EXA 863 - MI-PROGRAMAÇÃO 2018.2
Relatório – Roteamento de Veículos

Ian Zaque Pereira de Jesus dos Santos[1]

[1]Curso de Bacharelado em Eng. de Computação – Universidade Estadual de Feira de Santana
(UEFS) – Campus Feira de Santana
CEP 44.036-900 – Feira de Santana – BA – Brasil
ianzaque.uefs@gmail.com

1. Introdução

O Brasil é um país de dimensões continentais. Mesmo sendo tão vasto, seu sistema rodoviário possui vários pontos a serem melhorados. Determinados pontos da Bahia, por exemplo, alguns trechos de estradas estão muito esburacados e em outros ela não é pavimentada completa ou corretamente. Em certas ocasiões, novas vias de tráfego poderiam ser construídas e, deste modo, conectando mais cidades levando a uma maior interligação das rodovias.

Por conta de grandes distâncias entre cidades, preços altos de combustível e visando redução de custos dos transportes, uma empresa de entregas sediada em Feira de Santana, na Bahia, procurou uma forma eficaz e lógica de cortar os gastos e tentar render mais. A solução pensada foi diminuir o quanto que os veículos percorreriam. Para tal, foi preciso a criação de um roteiro por onde os transportes devem passar para chegar em todos os locais de entrega. A empresa decidiu contatar a Universidade Estadual de Feira de Santana para que os estudantes do curso de Engenharia de Computação pudessem resolver o problema que a assola. Estes alunos, portanto, deveriam construir um programa que leria um arquivo .txt com as informações das cidades que o transporte pode passar e fazer as entregas.

Como a empresa tem sede em Feira de Santana, o ponto de saída é o depósito da empresa na cidade e podendo ir a qualquer outra cidade, para isso, pode também passar por outras localizações que estão definidas no arquivo .txt. O software, com seu código-fonte escrito na linguagem de programação Java, deve ter uma interface gráfica que permite interação com o usuário. Nesta interface gráfica, o utilizador deve escolher as cidades que existe local de entrega que o programa fará o trajeto do menor caminho, podendo cancelar a qualquer momento o cálculo do roteiro.

2. Fundamentação teórica e Metodologia

A linguagem de programação que o software foi escrito é baseada em P.O.O (programação orientada a objetos). Este tipo de programação envolve e visa dar funcionalidades a certos objetos que são instâncias de uma classe onde é desenvolvido o código para este objeto. Pode-se então ter várias classes diferentes e de vários tipos e também vários objetos de uma mesma classe.

2.1 Classes do projeto:

Estas são as classes desenvolvidas para este projeto:

2.1.1 Classe Cidade

Essa classe é relevante em vários quesitos neste programa. Ela é responsável por guardar em seus atributos o nome de uma cidade, coordenada X e coordenada Y - como se localiza-se as cidades num plano cartesiano - desta mesma cidade. Além disso, os objetos desta classe possuem um atributo *ArrayList* que guarda objetos do tipo Cidade e são considerados como cidades que tem uma estrada ligando-as diretamente. Em seu construtor, a classe recebe os parâmetros que serão repassados aos atributos e em adicional recebe o parâmetro *int index* que é o índice desta cidade para identificação. Entretanto, este índice é usado em outras classes exploradas posteriormente (Olhar Figura 1).

2.1.2 Classe Vertice

A classe Vertice faz parte da estrutura abstrata de dados chamada grafo. Ela guarda um objeto Cidade e outro Vertice que chamado anterior é usado para fazer o cálculo de menor caminho até determinado vértice. O construtor da classe recebe a Cidade a ser guardada em seu atributo e inicializa os atributos *int custo* e *boolean visitado*, também usados no algoritmo do menor caminho de um ponto ao outro(Olhar Figura 2).

2.1.3 Classe Aresta

Esta classe é formada por 2 objetos Vertice. Ela é responsável por simbolizar a ligação desses dois vértices e possui eles como atributo e um *int peso* que representa a distância real das duas cidades dos respectivos vértices. É interessante notar que o construtor desta classe lança uma *Exception* se os dois vértices não forem vizinhos, ou seja, as cidades que guardam não fazem fronteira. A *Exception* lançada é capturada e lança-se um objeto da classe *CidadesNaoVizinhasException* que alerta deste erro (Checar Figuras 3 e 4).

2.1.4 Classe Grafo

Um objeto do tipo Grafo possui atributos do tipo *Set<Vertice>*, *Set<Aresta>*, *Map<Vertice, Set<Aresta>>* e uma *String* que encapsula os vértices, arestas, ligação entre vértices e arestas e o nome do mapa desse grafo. Um *Set<K>* é uma estrutura de dados que guarda valores do tipo K em seu parâmetro e não permite que haja repetições deste objeto, por este motivo é necessário o método sobrescrito *hashCode()* nas classes Vertice e Aresta. Um *Map<K,V>* é uma estrutura de dados que gere a entrada de uma chave do tipo K do parametro que redireciona para um valor V presente neste *Map.* O construtor da classe Grafo inicializa estes atributos citados. Esta classe é responsável por guardar os vértices e as arestas através dos métodos de adicionar objetos nos *Set<K>* correspondentes.

Nota-se a presença de outros métodos para ajudar na performance do método *dijkstra(Grafo,int,int)*. Tal método instancia um objeto *Dijkstra*, explanado posteriormente, que chama o método que retorna o menor caminho no grafo do vértice de index

correspondente ao primeiro parâmetro *int* do método até o vértice de index correspondente ao segundo parâmetro *int* (Checar Figura 5 e 6).

