Технически университет – Варна

Факултет по изчислителна техника и автоматизация

Компютърни науки и технологии

Софтуерни и интернет технологии

СКЛАД С НАЛИЧНОСТИ

Тодор Кирилов Пенчев – 20621541

Момчил Антонов Милков - 20621535

Съдържание

1. Задание на проекта

Да се разработи информационна система, предоставяща услуга склад. Програмата съхранява и обработва данни за складови помещения. Системата позволява множествен достъп.

Системата поддържа два вида потребители администратор и оператори (складов агент) с различни роли за достъп до функционалностите в системата.

Операции за работа с потребители:

- Създаване на складови оператори от администратор;
- Създаване на доставчици;
- Създаване на клиенти;
- Създаване на каса (Парична наличност).

Системата поддържа операции за работа със събития:

- Създаване на номенклатури;
- Работа с фактури
    - Приемане на стока от доставчик на доставна цена;
    - Изписване на стока на продажна цена;
- Наблюдение за наличност на стоки в склада;
- Наблюдение за наличност на пари в касата;

Системата поддържа справки по произволен период за:

- Доставки и доставчици;
- Изписване и клиенти;
- Дейност на складовите оператори;
- За наличности в склада;
- Разходи,приходи,печалба.
- Движение на наличността в касата.

Системата поддържа Известия за събития:

- Критичен минимум и липса на стока;
- Критичен минимум и липса на парична наличност;

2. Анализ на проблема

2.1 Функционални изисквания

Системата трябва да поддържа два типа акаунти – администратор и оператор, като първият има повече права и може да прави всичко, което може и оператора. Системата идва с администраторски акаунт по подразбиране с потребителско име **admin** и парола **Admin123**, чрез който се осъществява първоначалното влизане в програмата.

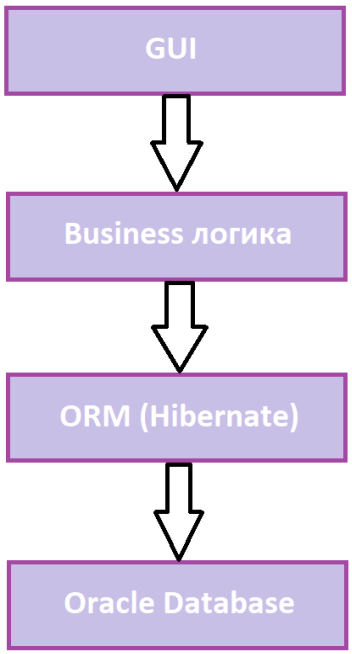
След успешна автентикация потребителя има достъп до функционалността предоставена от приложението. Опциите за работа са следните:

- **Създаване на нови потребители (оператори или администратори)** – за всеки нов потребител трябва да бъдат въведени валидно потребителско име, което не е вече заето и парола, която трябва да е поне 8 символа, съдържаща поне една главна и поне една малка буква
- **Създаване на партньори (доставчици и клиенти)** – За тях трябва да бъде въведено уникално име на фирма, валиден имейл адрес и валиден български мобилен телефонен номер
- **Създаване на каса** – тъй като проектът обслужва нуждите само на един склад, създаването на каса е процес, който се извършва еднократно и в базата данни се съхранява **само една** каса. Преди създаване на каса няма как да бъдат извършени част от операциите в приложението, зависещи от финансови операции свързани с покупко-продажбата на стоки.
- **Създаване на номенклатури** – всяка номенклатура представлява стока, която складът има в наличност, е имал или ще има в бъдеще. Изтриването на номенклатури не е осъществено, тъй като то е свързано с вече създадените фактури и при изтриване на номенклатура, би се загубила информация какво е било доставено в склада или изписано от него за дадена фактура. За всяка номенклатура трябва да бъде въведено уникално име, валидна цена (неотрицателна), валидно количество (може да е 0, но не и по-малко) и валидно минимално количество
- **Работа с фактури** – Създаване на фактура за приемане или изписване на стоки. За създаването на фактура е необходимо да се посочат партньор, дата на фактурата, тип на фактурата (доставка или изписване) и стоките заедно с техните количества, за които се отнася фактурата
- **Наблюдение за наличност на стоки в склада** – при количество под минималното за съответната стока излиза съобщение за потребителя
- **Наблюдение за наличност на пари в касата** – при баланс под **5000 лв**. в касата излиза съобщение за потребителя

- **Справки**
    - За доставки и доставчици или изписване и клиенти – тъй като доставчиците и клиентите в нашето приложение са генерализирани като партньори, за съответните справки трябва да бъде подадени само датите, които ни интересуват и типът на фактурата (доставка или изписване).
    - Дейност на складовите оператори – работи по аналогичен начин на гореописаните справки, но тук подаваме името на потребителя и датите, за които ни трябва справка
    - Наличности в склада – списък със всички номенклатури и техните наличности (табличен вид)
    - Разходи, приходи и печалби – за даден период от време да се намират разходите на склада, неговите приходи и печалбата/загубата, която е генерирана между посочените дати
    - Движение на наличността в касата – списък със всички транзакции за зададен период от време

2.2 Структура на проекта

Структурата е разделена по слоеве. Най-ниско седи базата данни. Следващия слой е *ORM* слоят, който чрез *Hibernate*, осъществява връзката с базата данни на *Oracle*. Следващото ниво е *Business* логиката, която отговаря за всички алгоритми, търсения, справки и заявки, от които системата има необходимост. Най-горният слой е графичният интерфейс, чрез който потребителите ще боравят със софтуера. (Фиг. 2.1)



Фиг. 2.1

2.3 Дефиниция на модулите на системата

- **Oracle Database –** Проста БД, в която се съхраняват данните на системата.
- **ORM –** *Data Persistency* слой на приложението, който представя класове като таблиците от базата данни и реализира нужните им функционалности.
- **Business логика –** Основният модул, в който се реализират повечето бизнес изисквания.
- **GUI –** Графичният интерфейс на системата, чрез който потребителя указва желаните от него операции.
- **Logging** – Модул съдържащ класовете отговорни за логването на събитията.

3. Проектиране на системата

3.1 Проектиране на отделните модули

3.1.1 База данни (Oracle)

Базата данни, е разделена на 8 таблици(фиг. 3.3.1.), всяка от които е отражението на определен клас, чиито данни ще се записват във въпросната таблица. Единственото което се изисква от този модул е да предостави структурата, която ще поддържа операциите в системата. Всички останали функционалности, свързани с БД са разпределени в по- горните слоеве (най- вече ORM).

