Docentes
Teórico-prática: Patrícia Macedo
Laboratório: Luís Damas

Alunos (Turma EI-BS)
Alexandre Antunes de Oliveira 202002394
Daniel Cruz Moreira 202001678
Henrique Palma Carvalho 202000390
Rafael Cruz Cardoso 202001529

Relatório - Rede Logística

Programação Avançada

Ano letivo 2021-2022

IPS Instituto Politécnico de Setúbal
Escola Superior de Tecnologia de Setúbal

CONTEÚDO

ÍNDICE DE FIGURAS

TIPOS ABSTRATOS DE DADOS

Graph<V, E>

O TAD Graph foi utilizado para a implementação da classe GraphAdjacencyList (implementação requerida para este projeto).

Vertex<V> e Edge<E, V>

Diretamente relacionados ao Graph, temos os ADT's Vertex e Edge, os quais armazenam elementos genéricos. No nosso caso, o elemento que o vértice armazena é o Hub, enquanto que o elemento que a Edge armazena é a Route.

Map<K, V>

Este ADT foi utilizado dentro da implementação de GraphAdjacencyList, pois esta implementação é composta por um atributo que guarda todos os vértices existentes num mapa. Também foi utilizado para a realização do método showCentrality(), o qual devolve um mapa de todos os Hubs existentes e o número correspondente de vizinhos (3.1.2 – cálculo de métricas).

Stack<E>

Para a realização da ação de Undo no nosso projeto, utilizámos o ADT Stack, que tem como objetivo guardar uma coleção de Actions. Uma Action – como o próprio nome diz – define uma das 4 ações que alteram a ordem/dimensão do grafo: adicionar um Hub, remover um Hub, adicionar uma Route e remover uma Route. Como estas ações precisam de ser feitas da mais recente para a mais antiga, utilizámos a Stack (em vez do ADT Queue), pois contém uma política LIFO.

List<E>

Utilizámos este ADT de forma geral em grande parte das funcionalidades do projeto, pois a grande maioria das funções retorna uma lista de Hubs (List<Hub>), como é o exemplo do caminho mais curto entre dois Hubs (3.1.5), os dois Hubs mais distantes um do outro (3.1.6), ou mesmo o conjunto de Hubs que distam N rotas de um Hub H (3.1.7).

INTERFACE GRÁFICA

A interface obtida no final destas três fases de desenvolvimento foi além do esperado, pois não só conseguimos replicar o que desenhámos no Mockup, como também conseguimos melhorar alguns aspetos, como é o exemplo da imagem de fundo da aplicação e algumas ações extra.

