

Документация по проект на

Тема № 4: „Билетен Център“

Изготвили: Добрин Севдалинов Петров и Павел Василев Василев

Факултет: Факултет по изчислителна техника и автоматизация

Катедра: Софтуерни и Интернет Технологии

Специалност: Софтуерни и Интернет Технологии

Факултетни номера: 22621635 и 22621644

Съдържание

[Глава 1: Условие на задание 2](#_Toc186294262)

[Глава 2: Анализ на проблема 3](#_Toc186294263)

[Глава 2.1: Функционални изисквания 3](#_Toc186294264)

[Глава 2.2 Технологии на реализация 4](#_Toc186294265)

[Глава 2.3 Структура на проекта 5](#_Toc186294266)

[Глава 2.4 Дефиниция на модулите на системата 6](#_Toc186294267)

[Глава 3: Проектиране на системата 7](#_Toc186294268)

[Глава 3.1: Проектиране на отделните модули 7](#_Toc186294269)

[Глава 3.2: UML диаграми 11](#_Toc186294270)

[Глава 3.3: ER диаграми 14](#_Toc186294271)

[Глава 4: Реализация на система 15](#_Toc186294272)

[Глава 4.1: Реализация на база от данни 15](#_Toc186294273)

[Глава 4.2: Реализация на слоя за работа с база от данни 15](#_Toc186294274)

[Глава 4.3: Реализация на бизнеслогика и графичен дизайн 16](#_Toc186294275)

[Глава 4.4: Реализация на модул за регистриране на събития в системата 19](#_Toc186294276)

[Глава 5: Тестови резултати 20](#_Toc186294277)

[Глава 5.1: Junit тестове 20](#_Toc186294278)

[Глава 5.2: Функционални тестове 20](#_Toc186294279)

[Глава 5.3 Интеграционни тестове 22](#_Toc186294280)

[Глава 6: Заключение 23](#_Toc186294281)

# Глава 1: Условие на задание

Да се разработи информационна система, предоставяща услуга за билетен център. Програмата съхранява и обработва данни за разпространение на билети. Системата позволява множествен достъп.

Системата поддържа два вида потребители- администратор и клиенти(организатор, разпространител) с различни роли за достъп до функционалностите в системата.

Операции за работа с потребители:

* Създаване на организатори от администратор;
* Създаване на дистрибутори от администратор;
* Създаване, редактиране на събития с избор на един или списък разпространители от собственик за продаване на билети;
* Поддържане на профили с характеристики на организатори и разпространители (хонорар и др…);
* Рейтинговане на разпространителите;

Системата поддържа операции за работа със събития:

* Добавяне на ново събитие от организатор (вид на събитието, брой места, видове места, цена на билети по видове, ограничение в закупуването на билет от едно лице и др…);
* Продаване на билети за събитие от разпространител, създаване на формуляр за закупуване (информация за купувача, избор на място и др…);

Системата поддържа справки по произволен период за:

* Разпространител (разпродадени билети от различни видове събития, данните на разпространителя, рейтинг);
* Събития (дата, статус, местоположение, и др…);

Организатор на събитие достъпва справки само за събитията, на които е организатор. Организаторът достъпва справки за всички организирани от него събития. Разпространителят има право на справки за събития, които са му заявени за дистрибуция от организатор.

Системата поддържа известия за събития:

* Новопостъпила заявка за събитие (в профила на разпространител);
* Периодично уведомление за продадени билети от събитие (в профила на собственика);
* Наближаващо събитие с не продадени билети (собственик, разпространител на билети);

# Глава 2: Анализ на проблема

## Глава 2.1: Функционални изисквания

За изпълнението на този проект първата важна стъпка е да изберем нужните компоненти, които ще използваме за осъществяването на правилна и пунктуална функционалност за нашата програма. От условието се подразбира, че ще работим с данни, които можем да добавяме, обработваме и извличаме, за да можем да изпълним нужните операции. За целта трябва да се създаде база данни, която ще съхранява нашата информация, която използваме. Чрез базата данни можем да боравим с процедури и тригери, които ще подпомогнат функциите за обработка на информацията в базата и така по-лесно ще можем да извикваме нужната функция и да й зададем необходимите аргументи за изпълнение в програмата. За да може, обаче, програмата да борави с базата данни, ще е нужно да се конфигурира драйвер към pom файла на нашата програма. Ние използвахме база данни на Oracle, така че конфигурацията за работа с база данни- Java Database Connectivity (jdbc), трябва също да се отнася за Oracle. Чрез нея, програмата има достъп до класове с функции, свързани с изпълнение на обработка на информация в база данни в Java среда.

Както споменахме, в pom файл съхраняваме конфигурации, които ни позволяват да използваме съвкупност от обекти и функции. За да можем да постигнем това, в програмата трябва да се вгради автоматичен билдър, който ще конфигурира желаните драйвери. В нашия проект е използван Maven билдър. Maven е инструмент за автоматизиране на изграждането и управление на проекти с отворен код, широко използван за Java приложения. Като инструмент за автоматизация на изграждане, той автоматизира компилирането на изходния код и управлението на зависимостите, сглобява двоични кодове в пакети и изпълнява тестови скриптове. Maven превежда и пакетира вашия изходен код, така че да стане изпълнимо приложение.

