального идентификатора определённого вида продукта с дальнейшей обработкой данных

Таблица «*DayPlans»* предназначена для хранения дополнительной информации о таблице «*ProductsPlans»*.

Таблица «*ProductsPlans»* предназначена для хранения даты запланированного события, продукта со склада, а также количество продукта.

Таблица «*DayProductions»* предназначена для хранения дополнительной информации о таблице «*ProductsProductions»*.

Таблица «*ProductsProductions»* предназначена для хранений уникальных данных о продуктах с указаний количества.

Между двумя сущностями может быть установлена связь. Отношения между сущностями характеризуются глаголом, который можно применить для взаимодействия между ними. Связь – это некое отношение между двумя типами сущностей.

Между типами сущностей различают следующих 3 типа связей.

«Один-к-одному». Это значит, что одному экземпляру некоторой сущности может соответствовать только один экземпляр другой сущности.

«Один-ко-многим». Это значит, что одному экземпляру сущности может соответствовать любое количество экземпляров другой сущности. Если известно значение максимального количества экземпляров, то непосредственно указывается само значение.

«Многие-ко-многим». Это означает, что нескольким экземплярам одной сущности может соответствовать несколько экземпляров другой сущности.

Таблица 2.1 – Связи между сущностями

|  |  |  |  |
| --- | --- | --- | --- |
| **Имя таблицы** | | **Тип связи** | **Имя таблицы** |
| 1 | | 2 | 3 |
| Products | | Один ко многим | Norms |
| ProductsMateruals |
| ProductsPlans |
| ProductsProductions |
| ProductsMaterials |
| Materials | | Один ко многим | ProductsMateruals |
| Raws | Один ко многим | | Norms |
| DayPlans | Один ко многим | | ProductsPlans |
| DayProductions | Один ко многим | | ProductsProductions |
| Materials | Один ко многим | | ProductsMaterials |

Между всеми таблицами было установлено обеспечение целостности данных. Целью обеспечения целостности данных является предотвращение появления непарных записей, ссылающихся на несуществующие записи. В результате *SQL* отменяет для этого отношения все действия, которые могут нарушить целостность данных.

Также может потребоваться удаление строки и всех связанных с ней записей – например, записи в таблице «*Clients*» и всех связанных с этим клиентом заказов. Для этого был установлен параметр «Каскадное удаление связанных записей». При удалении записи, содержащей первичный ключ, будут автоматически удалены все записи, связанные с этим первичным ключом. Также стоит заметить, если каскадное удаление не разрешено, невозможно удалить запись в главной таблице, если имеются связанные с ней записи в подчиненной.

1. **Документная модель данных**

При переносе реляционной модели базы данных в документную представляется в виде нескольких подходов. Первый из них это полный перенос реляционной модели базы данных в документную и последующее связывание коллекций на слое бизнес-логики.

Второй способ заключается в реализации составных коллекций документной базы данных с использованием справочников, для валидации данных.

Документная модель – СУБД, специально предназначенная для хранения иерархических структур данных (документов) и обычно реализуемая с помощью подхода *NoSQL*.

Документная база данных – это тип нереляционных баз данных, предназначенный для хранения и запроса данных в виде документов в формате, подобном *JSON*. Документные базы данных позволяют разработчикам хранить и запрашивать данные в БД с помощью той же документной модели, которую они используют в коде приложения. Гибкий, полуструктурированный, иерархический характер документов и документных баз данных позволяет им развиваться в соответствии с потребностями приложений.

Данная схема отражает модель базы данных, на которой мы можем заметить, что у нас имеется 4 коллекции: *Materials, Products, Raws, Users.*

Коллекция «*Materials*» предназначена для наименования материала. Информация о полях представлена в таблице 2.2.

Таблица 2.2 – Информация о полях коллекции *Materials*

|  |  |  |
| --- | --- | --- |
| **Имя атрибута** | **Тип атрибута** | **Назначение** |
| Id | ObjectId | Хранение уникального идентификатора |
| Name | string | Хранение уникального имени материала |

Коллекция «*Products*» предназначена для хранения данных о продукции. Информация о полях представлена в таблице 2.3.

Таблица 2.3 – Информация о полях коллекции *Products*

|  |  |  |
| --- | --- | --- |
| **Имя атрибута** | **Тип атрибута** | **Назначение** |
| id | ObjectId | Хранение уникального идентификатора |
| Name | string | Хранение уникального имени продукта |

Коллекция «*Raws*» предназначена для хранения данных сырье. Информация о полях представлена в таблице 2.4.

Таблица 2.4 – Информация о полях коллекции *Raws*

|  |  |  |
| --- | --- | --- |
| **Имя атрибута** | **Тип атрибута** | **Назначение** |
| id | ObjectId | Хранение уникального идентификатора |
| Name | string | Хранение уникального имени сырья |

Коллекция «*Users*» предназначена для хранения уникальной информации присущей каждому пользователю информационной системы, такой как логин, пароль и уровень доступа в систему. Информация о полях представлена в таблице 2.5.

Таблица 2.5 – Информация о полях коллекции *Users*

|  |  |  |
| --- | --- | --- |
| **Имя атрибута** | **Тип атрибута** | **Назначение** |
| id | ObjectId | Хранение уникального номера. |
| Level | String | Уровень доступа в систему |
| Login | String | Хранение логина пользователя |
| Password | String | Хранение пароля пользователя |

# 2.3 Анализ общности и различия моделей

Общность моделей заключается в том, что они описывают одну и ту же модель. Это значит, что есть возможность организации высших слоев так, чтобы они одинаково работали с обеими моделями данных, что в разы уменьшит нагрузку на программистов.