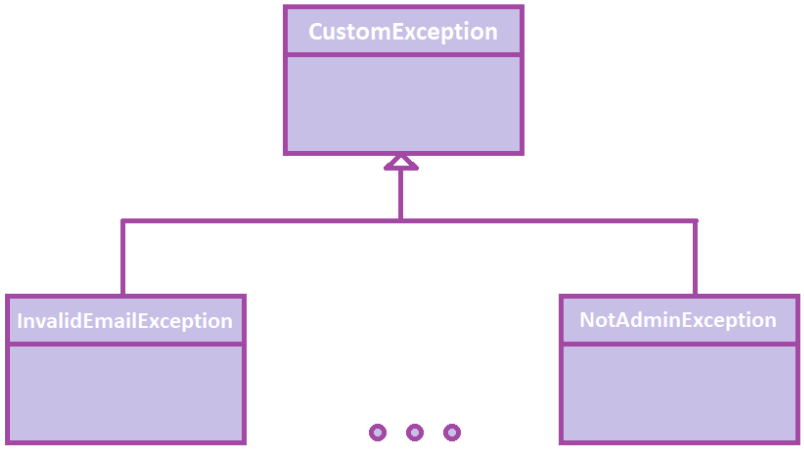
3.1.2 ORM (Hibernate)

Както е споменато в 2.3, ORM модулът играе ролята на посредник между основната система и базата данни. За тази цел, се изпълняват няколко важни процедури върху всеки един клас, който ще бъде записван в БД. Първо и задължително, чрез анотацията в Hibernate ***@Entity***, декларираме избраните от нас класове за такива, които ще превръщаме в таблици. Освен това, във всеки един клас присъства поле с анотация ***@ID***, което ще представлява PK във въпросната таблица. В брой от класовете се използват и ***@SequenceGenerator / @GeneratedValue***, чрез които създаваме Sequence в БД, който използваме за да Auto-Increment-ираме ID-тата на записи в таблиците, където това е нужно (таблиците с неопределен брой записи). При таблици с фиксиран брой записи(Roles, Transactions, Registers). Нужда от такава функционалност няма. Последният важен елемент, реализиран от ORM, са връзките между таблиците. Това се постига, чрез анотация ***@OneToMany*** в клас1 и респективно ***@ManyToOne*** в клас 2, като както подсказват анотациите, се осъществява връзка от тип 1:n от клас1 към клас2. В системата не присъстват връзки m:n.

3.1.3 Business

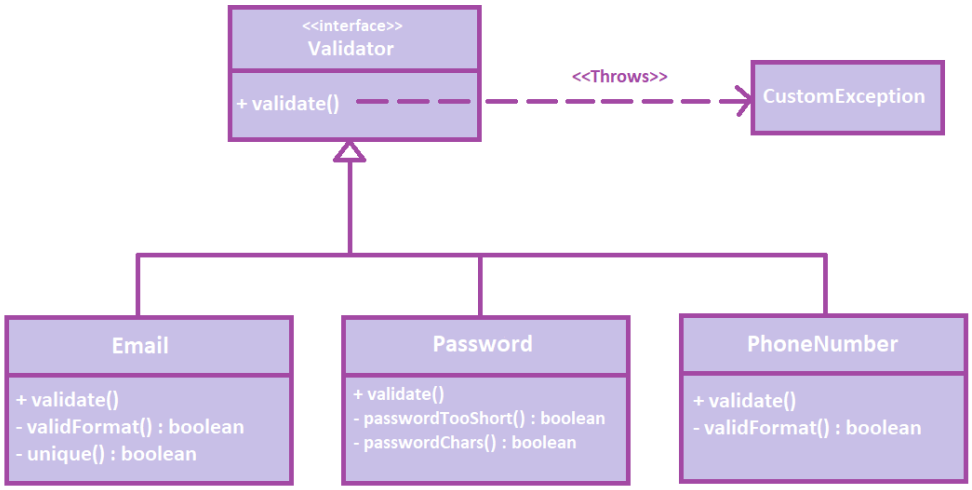
Слоят, в системата, извършващ процедурите, нужни за да се изпълни зададената бизнес логика, за който, по- подробна информация може да бъде намерена в 4.3. В бизнес слоя присъстват следните елементи:

- **Create** – пакет съдържащ класове, които обработват заявките за създаване на нов запис в базата данни. Принципът им е аналогичен: приемат се въведените от потребителя данни, валидират се и само ако са валидни се прави връзка с базата данни и се съхраняват като нови записи
- **Exceptions** – всички предвидени изключения, които програмата може да генерира. Те наследяват класа *CustomException* и всички са свързани с неправилни действия от страна на потребителя



Фиг 2.2. Class Diagram на Exception класовете

- **Repository –** съдържа класове, които отговарят за връзката с *ORM* слоя на приложението. Тези класове съдържат методи, които по зададени критерии създават заявки към базата данни и връщат техния резултат. Заявките са базирани на *Criteria API*, изцяло програмни са, и не включват никакъв *SQL* код. Това е голяма удобство при евентуална смяна на типът база данни, понеже единствената необходима промяна ще бъде в конфигурацията на *Hibernate*.
- **Validators –** много ключов компонент на приложението. Тук се взимат решения дали данните въведени от потребителя са коректни или не. При коректни данни изпълнението на програмата продължава, а в обратен случай се хвърля изключение, което предизвиква даването на подсказка на потребителя кои от въведените данни не са правилни. Валидаторите имплементират интерфейса **Validator**:



Фиг. 2.3. Пример за *Class diagram* на класовете за валидация

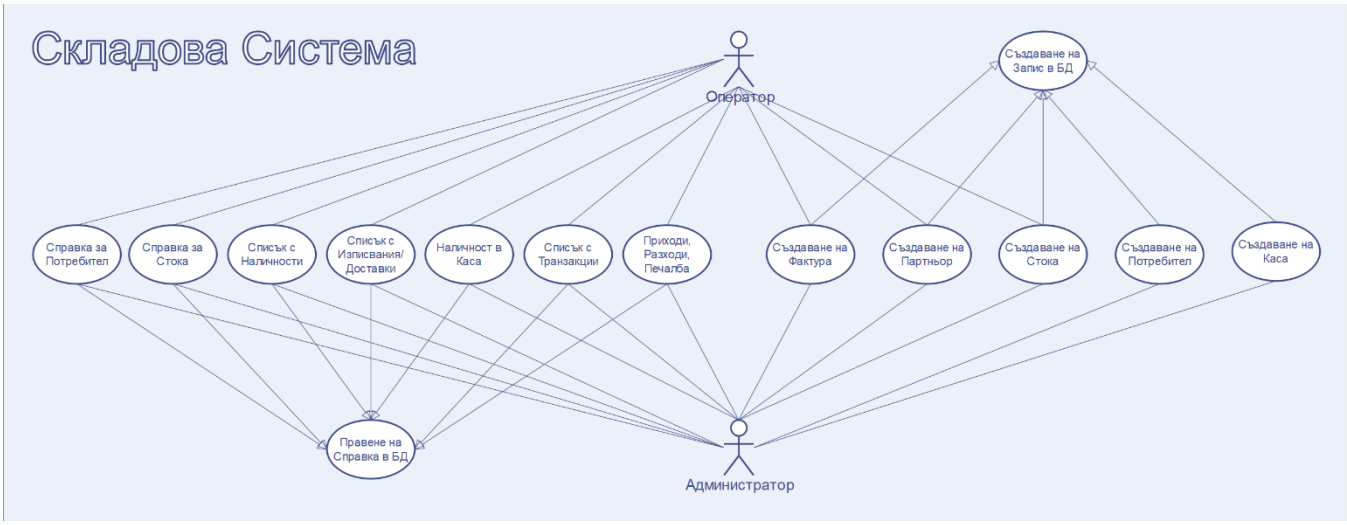
- **CurrentUser.java** – тъй като приложението може да се ползва само от един потребител в даден момент от времето, то този клас е Singleton и съхранява информацията за логнатия потребител
- **GetSession.java –** честото използване на връзка с базата данни наложи създаването на този клас. Единствената му цел е да създаде сесия и да ни я върне, като ни спести досадно писане
- **InitializeData.java –** при пускане на програмата проверява дали е създаден администратор по подразбиране и основните роли и типове транзакции в приложението. Ако не ги намери ги инициализира и по този начин имаме създаден потребител по подразбиране