2.1.5 Classe Dijkstra

O algoritmo de Dijkstra é aquele que pode-se calcular a distância e o menor percurso de um vértice a outro de um grafo. Na classe *Dijkstra*, o construtor inicializa os objetos *List<Vertice> menorCaminho* e *List<Vertice>naoVisitados* que são usados para representar e guardar o menor caminho percorrido e a lista de vértices não visitados pelo algoritmo, respectivamente. Possui também como atributo objetos Vertice usados no algoritmo de menor trajeto. Esta classe auxiliar dá suporte à classe Grafo ao fazer o cálculo citado e retorná-lo. Caso não seja concluído o cálculo, lança-se a exceção *DijkstraCorrompidoException* que informa do ocorrido (Checar Figura 6).

2.1.7 Classe Veiculo

Para poder ter embasamento e checar se é rentável o percurso desejado, calcula-se o quanto gastaria teoricamente. Tais contas são realizadas na classe Veiculo que simboliza um transporte da empresa. Possui como atributos uma identificação *int idVeiculo* e um valor *double custo* que guarda quanto se gastará com esta viagem. O construtor da classe recebe a identificação de qual veículo faz tal trajeto, o quanto deve-se percorrer no caminho e o preço do combustível que foi inserido no transporte. Com isso, o custo do trajeto é a multiplicação da quilometragem da viagem e o valor gasto no combustível (Checar Figura 7).

2.1.8 Classe Controlador

Como foi pedido as informações a serem usadas para as cidades deveriam ser lidas de um arquivo .txt e guardadas em um grafo. A classe Controlador detém esta responsabilidade. Possui um objeto do tipo *File* e outro do tipo *Grafo* como atributos que são inicializados no construtor - que recebe de parâmetro uma string representando o local de onde está os arquivos .txt usados - da classe. Há um método chamado *geraGrafo()* que retorna o grafo já com os vértices e arestas preenchidos. Neste método cria-se um objeto que é capaz de manipular *streams* e utiliza isto para ler os .txt.

Enquanto não estiver no fim do arquivo que possui as informações das cidades, o algoritmo instancia uma cidade com os atributos necessários e adiciona a um vértice do grafo. Ao terminar, faz o mesmo com o arquivo com os termos das adjacências das cidades e cria a aresta e adiciona ao grafo com seu peso definido pelo arquivo .txt. Ao final, fecha o objeto manipulador de *Streams* e retorna o grafo completo.

Caso capture-se uma exceção de entrada e saída de *stream*, é lançada a exceção *FalhaGeracaoInformacoesException*, caso qualquer outra exceção seja lançada, lança-se a exceção *GrafoIncompletoException* que alerta que não foi possível instanciar completamente o grafo (Analisar Figura 8 e 9).

2.1.9 Classe PainelGrafo

Esta classe é já parte da interface gráfica do software. Ela estende da classe *JPanel*, utilizando dos conceitos de herança e sobrescreve - polimorfismo característico da linguagem Java - o método *paintComponent(Graphics)*. Neste método, as arestas e vértices de um grafo são postos em vetores do tipo *Object[]* (Quando necessário, há o *casting* para manipular objetos do mesmo tipo) e, a partir do grafo gerado pela leitura dos arquivos .txt através da classe Controlador, os vértices e arestas são pintados no *JPanel* da interface gráfica. Para pintá-los usa-se o objeto do tipo *Graphics* - recebido pelo parâmetro - que é convertido em um objeto do tipo *Graphics2D* (Checar Figura 10).

2.1.10 Classe JForm Rotacionamento

A IDE NetBeans permite criar interfaces gráficas através de um plugin nativo do programa. E, para compor a interface do software utilizou-se este plugin. Através deste plugin pode-se acessar os componentes do pacote do *Java Swing* e *AWT* que compreendem ao botões, caixas de texto, containers e etc, usados para montar uma interface gráfica moderna e estilizada. Este plugin possibilita a manipulação de objetos na interface e criação de eventos. Esta classe tem como atributos um objeto *PainelGrafo* que desenha o grafo do mapa na interface gráfica e uma *ArrayList* que guarda quais cidades deseja-se fazer o cálculo de menor caminho. No construtor inicializa-se seus componentes e é criada a interface gráfica (Checar Figura 11 e 12).

3. Resultados e discussões

Os resultados após a conclusão do código e do diagrama de classes foi proveitoso em sua essência. Foi possível revisar os conhecimentos adquiridos no semestre e no problema anterior sobre listas encadeadas, polimorfismo, encapsulamento, estrutura de dados, Javadoc e etc. Além disso, novos estudos renderam novos saberes sobre estruturas de dados mais complexas, *Java Collections*, interface gráfica, programação à eventos, *Java Swing e AWT*, *streams*, leitura de arquivo, grafos, modelagem de testes de unidade dentre muitos outros tópicos sobre programação em Java.

Este relatório explicita as etapas que foram utilizadas para o funcionamento do programa desenvolvido. E, partindo desses procedimentos, pode-se visualizar que o software cumpre com as exigências estabelecidas pela empresa com relação à parte de *back-end* do código, como os tipos abstrato de dados Grafo, Vertice, Aresta, manipulação de alguns componentes de interface gráfica, lançamento de exceções e vários outros tópicos. Entretanto, o programa não é capaz de calcular o menor caminho e a interação com a interface não está completamente funcional.