В условието се описват операции за въвеждане и обработка на данни под формата на формуляр. За да можем да изпълним това, ще ни е нужен потребителски интерфейс, който може да изобразява нужните компоненти, в които ще попълваме и извеждаме информацията от базата данни. За Java това може да се постигне чрез използването на JavaFX. JavaFX представлява набор от графични и медийни пакети, които позволяват на разработчиците да проектират, създават, тестват, отстраняват грешки и внедряват богати клиентски приложения, които работят последователно в различни платформи. Той се състои от fxml файлове, съдържащи компоненти като текстови полета, таблици, бутони и др. Тези файлове могат да се управляват от класове с обозначени анотации за fxml променливи и функции, които се наричат още контролери. След създаването на съответния контролер, той задължително трябва да се посочи в fxml файла, за който се отнася под атрибут fx:controller.

## Глава 2.2 Технологии на реализация

От най-голямо значение, за да може да работи нашата програма, е да се създаде база данни, в която да се съхраняват необходимите данни. За целта, първо е нужно да планираме как ще я съставим и как ще взаимодействат елементите в нея. За целта ще ни е нужна програмата за моделиране на база данни- *Datamodeler*. Чрез Datamodeler, ние можем да визуализираме нашите таблици с техните данни, описани по тип и как влияят на таблицата- като главен ключ в самата нея или чужд в друга таблица използвайки нейните данни. Също така може да се опишат и връзките между обектите за по-ясна представа как трябва да подготвим таблиците преди да ги свързваме помежду си. В тази програма могат да се създадат 2 вида диаграми:

* Логически
* Релационни

След като имаме плана създаване на база данни, може да започнем да я съставяме. За нашия проект сме използвали програмата за създаване на база данни в Orcale сървър- *SQLDeveloper*. Чрез SQLDeveloper можем да създадем връзки към сървър, които ще съхраняват данните на посочения в нея потребител. Използвайки sql кодиране, можем да създадем таблици с данни, процедури за обработка на данните в тях и тригери, които автоматично ще изпълняват действие, посочено от администратора на връзката, в определени случаи. След като сме създали всичко необходимо в тази връзка, ще я свържем към нашата програма.

Както споменахме, ще ни бъдат нужни fxml файлове, чрез които ще можем да визуализираме потребителски интерфейс за работа с програмата. В него компоненти трябва да са подредени в чист и приличен вид, който ще улесни начина на употреба на потребителя. Част от нужните компоненти за нашата програма включват текстови полета, бутони и етикети. Чрез тях можем съответно да въвеждаме данни, да изпълняване функция и да опишем каква информация е нужна на настоящия файл. За изваждане на информация може да използване обект Alert, който извежда прозорец с информация, която можем да опишем като грешки, информация, предупреждение и др. Ако желаем да изведем множество данни от таблица, тогава най-подходящо за употреба ще е *table view*. Table view представлява таблица, чиито данни се съхраняват в обект със съответните му променливи, които можем да достъпваме и обработваме, както пожелаем, спазвайки предназначения им тип. Обектите в таблицата се съхраняват в лист предназначен за колекциите на fxml файловете (Observable List). За избор от различни възможности, изпилвахме радио бутони и комбо-боксове. Всичките тези компоненти могат да бъдат създадени и обработени на ръка, но за по-бързо и ефикасно ние използвахме програмата *Scene Builder*. Scene Builder ни позволява да създадем и визуализираме fxml файлове с опцията да вмъкваме различни видове компоненти и да манипулираме тяхната външност и функционалност за нашите нужди.

За да можем, обаче, да минаваме през fxml файловете, трябва всеки път да зареждаме в сцена URL адреса на следващия ни файл в проекта. За целта използвахме един от поведенческите модели на Обектното-ориентирано програмиране- *Command*. Чрез command модела ние можем в един обект да съхраним функциите описани в интерфейс, след което да заредим всяка една функция в отделен обект и накрая, чрез обект наречен *invoker* да извикаме изпълнението на тази функция, без да се обръщаме към останалите и да запълваме паметта на устройството. За нашата програма, функциите, които ще са ни нужни за този модел са за смяна на файл и за затваряне на настоящия. За аргументи използвахме адреса на fxml файла от ресурсите на проекта и променлива от тип *Action Event*, която следи в кой fxml файл се изпълнява съответното действие.

Програмата ще работи с няколко вида потребители, така че ще е нужно при логване или извеждане на потребители, ни трябва ефикасен начин за тяхното създаване в програмата. За целта използвахме моделът за създаване на обекти в Обектното-ориентирано програмиране- *Abstract factory*. Чрез abstract factory модела програмата лесно и ефикасно може да създаде потребители, спрямо информацията, която сме извадили от базата данни. Всеки един обект на потребител ще бъде наследен от абстрактен клас *User* с абстрактни методи за връщане на общите данни между тях- идентификатор, име, потребител и парола. Във фабриката за потребители ще зареждаме резултата от справките свързани с потребители и спрямо каква информация предоставя, ще я изпрати съответната фабрика, която ще зареди данните в програмата и ще върне съответния потребител.

Един проблем, през който може да мине програмата е пренасянето на данни на настоящия влязъл потребител и товаренето на паметта на устройството с ненужни инстанции на обекти. Този проблем е лесно разрешим, чрез използването на шаблона на проектиране- *Singleton*. Чрез този модел, всеки обект съдържа статична инстанция, която винаги се инициализира при първо повикване. При следващите повиквания, се връща адреса на тази инстанция на обекта като можем да обработваме или извличаме информацията в нея. В нашия проект сме използвали singleton шаблона за обектите свързани с потребителите, връзката към базата данни и други данни, които ще са ни нужни в различни обекти на програмата.

