Технически университет – София

Факултет Компютърни системи и технологии

|  |  |
| --- | --- |
| Изготвил: | Възложил: |
| Иван Ивайлов Милев  ФК №121218022  Група 52 | ас П. Маринов |
| Дата на защита:  14.06.2021 |  |

Курсова работа

по

Програмни Среди

Тема:

Система за складова администрация

# Задание

Да се разработи система за складова администрация. Системата трябва да поддържа:

1. Приемането на продукти в склада
2. Отдаването на продукти от склада
3. Добавянето на нови потребители от администратора
4. Различни нива на достъп според типа потребител
5. Извадки за броя приети и отдадени продукти в даден период
6. Филтрации по различни критерии

За разработката на систематa трябва да се използват:

1. C# и System – различни терники и технологии от C# и System пространството
2. LINQ
3. WPF
4. XAML
5. Entity Framework
6. Различни шаблони за разработка

# Увод

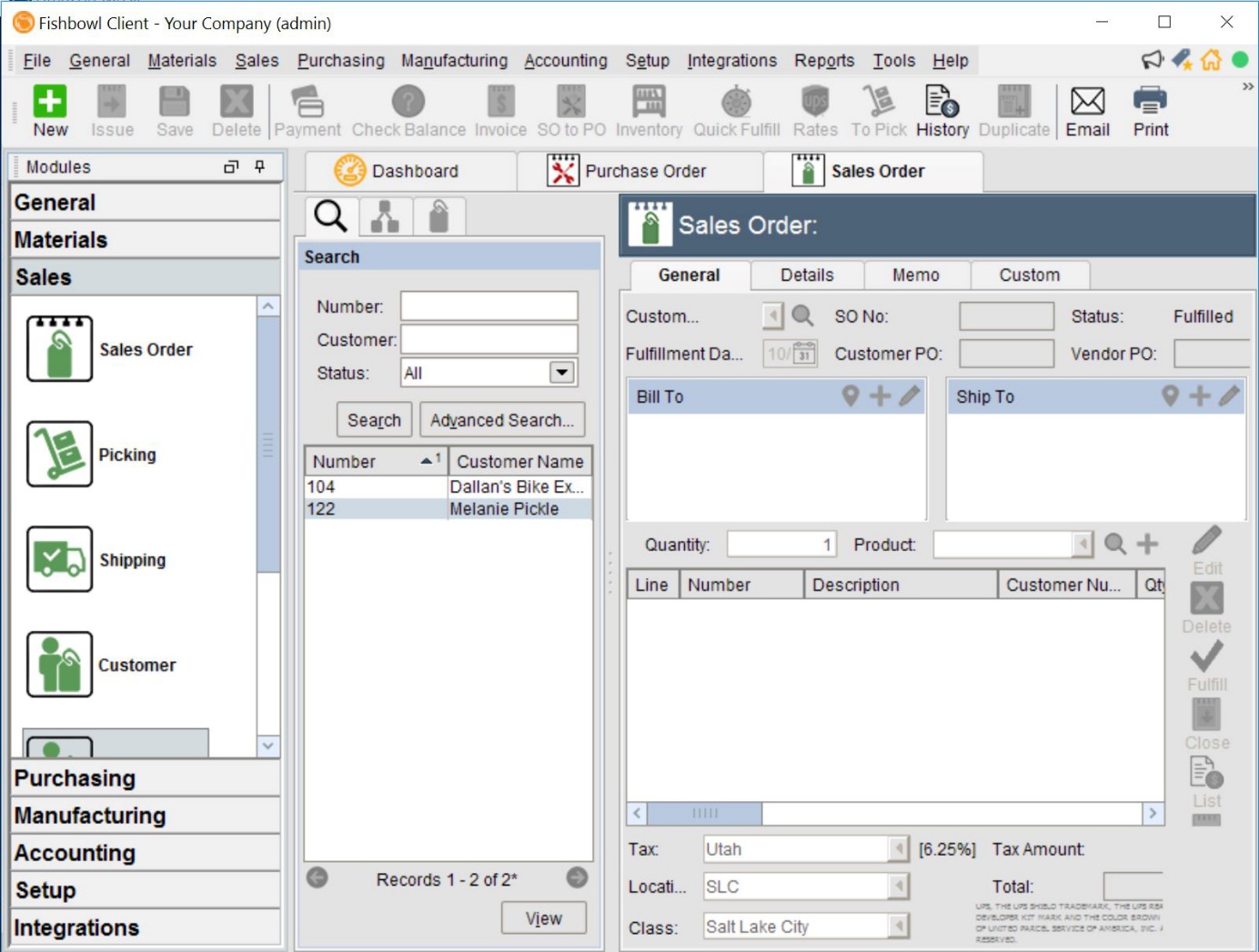
За дадения курсов проект е реализирана система за складова администрация. При малък склад нужда от такава система вероятно отсъства тъй като проследяването на каква стока има в наличност и проследяването на редките доставки не е толкова сложна дейност и може да се извършва от един човек. С нарастването на обема на получаваната и отдавана стока става по-комплексно и следенето на складовата наличност ако продължава един служител да се грижи за това нещо то той ще се превърне в пречка, която ще забавя различните доставчици затова и се поражда нуждата от система, която да позволява на доставичиците да използват системата.

