ОТЧЕТ

по лабораторной работе № 6

«Исследование распределенной системы контроля версий Git при коллективной разработке программных продуктов»

по дисциплине

«ТЕХНОЛОГИИ СОЗДАНИЯ ПРОГРАММНЫХ ПРОДУКТОВ»

Выполнил студент группы ИС/б-22о

Горбенко К.Н.

Проверил:

Строганов В.А.

* 1. ЦЕЛЬ РАБОТЫ

Исследовать основные подходы к организации взаимодействия команды разработчиков с использованием распределенной системы контроля версий (DVCS). Приобрести практические навыки установки и настойки DVCS Git, организации ветвей разработки и осуществление слияния.

* 1. ЗАДАНИЕ НА РАБОТУ

Разработать программный продукт, соответствующий следующему варианту предметной области, используя распределенную систему контроля версий Git:

Система – “Магазин по продаже компьютерной техники”.

Система предназначена для автоматизации работы магазина, в котором необходимо предусмотреть работу нескольких подразделений.

В системе должны быть реализованы следующие функции: для руководства магазина: заказ товаров у поставщиков, просмотр статистики по продажам за разные периоды и различные виды товаров; для работников склада: прием товаров на баланс, учет товаров на складе (просмотр наличия товара на складе, его количества), формирование отчета по изменению загруженности склада; для кассиров – принять деньги за товар, выдать деньги в случае возврата товара, сформировать чек для покупателя; для продавцов – формирование накладной для покупателя.

Диаграмма классов, соответствующая предметной области, представлена на рисунке № 1:

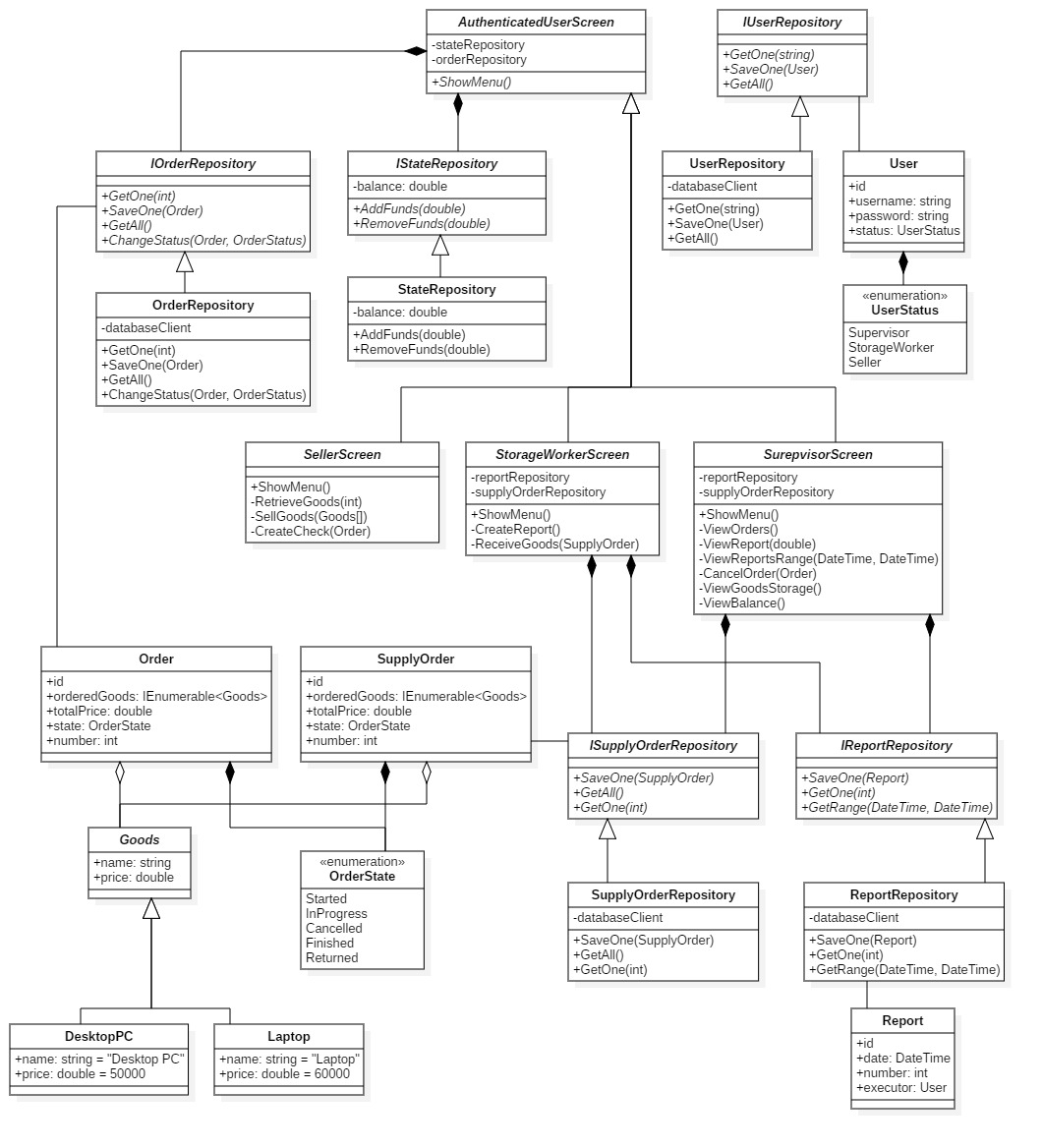


Рисунок 1 – Диаграмма классов

* 1. ХОД РАБОТЫ
     1. Создание репозитория и начало работы

Создадим репозиторий:

cd Source\Repos\

md OfflineElectronicsStore

cd OfflineElectronicsStore

git init

Результат:



Рисунок 2 – Результат выполнения команды git init

Добавим к репозиторию файл VisualStudio.gitignore для того, чтобы автоматически игнорировать файлы, используемые средой разработки Visual Studio и не относящиеся к реализации программного кода проекта. Содержимое файла взято из следующего репозитория: <https://github.com/github/gitignore/blob/master/VisualStudio.gitignore>.

Создадим проект библиотеки классов и совершим коммит с новыми файлами:

git add \*

git commit -m "Created store class library for domain model"

* + 1. Работа в ветви для реализации доменной модели магазина

Создадим ветвь:

git checkout -b implementing-store-model

Результат:

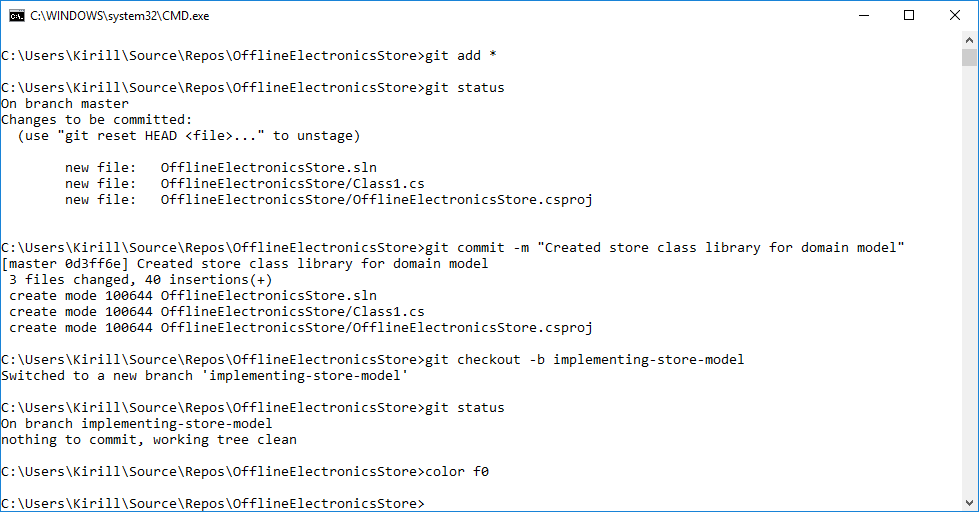


Рисунок 3 – Результат выполнения команды git checkout

Создадим классы пользователя, продукта, заказа и заказа на поставку, содержащие следующий код:

public class User

{

public long Id { get; set; }

public string Username { get; set; }

public string Password { get; set; }

public UserRole Status { get; set; }

}

public enum UserRole

{

Supervisor,

StorageWorker,

Seller

}

public class Product

{

public long Id { get; set; }

public string Name { get; set; }

public double Price { get; set; }

public int Availability { get; set; }

public void SetPrice(double price) => Price = price;

}

public class Order

{

public long Id { get; set; }

public IEnumerable<Product> Products { get; set; }

public OrderStatus State { get; set; }

public double TotalPrice => Products.Sum(x => x.Price);

