**Universidade Federal de Ouro Preto Departamento de Computação**

Augusto Ferreira Guilarducci (20.1.4012)

Caio Monteiro (20.1.4110)

Matheus Silva Araújo (20.1.1018)

Paulo Correa (20.1.4036)

Pedro Lucas Damasceno (20.1.4003)

**Trabalho Prático II – Programação Orientada a Objetos**

Implementação de uma corretora de imóveis em Java com interface gráfica

**Ouro Preto, Minas Gerais**

**2021**

**SUMÁRIO**

1. **Introdução** .................................................................................................. 3
2. **toString() e toWrite()** ................................................................................. 4
3. **Leitura do arquivo e array list polimórfico de objetos** ........................... 4
4. **Registro da coleção em memória secundária** ........................................... 5
5. **Comparação entre objetos** ......................................................................... 5
6. **Busca por atributos** .................................................................................... 6
7. **Busca por tipo de objeto** ............................................................................ 6
8. **Interface gráfica** .......................................................................................... 7
9. **Introdução**

Este relatório se refere ao segundo trabalho prático da disciplina Programação Orientada a Objetos, realizado durante o período 21.1, cujo objetivo consiste na implementação de uma corretora de imóveis em Java com interface gráfica.

As etapas do desenvolvimento serão explicadas em detalhes adiante e o trabalho pode ser visualizado no repositório do *Github* a partir do link https://github.com/augustofgui/grupo-21.1

1. ***toString()* e *toWrite()***

De forma análoga à sobrecarga do operador *cout* da linguagem C++, utilizamos o

*override* do método *toString()* para imprimir as informações de cada imóvel (incluindo as particulares) conforme o modelo enunciado no trabalho. Seguimos uma abordagem praticamente idêntica para gravar os dados de uma coleção em memória secundária e implementamos o método *toWrite()*.

@Override

public String toString**()** **{**

**return** " Proprietário: " **+** **this.**getProprietario**()** **+** "\nValor: " **+** **this.**getValor**()**

**+** "\nNúmero de quartos: " **+** **this.**getQuartos**()** **+** "\nRua: " **+** **this.**getRua**()** **+**

"\nBairro: " **+** **this.**getBairro**()** **+** "\nCidade: " **+** **this.**getCidade**()** **+** "\nAndares: " **+** **this.**getAndares**()** **+** "\n"**;**

**}**

@Override

public String toWrite**()** **{**

**return** "casa;" **+** **this.**getValor**()** **+** ";" **+** **this.**getProprietario**()** **+** ";" **+** **this.**getRua**()** **+** ";"

**+** **this.**getBairro**()** **+** ";" **+** **this.**getCidade**()** **+** ";" **+** **this.**getNumero**()** **+** ";" **+** **this.**getQuartos**()**

**+** ";" **+** **this.**getBanheiros**()** **+** ";" **+** **this.**getAndares**()** **+** ";"

**+** booleanToInt**(this.**isSala\_jantar**())** **+** ";\n"**;**

**}**

1. **Leitura do arquivo e *array list* polimórfico de objetos**

Estabelecemos que o arquivo de imóveis seria selecionado manualmente pelo usuário através da interface gráfica para eliminar problemas de diretório no momento da execução do programa. A partir disso, criamos um Scanner e utilizamos as funções de *hasNextLine()* e *nextLine()* para realizar a navegação no arquivo e a função *split()* para separar os dados de cada linha a partir do divisor “;”. Após a leitura e separação das informações, criamos um objeto de acordo com o conteúdo do índice zero do vetor de *strings* que determina o tipo do imóvel (casa, apartamento ou chácara), inserimos os dados a partir do construtor do objeto e, por fim, inserimos o objeto em um *ArrayList<Imovel>* polimórfico.