Главное различие между моделями заключается в том, как организовываются связи между сущностями. В реляционной модели связи существуют внутри самой модели и заключаются в внешних ключах, где программист прописывает правила взаимодействия между таблицами. В документной модели же, связи между сущностями программист должен организовывать на более высоких уровнях, что возлагает на него дополнительную нагрузку. Так же важным будет заметить то, что в все таблицы в реляционной модели являются плоскими. Это значит, что внутри таблицы не предусматривают вложенные сущности, которые уместны в этом проекте. В следствии этого, программисту необходимо создавать дополнительные связи и тем самым усложнять проект. В это время документная модель предусматривает вложенные сущности, что значительно облегчает базу данных, однако, есть необходимость создания справочных сущностей.

Основываясь на том, что было написано выше, можно сделать вывод – обе модели можно использовать для реализации задачи оптимизации

Основные возможности сервера *SQL*:

* реляционная база данных – это набор таблиц, содержащих данные, которые подразделяются на предварительно определенные категории;
* каждая таблица содержит одну или несколько категорий данных в столбцах;
* каждая строка содержит уникальный экземпляр данных для категорий, определенных столбцами;
* пользователь может получить доступ к данным из базы данных, не зная структуру таблицы базы данных.

Ограничения для базы данных *SQL*.

Масштабируемость**.** Пользователи должны масштабировать реляционную базу данных на мощных серверах, которые дороги и сложны в обращении. Для масштабирования реляционной базы данных она должна быть распределена по нескольким серверам. Обработка таблиц на разных серверах – это хаос.

Сложность**.** В любом случае данные *SQL*-сервера должны вписываться в таблицы. Если ваши данные не помещаются в таблицы, вам нужно спроектировать структуру базы данных, которая будет сложной и, опять же, сложной для обработки.

В последние несколько лет как научные, так и веб-компании подвергли сомнению единый подход, подходящий для всех, в отношении хранилищ данных, что должно привести к появлению большого разнообразия альтернативных баз данных. Движение, а также новые хранилища данных обычно относятся к термину *NoSQL*.

Основное качество *NoSQL* состоит в том, что он может не требовать фиксированных схем таблиц, обычно избегать операций объединения и обычно масштабируется горизонтально. Академические исследователи обычно называют эти базы данных структурированным хранилищем, термин, который включает в себя классические реляционные базы данных в качестве подмножества.

База данных *NoSQL* также заменяет *ACID* (атомарность, согласованность, изоляция и долговечность). Базы данных *NoSQL*, в той или иной степени, допускают различия в схеме данных от записи к записи.

*NoSQL* – аналог реляционной базы данных, в которой информация хранится без строгой структуры и явной связи между другими сведениями. Данные здесь могут храниться не только в табличной, но и текстовой, в графической, аудио-, видео- и любой другой форме. Они используются тогда, когда в приоритете не четкое структурирование данных, а гибкая, масштабируемая база с высоким уровнем производительности. Здесь нет никаких ограничений ни при хранении, ни при использовании данных.

Чтобы понять, какой вид технологии предоставляет больше возможностей для разработки, выполним некоторое сравнение баз данных на основе реляционных и нереляционных систем:

* структура и тип данных. *SQL* требует жесткой структуризации на основании шаблонов. В *NoSQL* по отношению к структуре не предъявляется никаких требований;
* масштабируемость. В реляционной базе данных *SQL* предусмотрено вертикальное масштабирование. В *NoSQL* можно использовать и вертикальное, и горизонтальное. Но второй вариант более простой и практичный и используется чаще;
* запросы. В реляционных системах получить данные можно при помощи языка *SQL*. А вот в каждой *NoSQL*-базе предусмотрен свой алгоритм работы.

# 3 СТРУКТУРА И ОСНОВНЫЕ АЛГОРИТМЫ СОЗДАВАЕМОГО ПРИЛОЖЕНИЯ

# 3.1 Описание интерфейсов и классов доступа к данным в СУБД

Чтобы осуществить связь между реляционной базой данных и приложением на *C#* необходим посредник. И именно таким посредником является технология *ADO.NET*. Это набор классов (фреймворк) для работы с базами данных, а также *XML* файлами. Аббревиатура *ADO* расшифровывается как *ActiveX Data Objects.* Данная технология имеет методы и классы для извлечения и обработки данных.

Для подключения к базе данных был создан объект класса *OleDbConnection*, который в качестве параметра принимает строку подключения. Для открытия и закрытия используются методы класса *OleDbConnection Open* и *Close* соответственно. Класс *OleDbCommand* создает запрос к БД и принимает объект *OleDbConnection*. Во избежание инъекций используются свойства класса *OleDbCommand Parameters* и *AddWithValue.* Объект *DataReader* считывает полученные в результате запроса данные. Объект *DataSet* предназначен для хранения данных из БД и позволяет работать с ними независимо от БД. Объект *DataAdapter* является посредником между *DataSet* и источником данных. Через эти объекты ведется основная работа с базой данных.

В процессе разработки были созданы классы доступа к базе данных: *Materials, Products, Raws, Users*.

