ТЕХНИЧЕСКИ УНИВЕРСИТЕТ – ВАРНА

КУРСОВ ПРОЕКТ

по

Обектно-ориентирано програмиране –

1-ва част

Изготвил: Проверил: ……………….

Михаил Георгиев ас. Павлина Линова

Специалност: СИТ

Факултетен номер: 19621820

Курс: Втори

Група: I a

1. Условие на задачата

**Имоти**

Да се дефинира абстрактен клас имот Estate​.

Всеки се дефинира чрез:

● Адрес на имота;

● Собственик;

● Цена;

● Площ;

Да се реализира функционалност, която извежда информацията за даден имот в конзолата.

Всеки имот може да бъде един от следните типове:

● Апартамент (**Flat**​) - съдържа информация за брой стаи и етаж.

● Къща (**House**​)- съдържа информация за брой стаи, брой етажи, площ на двора.

● Парцел (**Plot**) ​- съдържа информация за налични комуникации (списък от елементи от изброим тип Communication (вода, ток, телефон, близост до път, канализация) и площ.

За да съществува един имот той трябва да бъде регистриран.

Да се дефинира клас, който поддържа списък на регистрираните недвижими имоти **RealEstates**​, съхраняващ контейнер от указатели към обекти клас **Estate**​, в който ще се съхраняват обекти от тип имот.

Ние можем, както да регистрираме, така и да премахваме имот от списъка с регистрирани имоти.

За да можем да намерим определен имот се налага да използваме брокери. Всеки такъв разполага със списъка от регистрираните имоти, но използва различно оценяване на имот при търсене.

Всички брокери **завишават цените на имотите, за да включат техния хонорар.**

Всеки брокер се дефинира чрез:

● Име

● Процент

● Тип

Различаваме три типа брокера:

○ **HelpfulBroker / ThoughfulBroker - притежава списък с ВИП обяви**​, които се визуализират преди останалите обяви и подобрява шансът даден имот да бъде представен.

○ **DishonestBroker** - ​притежава списък с ВИП обяви, **но ги представя с подобрени цена и площ на случаен принцип​.**

○ **SimpleBroker**​ - не се държи по-различно от обикновена търсачка.

Да се реализира функционалност, която позволява добавяне и премахване на имот към списъка с ВИП имоти в брокерите от първите два типа.

За да преглеждаме имоти, ще ни се наложи да си изберем брокер, който да ни обслужи. Добре, а как търсим?

След като си изберем брокер, можем да подадем нашите критерии (задаването критерии, може да става чрез меню или по друг подходящ и избран от вас начин) за търсене и той да ни представи списък от предложения.

За тази цел програмата трябва да поддържа следните функционалности:

● Търсене по различни изисквания на клиента - вид на имота, цена от зададен интервал, площ, налични комуникации, дадено населено място.

● Извеждане на всички имоти от зададен вид, сортирани по цена в низходящ ред.

● Извеждането на имотите трябва да става в табличен вид.

Да се демонстрира работата на класа **RealEstates​**, като се дефинира обект от този клас и списък от брокери и се предостави възможност за различни справки, които демонстрират методите за търсене.

1. Тестови данни

void UI::loadData() {

House\* house = new House("Varna, Center", "Pavel", 50000, 200.00, 1, 3, 300.00);

estates->registerEstate(house);

House\* house1 = new House("Varna, Troshevo", "Pavel", 500000, 150.50, 10, 2, 500.50);

estates->registerEstate(house1);

House\* house2 = new House("Varna, Levski", "Hristiyan", 110000, 130, 4, 8, 300);

estates->registerEstate(house2);

House\* house3 = new House("Varna, Levski", "Kameliya", 80000, 90, 3, 1, 50);

estates->registerEstate(house3);

Flat\* flat = new Flat("Varna, Vladislavovo", "Kameliya", 45000, 90, 5, 2);

estates->registerEstate(flat);

Flat\* flat1 = new Flat("Varna, Center", "Pavel", 90000, 75.4, 3, 2);

estates->registerEstate(flat1);

Flat\* flat2 = new Flat("Varna, Levski", "Hristiyan", 90000, 130, 4, 5);

estates->registerEstate(flat2);

Flat\* flat3 = new Flat("Varna, Briz", "Hristiyan", 70000, 70, 2, 4);

HonestBroker\* honestBroker = new HonestBroker("Mihail", 0.4, estates);

honestBroker->addVIP(house);

honestBroker->addVIP(flat);

honestBroker->addVIP(house2);

honestBroker->addVIP(house3);

DishonestBroker\* dishonestBroker = new DishonestBroker("Ivan", 0.2, estates);

dishonestBroker->addVIP(house2);

dishonestBroker->addVIP(flat2);

SimpleBroker\* simpleBroker = new SimpleBroker("Maria", 0.3, estates);

brokers.push\_back(honestBroker);

brokers.push\_back(dishonestBroker);

brokers.push\_back(simpleBroker);

}

III. Функционалност на класове

Дефинирани са 11 класа:

* Estate
* Flat
* House
* Plot
* RealEstate
* Broker
* VIPBroker
* HonestBroker
* DishonestBroker
* SimpleBroker
* UI

Класът **Estate** е абстрактен клас. Съдържа базова информация за всеки имот (адрес на имота, собственик, цена, площ).

Класът **Flat** е за имот тип апартамент. Съдържа информация за брой стай и етаж.

Класът **House** е за имот тип къща. Съдържа информация за брой стаи, брой етажи, площ на двора.

