**Міністерство освіти і науки України**

**Чернівецький національний університет**

**імені Юрія Федьковича**

Факультет математики та інформатики

(повна назва інституту/факультету)

Кафедра математичного моделювання

(повна назва кафедри)

**Курсова робота**

**на тему: Розробка** **системи управління орендуванням автомобілів**

Студента 3 курсу 311 групи

Спеціальності 122 – Комп’ютерні науки

Пасічника В.А\_\_\_\_\_\_\_\_\_

(прізвище та ініціали)

Керівник \_асистент Горбатенко М.Ю

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Національна шкала \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кількість балів: \_\_\_\_Оцінка: ECTS \_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис керівника) (прізвище та ініціали)

Чернівці - 2019

Анотація

В роботі розглядається питання розробки системи автоматизації процесу

управління орендування автомобілів.

Зміст

Вступ……………………………………………………………………………………...4

Розділ 1. Аналіз предметної області …………………………………………………...6

* 1. Розборка технічного завдання……………………………………………………...9
  2. Розробка структур даних для виділення властивостей для об’єктів……………10

Розділ 2. Розробка програмного забезпечення засобами Java……………………….16

* 1. Використання можливостей інтегрованого середовища розробки IntelliJ……..17
  2. Розробка консольного меню взаємодії з користувачем………………………….19

Висновки…………………………………………………………………………….23

Перелік посилань на джерела……………………………………………………...24

Додатки……………………………………………………………………………..25

ВСТУП

Розробка програмного забезпечення - процес, спрямований на створення та підтримку працездатності, якості та надійності програмного забезпечення, використовуючи технології, методологію та практики з інформатики, керування проектами, математики, інженерії та інших областей знання.

Як й інші традиційні інженерні дисципліни, розробка програмного забезпечення має справу з проблемами якості, вартості та надійності. Деякі програми містять мільйони рядків вихідного коду, які, як очікується, повинні правильно виконуватися в умовах, що змінюються. Складність ПЗ рівноцінна зі складністю найбільш складних з сучасних машин.

Протягом кількох десятиліть стоїть завдання пошуку повторюваного, передбачуваного процесу або методології, яка б поліпшила продуктивність, якість і надійність розробки. Одні намагалися систематизувати та формалізувати мало передбачуваний процес. Інші застосовували до нього методи управління проектами та методи програмної інженерії. Треті вважали, що без постійного контролю з боку замовника розробка ПЗ виходить з-під контролю, з’їдаючи зайвий час і кошти. Досвід управління розробкою програм відбивається у відповідних посібниках, звичаях і стандартах. Якщо при розробці використовується декілька стандартів і нормативних документів, то має сенс скласти профіль. Інформатика як наукова дисципліна пропонує і використовує на базі методів структурного програмування технологію надійної розробки програмного забезпечення, використовуючи тестування програм та їх верифікацію на основі методів доказового програмування для систематичного аналізу правильності алгоритмів і розробки програм без алгоритмічних помилок. Дана методологія спрямована на вирішення завдань на ЕОМ, аналогічної технології розробки алгоритмів і програм, використовуваної на олімпіадах з програмування вітчизняними студентами та програмістами з використанням тестування і структурного псевдокоду для документування програм в корпорації IBM з 70-х років. Методологія структурного проектування програмного забезпечення може використовуватися з застосуванням самих різних мов і засобів програмування для розробки надійних програм самого різного призначення. Одним з таких проектів була розробка бортового програмного забезпечення для космічного корабля «Буран», в якому вперше використовувався бортовий комп'ютер для автоматичного управління апарату, яка виконала успішний старт і посадку космічного корабля. При виборі методології розробки програмного забезпечення слід керуватися тим, що складність методології рівноцінна з складністю структури програмного продукту, і невиправдана для продукту даної складності складність методології тільки невиправдано збільшить вартість розробки. Прикладом сучасної методології проектування може бути проблемно-орієнтоване проектування.

В даній курсовій роботі буде використано проблемно орієнтовану модель, на основні аналізу такої предметної області як “Система оренди автомобілів”. Аналіз буде проведено використовуючи ООП підхід, який передбачатиме наступні цілі:

* пошук об'єктів предметної області та виділення їх властивостей буде відображено в розділі першому;
* розробка консольного варіанту інтерфейсу користувача для взаємодії
* розробка алгоритмів та базового функціоналу для вирішення основних задач які будуть виділені при аналізі предметної області.
* розробка можливості збереження результатів роботи програми в базу даних у вигляді файлу який буде основних сховищем даних.
* тестування роботи програми в різних режимах які також будуть виділені при аналізі предметної області

В результаті виконання даної курсової роботи буде спроектовано та розроблено програмне забезпечення яке буде відповідати мінімально необхідним вимогам для демонстрування можливостей його розвитку в майбутньому.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

Основною задачею даного розділу є аналіз предметної області, згідно варіанту це – система орендування автомобілів. При проектуванні було використано об’єктно орієнтований метод проектування.

Об'єктно-орієнтоване проектування — це метод програмування, заснований на поданні програми у вигляді сукупності взаємодіючих об'єктів, кожен з яких є екземпляром певного класу, а класи є членами певної ієрархії наслідування.

Спочатку пишуть клас, а на його основі при виконанні програми створюються конкретні об'єкти (екземпляри класів). На основі класів можна створювати нові, які розширюють базовий клас і таким чином створюється ієрархія класів.

Клас – це спеціальна конструкція, яка використовується для групування пов’язаних змінних та функцій. При цьому згідно з термінологією ООП глобальні змінні класу(члени-змінні) називаються полями даних, а члени-функції називають методами класу. Створений та ініціалізований екземпляр класу називають об’єктом класу. На основі одного класу можна створити безліч об’єктів, що відрізняються один від одного своїм станом.

На думку Алана Кея, якого вважають одним з «батьків-засновників» ООП, об'єктно-орієнтований підхід полягає в наступному наборі основних принципів:

* все є об'єктами;
* всі дії та розрахунки виконуються шляхом взаємодії (обміну даними) між об'єктами, при якій один об'єкт потребує, щоб інший об'єкт виконав деяку дію. Об'єкти взаємодіють, надсилаючи і отримуючи повідомлення. Повідомлення — це запит на виконання дії, доповнений набором аргументів, які можуть знадобитися при виконанні дії.
* кожен об'єкт має незалежну пам'ять, яка складається з інших об'єктів.
* кожен об'єкт є представником (екземпляром, примірником) класу, який виражає загальні властивості об'єктів.
* у класі задається поведінка (функціональність) об'єкта. Таким чином усі об'єкти, які є екземплярами одного класу, можуть виконувати одні й ті ж самі дії.
* класи організовані у єдину деревоподібну структуру з загальним корінням, яка називається ієрархією успадкування. Пам'ять та поведінка, зв'язані з екземплярами деякого класу, автоматично доступні будь-якому класу, розташованому нижче в ієрархічному дереві.

