**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 ??? Милков - ???

**Съдържание**

1. Задание на проекта
2. Анализ на проблема

2.1 Функционални изисквания

2.2 Структура на проекта

2.3 Дефиниция на модулите на системата

1. Проектиране на системата

3.1 Проектиране на отделните модули

3.2 UML – Use Case, Class Diagram, Sequence diagram и други;

3.3 Концептуален модел на базата от данни

1. Реализация на системата

4.1 Реализация на базата от данни (Oracle)

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

4.3 Реализация на бизнес логика и графичен интерфейс (Java, JavaFX)

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

1. Тестови резултати
   1. JUnit tests
2. **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;

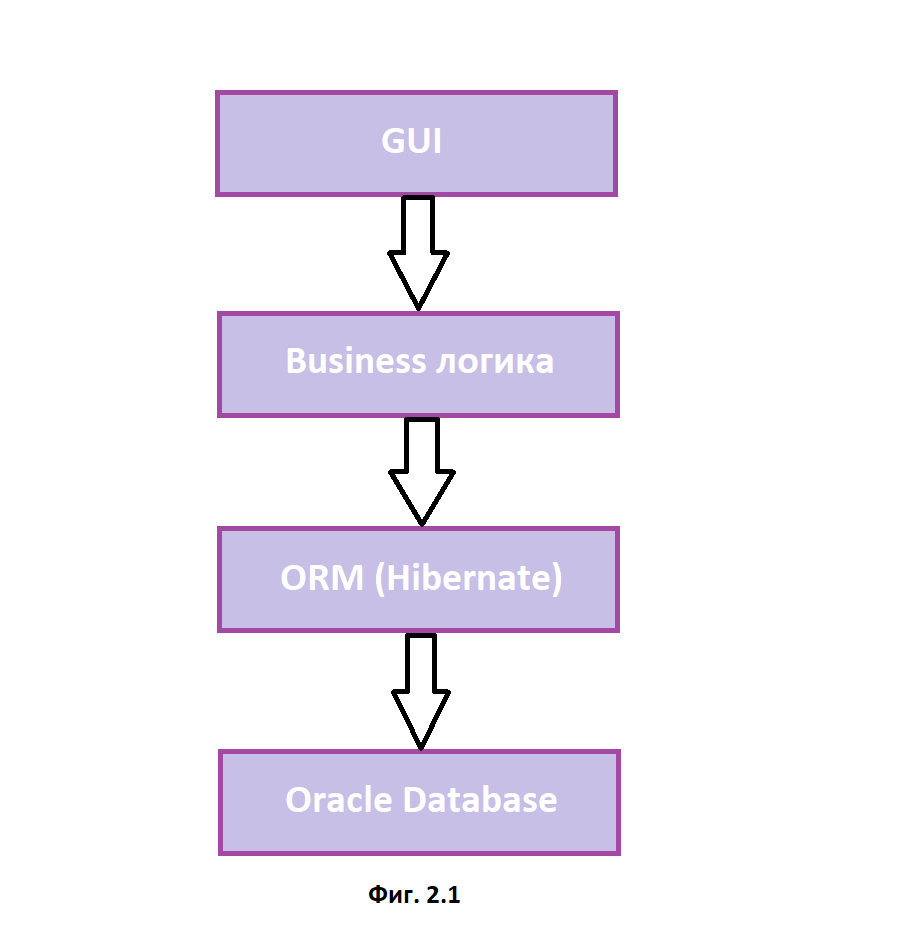
1. **Анализ на проблема**
   1. **Функционални изисквания**

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.