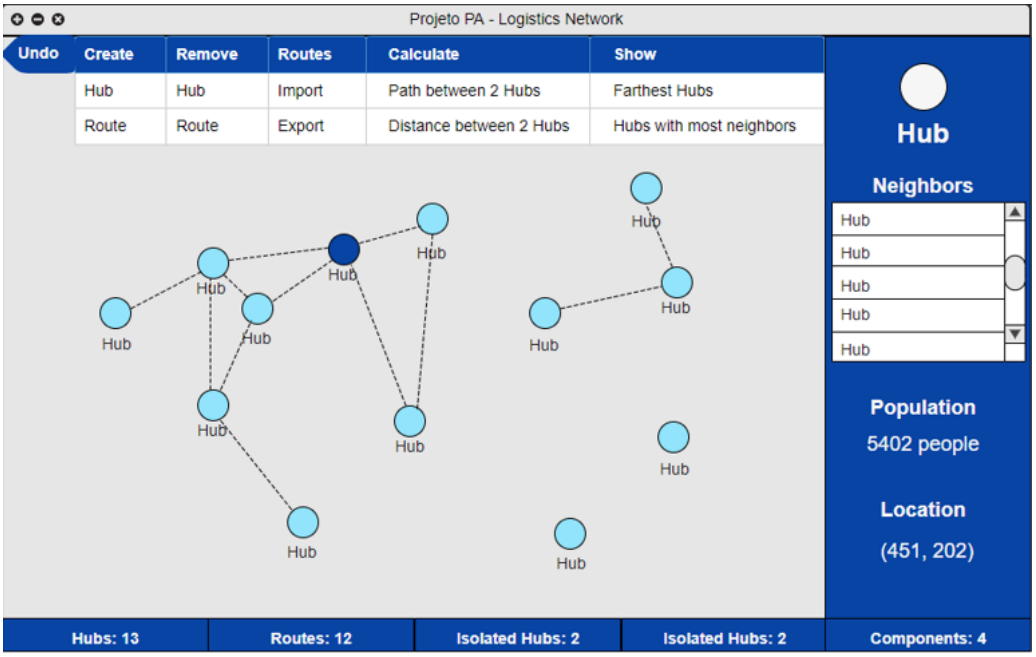

Figura 1 - Mockup de baixa/média fidelidade

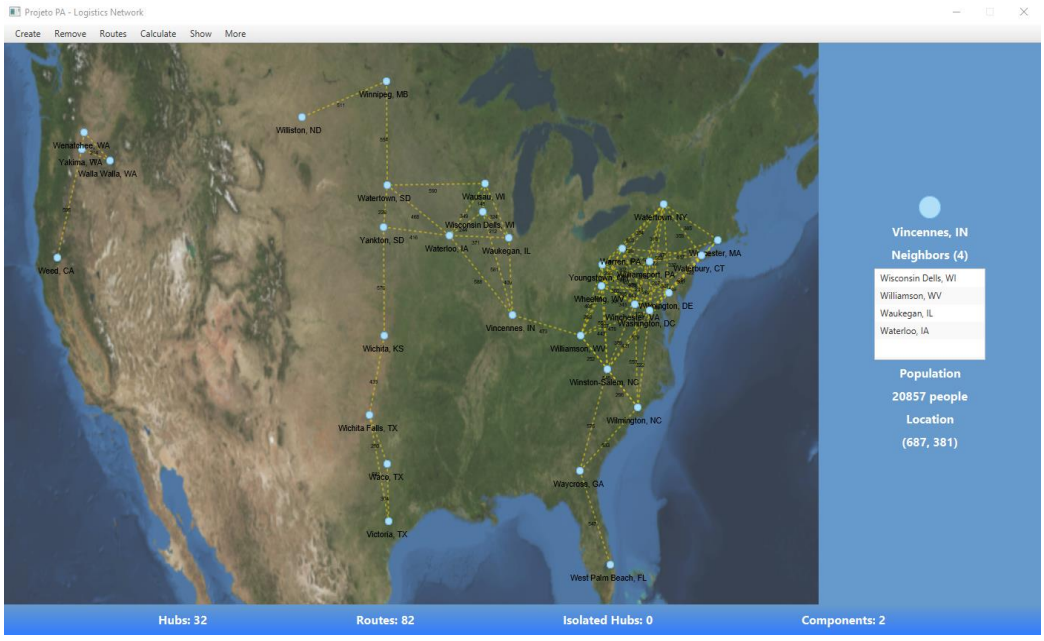

Figura 2 - Interface gráfica final

DIAGRAMA DE CLASSES

De seguida, apresenta-se um diagrama de classes UML, onde é possível perceber qual a relação das diversas classes criadas:

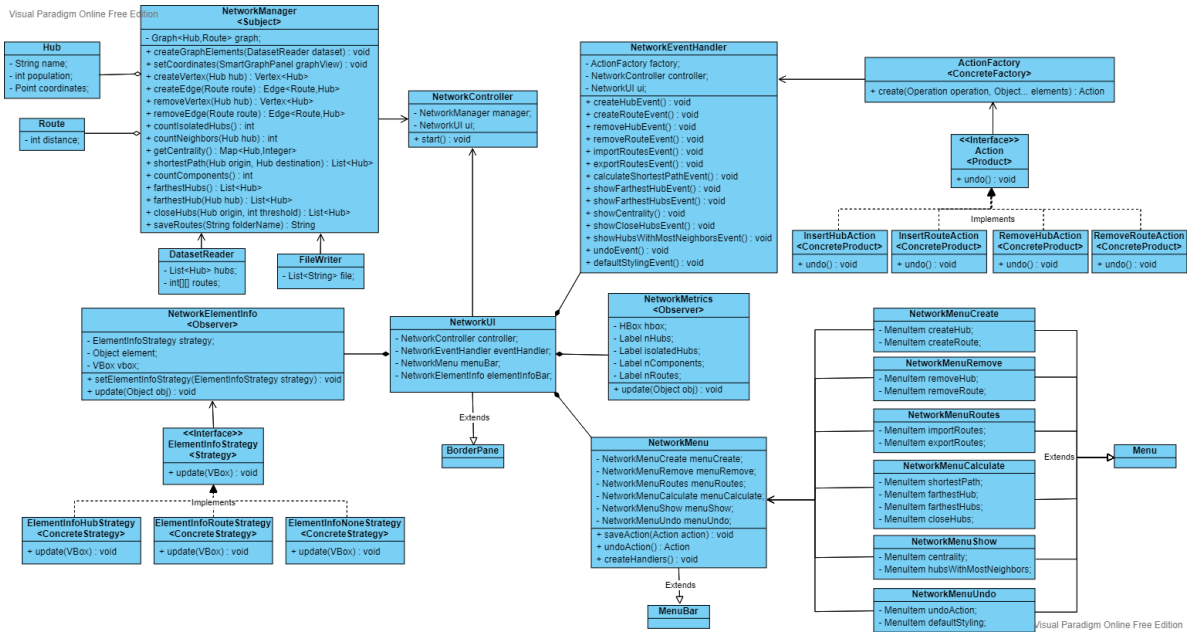

Figura 3 - Diagrama de classes UML

PADRÕES DE SOFTWARE

Observer

"Pretende-se assegurar que, quando um objeto muda de estado, um número de objetos dependentes é notificado automaticamente."

Por outras palavras, quando ocorre uma ação que altera a ordem ou a dimensão do grafo, é preciso "avisar" todos os outros componentes do programa, de forma a que estes tenham sempre a informação atualizada.

Na nossa aplicação, a classe responsável por alterar o grafo – NetworkManager – é o Subject, ou seja, a entidade que "avisa" todos os outros componentes sempre que há alguma alteração.

Por outro lado, a classe NetworkMetrics (barra inferior que apresenta as métricas) e a classe NetworkElementInfo (entidade que apresenta informação do elemento selecionado no lado direito do ecrã) são os Observers, pois estes precisam de ser atualizados de acordo com o que acontece.

Para exemplificar, este método na classe NetworkManager que adiciona um novo vértice ao grafo, agora passa a "notificar" os observadores de que o grafo foi alterado:

```
// Given a Hub, adds a new Vertex to the graph
public Vertex<Hub> createVertex(Hub hub) throws InvalidVertexException {
    Vertex<Hub> vertex = graph.insertVertex(hub);
    hubs.add(hub);
    notifyObservers( obj: this);
    return vertex;
}
```

Figura 4 - Subject

No lado oposto, na classe NetworkMetrics (métricas da aplicação), é feita uma atualização aos dados de forma imediata:

```
@Override
public void update(Object obj) {
    NetworkManager manager = (NetworkManager)obj;
    hbox.getChildren().clear();

    nHubs.setText("Hubs: " + manager.countHubs());
    nRoutes.setText("Routes: " + manager.countRoutes());
    isolatedHubs.setText("Isolated Hubs: " + manager.countIsolatedHubs());
    nComponents.setText("Components: " + manager.countComponents());

    hbox.getChildren().addAll(nHubs, nRoutes, isolatedHubs, nComponents);
}
```

Figura 5 - Observer

Strategy

"O padrão Strategy sugere pegar numa classe que faz uma tarefa específica de muitas formas diferentes e extrair estes ''algoritmos'' para classes separadas chamadas de estratégias."

Este padrão foi utilizado para que a informação mostrada na parte direita do ecrã - informação do elemento selecionado - fosse atualizada.

Cada vez que um vértice ou aresta (ou nada) é selecionado, a estratégia presente na classe NetworkElementInfo é alterada (ou seja, a implementação de Strategy é alterada) e o método que atualiza a informação é executado. Desta forma, conseguimos separar de forma bastante eficaz as três informações que podem ser exibidas ao utilizador: um Hub, uma Route ou nada.