Горно описаните шаблони подпомагат за спазването на SOLID принципа на Обектното-ориентирано програмиране:

* Единна отговорност (Single Responsibility Principle);
* Отворени за разширение, затворени за модификация (Open-Closed Principle);
* Принципа на Лисков (Liskov Substitution Principle);
* Интерфейсна сегрегация (Interface Segregation Principle);
* Принцип на инверсия на зависимост (Dependency Inversion Principle);

## Глава 2.3 Структура на проекта

За нашия проект е нужно да определим как да структурираме проекта, за да може да бъде лесен за употреба, ефикасен в работата си и правилно да функционира. За целта, както споменахме сме използвали fxml файлове, които са логически свързани помежду си, за да покажат последователност от действия. Всичко започва с fxml доставящ опцията за влизане в системата с уникални за потребителя потребителско име и парола. Ако се въведат не съвпадащи данни, за който и да е потребител, програмата ще изведе съобщение за не намерен потребител. В противен случай, програмата първо ще извлече данните на потребителя със съвпадаща информация за логване, след което ще затвори началния fxml файл и ще зареди, подходящия за този потребител файл с всички функции, на които те имат право да изпълнят в програмата. По този начин могат да бъдат достъпвани и останалите fxml файлове като всеки ново отворен е с фокус, което му позволява да работи приоритетно, докато не се затвори, и предотвратява предишно отворените файлове да не пречат на работата на фокусирания файл, докато и те са още отворени. За да може да добавим функционалност на всеки един fxml файл, трябва да им се посочи контролер обект. Чрез тези контролер обекти, можем да манипулираме елементите на fxml файл и да добавяме функции, които да връщат резултат във или извън него. В тези обекти използваме функциите на други класове, за да можем програмата да работи с базата данни и да навигира през всички fxml файлове логически.

## Глава 2.4 Дефиниция на модулите на системата

Нашата програма се разделя на 4 главни модула:

* База данни
* Java код
* Ресурси
* Тестове

Модула на базата данни представлява връзка към базата данни, използвана да съхранява нужната информация в таблици. Тя се състои от тези таблици, процедури, чрез които програмата може да изпълни различни функции в самата нея и тригери, които следят при специфични случаи на обработка на данни да изпълнят определени действия, зададени от администратора. Така програмата ще има защитено място за съхранение на данни, което може да се обработва само от потребителите чрез програмния интерфейс.

В модула за Java код се съхраняват обекти с функциите на програмата. Тези обекти включват контролерите на fxml файловете, функциите за свързване на програмата с база данни, класове с информация за всеки потребител, събитие и др., и функции за обхождане на fxml файлове. Всеки обект е логически подреден в пакети за по-чиста подредба на проекта и лесно намиране при нужда на разширение на кода. В този модул повечето обекти работят помежду си, за да могат да осъществят желаната функционалност на програмата.

В модула за ресурси се записват файловете, които програмата използва, за да може да извърши желано действие. В нашия проект, това включва fxml файловете, които са използвани като потребителски интерфейс на програмата, и файл за конфигурация за логер на грешки. Като посочим пътеката, която води до съответния ресурс, в java код модула, програмата може да го манипулира, за да постигне желан резултат.

Последния модул служи за тестване на функционалния модул на програмата в различни сценарии. Като се извикаме инстанция на съответния обект, който ще тестваме, можем да проверим дали неговите функции работят, както подобават, като след изпълнение проверим какъв ще е очаквания резултат. Колкото повече тестове с различни сценарии за една функция има, толкова ще сме по-сигурни, че предоставената функция ще работи правилно.

# Глава 3: Проектиране на системата

## Глава 3.1: Проектиране на отделните модули

За нашата програма ще са нужни 13 таблици, съхраняващи нужните данни за изпълнение. След създаването им в SQLDeveloper връзката, в която се съхраняват на сървъра, ще изглежда по следния начин:

Картина, която съдържа текст, компютър, екранна снимка, софтуер

Описанието е генерирано автоматичноКартина, която съдържа текст, компютър, екранна снимка, софтуер

Описанието е генерирано автоматично

Фиг. 1: Примерна връзка на база данни с таблици и техните данни

Също така за потребителския интерфейс ще са ни нужни 14 fxml файла. При стартиране, програмата ще изведе fxml файл за логване, който ще отведе потребителя в подходящия за тях fxml файл с указани права в програмата. В случай на логване на организатор, файловете през които може премине изглеждат по следния начни:

Картина, която съдържа текст, софтуер, екранна снимка, Мултимедиен софтуер

Описанието е генерирано автоматично

Фиг. 2: FXML файл за логване на потребител

Картина, която съдържа текст, екранна снимка, софтуер, Мултимедиен софтуер

Описанието е генерирано автоматично

Фиг. 3: FXML файл с функциите на организатор в програмата

Картина, която съдържа текст, екранна снимка, софтуер, Мултимедиен софтуер

Описанието е генерирано автоматично

Фиг. 4: FXML файл с функция за вмъкване на събитие от организатор

Картина, която съдържа текст, софтуер, екранна снимка, Мултимедиен софтуер

Описанието е генерирано автоматично

Фиг. 5: FXML файл с функция за редактиране на събитие от организатор

Картина, която съдържа текст, софтуер, екранна снимка, Мултимедиен софтуер

Описанието е генерирано автоматично

Фиг. 6: FXML файл с функция за оставяне на рейтинг на дистрибутор от организатор

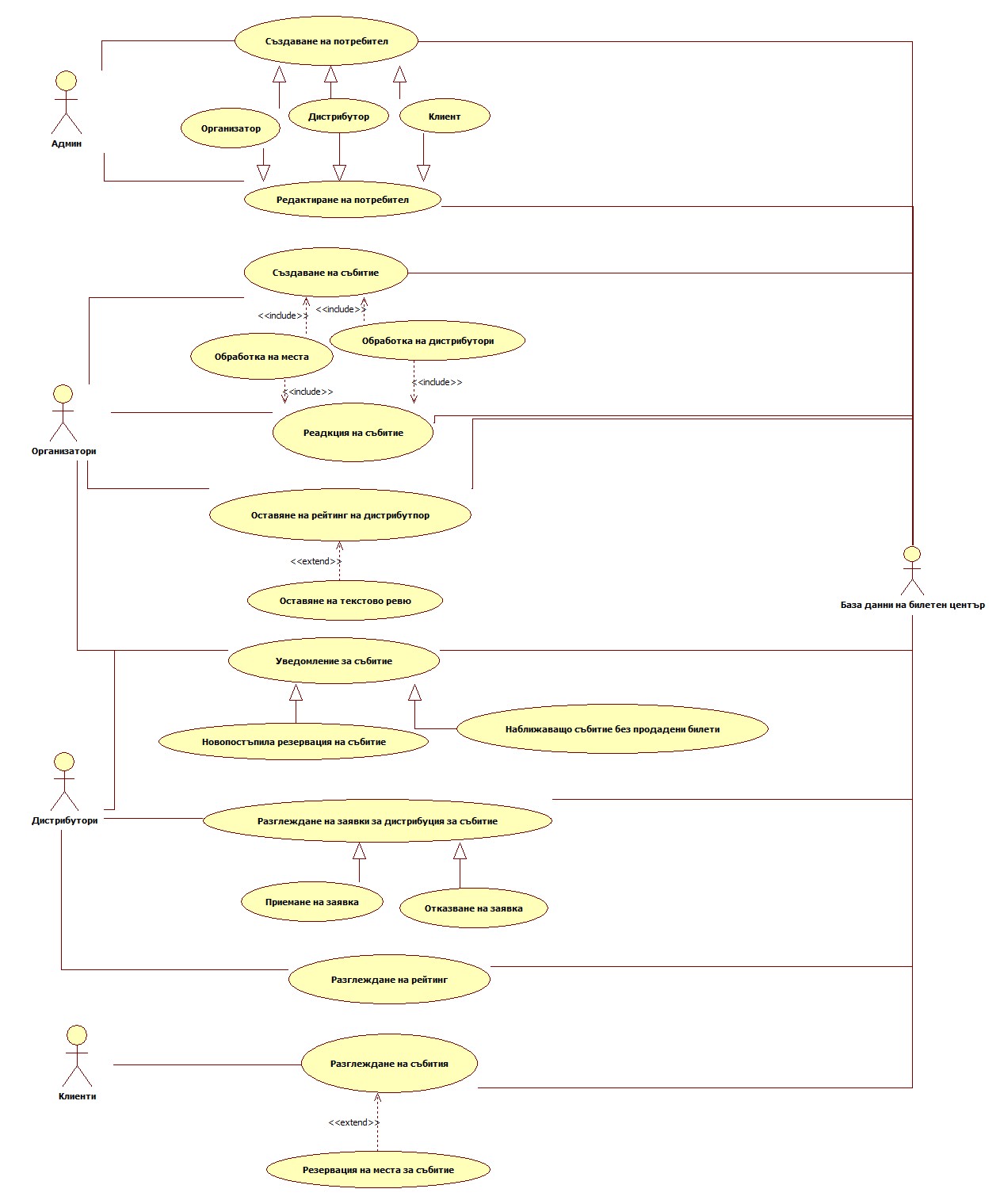
Картина, която съдържа текст, софтуер, екранна снимка, Мултимедиен софтуер

Описанието е генерирано автоматично

Фиг. 7: Съобщение за извеждане на продадени билети на събития на организатор

## Глава 3.2: UML диаграми

За да знаем как трябва да работи програмата, първо трябва да разучим какво трябва да прави всеки един потребител в нея. Това може да се визуализира и анализира, чрез употребата на *Use Case* диаграми. Use Case диаграмите представляват диаграми с актьори и случаи, които описват действията, които могат да се изпълнят в един проект и да покаже всеки как участва в процеса на работа. За нашето задание Use Case диаграмата ще изглежда по следния начин:

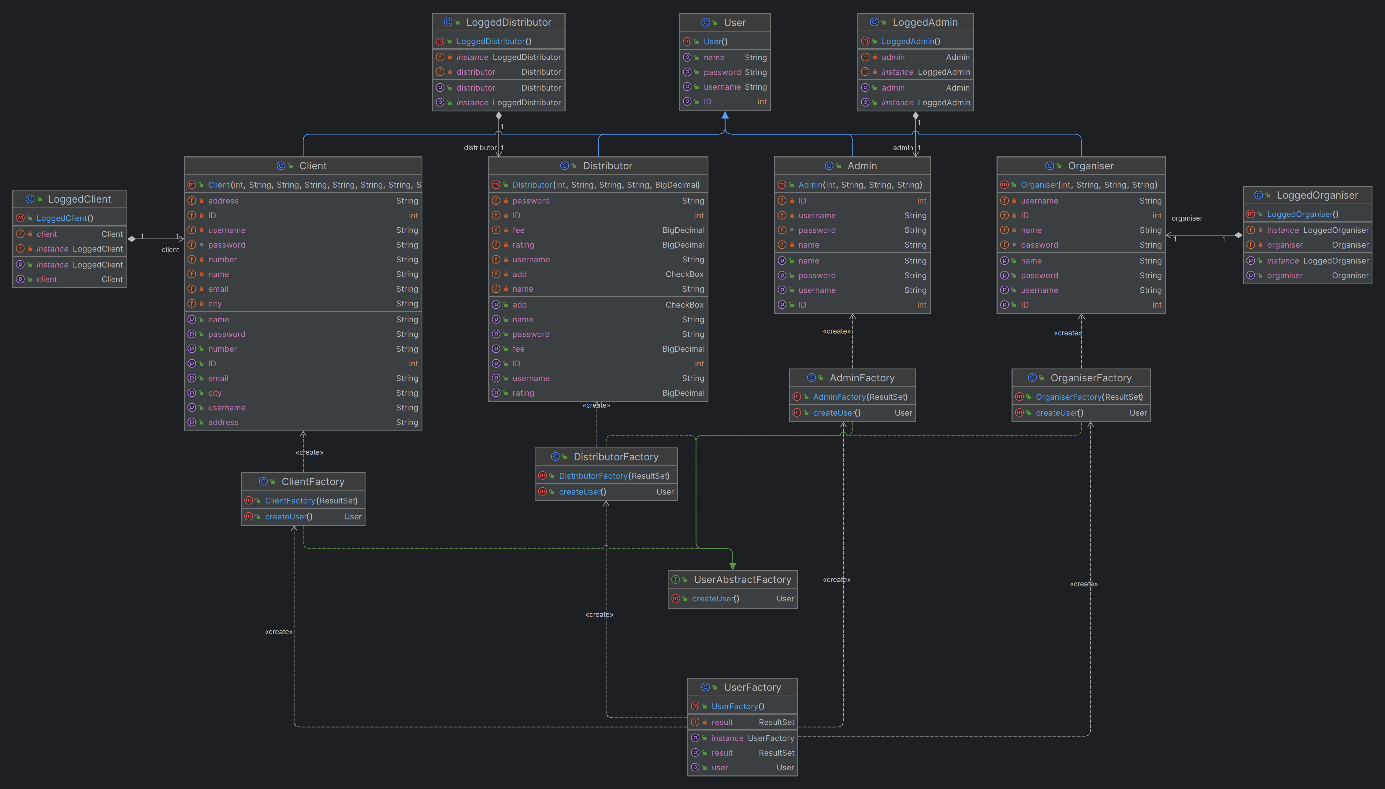


Фиг. 8: Use Case диаграма на задание

UML диаграмите също са полезни и за описване на връзки на взаимодействие между обекти в програмата. Този тип диаграма се нарича *Class* диаграма. Следващите фигури, ще опишат взаимодействията между обектите за зареждане на потребител в системата и промяна на fxml файл:

Картина, която съдържа текст, екранна снимка, диаграма, Шрифт

Описанието е генерирано автоматично

Фиг. 9: Class диаграма за промяна на fxml файл

Фиг. 10: Class диаграма за зареждане на потребител в програмата

## Глава 3.3: ER диаграми

Както споменахме по-рано, за да можем да създадем база данни, ни е нужно да планираме как ще са свързани нашите данни в нея. За целта, можем да опишем това в ER диаграма или още наречено- логическа диаграма. Следващия пример показва как би изглеждала ER диаграмата на нашата база данни:

Картина, която съдържа текст, диаграма, екранна снимка, План

Описанието е генерирано автоматично

Фиг. 11: ER диаграма на база данни

# Глава 4: Реализация на система

## Глава 4.1: Реализация на база от данни

Картина, която съдържа текст, диаграма, Шрифт, План

Описанието е генерирано автоматичноЗа нашето задание ще ни бъде нужна база данни с 14 таблици [12]. За целта, ние създадохме Oracle база данни с 4 таблици съхраняващи информацията за съответно всеки потребител на програмата, 1 таблица с информация за събитията, посочваща града, където ще се проведе и неговия организатор, таблица с видовете места налични в системата, 2 таблици с информация за събитието, дистрибутора и местата, предлагани за това събитие, таблица с данни за рейтинга на дистрибуторите с данни за оценяващия организатор и таблица за продадените билети с информация за мястото на събитието клиента и колко билети са закупени. Всяка таблица е с главен ключ за описване на данните в други таблици и почти всеки е използван в 1 или повече таблици, с изключение на администратора, тъй като неговите данни не трябва да бъдат достъпвани от останалите потребители.

Фиг. 12: Релационна диаграма на база данни

## Глава 4.2: Реализация на слоя за работа с база от данни

За създаването на таблиците с техните процедури и тригери, използвахме *SQLDeveloper*. Чрез него всяка една таблица има процедура за вмъкване и обработка на данните, както и функции за извличане на определени данни от различни таблици в зависимост за какво ни е нужна информацията. Цялата база данни е на локален сървър, което означава, че данните ще са достъпни само на устройството, на което се намира сървъра.