Таким чином, програма являє собою набір об'єктів, що мають стан та поведінку. Об'єкти взаємодіють використовуючи повідомлення. Будується ієрархія об'єктів: програма в цілому — це об'єкт, для виконання своїх функцій вона звертається до об'єктів що містяться у ньому, які у свою чергу виконують запит шляхом звернення до інших об'єктів програми. Звісно, щоб уникнути безкінечної рекурсії у зверненнях, на якомусь етапі об'єкт трансформує запит у повідомлення до стандартних системних об'єктів, що даються мовою та середовищем програмування. Стійкість та керованість системи забезпечуються за рахунок чіткого розподілення відповідальності об'єктів (за кожну дію відповідає певний об'єкт), однозначного визначення інтерфейсів між об'єктної взаємодії та повної ізольованості внутрішньої структури об'єкта від зовнішнього середовища (інкапсуляції).

Взявши за основу вище згадані принципи було виділено наступні об’єкти : автомобіль, клієнт, компанія, дилер, замовлення, користувач системи, а також гараж. Кожен із зазначених об’єктів пройшов процес аналізу на пошук властивостей які мінімально необхідну для функціонування даної системи. Опис кожного об'єкта наведено нижче, а саме процес формалізації його в клас.

Також було визначено основні способи зв'язків між об'єктами, а саме такі як агрегація та композиція та наслідування.

В об'єктно-орієнтованому програмуванні під агрегацією (або як її ще називають - делегуванням) мають на увазі методику створення нового класу з уже існуючих класів шляхом їх включення. Про агрегування також часто говорять як про «відносно приналежності» за принципом «у машини є корпус, колеса і двигун».

На базі агрегування реалізується методика делегування, коли поставлена ​​перед зовнішнім об'єктом завдання перепоручается внутрішньому об'єкту, що спеціалізується на вирішенні завдань такого роду.

На відміну від наслідування, в композиції тип відносин є Has-a тобто має (машина має двигун). В наслідуванні ж тип відносин між породженим об'єктом і батьківським є Is-a зв'язком, тобто якщо об'єкт кішка породжено від тварина, то кішка є тварина (cat is a pet).

Вкладені об'єкти нового класу зазвичай оголошуються закритими, що робить їх недоступними для прикладних програмістів, які працюють з класом. З іншого боку, творець класу може змінювати ці об'єкти, не порушуючи при цьому роботи існуючого клієнтського коду. Крім того, заміна вкладених об'єктів на стадії виконання програми дозволяє динамічно змінювати її поведінку. Механізм успадкування такою гнучкістю не володіє, оскільки для похідних класів встановлюються обмеження, що перевіряються на стадії компіляції.

1.1 Розробка технічного завдання

Система орендування автомобілів є підвидом систем в яких основною операцією є надання послуг, тобто взаємодія клієнт-послуга. В даній системі всі послуги буде надавати дилер, який після авторизації в програмному забезпеченні отримує доступ до всіх актуальних даних компанії. Саме програмне забезпечення повинно надати наступні функціональні можливості:

* реєстрація нового транспортного засобу;
* реєстрація нового клієнта;
* авторизація в системі;
* реєстрація в системі нового дилера;
* формування нового замовлення
* виведення списків клієнтів, замовлень, автомобілів;
* збереження даних в аналог бази даних;
* взаємодія користувача через консольне меню

Дилер повинен буде вручну реєструвати нових клієнтів, а також нові автомобілі. Вся робота з новими угодами по оренді автомобілів повинна бути доступна тільки після авторизація дилера в системі. Клієнт не має мати доступу до даних по оренді.

Буде розроблено інтерфейс користувача у вигляді меню із довільним вибором пункту в залежності від того що користувач хоче зробити. Окремі задачі будуть реалізовані у вигляді окремих функцій які будуть викликатися при необхідності. Після вибраного користувачем пункту буде виконана певна функція, після завершення виконання функції на екран буде виведено результат виконання функції і користувач буде повернений у меню вибору пункту. Опис консольного меню буде наведено в деталях в наступному розділі.

Взаємодія клієнта з програмним забезпечення на надому етапі розробки не буде забезпечено, так як консольний варіант програми та обмежені можливості прототипу бази даних не дадуть змоги отримувати доступ до даних.

1.2 Розробка структур даних для виділення властивостей для об’єктів

Поля класів в основному будуть 2 типів даних int(цілочисельна змінна) та string(змінна для зберігання строк), localDateTime (для збереження дат) а також будуть використані можливості агрегації, композиції та наслідування для побудови зв'язків між класами.

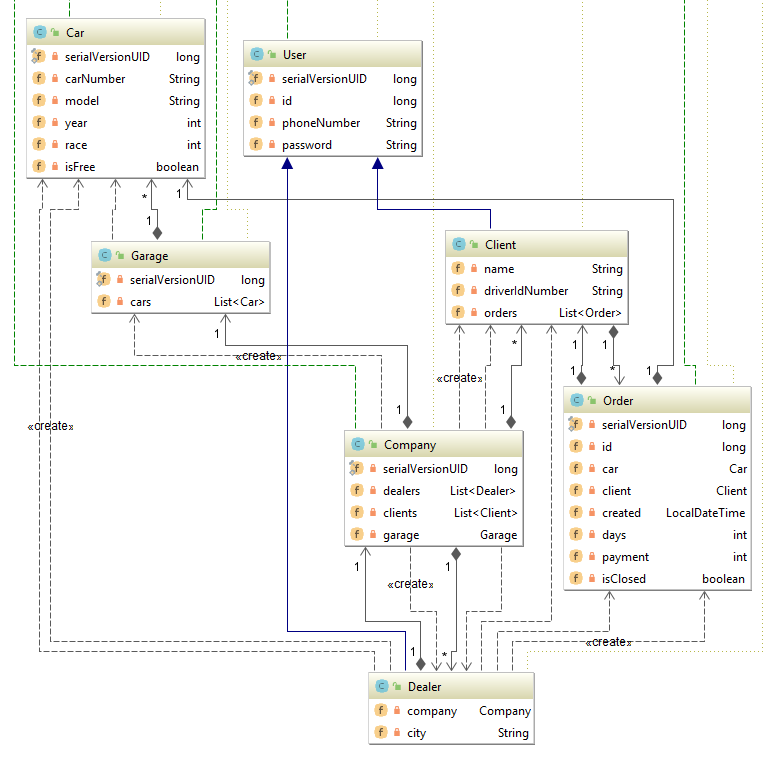


Рисунок 1.1 - UML діаграма взаємозв'язків між об'єктами предметної області

Аналіз створених класів та відповідно обраних для них властивостей будемо проводити згідно **UML** діаграми класів наведеної вище на **рисунку 1.1** починаючи з верхніх класів, так як вони є основою для інших класів. В кожному класі буде присутня властивість **serialVersionUID** типу **long** яка відповідатиме за можливість розпізнавання об'єкта системою після зчитування з бази даних, тому опис цього поля не буде відображатися повторно.

Також в кожному класі буде реалізовано підтримку всіх можливих методів доступу **set**, **get** для кожної властивості використовуючи можливості бібліотеки **Lombok** яка скорочує кількість коду завдяки використанню анотація типу **@Data** яка автоматично буде генерувати ці методи.