As sessões tutoriais, mesmo tendo o recesso e sendo poucas, foram indispensáveis para a implementação do código pois dúvidas que somente surgem na prática foram sanadas. Várias ideias, sugestões, metas e questionamentos surgiram, fato que contribuiu para a melhoria de todos os estudantes e apesar da não totalidade do software, o problema PBL foi realmente interessante.

4. Conclusão

Ao finalizar este relatório, conclui-se que o foi possível compreender e entender como se utiliza listas encadeadas, polimorfismo, herança, encapsulamento, Javadoc, estrutura de dados, criação de testes de unidade, criação de exceções, programação orientada a eventos e interface gráfica, Java *Collections* e muitos outros tópicos já vistos e que foram relembrados.

É interessante ressaltar que, mesmo com dificuldades iniciais para a compreensão do problema, o desenvolvimento e evolução do programa foram ocorrendo simultaneamente ao ganho de entendimento nas sessões e nas aulas teóricas.

O acesso ao código fonte do programa, ao diagrama de classes e este relatório são possíveis através do link do Drive a seguir: https://drive.google.com/drive/folders/19Ngg0Kc1nol9y6tzYGaVAeshjHW2CDOc?usp=sharing

5.Anexo

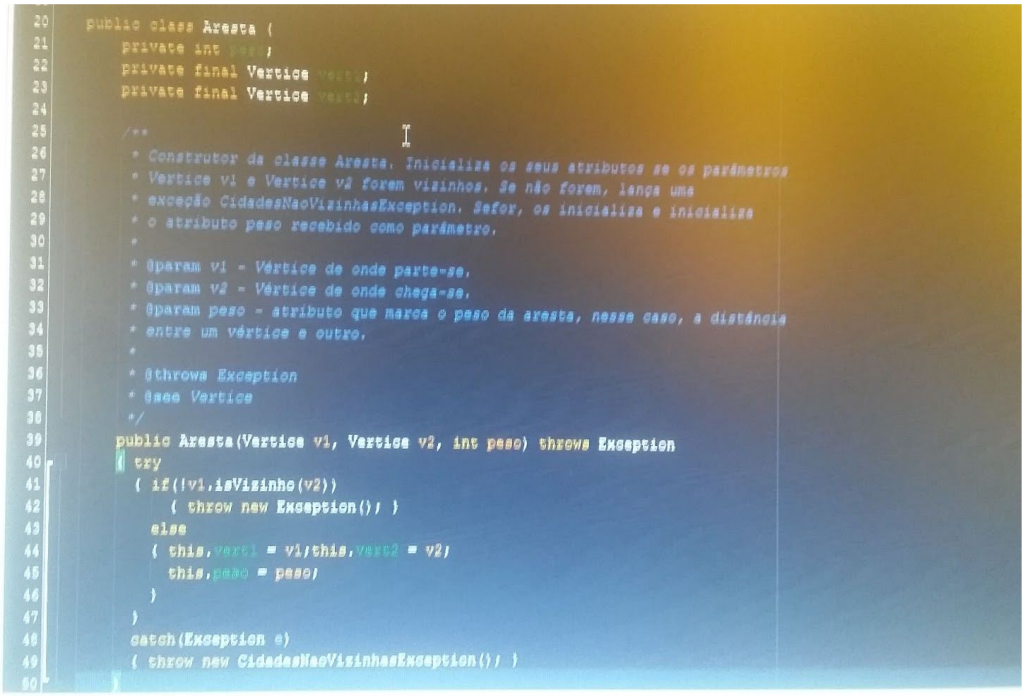
Figura 1. Classe Cidade

```
19
20  public class Cidade {
21      private final String nome;
22      private final int posX;
23      private final int posY;
24      private final int index;
25      private ArrayList listaVizinhos;
26
27      /**
28       * Construtor da classe Cidade. Recebe como parâmetro a String nome da cidade,
29       * as coordenadas e o indice de identificação da cidade. Inicializa seus
30       * atributos respectivos.
31       * @param nomeCidade - String que será o nome da cidade.
32       * @param x - int posição X que é a coordenada X num plano cartesiano.
33       * @param y - int posição Y que é a coordenada Y num plano cartesiano.
34       * @param index - indice de identificação do objeto Cidade
35       */
36      public Cidade(String nomeCidade, int x, int y, int index)
37      {   this.nome = nomeCidade; this.listaVizinhos = new ArrayList();
38          this.posY = y; this.posX = x;this.index = index;
39      }
40
41      /**
42       * Método que retorna o atributo nome desta classe.
43       * @return - retorna o atributo nome desta classe.
```

Figura 2. Classe Vertice

```
16
17  import java.util.Objects;
18
19  public class Vertice{
20      private final Cidade city;
21      private int custo;
22      private boolean visitado;
23      private Vertice anterior;
24
25      /**
26       * Construtor da classe Vertice. Instancia um objeto Vertice e
27       * inicializa seus atributos.
28       *
29       * @param cid - Cidade que seu atributo se tornará.
30       * @see Cidade
31       */
32      public Vertice(Cidade cid)
33      {   this.city = cid;
34          custo = Integer.MAX_VALUE; visitado = false;
35      }
36
37      /**
38       * Método que retorna o atributo city desta classe.
39       * @return - retorna o atributo city desta classe.
```

