UNIVERSIDADE FEDERAL DE OURO PRETO - UFOP
DEPARTAMENTO DE COMPUTAÇÃO - DECOM
PROGRAMAÇÃO ORIENTADA A OBJETOS - BCC221

Gabriel Ferreira Pereira - 20.1.4015
Lavínia Fonseca Pereira - 19.1.4170
Pablo Martins Coelho - 20.1.4113

TRABALHO PRÁTICO II DE PROGRAMAÇÃO ORIENTADA A OBJETOS

Ouro Preto - MG
2021

1. Introdução

O presente relatório, tem por finalidade se tratar da implementação do código voltado para uma corretora de imóveis, essa corretora terá um sistema que gerencia imóveis, essa implementação é feita em Java e conta com a presença de uma interface gráfica.

Escolhemos os métodos utilizados através das funcionalidades necessárias para cada um, o contexto e os algoritmos responsáveis pela manipulação, adequando cada parte do código a sua respectiva necessidade e função.

Com esse trabalho foi possível implementar um código eficaz e entender o proposto a cada parte do código criado, adequando-o em todos os quesitos para que ele realize o que lhe foi proposto. Através dessa implementação, conseguimos aumentar nosso conhecimento em Java e seus adendos, sem contar o aprimoramento de nossos conhecimentos.

2. Desenvolvimento

2.1. Containers

No código só foi utilizado um container, ArrayList. Este container é o equivalente em Java para um array dinâmico. Seu tempo de acesso é O(1) o que facilita no momento de percorrer os seus valores no momento de utilizar os filtros. Além disso, como ele é uma array dinâmico, é possível aumentá-lo à medida que novos registros são lidos do arquivo de texto. Uma desvantagem desta estrutura de dados poderia ser em relação à manipulação (remoção e adição) de elementos que tem complexidade O(n), no entanto como os filtros sempre executam mais acessos do que modificações nas estruturas, ele continua sendo uma opção adequada.

2.2. Imóvel

Na classe imóvel tem-se algumas variáveis que são comuns a todos os imóveis tratados no código, ou seja, todos os imóveis utilizam dessa classe, essas variáveis são o id do imóvel, o valor do mesmo, o nome do proprietário, a localização em que o imóvel se encontra, que é a rua, o bairro e a cidade, tem também o número do imóvel, a quantidade de banheiros e a quantidade de quartos, sendo que todos esses atributos são protegidos. Essa classe é chamada de classe base, pois ela tem as operações de herança pública, pois todos os imóveis compartilham dessas informações. Nas funções do imóvel, encontram-se os getters e os setters, sendo que os setters têm a mesma finalidade nas funções, que é definir o id, o valor, o proprietário, a rua, o bairro, a cidade, o número, a quantidade de quartos e banheiros de todos os imóveis presentes no código e as funções de

getters têm o objetivo de pegar esses mesmos valores, sendo assim, com get defini e com set retorna o valor.

É possível também notar a presença da função print, que retorna na tela os dados que são comuns a todos os usuários, ou seja, todos os atributos que têm dentro da função print são comuns aos imóveis. Essa implementação traz consigo, como já foi dito, as funções e variáveis comuns a todos os imóveis, sendo a classe base, que tem como método a herança pública, onde outras classes podem herdar dessa implementação.

2.3. Apartamento

Começando pelo construtor *Apartamento*, ele possui os parâmetros de entrada próprios e também os parâmetros de entrada com relações da classe Imóvel, isso pode ser percebido através do uso da palavra ‘super’ que faz com que os objetos da classe pai sejam gerenciados, sendo que a classe Imovel é a classe pai. Os atributos que são características apenas desse tipo de imóvel são andar, que serve para determinar qual o andar do apartamento, a variável taxa_condominio, que determina a taxa de condomínio do apartamento a ser paga, tem também as variáveis elevador e sacada que servem para verificar se o apartamento possui elevador ou se ele possui sacada.

No método para realizar a impressão na tela das informações do apartamento se encontra as respectivas informações: nome do proprietário, valor do apartamento, quantidade de quartos, localização do apartamento que é a rua, o bairro e a cidade, que são todos recebidos pela palavra super da classe pai, por fim é impresso se o apartamento possui ou não possui elevador que é uma característica única desse imóvel, é possível perceber o uso do ‘@Override’ que quer dizer que um método está sendo sobrescrito da classe pai.