Exemplificando, a classe NetworkElementInfo funciona como o ator "Context", pois mantém uma referência para a estratégia utilizada:

```
public class NetworkElementInfo implements Observer {

    // Strategy for the element details' creation (none, hub or route)
    ElementInfoStrategy elementInfoStrategy;
```

Figura 6 - Context

Quando se clica, por exemplo, num Hub, esta estratégia é alterada e o método que atualiza a informação apresentada é executado:

```
controller.getGraphView().setVertexDoubleClickAction((SmartGraphVertex<Hub> graphVertex) -> {
    ui.getElementInfoBar().setElementInfoStrategy(new ElementInfoHubStrategy());
    ui.getElementInfoBar().setElement(graphVertex.getUnderlyingVertex().element(),controller.getManager());
});
```

Figura 7 - Client

```
public interface ElementInfoStrategy {

    void update(NetworkManager manager, VBox vbox, Object element);

}
```

Figura 8 - Strategy

```
public class ElementInfoHubStrategy implements ElementInfoStrategy {

    @Override
    public void update(NetworkManager manager, VBox vbox, Object element) {
        Hub hub = (Hub)element;
```

Figura 9 - ConcreteStrategy

Factory

"Centralizar a criação de variantes de objetos em vez de termos new e lógica condicional associada espalhados pelo código-fonte."

Na nossa aplicação, existem quatro tipos de ações: InsertHubAction (ação de inserir um Hub), InsertRouteAction (ação de inserir uma Route), RemoveHubAction (ação de remover um Hub) e RemoveRouteAction (ação de remover uma Route). Para não termos de criar uma lógica condicional extensa que faz ações diferentes dependendo da ação, decidimos criar uma "fábrica" que delega qual a instância correta de Action a ser criada, dependendo da operação (enumerado).

Ou seja, inicialmente criámos um enumerado que contém os quatro tipos de operações possíveis:

```
public enum Operation {
    INSERT_HUB,
    REMOVE_HUB,
    INSERT_ROUTE,
    REMOVE_ROUTE
}
```

Figura 10 - Enumerado de Action

Depois, criámos a fábrica que cria instâncias diferentes, dependendo do enumerado recebido:

```
public class ActionFactory {

    public Action create(Operation operation, Object... elements) {
        switch (operation) {
            case INSERT_HUB:
                return new RemoveHubAction(elements[0],elements[1]);
            case REMOVE_HUB:
                return new InsertHubAction(elements[0],elements[1],elements[2],elements[3]);
            case INSERT_ROUTE:
                return new RemoveRouteAction(elements[0],elements[1]);
            case REMOVE_ROUTE:
                return new InsertRouteAction(elements[0],elements[1],elements[2],elements[3]);
            default:
                throw new RuntimeException("Unsupported action operation!");
        }
    }
```

Figura 11 - ActionFactory

Na adição, por exemplo, de uma nova Route, esta função da fábrica é chamada e uma instância de Action é criada. Isto facilita a implementação de reverter ações (undo), pois basta chamar a função "undo()" da instância de Action criada:

```
// Evento - Fazer undo da última ação
public void undoEvent(MenuItem menuItem) {
    menuItem.setOnAction(actionEvent1 -> {
        defaultStyling();
        try {
            Action action = ui.getMenuBar().undoAction();
            if (action == null)
                throw new RuntimeException("There is no action to undo!");
            action.undo();
```

Figura 12 - Evento de Undo

Cada implementação de Action faz uma ação diferente no método "undo()".

REFACTORING

"Permite reestruturar o código interno de um programa de software sem alterar o seu comportamento externo."

Foram detetados dois tipos de bad smells no nosso programa, duplicate code e switch statements. Foram aplicados os processos de refactoring de forma tornar o código mais eficiente, de fácil manutenção e permite uma maior compreensão do código em questão.

Tal como diria Martin Fowler:

"Qualquer pessoa pode escrever código que um computador possa compreender. Bons programadores escrevem códigos que os humanos conseguem compreender."

De seguida, apresentamos uma tabela com todos os Bad Smells detetados, seguido por imagens (antes e depois) de cada um deles.

Bad Smell	Nº de vezes detetado	Técnica de Refactoring
Duplicate Code	12	Nome: Extract Method Sumário: Existe código repetido em várias funções semelhantes, na classe NetworkEventHandler Mecanismo: Mover este código para uma nova função e substituir o código antigo pela chamada desta nova função
Switch Statements	1	Nome: Replace Conditional with Polymorphism Sumário: Existe uma cadeia longa de switch case para realizar a ação de undo. Mecanismo: Criar subclasses que correspondem a cada "case". Em cada subclasse, é criado um método que realiza o que já era feito nessa ramificação. Depois disto, basta substituir o switch case pela chamada da função.
Message Chains	~50	Nome: Hide Delegate Sumário: O acesso a um objeto requer acesso a outro objeto, assim sucessivamente. Mecanismo: Em vez de se aceder a um objeto constantemente através de outros, criamos um atributo do objeto que queremos. Desta forma, não são criadas estas cadeias de acesso.
Dead Code	5	Nome: Remove Code Sumário: O código apresentado não é utilizado vez nenhuma na aplicação, nem tem impacto. Mecanismo: Remover o código.