Scanner txt **=** **new** Scanner**(**database\_imoveis**);**

**while** **(**txt**.**hasNextLine**())** **{**

String imovel **=** txt**.**nextLine**();**

String**[]** data **=** imovel**.**split**(**";"**);**

**if** **(**data**[**0**].**equals**(**"casa"**))**

imoveisArquivo**.**add**(new** Casa**(...));**

**else** **if** **(**data**[**0**].**equals**(**"apartamento"**))**

imoveisArquivo**.**add**(new** Apartamento**(...));**

**else** **if** **(**data**[**0**].**equals**(**"chacara"**))**

imoveisArquivo**.**add**(new** Chacara**(...));**

**else**

System**.**out**.**println**(**"ERRO : Imóvel inválido.\n"**);**

**}**

txt**.**close**();**

**}** **catch** **(**FileNotFoundException e**)** **{**

System**.**out**.**println**(**"ERRO : Arquivo não encontrado.\n"**);**

e**.**printStackTrace**();**

**}**

**return** imoveisArquivo**;**

**}**

1. **Registro da coleção em memória secundária**

Utilizamos o método *toWrite()* para formatar a *string* seguindo o modelo do arquivo *database\_imoveis.txt*. Em seguida, criamos um *FileWriter* e iteramos sobre a coleção a ser salva.

public static void salvarColecao**(**ArrayList**<**Imovel**>** imoveis**)** **{**

**try** **{**

FileWriter colecao **=** **new** FileWriter**(**"colecao\_imoveis.txt"**);**

**for** **(**Imovel imovel **:** imoveis**)**

colecao**.**write**(**imovel**.**toWrite**());**

colecao**.**close**();**

System**.**out**.**println**(**"Coleção salva em colecao\_imoveis.txt!"**);**

**}** **catch** **(**IOException e**)** **{**

System**.**out**.**println**(**"ERRO : I/O."**);**

e**.**printStackTrace**();**

**}**

**}**

1. **Comparação entre objetos**

Implementamos a interface *Java Comparable* na classe base para viabilizar a comparação de objetos a partir do atributo *valor* e ordená-los. Ademais, implementamos também o método *compareTo* e, dessa forma, para realizar a ordenação, basta invocar o método *Collections.sort()* da classe *java.util.Collections*.

1. **Busca por atributos**

public int compareTo**(**Imovel imovel**)**

**{**

**return** Float**.**compare**(this.**getValor**(),** imovel**.**getValor**());**

**}**

Todas as funções de busca por atributos seguem um mesmo padrão. Iteramos sobre a coleção e, caso a condição se satisfaça, o objeto é inserido em uma nova coleção e retornado pela função. Para comparar atributos numéricos utilizamos operadores aritméticos e para comparar *strings* utilizamos a função *equals()*. Segue um exemplo que realiza a busca por proprietário.

public static ArrayList**<**Imovel**>** buscarProprietario**(**ArrayList**<**Imovel**>** imoveis**,** String proprietario**)** **{**

ArrayList**<**Imovel**>** resultado **=** **new** ArrayList**<**Imovel**>();**

**for** **(**Imovel imovel **:** imoveis**)** **{**

**if** **(**imovel**.**getProprietario**().**equals**(**proprietario**))**

resultado**.**add**(**imovel**);**

**}**

1. **Busca por tipo de objeto**

Utilizamos o operador instanceof para identificar a natureza do objeto pertencente à lista polimórfica e adicioná-lo a uma nova coleção selecionada que é retornada pela função.

**switch** **(**tipo\_imovel**)** **{**

**case** 1**:**

**for** **(**Imovel imovel **:** imoveis**)** **{**

**if** **(**imovel instanceof Casa**)**

resultado**.**add**(**imovel**);**

**}**

**break;**

**case** 2**:**

**for** **(**Imovel imovel **:** imoveis**)** **{**

**if** **(**imovel instanceof Apartamento**)**

resultado**.**add**(**imovel**);**

**}**

**break;**

**case** 3**:**

**for** **(**Imovel imovel **:** imoveis**)** **{**

**if** **(**imovel instanceof Chacara**)**

resultado**.**add**(**imovel**);**

**}**

**break;**

**}**

1. **Interface Gráfica**

A Interface Gráfica foi feita em Swing com auxílio do Window Builder , puglin da IDE Eclipse. Assim foi possível criar visualmente o esqueleto da interface, e ao mesmo tempo, alterar porções do código em swing para refinar alguns detalhes da tela. As telas tiveram um design planejado no Figma, que foi seguido, exportando planos de fundo para a aplicação, que foram inseridas por um JLabel.

Com esse conjunto foi possível chegar num resultado bastante favorável quanto ao design, funcionalidade e fluidez da aplicação.