Nos métodos do apartamento se encontra a função getAndar, que é responsável por retornar qual o andar em que se encontra o apartamento, a função setAndar, que é responsável por definir o valor da variável andar, a função getTaxaCondominio é responsável por retornar qual a taxa de condomínio que deve ser paga, a setTaxaCondominio serve para definir a taxa de condomínio, as demais funções, getElevador, setElevador, getSacada e setSacada servem para retornar se o apartamento tem os respectivos adendos e para definir se eles tem ou não. Em geral as funções que iniciam com get fazem o retorno da informação e as funções com início set definem o valor das variáveis.

2.4. Casa

O construtor *casa* tem como parâmetros de entrada próprios além dos parâmetros de entrada da classe imóvel. Suas variáveis são *andares*, que nos informa quantos andares possuem a casa, sala_jantar que nos informa se a casa possui ou não uma sala de jantar.
Além dessas variáveis também temos as que estão presentes na classe imóvel, que nos informa no geral o valor, o proprietário, a rua, entre outros atributos mencionados no tópico imóvel. Ela também possui seus respectivos geters e seters para utilização durante a implementação do código.

Também possui o método para realizar a impressão na tela das informações do imóvel: nome do proprietário, valor do apartamento, quantidade de quartos, localização do apartamento que é a rua, o bairro e a cidade, por fim é impresso o número de andares que a casa possui.

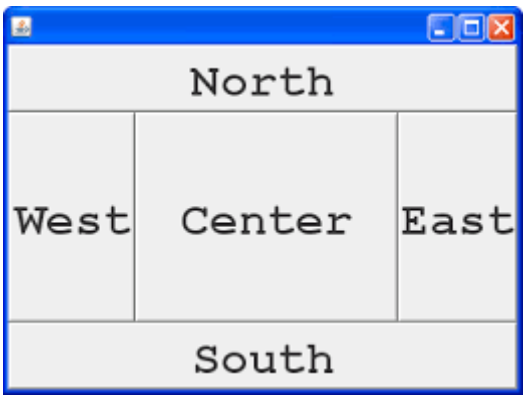
2.5. Chácara

A chacara, como os outros imóveis possui além dos parâmetros de entrada da classe imóvel, seus próprios parâmetros que são: salão_de_festa*,* salao_jogos*,* campo_futebol*, churrasqueira* e *piscina* que nos informa se a chácara possui esses respectivos adendos. Essas variáveis são todas do tipo booleanas, pois basta nos informar se o atributo é verdadeiro ou falso.

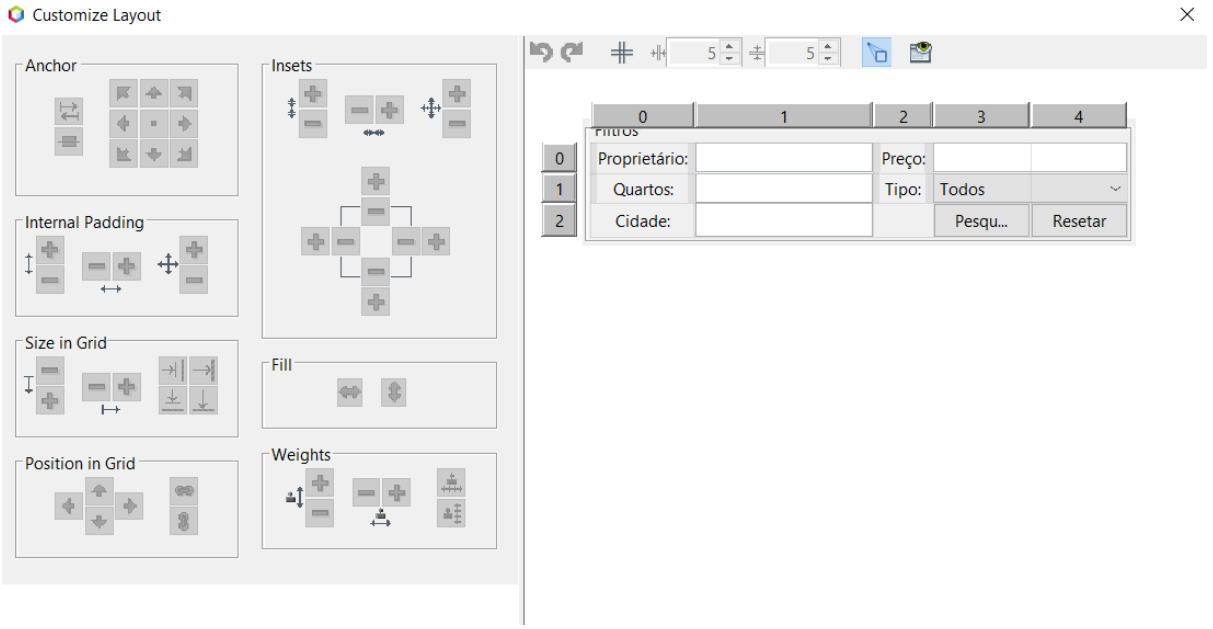
Seus Getters e Setters são referentes às suas variáveis, mas também possui acesso aos getters e setters da classe imóvel. Possui também o método para realizar a impressão na tela das informações do imóvel: nome do proprietário, valor do apartamento, quantidade de quartos, localização do apartamento que é a rua, o bairro e a cidade, que são atributos herdados da classe pai e por fim é impresso se a chácara possui piscina que é uma característica dela.