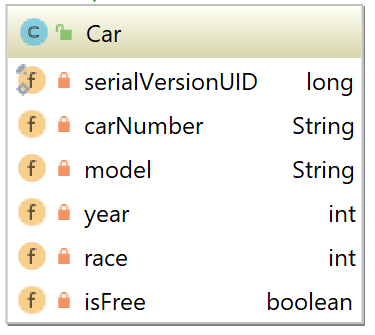


Рисунок 1.2 - Властивості класу Car

Клас Автомобіль (Car) представляє собою інформаційну модель реального автомобіля який володіє наступними властивостями:

* carNumber - відображає реєстраційний номер в реальному житті;
* **model** - виробник та модель автомобіля;
* **year** - рік випуску автомобіля;
* **race** - пробіг автомобіля на момент реєстрації в системі
* **isFree-** відображає чи автомобіль доступний для орендування

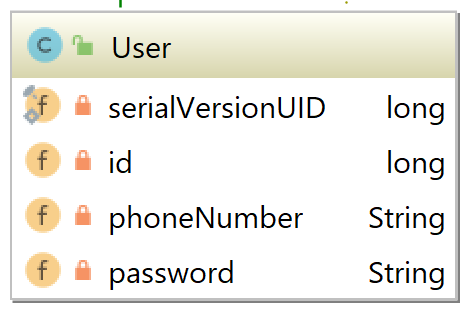


Рисунок 1.3 - Властивості класу User

Клас користувач (User) представляє собою інформаційну модель користувача який володіє наступними властивостями:

* id- унікальний номер користувача в системі;
* **phoneNumber**- номер мобільного телефону який слугуватиме логіном;
* **password**- пароль для входу систему;

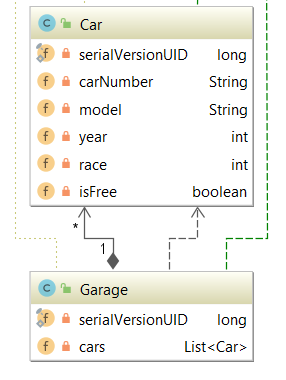


Рисунок 1.4 - Властивості класу Garage

Клас гараж (Garage) представляє собою інформаційну модель який володіє наступними властивостями:

* **cars -** список доступних автомобілів який агрегований в даний клас

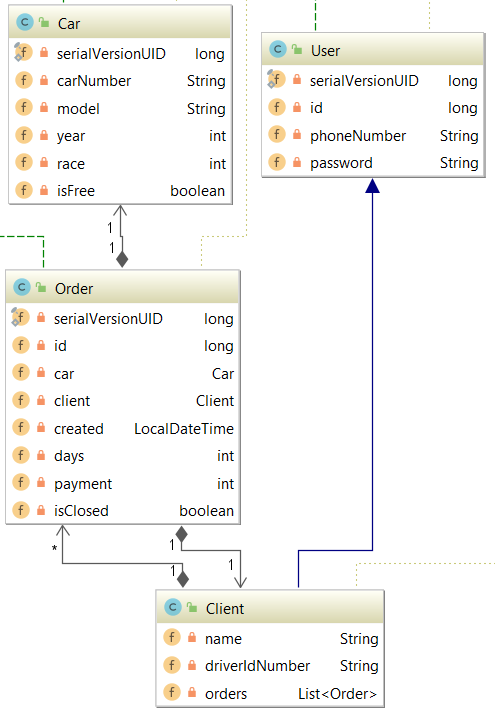


Рисунок 1.5 - Властивості класу Client та Order

Клас клієнт (Client) представляє собою інформаційну модель клієнта який наслідує всі властивості класу User та доповнений наступними властивостями:

* name - унікальний номер користувача в системі;
* **driverIdNumber**- номер водійського посвідчення;
* **orders -** представляє список всіх оренд які здійснив клієнт;

Клас замовлення (Orders) представляє собою інформаційну модель яка об'єднує об'єкти інших класів в собі а також володіє своїми унікальними властивостями:

* id- унікальний номер замовлення в системі;
* car - автомобіль який обраний для оренди
* **client**  - клієнт який бере авто в оренду
* **created -** дата створення замовлення;
* **days -** кількість днів оренди;
* **payment** - оплата за оренду
* **isClosed** - статус замовлення

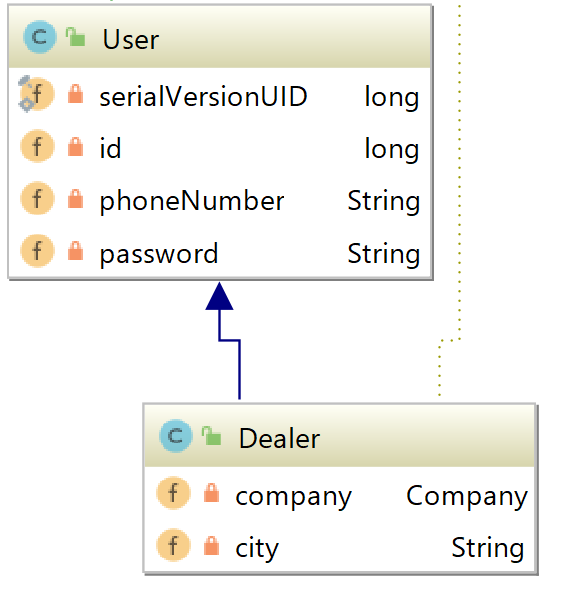


Рисунок 1.6 - Властивості класу Dealer

Клас дилер (Dealer) представляє собою інформаційну модель яка наслідує всі властивості класу User та доповнений наступними властивостями:

* company- компанія яку представляє дилер;
* **city -** місто в якому дилер надає свої послуги.

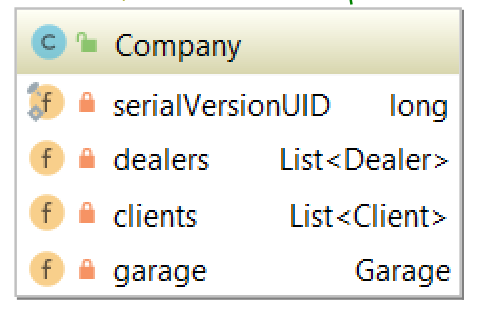


Рисунок 1.7 - Властивості класу Company

Клас дилер (Company) представляє собою інформаційну модель яка наслідує всі властивості класу User та доповнений наступними властивостями:

* company- компанія яку представляє дилер;
* **city -** місто в якому дилер надає свої послуги.

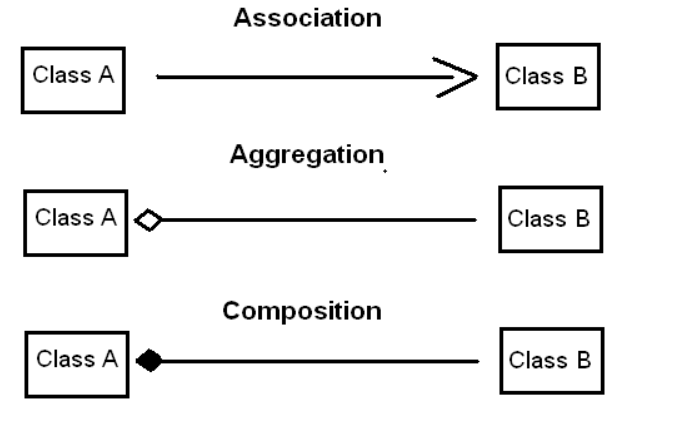


Рисунок 1.8 - Види ліній для відображення зв'язків між об'єктами

Всі спроектовані класи будуть використані при написанні програмного забезпечення та відображені у виконанні поставлених задач при аналізі предметної області. При проектуванні класів було використано всі можливості для створення правильних зв’язків між об’єктами які були виділені а саме використовуючи агерацію композицію та наслідування. Створена модель була описана та відображення засобами уніфікованої мови моделювання UML яка відображає зв'язки між об'єктами згідно рисунку 1.8

**2 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ЗАСОБАМИ МОВИ JAVA**

Розробка програмного забезпечення на основі раніше спроектованої предметної області буде розбита на декілька взаємопов'язаних етапів виконання а саме : розробка консольного меню користувача та моделі взаємодії; вирішення поставлених задач згідно ТЗ; реалізація можливості довготривалого збереження даних в аналогу бази даних.