3.1.4 GUI (JavaFX)

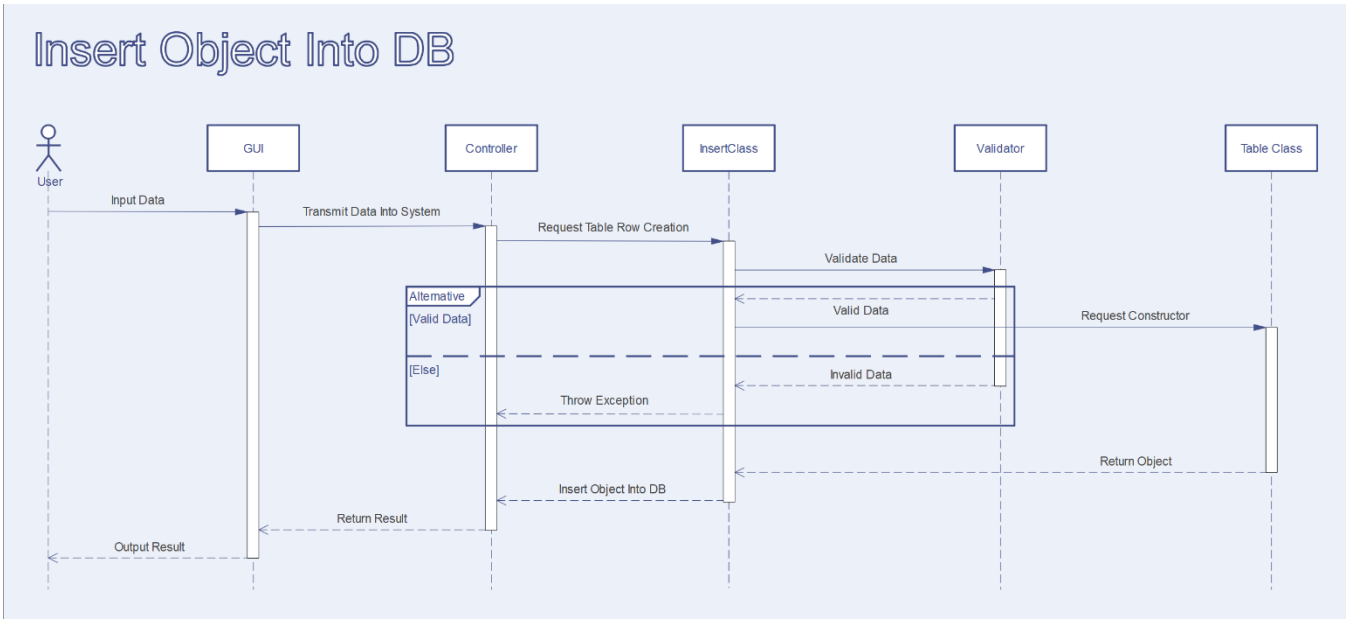
Графичният интерфейс се състои от следните модули:

- **Controllers** – отговарящи за данните, функциите обработващи потребителските събития, както и комуникацията с business логиката на приложението. В тях се обработват и всички възникнали exception-и, като за предвидените от тях се изкарва съобщение какво трябва да направи потребителя, а за останалите exception-и, се извежда съобщение за възникнала грешка и се създава log съобщение
- **Views** – съдържащи визуалните **javafx** елементи и техния дизайн. Файловете са във **fxml** и в приложението се избягва програмното създаване на елементи. Държи се на строгото разграничаване от оформление и функционалност.

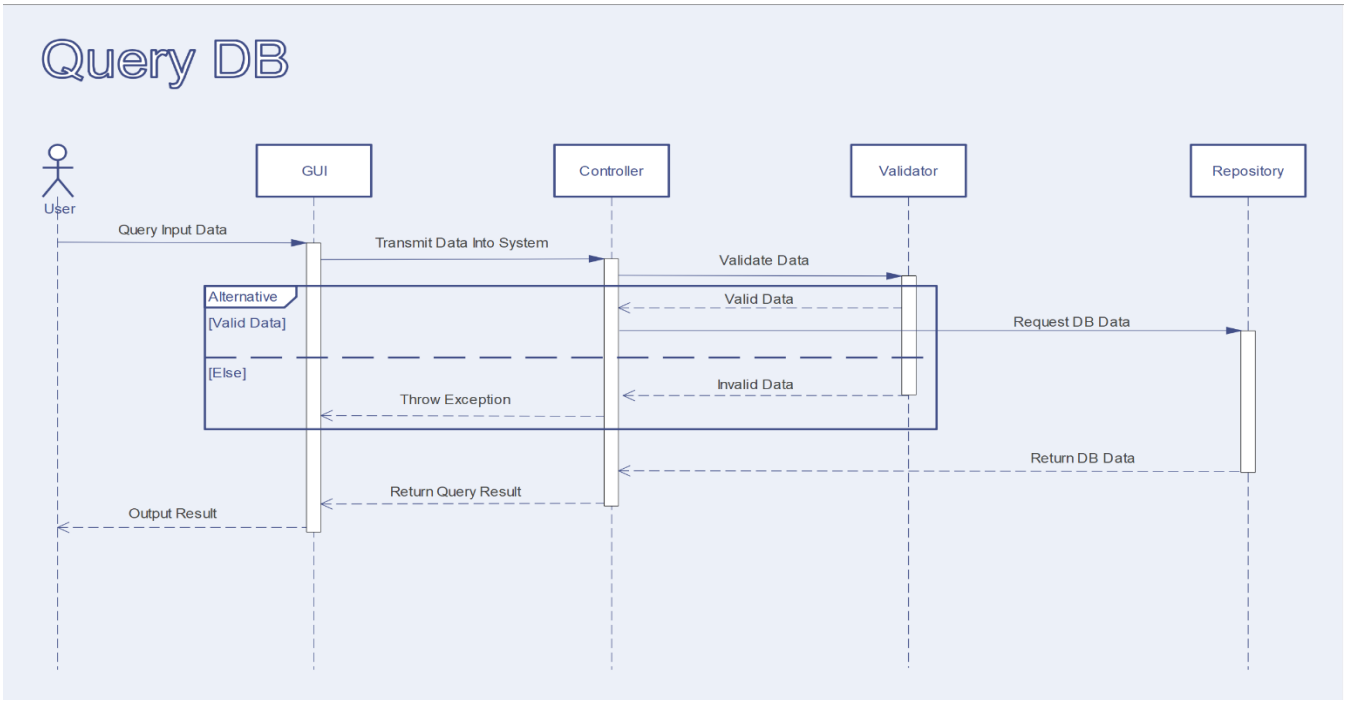
3.2 UML и Relational диаграми.