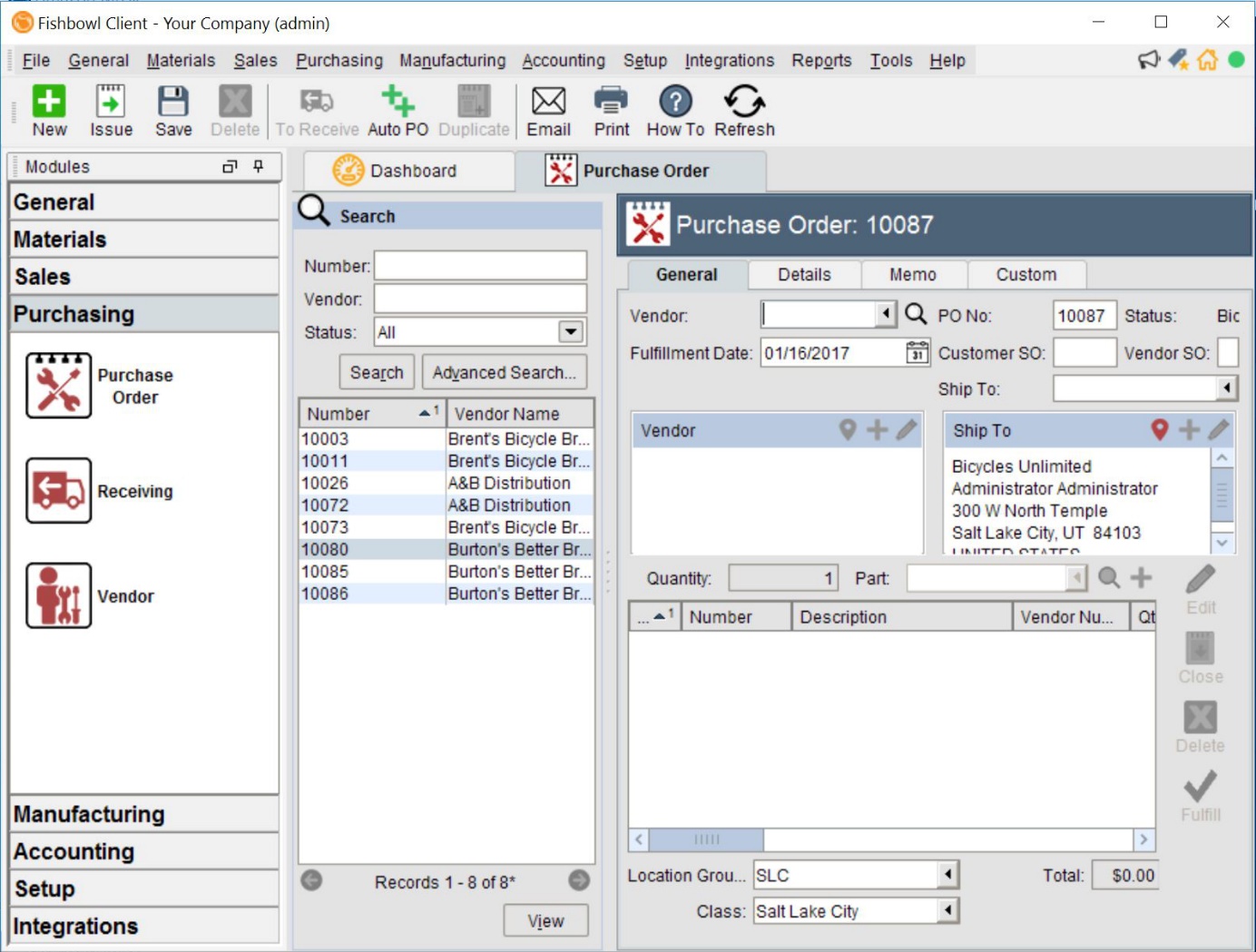
# Анализ на съществуващи разработки

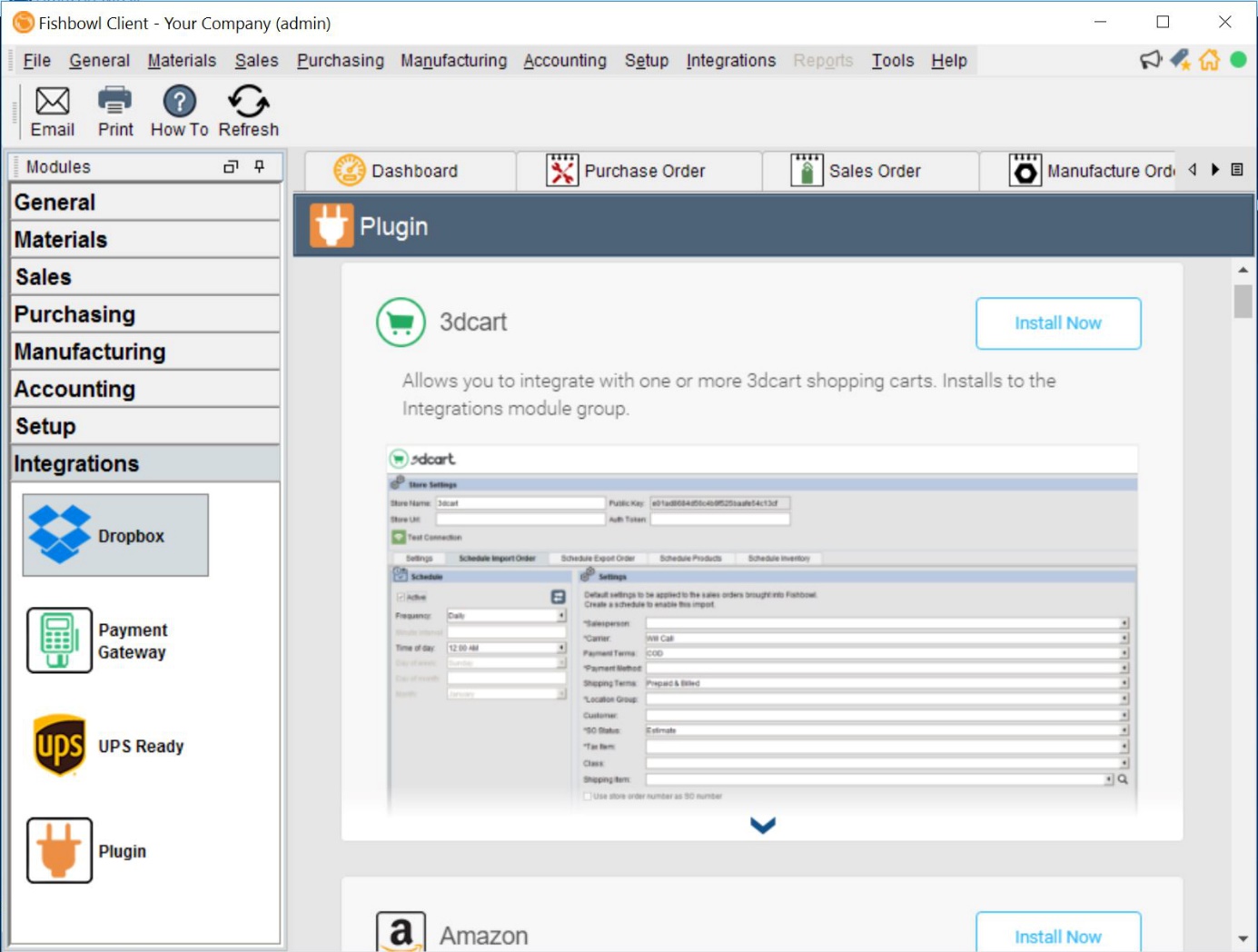
## Fishbowl Software

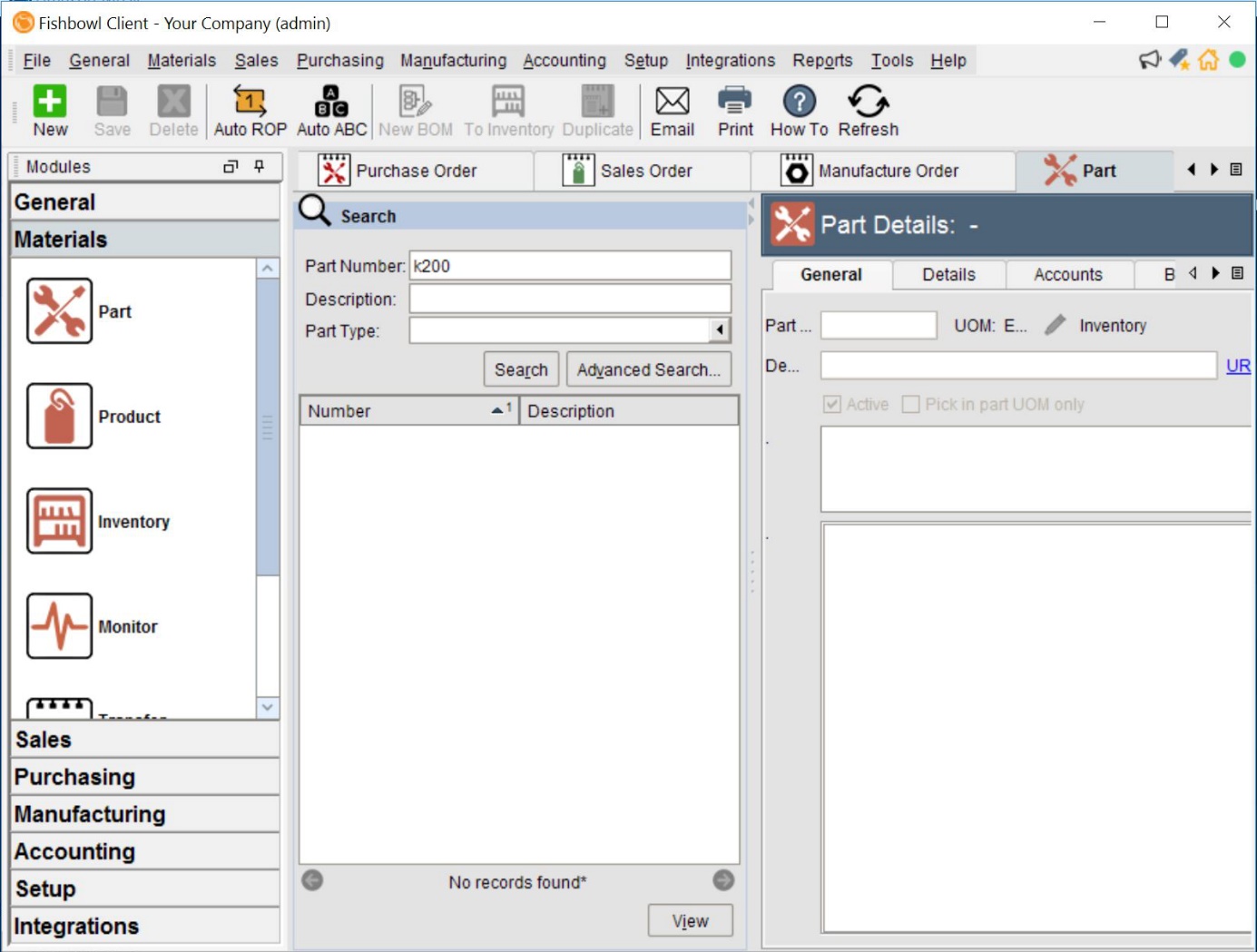
Fishbowl Software е софтуер за оправление на производство и складове, който е таргетиран към малкия и среден бизнес. Основните му функционалности са контрол на инвентара, възможност за планиране на нужни материали, контрол на различни поръчки и генериране на касов бон. Софтуерът е изключително богат на функционалност, с което идва и комплексността му при използване, за да бъде използван продукта най-вероятно ще е нужно не кратко обучение на персонала, който ще го използва.

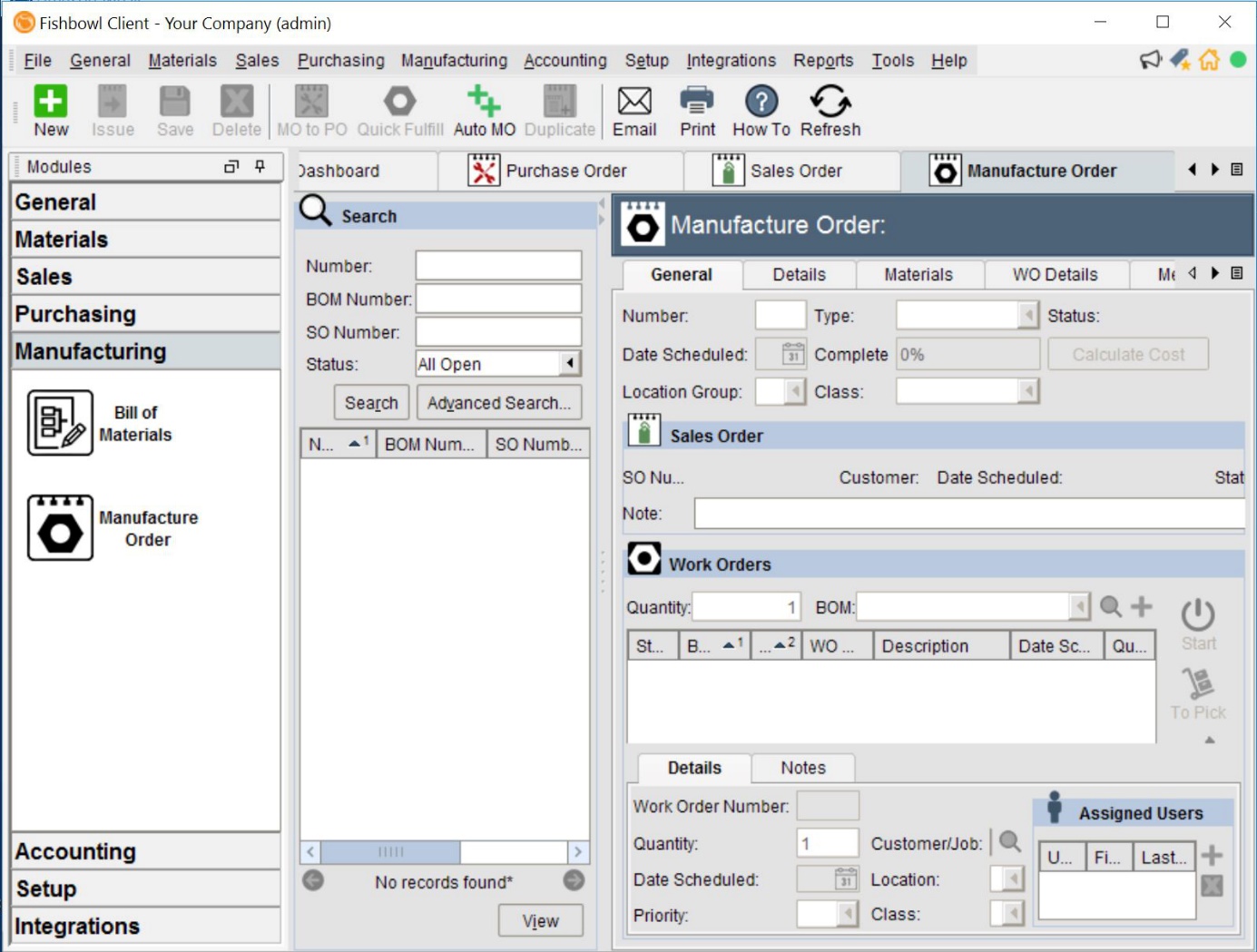
Ралични екрани от софтуера:

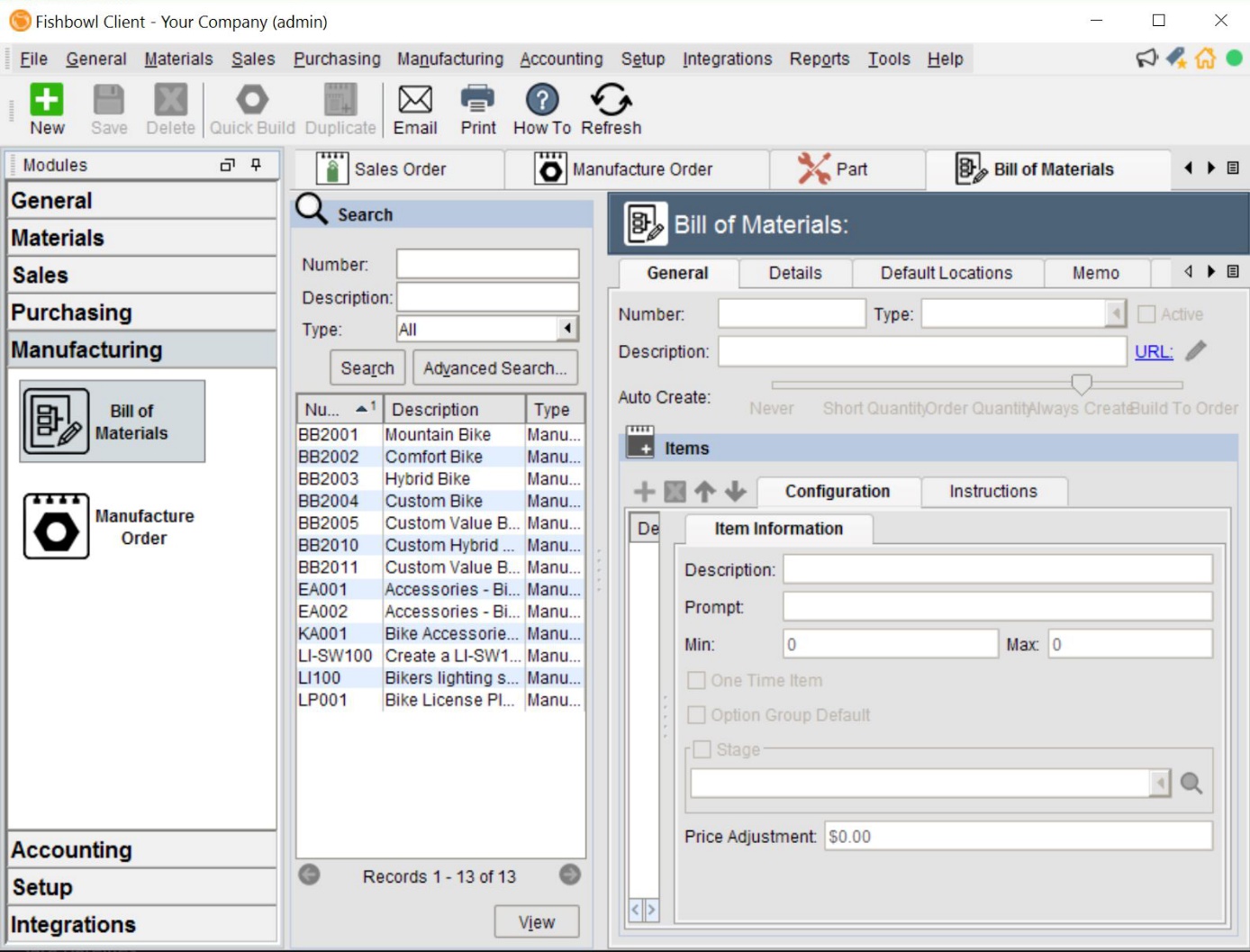








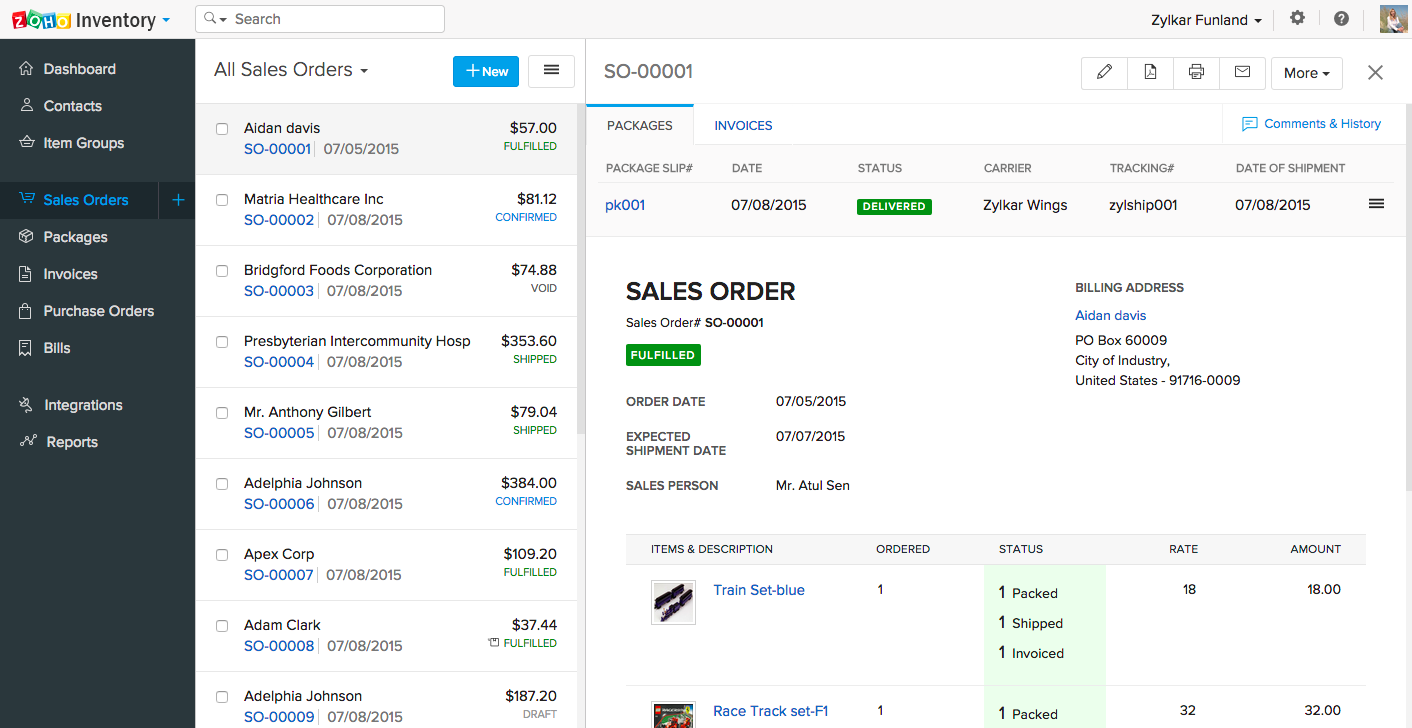


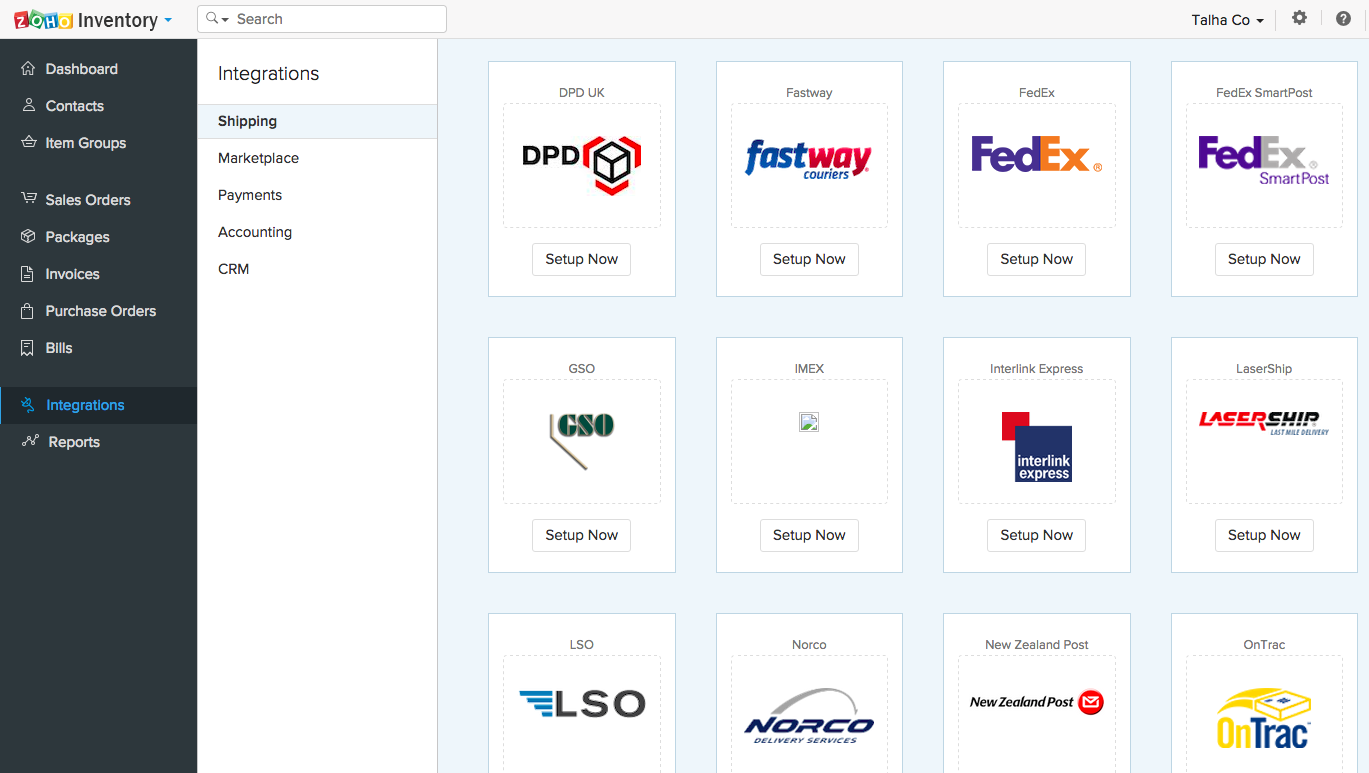


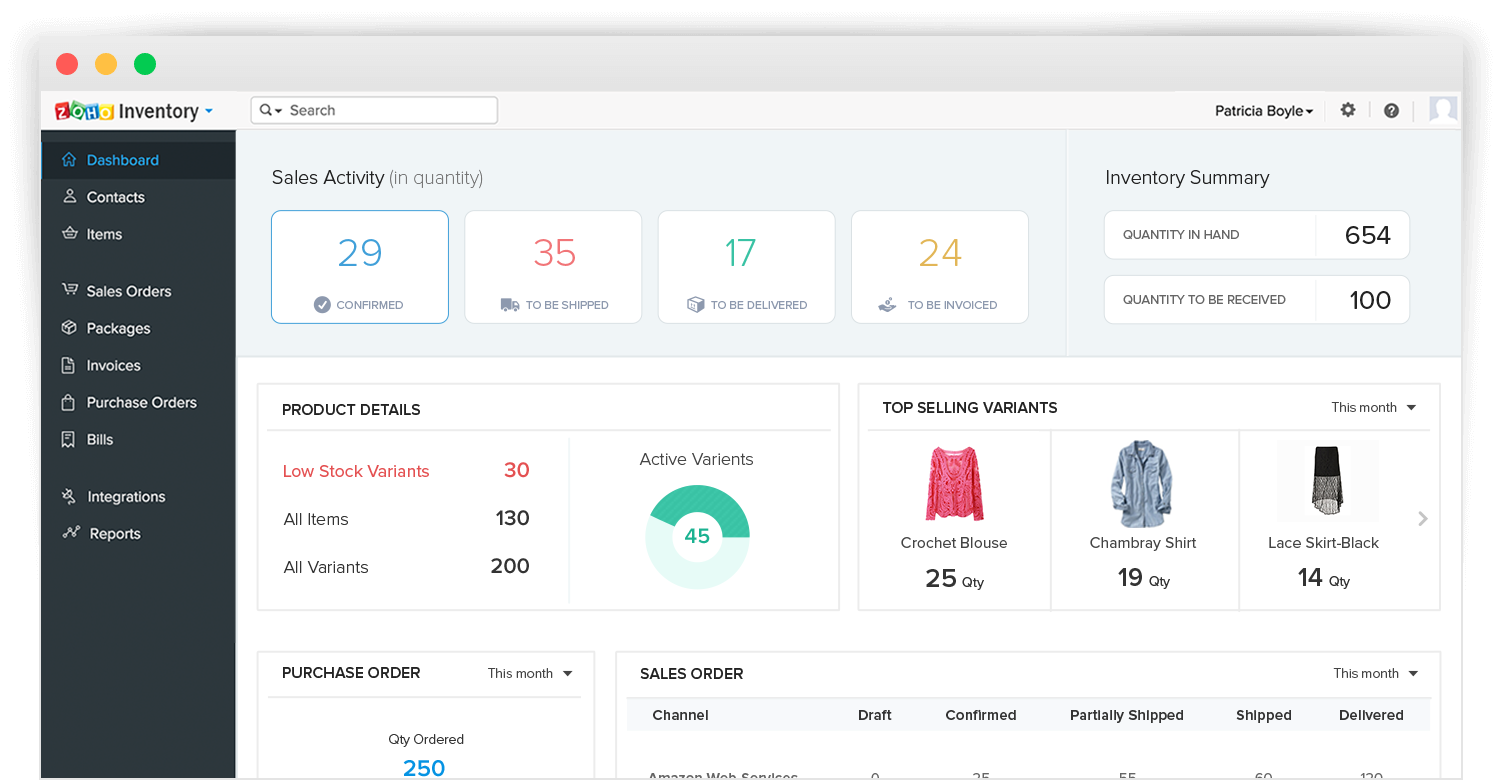
## Zoho Inventory Software

Zoho Inventory е облъчно базирано решение за оправление на инвентара – отново таргетиращо малкия и среден бизнес. Основни функционалности са вградените отчети и анализи заедно с проследяване на доставки според източника на доставка и други. Приложението също може да се използва под андроид и iOS.

Основен недостатък за това решение е нуждата от връзка с публичния интернет, за да бъде използвано.







# Проектиране

## Потребители на продукта

Основно продукта ще се използва от хора, които приемат и отдават стока на куриери. Тъй като това няма да се случва в офис трябва потребителския интерфейс да е прост с малко полета за попълване и относително големи и лесни за натискане бутони. Тъй като служителите за приемане и отдаване на поръчки пръвоначално могат да са едни и същи е полезно да могат да приемат и отдават продукти от един екран, за да не се сменят екрани и да стават обърквания също служителите могат да искат да се консултират със съществуващия инвентар затова и той е на същия прозорец като има нужните филтрации за да е удобно използването му.

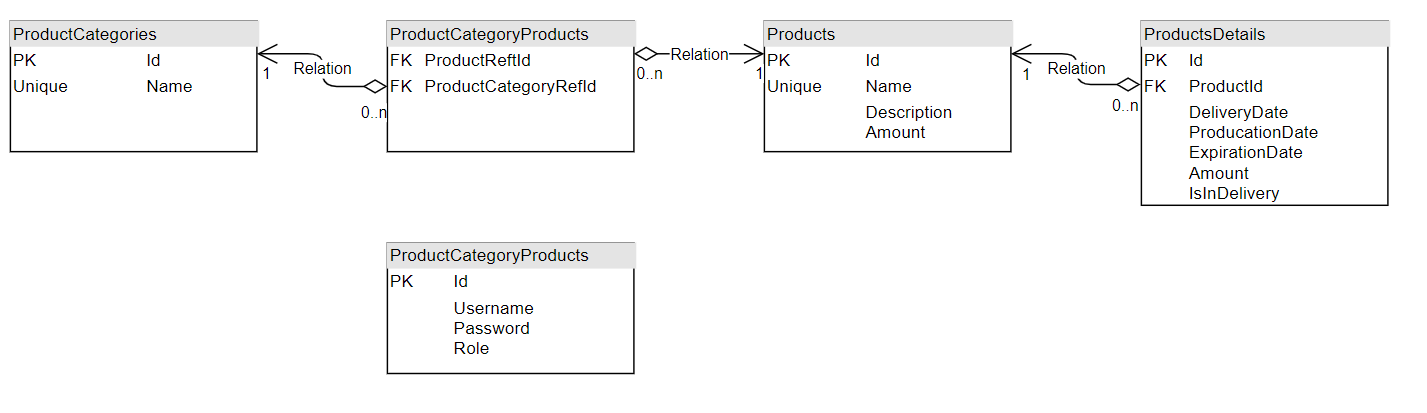
Тъй като йерархията на различните компании е различна затова има и различни категории потребители – администратор, касиер, доставчик. Като касиерите и доставчиците имат ограничени права. Нови потребителите се създават само и единствено от администратора на системата, за да няма възможност за злоупотреби от потребителите. Администраторската роля е с идеята да бъде използвана от системния администратор на бизнеса и/или собственика на бизнеса тъй като той има достъп до различни извадки, добавянето на нови категории за доставка и правата на потребителите на системата.