Во всех этих классах были реализованы следующие методы:

* *getCollection.* Данный метод позволяет нам получить все данные из коллекций;
* *find.* Данный метод вернет найденную запись в коллекции;
* *remove.* Данный методудалит выбранную запись из коллекции;
* *update.* Данный метод обновит любую запись в коллекции при общении к нему.

Для получения базы данных применяется метод *GetDatabase*, в который передается название базы. Этот метод возвращает объект *IMongoDatabase*. Данные в базе данных хранятся в коллекциях, которые представлены объектом *IMongoCollection.*

# 3.2 Описание интерфейсов и классов реализации пользовательских функций

Описание ООП-интерфейса, если отвлечься от деталей синтаксиса конкретных языков, состоит из двух частей: имени и методов интерфейса.

Имя интерфейса строится по тем же правилам, что и другие идентификаторы используемого языка программирования. Разные языки и среды разработки имеют различные соглашения по оформлению кода, в соответствии с которыми имена интерфейсов могут формироваться по некоторым правилам, которые помогают отличать имя интерфейса от имён других элементов программы.

Методы интерфейса. В описании интерфейса определяются имена и сигнатуры входящих в него методов, то есть процедур или функций класса.

При непосредственной интеграции базы данных в приложение понятие «сущность» заменяется на «доменный класс».

Доменные классы – это классы в объектно-ориентированных компьютерных программах, выражающие сущности из модели предметной области, относящейся к программе, и реализующие бизнес-логику программы.

Задачи доменных классов представлена в таблице 3.1.

Таблица 3.1 – Назначения доменных классов

|  |  |  |
| --- | --- | --- |
| **Исходное название доменного класса** | **Назначение доменного класса** | |
| 1 | 2 | |
| ProductsProductions | Хранение данных о продукции | |
| DayProductions | Хранение дополнительных данных о продукции | |
| ProductsMaterials | Хранение данных о материалах | |
| Materials | Хранение дополнительных данных о материалах | |
| Raws | Хранение дополнительных данных о сырье | |
| Norms | Хранение данных о сырье | |
| DayPlans | Хранение даты о событии | |
| ProductsPlans | Хранением данных о событиях и материалах с учётом их количества | |
| DayProductions | Хранение даты о событии с продукцией |

Продолжение таблицы 3.1 – Назначение доменных классов

|  |  |
| --- | --- |
| 1 | 2 |
| ProductsProductions | Хранением данных о событиях и продукции с учётом их количества |

В таблице 3.2 предоставлено описание доменного класса *Raws.*

Таблица 3.2 – Описание полей доменного класса *Raws*

|  |  |  |
| --- | --- | --- |
| **Тип данных свойства** | **Свойство доменного класса** | **Предназначение свойства доменного класса** |
| int | rawid | Необходим для хранения уникального номера сырья |
| string | rawName | Необходим для хранения имени сырья |

В таблице 3.3 предоставлено описание доменного класса *Norms.*

Таблица 3.3 – Описание полей доменного класса *Norms*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Тип данных свойства** | | **Свойство доменного класса** | **Предназначение свойства доменного класса** | |
| int | | normId | Необходим для хранения уникального номера сырья | |
| int | | rawId | Необходим для хранения уникальной связи с таблицей Raws | |
| int | | productId | Необходим для хранения уникального номера продукта | |
| int | quantity | | | Необходим для хранения количества сырья |

В таблице 3.4 предоставлено описание доменного класса *DayPlans.*

Таблица 3.4 – Описание полей доменного класса *DayPlans*

|  |  |  |
| --- | --- | --- |
| **Тип данных свойства** | **Свойство доменного класса** | **Предназначение свойства доменного класса** |
| int | dayPlanid | Для хранения уникального номера события |
| string | date | Необходим для хранения даты к событию |

В таблице 3.5 предоставлено описание доменного класса *ProductsPlans.*

Таблица 3.5 – Описание полей доменного класса *ProductsPlans*

|  |  |  |
| --- | --- | --- |
| **Тип данных свойства** | **Свойство доменного класса** | **Предназначение свойства доменного класса** |
| int | produtPlan | Необходим для хранения уникального номера события |
| int | dayPlanid | Для хранения уникального номера события |
| int | productId | Необходим для хранения уникального номера продукта |
| int | count | Необходим для хранения количества |

В таблице 3.6 предоставлено описание доменного класса *DayProductions.*

Таблица 3.6 – Описание полей доменного класса *DayProductions*

|  |  |  |
| --- | --- | --- |
| **Тип данных свойства** | **Свойство доменного класса** | **Предназначение свойства доменного класса** |
| int | dayProductionsId | Необходим для хранения уникального номера продукта |
| String | date | Необходим для хранения даты к событию |

В таблице 3.7 предоставлено описание доменного класса *ProductsProductions.*

Таблица 3.7 – Описание полей доменного класса *ProductsProductions*

|  |  |  |
| --- | --- | --- |
| **Тип данных свойства** | **Свойство доменного класса** | **Предназначение свойства доменного класса** |
| int | productProductionid | Необходим для хранения уникального номера продукции |
| string | dayproductionsid | Необходим для хранения уникального номера продукта из таблицы *DayProductions* |
| int | productid | Необходим для хранения уникального номера |
| int | count | Необходим для хранения количества |

В таблице 3.8 предоставлено описание доменного класса *Materials.*

Таблица 3.8 – Описание полей доменного класса *Materials*

|  |  |  |
| --- | --- | --- |
| **Тип данных свойства** | **Свойство доменного класса** | **Предназначение свойства доменного класса** |
| int | materialId | Необходим для хранения уникального номера материала |
| string | materialName | Необходим для хранения названия материала |