Фиг.3.2.1 Use Case диаграма на възможностите на потребителите в системата.

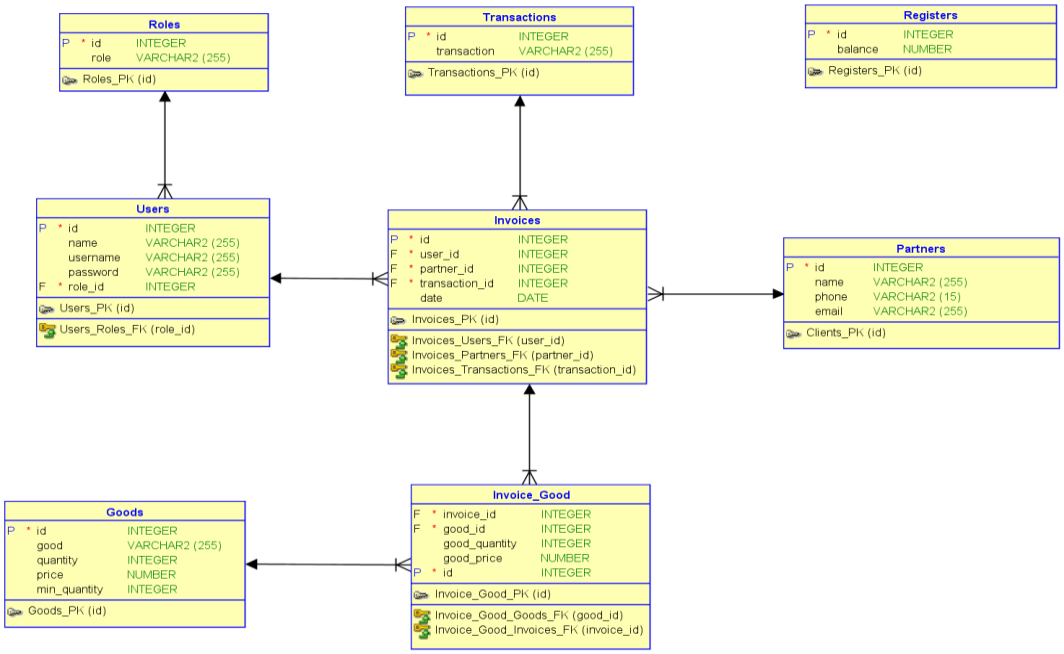


Фиг.3.2.2 Sequence диаграма визуализираща добавянето на запис(обект) в БД.



Фиг.3.2.3 Sequence диаграма визуализираща извличането на запис(обект) от БД.

3.3 Концептуален модел на базата от данни



Фиг.3.3.1 Релационен модел на БД.

4. Реализация на системата

4.1 Реализация на базата от данни (Oracle)

След създаването на концептуалния *Entity Relationship* модел, преминахме към реализацията ѝ на практика. Таблиците и връзките в базата данни се генерират от *Hibernate* и няма никаква ръчна работа за реализацията на базата данни. Повече детайли са описани в следващата точка. Единственото необходимо действие е да се създаде потребител WAREHOUSE в *Oracle SQL Developer* с парола Warehouse123.

Таблици в SQL са както следва:

- **Goods**

	COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
1	ID	NUMBER(10,0)	No	(null)	1	(null)
2	GOOD	VARCHAR2(255 CHAR)	Yes	(null)	2	(null)
3	MINQUANTITY	NUMBER(10,0)	No	(null)	3	(null)
4	PRICE	FLOAT	No	(null)	4	(null)
5	QUANTITY	NUMBER(10,0)	No	(null)	5	(null)

- **Invoice_good**

	COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
1	ID	NUMBER(10,0)	No	(null)	1	(null)
2	GOOD_PRICE	FLOAT	Yes	(null)	2	(null)
3	GOOD_QUANTITY	NUMBER(10,0)	Yes	(null)	3	(null)
4	GOOD_ID	NUMBER(10,0)	Yes	(null)	4	(null)
5	INVOICE_ID	NUMBER(10,0)	Yes	(null)	5	(null)

- **Invoices**

	COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
1	ID	NUMBER(10,0)	No	(null)	1	(null)
2	IN_DATE	DATE	Yes	(null)	2	(null)
3	PARTNER_ID	NUMBER(10,0)	Yes	(null)	3	(null)
4	TRANSACTION_ID	NUMBER(10,0)	Yes	(null)	4	(null)
5	USER_ID	NUMBER(10,0)	Yes	(null)	5	(null)

- **Partners**

	COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
1	ID	NUMBER(10,0)	No	(null)	1	(null)
2	EMAIL	VARCHAR2(255 CHAR)	Yes	(null)	2	(null)
3	NAME	VARCHAR2(255 CHAR)	Yes	(null)	3	(null)
4	PHONE	VARCHAR2(255 CHAR)	Yes	(null)	4	(null)

- **Registers**

	COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
1	ID	NUMBER(10,0)	No	(null)	1	(null)
2	BALANCE	FLOAT	No	(null)	2	(null)

- **Roles**

	COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
1	ID	NUMBER(10,0)	No	(null)	1	(null)
2	ROLE	VARCHAR2(255 CHAR)	Yes	(null)	2	(null)

- **Transactions**

	COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
1	ID	NUMBER(10,0)	No	(null)	1	(null)
2	TRANSACTION	VARCHAR2(255 CHAR)	Yes	(null)	2	(null)

- **Users**

	COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
1	ID	NUMBER(10,0)	No	(null)	1	(null)
2	NAME	VARCHAR2(255 CHAR)	Yes	(null)	2	(null)
3	PASSWORD	VARCHAR2(255 CHAR)	Yes	(null)	3	(null)
4	USERNAME	VARCHAR2(255 CHAR)	Yes	(null)	4	(null)
5	ROLE_ID	NUMBER(10,0)	Yes	(null)	5	(null)