Процес розробки консольного меню буде відображено тільки для одного із можливих варіантів меню, так яка модель взаємодії яка буде закладена буде поширюватись на всі інші меню.

Вирішення поставлених завдань буде організовано та розбито згідно пунктів консольного меню яке буде спроектовано першим.

Можливість збереження в базі даних буде реалізовано у вигляді серіалізації об'єктів засобами мови Java. Процес стерилізації передбачатиме підготовку об'єктів до збереження в файл, та їх зчитування при запуску програмного забезпечення.

Спроектоване програмне забезпечення буде підготовлено до автономної роботи без використання середовища розробки на інсталювання на будь-якій машині користувачів без використання додаткових попередньо встановлених пакетів та незалежно від операційної системи.

Розробками буде вестись засобами мови програмування Java 8 та середовищем інтегрованої розробки IntelliJ Idea. Процес упаковки програмного забезпечення в портативний формат також буде реалізовано засобами вище вказаного середовища розробки.

**2.1 Використання можливостей інтегрованого середовища розробки IntelliJ**

**IntelliJ IDEA** — комерційне інтегроване середовище розробки для різних мов програмування (Java, Python, Scala, PHP) від компанії JetBrains. Система поставляється у вигляді зрізаноїІ по функціональності безкоштовної версії «Community Edition» і повнофункціональної комерційної версії «Ultimate Edition», для якої активні розробники відкритих проектів мають можливість отримати безкоштовну ліцензію.

В даному проекті використовувались можливості повнофункціональної версії так як тільки вона дає змогу повноцінно проводити всі етапи проектування та написання програмного забезпечення.

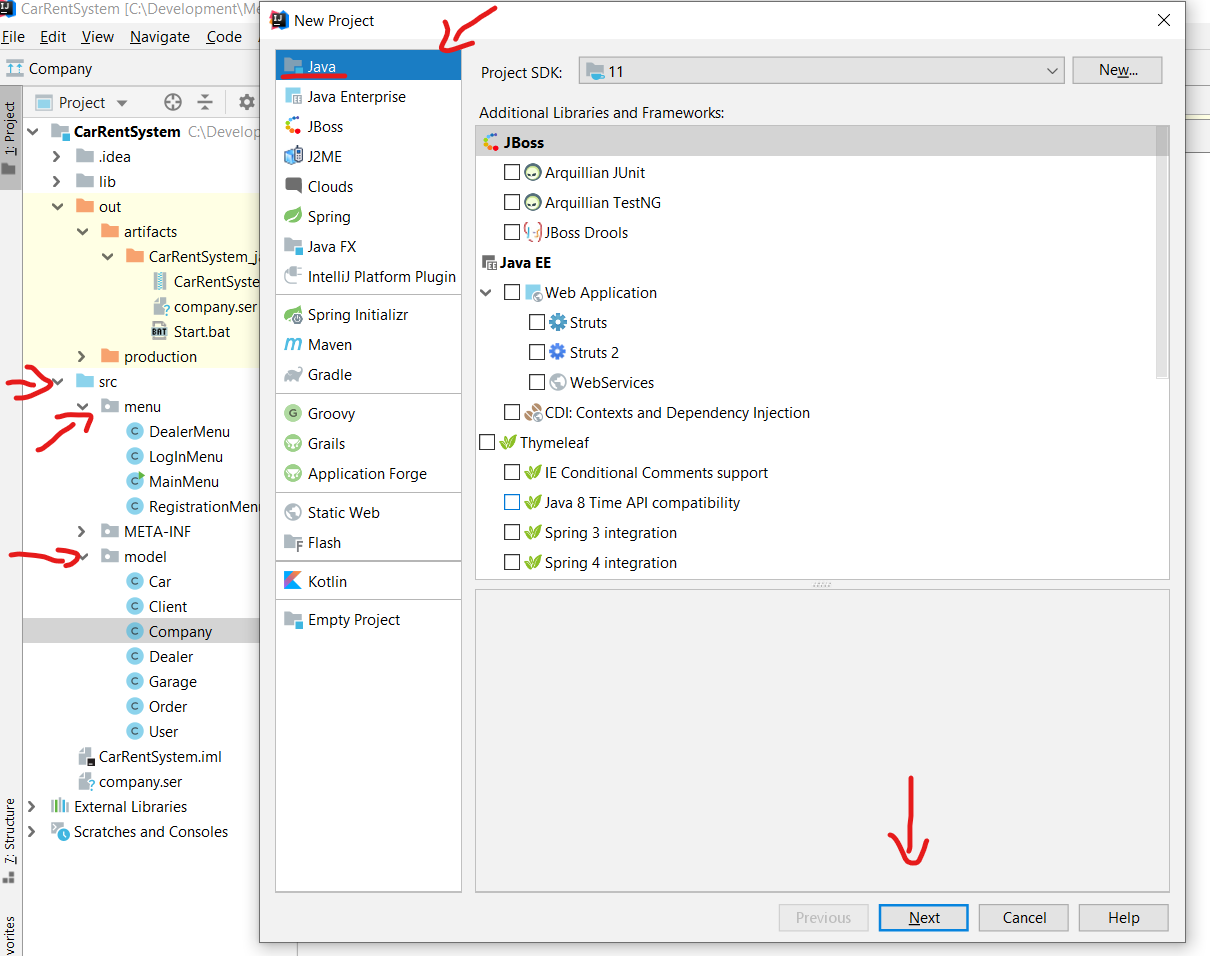


Рисунок 2.1 - Створення проекту та закладення його початкової структури

На рисунку 2.1 наведено приклад створення Java проекту в середовищі Intellij Idea використовуючи базову структуру проекту з основною папкою **src** та папкою з результатами компілювання проекту **out.**

Структурно проект розділений на окремі пакети, а саме пакет **model** в якому були створені всі основні класи згідно моделі, яка була спроектована на попередньому етапі. Всього в цьому пакеті наявні **7 класів**, які строго відповідають вимогам **ТЗ**. В пакеті **menu** розміщені додаткові класи, на основі яких і реалізовано взаємодію користувача з програмою, тобто консольне меню. Приклад взаємодії з консольним меню буде наведено нижче.

В папці out як можна побачити на рисунку 2.1 після компіляції проекту появився файл з розширення jar який і представляє готове програмне забезпечення яке можна запускати на будь-якій платформі. Також додатково до нього було створено файл запуску на операційній системі windows.