Duplicate Code

Quando vários fragmentos de código são semelhantes.

Antes

Aqui notamos a repetida utilização do atributo *dialog*.

Figura 13 - Criação de Stage em eventos (antes)

Depois

Aqui recorremos à chamada da nova função *createStage*.

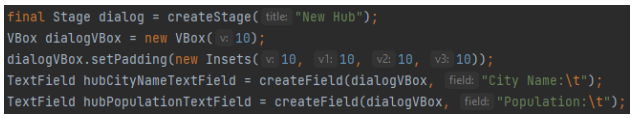

Figura 14 - Criação de Stage em eventos (depois)

Foi adicionada uma nova função *createStage* de forma a que o código duplicado seja transformado nesta nova função onde recebe o título do *stage* e faz as alterações necessárias.

Figura 15 - Função para criar Stage

Switch Statements

Quando existe um operador switch complexo, ou um esquema if encadeado.

Antes

```
switch (action.getReverseOperation()) {
    case INSERT_HUB:
        Hub insertHub = (Hub)action.element()[0];
        Map<Route,Hub> adjacentRoutes = (Map<Route,Hub>)action.element(
        List<Hub> hubs = (List<Hub>)action.element()[2];
        Vertex<Hub> vertex = controller.getManager().createVertex(inser
        for (Route route : adjacentRoutes.keySet())
            controller.getManager().createEdge(insertHub,adjacentRoutes
        controller.getGraphView().updateAndWait();
        controller.getGraphView().setVertexPosition(vertex,insertHub.ge
        break;
    case REMOVE_HUB:
        Hub removeHub = (Hub)action.element()[0];
        controller.getManager().removeVertex(removeHub);
        controller.getGraphView().updateAndWait();
        break;
    case INSERT_ROUTE:
        Hub origin = (Hub)action.element()[0];
        Hub destination = (Hub)action.element()[1];
        Route route = (Route)action.element()[2];
        controller.getManager().createEdge(origin,destination,route);
        controller.getGraphView().updateAndWait();
        break;
    case REMOVE_ROUTE:
        Edge<Route,Hub> removeEdge = (Edge)action.element()[0];
        controller.getManager().removeEdge(removeEdge.element());
        controller.getGraphView().updateAndWait();
        break;
}
```

Figura 16 - Evento de Undo (antes)

Depois

```
Action action = ui.getMenuBar().undoAction();
if (action == null)
    throw new RuntimeException("There is no action to undo!");
action.undo();
```

Figura 17 - Evento de Undo (depois)

```
@Override
public void undo() {
    Vertex<Hub> vertex = controller.getManager().createVertex(hub,hubs);
    for (Route route : adjacentRoutes.keySet())
        controller.getManager().createEdge(hub,adjacentRoutes.get(route),route);
    controller.getGraphView().updateAndWait();
    controller.getGraphView().setVertexPosition(vertex,hub.getCoordinates().getX(), y: hub.getCoordinates().getY() - 25);
}
```

Figura 18 - Classe InsertHubAction

```
@Override
public void undo() {
    controller.getManager().createEdge(origin,destination,route);
    controller.getGraphView().updateAndWait();
}
```

Figura 19 - Classe InsertRouteAction

```
@Override
public void undo() {
    controller.getManager().removeVertex(hub);
    controller.getGraphView().updateAndWait();
}
```

Figura 20 - Classe RemoveHubAction

```
@Override
public void undo() {
    controller.getManager().removeEdge(edge.element());
    controller.getGraphView().updateAndWait();
}
```

Figura 21 - Classe RemoveRouteAction

Message Chains

Quando são visíveis cadeias de chamadas de funções.