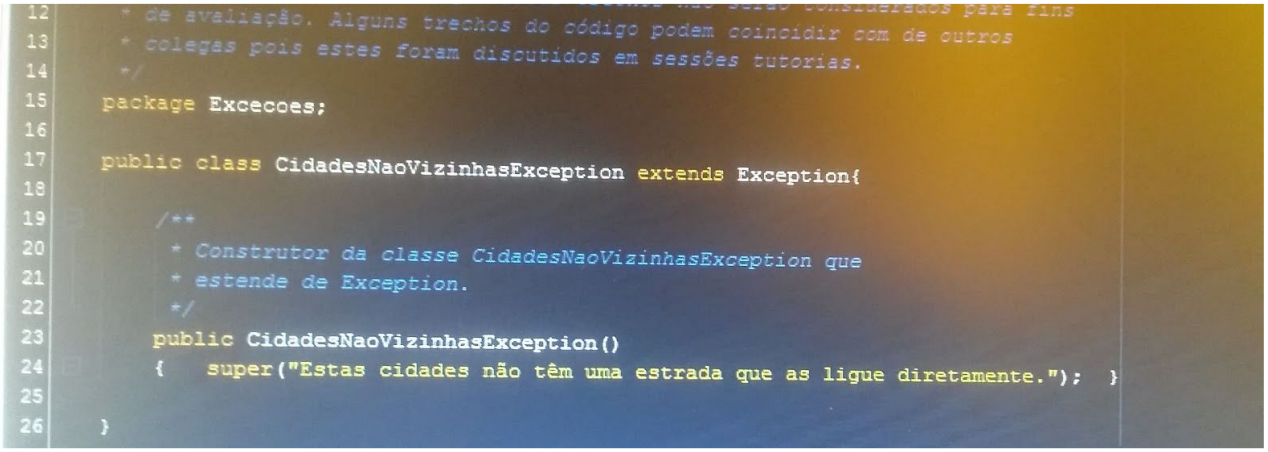
Figura 3. Classe Aresta



```
public class Aresta {
    private int peso;
    private final Vertice vert1;
    private final Vertice vert2;

    /**
     * Construtor da classe Aresta. Inicializa os seus atributos se os parâmetros
     * Vertice v1 e Vertice v2 forem vizinhos. Se não forem, lança uma
     * exceção CidadesNaoVizinhasException. Sefor, os inicializa e inicializa
     * o atributo peso recebido como parâmetro.
     *
     * @param v1 - Vértice de onde parte-se.
     * @param v2 - Vértice de onde chega-se.
     * @param peso - atributo que marca o peso da aresta, nesse caso, a distância
     * entre um vértice e outro.
     *
     * @throws Exception
     * @see Vertice
     */
    public Aresta(Vertice v1, Vertice v2, int peso) throws Exception
    { try
      { if(!v1.isVizinho(v2))
          { throw new Exception(); }
        else
        { this.vert1 = v1;this.vert2 = v2;
          this.peso = peso;
        }
      }
      catch(Exception e)
      { throw new CidadesNaoVizinhasException(); }
    }
```

Figura 4. Classe CidadesNaoVizinhasException



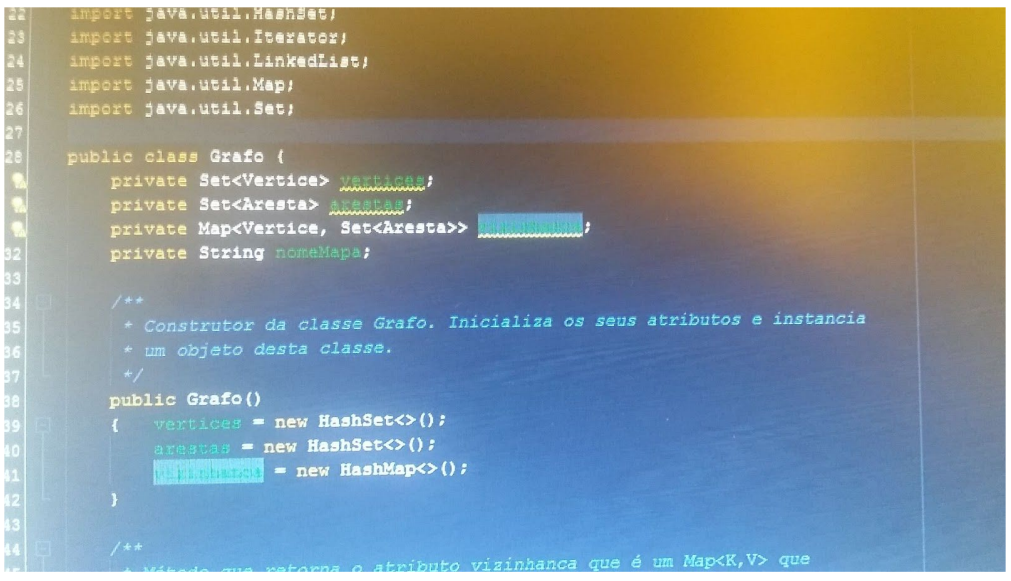
```
 * de avaliação. Alguns trechos do código podem coincidir com de outros
 * colegas pois estes foram discutidos em sessões tutorias.
 */
package Excecoes;

public class CidadesNaoVizinhasException extends Exception{

    /**
     * Construtor da classe CidadesNaoVizinhasException que
     * estende de Exception.
     */
    public CidadesNaoVizinhasException()
    {   super("Estas cidades não têm uma estrada que as ligue diretamente."); }

}
```