## Структура на запазените данни

Тъй като приложението е за складова администрация то би била подходяща следната структура на данните: Категория като в категорията би имало различните продукти и за всеки продукт има всички доставки към и от склада.

Базата от данни ще бъде популирана от доставките (към и от склада) следователно това е начална точка. Всяка доставка е за конкретен продукт следователно трябва да имаме връзката, че наличността от даден продукт се състои от множеството доставки. Различните продукти могат да бъдат категоризирани за по-лесно търсене и намиране в последствие, като един продукт разбира се може да принадлежи към повече от една категория.

### Диаграма на данните



## Описание на потребителския интерфейс

Както по-горе е описано се цели функционален и лесен за използване интерфейс, но не задължително естетически издържан.

При пускането на приложението всеки потребител ще бъде посрещнат с прозорец за вход. След като се впише успешно трябва да бъде посрещнат от меню с функционалностите на системата – Асортимент, Потребител(за администратори), Отчет(за администратори) и изход(изход от системата, за да се впишем като друг потребител).

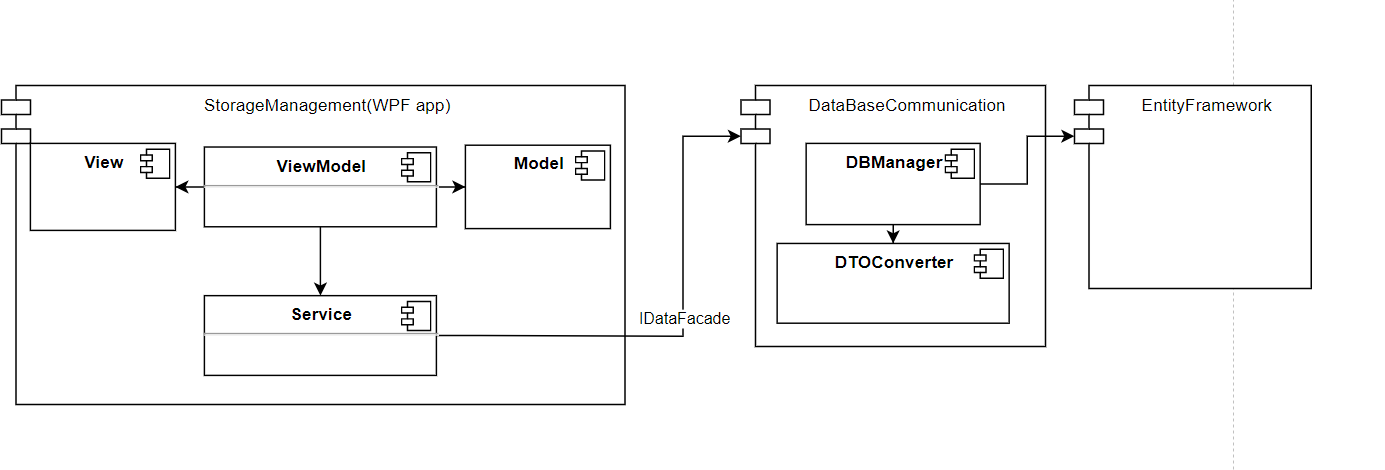
Основния прозорец за системата е Асортимент тъй като той съдържа основната функционалност на продукта. За да се визуализират данните в него ще се използва вече утвърден шаблон за UX – Master – Detail, което ще предостави на потребителите да изберат дадена категория, при което ще се появят всички продукти за тази категория след което се избира конкретен продукт и за него се появяват всички видове доставки.

Под всяка секция (категория, продукт, доставки) потребителя, ще може да добавя нови данни за конкретната секция и/или да обновява вече съществуващите.

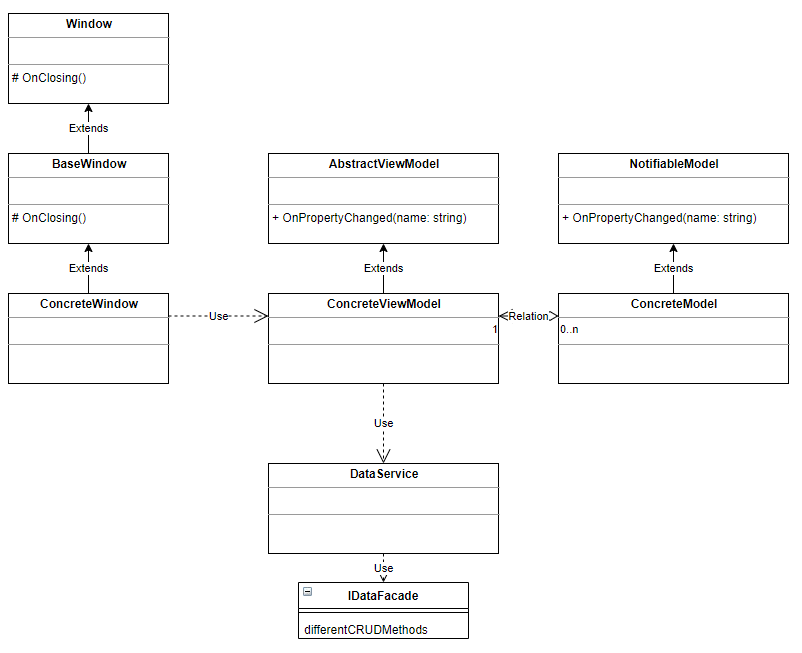
За всяка секция трябва да има и филтрация за по-удобна навигация.