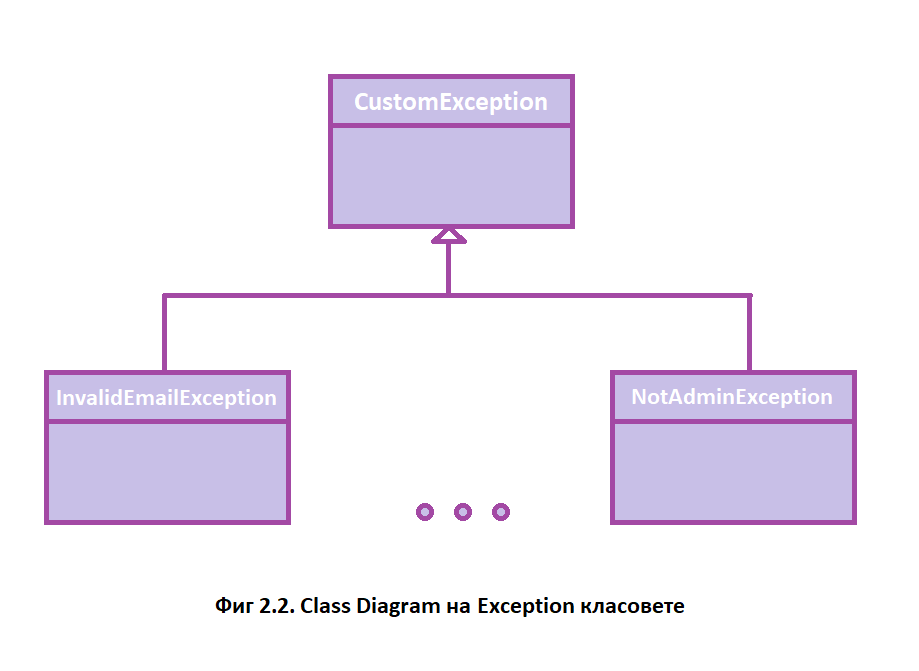
След успешна автентикация потребителя има достъп до фукционалността предоставена от приложението. Опциите за работа са следните:

* **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 съответние 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
  1. **Структура на проекта**

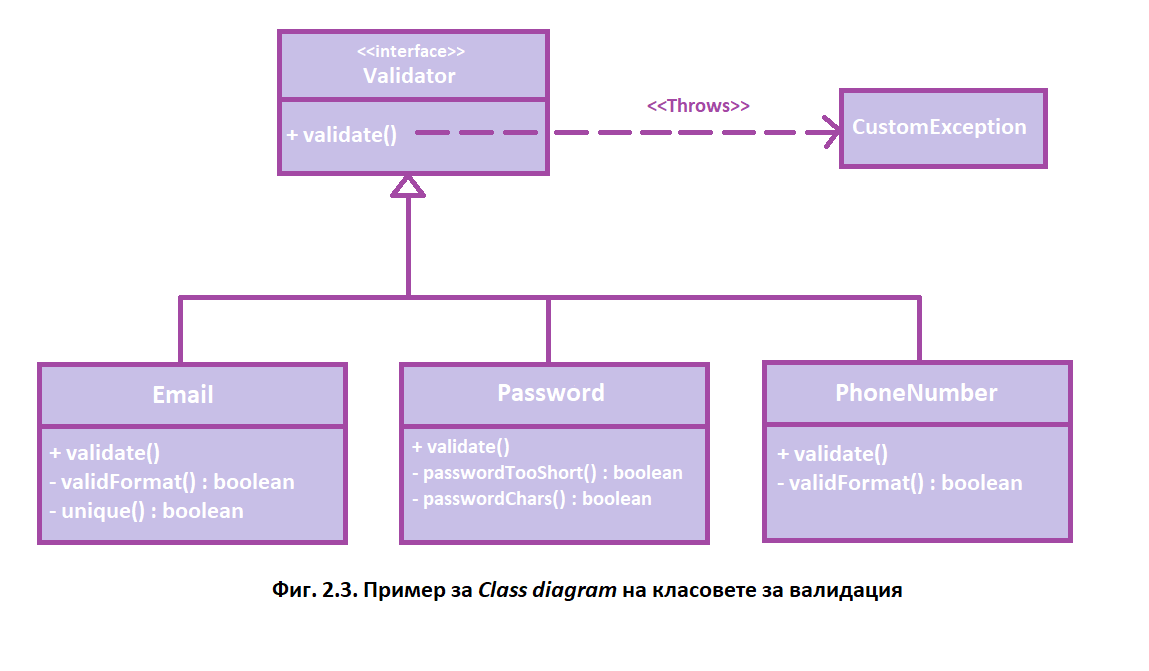
Структурата е разделена по слоеве. Най-ниско седи базата данни. Следващия слой е *ORM* слоят, който чрез *Hibernate*, осъществява връзката с базата данни на *Oracle*. Следващото ниво е *Business* логиката, която отговаря за всички алгоритми, търсения, справки и заявки, от които системата има необходимост.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