Figura 5. Classe Grafo



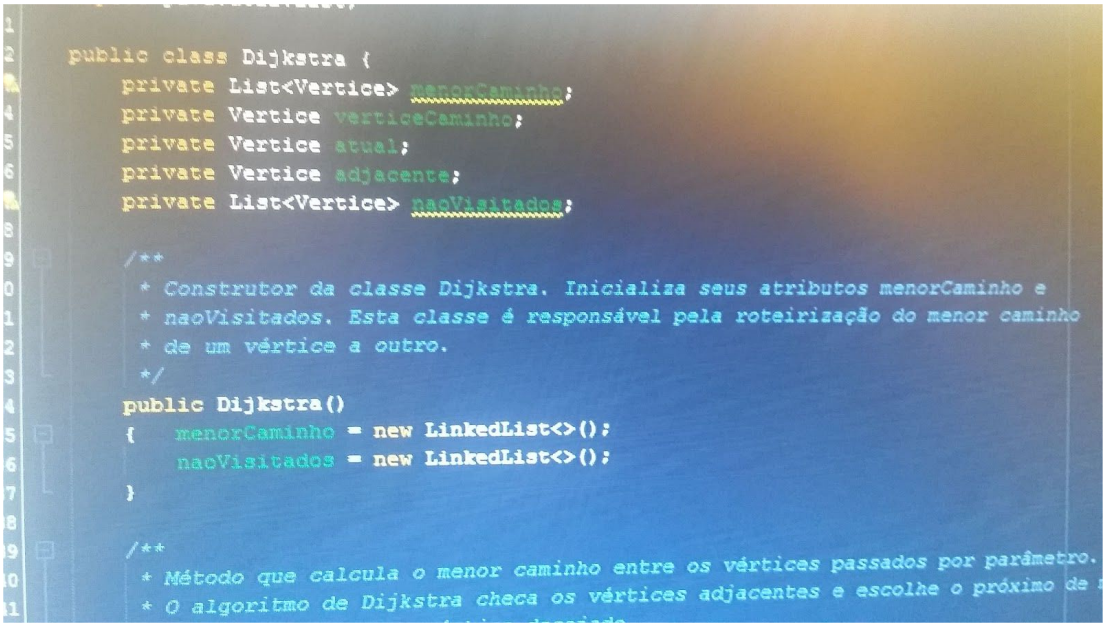
```
import java.util.HashSet;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.Map;
import java.util.Set;

public class Grafo {
    private Set<Vertice> vertices;
    private Set<Aresta> arestas;
    private Map<Vertice, Set<Aresta>> vizinhanca;
    private String nomeMapa;

    /**
     * Construtor da classe Grafo. Inicializa os seus atributos e instancia
     * um objeto desta classe.
     */
    public Grafo()
    {   vertices = new HashSet<>();
        arestas = new HashSet<>();
        vizinhanca = new HashMap<>();
    }

    /**
     * Método que retorna o atributo vizinhanca que é um Map<K,V> que
```

Figura 6. Classe Dijkstra

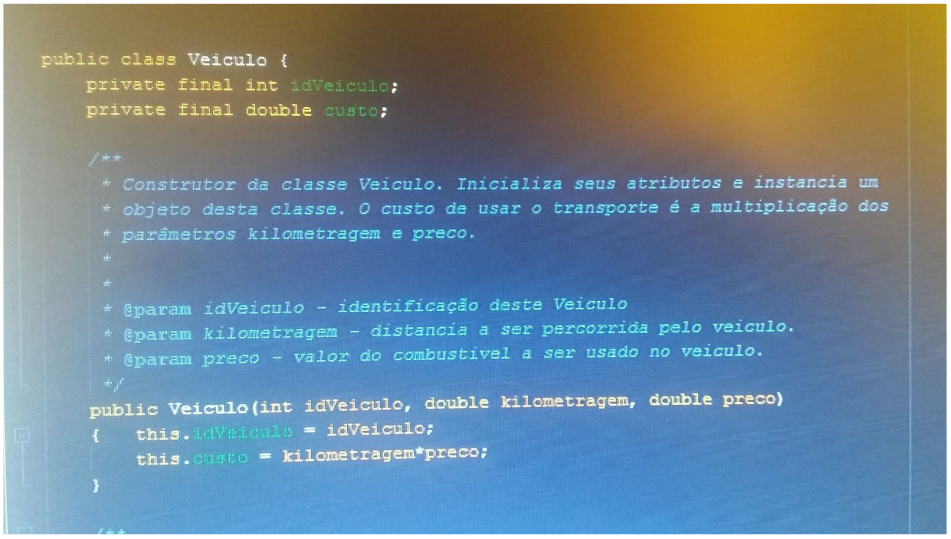


```
public class Dijkstra {
    private List<Vertice> menorCaminho;
    private Vertice verticeCaminho;
    private Vertice atual;
    private Vertice adjacente;
    private List<Vertice> naoVisitados;

    /**
     * Construtor da classe Dijkstra. Inicializa seus atributos menorCaminho e
     * naoVisitados. Esta classe é responsável pela roteirização do menor caminho
     * de um vértice a outro.
     */
    public Dijkstra()
    {   menorCaminho = new LinkedList<>();
        naoVisitados = new LinkedList<>();
    }

    /**
     * Método que calcula o menor caminho entre os vértices passados por parâmetro.
     * O algoritmo de Dijkstra checa os vértices adjacentes e escolhe o próximo de
```