В папці lib знаходиться бібліотека Lombok завдяки якій код вдалося мінімізувати використовуючи такий інструмент мови програмування Java як анотації. Приклад різниці в коді наведено нижче на рисунку 2.2.

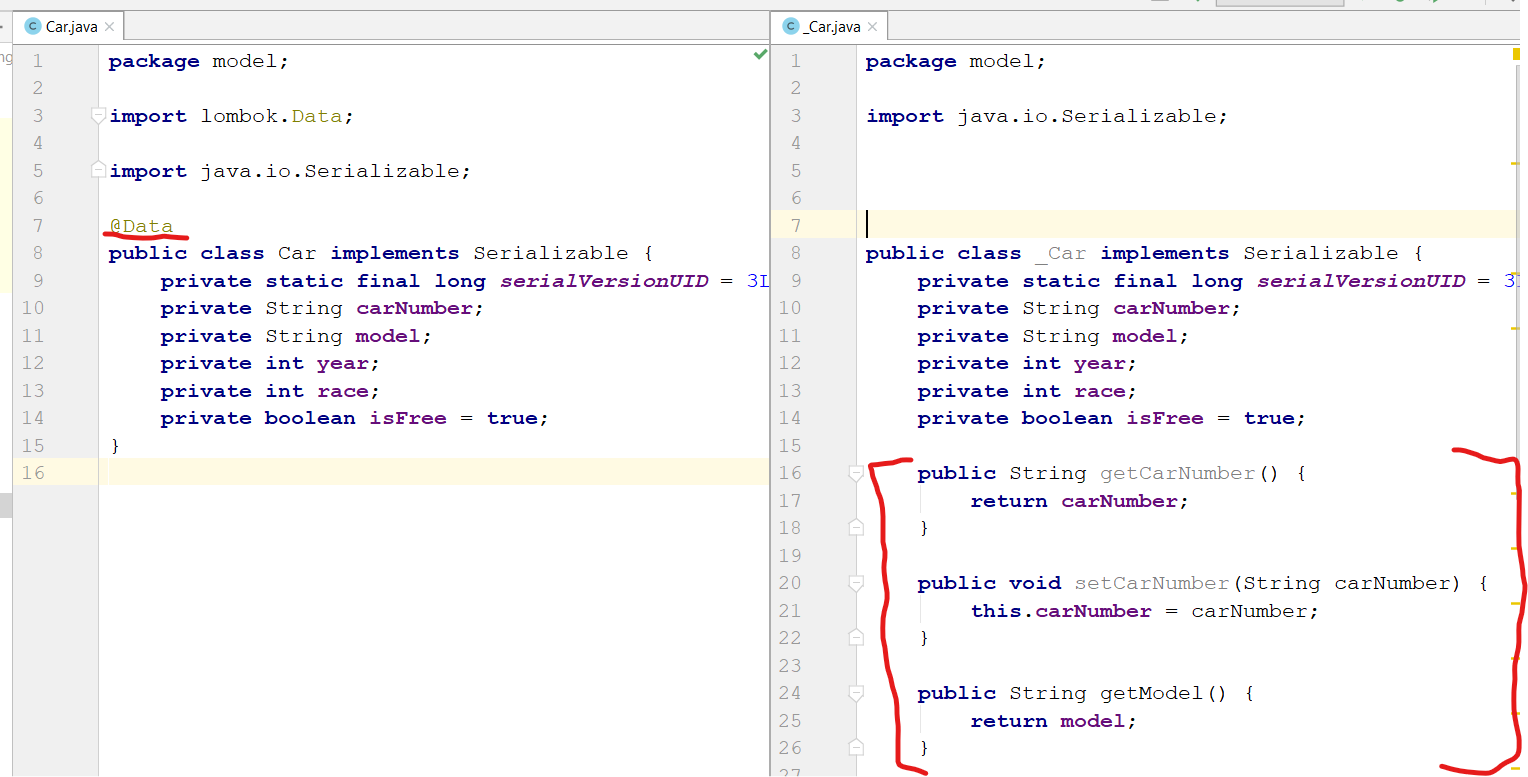


Рисунок 2.2 - Різниці в коді при використанні атоцій та без

**2.2 Розробка консольного меню взаємодії з користувачем**

В даній роботі було прийнято рішення створити консольне меню, так як воно дає можливість більше зосередиться на реалізації всього потрібного функціоналу не витрачаючи багато часу на створення віконних режимів взаємодії з користувачем. Проте спроектована система може бути адаптована під будь-який графічний інтерфейс а також реалізована у вигляді веб-сайту.

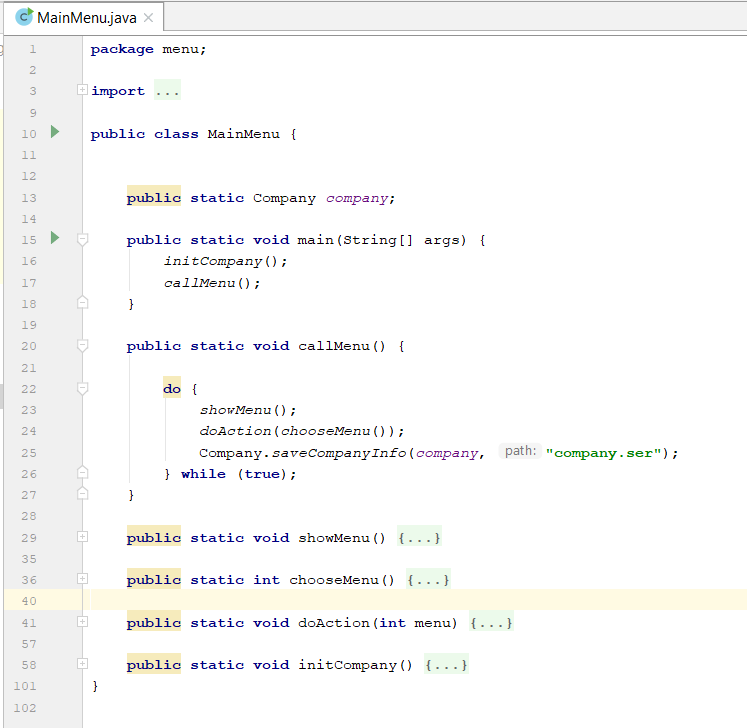


Рисунок 2.3 - Приклад структури коду для консольного меню

Як можна побачити на рисунку 2.3 структурно меню було розділено на окремі функціональні блоки, що дало змогу легко адаптовувати його під вимоти ТЗ.

Основною функцією є main функція в рядку 15 яка запускає програмне забезпечення і одночасно слугує точкою відліку роботи програми. В ній відбувається виклик двох інших функцій (рядок 16, 17) а саме функції початкової ініціалізації роботи програми та функції виклику головного меню відповідно.