След като всички нужни функции са създадени в базата данни, свързахме връзката от сървъра към проекта с URL адреса, потребителското име и паролата на администратора. След успешно свързване, чрез вграждането на jdbc за Oracle, създадохме обект с инстанция за връзката към базата данни, която можем да извикваме за достъп до базата данни. В този обект можем да контролираме връзката като й придадем функции за отваряне и затваряне. Така позволяваме да достъпваме всичко нужно в базата, и когато приключим действията с нея можем да я затворим, за да не освободим паметта.

След като сме се свързали към базата данни, можем да използваме нейните процедури в Java среда. Повечето процедури, връщат информация за потребители или събития, които чрез jdbc зареждаме в *ResultSet*. ResultSet представлява вид лист работещ като опашка, в който можем да съхраним данни от справки на база от данни. Щом намерим нужния резултат, можем да го обходим и заредим в съответния му за програмата обект. Ако е потребител, ще заредим резултата в Abstract Factory модела за създаване на потребители, в противен случай, ще за създадем нова инстанция за обекта, който ни е нужен. По този начин данните могат да се запазват в сървър без да се страхуваме от загубата им и лесно можем да ги заредим в програмата.

## Глава 4.3: Реализация на бизнеслогика и графичен дизайн

За графичния дизайн на нашата програма използвахме SceneBuilder, за да обработим нашите fxml файлове с техните нужни компоненти. Всеки един файл се съхранява в ресурс модула на проекта [13] и за да е по-лесно описването на всеки адрес на файл, създадохме обект съхраняващ всички адреси от ресурсите. В него всички променливи са статични, което предотвратява промяна по време на работата. Само при извикване на този клас с определения му адрес лесно и ефикасно можем да задаваме кой fxml файл е нужен да се отвори чрез изпилването на *Command* шаблона.

За Java код модула, всеки обект е подреден в пакет за по-лесно откриване [14]. На първо място имаме апликацията на нашата програма, която стартира целия код като зарежда първия fxml файл за влизане на потребител в системата. След него имаме пакет с всички адреси към ресурсите на нашата програма. Както споменахме, те са статични и служат само за извикване при нужда за смяна на файл. Също така съхранява адреса към ресурса за конфигурация на логер. В следващия пакет съхраняваме контролерите на fxml файловете като всеки е подреден в под пакет за съответния потребител който се отнася- администратор, организатор, дистрибутор и клиент. Всеки контролер съдържа специални идентификатори, описани в съответния му fxml файл, чрез които можем да контролираме указания компонент. Също така съдържат специални функции, които могат да се изпълнят при определено действие на посочения компонент. Всички тези променливи и функции се описват в код чрез анотацията *@FXML*. Следващия пакет се отнася за връзката между програмата и базата данни. Благодарение на jdbc драйвера, имаме променлива, чрез която контролираме връзката към базата и й придаваме функции за отваряне и затваряне. Този клас също е използван като Singleton, за да спестим паметта на устройството. В следващите пакети съхраняват данните за събитията и техните места. Когато имаме резултат от базата данни, съхраняващ данните ги зареждаме в тези обекти и ги използваме за нужната цел на програмата. В следващите пакети съхраняваме двата шаблони, използвани в програмата- *Command* и *Abstract Factory*.

Command шаблона го използваме за създаване на команди за смяна на fxml файл. За да може да се изпълни ни трябва 2 интерфейса- един за описване на функциите, които ще изпълняват и един за изпълнение на дадената команда. В нашия случай ще имаме функции за смаяна на fxml файл и за затваряне. Този интерфейс ще се имплементира в един обект като опише как се изпълняват:

@Override  
public void changeScene(String fxml, ActionEvent event) throws IOException {  
 if(fxml!=null) {  
 root = FXMLLoader.*load*(getClass().getResource(fxml));  
 stage = new Stage();  
 scene = new Scene(root);  
 stage.setScene(scene);  
 stage.initModality(Modality.*APPLICATION\_MODAL*);  
 stage.show();  
 }else{  
 *logger*.error("Couldn't open fxml file");  
 }  
}  
  
@Override  
public void closeScene(ActionEvent event) {  
 stage=(Stage)((javafx.scene.Node) event.getSource()).getScene().getWindow();  
 stage.close();  
}

Следващите обекти, които ще ни трябват ще имплементират интерфейса за изпълнение на команда като в него съхраняват главната имплементация на функциите, които може да изпълни. Във функцията за главния интерфейс извикваме само функцията, която искаме да се изпълни. Накрая имаме обект, който служи като изпълнител, който ще съхранява в себе си командата, която искаме да се изпълни. Чрез този шаблон изпълняваме един от принципите на Обектното-ориентирано програмиране за разширение на код без неговата промяна. Това означава, че към този шаблон можем да добавяме колкото си искаме функции, без да имаме нужда да променим вече съществуващия в него код и неговата функционалност. В случая на Abstract factory, той се използва за създаване на потребители от база данни в обект на програмата. За неговата цел, ще е нужно да имаме абстрактен клас представляващ потребител с абстрактни функции за връщане на имена, потребител, парола и идентификатор в базата данни. Също така ще ни е нужен интерфейс, който ще върне абстрактния обект. Като модели ще наследим главния ни потребител към различните видове, които може да е:

* Администратор
* Организатор
* Дистрибутор
* Клиент

Всеки един от тях се разширява с данни, определени за неговата нужна информация в базата данни и имплементират абстрактните функции на родителския им клас. Следващата стъпка е да създадем фабрика за всеки вид потребител като имплементират интерфейса. При инициализацията на всяка фабрика също трябва да заредим нужната информация от базата данни като тя ще е съхранена в ResultSet. След като заредим информацията, фабриката ще изпълни имплементирания метод като върне определения за нея потребител с техните данни. За да можем да извършваме това действие, без да викаме всяка отделна фабрика из програмата, ще ни е нужна една обща фабрика, която ще разпредели, която фабрика ще зареди своя потребител, спрямо резултата, който предадем. Всеки ResultSet идва с имената на данните, които предоставя и чрез извличане на мета данните, можем да проверим за кой потребител се отнасят данните и така да изпълним функцията на правилната фабрика:

public User getUser() throws SQLException, ClassNotFoundException {  
 UserAbstractFactory abstractFactory = null;  
 DBConnection connection=DBConnection.*getInstance*();  
 connection.connect();  
 ResultSetMetaData md= getResult().getMetaData();  
 if(md.getColumnName(1).equalsIgnoreCase("Admin\_ID")){  
 abstractFactory=new AdminFactory(getResult());  
 } else if (md.getColumnName(1).equalsIgnoreCase("Organiser\_ID")) {  
 abstractFactory=new OrganiserFactory(getResult());  
 }  
 else if (md.getColumnName(1).equalsIgnoreCase("Distributor\_ID")) {  
 abstractFactory=new DistributorFactory(getResult());  
 }  
 else if (md.getColumnName(1).equalsIgnoreCase("Client\_ID")) {  
 abstractFactory=new ClientFactory(getResult());  
 }  
 connection.closeConnection();  
  
 return abstractFactory.createUser();  
}

Този шаблон също изпълнява принципа на Обектното-ориентирано програмиране за разширение на код без неговата промяна. При нужда лесно можем да добавим и различни видове потребители.

Картина, която съдържа текст, екранна снимка, софтуер, Мултимедиен софтуер

Описанието е генерирано автоматично

Фиг. 13: Ресурсния модул на проект

Картина, която съдържа екранна снимка, текст, софтуер, Мултимедиен софтуер

Описанието е генерирано автоматично

Фиг. 14: Под пакети в модул за Java код

## Глава 4.4: Реализация на модул за регистриране на събития в системата

За да можем да регистрираме събития като грешки, предупреждения и др. в нашата програма, ние сме имплементирали *log4j*. Чрез log4j ние можем да създадем логер файл, който ще съхранява всички събития, които сме му задали с информацията, която ни е нужна. В нашето задание сме създали *properties* файл, който при пускане на програмата конфигурира как ще работи нашия логер файл. Ние сме го задали да записва само грешките появили се по време на изпълнение на програмата като извежда нишката, която изведе съобщението, нивото на важност на съобщението, името на логера, което е генерирало съобщението, допълнителната информация за логера (като ID) и съобщението на логера, зададено от програмиста:

log4j.rootLogger=ERROR, console, file  
  
log4j.appender.console=org.apache.log4j.ConsoleAppender  
log4j.appender.console.layout=org.apache.log4j.PatternLayout  
log4j.appender.console.layout.ConversionPattern=[%t] %-5p %c %x - %m%n  
  
log4j.appender.file=org.apache.log4j.RollingFileAppender  
log4j.appender.file.layout=org.apache.log4j.PatternLayout  
log4j.appender.file.layout.ConversionPattern=[%t] %-5p %c %x - %m%n  
log4j.appender.file.File=application.log

Всички грешки се записват във файл *application.log*.

# Глава 5: Тестови резултати

## Глава 5.1: Junit тестове

Junit тестовете представляват сценарии, през които можем да пуснем Java функции на един обект като така проверяваме дали кода на програмата функционира правилно. За да можем да създаваме тестове в нашата програма трябва първо да включим нужните зависимости в нашия *pom* файл. След неговото вграждане ние можем за всеки обект в нашата програма да създаден тестов обект с функции, които служат като наши сценарии за тестване на кода. Колкото повече сценарии имаме и те се изпълняват успешно, толкова повече ще сме сигурни, че нашия код работи правилно.

## Глава 5.2: Функционални тестове

Чрез функционалните тестове, ние можем да следим дали една функция ще върне определен резултат в специфичен сценарии. За нашата програма тестваме дали при зареждане на резултат в абстрактната фабрика за потребители ще върне правилния потребител:

class UserFactoryTest {  
 private UserFactory userFactory;  
 private DBConnection connection;  
 private CallableStatement stmt;  
 private ResultSet result;  
 @BeforeEach  
 void setUp() throws SQLException {  
 userFactory=UserFactory.*getInstance*();  
 connection=DBConnection.*getInstance*();  
 }  
 @Test  
 void testNoResultSet() throws SQLException, ClassNotFoundException {  
 userFactory.setResult(null);  
 *assertThrows*(NullPointerException.class, () -> {  
 userFactory.getUser();  
 });  
 }  
 @Test  
 void testReturnAdmin() throws SQLException, ClassNotFoundException {  
 connection.connect();  
 stmt=connection.getConnection().prepareCall("CALL FIND\_ADMIN(?, ?, ?)");  
 stmt.setString(1, "project\_admin");  
 stmt.setString(2, "admin123");  
 stmt.registerOutParameter(3, OracleTypes.*CURSOR*);  
 stmt.execute();  
  
 result= (ResultSet) stmt.getObject(3);  
  