Класът **Plot** е за имот тип парцел. Съдържа информация за налични комуникации (списък от елемнти от изброим тип Communication (вода, топ, телефон, близост до път, канализация)) и площ.

Класът **RealEstate** поддържа списък на регистрираните недвижими имоти. Съхранява контейнер от указатели към обекти клас Estate, в който ще се съхраняват обекти от тип имот.

Класът **Broker** е абстрактен за всеки тип брокер. Съдържа информация за име на брокер, процент и тип.

Класът **VIPBroker** е абстрактен. Поддържа списък за всички ВИП обяви на типа брокери Honest и Dishonest.

Класът **HonestBroker** съдържа функция, която извежда намерените обяви при търсене.

Класът **DishonestBroker**  съдържа функция, която извежда намерените обяви и променя вип обявите с подобрени цена и площ на случаен принцип.

Класът **SimpleBroker** съдържа функции за търсене и извеждане на всички обяви.

Класът **UI** съдържа функции с меню.

IV. Използвани функции

4.1. В класът **Estate** са използвани следните функции:

Estate(); -> подразбиращ се конструктор; инициализира празен обект

Estate(string Address, string Owner, int Price, double Size); -> експлицитен контруктор ; задава стойностите, подадени при създаването на обекта

Estate(const Estate& other); -> копиращ конструктор; копира стойностите на подаден обект при създавенето на нов обект

const string& getAddress() const; -> връща като резултат адрес

const int& getPrice() const; -> връща като резултат цена

const double& getSize() const; -> връща като резултат площ

const string& getOwner() const; -> връща като резултат собственик

void setPrice(const int& Price); -> презаписва цената

void setAddress(const string& Address); -> презаписва адреса

void setOwner(const string& Owner); -> презаписва собственика

void setSize(const double& Size); -> презаписва площта

bool operator<(Estate& e); -> оператор за сравнение; сравнява цената на имота

friend ostream& operator<<(ostream& out, Estate& e); -> предефинира глобалния оператор за потоков изход

virtual void printEstate(ostream& out) = 0; -> чисто виртуална функция; извежда информацията за всеки вид имот

virtual const int& getRoom() const; -> виртуална функция; връща като резултат броя на стаите

virtual const int& getFloor() const; -> виртуална функция; връща като резултат етаж

virtual const double& getYardSize() const; -> виртуална функция; връща като резултат площта на градината

virtual const list<string>& getComunications() const; -> виртуална функция; връща като резултат списък от видове комуникации на парцелите

4.2. В класът **Flat** са използвани следните функции:

Flat(string Address, string Owner, int Price, double Size, int Room, int Floor); -> експлицитен конструктор; задава стойностите, подадени при създаването на обекта

void printEstate(ostream& out); -> извежда информацията за апартамента

void setRoom(const int& Room); -> презаписва броят на стаите в апартамента

void setFloor(const int& Floor); -> презаписва етажа на апартамента

const int& getRoom() const; -> връща като резултат броя на стаите в апартамента

const int& getFloor() const; -> връща като резултат етажа на апартамента

friend ostream& operator<<(ostream& out, Flat& f); -> предефенира глобалния оператор за потоков изход

4.3. В класът **House** са използвани следните функции:

House(string Address, string Owner, int Price, double Size, int Room, int Floor, double YardSize); -> експлицитен конструктор; задава стойностите, подадени при създаването на обекта

void printEstate(ostream& out); -> извежда информацията за къщата

void setYardSize(const double& YardSize); -> презаписва площта на двора на къщата

const double& getYardSize() const; -> връща като резултат площта на двора на къщата

friend ostream& operator<<(ostream& out, House& h); -> предефинира глобалния оператор за потоков изход

4.4. В класът **Plot** са използвани следните функции:

Plot(string Address, string Owner, int Price, double Size, list<string> comunications, double YardSize); -> експлицитен конструктор; задава стойностите, подадени при създаването на обекта

void printEstate(ostream& out); -> извежда информацията за парцела

void setYardSize(const double& YardSize); -> презаписва площта на парцела

const double& getYardSize() const; -> връща като резултат площта на парцела

void setComunications(const list<string> comunications); -> презаписва комуникациите на парцела

const list<string>& getComunications() const; -> връща като резултат списък, съдържащ комуникациите на парцела

4.5. В класът **RealEstate** са използвани следните функции:

void registerEstate(Estate\* estate); -> добавя нов имот към списъка с имоти

void deleteEstate(Estate\* estate); -> изтрива имот от списъка с имоти

list<Estate\*> getEstates(); -> връща като резултат списък с всички имоти

void printEstates(); -> извежда информация за всички имоти

4.6. В класът **Broker** са използвани следните функции:

Broker(); -> подразбиращ се конструктор; инициализира празен обект

Broker(const string& Name, const double& Percent, RealEstate\* Estates); -> експлицитен конструктор; задава стойностите, подадени при създаването на обекта

void setEstates(RealEstate\* Estates); -> презаписва имотите, с които разполага брокера

RealEstate\* getEstates(); -> връща като резултат имотите, с които разполага брокера

void setName(const string& Name); -> презаписва името на брокера

const string& getName() const; -> връща като резултат името на брокера

void setPercent(const double& Percent); -> презаписва процента печалба на брокера

const double& getPercent() const; -> връща като резултат процент печалба на брокера

virtual void addVIP(Estate\* estate) = 0; -> чисто виртуална функция; добавя имот към списъка с ВИП обяви

virtual void deleteVIP(Estate\* estate) = 0; -> чисто виртуална функция; изтрива имот от списъка с ВИП обяви

virtual void search(string address, string owner, int minPrice, int maxPrice, double size, int room, int floor, double yardSize, list<string> comunications, string estateType) = 0; -> чисто виртуална функция; търси имот според изискванията на клиента

virtual void printFoundEstates(list<Estate\*> found, list<Estate\*>\* vip = nullptr) = 0; -> извежда всички намерени имоти