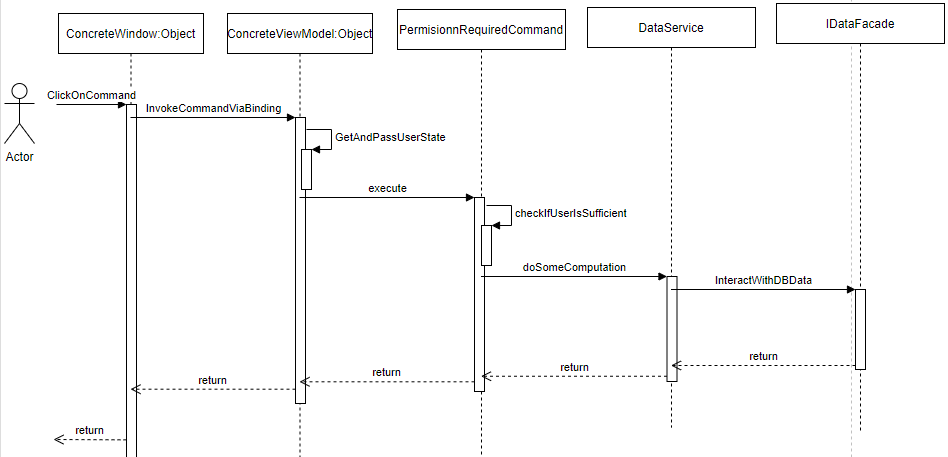
## Програмна архитектура на високо ниво

Програмната архитектура се дели на два основни проекта – StorageManagement, който е WPF проект и ще се грижи за работата с потребителския интерфейс и реализирането на различни функционалности свързани с продукта и DataBaseCommunication – който служи за връзка с базата и не дава на потребителите директен достъп до базата от съображения за сигурността от този проект по никакъв начин не излизат идентификатори от базата или пък под каквото и да е форма директни обекти свързани с базата, той енкапсулира базата.

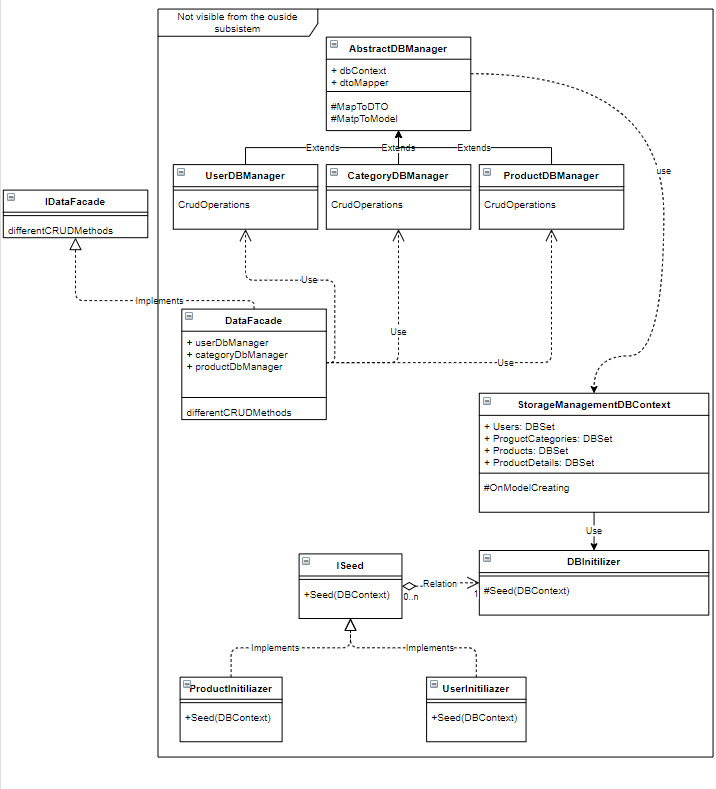


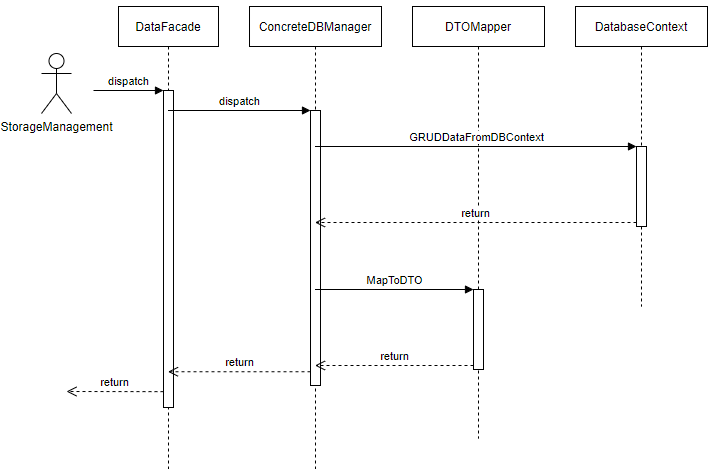
### Вътрешна структура на StorageManagement





### Вътрешна структура на DataBaseCommunication





### Технологии

#### Entity Framework

Entity Framework е софтуерна рамка (framework), която е най-известният ORM (Object relational mapping) – създава автоматично връзка между данните от базата към отговарящи на тяхната дефиниция класове и съответно създава правилно популирани обекти от съответните типове.

За работа с базата данни ще се използва Entity Framework с похвата на дефиниране на структурата на базата чрез код (Code first). Този начин позволоява да нямаме никакъв SQL код, което от своя страна значи по-малко технологии и скриптове, които трябва да се поддържат.

#### Ninject

Ninject е една от по-известните и големи библиотеки, която се грижи за инжектирането на зависими обекти едни в други.

В конкретната задача има два графа на зависимостите – първият от които се строи при започването на програмата. Този граф съдържа всички типове, които се иползват в StorageManagement WPF приложението и се грижи за правилното популиране на зависимостите в различните обекти. Вторият граф в приложението започва с инстанцирането на обект от типа IDataFacade – той се грижи за всички зависимости в DataCommunication и започва и свършва своя живот със създаването и унищожаването на имплементацията на IDataFacade.

### Шаблони за дизайн и архитектурни шаблони

#### MVVM (ModelViewViewModel)

За по-голямата част от екраните на приложението е изполван шаблона на Модел-Изглед-ИзгледМодел. По този се получава добро разделение на потребителския интерфейс от логиката зад него и все пак лесен начин да се променя презентационното ниво от това с логиката.

#### Сек - Singleton

Този шаблон е изключително използван в даденото приложение, тъй като повечето инстнции на класове трябва да са задължително една и съща или няма нужда да са повече от една. Шаблонът осигурява, че за даден тип има само една инстанция в случая на разработката на това приложение тъй като Ninject се грижи за създаването на различните обекти то той се грижи и за това кои да са Singleton.

#### Обекти за трансфер на данни - Data Transfer Object

Налагането на този шаблон идва от нуждата DataCommunication и StorageManagement да си комуникират. Тези класове нямат никаква логика в себе си освен такава обвързана със задаване на стойности на различните си полета.

#### Инжектиране на зависимост - Dependency Injection

Този шанлон се използва за полесното разрешаване на Inversion of Control, защото автоматично инстанцира типовете, на които зависят различните класове. Класовете винаги очакват тези типове през конструктура си, за да няма силна връзка между тях и конкретни имплементации.

#### Шаблон фасада - Facade Pattern

Идеята на фасадата е да замаскира структурата на подсистема за нейния ползвател като дава един интерфейс (фасада), която да бъде използвана. В случая на програмата има фасада, която да прикрие цялата работа с базата за WPF приложението и му предоставя един интерфейс, с който WPF приложението може да прави всичко, което му трябва свързано с базата данни.

# Реализация

## Реализация на WPF приложението

Приложението е реализирано на основата на WPF прозорци:

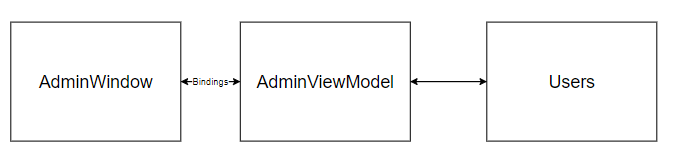
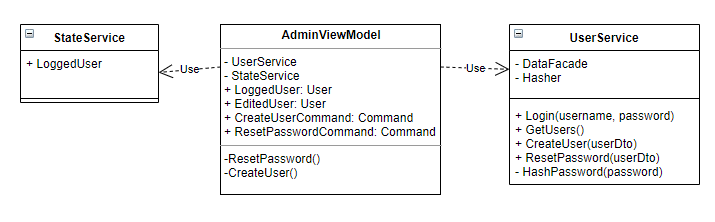
AdminWindow, MenuWindow, ProductWindow, ReportsWindow, LoginWindow

Като за структурата на всеки от тях е използван Grid. Тъй като има преизползвана логока за връщането назад към менюто в „подпрозорците“ е създадена и една контрола, която да обслужва тази логика – LoggedUserControl, която приема един параметър, а именно потребителското име на потребителя, който е активен в приложението.

### Реализация на MVVM

Прозорците AdminWindow, ProductWindow и ReportsWindow използват MVVM архитектурата, за да работят като всяка от тях използва нужните команди с съответните атрибути, за да ограничава достъпа на различните потребители. Всеки от конкретните MVVM имплементации отговаря за повече от един модел поради комплексността на съответните прозорци:

AdminWindow

Интересна е имплементацията на ограничения достъп според роля. Начина по който е релаизирано е чрез атрибути и специална команда наречена PermissionRequiredCommand

Имплементацията е следната:

class PermissionRequiredCommand : ICommand

{

public event EventHandler CanExecuteChanged;

private readonly Action action;

private readonly RoleDTO userRole;

public PermissionRequiredCommand(Action action, RoleDTO userRole)

{

this.action = action;

this.userRole = userRole;

}

public bool CanExecute(object parameter)

{

return HasPermission();

}

public void Execute(object parameter)

{

action();

}

private bool HasPermission()

{

var methodAttributes = action.Method.GetCustomAttributes(true);

if (methodAttributes.Length == 0)

{

return true;

}

return methodAttributes.Any(attr => (attr is PermissionRequired) &&

(attr as PermissionRequired).RequeredRoles.Where(rr => rr == userRole).Any());

}

}

От представения фрагмент код се вижда, че командата се създава чрез делегат, който реално изпълнява логиката на командата. За да определим дали конкретната команда е достъпна за потребителя се взима атрибутите, които са подадени на делегата, който трябва да изпълни командата и се сравнява дали подадената рола на текущия потребител отговаря на достъпа, който е определен от параметрите на атрибута.