 while (result.next()){  
 userFactory.setResult(result);  
 *assertInstanceOf*(Admin.class, userFactory.getUser());  
 }  
  
 connection.closeConnection();  
 }  
 @Test  
 void testReturnOrganiser() throws SQLException, ClassNotFoundException {  
 connection.connect();  
 stmt=connection.getConnection().prepareCall("CALL FIND\_ORGANISER(?, ?, ?)");  
 stmt.setString(1, "john\_lenon");  
 stmt.setString(2, "Johnathan123");  
 stmt.registerOutParameter(3, OracleTypes.*CURSOR*);  
 stmt.execute();  
  
 result= (ResultSet) stmt.getObject(3);  
  
 while (result.next()){  
 userFactory.setResult(result);  
 *assertInstanceOf*(Organiser.class, userFactory.getUser());  
 }  
  
 connection.closeConnection();  
 }  
 @Test  
 void testReturnDistributor() throws SQLException, ClassNotFoundException {  
 Platform.*startup*(() -> {});  
 connection.connect();  
 stmt=connection.getConnection().prepareCall("CALL FIND\_DISTRIBUTOR(?, ?, ?)");  
 stmt.setString(1, "exp\_share");  
 stmt.setString(2, "ShareIsCare");  
 stmt.registerOutParameter(3, OracleTypes.*CURSOR*);  
 stmt.execute();  
  
 result= (ResultSet) stmt.getObject(3);  
  
 while (result.next()){  
 userFactory.setResult(result);  
 *assertInstanceOf*(Distributor.class, userFactory.getUser());  
 }  
  
 connection.closeConnection();  
 }  
 @Test  
 void testReturnClient() throws SQLException, ClassNotFoundException {  
 connection.connect();  
 stmt=connection.getConnection().prepareCall("CALL FIND\_CLIENT(?, ?, ?)");  
 stmt.setString(1, "vankata");  
 stmt.setString(2, "Vankata23");  
 stmt.registerOutParameter(3, OracleTypes.*CURSOR*);  
 stmt.execute();  
  
 result= (ResultSet) stmt.getObject(3);  
  
 while (result.next()){  
 userFactory.setResult(result);  
 *assertInstanceOf*(Client.class, userFactory.getUser());  
 }  
 connection.closeConnection();  
 }  
}

## Глава 5.3 Интеграционни тестове

Чрез интеграционните тестове проверяваме дали компонентите, участващи в една функция работят правилно помежду си. За да демонстрираме този вид тестове, ние създадохме сценарии за абстрактната фабрика да проверява дали правилния идентификатор на потребител е зареден в обекта:

@Test  
void testReturnAdminWithCorrectID() throws SQLException, ClassNotFoundException {  
 connection.connect();  
 stmt=connection.getConnection().prepareCall("CALL FIND\_ADMIN(?, ?, ?)");  
 stmt.setString(1, "project\_admin");  
 stmt.setString(2, "admin123");  
 stmt.registerOutParameter(3, OracleTypes.*CURSOR*);  
 stmt.execute();  
 result= (ResultSet) stmt.getObject(3);  
 while (result.next()){  
 userFactory.setResult(result);  
 *assertInstanceOf*(Admin.class, userFactory.getUser());  
 *assertEquals*(1, userFactory.getUser().getID());  
 }  
 connection.closeConnection();  
}  
@Test  
void testReturnOrganiserWithCorrectID() throws SQLException, ClassNotFoundException {  
 connection.connect();  
 stmt=connection.getConnection().prepareCall("CALL FIND\_ORGANISER(?, ?, ?)");  
 stmt.setString(1, "john\_lenon");  
 stmt.setString(2, "Johnathan123");  
 stmt.registerOutParameter(3, OracleTypes.*CURSOR*);  
 stmt.execute();  
 result= (ResultSet) stmt.getObject(3);  
 while (result.next()){  
 userFactory.setResult(result);  
 *assertInstanceOf*(Organiser.class, userFactory.getUser());  
 *assertEquals*(301, userFactory.getUser().getID());  
 }  
 connection.closeConnection();  
}  
@Test  
void testReturnClientWithCorrectID() throws SQLException, ClassNotFoundException {  
 connection.connect();  
 stmt=connection.getConnection().prepareCall("CALL FIND\_CLIENT(?, ?, ?)");  
 stmt.setString(1, "vankata");  
 stmt.setString(2, "Vankata23");  
 stmt.registerOutParameter(3, OracleTypes.*CURSOR*);  
 stmt.execute();  
 result= (ResultSet) stmt.getObject(3);  
 while (result.next()){  
 userFactory.setResult(result);  
 *assertInstanceOf*(Client.class, userFactory.getUser());  
 *assertEquals*(403, userFactory.getUser().getID());  
 }  
 connection.closeConnection();  
}

# Глава 6: Заключение

С този проект постигнахме създаването на програма, която работи с информация от база данни, обработващи се чрез JavaFX интерфейс. Тя емулира работата на билетен център като може да работи с различни видове потребители, всеки със собствени права в системата, с работещи справки и функции за резервиране на места за събития. За бъдеще програмата може да се подобри като използва база от данни, която се намира в глобален сървър вместо на локална машина, може да се подобри дизайна на потребителския интерфейс и начина, по който програмата представя нужната информацията. Също така може да се подобри формуляра за резервиране на билети като на пръв поглед може да изглежда неясен на пръв поглед.