}

public class SupplyOrder : Order

{

public bool Received { get; set; }

}

public enum OrderStatus

{

Finished,

Returned

}

Добавим созданные файлы в репозиторий и совершим коммит:

git add \*

git commit -m "Created model for User, Order, SupplyOrder and Product"

Результат выполнения команды изображен на рисунке 4:

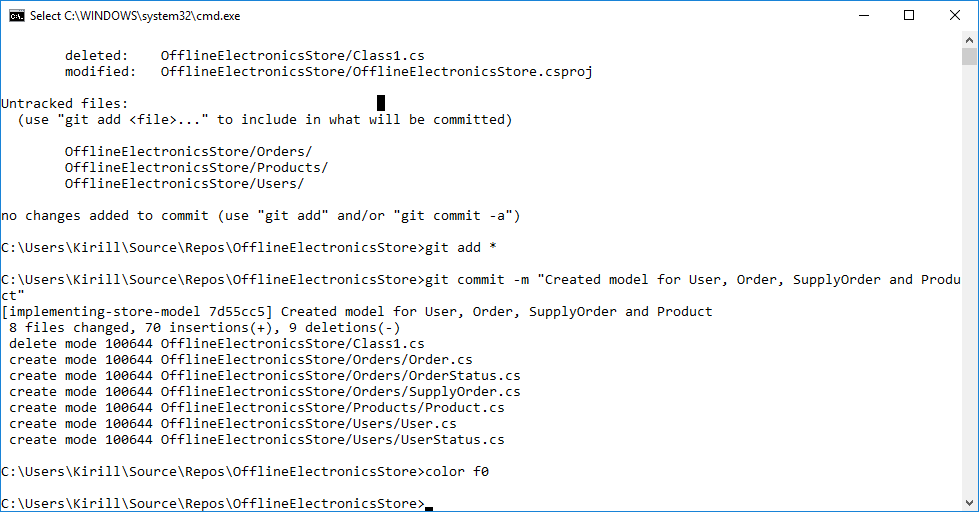


Рисунок 4 – Результат выполнения команды git commit

Далее создадим интерфейсы репозиториев для хранения данных:

public interface IUserRepository

{

Task<User> GetOneAsync(string username);

Task SaveOneAsync(User user);

}

public interface IProductRepository

{

Task AddProductAsync(Product product);

Task AddOneProductUnit(Product product);

Task RemoveOneProductUnit(Product product);

Task<IEnumerable<Product>> GetAvailableProductsAsync();

}

public interface IOrderRepository

{

Task AddOneAsync(Order order);

Task UpdateOneAsync(Order order);

Task<IEnumerable<Order>> GetFinishedOrdersAsync();

}

public interface ISupplyOrderRepository

{

Task AddOneAsync(SupplyOrder supplyOrder);

Task UpdateOneAsync(SupplyOrder order);

Task<IEnumerable<SupplyOrder>> GetStartedSupplyOrdersAsync();

}

Добавим созданные файлы в репозиторий и совершим коммит:

git add \*

git commit -m "Created interfaces for data storage repositories"

Результат выполнения команды изображен на рисунке 5:

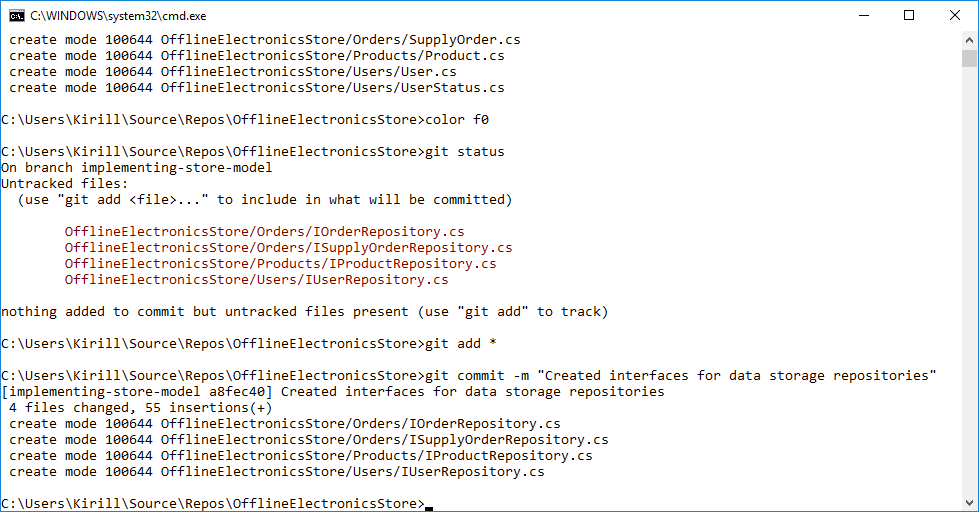


Рисунок 5 – Результат выполнения команды git commit

Создадим класс, представляющий состояние магазина (на данный момент он будет хранить только счет и методы, позволяющие начислять и снимать деньги):