4.7. В класът **VIPBroker** са използвани следните функции:

VIPBroker(); -> подразбиращ се конструктор; инициализира празен обект

VIPBroker(const string& Name, const double& Percent, RealEstate\* Estates, RealEstate\* vipEstates); -> експлицитен конструктор; задава стойностите, подадени при създаването на обекта

VIPBroker(const string& Name, const double& Percent, RealEstate\* Estates); -> експлицитен конструктор; задава стойностите, подадени при създаването на обекта

void addVIP(Estate\* estate); -> добавя имот към списъка с ВИП обяви

void deleteVIP(Estate\* estate); -> изтрива имот от списъка с ВИП обяви

void search(string address, string owner, int minPrice, int maxPrice, double size, int room, int floor, double yardSize, list<string> comunications, string estateType); -> търси имот според изискванията на клиента

4.8. В класът **HonestBroker** са използвани следните функции:

HonestBroker(); -> подразбиращ се конструктор; инициализира празен обект

HonestBroker(const string& Name, const double& Percent, RealEstate\* Estates, RealEstate\* vipEstates); -> експлицитен конструктор; задава стойностите, подадени при създаването на обекта

HonestBroker(const string& Name, const double& Percent, RealEstate\* Estates); -> експлицитен конструктор; задава стойностите, подадени при създаването на обекта

void printFoundEstates(list<Estate\*> found, list<Estate\*>\* vip = nullptr); -> извежда всички намерени имоти

4.9. В класът **DishonestBroker** са използвани следните функции:

DishonestBroker(); -> подразбиращ се конструктор; инициализира празен обект

DishonestBroker(const string& Name, const double& Percent, RealEstate\* Estates, RealEstate\* vipEstates); -> експлицитен конструктор; задава стойностите, подадени при създаването на обекта

DishonestBroker(const string& Name, const double& Percent, RealEstate\* Estates); -> експлицитен конструктор; задава стойностите, подадени при създаването на обекта

void printFoundEstates(list<Estate\*> found, list<Estate\*>\* vip = nullptr); -> извежда всички намерени имоти

4.10. В класът **SimpleBroker** са използвани следните функции:

SimpleBroker(); -> подразбиращ се конструктор; инициализира празен обект

SimpleBroker(const string& Name, const double& Percent, RealEstate\* Estates); -> експлицитен конструктор; задава стойностите, подадени при създаването на обекта

void addVIP(Estate\* estate); -> добавя имот към списъка с ВИП обяви

void deleteVIP(Estate\* estate); -> изтрива имот от списъка с ВИП обяви

void search(string address, string owner, int minPrice, int maxPrice, double size, int room, int floor, double yardSize, list<string> comunications, string estateType); -> търси имот според изискванията на клиента

void printFoundEstates(list<Estate\*> found, list<Estate\*>\* vip = nullptr); -> извежда всички намерени имоти

4.11. В класът **UI** са използвани следните функции:

UI(); -> подразбиращ се конструктор; инициализира празен обект

void showMainMenu(); -> извежда главното меню

void showSearchMenu(Broker\* broker); -> извежда менюто за търсене

V. Програмен код

**kursova.h**

#include <iostream>

#include <string>

#include <list>

#include <vector>

#include <algorithm>

#include <iomanip>

using namespace std;

**kursova.cpp**

#include "kursova.h"

#include "UI.h"

int main() {

UI\* menu = new UI();

menu->showMainMenu();

return 0;

}

**Estate.h**

#pragma once

#include "kursova.h"

class Estate

{

protected:

string address;

string owner;

int price;

double size;

public:

Estate();

Estate(string Address, string Owner, int Price, double Size);

Estate(const Estate& other);

const string& getAddress() const;

const int& getPrice() const;

const double& getSize() const;

const string& getOwner() const;

void setPrice(const int& Price);

void setAddress(const string& Address);

void setOwner(const string& Owner);

void setSize(const double& Size);

bool operator<(Estate& e);

friend ostream& operator<<(ostream& out, Estate& e);

virtual void printEstate(ostream& out) = 0;

virtual const int& getRoom() const;

virtual const int& getFloor() const;

virtual const double& getYardSize() const;

virtual const list<string>& getComunications() const;

};

**Estate.cpp**

#include "Estate.h"

Estate::Estate() {

address = "Shumen";

owner = "Mihail";

price = 50000;

size = 80.00;

}

Estate::Estate(string Address, string Owner, int Price, double Size) {

address = Address;

owner = Owner;

price = Price;

size = Size;

}

inline Estate::Estate(const Estate& other) {

address = other.address;

owner = other.owner;

price = other.price;

size = other.size;

}

const string& Estate::getAddress() const {

return address;

}

const int& Estate::getPrice() const {

return price;

}

const double& Estate::getSize() const {

return size;

}

const string& Estate::getOwner() const {

return owner;

}

void Estate::setPrice(const int& Price) {

price = Price;

}

void Estate::setAddress(const string& Address) {

address = Address;

}

void Estate::setOwner(const string& Owner) {

owner = Owner;

}

void Estate::setSize(const double& Size) {

size = size;

}

const int& Estate::getRoom() const {

return 0;

}

const int& Estate::getFloor() const {

return 0;

}

const double& Estate::getYardSize() const {

return 0;