В таблице 3.9 предоставлено описание доменного класса *ProdutsMaterials.*

Таблица 3.9 – Описание полей доменного класса *ProdutsMaterials*

|  |  |  |
| --- | --- | --- |
| **Тип данных свойства** | **Свойство доменного класса** | **Предназначение свойства доменного класса** |
| int | productMaterials | Необходим для хранения уникального номера материала |
| Int | materialId | Необходим для хранения идентификатора из таблицы *Materials* |
| Int | productId | Необходим для хранения идентификатора из таблицы *Products* |
| Int | quanity | Необходим для хранения количества материала по текущей таблице |

Так как *MongoDB* представляет документно-ориентированные базы данных, то все данные в ней хранятся в виде документов. Таким образом, база данных состоит из коллекций, а коллекции – из документов. Каждый документ представляет набор пар элементов ключ-значение наподобие словаря и представлен классом *BsonDocument* из пространства имен *MongoDB.BSON.*

В нереляционной базе данных *MongoDb* данные представлены в виде коллекций, которые состоят из документов. Также документ может включать в себя вложенный документ и т.д. Для работы с документами были разработаны основные (представляют документ коллекции) и вложенные классы (представляют вложенные документы). Листинг тестирования представлен в приложении А.

В таблице 3.10 приведено описание основных классов.

Таблица 3.10 – Основные классы

|  |  |
| --- | --- |
| **Исходное название класса** | **Назначение класса** |
| Materials | Необходим для хранения данных о материалах |
| Products | Необходим для хранения данных о продукции |
| Raws | Необходим для хранения данных о сырье |
| Users | Необходим для хранения данных о пользователях нашей информационной системы |

В таблице 3.11 приведено описание свойств класса *Materials*.

Таблица 3.11 – Описание свойств класса *Materials*

|  |  |  |
| --- | --- | --- |
| **Тип атрибута** | **Имя атрибута** | **Назначение** |
| ObjectId | Id | Необходим для хранения уникального номера объекта Materials |
| String | Name | Необходим для уникального имени материала |

В таблице 3.12 приведено описание свойств класса *Products*.

Таблица 3.12 – Описание свойств класса *Products*

|  |  |  |
| --- | --- | --- |
| **Тип атрибута** | **Имя атрибута** | **Назначение** |
| 1 | 2 | 3 |
| ObjectId | Id | Необходим для хранения уникального идентификатора продукта |
| string | Name | Необходим для хранения уникального имени продукта |
| Object | Plans | Необходим для хранения уникальной информации о событии с указанием даты и количества необходимой продукции |
| Object | Productions | Необходим для хранения уникальной информации о продукции с указанием даты и количества необходимой продукции |

Продолжение таблицы 3.12

|  |  |  |
| --- | --- | --- |
| 1 | 2 | 3 |
| Object | Norms | Необходим для хранения уникальной информации о сырье с указанием названия сырья и количества |
| Object | Materials | Необходим для хранения уникальной информации о материале с указанием названия материала и количества |

В таблице 3.13 приведено описание свойств класса *Raws*.

Таблица 3.13 – Описание свойств класса *Raws*

|  |  |  |
| --- | --- | --- |
| **Тип атрибута** | **Имя атрибута** | **Назначение** |
| ObjectId | Id | Необходим для хранения уникального идентификатора сырья |
| String | Name | Необходим для хранения имени сырья |

В таблице 3.14 приведено описание свойств класса *Users*.

Таблица 3.14 – Описание свойств класса *Users*

|  |  |  |
| --- | --- | --- |
| **Тип атрибута** | **Имя атрибута** | **Назначение** |
| ObjectId | \_id | Необходим для хранения уникального идентификатора пользователя информационной системы |
| String | Level | Необходим для хранения уникального уровня доступа для каждого пользователя информационной системы |
| String | Login | Необходим для хранения уникального логина для каждого пользователя информационной системы |
| String | Password | Необходим для хранения уникального пароля для каждого пользователя информационной системы |

# 4 ТЕСТИРОВАНИЕ

# 4.1 Unit тестирование

Тест-кейс – это профессиональная документация тестировщика, последовательность действий, направленная на проверку какого-либо функционала, описывающая как прийти к фактическому результату.

Для создания тест-кейсов выбран объект коллекции *Users «ProductMaterials»*

Информация об атрибутах и допустимых значениях справочника приведена в таблице 4.1.

Таблица 4.1 – Допустимые значения для полей справочника

|  |  |  |  |
| --- | --- | --- | --- |
| **Номер** | **Наименование полей** | **Тип** | **Требование к данным** |
| 1 | Quantity | Целочисленный | От 0, до 999 |
| 2 | Materials | Символьный | Не пустое поле |

Описание тест-кейсов для тестирования представлено в таблице 4.2.

Таблица 4.2 – Таблица тест-кейсов для тестирования.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Номер** | **Модуль** | **Заглавие тест-кейса** | **Шаги** | **Ожидаемый**  **результат** |
| 1 | Блок валидации | Позитивное  Тестирование | Тесты  Гр. 1 | True |
| 2 | Блок валидации | Негативное  Тестирование | Тесты  Гр. 2 | False |

Описание тестов группы 1 (Позитивное тестирование) представлено в таблице 4.3.