public class StoreState

{

private static StoreState \_instance;

private StoreState() { }

public static StoreState GetInstance()

{

if (\_instance == null)

\_instance = new StoreState();

return \_instance;

}

public double Funds { get; private set; }

public void Deposit(double funds)

=> Funds += funds;

public void Withdraw(double funds)

=> Deposit(-funds);

}

git add OfflineElectronicsStore\StoreState.cs

git commit -m "Created StoreState class responible for centralized storing of cash account"

Теперь создадим класс, предоставляющий интерфейс кассира:

public class SellerScreen

{

private readonly StoreState \_shopState;

private readonly IProductRepository \_productRepository;

private readonly IOrderRepository \_orderRepository;

public SellerScreen(StoreState shopState, IProductRepository productRepository, IOrderRepository orderRepository)

{

\_shopState = shopState ?? throw new ArgumentNullException(nameof(shopState));

\_productRepository = productRepository ?? throw new ArgumentNullException(nameof(productRepository));

\_orderRepository = orderRepository ?? throw new ArgumentNullException(nameof(orderRepository));

}

public async Task<IEnumerable<Product>> RetrieveProductsAsync()

=> await \_productRepository.GetAvailableProductsAsync();

public async Task<IEnumerable<Order>> RetrieveFinishedOrdersAsync()

=> await \_orderRepository.GetFinishedOrdersAsync();

public async Task<string> SellProductsAsync(IEnumerable<Product> products)

{

if (products == null) throw new ArgumentNullException(nameof(products));

var order = new Order

{

Products = products,

State = OrderStatus.Finished

};

await \_orderRepository.AddOneAsync(order);

foreach (var product in order.Products)

await \_productRepository.RemoveOneProductUnit(product);

\_shopState.Deposit(order.TotalPrice);

return CreateCheck(order);

}

public async Task ReturnOrder(Order order)

{

if (order == null) throw new ArgumentNullException(nameof(order));

order.State = OrderStatus.Returned;

await \_orderRepository.UpdateOneAsync(order);

\_shopState.Withdraw(order.TotalPrice);

foreach (var product in order.Products)

await \_productRepository.AddOneProductUnit(product);

}

private string CreateCheck(Order order)

{

if (order == null) throw new ArgumentNullException(nameof(order));

var products = string.Join("", order.Products);

return $"Order: {products}, total: {order.TotalPrice}, sell confirmed. <Replace this string with something more appropriate>";

}

}

git add \*

git commit -m "Created SellerScreen class to provide seller interface"

Создадим класс, предоставляющий интерфейс работника склада:

public class StorageWorkerScreen

{

private readonly StoreState \_shopState;

private readonly ISupplyOrderRepository \_supplyOrderRepository;

private readonly IProductRepository \_productRepository;

public StorageWorkerScreen(

StoreState shopState,

ISupplyOrderRepository supplyOrderRepository,

IProductRepository productRepository)

{

\_shopState = shopState ?? throw new ArgumentNullException(nameof(shopState));

\_supplyOrderRepository = supplyOrderRepository ?? throw new ArgumentNullException(nameof(supplyOrderRepository));

\_productRepository = productRepository ?? throw new ArgumentNullException(nameof(productRepository));

}

public async Task<IEnumerable<SupplyOrder>> GetNotReceivedProducts()

=> await \_supplyOrderRepository.GetStartedSupplyOrdersAsync();

public async Task ReceiveProducts(SupplyOrder order)

{

if (order == null) throw new ArgumentNullException(nameof(order));

foreach (var product in order.Products)

await \_productRepository.AddOneProductUnit(product);

\_shopState.Withdraw(order.TotalPrice);

order.Received = true;

await \_supplyOrderRepository.UpdateOneAsync(order);

}

}

git add \*

git commit -m "Created StorageWorkerClass to provide storage worker interface"