4.2 Реализация на слоя за работа с базата данни (Hibernate)

Цялата реализация се намира в пакета **app.orm**, който съдържа всички *Hibernate Entity*-та, от които се генерират и таблиците в *Oracle* при стартиране на приложението ако е зададена опцията **hbm2ddl.auto** да бъде **create** в **hibernate.cfg.xml**. Съответните класов представляват таблиците и тяхната структура както следва:

- **Good** – представлява таблица Goods, която съдържа в себе си номенклатурите на стоки, с които складът борави.
    - **Полета на класа**
        - Int **id** – уникален идентификатор, увеличава се с 1 за всеки нов запис чрез генерирани стойности от поредицата *good_seq*
        - String **good** – име на номенклатурата
        - Int **quantity** – налично количество от номенклатурата
        - Double **price** – текуща цена на стоката
        - Int **minQuantity** – критично минимално количество, което складът трябва да поддържа в наличност

    - **Релации**
        - **Invoice_goods** – връзка едно към много, чрез уникалния идентификатор на стоката, който се явява foreign key в таблицата Invoice_Good
- **Invoice** – представлява таблицата Invoices, която съдържа в себе си доставките и изписванията на склада
    - **Полета и релации на класа**
        - Int **id** – уникален идентификатор, увеличава се с 1 за всеки нов запис чрез генерирани стойности от поредицата *invoice_seq*
        - Calendar **calendar**- датата, на която е направена фактурата, представя колоната *in_date*
        - User **user** – потребителят, който е създал фактурата (оператор или администратор), представлява колоната *user_id*, която осъществява релация много към едно с таблицата *Users*
        - Partner **partner** – партньорът, за който се отнася сделката, представлява колоната *partner_id*, която осъществява
        - Transaction **transaction** – типът на транзакцията (покупка или продажба), представя колоната *transaction_id*, която осъществява връзка много към едно с таблицата *Transactions*
- **Invoice_Good** – представлява таблицата *Invoice_good*, която съдържа в себе си стоките, за които се отнасят транзакциите
    - **Полета и релации на класа**
        - Int **id** – уникален идентификатор, увеличава се с 1 за всеки нов запис чрез генерирани стойности от поредицата *invoice_good_seq*
        - Int **quantity** – количеството на стоката, представлява колоната good_quantity
        - Int **price** – цена на стоката към момента на продажба, представлява колоната good_price. Умишлено отделена от цената на стоката в таблица Goods, заради нейната възможна промяна в бъдеще време. Така е ясно на каква цена е изписана/доставена стоката в момента на събитието
        - Invoice **invoice** – фактурата, към която се отнася съответната стока, представлява колоната *invoice_id*, осъществява релацията много към едно с таблицата *Invoices*
        - Good **good** – стоката, за която се отнася записа в таблицата, представлява колоната good_id, осъществява релацията много към едно с таблицата *Goods*
- **Partner** – представлява таблицата Partners, която съдържа в себе си партньорите, с които складът взаимодейства (клиенти или доставчици)

    - **Полета**
        - int **id** – уникален идентификатор, увеличава се с 1 за всеки нов запис чрез генерирани стойности от поредицата *partner_seq*
        - String **name** – името на фирмата
        - String **phone** – телефонът на фирмата
        - String **email** – имейл адресът на фирмата
    - **Релации**
        - List<Invoice> **invoices** – връзка между таблица Partners и таблица Invoices. Уникалният идентификатор на партньора, се явява *foreign key* в таблицата с фактурите
- **Register** - представлява таблицата Registers, която съдържа в себе си касата на склада
    - **Полета**
        - Int **id** – уникален идентификатор
        - Double **balance** – наличните финансови ресурси в съответната каса
- **Role** – представлява таблицата *Roles*, която съдържа в себе си възможните роли за потребителите на приложението
    - **Полета**
        - Int **id** – уникален идентификатор
        - Roles **role** – ограничена до създадените в енумерацията *Roles*
    - **Релации**
        - List<User> **users** – връзка едно към много между таблиците *Role* и *Users*. Уникалният идентификатор на ролята се явява *foreign key* в таблицата с потребители
- **Transaction** – представлява таблицата *Transactions*, която съдържа в себе си различните типове транзакции
    - **Полета**
        - Int **id** – уникален идентификатор
        - Transactions **transaction** – ограничени до създадените в енумерацията *Transactions*
    - **Релации**
        - List<Invoice> **invoices** – връзка едно към много между таблиците *Transactions* и *Invoices*. Уникалният идентификатор на типът транзакция се явява *foreign key* в таблицата с фактури
- *User* – представлява таблицата *Users*, която съдържа в себе си всички потребителски акаунти за склада
    - **Полета**
        - Int **id** - уникален идентификатор, увеличава се с 1 за всеки нов запис чрез генерирани стойности от поредицата *user_seq*
        - String **name** – личното име на потребителя
        - String **username** – потребителското име за системата

- String **password** – парола на потребителя
- Role **role** – ролята на потребителя (оператор или администратор), представлява колоната role_id и осъществява релация много към едно

- **Релации**
  - List<Invoice> **invoices** – фактурите, които са свързани със съответния потребител. Осъществява релация едно към много между таблиците *User* и *Invoices*, като уникалният идентификатор на потребителя се явява *foreign key* в таблицата с фактури

4.3 Реализация на бизнес логика (Java)

Както е описано в 3.1.3, Business модула е изграден от пакети: ***create***, ***repository***, ***exceptions*** и ***validators*** (присъства и пакет ***tools***, в който има няколко класа с изнесена логика, използвана в create/repository). Exception и Validator класовете са обяснени в 3.1.3. Двата основни блока които ще разгледаме тук са Create и Repository, които общо казано реализират въвеждането и респективно извеждането на данни в и от БД.

4.3.1 Create

В този пакет реализираме въвеждането да данни в БД. Това става чрез ***Insert*** класовете. Всеки един Insert клас, отговаря за таблица от БД, като не притежава никакви полета. Единствено присъства статичен метод ***create***. Въпросният метод взима като параметри данните, нужни за създаването на новия обект, който искаме да запишем. За да се свържем с БД, първо създаваме сесия чрез ***GetSession*** класа(3.1.3). След това, валидираме всяко едно поле на новия клас, като ако някъде намерим грешни данни, въпросният валидатор хвърля exception, отговарящ на конкретната грешка. Ако данните ни са правилни, извикваме конструктура на въпросния клас от ORM слоя и му ги подаваме. След като създадем новия клас, започваме транзакция (**session.beginTransaction()**), записваме въпросния ред в БД (**session.save(нов обект)**), потвърждаваме транзакцията (**session.getTransaction().commit()**) и затваряме сесията (**session.close()**). Тази политика на „1 транзакция : 1 операция с БД“, се следва в цялата система. Ако класа който създаваме има връзки с други класове(таблици), тоест има полета чрез които са свързани (FK, ID на обекта, записан в таблицата с който трябва свързан новия), тези полета също се подават на метода като параметри. За всяка връзка с новия клас, се създава временен обект, чиито данни взимаме по неговото ID (***Временен_обект обект = session.get(Временен_обект.class, ID)***). След неговото създаване, всички полета за връзки в новия обект, както и във временния обект се попълват с респективните ID-та на обекта с който трябва да се свържат. След добавянето на въпросните връзки, трябва да обновим всички записи в БД, които вече сме свързали с нашия нов обект