Antes

```
removeHubButton.setOnAction(actionEvent2 -> {
    try {
        defaultTextField(hubTextField);
        if (hubTextField.getText().trim().isEmpty())
            throw new IncorrectFieldException("\"Hub\" field is empty!");
        else if (controller.getManager().getHub(hubTextField.getText().trim()) == null)
            throw new IncorrectFieldException("\"Hub\" doesn't exist!");
        else {
            Hub hub = controller.getManager().getHub(hubTextField.getText().trim());

            // Save action
            Map<Route, Hub> adjacentRoutes = new HashMap<>();
            Vertex<Hub> vertex = controller.getManager().getVertex(hub);
            for (Edge edge : controller.getManager().getGraph().incidentEdges(controller.getManager().getVertex(hub)))
                adjacentRoutes.put((Route) edge.element(), (Hub) controller.getManager().getGraph().opposite(vertex, edge).element());
            Action action = new Action(Operation.REMOVE_HUB, hub, adjacentRoutes, new ArrayList<>(controller.getManager().getHubs()));
            ui.getMenuBar().saveAction(action);

            controller.getManager().removeVertex(hub);
            controller.getGraphView().updateAndWait();
            dialog.close();

            System.out.println("Hub removed!");
```

Figura 22 - Acesso ao NetworkManager (antes)

Depois

```
removeHubButton.setOnAction(actionEvent2 -> {
    try {
        defaultTextField(hubTextField);
        String hubText = hubTextField.getText().trim();
        if (hubText.isEmpty())
            throw new IncorrectFieldException("\"Hub\" field is empty!");
        else if (manager.getHub(hubText) == null)
            throw new IncorrectFieldException("\"Hub\" doesn't exist!");
        else {
            Hub hub = manager.getHub(hubText);

            // Save action
            Map<Route, Hub> adjacentRoutes = new HashMap<>();
            Vertex<Hub> vertex = manager.getVertex(hub);
            for (Edge<Route,Hub> edge : manager.getGraph().incidentEdges(manager.getVertex(hub)))
                adjacentRoutes.put(edge.element(), manager.getGraph().opposite(vertex, edge).element());
            Action action = factory.create(Operation.REMOVE_HUB, controller, hub, adjacentRoutes, new ArrayList<>(manager.getHubs()));
            ui.getMenuBar().saveAction(action);

            manager.removeVertex(hub);
            controller.getGraphView().updateAndWait();
            dialog.close();

            System.out.println("Hub removed!");
```

Figura 23 - Acesso ao NetworkManager (depois)

Dead Code

Quando existe código que não é utilizado em qualquer parte do programa.

Antes

```
public ElementInfoStrategy getElementInfoStrategy() {
    return elementInfoStrategy;
}
```

Figura 24 - Dead Code na classe NetworkElementInfo

```
public void disableElementsInfoEvent() {
    controller.getGraphView().setOnMousePressed(null);
}
```

Figura 25 - Dead Code na classe NetworkEventHandler

```
private void removeHandlers() {
    networkUI.getEventHandler().removeHandler(menuCreate.getCreateHubItem());
    networkUI.getEventHandler().removeHandler(menuCreate.getCreateRouteItem());
    networkUI.getEventHandler().removeHandler(menuRemove.getRemoveHubItem());
    networkUI.getEventHandler().removeHandler(menuRemove.getRemoveRouteItem());
    networkUI.getEventHandler().removeHandler(menuRoutes.getImportRoutesItem());
    networkUI.getEventHandler().removeHandler(menuRoutes.getExportRoutesItem());
    networkUI.getEventHandler().removeHandler(menuCalculate.getShortestPathItem());
    networkUI.getEventHandler().removeHandler(menuCalculate.getFarthestHubItem());
    networkUI.getEventHandler().removeHandler(menuCalculate.getFarthestHubsItem());
    networkUI.getEventHandler().removeHandler(menuCalculate.getCloseHubsItem());
    //networkUI.getEventHandler().removeHandler(menuCalculate.getDistancePathItem());
    networkUI.getEventHandler().removeHandler(menuShow.getCentralityItem());
    networkUI.getEventHandler().removeHandler(menuShow.getHubsWithMostNeighborsItem());
    networkUI.getEventHandler().removeHandler(menuUndo.getUndoActionItem());
    networkUI.getEventHandler().removeHandler(menuUndo.getDefaultStylingItem());
}
```

Figura 26 - Dead Code na classe NetworkMenu

Depois

Estes códigos foram removidos, e a aplicação foi testada para confirmar que não houve impacto nestas ações.