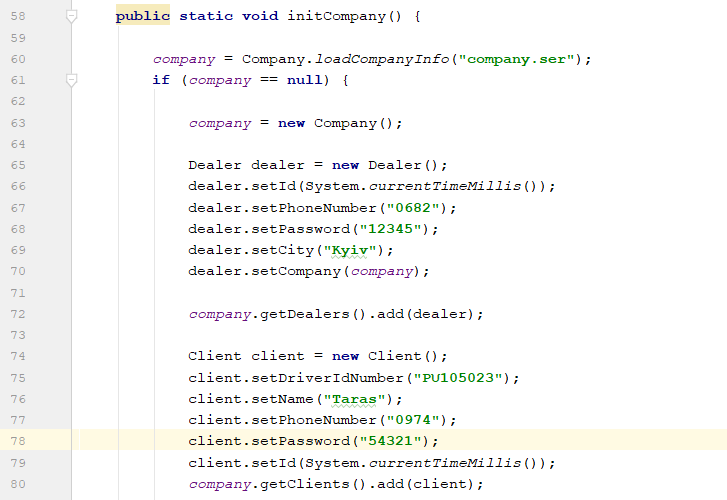


Рисунок 2.4 - Фрагмент функції початкової ініціалізації роботи програми

На рисунку 2.4 наведено частину коду, яка у випадку відсутності початкових даних в базі даних буде створювати перші об’єкти класів для тестового режиму роботи програми.

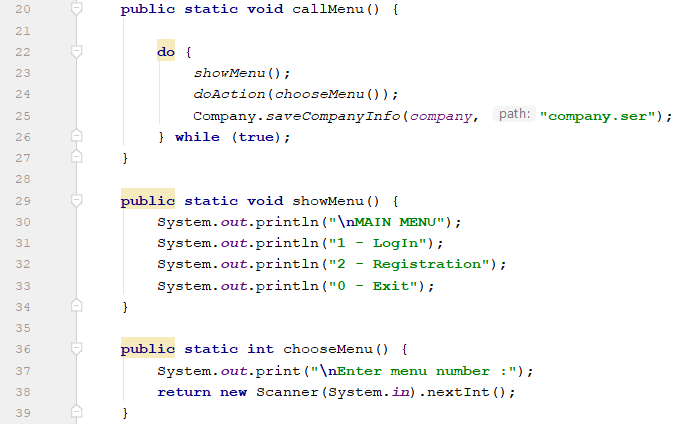


Рисунок 2.5 - Фрагмент коду функцій консольного меню

На рисунку 2.5 наведено основні функції які запускаються для відображення меню в консолі для користувача. Тут можна виділити три основні функція які і формують модель роботи будь-якого консольного меню, а саме showMenu() doAction() chooseMenu(). Саме ця модель буде використана для створення інших сторінок консольного меню.

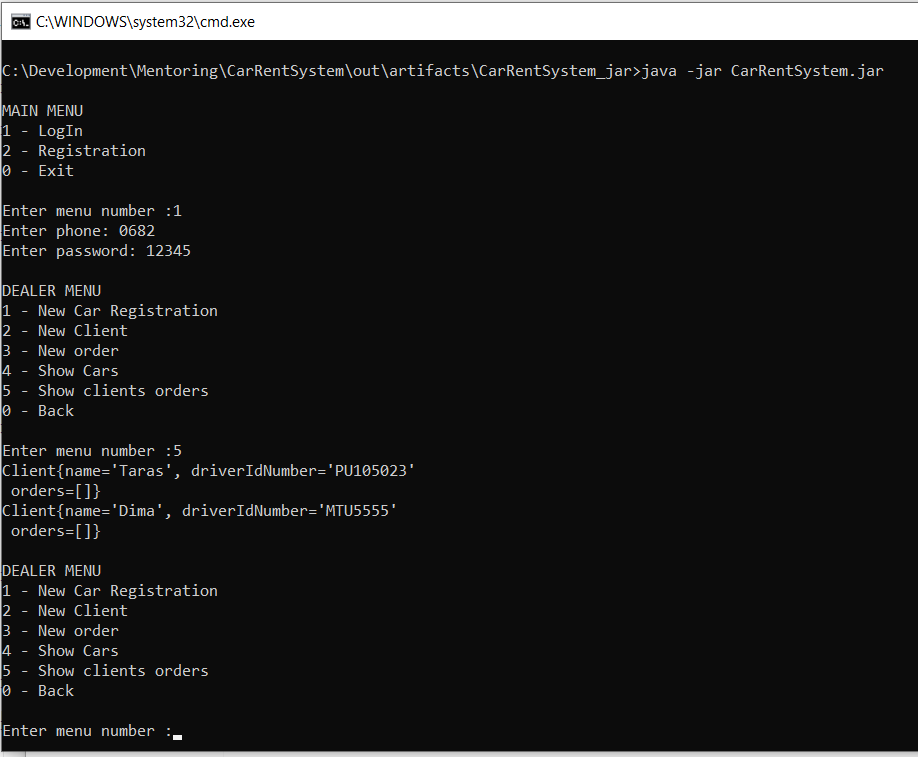


Рисунок 2.6 - Приклад запуску консольного меню програми

На рисунку 2.6 можна побачити приклад роботи уже готово програмного забезпечення. Спочатку дилер авторизуется в системі, а далі якщо всі дані були введені правильно йому відкривається меню я кому реалізовані всі функціональні можливості згідно ТЗ.

Також в користувача є можливість провести ручну реєстрацію в системі що спрощує використання клієнтами. Приклад реєстрації нових користувачів можна побачити на рисунку 2.7 ниже.

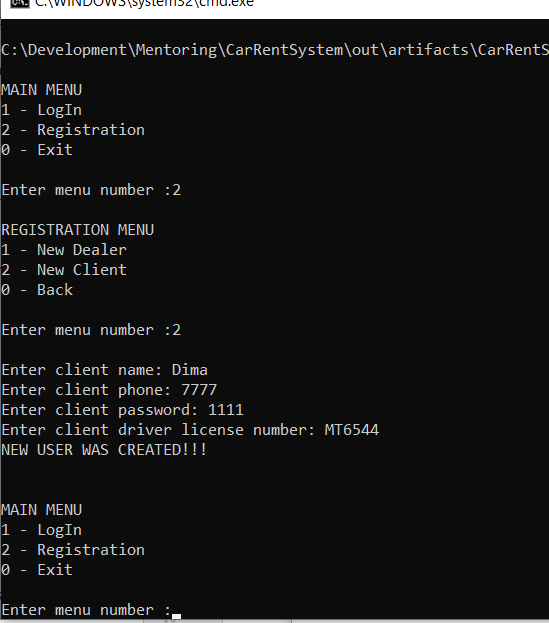


Рисунок 2.7 - Режим реєстрації нового користувача в системі

Як можна побачити при виборі цього пункту меню користувачу буде дано вибір кого зареєструвати, на рисунку 2.7 наведено приклад реєстрації нового клієнта в системі.

По даній моделі консольного меню працюють всі інші пункти меню. Тому немає необхідності в описі всіх пунктів краще виконати тестовий режим запуску програми для огляду реалізації функціональних можливостей згідно ТЗ.

**ВИСНОВКИ**

В результаті виконання даної курсової роботи була створена програма, із використанням консольного інтерфейсу, для предметної області – система управління орендуванням автомобілів. Проаналізувавши предметну область було виділено такі класи : автомобіль, клієнт, компанія, дилер, замовлення, користувач системи, а також гараж. Між даними класами був встановлений взаємозв’язок у вигляді агрегації композиції або наслідування, що в результаті дало об’єктну модель системи.