}

const list<string>& Estate::getComunications() const {

return list<string>();

}

bool Estate::operator<(Estate& e) {

return price < e.price;

}

ostream& operator<<(ostream& out, Estate& e) {

e.printEstate(out);

return out;

}

**Flat.h**

#pragma once

#include "Estate.h"

class Flat :

public Estate

{

protected:

int room;

int floor;

public:

Flat(string Address, string Owner, int Price, double Size, int Room, int Floor);

void printEstate(ostream& out);

void setRoom(const int& Room);

void setFloor(const int& Floor);

const int& getRoom() const;

const int& getFloor() const;

friend ostream& operator<<(ostream& out, Flat& f);

};

**Flat.cpp**

#include "Flat.h"

Flat::Flat(string Address, string Owner, int Price, double Size, int Room, int Floor) : Estate(Address, Owner, Price, Size) {

room = Room;

floor = Floor;

}

void Flat::printEstate(ostream& out) {

out << setw(10) << setfill(' ') << left << "|Flat"

<< "|" << setw(24) << setfill(' ') << left << address

<< "|" << setw(14) << setfill(' ') << left << owner

<< "|" << setw(9) << setfill(' ') << left << price

<< "|" << setw(9) << setfill(' ') << left << size

<< "|" << setw(9) << setfill(' ') << left << room

<< "|" << setw(9) << setfill(' ') << left << floor

<< "|" << setw(14) << setfill(' ') << left << ""

<< "|" << setw(19) << setfill(' ') << left << ""

<< "|" << endl;

out << setw(126) << setfill('-') << "" << endl;

}

void Flat::setRoom(const int& Room) {

room = Room;

}

void Flat::setFloor(const int& Floor) {

floor = Floor;

}

const int& Flat::getRoom() const {

return room;

}

const int& Flat::getFloor() const {

return floor;

}

ostream& operator<<(ostream& out, Flat& f) {

f.printEstate(out);

return out;

}

**House.h**

#pragma once

#include "Flat.h"

class House :

public Flat

{

private:

double yardSize;

public:

House(string Address, string Owner, int Price, double Size, int Room, int Floor, double YardSize);

void printEstate(ostream& out);

void setYardSize(const double& YardSize);

const double& getYardSize() const;

friend ostream& operator<<(ostream& out, House& h);

};

**House.cpp**

#include "House.h"

House::House(string Address, string Owner, int Price, double Size, int Room, int Floor, double YardSize) : Flat(Address, Owner, Price, Size, Room, Floor) {

yardSize = YardSize;

}

void House::printEstate(ostream& out) {

out << setw(10) << setfill(' ') << left << "|House"

<< "|" << setw(24) << setfill(' ') << left << address

<< "|" << setw(14) << setfill(' ') << left << owner

<< "|" << setw(9) << setfill(' ') << left << price

<< "|" << setw(9) << setfill(' ') << left << size

<< "|" << setw(9) << setfill(' ') << left << room

<< "|" << setw(9) << setfill(' ') << left << floor

<< "|" << setw(14) << setfill(' ') << left << yardSize

<< "|" << setw(19) << setfill(' ') << left << ""

<< "|" << endl;

out << setw(126) << setfill('-') << "" << endl;

}

void House::setYardSize(const double& YardSize) {

yardSize = YardSize;

}

const double& House::getYardSize() const {

return yardSize;

}

ostream& operator<<(ostream& out, House& h) {

h.printEstate(out);

return out;

}

**Plot.h**

#pragma once

#include "Estate.h"

enum class Comunication { water, electricity, phone, road, sewage };

class Plot :

public Estate

{

private:

list<Comunication> comunication;

double yardSize;

public:

Plot(string Address, string Owner, int Price, double Size, list<string> comunications, double YardSize);

void printEstate(ostream& out);

void setYardSize(const double& YardSize);

const double& getYardSize() const;

void setComunications(const list<string> comunications);

const list<string>& getComunications() const;

};

**Plot.cpp**

#include "Plot.h"

Plot::Plot(string Address, string Owner, int Price, double Size, list<string> comunications, double YardSize) : Estate(Address, Owner, Price, Size) {

setComunications(comunications);

yardSize = YardSize;

}

void Plot::printEstate(ostream& out) {

out << setw(10) << setfill(' ') << left << "|Plot"

<< "|" << setw(24) << setfill(' ') << left << address

<< "|" << setw(14) << setfill(' ') << left << owner

<< "|" << setw(9) << setfill(' ') << left << price

<< "|" << setw(9) << setfill(' ') << left << size

<< "|" << setw(9) << setfill(' ') << left << ""

<< "|" << setw(9) << setfill(' ') << left << ""

<< "|" << setw(14) << setfill(' ') << left << yardSize

<< "|" << setw(19) << setfill(' ') << left;

list<string> comunications = getComunications();

for (auto c : comunications) {

cout << " " << c << endl;

}

out << "|" << endl;

out << setw(126) << setfill('-') << "" << endl;

}

void Plot::setYardSize(const double& YardSize) {

yardSize = YardSize;

}

const double& Plot::getYardSize() const {

return yardSize;

}

void Plot::setComunications(const list<string> comunications) {

for (string c : comunications) {

if (c == "water") {

comunication.push\_back(Comunication::water);

}

else if (c == "electricity") {

comunication.push\_back(Comunication::electricity);

}

else if (c == "phone") {

comunication.push\_back(Comunication::phone);

}

else if (c == "road") {

comunication.push\_back(Comunication::road);

}

else if (c == "sewage") {

comunication.push\_back(Comunication::sewage);