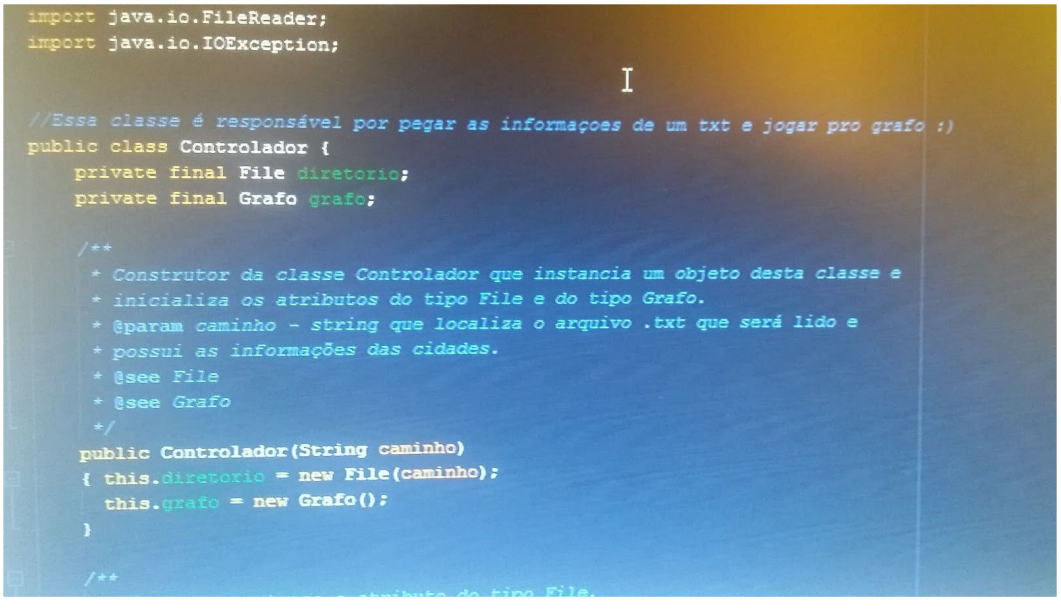
Figura 7. Classe Veiculos



```
public class Veiculo {
    private final int idVeiculo;
    private final double custo;

    /**
     * Construtor da classe Veiculo. Inicializa seus atributos e instancia um
     * objeto desta classe. O custo de usar o transporte é a multiplicação dos
     * parâmetros kilometragem e preco.
     *
     *
     * @param idVeiculo - identificação deste Veiculo
     * @param kilometragem - distancia a ser percorrida pelo veiculo.
     * @param preco - valor do combustível a ser usado no veiculo.
     */
    public Veiculo(int idVeiculo, double kilometragem, double preco)
    {   this.idVeiculo = idVeiculo;
        this.custo = kilometragem*preco;
    }
```

Figura 8. Classe Controlador

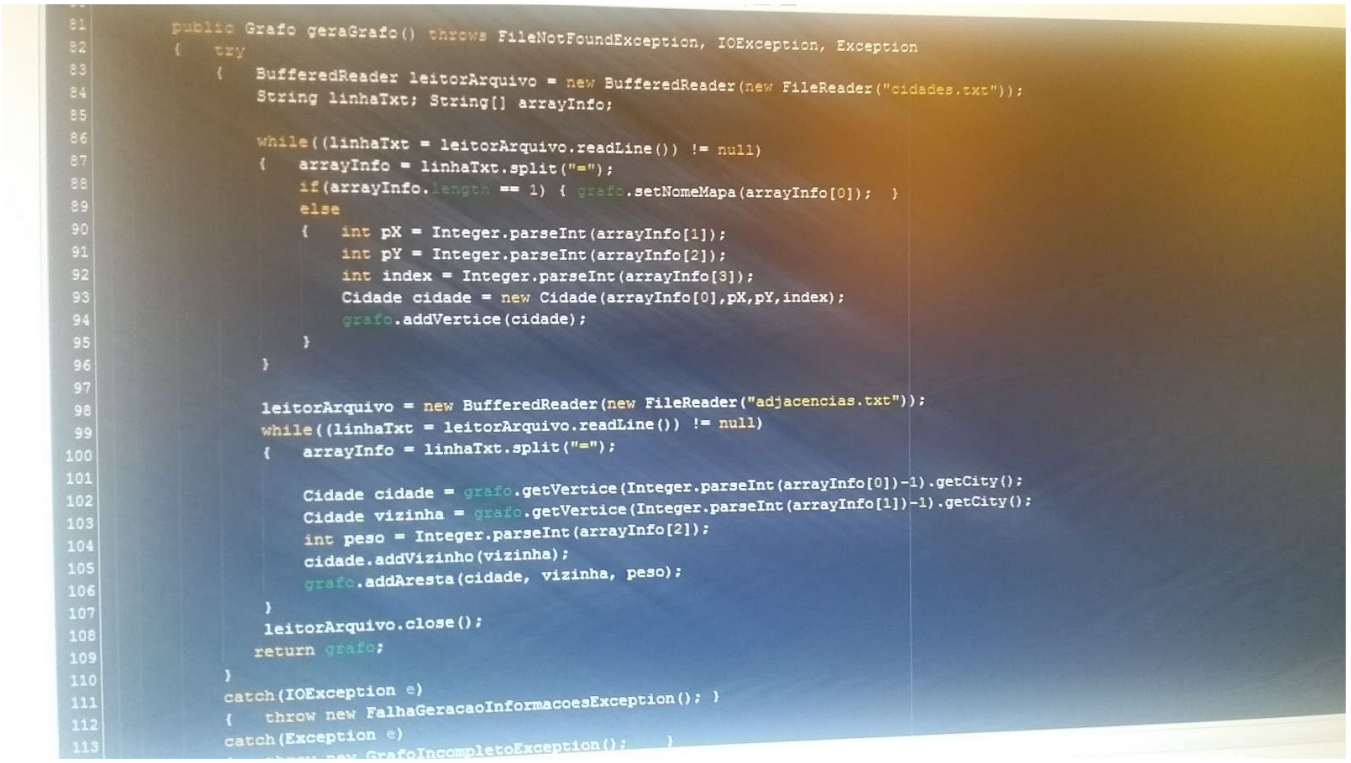


```
import java.io.FileReader;
import java.io.IOException;

//Essa classe é responsável por pegar as informaçoes de um txt e jogar pro grafo :)
public class Controlador {
    private final File diretorio;
    private final Grafo grafo;

    /**
     * Construtor da classe Controlador que instancia um objeto desta classe e
     * inicializa os atributos do tipo File e do tipo Grafo.
     * @param caminho - string que localiza o arquivo .txt que será lido e
     * possui as informações das cidades.
     * @see File
     * @see Grafo
     */
    public Controlador(String caminho)
    { this.diretorio = new File(caminho);
      this.grafo = new Grafo();
    }
```