Інтерфейс програми було реалізовано використовуючи консольне вікно. В основі меню використано текстові позначки пунктів, та відповідна прив’язка до реалізованих функціональних можливостей. Всі завдання згідно ТЗ було реалізовано а також створений аналог бази даних який дає можливість зберегти дані на довгий термін використання.

Дану програму можна вдосконалити перенесенням інтерфейсу у звичний для користувачів віконний вигляд, з додавання зручних елементів управління, або розвинути в сторону веб-сервісу. Також можна буде в майбутньому реалізувати підтримку бази даних такої як MySQL

**ПЕРЕЛІК ПОСИЛАНЬ НА ДЖЕРЕЛА**

1. Ю. Бєлов. Вступ до програмування мовою Java. Організація обчислень Ж навч. посібник. / Ю. А. Бєлов, Ю. В. Коваль. — К.: Видавничо-поліграфічний центр «Київський університет», 2012. — 175 с.

2. С. Липпман, Ж. Лажойе, Б. Му. Язык программирования Java. Вводный курс. - М.:«Вильямс», 2007.

3. Вікіпедія Вільна енциклопедія [Електронний ресурс] : [Веб-сайт] - Режим доступу: uk.wikipedia.org (дата звернення 20.09.2015)

4. ДСТУ 7.1:2006 Бібліографічний запис. Бібліографічний опис. Загальні вимоги та правила складання.

**ДОДАТКИ**

@Data

**public class** Car **implements** Serializable {

**private static final long *serialVersionUID*** = 3L;

**private** String **carNumber**;

**private** String **model**;

**private int year**;

**private int race**;

**private boolean isFree** = **true**;

}

@Data

**public class** Client **extends** User {

**private** String **name**;

**private** String **driverIdNumber**;

**private** List<Order> **orders** = **new** ArrayList<>();

**public** String getClientInfo(){

**return "Client{"** +

**"name='"** + **name** + **'\''** +

**", driverIdNumber='"** + **driverIdNumber** + **'}'**;

}

@Override

**public** String toString() {

**return "Client{"** +

**"name='"** + **name** + **'\''** +

**", driverIdNumber='"** + **driverIdNumber** + **'\''** +

**"\n orders="** + **orders** +

**'}'**;

}

}

**package** model;

**import** lombok.Data;

**import** menu.MainMenu;

**import** java.io.\*;

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.Scanner;

@Data

**public class** Company **implements** Serializable {

**private static final long *serialVersionUID*** = 1L;

**private** List<Dealer> **dealers** = **new** ArrayList<>();

**private** List<Client> **clients** = **new** ArrayList<>();

**private** Garage **garage** = **new** Garage();

**public void** addNewClient() {

Scanner scanner = **new** Scanner(System.***in***);

Client client = **new** Client();

client.setId(System.*currentTimeMillis*());

System.***out***.print(**"\nEnter client name: "**);

client.setName(scanner.next());

System.***out***.print(**"Enter client phone: "**);

client.setPhoneNumber(scanner.next());

System.***out***.print(**"Enter client password: "**);

client.setPassword(scanner.next());

System.***out***.print(**"Enter client driver license number: "**);

client.setDriverIdNumber(scanner.next());

**clients**.add(client);

System.***out***.println(**"NEW USER WAS CREATED!!!\n"**);

MainMenu.*callMenu*();

}

**public void** addNewDealer() {

Scanner scanner = **new** Scanner(System.***in***);

Dealer dealer = **new** Dealer();

dealer.setId(System.*currentTimeMillis*());

System.***out***.print(**"\nEnter dealer phone: "**);

dealer.setPhoneNumber(scanner.next());

System.***out***.print(**"Enter dealer password: "**);

dealer.setPassword(scanner.next());

System.out.print(**"Enter dealer city: "**);

dealer.setCity(scanner.next());

dealer.setCompany(**this**);

dealers.add(dealer);

System.out.println(**"NEW DEALER WAS CREATED!!!\n"**);

MainMenu.callMenu();

}

**public static** Company loadCompanyInfo(String path) {

Company company = **null**;

**try** {

FileInputStream file = **new** FileInputStream(path);

ObjectInputStream in = **new** ObjectInputStream(file);

company = (Company) in.readObject();

in.close();

file.close();

} **catch** (IOException ex) {

System.out.println(**"IOException is caught"**);

} **catch** (ClassNotFoundException ex) {

System.out.println(**"ClassNotFoundException is caught"**);

}

**return** company;

}

**public static void** saveCompanyInfo(Company company, String path) {

**try** {

FileOutputStream file = **new** FileOutputStream(path);

ObjectOutputStream out = **new** ObjectOutputStream(file);

out.writeObject(company);

out.close();

file.close();

} **catch** (IOException ex) {

System.out.println(**"IOException is caught"**);

}

}

}

**package** model;

**import** lombok.Data;

**import** java.time.LocalDateTime;

**import** java.util.Scanner;

@Data

**public class** Dealer **extends** User {

**private** Company **company**;

**private** String **city**;

**public void** carRegistration() {

Scanner scanner = **new** Scanner(System.***in***);

Car car = **new** Car();

System.***out***.print(**"\nEnter car model:"**);

car.setModel(scanner.nextLine());

System.***out***.print(**"Enter car registration number:"**);

car.setCarNumber(scanner.next());

System.***out***.print(**"Enter car year:"**);

car.setYear(scanner.nextInt());

System.***out***.print(**"Enter car race:"**);

car.setRace(scanner.nextInt());

**company**.getGarage().addCar(car);

}

**public void** createOrder() {

Scanner scanner = **new** Scanner(System.***in***);

Order order = **new** Order();

Client currentClient = getClient();

order.setClient(currentClient);

System.***out***.print(**"Enter car model:"**);

order.setId(System.*currentTimeMillis*());

order.setCar(**company**.getGarage().getCar());

order.setCreated(LocalDateTime.*now*());

System.***out***.print(**"Enter days amount:"**);

order.setDays(scanner.nextInt());

currentClient.getOrders().add(order);

}

**public void** showAllOrders() {

**for** (**int** i = 0; i < **company**.getClients().size(); i++) {

System.***out***.println(**company**.getClients().get(i).toString());

}

}

**public void** showClients() {

**for** (**int** i = 0; i < **company**.getClients().size(); i++) {

System.***out***.println(**"#"** + i + **" "** + **company**.getClients().get(i).getClientInfo());

}

}

**public** Client getClient() {

Scanner scanner = **new** Scanner(System.***in***);

showClients();

System.***out***.print(**"Choose client by number:"**);

**int** client = scanner.nextInt();

**return company**.getClients().get(client);

}

}

**package** model;

**import** lombok.Data;

**import** java.io.Serializable;

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.Scanner;

@Data

**public class** Garage **implements** Serializable {

**private static final long *serialVersionUID*** = 4L;

**private** List<Car> **cars** = **new** ArrayList<>();

**public void** addCar(Car car) {

**cars**.add(car);

}

**public void** showAllCars() {

System.***out***.println(**"\nALL CARS"**);

**for** (**int** i = 0; i < **cars**.size(); i++) {

System.***out***.print(**"#"** + i + **" "**);

System.***out***.println(**cars**.get(i).toString());

}

}

**public void** showFreeCars() {

System.***out***.println(**"\nFREE CARS"**);