Таблица 4.3 – Тесты группы 1 (Позитивное тестирование)

|  |  |  |  |
| --- | --- | --- | --- |
| **Номер** | **Тестирование** | **Значения полей** | |
| ***Quantity*** | ***Materials*** |
| 1 | На границе слева | 1 | >= 1 символу |
| 2 | В пределах границ | Любое число от 1, до 999 | - |
| 3 | На границе справа | 999 | - |

Описание тестов группы 2 (Негативное тестирование) представлено в таблице 4.4.

Таблица 4.4 – Тесты группы 2 (Негативное тестирование)

|  |  |  |  |
| --- | --- | --- | --- |
| **Номер** | **Тестирование** | **Значения полей** | |
| ***Quantity*** | ***Materials*** |
| 1 | За границей слева | 0 | < 1 |
| 2 | За границей справа | 1000 | - |

Описание валидационного тестирования представлено в таблице 4.5.

Таблица 4.5 – Валидационное тестирование класса *Repositories*.

|  |  |  |  |
| --- | --- | --- | --- |
| **Название теста** | **Поступаемые данные** | **Описание теста** | **Ожидаемый результат** |
| GetCollection\_Correct | - | Вызывается метод *GetCollection()* и данные полученные из него проверяются на соответствие с данными в базе данных | Полное соответствие данных |
| Update | Объект со схожим типом данных | Вызывается метод *Update(T t)*, по заранее переданному значению. | Полное изменение данных |
| Add\_ ReturnNewItem | - | Вызывается метод *Add*, в который передается заранее созданный объект класса *Collection*. Затем, созданный элемент проверяется на соответствие с тем, что передали в метод | Полное соответствие элементов |

**Кроме фреймворков для создания и проведения юнит-тестов при тестировании часто бывают полезны такие фреймворки, которые позволяют имитировать или эмулировать какую-то функциональность или создавать мок-объекты. Подобных фреймворков существует множество, и одним из самых популярных является *Moq*.**

*Moq* – это простой и легковесный изоляционный фреймврк (*Isolation Framework*), который построен на основе анонимных методов и деревьев выражений. Для создания моков он использует кодогенерацию, поэтому позволяет «мокать» интерфейсы, виртуальные методы (и даже защищенные методы) и не позволяет «мокать» невиртуальные и статические методы.

***Moq* предназначен для имитации объектов. В данном случае имитируется функциональность интерфейса *Products*. Описание тестов представлено в таблице 4.6.**

Таблица 4.6 – *Moq* тестирование интерфейса *Products*.

|  |  |  |  |
| --- | --- | --- | --- |
| **Название теста** | **Поступаемые данные** | **Описание теста** | **Ожидаемый результат** |
| *ProductTest* | - | Вызывается метод *Valid*. Данные, полученные из него, сравниваются с имеющимся набором | Полное совпадение |
| *ProductNMaterialTest* | *-* | Вызывается метод *Valid*. Данные, полученные из него, сравниваются с имеющимся набором | Полное совпадение |

В Moq нет разделения между моками и стабами, однако нам с вами будет значительно проще различать два синтаксиса инициализации заглушек. Так, «*moq functional specification*» синтаксис может использоваться только для тестирования состояния (т.е. для стабов) и не может применяться для задания поведения. Инициализация же заглушек методом *Setup* может быть, во-первых, более многословной, а во-вторых, при ее использовании не совсем понятно, собираемся ли мы проверять поведение или состояние.

В некоторых случаях неудобно использовать несколько методов *Verify* для проверки нескольких вызовов. Вместо этого можно создать мок-объект и задать ожидаемое поведение с помощью методов *Setup* и проверять все эти допущения путем вызова одного метода *Verify*(). Такая техника может быть удобной для повторного использования мок-объектов, создаваемых в методе *Setup* теста. Листинг тестирования представлен в приложении Б.

На рисунке 4.1 создается объект класса *Mock* и говорится, что он будет имитировать методы интерфейса *Products*. Затем вызывается имитированный метод и полученный из него данные сравниваются с исходными данными.

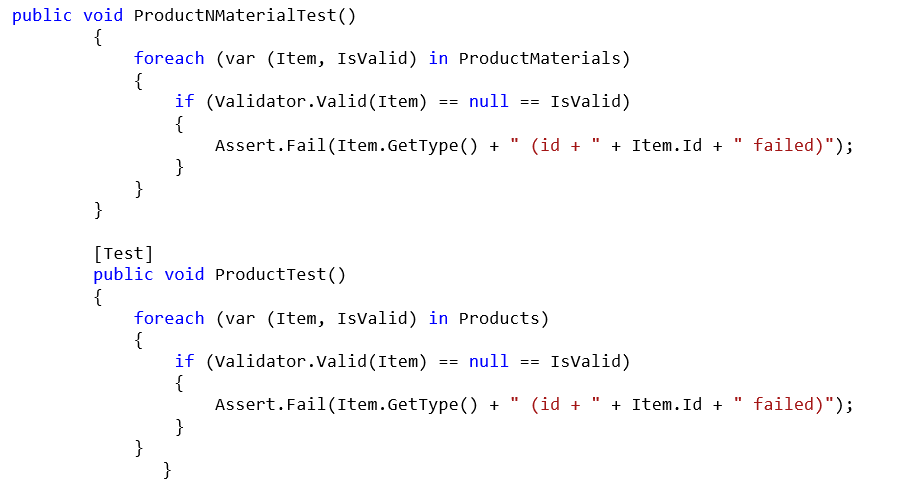


Рисунок 4.1 – Пример *Moq* тестирования

# ЗАКЛЮЧЕНИЕ

В результате выполнения, данной курсовой работы, было спроектировано и разработано приложение. Для этого было проведено изучение предметной области и выявление необходимых данных. Для хранения данных спроектирована и разработана база данных *MongoDB*, удовлетворяющая необходимым требованиям. Разработан пользовательский интерфейс с использованием технологии *WPF*. Также реализована ролевая политика для установки ограничения редактирования и просмотра данных пользователям для обеспечения защиты хранимых данных. Разработанное приложение содержит необходимый функционал для выполнения поставленной задачи.