2.6. Principal

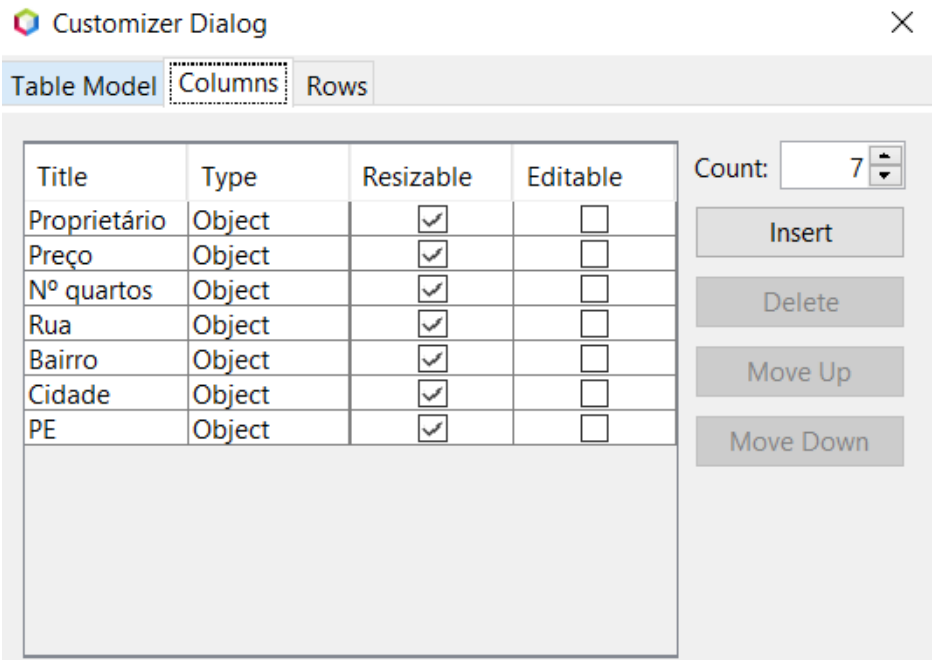
A classe principal constitui da interface gráfica e os métodos necessários para a sua manipulação. A interface gráfica consiste de um *JFrame* que utiliza o *BorderLayout*.