(***session.update(обект)***). Нека за пример да вземем добавянето на нов потребител в БД чрез класа ***InsertUser***:

```
public class InsertUser {
    public static void create(String name, String username, String password, String passwordConfirm, int roleID) throws Exception {
        Session session = GetSession.getSession();

        //Data validation
        new Password(password, passwordConfirm).validate();
        new Username(username).validate();

        //Creating the user object with the data
        //Some sort of data validation before this step...
        User newUser = new User(name, username, password);

        //Transaction begins now, because we need to update "ROLES" object with PK "roleID" in order
        //to create the FK column in the new row we are currently creating with said "roleID" PK
        session.beginTransaction();

        Role role = session.get(Role.class, roleID);
        newUser.setRole(role);
        role.getUsers().add(newUser);

        //Updating row "role" and saving the new row "user"
        session.save(newUser);
        session.update(role);

        session.getTransaction().commit();
        session.close();
    }
}
```

4.3.2 Repository

В Repository пакета, обратно на Create, извличаме данните от БД. Това става чрез ***Repository*** класовете. Всеки клас, за който ще искаме да правим справки в БД, си има собствен Repository клас. Repository класовете, подобно на Insert класовете нямат полета, а само статични ***findBy*** методи, чрез които се правят справки по различни критерии, изпрани от потребителя. На findBy метода, като параметър, се подава стойността, по която ще се извършва справката. Аналогично на предишната точка, първо правим връзка с базата данни чрез ***GetSession*** класа(3.1.3). След това е време да създадем нашата справка. Това се случва чрез използване на ***javax.persistence.criteria*** библиотеката. Първо създаваме ***CriteriaBuilder***, с който ще създаваме критериите за правенето на справката ни (***session.getCriteriaBuilder()***). Използваме CriteriaBuilder-а ни за да създадем нова справка, така наречената ***CriteriaQuery***, като задаваме класа от ORM, който отговаря на таблицата в БД, в която ще правим нашата справка (***CriteriaQuery<Клас> criteriaQuery = criteriaBuilder.createQuery(Клас.class)***). Следва да създадем инстанция на класа ***Root***, като отново задаваме класа, за който правим справката (***Root<Клас> root = criteriaQuery.from(Клас.class)***). Въпросният клас се използва, за да се създаде временно копие на текущия ред на таблицата на който се намираме по време на търсене под формата на обект, като ако е изпълнен критерият по който правим

справката, въпросния ред се записва в списък с обекти, които отговарят на изискванията. Ако текущия ред не отговаря на критерия, той се пропуска. Следва да се добави и самия критерий по който ще правим справката (***criteriaQuery.where(criteriaBuilder.like(root.get("колонка в таблицата за сравняване"), подадена от потребителя стойност))***). Тази стъпка може да бъде пропусната, тоест да не зададем определен критерий по който да направим справката. В такъв случай, всеки ред от таблицата ще бъде записан в резултатния списък.След извършване на справката, резултатите се подават на нов списък, като се взимат от по- рано споменатия списък с редове, отговарящи на условията. Остава само да се затвори сесията и да се върне списъка с резултати за по-нататъчна обработка. Нека за пример да вземем класа ***GoodRepository***, в който имаме два метода за правене на справки: за търсене на всички стоки (***findAll()***) и за извеждане само на стоката, чието име отговаря на това, подадено от потребителя (***findByGood(String good)***) (за тази справка, можем да имаме единствено 0 или 1 обект в резултатния списък, понеже дублирането на имената на стоките в БД не е позволено):

```java
public class GoodRepository {
    public static List<Good> findAll(){
        Session session = GetSession.getSession();

        CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();
        CriteriaQuery<Good> criteriaQuery = criteriaBuilder.createQuery(Good.class);
        criteriaQuery.from(Good.class);

        List<Good> result = session.createQuery(criteriaQuery).getResultList();

        session.close();
        return result;
    }

    public static List<Good> findByGood(String good) {
        Session session = GetSession.getSession();

        CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();
        CriteriaQuery<Good> criteriaQuery = criteriaBuilder.createQuery(Good.class);
        Root<Good> root = criteriaQuery.from(Good.class);
        criteriaQuery.where(criteriaBuilder.like(root.get("good"), good));

        List<Good> result = session.createQuery(criteriaQuery).getResultList();

        session.close();
        return result;
    }
}
```

4.4 Реализация графичен интерфейс (JavaFX)

Графичният интерфейс е реализиран чрез JavaFX, използван в комбинация с визуалния инструмент SceneBuilder, който значително улеснява процеса на изграждане на графична среда и предоставя по- голям брой естетически опции. В JavaFX, прозорецът в който работим, наричаме ***Stage***, а съдържанието – ***Scene***. Класът ***SceneManager*** отговаря за изграждането на Stage и зареждането на Scene в него:

```
public class SceneManager {
    public void load(Stage stage, String title, String scenePath) {
        try {
            Parent mainLayout = FXMLLoader.load(getClass().getResource(scenePath));
            stage.setTitle(title);
            stage.setScene(new Scene(mainLayout,  width: 1330,  height: 800));
            stage.show();
        } catch (Exception e) {
            loadFailed(e);
        }
    }
```

Метод ***load*** извикваме в ***Main*** метода ни, за да заредим първоначалния прозорец. За да можем да настройваме какво ще бъде изобразено във нашия текущ scene, както и как различните му елементи ще взаимодействат с действията на потребителя, използваме пакетите ***Views*** и ***Controllers***.

4.4.1 View

Инструментът SceneBuilder позволява графичния дизайн на интерфейса (бутони, полета, кутии и тн.) да се изгражда на ръка чрез интуитивни инструменти и менюта. След изграждане на прозорец, ние можем да преработим графичната информация за него в нов файл с разширение ***fxml***. Файлове от този тип наричаме ***View***. След задаване на Stage, ако искаме да променяме Scene, който се намира във зададения Stage, тоест съдържанието му, използваме клас ***ViewManager***, на който подаваме новия Scene, под формата на View.