Фрагмент на команда с ограничение при използването както и имплементацията на самия атрибут:

[AttributeUsage(AttributeTargets.Method, AllowMultiple = true)]

class PermissionRequired : Attribute

{

public PermissionRequired(params RoleDTO[] requeredRoles)

{

RequeredRoles = requeredRoles;

}

public RoleDTO[] RequeredRoles { get; }

}

[PermissionRequired(RoleDTO.Admin)]

private void ResetPassword()

{

userService.ResetPassword(EditedUser);

}

Организацията и имплементацията на останалите прозорци е по подобен модел. По-интересно е това, че когато се използва контролата LoggedUserControl тя трябва да отвори прозореца на менюто, до което няма достъп и не може да бъде подадено чрез data binding, за целта menu-то както и други прозорци не се затварят, а само крият (така се печели и бързодействие). Когато трябва да се отвори менюто контролата ще потърси в прозорците на приложението и ще го покаже от там:

foreach(Window window in Application.Current.Windows)

{

if(window is MenuWindow)

{

window.Show();

break;

}

}

Window.GetWindow(this).Hide();

За жалост криенето на прозорци води до нов проблем при затварянето на активния прозорец (това е и единственият видим за потребителя) приложението не се затваря тъй като другите прозорци са само скрити, но не и спрени. Затова се налага въвеждането на нов абстрактен прозорец, който да се грижи да затвори не само себе си но и всички останали прозорци – BaseWindow.cs

public class BaseWindow : Window

{

protected override void OnClosing(CancelEventArgs e)

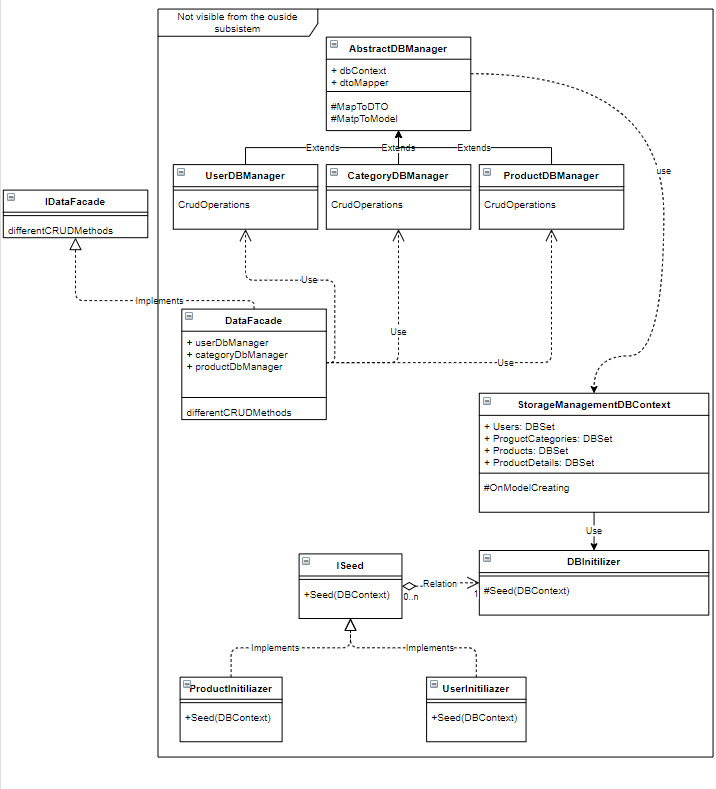
{

Application.Current.Shutdown();

}

}

## Реализация на DataCommunication:



Предложената инфрастуктура отговаря и на реализцаията на DatabaseCommunication проекта. Цялата идея е да се енкапсулира работата с базата данни от WPF приложението, което да работи само и единствено с DTO обекти, които се предоставят от този проект.

### Реализация на Façade

Както се вижда и от диаграмата по-горе цялата имплементация на DatabaseCommunication се крие под една фасада. Този шаблон е използван в случая тъй като за проектите, които използват DatabaseCommunication не е важно и нужно да знаят вътрешната структура на проекта, за тях единствено е важен интерфейсът, с който да комуникират с базата. По този начин се енкапсулира цялата подсистема и потребителите не могат да работят директно с обекти на базата, което предпазва и от различни злоупотреби с нея. Комуникацяита става само и единствено през фасадата и DTO обекти. Всички други членове на проекта са видими само в него.

Цялата под система се обосновав и на собствен граф на зависимостите и е независима от своите ползватели както и би се очаквало да бъде.

### Реализация на Data Transfer Object

За реализацията на DTO се използва AutoMapper, който живее и се използва на нивото на мениджърите, които застават между фасадата и базата(entity framework). Те имат за цел да извършват нужните действия с базата и след това да конвертират резултата си към обекти за трансфер на данни.

Идеята за мениджърите е замислена с цел да има по един такъв за всяко entity от базата затова има и следната абстракция, която определя обръщането на entity в DTO и обратно:

abstract internal class AbstractDbManager<DbModel, DTO>

{

protected StorageManagementDBContext dbContext;

protected IMapper dtoMapper;

public AbstractDbManager(StorageManagementDBContext dBContext, IMapper dtoMapper)

{

this.dbContext = dBContext;

this.dtoMapper = dtoMapper;

}

protected DTO MapToDTO(DbModel model)

{

return dtoMapper.Map<DbModel, DTO>(model);

}

protected DbModel MapToModel(DTO model)

{

return dtoMapper.Map<DTO, DbModel>(model);

}

}

Всяка конкретна имплементация на DBManager ще даде типовете, с които работи както следва

internal class CategoryDBManager: AbstractDbManager<ProductCategory, ProductCategoryDTO>

В конкретния пример знаем че CategoryDBManager-а работи с ProductCategoryDTO и ProductCategory и във всичките му методи може лесно да се извика: MapToDTO(productEntity) и обратно MapToModel(productCategoryDTO)

### Реализация на Dependency Injection, Singleton

За реализацията на инжектирането на зависимостите е използвана библитеката Ninject, като в приложението има два графа на зависимостите – един който се създава при стартирането на приложението и е обвързан с всичко в WPF апликацията и втори, който се създава при инициализацията на DataFacade (който се инстанцира от WPF апликацията). Всички Service-и са singleton-и (т.е. имат само една единствена инстация през живота на приложението) това отново се постига с помощта на Ninject. Начина по-който се работи с Ninject е чрез регистриране на всички типове, които трябва да инстанцира Ninject.

Регистрациите за Ninject в приложението са както следва:

class SecondPartyModule : NinjectModule

{

public override void Load()

{

Bind<IDataFacade>().To<DataFacade>().InSingletonScope();

}

}

class ServicesModule : NinjectModule

{

public override void Load()

{

Bind<UserService>().ToSelf().InSingletonScope();

Bind<StateService>().ToSelf().InSingletonScope();

Bind<ProductService>().ToSelf().InSingletonScope();

Bind<ReportService>().ToSelf().InSingletonScope();

}

}

public partial class App : Application

{

private IKernel kernel;

protected override void OnStartup(StartupEventArgs e)

{

base.OnStartup(e);

kernel = new StandardKernel(new SecondPartyModule(), new ServicesModule());

Current.MainWindow = kernel.Get<LoginWindow>();

Current.MainWindow.Show();

}

}

За DatabaseCommunication е следното:

internal class DataManagerModule : NinjectModule

{

public override void Load()

{

Bind<IMapper>().ToConstant(CreateDTOMapper());

Bind<StorageManagementDBContext>().ToSelf().InSingletonScope();

Bind<UserDBManager>().ToSelf().InSingletonScope();

Bind<ProductDBManager>().ToSelf().InSingletonScope();

Bind<CategoryDBManager>().ToSelf().InSingletonScope();

}

private IMapper CreateDTOMapper()

{

var config = new MapperConfiguration(cfg =>

{

cfg.CreateMap<User, UserDTO>().ReverseMap();

cfg.CreateMap<Product, ProductDTO>().ReverseMap();

cfg.CreateMap<ProductDetails, ProductDetailsDTO>().ReverseMap();

cfg.CreateMap<ProductCategory, ProductCategoryDTO>().ReverseMap();

});

return config.CreateMapper();

}

}

internal class InitializersModule : NinjectModule

{

public override void Load()

{

Bind<ISeed>().To<UserInitializer>();

Bind<ISeed>().To<ProductInitializer>();

Bind<DBInitializer>().ToSelf();

}

}

public class DataFacade : IDataFacade

{

private readonly UserDBManager userManager;

private readonly CategoryDBManager categoryManager;

private readonly ProductDBManager productManager;

public DataFacade()

{

IKernel kernel = new StandardKernel(new DataManagerModule(), new InitializersModule());

userManager = kernel.Get<UserDBManager>();

categoryManager = kernel.Get<CategoryDBManager>();

productManager = kernel.Get<ProductDBManager>();

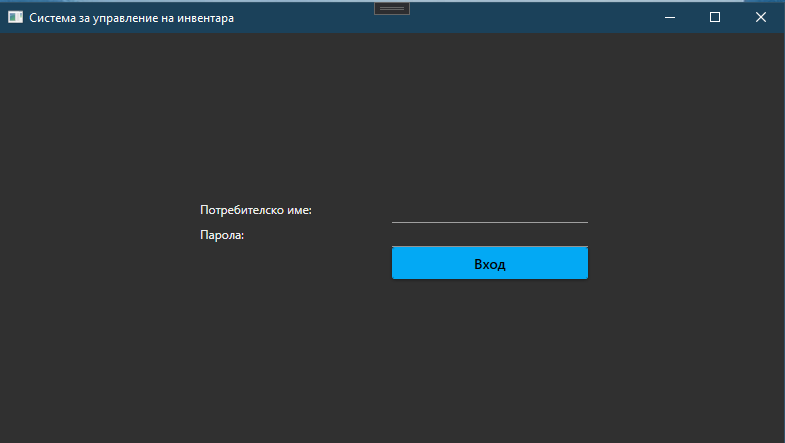
}

....