}

}

}

const list<string>& Plot::getComunications() const {

list<string> comunications;

for (Comunication c : comunication) {

switch (c) {

case Comunication::water:

comunications.push\_back("water");

break;

case Comunication::electricity:

comunications.push\_back("electricity");

break;

case Comunication::phone:

comunications.push\_back("phone");

break;

case Comunication::road:

comunications.push\_back("road");

break;

case Comunication::sewage:

comunications.push\_back("sewage");

break;

}

}

return comunications;

}

ostream& operator<<(ostream& out, Plot& p) {

p.printEstate(out);

return out;

}

**Broker.h**

#pragma once

#include "kursova.h"

#include "RealEstate.h"

class Broker

{

protected:

RealEstate\* estates;

string name;

double percent;

public:

Broker();

Broker(const string& Name, const double& Percent, RealEstate\* Estates);

void setEstates(RealEstate\* Estates);

RealEstate\* getEstates();

void setName(const string& Name);

const string& getName() const;

void setPercent(const double& Percent);

const double& getPercent() const;

virtual void addVIP(Estate\* estate) = 0;

virtual void deleteVIP(Estate\* estate) = 0;

virtual void search(string address, string owner, int minPrice, int maxPrice, double size, int room, int floor, double yardSize, list<string> comunications, string estateType) = 0;

virtual void printFoundEstates(list<Estate\*> found, list<Estate\*>\* vip = nullptr) = 0;

};

**RealEstate.h**

#pragma once

#include "kursova.h"

#include "Estate.h"

class RealEstate

{

private:

list<Estate\*> estates;

public:

void registerEstate(Estate\* estate);

void deleteEstate(Estate\* estate);

list<Estate\*> getEstates();

void printEstates();

};

**RealEstate.cpp**

#include "RealEstate.h"

void RealEstate::registerEstate(Estate\* estate) {

estates.push\_back(estate);

}

void RealEstate::deleteEstate(Estate\* estate) {

estates.remove(estate);

}

list<Estate\*> RealEstate::getEstates() {

return estates;

}

void RealEstate::printEstates() {

cout << setw(126) << setfill('-') << "" << endl;

cout << setw(10) << setfill(' ') << left << "|Type"

<< setw(25) << setfill(' ') << left << "|Address"

<< setw(15) << setfill(' ') << left << "|Owner"

<< setw(10) << setfill(' ') << left << "|Price"

<< setw(10) << setfill(' ') << left << "|Size"

<< setw(10) << setfill(' ') << left << "|Rooms"

<< setw(10) << setfill(' ') << left << "|Floor"

<< setw(15) << setfill(' ') << left << "|Yard size"

<< setw(20) << setfill(' ') << left << "|Comunications"

<< "|" << endl;

cout << setw(126) << setfill('-') << "" << endl;

for (auto estate : estates)

{

estate->printEstate(cout);

}

}

**Broker.cpp**

#include "Broker.h"

Broker::Broker() {

name = "Mihail";

percent = 0.5;

estates = new RealEstate();

}

Broker::Broker(const string& Name, const double& Percent, RealEstate\* Estates) {

name = Name;

percent = Percent;

estates = Estates;

}

void Broker::setEstates(RealEstate\* Estates) {

estates = Estates;

}

RealEstate\* Broker::getEstates() {

return estates;

}

void Broker::setName(const string& Name) {

name = Name;

}

const string& Broker::getName() const {

return name;

}

void Broker::setPercent(const double& Percent) {

percent = Percent;

}

const double& Broker::getPercent() const {

return percent;

}

**VIPBroker.h**

#pragma once

#include "Broker.h"

class VIPBroker :

public Broker

{

protected:

RealEstate\* vip;

public:

VIPBroker();

VIPBroker(const string& Name, const double& Percent, RealEstate\* Estates, RealEstate\* vipEstates);

VIPBroker(const string& Name, const double& Percent, RealEstate\* Estates);

void addVIP(Estate\* estate);

void deleteVIP(Estate\* estate);

void search(string address, string owner, int minPrice, int maxPrice, double size, int room, int floor, double yardSize, list<string> comunications, string estateType);

};

**VIPBroker.cpp**

#include "VIPBroker.h"

VIPBroker::VIPBroker() : Broker() {

vip = new RealEstate();

}

VIPBroker::VIPBroker(const string& Name, const double& Percent, RealEstate\* Estates, RealEstate\* vipEstates) :

Broker(Name, Percent, Estates) {

vip = vipEstates;

}

VIPBroker::VIPBroker(const string& Name, const double& Percent, RealEstate\* Estates) :

Broker(Name, Percent, Estates) {

vip = new RealEstate();

}

void VIPBroker::addVIP(Estate\* estate) {

vip->registerEstate(estate);

}

void VIPBroker::deleteVIP(Estate\* estate) {

vip->deleteEstate(estate);

}

void VIPBroker::search(string address, string owner, int minPrice, int maxPrice, double size, int room, int floor, double yardSize, list<string> comunications, string estateType) {

list<Estate\*> found;

for (auto estate : estates->getEstates()) {

if (

(address != "" && estate->getAddress() == address) ||

(owner != "" && estate->getOwner() == owner) ||

(minPrice != 0 && estate->getPrice() >= minPrice) ||

(maxPrice != 0 && estate->getPrice() <= maxPrice) ||

(size != 0 && estate->getSize() == size) ||

(room != 0 && estate->getRoom() == room) ||

(floor != 0 && estate->getFloor() == floor) ||

(yardSize != 0 && estate->getYardSize() == yardSize) ||

(comunications != list<string>() && estate->getComunications() == comunications) ||