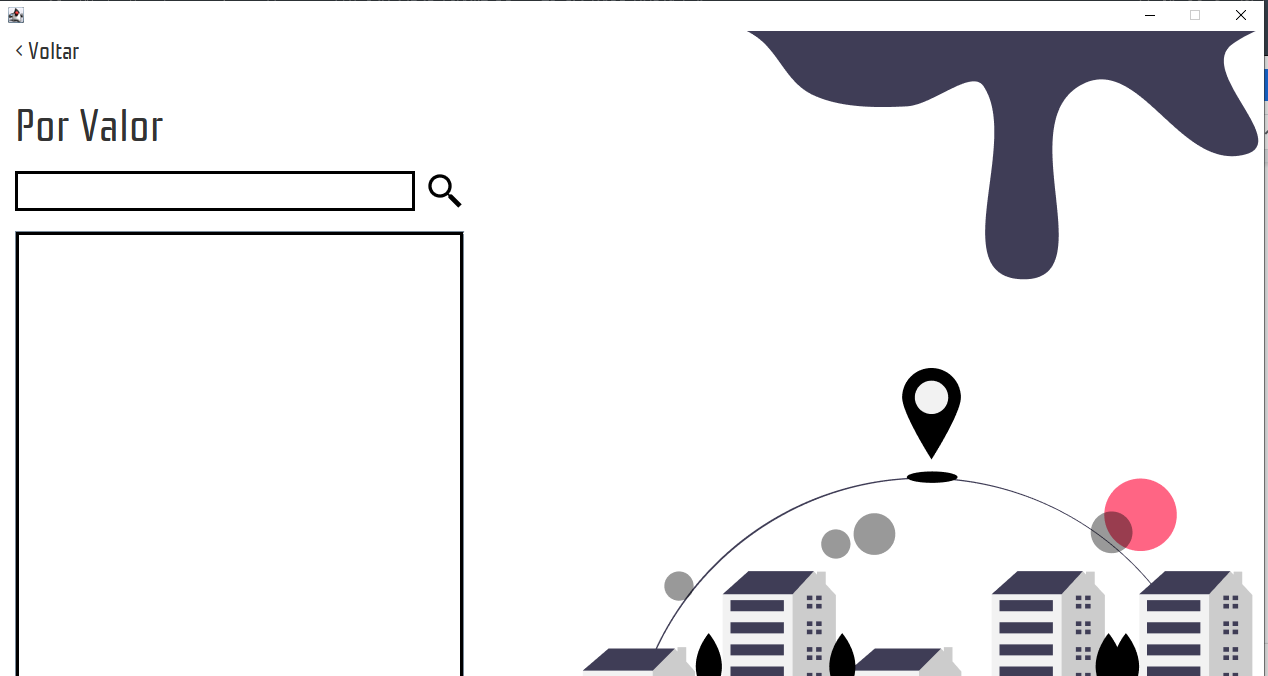
A MainPage é uma classe feita para criar a parte inicial da interface, é nessa janela onde o arquivo que iremos utilizar é escolhido - nesse caso, seria o “database\_imoveis.txt” - e o processo da busca é de fato realizado. Dito isso, utilizamos funções para indicar o tamanho de que seria a tela e fizemos sua personalização usando de base um design que fizemos através da web.



A LandingPage é uma classe feita para criar a segunda janela da interface, assim que o arquivo for escolhido, essa página é aberta e nela será possível selecionar qual o método de pesquisa que mais seria de interesse do usuário - o valor, o proprietário, o tipo, a cidade e o número de quartos.



A SearchPage é uma classe feita para criar a parte final da interface gráfica, local responsável pela pesquisa propriamente dita. A terceira página permite ao usuário filtrar apenas os imóveis que atendem exatamente aos seus requisitos, facilitando a busca do usuário.



A AplicacaoController é uma classe criada que serve para controlar o uso da interface, ativando a página que está em uso e desativar as que não estão em uso. Assim as funções de cada página ficam desacopladas uma da outra.

Supondo que a página ativa fosse a principal, o código funcionaria dessa maneira.

public void showMain() {

landing.setVisible(false);

main.setVisible(true);

search.setVisible(false);

imobiliaria.imoveis = Imobiliaria.leArquivo(arquivoSelecionado);

}

A Aplicação é uma classe criada para basicamente representar o ‘main’ geral de todo o programa que será compilado. Assim, nela é onde o JFrame é criado e gerenciado, além da inserção de suas respectivas telas - MainPage, LandingPage e SearchPage. A classe é o local responsável por colocar todo o código em prática de fato e podendo ser executado junto com a interface gráfica construída.