Создадим класс, предоставляющий интерфейс руководства магазина:

public class SupervisorScreen

{

private readonly StoreState \_storeState;

private readonly IOrderRepository \_orderRepository;

private readonly ISupplyOrderRepository \_supplyOrderRepository;

private readonly IProductRepository \_productRepository;

public SupervisorScreen(StoreState storeState,

IOrderRepository orderRepository,

ISupplyOrderRepository supplyOrderRepository,

IProductRepository productRepository)

{

\_storeState = storeState ?? throw new ArgumentNullException(nameof(storeState));

\_orderRepository = orderRepository ?? throw new ArgumentNullException(nameof(orderRepository));

\_supplyOrderRepository = supplyOrderRepository ?? throw new ArgumentNullException(nameof(supplyOrderRepository));

\_productRepository = productRepository ?? throw new ArgumentNullException(nameof(productRepository));

}

public double GetCurrentBalance()

=> \_storeState.Funds;

public void DepositFunds(double funds)

=> \_storeState.Deposit(funds);

public void WithdrawFunds(double funds)

=> DepositFunds(-funds);

public async Task<IEnumerable<Product>> GetProducts()

=> await \_productRepository.GetAllProductsAsync();

public async Task<IEnumerable<Order>> GetFinishedOrdersAsync()

=> await \_orderRepository.GetFinishedOrdersAsync();

public async Task<IEnumerable<SupplyOrder>> GetSupplyOrdersAsync()

=> await \_supplyOrderRepository.GetStartedSupplyOrdersAsync();

public async Task CreateSupplyOrder(IEnumerable<Product> products)

{

if (products == null) throw new ArgumentNullException(nameof(products));

if (products.Count() == 0) throw new InvalidOperationException("Supply order should contain products");

var order = new SupplyOrder

{

Products = products,

State = SupplyOrderStatus.NotReceived

};

await \_supplyOrderRepository.AddOneAsync(order);

}

}

Кроме того, изменим класс SupplyOrder следующим образом:

public class SupplyOrder : Order

{

public new SupplyOrderStatus State { get; set; }}

Внесем полученные изменения в репозиторий:

git add \*

git commit -m "Created SupervisorScreen class to provide supervisor interface and changed SupplyOrder class State property to hide Order's State property"

Теперь создадим проект с библиотекой классов, который будет содержать модульные тесты для реализованной функциональности. Кроме того, установим пакеты с тестовым фреймворком и библиотеку для создания мок-объектов. Добавим новые файлы к репозиторию:

git add \*

git commit -m "Created a classlib for unit tests and installed nUnit and NSubstitute"

Для реализации модульного тестирования создадим интерфейс для реализации состояния (который будет реализовывать класс ShopState):

public interface IStoreState

{

double Funds { get; }

void Deposit(double funds);

void Withdraw(double funds);

}

public class StoreState : IStoreState

{

private static StoreState \_instance;

private StoreState()

{

//put serialization logic here

}

public static StoreState GetInstance()

{

if (\_instance == null)

\_instance = new StoreState();

return \_instance;

}

public double Funds { get; private set; }

public void Deposit(double funds)

=> Funds += funds;

public void Withdraw(double funds)

=> Deposit(-funds);

}

git add OfflineElectronicsStore

git commit -m "Created IStoreState interface and removed dependencies from concrete StoreState class"

Напишем модульные тесты для класса SellerScreen:

git add \*

git commit -m "Added unit tests for SellerScreen and updated Order class to implement Equals method"

Добавим модульные тесты для класса StorageWorkerScreen:

git add \*

git commit -m "Added unit tests for StorageWorkerScreen adn updated SupplyOrder class to implement Equals method"

Добавим модульные тесты для класса SupervisorScreen:

git add \*

git commit -m "Added unit tests for SupervisorScreen"

Теперь необходимо отправить полученные изменения в удаленный репозиторий:

git remote add origin <https://github.com/astro6703/OfflineElectronicsStore.git>

git checkout master

git merge implementing-store-model

git push --set-upstream origin master

* 1. ВЫВОД

В ходе лабораторной работы была изучена распределенная система контроля версий Git. С помощью этой системы был реализован проект оффлайн-магазина согласно варианту. Для организации разработки была создана отдельная ветка. Однако, эта ветвь была совмещена с ветвью master до того, как была помещена в удаленный репозиторий. В результате, в удаленном репозитории не отображаются сведения о данной ветке.