* 1. **Дефиниция на модулите на системата**
* **ORM –** *Data Persistency* слой на приложението, който представя като класове таблиците от базата данни.
* **Business** 
  + **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



* + **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**:

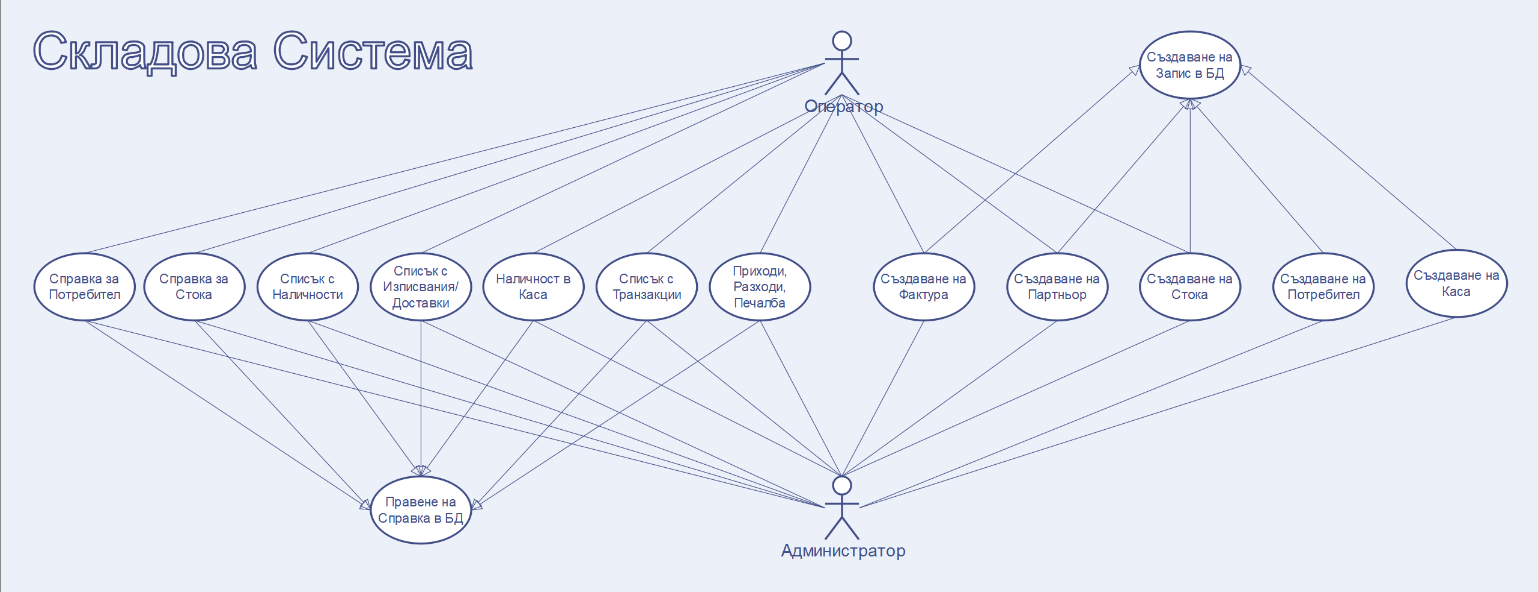
****

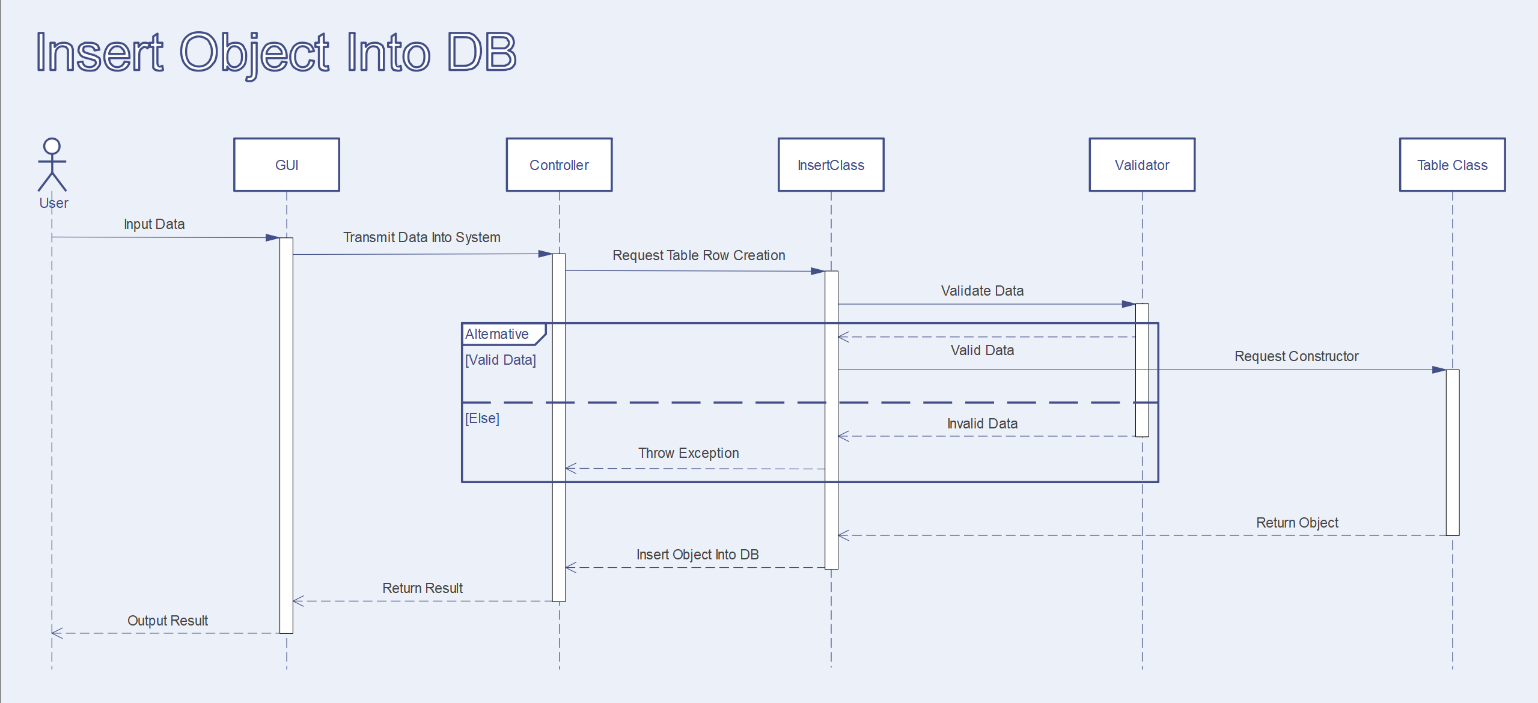
* + **CurrentUser.java** – тъй като приложението може да се ползва само от един потребител в даден момент от времето, то този клас е Singleton и съхранява информацията за логнатия потребител
  + **GetSession.java –** честото използване на връзка с базата данни наложи създаването на този клас. Единствената 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
* **GUI –** Състои се от два модула
  + **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.
* **Logging** – Модул съдържащ класовете отговорни за логването на събитията