4.4.2 Controller

Във всяко едно View, присъстват интерактивни елементи като бутони, текстови полета, кутии за избор и др. В зависимост от това какво иска да направи потребителя, видовете операции се свеждат до два типа: извличане на данни от БД и изобразяването им на екрана / записване на данни, зададени от потребителя в БД. За да се постигне тази интерактивност използваме класовете от тип ***Controller***. За всяко едно View имаме един Controller, управляващ неговите елементи. На всеки един интерактивен елемент се задава ID, още по време на изработката на View-то в който той се намира. В Controller класа, тези ID-та на елементи се използват като имена на променливи(обекти), отговарящи на въпросния елемент,

които предоставя JavaFX. Някои от тях са: Button, TextField, Label, ComboBox и всички останали стандартни елементи на форми. Вече имайки обектни имплементации на визуално-изобразените елементи на прозореца ни, можем лесно да достъпваме, променяме и запазваме данните въведени от потребителя, като полета на въпросните обекти. Лесен пример за това е извличането на текст, зададен от потребителя, от елемент от тип TextField: ***String съдържание = TextField.getText().*** По този начин, при правене на записи в БД, взимаме въведените от потребителя данни, и ги подаваме на подходящия ***Insert(4.3.1)*** клас, а респективно при правене на справки, изпращаме изискванията въведени от потребителя на подходящия ***Repository(4.3.2)*** клас, след което извеждаме резултатите върнати от споменатия клас на екрана. Желаната от нас логика се задейства се изпълнява на принципа на събитията. Събитията могат да бъдат най-различни: първоначално зареждане на прозореца, натискане на бутон, въвеждане на текст, дори натискане на статичен текст по екрана. Всяко едно такова събитие също получава ID по време на проектирането в SceneBuilder, като за разлика от самите елементи, които имплементираме чрез обекти, събитията имплементираме чрез методи. При активация на определено събитие, се задейства методът, свързан с него чрез неговото ID и съответно се изпълнява логиката вътре в него.

При пускане на програмата, в Main класа ни, задействаме метод ***start*** (метод от JavaFX, който се използва за инициализиране на началния Stage), в който за Scene задавме ***login.fxml*** View, като по този начин зареждаме екрана за влизане в системата.

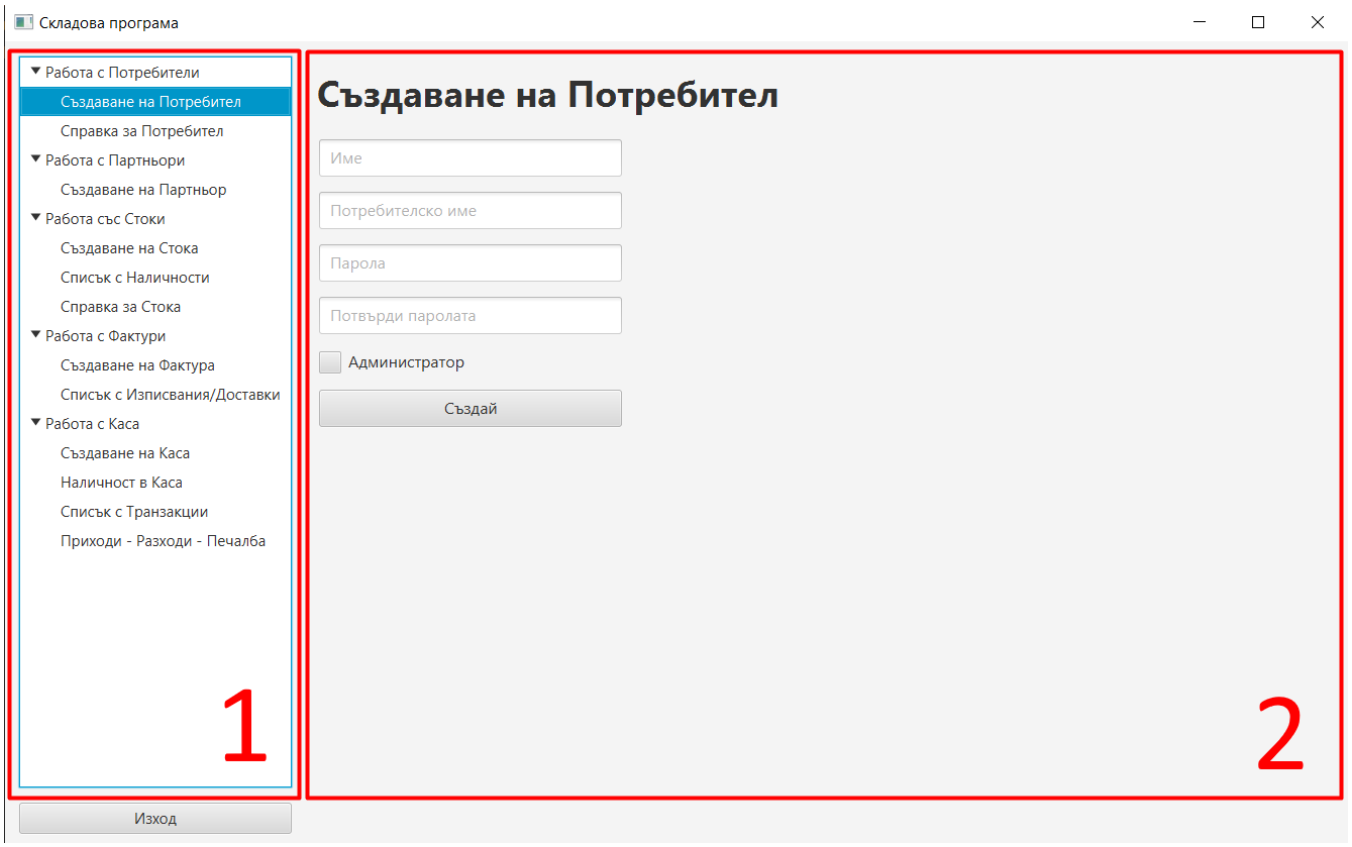
```
public class Main extends Application{

    @Override
    public void start(Stage primaryStage) throws Exception {
        //Initialize DB
        new InitializeData();

        //Log4j
        new InfoLogging().log( msg: "Starting the application...");

        //JavaFX Setup
        SceneManager loadScene = new SceneManager();
        loadScene.load(primaryStage,  title: "Вход в системата",  scenePath: "views/login.fxml");
```

При успешно влизане в системата (въведени правилни име и парола на съществуващ потребител) за текущ потребител се задава този с когото сме влезли и сме пренасочени към главното меню (***layout.fxml***), чрез смяна на сцената извършена в Controller - ***Login***. Главното меню е изградено от 2 елемента: Вертикална кутия, съдържаща различните опции в програмата(1), както и контейнер, съдържащ елементите на избраната от нас текуща опция(2).