(estateType != "" && typeid(\*estate).name() == "class " + estateType)

) {

found.push\_back(estate);

}

}

list<Estate\*> vipEstates = vip->getEstates();

found.sort();

vipEstates.sort();

list<Estate\*> foundVipEstates;

set\_intersection(found.begin(), found.end(), vipEstates.begin(), vipEstates.end(), back\_inserter(foundVipEstates));

list<Estate\*> foundWithoutVipEstates;

set\_difference(found.begin(), found.end(), foundVipEstates.begin(), foundVipEstates.end(), back\_inserter(foundWithoutVipEstates));

printFoundEstates(foundWithoutVipEstates, &foundVipEstates);

}

**HonesBroker.h**

#pragma once

#include "VIPBroker.h"

class HonestBroker :

public VIPBroker

{

public:

HonestBroker();

HonestBroker(const string& Name, const double& Percent, RealEstate\* Estates, RealEstate\* vipEstates);

HonestBroker(const string& Name, const double& Percent, RealEstate\* Estates);

void printFoundEstates(list<Estate\*> found, list<Estate\*>\* vip = nullptr);

};

**HonesBroker.cpp**

#include "HonestBroker.h"

HonestBroker::HonestBroker() :

VIPBroker() {}

HonestBroker::HonestBroker(const string& Name, const double& Percent, RealEstate\* Estates, RealEstate\* vipEstates) :

VIPBroker(Name, Percent, Estates, vipEstates) {}

HonestBroker::HonestBroker(const string& Name, const double& Percent, RealEstate\* Estates) :

VIPBroker(Name, Percent, Estates) {}

void HonestBroker::printFoundEstates(list<Estate\*> found, list<Estate\*>\* vip) {

cout << setw(126) << setfill('-') << "" << endl;

cout << setw(10) << setfill(' ') << left << "|Type"

<< setw(25) << setfill(' ') << left << "|Address"

<< setw(15) << setfill(' ') << left << "|Owner"

<< setw(10) << setfill(' ') << left << "|Price"

<< setw(10) << setfill(' ') << left << "|Size"

<< setw(10) << setfill(' ') << left << "|Rooms"

<< setw(10) << setfill(' ') << left << "|Floor"

<< setw(15) << setfill(' ') << left << "|Yard size"

<< setw(20) << setfill(' ') << left << "|Comunications"

<< "|" << endl;

cout << setw(126) << setfill('-') << "" << endl;

for (auto e : \*vip) {

int prevPrice = e->getPrice();

e->setPrice(prevPrice + prevPrice \* percent);

cout << \*e;

e->setPrice(prevPrice);

}

for (auto e : found) {

cout << \*e;

}

}

**DishonestBroker.h**

#pragma once

#include "VIPBroker.h"

class DishonestBroker :

public VIPBroker

{

public:

DishonestBroker();

DishonestBroker(const string& Name, const double& Percent, RealEstate\* Estates, RealEstate\* vipEstates);

DishonestBroker(const string& Name, const double& Percent, RealEstate\* Estates);

void printFoundEstates(list<Estate\*> found, list<Estate\*>\* vip = nullptr);

};

**DishonestBroker.cpp**

#include "DishonestBroker.h"

DishonestBroker::DishonestBroker() :

VIPBroker() {}

DishonestBroker::DishonestBroker(const string& Name, const double& Percent, RealEstate\* Estates, RealEstate\* vipEstates) :

VIPBroker(Name, Percent, Estates, vipEstates) {}

DishonestBroker::DishonestBroker(const string& Name, const double& Percent, RealEstate\* Estates) :

VIPBroker(Name, Percent, Estates) {}

void DishonestBroker::printFoundEstates(list<Estate\*> found, list<Estate\*>\* vip) {

cout << setw(126) << setfill('-') << "" << endl;

cout << setw(10) << setfill(' ') << left << "|Type"

<< setw(25) << setfill(' ') << left << "|Address"

<< setw(15) << setfill(' ') << left << "|Owner"

<< setw(10) << setfill(' ') << left << "|Price"

<< setw(10) << setfill(' ') << left << "|Size"

<< setw(10) << setfill(' ') << left << "|Rooms"

<< setw(10) << setfill(' ') << left << "|Floor"

<< setw(15) << setfill(' ') << left << "|Yard size"

<< setw(20) << setfill(' ') << left << "|Comunications"

<< "|" << endl;

cout << setw(126) << setfill('-') << "" << endl;

for (auto e : \*vip) {

int prevPrice = e->getPrice();

int prevSize = e->getSize();

e->setPrice(prevPrice + prevPrice \* percent);

e->setPrice(e->getPrice() \* ((double)rand() / (RAND\_MAX)));

e->setSize(e->getSize() \* ((double)rand() / (RAND\_MAX)));

cout << \*e;

e->setPrice(prevPrice);

e->setSize(prevSize);

}

for (auto e : found) {

cout << \*e;

}

}

**SimpleBroker.h**

#pragma once

#include "Broker.h"

class SimpleBroker :

public Broker

{

public:

SimpleBroker();

SimpleBroker(const string& Name, const double& Percent, RealEstate\* Estates);

void addVIP(Estate\* estate);

void deleteVIP(Estate\* estate);

void search(string address, string owner, int minPrice, int maxPrice, double size, int room, int floor, double yardSize, list<string> comunications, string estateType);

void printFoundEstates(list<Estate\*> found, list<Estate\*>\* vip = nullptr);

};