**for** (**int** i = 0; i < **cars**.size(); i++) {

**if** (**cars**.get(i).isFree()) {

System.***out***.print(**"#"** + i + **" "**);

System.***out***.println(**cars**.get(i).toString());

}

}

}

**public** Car getCar() {

Scanner scanner = **new** Scanner(System.***in***);

showFreeCars();

System.***out***.print(**"Choose car by number:"**);

**int** carNumber = scanner.nextInt();

Car current = **cars**.get(carNumber);

current.setFree(**false**);

**return** current;

}

}

**package** model;

**import** lombok.Data;

**import** java.io.Serializable;

**import** java.time.LocalDateTime;

@Data

**public class** Order **implements** Serializable {

**private static final long *serialVersionUID*** = 5L;

**private long id**;

**private** Car **car**;

**private** Client **client**;

**private** LocalDateTime **created**;

**private int days**;

**private int payment**;

**private boolean isClosed**;

@Override

**public** String toString() {

**return "\nOrder{"** +

**"id="** + **id** +

**", car="** + **car** +

**", created="** + **created** +

**", days="** + **days** +

**", isClosed="** + **isClosed** +

**'}'**;

}

}

**package** model;

**import** lombok.Data;

**import** java.io.Serializable;

@Data

**public class** User **implements** Serializable {

**private static final long *serialVersionUID*** = 2L;

**private long id**;

**private** String **phoneNumber**;

**private** String **password**;

}

**package** menu;

**import** model.Car;

**import** model.Client;

**import** model.Company;

**import** model.Dealer;

**import** java.util.Scanner;

**public class** MainMenu {

**public static** Company *company*;

**public static void** main(String[] args) {

*initCompany*();

*callMenu*();

}

**public static void** callMenu() {

**do** {

*showMenu*();

*doAction*(*chooseMenu*());

Company.*saveCompanyInfo*(*company*, **"company.ser"**);

} **while** (**true**);

}

**public static void** showMenu() {

System.***out***.println(**"\nMAIN MENU"**);

System.***out***.println(**"1 - LogIn"**);

System.***out***.println(**"2 - Registration"**);

System.***out***.println(**"0 - Exit"**);

}

**public static int** chooseMenu() {

System.***out***.print(**"\nEnter menu number :"**);

**return new** Scanner(System.***in***).nextInt();

}

**public static void** doAction(**int** menu) {

**switch** (menu) {

**case** 1:

LogInMenu.*callMenu*();

**break**;

**case** 2:

RegistrationMenu.*callMenu*();

**break**;

**case** 0:

Company.*saveCompanyInfo*(*company*, **"company.ser"**);

System.*exit*(0);

**default**:

System.***out***.println(**"WRONG MENU NUMBER"**);

}

}

**public static void** initCompany() {

*company* = Company.*loadCompanyInfo*(**"company.ser"**);

**if** (*company* == **null**) {

*company* = **new** Company();

Dealer dealer = **new** Dealer();

dealer.setId(System.*currentTimeMillis*());

dealer.setPhoneNumber(**"0682"**);

dealer.setPassword(**"12345"**);

dealer.setCity(**"Kyiv"**);

dealer.setCompany(*company*);

*company*.getDealers().add(dealer);

Client client = **new** Client();

client.setDriverIdNumber(**"PU105023"**);

client.setName(**"Taras"**);

client.setPhoneNumber(**"0974"**);

client.setPassword(**"54321"**);

client.setId(System.*currentTimeMillis*());

*company*.getClients().add(client);

Car car1 = **new** Car();

car1.setModel(**"Tesla model X"**);

car1.setCarNumber(**"ВХ77777"**);

car1.setYear(2016);

car1.setRace(15000);

car1.setFree(**true**);

*company*.getGarage().addCar(car1);

Car car2 = **new** Car();

car2.setModel(**"BMW X6"**);

car2.setCarNumber(**"ВХ666"**);

car2.setYear(2017);

car2.setRace(1500);

car2.setFree(**true**);

*company*.getGarage().addCar(car2);

}

}

}

**package** menu;

**import** model.Dealer;

**import** java.util.Scanner;

**public class** LogInMenu {

**public static void** callMenu() {

Scanner scanner = **new** Scanner(System.***in***);

System.***out***.print(**"Enter phone: "**);

String phone = scanner.next();

System.***out***.print(**"Enter password: "**);

String password = scanner.next();

**for** (**int** i = 0; i < MainMenu.*company*.getDealers().size(); i++) {

Dealer dealer = MainMenu.*company*.getDealers().get(i);

**if** (dealer.getPhoneNumber().equals(phone)) {

**if** (dealer.getPassword().equals(password)) {

DealerMenu.*callMenu*(dealer);

}

}

}

System.***out***.println(**"WRONG LOGIN OR PASSWORD"**);

}

}

**package** menu;

**import** model.Dealer;

**import** java.util.Scanner;

**public class** DealerMenu {

**private static** Dealer *currentDealer*;

**public static void** callMenu(Dealer dealer) {

*currentDealer* = dealer;

**do** {

*showMenu*();

*doAction*(*chooseMenu*());

} **while** (**true**);

}

**public static void** showMenu() {

System.***out***.println(**"\nDEALER MENU"**);

System.***out***.println(**"1 - New Car Registration"**);

System.***out***.println(**"2 - New Client"**);

System.***out***.println(**"3 - New order"**);

System.***out***.println(**"4 - Show Cars"**);

System.***out***.println(**"5 - Show clients orders"**);

System.***out***.println(**"0 - Back"**);

}

**public static int** chooseMenu() {

System.***out***.print(**"\nEnter menu number :"**);

**return new** Scanner(System.***in***).nextInt();

}

**public static void** doAction(**int** menu) {

**switch** (menu) {

**case** 1:

*currentDealer*.carRegistration();

**break**;

**case** 2:

*currentDealer*.getCompany().addNewClient();

**break**;

**case** 3:

currentDealer.createOrder();

**break**;

**case** 4:

MainMenu.company.getGarage().showAllCars();

**break**;

**case** 5:

currentDealer.showAllOrders();

**break**;

**case** 0:

MainMenu.callMenu();

**default**:

System.out.println(**"WRONG MENU NUMBER"**);

}

}

}

**package** menu;

**import** java.util.Scanner;

**public class** RegistrationMenu {

**public static void** callMenu() {

*showMenu*();

**do** {

*doAction*(*chooseMenu*());

} **while** (**true**);

}

**public static void** showMenu() {

System.***out***.println(**"\nREGISTRATION MENU"**);

System.***out***.println(**"1 - New Dealer"**);

System.***out***.println(**"2 - New Client"**);

System.***out***.println(**"0 - Back"**);

}

**public static int** chooseMenu() {

System.***out***.print(**"\nEnter menu number :"**);

**return new** Scanner(System.***in***).nextInt();

}

**public static void** doAction(**int** menu) {

**switch** (menu) {

**case** 1:

MainMenu.*company*.addNewDealer();

**break**;

**case** 2:

MainMenu.*company*.addNewClient();

**break**;

**case** 0:

MainMenu.*callMenu*();

**default**:

System.***out***.println(**"WRONG MENU NUMBER"**);

}

}

}