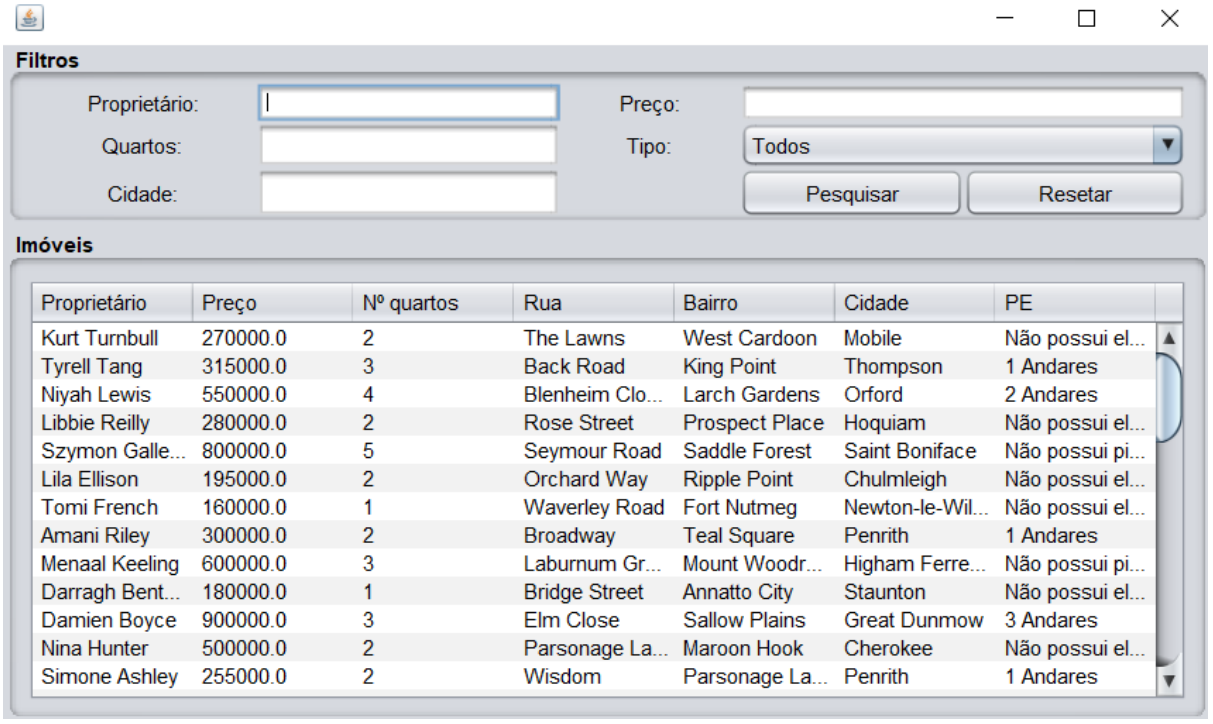
Na posição NORTH foi utilizado um JPanel que armazena os componentes relacionados aos parâmetros de filtros. Para a criação desta parte da interface foram utilizados JLabel para exibir textos, JTextField para permitir que o usuário digite e JComboBox para que o usuário possa selecionar o tipo de imóvel que ele deseja buscar. Além disso, foram disponibilizados dois botões, um para executar a pesquisa, e outro para resetar os parâmetros. Esse JPanel utiliza o GridBagLayout que organiza os elementos em células.



Na posição CENTER foi utilizado um JPanel que armazena um elemnto JScollPane que armazena um JTable. Essa tabela utiliza como modelo a seguinte configuração:



A classe principal possui também uma coleção polimórfica de objetos do tipo Imoveis, que são estabelecidos utilizando a função *getdb*. Essa coleção em seguida é enviada para o método *fillTable* que vai inserir os valores na tabela da interface gráfica. Abaixo pode ser observado a organização final da interface gráfica.



2.7. Filtros

Na classe Principal são encontrados alguns filtros, que serão responsáveis por filtrar os dados dos imóveis que serão retornados ao usuário.

Começando pelo *checkProprietario* essa função usa como principal parâmetro o atributo proprietário, em seu interno ela faz a comparação da entrada fornecida pelo usuário com todos os proprietários presentes dentro da classe imóvel. Essa comparação é feita pegando cada imóvel do vetor do tipo imóvel. No final ela retorna true para caso o proprietário seja encontrado e false para caso não seja.

Depois temos a função *filterValor* que tem como atributo principal o valor do imóvel, nesta função como as outras tem o principal intuito ir comparando valor por valor de cada imóvel, até que encontremos um valor que seja menor ou igual ao desejado pelo usuário. Depois que satisfeita essa condição o usuário tem como retorno as características do imóvel.

A função *filterQuartos* funciona de uma maneira muito semelhante à *filterValor*. É recebido o valor mínimo de quartos que o imóvel deve possuir e é devolvido uma lista com os elementos que obedecem a essa condição.

Na função *filterTipo* é verificado qual o tipo de imóvel o usuário deseja buscar, se é um apartamento, casa ou chácara. Para isso iremos passar imóvel por imóvel fazendo a verificação, por isso usamos o size para ter como retorno o tamanho do vetor de imóveis. Logo depois teremos 3 comparações, nessas comparações 2 condições devem ser satisfeitas, a primeira é com base na entrada do usuário, verificaremos se o imóvel que está sendo verificado atualmente condiz com o tipo requisitado pelo parâmetro. Se sim iremos verificar a segunda condição que verifica o tipo do imóvel, utilizando da palavra chave *instanceof*. Quando ambas as condições estiverem satisfeitas iremos retornar para o usuário o imóvel correspondente. E por último temos uma ordenação, para retornar os imóveis ordenados com base no valor para o usuário. Essa ordenação é feita utilizando o método sort da biblioteca Collections do Java. Para realizar as comparações de ordenação foi passado uma função *compare* que define como os objetos devem ser comparados.