**SimpleBroker.cpp**

#include "SimpleBroker.h"

SimpleBroker::SimpleBroker() : Broker() {}

SimpleBroker::SimpleBroker(const string& Name, const double& Percent, RealEstate\* Estates) : Broker(Name, Percent, Estates) {}

void SimpleBroker::addVIP(Estate\* estate) {

cout << "Simple broker doesn't have VIP estates";

}

void SimpleBroker::deleteVIP(Estate\* estate) {

cout << "Simple broker doesn't have VIP estates";

}

void SimpleBroker::search(string address, string owner, int minPrice, int maxPrice, double size, int room, int floor, double yardSize, list<string> comunications, string estateType) {

list<Estate\*> found;

for (auto estate : estates->getEstates()) {

if (

(address != "" && estate->getAddress() == address) ||

(owner != "" && estate->getOwner() == owner) ||

(minPrice != 0 && estate->getPrice() >= minPrice) ||

(maxPrice != 0 && estate->getPrice() <= maxPrice) ||

(size != 0 && estate->getSize() == size) ||

(room != 0 && estate->getRoom() == room) ||

(floor != 0 && estate->getFloor() == floor) ||

(yardSize != 0 && estate->getYardSize() == yardSize) ||

(comunications != list<string>() && estate->getComunications() == comunications) ||

(estateType != "" && typeid(\*estate).name() == "class " + estateType)

) {

found.push\_back(estate);

}

}

printFoundEstates(found);

}

void SimpleBroker::printFoundEstates(list<Estate\*> found, list<Estate\*>\* vip) {

cout << setw(126) << setfill('-') << "" << endl;

cout << setw(10) << setfill(' ') << left << "|Type"

<< setw(25) << setfill(' ') << left << "|Address"

<< setw(15) << setfill(' ') << left << "|Owner"

<< setw(10) << setfill(' ') << left << "|Price"

<< setw(10) << setfill(' ') << left << "|Size"

<< setw(10) << setfill(' ') << left << "|Rooms"

<< setw(10) << setfill(' ') << left << "|Floor"

<< setw(15) << setfill(' ') << left << "|Yard size"

<< setw(20) << setfill(' ') << left << "|Comunications"

<< "|" << endl;

cout << setw(126) << setfill('-') << "" << endl;

for (auto e : found) {

cout << \*e;

}

}

**UI.h**

#pragma once

#include "kursova.h"

#include "RealEstate.h"

#include "Broker.h"

#include "House.h"

#include "HonestBroker.h"

#include "DishonestBroker.h"

#include "SimpleBroker.h"

class UI

{

private:

RealEstate\* estates;

vector<Broker\*> brokers;

void loadData();

public:

UI();

void showMainMenu();

void showSearchMenu(Broker\* broker);

};

**UI.cpp**

#include "UI.h"

void UI::loadData() {

House\* house = new House("Varna, Center", "Pavel", 50000, 200.00, 1, 3, 300.00);

estates->registerEstate(house);

House\* house1 = new House("Varna, Troshevo", "Pavel", 500000, 150.50, 10, 2, 500.50);

estates->registerEstate(house1);

House\* house2 = new House("Varna, Levski", "Hristiyan", 110000, 130, 4, 8, 300);

estates->registerEstate(house2);

House\* house3 = new House("Varna, Levski", "Kameliya", 80000, 90, 3, 1, 50);

estates->registerEstate(house3);

Flat\* flat = new Flat("Varna, Vladislavovo", "Kameliya", 45000, 90, 5, 2);

estates->registerEstate(flat);

Flat\* flat1 = new Flat("Varna, Center", "Pavel", 90000, 75.4, 3, 2);

estates->registerEstate(flat1);

Flat\* flat2 = new Flat("Varna, Levski", "Hristiyan", 90000, 130, 4, 5);

estates->registerEstate(flat2);

Flat\* flat3 = new Flat("Varna, Briz", "Hristiyan", 70000, 70, 2, 4);

HonestBroker\* honestBroker = new HonestBroker("Mihail", 0.4, estates);

honestBroker->addVIP(house);

honestBroker->addVIP(flat);

honestBroker->addVIP(house2);

honestBroker->addVIP(house3);

DishonestBroker\* dishonestBroker = new DishonestBroker("Ivan", 0.2, estates);

dishonestBroker->addVIP(house2);

dishonestBroker->addVIP(flat2);

SimpleBroker\* simpleBroker = new SimpleBroker("Maria", 0.3, estates);

brokers.push\_back(honestBroker);

brokers.push\_back(dishonestBroker);

brokers.push\_back(simpleBroker);

}

UI::UI() {

estates = new RealEstate();

loadData();

}

void UI::showMainMenu() {

int choice;

do

{

cout << "+++++++++++++++++++++" << endl;

cout << "+ Choose: +" << endl;

cout << "+1. Helpful Broker +" << endl;

cout << "+2. Dishonest Broker+" << endl;

cout << "+3. Simple Broker +" << endl;

cout << "+4. Exit +" << endl;

cout << "+++++++++++++++++++++" << endl;

cout << "Enter your choice: ";

cin >> choice;

switch (choice)

{

case 1:

system("cls");

cout << "You chose the HONEST BROKER!" << endl;

showSearchMenu(brokers[0]);

break;

case 2:

system("cls");

cout << "You chose the DISHONEST BROKER!" << endl;

showSearchMenu(brokers[1]);

break;

case 3:

system("cls");

cout << "You chose the SIMPLE BROKER!" << endl;

showSearchMenu(brokers[2]);

break;

case 4:

break;

default:

system("cls");

cin.clear();

cin.ignore(1000, '\n');

cout << "Error!!!" << endl;

cout << "Choose again(between 1-4)!" << endl << endl;