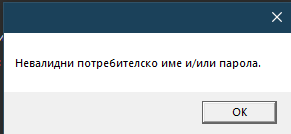
}

# Потребителско ръководство

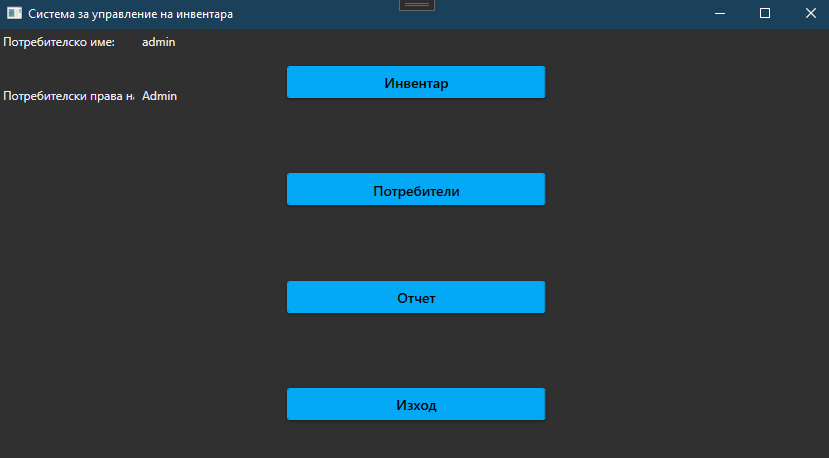
След като потребителя включи приложението е посрещнат от прозорец за вход:



При грешно въведени име и парола излиза съобщение за грешка:

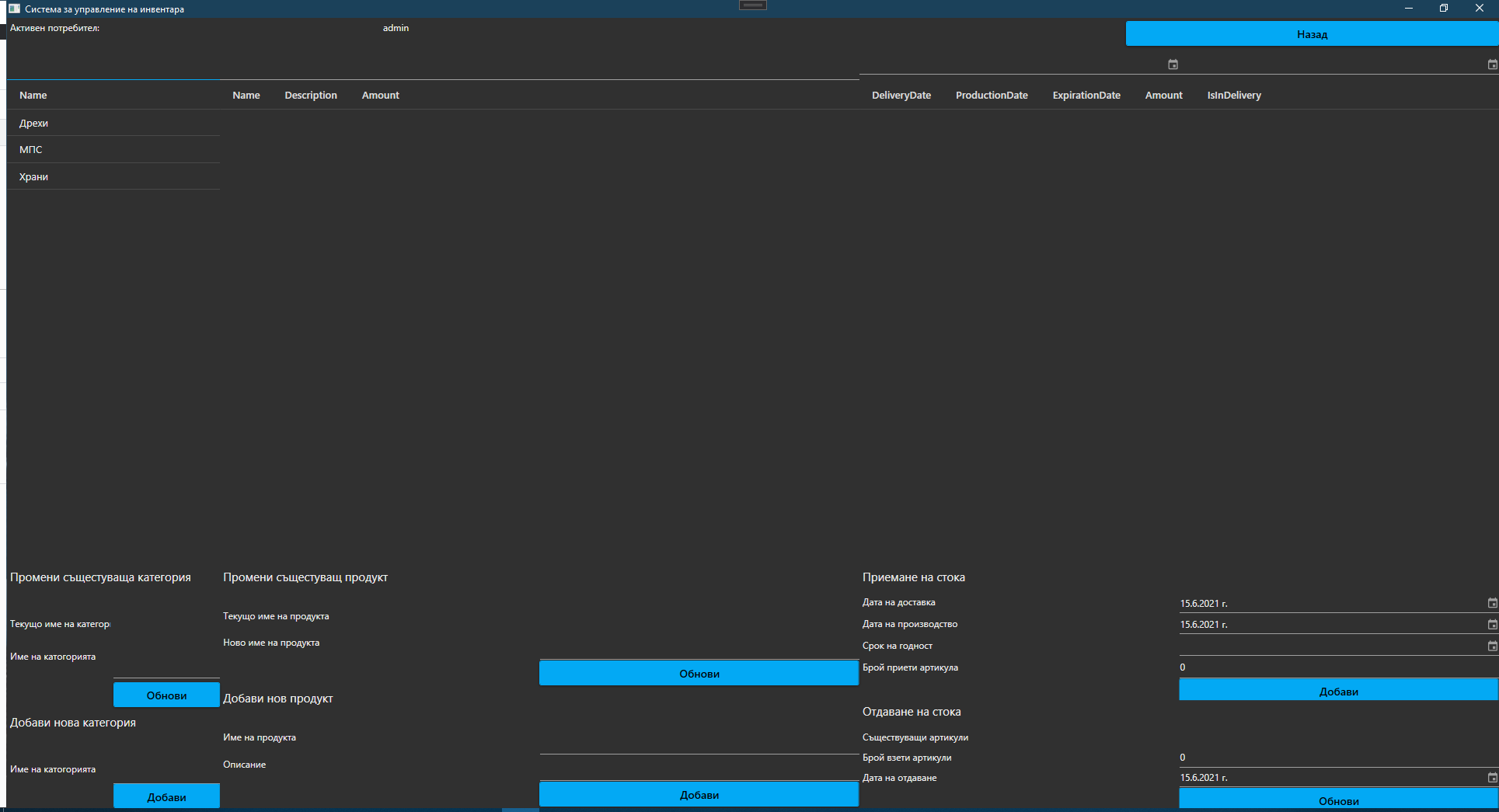


След успешно вписване в системата за потребителя се отваря нов прозорец:



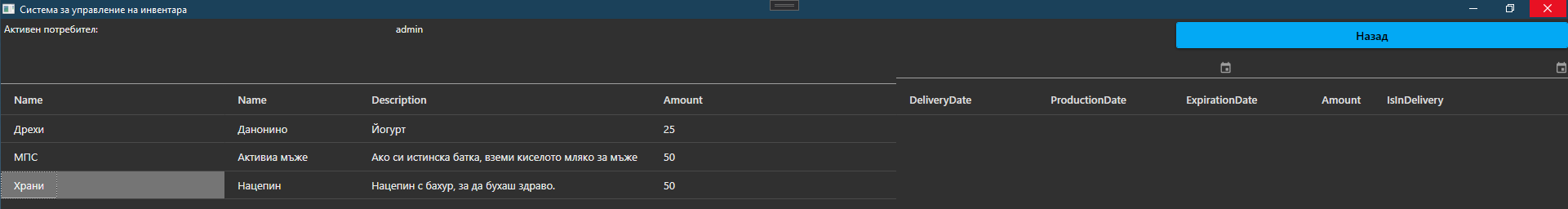
От този прозорец потребителя може да избере дали иска да излезе отново към входния екран – изход или да отовори някой от другите екрани – Инвентар, Потребители и отчет

Прозореца инвентар:

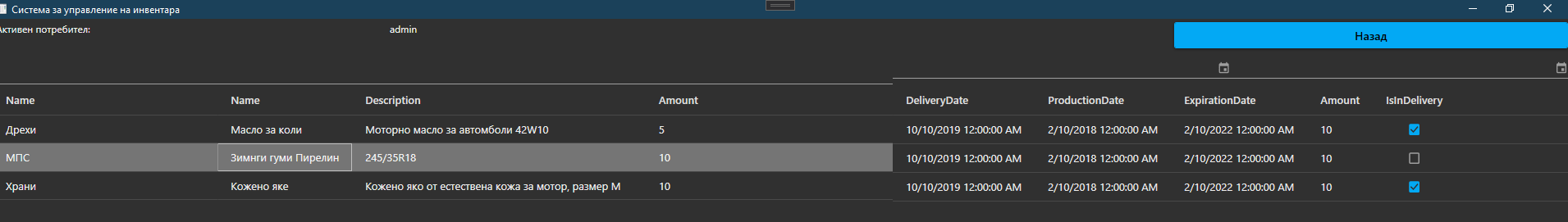


На този прозорец се вижда инвентара разделен по категории – продукти – доставки. Като в менюто отдолу могат да се добавят и променят категории, продукти и доставки.

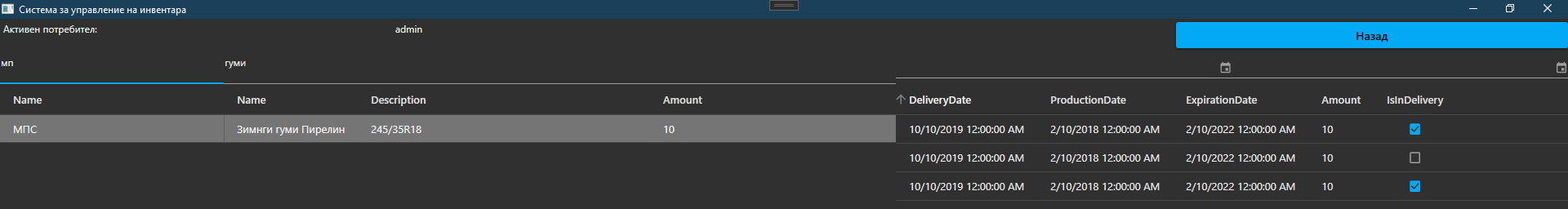
За да се отворят продуктите от дадена категория тя се избира:



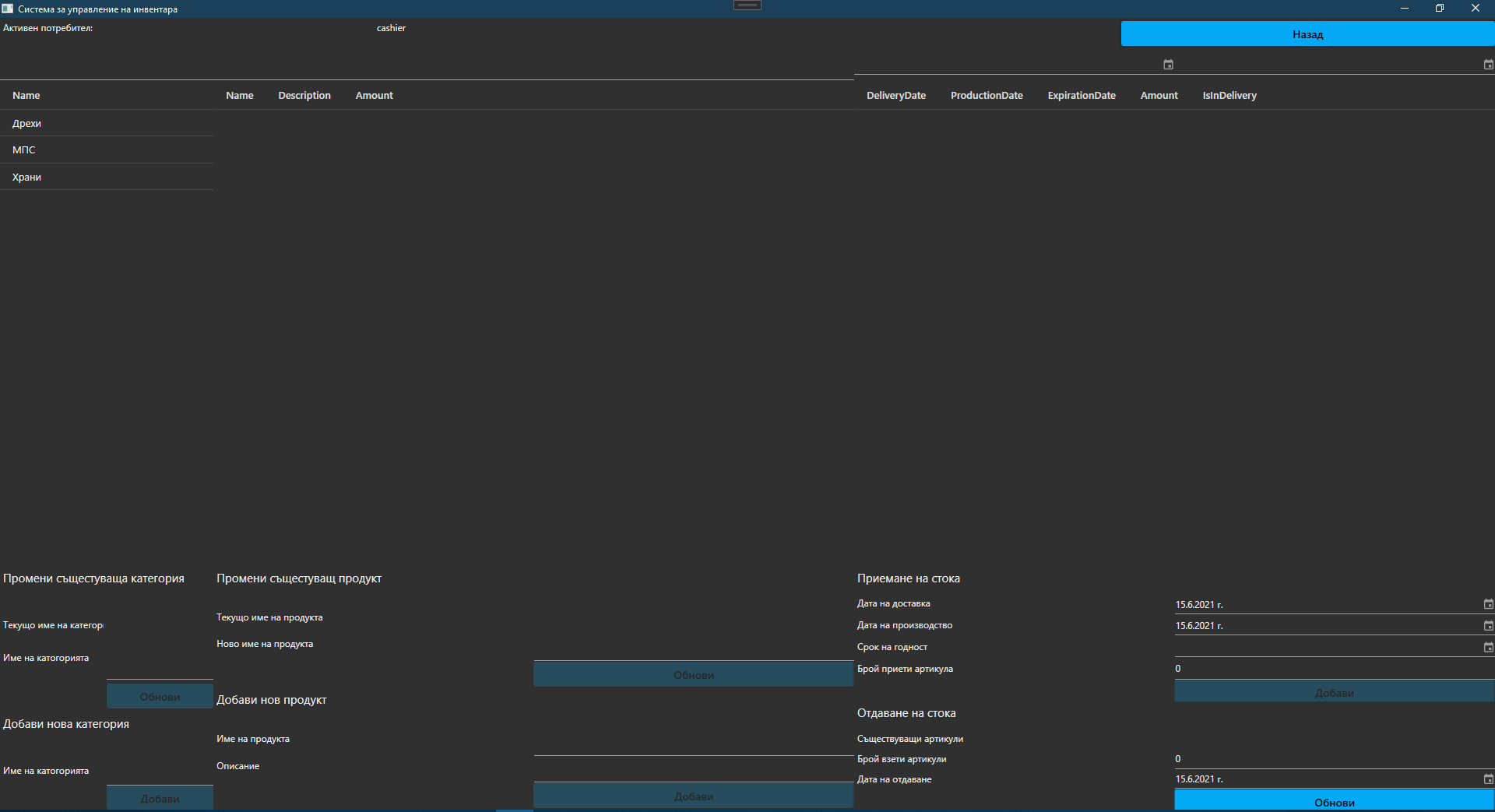
След това, за да изберем конкретен продук и да видим какви са неговите доставки избирам конкретен продукт:

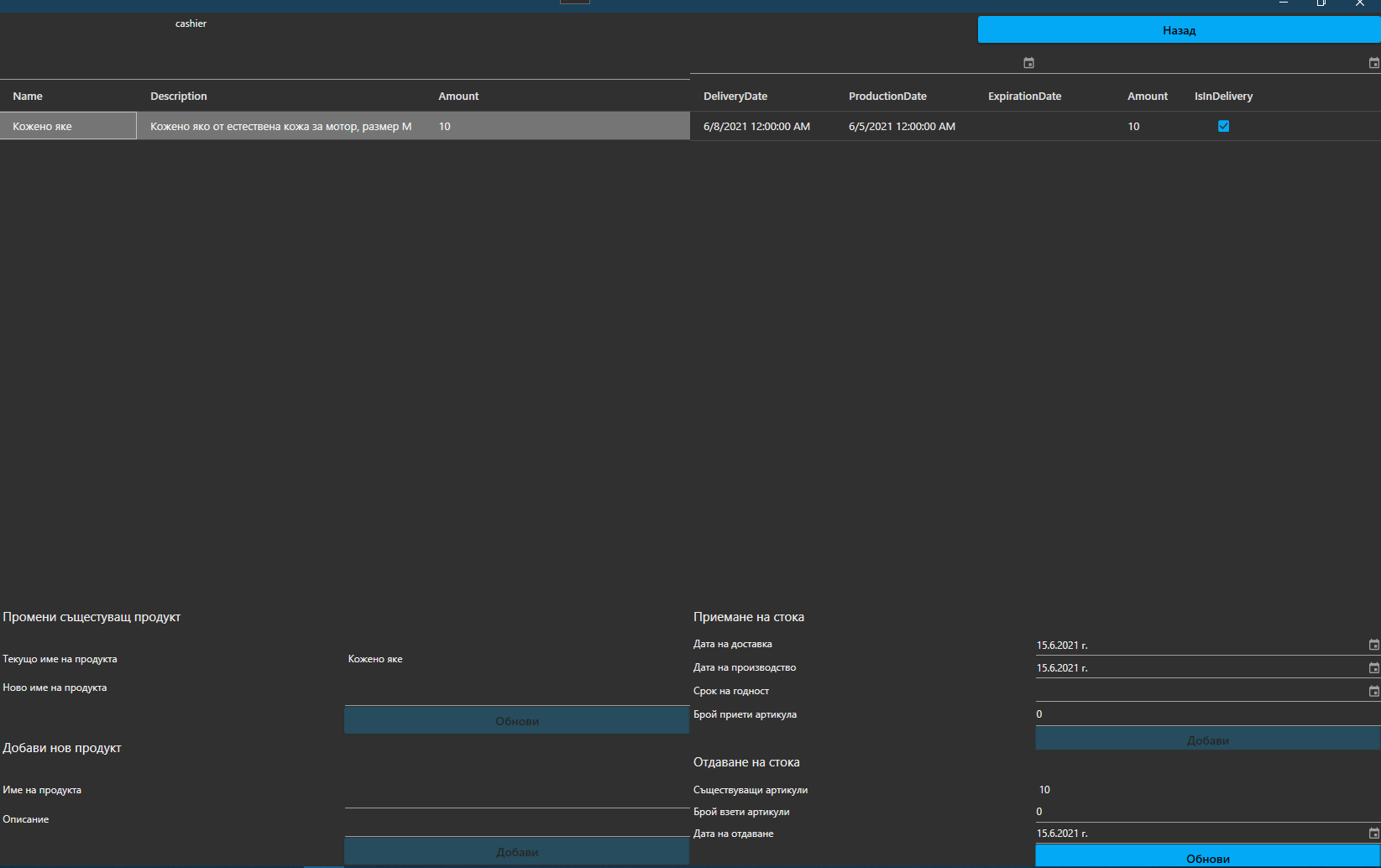


Всичките колони в таблицата са сортируеми при натискане на името на колоната, а над имената има ръчни сортирания по име и дата за доставките:



Ако сме влезли с потребител, който няма всички права например с касиер то не цялата функционалност ще е достъпна за нас и те ще бъдат деактивирани:

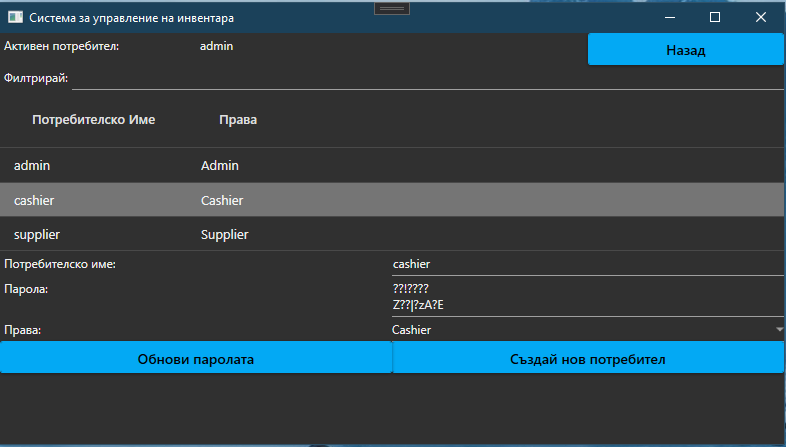


За да отдаваме стока трябва да сме избрали конкретен продукт, от който ще се отдава тогава и полето за съществуващи артикули ще бъде не нулево:  


С бутона назад се връщаме към менюто.

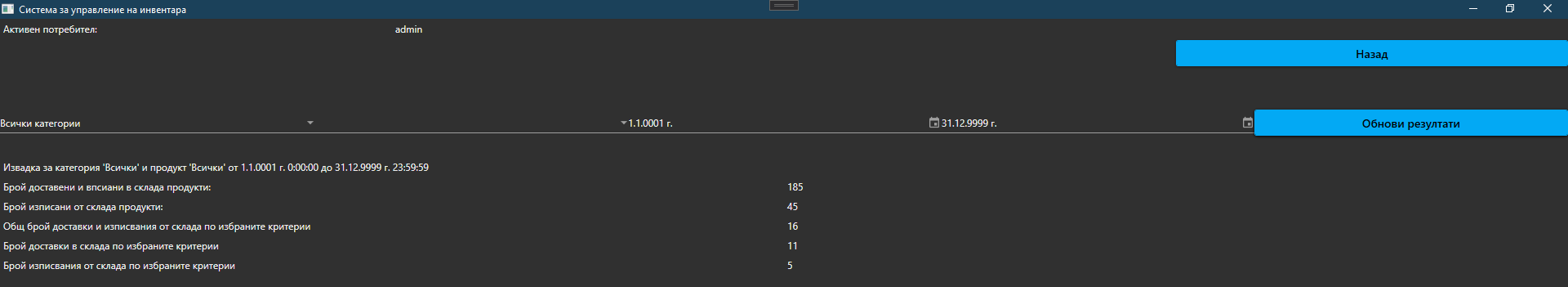
Екрана за потребители, той е създаден за администратори на системата и само за него ще бъдат активни функционалностите на прозореца.

От този прозорец могат да се създават нови потребители и да се сменя паролата на потребители, които са си я забравили. Паролата се хешира и чужди потребители не могат да видят паролата на съществуващите. Това е и единственото място, от което се създават потребители.

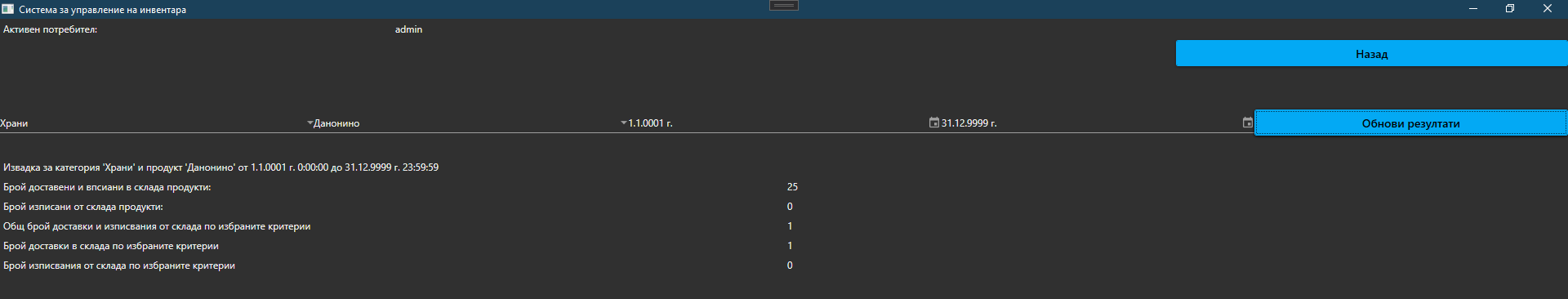


С назад отново се връща към основното меню.

Последният прозорец е прозореца Отчет:



В този прозорец може да се извадят данни за конкретни продукти и/или категории в определени граници от дати. След избор на категория във второто падащо меню ще излизат само конкретните продукти за тази категория.



И отново с бутона назад се връща в основното меню.

# Заключение

Продуктът е в състояние, което позволява използването му от потребителите, всички начални изисквания са изпълнени. Има функционалността за приемане и отдаване на продукти с различни нива на достъпа за различните потребители, съществуват различни филтрации и генерирането на отчети за дадени периоди и критерии.

За бъдещо развитие към продукта могат да се добавят съхранение на сегашното състояние на склада към файлове с формат .csv и .xlsx същото може да се добави и за създаването на отчет. Към пазената информация за различните продукти трябва да се добави и обем заето място в склада тъй като към момента тази функционалност отсъства.

# Литература

<https://docs.microsoft.com>

Design Patterns: Elements of Reusable Object-Oriented Software – Erich Gamma, Richard Helm, Ralph Johnson, John Vissides

# Приложение

Кодът за приложението както и презентацията и документацията изготвени за този проетк могат да бъдат намерени в хранилщето за код на този адрес –<https://github.com/ivanmilevtues/StorageManagement>