A função *filterCidade* irá funcionar como as outras, iremos ter a entrada do usuário nos informando qual cidade deseja procurar, depois iremos fazer a comparação imóvel por imóvel e retornar ao usuário os imóveis presentes naquela cidade, porém esse retorno será ordenado com base nos valores, da mesma forma que o método anterior (com a diferença de que um está sendo ordenado de maneira crescente e o outro decrescente.

Na função *filterProprietario* tem como principal parâmetro o atributo proprietário, essa função diferente da *checkProprietario* retorna para o usuário um iterador para cada tipo de imóvel que o proprietário possua. Nesta função é utilizado um for, que vai do começo do vetor de imóveis até o final. Neste for iremos comparar a informação do proprietário fornecida pelo usuário com o atributo proprietário dos imóveis. Depois é feita a comparação do tipo do imóvel. Satisfeitas as duas condições será retornado os imóveis que correspondem a busca.

2.8. Melhor contêiner para as seguintes situações:

- Acessar uma posição específica de um contêiner;
  ArrayList possui complexidade O(1) no acesso, o que o torna a opção ideal.
- Adicionar um elemento e manter somente elementos únicos no contêiner;
  Para manter elementos únicos é possível utilizar estruturas do tipo Set ou Map e suas variações. A escolha depende do tipo de dado que vai ser utilizado, se os dados precisam estar ordenados, etc.

- Inserção no final;
  LinkedList possui um custo menor do que o ArrayList por ser uma lista duplamente encadeada possui complexidade de O(1).

- Remoção no final;
  A melhor estrutura é a LinkedList pelo mesmo motivo anterior.s

- Retornar um valor baseado em uma chave específica (Não necessariamente inteiros);
  Map, devido a possibilidade de associar quaisquer tipos de dados em um par de key e valor.

- Inserção no início;
  A melhor estrutura é a LinkedList pelo mesmo motivo que a inserção no final.

- Remoção no início;
  A melhor estrutura é a LinkedList pelo mesmo motivo que a remoção no final.

- Busca por um elemento;
  Ambos map e set possuem uma complexidade usual de O(1), sendo que a escolha entre os dois deve ser feita baseado em qual estrutura representa os dados de maneira mais apropriada.

- Contêiner com o comportamento de primeiro a entrar é o último a sair;
  Um contêiner Stack segue a semântica de UEPS (último a entrar, primeiro a sair, ou seja, primeiro a entrar, último a sair). O primeiro elemento enviado por push para a pilha é o último elemento a ser removido da pilha como o menos recente.

- Contêiner com o comportamento de primeiro a entrar é o primeiro a sair.
  Um contêiner Queue segue a semântica de PEPS (primeiro a entrar, primeiro a sair). O primeiro elemento enviado por push, ou seja, inserido na fila, é o primeiro a ser removido como o mais recente da pilha, ou seja, removido da fila.

3. Conclusão

No presente trabalho foi possível observar como as implementações de algoritmos e estruturas de dados do Java são ferramentas poderosas para o desenvolvimento de sistemas. Também foi possível utilizar os conceitos de Orientação a Objetos para reduzir a quantidade de código e aumentar a qualidade do código, utilizando de técnicas como polimorfismo, sobrecarga e downcasting, sem contar da interface gráfica feita que melhorou muito o contato do programa com o usuário.

Concluímos este relatório certos de ter acrescido informações e práticas que nos acompanharão pelo resto de nossa caminhada profissional, buscando sempre nos aprofundar e aprender coisas novas a respeito, com intuito de nos capacitarmos cada vez mais.

4. Referências

When to use LinkedList over ArrayList in Java? Disponível em: <https://stackoverflow.com/questions/322715/when-to-use-linkedlist-over-arraylist-in-java>. Acesso em: 31 dez. 2021.

Difference between ArrayList and LinkedList. Disponível em: <https://www.javatpoint.com/difference-between-arraylist-and-linkedlist>. Acesso em: 31 dez. 2021.

Java hashmap vs hashset performance. Disponível em: <https://stackoverflow.com/questions/42530042/java-hashmap-vs-hashset-performance>. Acesso em: 31 dez. 2021.

ArrayList vs. LinkedList vs. Vector. Disponível em: <https://www.programcreek.com/2013/03/arraylist-vs-linkedlist-vs-vector/>. Acesso em: 31 dez. 2021.

HashMap get/put complexity. Disponível em: <https://stackoverflow.com/questions/4553624/hashmap-get-put-complexity>. Acesso em: 31 dez. 2021.