В процессе создания приложения были пройдены такие этапы разработки, как постановка задачи, составление логической модели предметной области, создание и реализация алгоритма работы программы с последующим тестированием.

Преимуществом разработанной нами системы является ее относительно низкие требования к установке и использования, а также простота и удобство в эксплуатации, гибкость программирования, что позволит дальнейшую доработку по запросу владельца предприятия в случае внесения каких-либо изменений в комплексной работе предприятия.

# Список использованных источников

1. Джозеф Альбахари и Бен Альбахари. C# 7.0. Карманный справочник. – «O’Reilly Media, Inc», 2017. – 156 с.
2. Веб-приложение «Основы ADO.NET» – Электрон. данные.: – Режим доступа: https://itvdn.com/ru/blog/article/basicsado#:~:text=Введение,для%20извлечения%20и%20обработки%20данных – Дата доступа: 01.12.2022.
3. Рихтер, Джеффри CLR via C#. Программирование на платформе Microsoft .NET Framework 4.0 на языке C# / Джеффри Рихтер. - М.: Питер, 2013. - 928 c
4. Когаловский М. Р. Энциклопедия технологий баз данных. – М.: Финансы и статистика, 2002. – 800 с.
5. Веб-приложение «Логическая модель предметной области» – Электрон. данные.: – Режим доступа: <http://analyst.by/diagrams/logicheskaya-model-predmetnoy-oblasti> – Дата доступа: 01.12.2022.
6. Веб-приложение «Введение в WPF» – Электрон. данные.: – Режим доступа: <https://studfile.net/preview/4532318/> – Дата доступа: 01.12.2022.

# ПРИЛОЖЕНИЕ А

# (Листинг доступа к данным)

Папка Models

public class Material

{

public int Id { get; set; }

public string Name { get; set; }

}

public class Norm

{

public int Id { get; set; }

public Raw { get; set; }

public double Quantity { get; set; }

}

public class Product

{

public int Id { get; set; }

public string Name { get; set; }

public List<ProductPlan> Plans { get; set; } = new();

public List<ProductProduction> Productions { get; set; } = new();

public List<Norm> Norms { get; set; } = new();

public List<ProductMaterial> Materials { get; set; } = new();

}

public class ProductMaterial

{

public int Id { get; set; }

public double Quantity { get; set; }

public Material { get; set; }

}

public class ProductPlan

{

public int Id { get; set; }

public DateTime Date { get; set; }

public int Count { get; set; }

public string DateStr => Date.ToString("d");

}

public class ProductProduction

{

public int Id { get; set; }

public DateTime Date { get; set; }

public int Count { get; set; }

}

public class Raw

{

public int Id { get; set; }

public string Name { get; set; }

}

public class User

{

public int Id { get; set; }

public string Login { get; set; }

public string Password { get; set; }

public int Level { get; set; }

public string Access => Accesses[Level];

public static readonly Dictionary<int, string> Accesses = new()

{

[1] = "Администратор",

[2] = "Менеджер",

[3] = "Пользователь"

};

}

ublic static class Validator

{

private static bool IsRussian(this string str)

{

return Regex.IsMatch(str, @"\p{IsCyrillic}");

}

private static bool CheckLength(this string str, int leftBorder = 3, int rightBorder = 100)

{

return (str.Length >= 3) && (str.Length <= 100);

}

public static User Auth(DataLayer data, string login, string password)

{

return data.Users.GetAll().FirstOrDefault(x => (x.Login == login) && (x.Password == password));

}

public static bool CompareDate(DateTime first, DateTime second)

{

return (first.Day == second.Day) && (first.Month == second.Month) && (first.Year == second.Year);

}

public static string Valid(ProductProduction material, Product product)

{

if ((material.Count <= 0) || (material.Count >= 100000))

{

return "Неверное количество";

}

if ((material.Date <= DateTime.Now.AddYears(-50)) || (material.Date >= DateTime.Now.AddYears(50)))

{

return "Неверная дата";

}

if (product.Plans.Any(x => CompareDate(x.Date, material.Date)))

{

return "На этот день уже назначен результат";

}

return null;

}

public static string Valid(ProductPlan material, Product product)

{

if ((material.Count <= 0) || (material.Count >= 100000))

{

return "Неверное количество";

}

if ((material.Date <= DateTime.Now.AddYears(-50)) || (material.Date >= DateTime.Now.AddYears(50)))

{

return "Неверная дата";

}

if (product.Plans.Any(x => CompareDate(x.Date, material.Date)))

{

return "На этот день уже назначен план";

}

return null;

}

public static string Valid(Norm material)

{

if (material.Raw == null)

{

return "Не выбрано сырьё";

}

if (material.Quantity is < 0.01 or > 10000)