Figura 9. Método geraGrafo() da classe Controlador



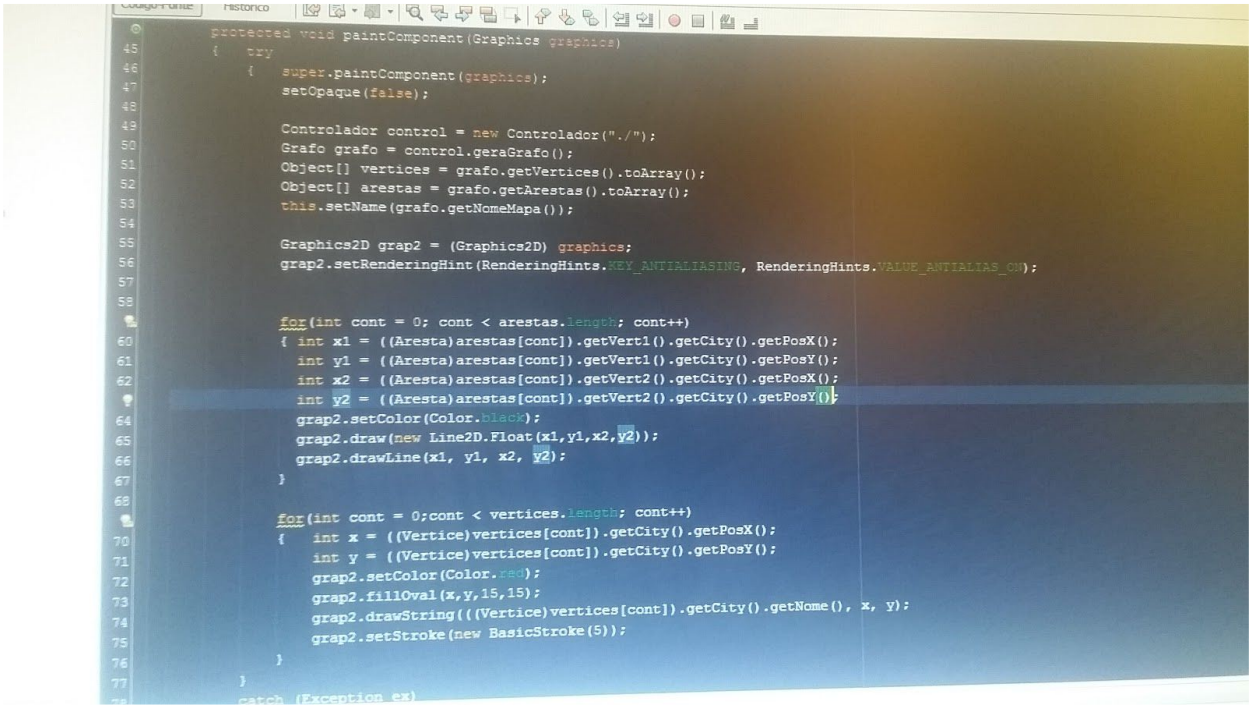
```
public Grafo geraGrafo() throws FileNotFoundException, IOException, Exception
{   try
    {   BufferedReader leitorArquivo = new BufferedReader(new FileReader("cidades.txt"));
        String linhaTxt; String[] arrayInfo;

        while((linhaTxt = leitorArquivo.readLine()) != null)
        {   arrayInfo = linhaTxt.split("=");
            if(arrayInfo.length == 1) { grafo.setNomeMapa(arrayInfo[0]);  }
            else
            {   int pX = Integer.parseInt(arrayInfo[1]);
                int pY = Integer.parseInt(arrayInfo[2]);
                int index = Integer.parseInt(arrayInfo[3]);
                Cidade cidade = new Cidade(arrayInfo[0],pX,pY,index);
                grafo.addVertice(cidade);
            }
        }

        leitorArquivo = new BufferedReader(new FileReader("adjacencias.txt"));
        while((linhaTxt = leitorArquivo.readLine()) != null)
        {   arrayInfo = linhaTxt.split("=");

            Cidade cidade = grafo.getVertice(Integer.parseInt(arrayInfo[0])-1).getCity();
            Cidade vizinha = grafo.getVertice(Integer.parseInt(arrayInfo[1])-1).getCity();
            int peso = Integer.parseInt(arrayInfo[2]);
            cidade.addVizinho(vizinha);
            grafo.addAresta(cidade, vizinha, peso);
        }
        leitorArquivo.close();
       return grafo;
    }
    catch(IOException e)
    {   throw new FalhaGeracaoInformacoesException(); }
    catch(Exception e)
```