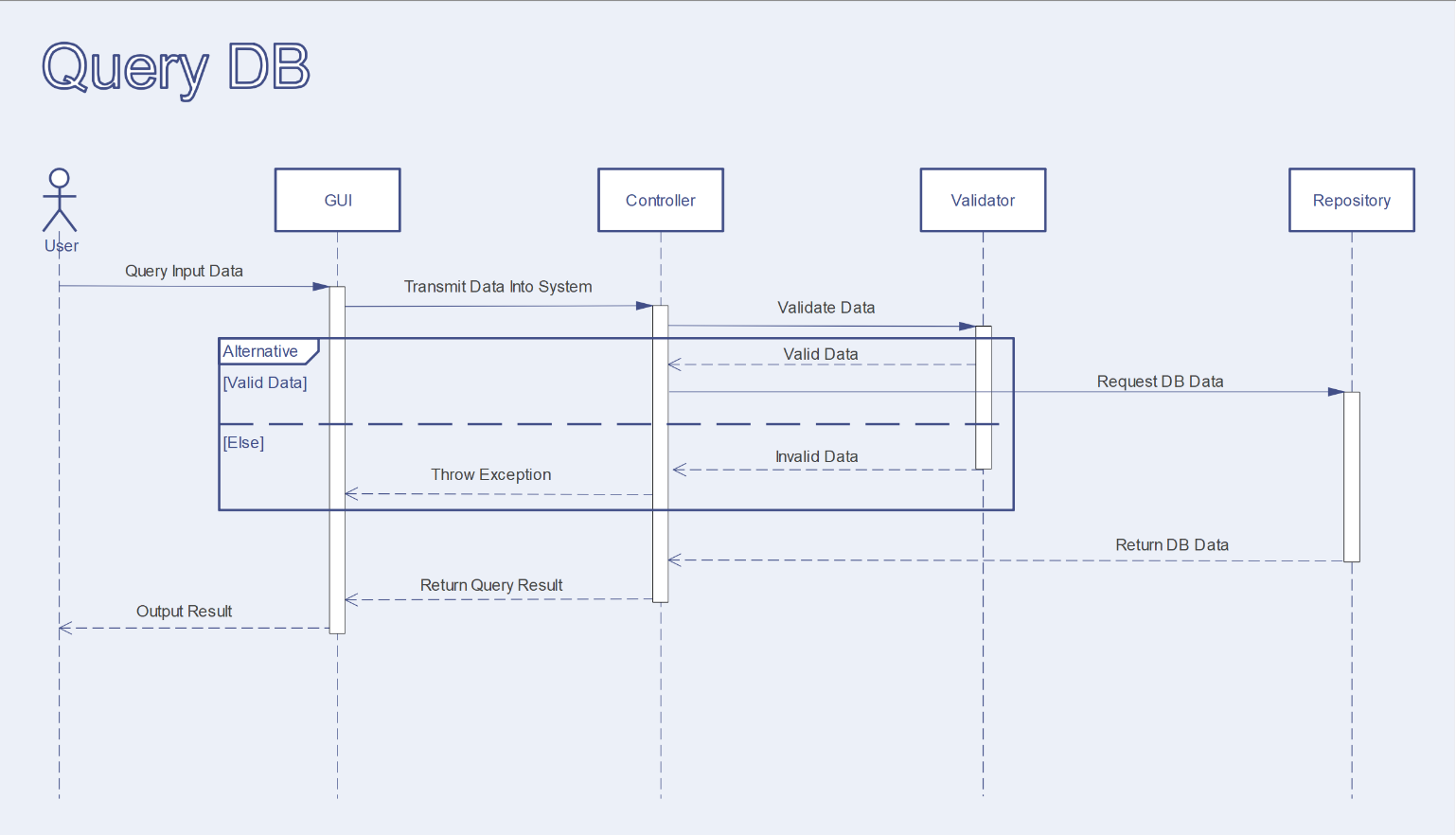
**3. Проектиране на системата**

* 1. **Проектиране на отделните модули**

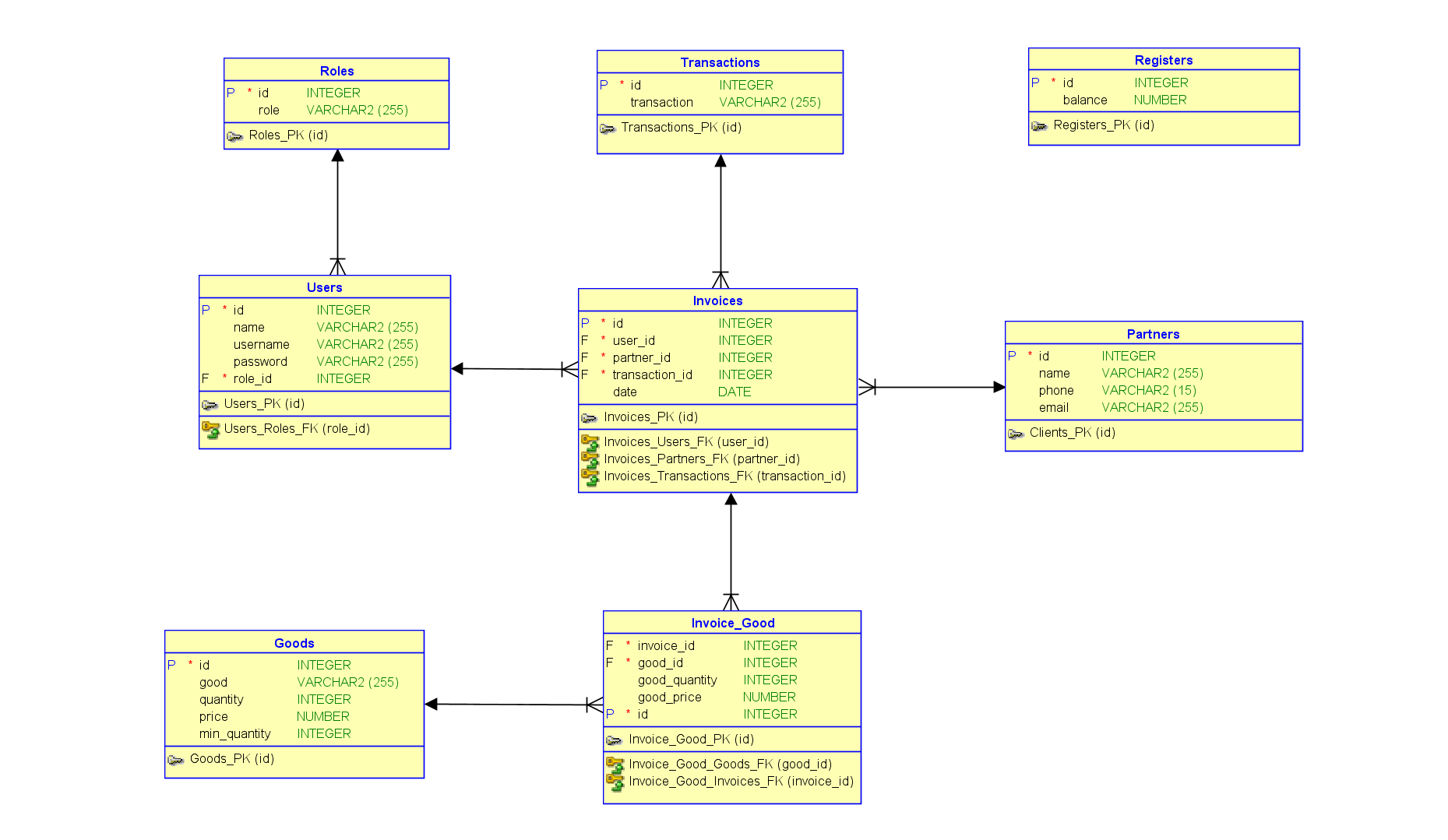
**3.2 UML – Use Case, Class Diagram, Sequence diagram и други**

****

****

****

* 1. **Концептуален модел на базата от данни**

****

**4. Реализация на системата**

* 1. **Реализация на базата от данни (Oracle)**

След създаването на концептуалния *Entity Relationship* модел, преминахме към реализацията й 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**

Table

Description automatically generated

* **Invoice\_good**

Table

Description automatically generated

* **Invoices**

Table

Description automatically generated

* **Partners**

Table

Description automatically generated

* **Registers**

Table

Description automatically generated

* **Roles**

**Graphical user interface

Description automatically generated with low confidence**

* **Transactions**

**Graphical user interface

Description automatically generated with low confidence**

* **Users**

**Table

Description automatically generated**

* 1. **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 възможнна 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** – преставлява 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** - преставлява 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*. Уникланият 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* – прадставлява 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
  1. **Реализация на бизнес логиката и графичен интерфейс (Java, JavaFX)**
  2. **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 заето
    - goodExists – проверка при подадено име, което вече е заето
  + PartnerNameTest - тества коректно ли работи валидаторът за име на партньор
    - partnerNotExists - проверка при подадено име, което не е вече заето
    - partnerExists - проверка при подадено име, което вече е заето
  + 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 дълът 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