{

return "Неверное количество";

}

return null;

}

public static string Valid(ProductMaterial material)

{

if (material.Material == null)

{

return "Не выбран материал";

}

if (material.Quantity is < 0.01 or > 10000)

{

return "Неверное количество";

}

return null;

}

public static string Valid(Product raw)

{

if (!raw.Name.CheckLength())

{

return "Неверный формат наименования";

}

if (!raw.Name.IsRussian())

{

return "Название должно быть на кириллице";

}

return null;

}

public static string Valid(Material raw)

{

if (!raw.Name.CheckLength())

{

return "Неверный формат наименования";

}

if (!raw.Name.IsRussian())

{

return "Название должно быть на кириллице";

}

return null;

}

public static string Valid(Raw raw)

{

if (!raw.Name.CheckLength())

{

return "Неверный формат наименования";

}

if (!raw.Name.IsRussian())

{

return "Название должно быть на кириллице";

}

return null;

}

public static string Valid(User, IRepository<User> data, bool ignoreLogin = false)

{

if (!ignoreLogin)

{

if (data.GetAll().Any(x => x.Login == user.Login))

{

return "Уже существует пользователь с таким логином";

}

}

if (user.Login.Contains(' '))

{

return "В строке логин не может быть пробелов";

}

if (!user.Login.CheckLength())

{

return "Строка логин некорректна";

}

if (!user.Password.CheckLength())

{

return "Строка пароль неккоректна";

}

return null;

}

}

Папка Mongo.Entities

public static class Data

{

public static DataLayer Get()

{

return new(new Materials(), new Products(), new Raws(), new Users());

}

}

public class Material

{

[BsonElement("\_id")]

public int Id { get; set; }

[BsonElement("name")]

public string Name { get; set; }

}

public class Norm

{

[BsonElement("quanity")]

public double Quantity { get; set; }

[BsonElement("raw")]

public Raw { get; set; }

}

public class Product

{

[BsonElement("\_id")]

public int Id { get; set; }

[BsonElement("name")]

public string Name { get; set; }

[BsonElement("production")]

public List<ProductProduction> Productions { get; set; } = new();

[BsonElement("plans")]

public List<ProductPlan> Plans { get; set; } = new();

[BsonElement("materials")]

public List<ProductMaterial> Materials { get; set; } = new();

[BsonElement("norms")]

public List<Norm> Norms { get; set; } = new();

}

public class ProductMaterial

{

[BsonElement("quantity")]

public double Quantity { get; set; }

[BsonElement("material")]

public Material { get; set; }

}

public class ProductPlan

{

[BsonElement("date")]

public DateTime Date { get; set; }

[BsonElement("count")]

public int Count { get; set; }

}

public class ProductProduction

{

[BsonElement("date")]

public DateTime Date { get; set; }

[BsonElement("count")]

public int Count { get; set; }

}

public class Raw

{

[BsonElement("\_id")]

public int Id { get; set; }

[BsonElement("name")]

public string Name { get; set; }

}

public class User

{

[BsonElement("\_id")]

public int Id { get; set; }

[BsonElement("login")]

public string Login { get; set; }

[BsonElement("password")]

public string Password { get; set; }

[BsonElement("access")]

public int Level { get; set; }

}

Папка Mongo.Repositiories

public class Materials : IRepository<Material>

{

private static readonly string collectionName = "Materials";

public bool Add(Material itemToAdd)

{

try

{

GetCollection().InsertOne(new()

{

Id = GetFreeId(),

Name = itemToAdd.Name

});

}

catch (Exception)

{

return false;

}

return true;

}

public static IMongoCollection<Entities.Material> GetCollection()

{

return DataBaseClient.GetDatabase().GetCollection<Entities.Material>(collectionName);

}

public Material Find(int id)

{

return GetAll().FirstOrDefault(x => x.Id == id);

}

public IEnumerable<Material> GetAll()

{

return GetCollection().Find(new BsonDocument()).ToList().Select(x => new Material()

{

Id = x.Id,

Name = x.Name,

});

}

public int GetFreeId()

{

var data = GetAll();

return data.Any() ? data.Max(x => x.Id) + 1 : 1;

}

public bool Remove(int id)

{

try

{

GetCollection().DeleteOne(x => x.Id == id);

}

catch (Exception)

{

return false;

}

return true;

}

public bool Update(Material itemToUpdate)

{

Entities.Material item = new()

{

Id = itemToUpdate.Id,

Name = itemToUpdate.Name

};

try

{

GetCollection().ReplaceOne(x => x.Id == itemToUpdate.Id, item);

}

catch (Exception)

{

return false;

}

return true;

}

}

public class Products : IRepository<Product>

{

private static readonly string collectionName = "Products";

public bool Add(Product itemToAdd)

{

try

{

GetCollection().InsertOne(new()

{

Id = GetFreeId(),

Name = itemToAdd.Name

});

}

catch (Exception)

{

return false;

}

return true;

}

public static IMongoCollection<Entities.Product> GetCollection()

{

return DataBaseClient.GetDatabase().GetCollection<Entities.Product>(collectionName);

}

public Product Find(int id)

{

return GetAll().FirstOrDefault(x => x.Id == id);

}

public IEnumerable<Product> GetAll()

{

return GetCollection().Find(new BsonDocument()).ToList().Select(x => new Product()

{

Id = x.Id,

Name = x.Name,

Plans = x.Plans.Select(p => new ProductPlan()