}

} while (choice != 4);

}

inline void UI::showSearchMenu(Broker\* broker) {

int choice;

string address = "";

string owner = "";

int minPrice = 0;

int maxPrice = 0;

double size = 0;

int room = 0;

int floor = 0;

double yardSize = 0;

string estateType = "";

list<string> comunications;

bool running = true;

do

{

cout << "Input search criterias" << endl;

cout << "-------------------------" << endl;

cout << "1. Address" << (address != "" ? " (" + address + ")" : "") << endl;

cout << "2. Owner" << (owner != "" ? " (" + owner + ")" : "") << endl;

cout << "3. Min price" << (minPrice != 0 ? " (" + to\_string(minPrice) + ")" : "") << endl;

cout << "4. Max price" << (maxPrice != 0 ? " (" + to\_string(maxPrice) + ")" : "") << endl;

cout << "5. Size" << (size != 0 ? " (" + to\_string(size) + ")" : "") << endl;

cout << "6. Rooms" << (room != 0 ? " (" + to\_string(room) + ")" : "") << endl;

cout << "7. Floor" << (floor != 0 ? " (" + to\_string(floor) + ")" : "") << endl;

cout << "8. Yard Size" << (yardSize != 0 ? " (" + to\_string(yardSize) + ")" : "") << endl;

cout << "9. Comunications";

if (comunications != list<string>()) {

list<string>::iterator final\_iter = comunications.end();

--final\_iter;

cout << "(";

for (list<string>::iterator c = comunications.begin(); c != comunications.end(); ++c) {

cout << \*c;

if (c != final\_iter) {

cout << ", ";

}

}

cout << ")" << endl;

}

else {

cout << endl;

}

cout << "10. Estate type" << (estateType != "" ? " (" + estateType + ")" : "") << endl;

cout << "-------------------------" << endl;

cout << "11. Search" << endl;

cout << "12. Exit" << endl;

cout << "Enter your choice: ";

cin >> choice;

switch (choice)

{

case 1: {

string city, district;

cout << endl << "Enter city: ";

cin >> city;

cout << "Enter district: ";

cin >> district;

address = city + ", " + district;

system("cls");

}

break;

case 2:

cout << endl << "Enter owner name: ";

cin >> owner;

system("cls");

break;

case 3:

cout << endl << "Enter price: ";

cin >> minPrice;

system("cls");

break;

case 4:

cout << endl << "Enter price: ";

cin >> maxPrice;

system("cls");

break;

case 5:

cout << endl << "Enter size: ";

cin >> size;

system("cls");

break;

case 6:

cout << endl << "Enter rooms count: ";

cin >> room;

system("cls");

break;

case 7:

cout << endl << "Enter floor: ";

cin >> floor;

system("cls");

break;

case 8:

cout << endl << "Enter yard size: ";

cin >> yardSize;

system("cls");

break;

case 9: {

string comunication;

char isContinue;

do {

cout << endl << "Enter comunication: ";

cin >> comunication;

comunications.push\_back(comunication);

cout << "Do you want to continue(y or n)? ";

cin >> isContinue;

} while (isContinue == 'y');

system("cls");

}

break;

case 10:

cout << endl << "Enter estate type: ";

cin >> estateType;

if (!estateType.empty())

{

estateType[0] = std::toupper(estateType[0]);

for (int i = 1; i < estateType.length(); ++i)

estateType[i] = std::tolower(estateType[i]);

}

system("cls");

break;

case 11:

system("cls");

cout << endl << "Results:" << endl

<< "-------------------------" << endl;

broker->search(address, owner, minPrice, maxPrice, size, room, floor, yardSize, comunications, estateType);

break;

case 12:

running = false;

system("cls");

break;

default:

system("cls");

cin.clear();

cin.ignore(1000, '\n');

cout << "Error!!!" << endl;

cout << "Try again!!" << endl << endl;

break;

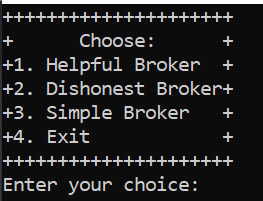
}

} while (choice != 12 && running);

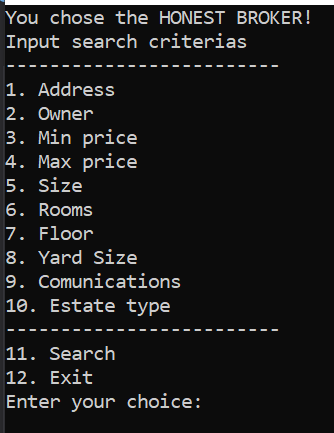
}

VI. Снимки от изпълнението на проекта

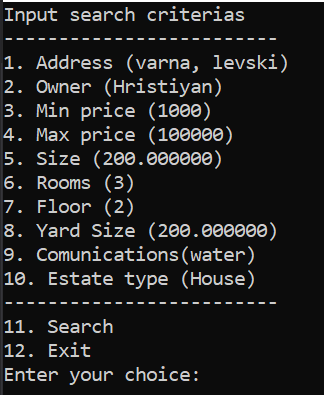
При стартиране на програмата се извежда главно меню с избор на опции:



При избор на която и да е опция (без 4) извежда меню с избор на критерии.



Потребителят въвежда критериите си, в следствие на което на конзолата се извеждат въведените данни.



При избор на опция 11 на конзолата се извеждат всички имоти в табличен вид.