Figura 10. Método sobrescrito da classe PainelGrafo



```
protected void paintComponent(Graphics graphics)
{   try
    {   super.paintComponent(graphics);
        setOpaque(false);

        Controlador control = new Controlador("./");
        Grafo grafo = control.geraGrafo();
        Object[] vertices = grafo.getVertices().toArray();
        Object[] arestas = grafo.getArestas().toArray();
        this.setName(grafo.getNomeMapa());

        Graphics2D grap2 = (Graphics2D) graphics;
        grap2.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);


        for(int cont = 0; cont < arestas.length; cont++)
        { int x1 = ((Aresta)arestas[cont]).getVert1().getCity().getPosX();
          int y1 = ((Aresta)arestas[cont]).getVert1().getCity().getPosY();
          int x2 = ((Aresta)arestas[cont]).getVert2().getCity().getPosX();
          int y2 = ((Aresta)arestas[cont]).getVert2().getCity().getPosY();
          grap2.setColor(Color.black);
          grap2.draw(new Line2D.Float(x1,y1,x2,y2));
          grap2.drawLine(x1, y1, x2, y2);
        }

        for(int cont = 0;cont < vertices.length; cont++)
        {   int x = ((Vertice)vertices[cont]).getCity().getPosX();
            int y = ((Vertice)vertices[cont]).getCity().getPosY();
            grap2.setColor(Color.red);
            grap2.fillOval(x,y,15,15);
            grap2.drawString(((Vertice)vertices[cont]).getCity().getNome(), x, y);
            grap2.setStroke(new BasicStroke(5));
        }
    }
    catch (Exception ex)
```

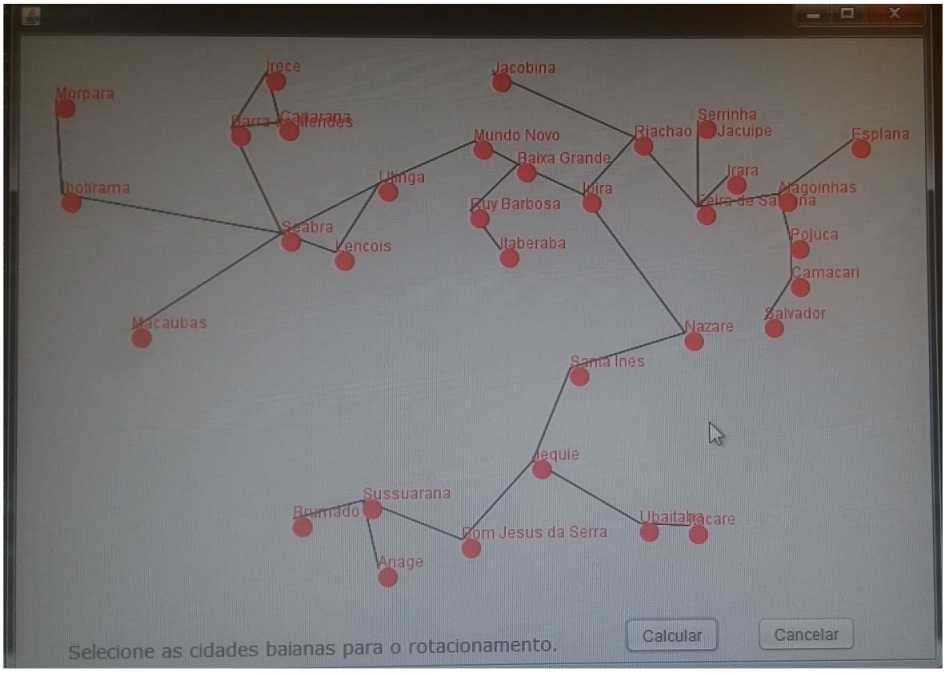
Figura 11. Classe Rotacionamento

```
public class Rotacionamento extends javax.swing.JFrame {
    private PainelGrafo painel;
    private ArrayList listaCidades;

    /**
     * Construtor da classe Rotacionamento que estende de JFrame e é a interface
     * do programa. Instancia e inicializa seus atributos.
     */
    public Rotacionamento()
    {   initComponents();
        this.painel.setLayout(new BorderLayout());
    }

    /**
     * This method is called from within the constructor to initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is always
     * regenerated by the Form Editor.
     */
    @SuppressWarnings("unchecked")
    Generated Code
```

Figura 12. Interface Gráfica do software



6.Referências bibliográficas

http://sites.ecomp.uefs.br/joao/home/courses/exa806/aulas

http://sites.ecomp.uefs.br/joao/home/courses/exa805/aulas

http://www2.dcc.ufmg.br/livros/algoritmos-java/cap7/transp/completo4/cap7.pdf

https://github.com/franzejr/Dijkstra-Algorithm-Java-GUI

https://www.baeldung.com/java-dijkstra

https://gist.github.com/roooodcastro/6325153/revisions