{

Date = p.Date,

Count = p.Count

}).ToList(),

Productions = x.Productions.Select(p => new ProductProduction()

{

Count = p.Count,

Date = p.Date

}).ToList(),

Norms = x.Norms.Select(n => new Norm()

{

Raw = new Raws().Find(n.Raw.Id),

Quantity = n.Quantity

}).ToList(),

Materials = x.Materials.Select(m => new ProductMaterial()

{

Material = new Materials().Find(m.Material.Id),

Quantity = m.Quantity

}).ToList()

});

}

public int GetFreeId()

{

var data = GetAll();

return data.Any() ? data.Max(x => x.Id) + 1 : 1;

}

public bool Remove(int id)

{

try

{

GetCollection().DeleteOne(x => x.Id == id);

}

catch (Exception)

{

return false;

}

return true;

}

public bool Update(Product itemToUpdate)

{

Entities.Product item = new()

{

Id = itemToUpdate.Id,

Name = itemToUpdate.Name,

Plans = itemToUpdate.Plans.Select(p => new Entities.ProductPlan()

{

Date = p.Date,

Count = p.Count

}).ToList(),

Productions = itemToUpdate.Productions.Select(p => new Entities.ProductProduction()

{

Count = p.Count,

Date = p.Date

}).ToList(),

Norms = itemToUpdate.Norms.Select(n => new Entities.Norm()

{

Raw = Raws.GetCollection().Find(z => z.Id == n.Raw.Id).FirstOrDefault(),

Quantity = n.Quantity

}).ToList(),

Materials = itemToUpdate.Materials.Select(m => new Entities.ProductMaterial()

{

Quantity = m.Quantity,

Material = Materials.GetCollection().Find(x => x.Id == m.Material.Id).FirstOrDefault()

}).ToList()

};

try

{

GetCollection().ReplaceOne(x => x.Id == itemToUpdate.Id, item);

}

catch (Exception)

{

return false;

}

return true;

}

}

public class Raws : IRepository<Raw>

{

private static readonly string collectionName = "Raws";

public bool Add(Raw itemToAdd)

{

try

{

GetCollection().InsertOne(new()

{

Id = GetFreeId(),

Name = itemToAdd.Name

});

}

catch (Exception)

{

return false;

}

return true;

}

public static IMongoCollection<Entities.Raw> GetCollection()

{

return DataBaseClient.GetDatabase().GetCollection<Entities.Raw>(collectionName);

}

public Raw Find(int id)

{

return GetAll().FirstOrDefault(x => x.Id == id);

}

public IEnumerable<Raw> GetAll()

{

return GetCollection().Find(new BsonDocument()).ToList().Select(x => new Raw()

{

Id = x.Id,

Name = x.Name

});

}

public int GetFreeId()

{

var data = GetAll();

return data.Any() ? data.Max(x => x.Id) + 1 : 1;

}

public bool Remove(int id)

{

try

{

GetCollection().DeleteOne(x => x.Id == id);

}

catch (Exception)

{

return false;

}

return true;

}

public bool Update(Raw itemToUpdate)

{

Entities.Raw item = new()

{

Id = itemToUpdate.Id,

Name = itemToUpdate.Name

};

try

{

GetCollection().ReplaceOne(x => x.Id == itemToUpdate.Id, item);

}

catch (Exception)

{

return false;

}

return true;

}

}

public class Users : IRepository<User>

{

private readonly string collectionName = "Users";

public bool Add(User itemToAdd)

{

try

{

GetCollection().InsertOne(new()

{

Id = GetFreeId(),

Level = itemToAdd.Level,

Login = itemToAdd.Login,

Password = itemToAdd.Password

});

}

catch (Exception)

{

return false;

}

return true;

}

private IMongoCollection<Entities.User> GetCollection()

{

return DataBaseClient.GetDatabase().GetCollection<Entities.User>(collectionName);

}

public User Find(int id)

{

return GetAll().FirstOrDefault(x => x.Id == id);

}

public IEnumerable<User> GetAll()

{

return GetCollection().Find(new BsonDocument()).ToList().Select(x => new User()

{

Id = x.Id,

Level = x.Level,

Login = x.Login,

Password = x.Password

});

}

public int GetFreeId()

{

var data = GetAll();

return data.Any() ? data.Max(x => x.Id) + 1 : 1;

}

public bool Remove(int id)

{

try

{

GetCollection().DeleteOne(x => x.Id == id);

}

catch (Exception)

{

return false;

}

return true;

}

public bool Update(User itemToUpdate)

{

Entities.User item = new()

{

Id = itemToUpdate.Id,

Level = itemToUpdate.Level,

Login = itemToUpdate.Login,

Password = itemToUpdate.Password

};

try

{

GetCollection().ReplaceOne(x => x.Id == itemToUpdate.Id, item);

}

catch (Exception)

{

return false;

}

return true;

}

}

Класс доступа к данным DataBaseClient

public static class DataBaseClient

{

private static readonly string conStr;

private static readonly MongoClient client;

private static IMongoDatabase db;

private static readonly string dbName = "FactoryDb";

static DataBaseClient()

{

conStr = "mongodb://localhost:27017";

client = new(conStr);

}

public static IMongoDatabase GetDatabase()

{

if (db is null)

{

db = client.GetDatabase(dbName);

return db;

}

else

{

return db;

}

}

}

# ПРИЛОЖЕНИЕ Б

# (Листинг тестирования)