Елемент 1, е постоянен за прозореца, докато елемент 2 се третира като вътрешен прозорец, който може да зарежда и променя свой собствен Scene, индивидуално от общия прозорец в който се намира. Тази възможност се дължи на факта, че контейнерът (2) е от тип ***AnchorPane(JavaFX)***, който позволява индивидуална манипулация на вграден прозорец(панел). Имайки предвид тази структура, чрез натискане на опция от главното меню, страничният панел променя своето View в зависимост от това какво сме натиснали. Натискането на опция се регистрира като събитие в Controller ***Layout,*** който от своя страна задейства клас ***ViewManager***, на който се подава адреса на View-то което отговаря на опцията натисната от потребителя. Използвайки този адрес, във ViewManager се задейства метод ***load,*** който заменя текущия Scene в новия избран.

```
public void load(AnchorPane pane, String path) {
    try {
        Parent newFile = FXMLLoader.load(getClass().getResource(path));
        pane.getChildren().setAll(newFile);
    } catch(Exception e) {
        new ErrorLogging().log(ExceptionToString.convert(e));
    }
}
```

AnchorPane pane е страничния панел, който подаваме на функцията, а ***String path*** е пътят към View-то, което сме избрали. Този път получаваме, от switch, който определя какъв string да подаде, в зависимост от това коя опция сме натиснали във метод ***chooseView***.

```
public void chooseView(AnchorPane pane, String option) throws Exception {
    switch(option) {
        case "Създаване на Потребител":
            new Admin().validate();
            this.load(pane, path: "views/createUser.fxml");
            break;
        case "Справка за Потребител":
            this.load(pane, path: "views/userQuery.fxml");
            break;
        case "Създаване на Партньор":
            this.load(pane, path: "views/createPartner.fxml");
            break;
        case "Създаване на Стока":
            this.load(pane, path: "views/createGood.fxml");
            break;
```

Показана е само примерна част от опциите за нагледност.

При извеждане на съобщения от системата, при извършването на различните операции, било то поради възникнали грешки, или успешна операция, се извиква метода ***display*** на клас ***AlertBox***. Въпросният метод създава нов временен прозорец (с нов Stage), като докато не бъде затворен, достъпът до главния прозорец е блокиран.

```
public class AlertBox {
    public static void display(String title, String message) {
        Stage window = new Stage();

        window.initModality(Modality.APPLICATION_MODAL);
        window.setTitle(title);
        window.setMinWidth(500);
        window.setMinHeight(250);

        Label label = new Label();
        label.setText(message);
        label.setFont(new Font( size: 20));

        Button closeButton = new Button( text: "Затвори");
        closeButton.setFont(new Font( size: 16));
        closeButton.setOnAction(e -> window.close());

        VBox layout = new VBox();
        layout.getChildren().addAll(label, closeButton);
        layout.setAlignment(Pos.CENTER);
        layout.setSpacing(10);
        layout.setPadding(new Insets( topRightBottomLeft: 10));

        Scene scene = new Scene(layout);
        window.setScene(scene);
        window.showAndWait();
    }
}
```

Под главното меню, се намира бутон „изход“, при натискане на който, текущият потребител се нулира и сме върнати в прозореца за влизане в системата.

4.5 Реализация на модул за регистриране на събития в системата (Log4J)

Този модул не е особено обемен. Състои се от 4 класа:

- **Logging** – абстрактен клас, който създава логър и създава необходимата структура за класовете, които ще го наследяват
- **InfoLogging** – клас разширяващ класа Logging и реализиращ метода log(String msg), който създава инфо лог
- **ErrorLogging** – клас разширяващ класа Logging и реализиращ метода log(String msg), който създава error лог и изкарва съответното съобщение на потребителя
- **ExceptionToString** – чрез метода convert(Exception e) обръща изключението в стринг. Използва се в класа ErrorLogging

5. Тестови резултати

5.1 JUnit tests

Използват се следните тестове:

- Create – Тества методите, които създават записи в базата данни
  - InsertGoodTest – създаване на стока
    - successfullyCreatedGood
  - InsertPartnerTest – създаване на партньор
    - successfullyAddedPartner
  - InsertUserTest – създаване на потребител
    - successfullyCreatedUser
- Validators
  - AdminTest – тества коректно ли работи валидаторът за администраторски права
    - userIsAdmin – проверява поведението, когато подаденият потребител е админ
    - userIsNotAdmin – проверява поведението когато подаденият потребител не е админ,
  - DatesConsecutiveTest – тества коректно ли работи валидаторът за последователни дати
    - consecutiveDates – проверка при подадени последователни дати
    - nonConsecutiveDates – проверка при подадени непоследователни дати
  - DateValidatorTest – тества коректно ли работи валидаторът за формат на дати

        - Validate – проверка при подадени валидни данни
        - invalidate - проверка при подадени невалидни данни
    - EmailTest - тества коректно ли работи валидаторът за формат на електронни пощи
        - basicPatern – проверка при липса на @
        - strictPattern – проверка при лиспа на валиден домейн
        - RFC5322Patter – проверява за ‘, “, |
        - topLevelDomainPattern – проверява за наличие на валиден топ левъл домейн
        - dotsPattern – проверява за наличие на последователни точки
    - GoodQuantityTest - тества коректно ли работи валидаторът за количество на стоката
        - validQuantity – подава валидно количество
        - negativeQuantity – подава отрицателно количество
        - zeroQuantity – подава 0 за количество
        - notEnoughQuantity – подава количество, по-голямо от необходимото
    - GoodValidatorTest - тества коректно ли работи валидаторът за име на стоката
        - goodNotExists – проверка при подадено име, което не е вече зает
        - goodExists – проверка при подадено име, което вече е зает
    - PartnerNameTest - тества коректно ли работи валидаторът за име на партньор
        - partnerNotExists - проверка при подадено име, което не е вече зает
        - partnerExists - проверка при подадено име, което вече е зает
    - PasswordTest - тества коректно ли работи валидаторът за парола
        - correctPassword – проверка при правилно подадена парола
        - noUppercasePassword – проверка при парола без главни букви
        - noLowercasePassword – проверка при парола без малки букви
        - noNumbersPassword – проверка при парола без числа
        - tooShortPassword – проверка при твърде къса парола
        - notMatchingPassword – проверка при несъвпадаща парола за потвърждение
    - PhoneNumberTest – тества коректно ли работи валидаторът за телефонни номера
        - correctPhone – при правилно подаден телефон
        - tooShortPhone – при твърде къс телефонен номер
        - tooLongPhone – при твърде дълъг телефонен номер
        - invalidFormatPhone – при номер с невалиден формат

- PriceTest – тества коректно ли работи валидаторът за цена
  - validPrice – при подадена валидна цена
  - invalidPrice – при подадена невалидна цена (твърде голяма)
  - negativePrice – при подадена отрицателна цена
- QuantityTest – тества коректно ли работи валидаторът за количество
  - validQuantity – при подадено валидно количество
  - invalidQuantity – при подадено невалидно количество
- UsernameTest – тества коректно ли работи валидаторът за потребителско име
  - usernameDoesNotExist - проверка при подадено потребителско име, което не е вече заето
  - usernameExists - проверка при подадено потребителско име, което е вече заето