



Desenvolvimento de Sistemas

Validação e depuração de *software*: *debug*, detecção e correção de erros

Debug

Para entender o que é um *debug*, inicialmente é necessário entender o que é um *bug*, pois são palavras de uma linguagem que não é usual para quem não é deste segmento de tecnologia. É uma palavra criada a partir de uma situação muito antiga e que talvez você desconheça.

Os *bugs* nada mais são do que erros em programas de computadores.

Nas unidades curriculares anteriores, você deve ter estudado alguns deles, mas, por que eles são chamados de *bugs* no meio computacional?



Figura 1 – *Bug* ou “inseto”

Fonte: Wikipedia (2022)

Pode parecer estranho, mas o termo em inglês *bug* significa “inseto”. Então, qual será a relação entre insetos e os erros nos códigos?

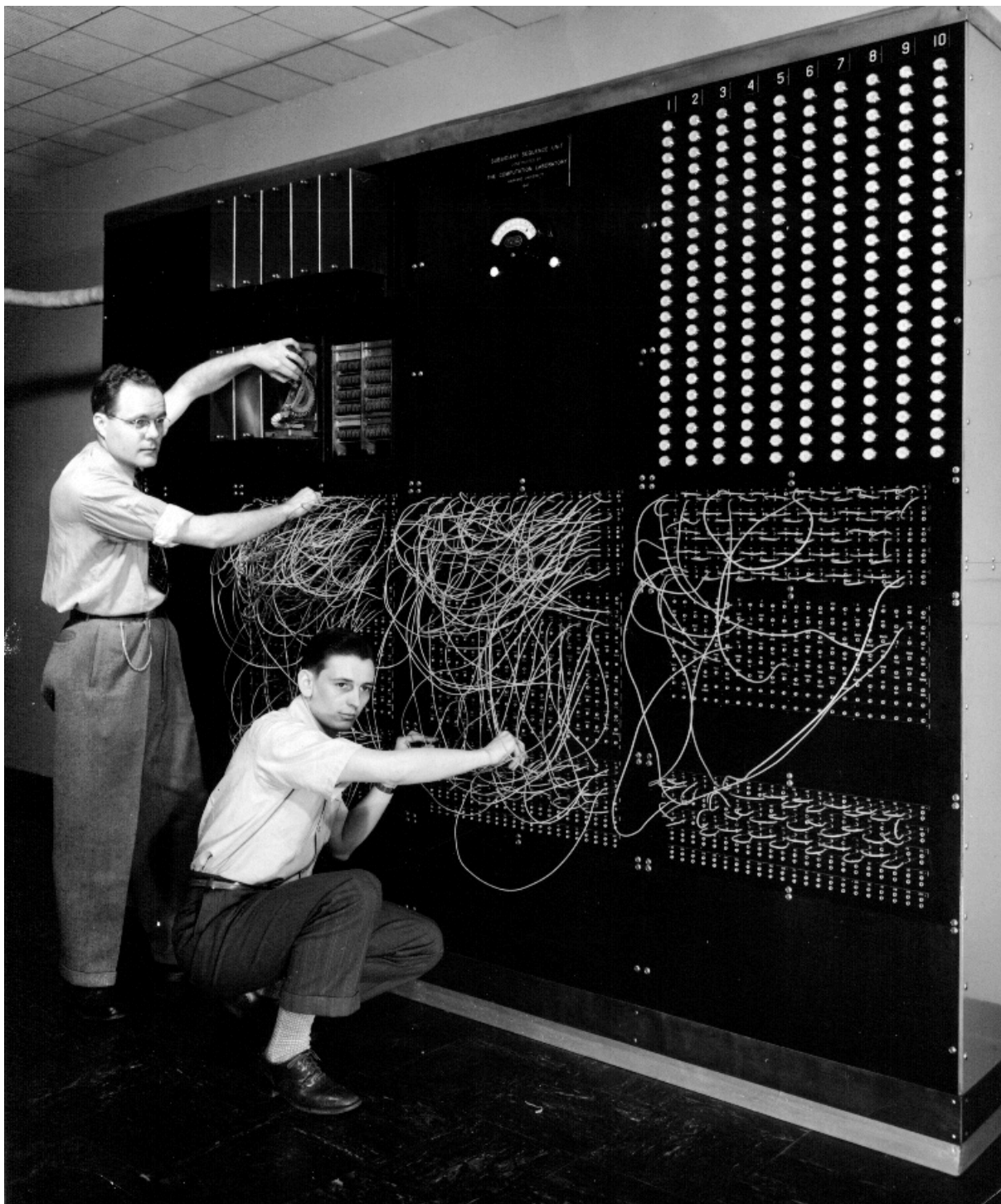


Figura 2 – Operação de computadores antigos

Fonte: Máximo *et al.* (s. d.)

Basicamente, os primeiros computadores eram muito grandes e era inevitável que alguns insetos conseguissem entrar neles. Certo dia, um desses grandes computadores parou de funcionar. Buscando solucionar o erro, uma engenheira

descobriu que o problema eram os insetos, que, depois de entrarem no computador, morriam no interior do aparelho, causando falhas em sua inicialização. Foi a partir disso que o termo *bug* começou a ser utilizado e se popularizou no mundo todo, afinal, os *bugs* computacionais são uma constante realidade do processo de programação.

Mas, e quanto ao *debug*?

Para que a definição de *debug* seja simples e objetiva, considere novamente a ocasião em que a engenheira achou insetos dentro do computador e verificou que eles seriam o motivo pelo qual o equipamento não funcionava. Pois bem, o que ela fez foi *debug*, ou seja, procurou o erro.

Mas fique tranquilo, pois você não precisará procurar nenhum inseto. Este conteúdo visa a ajudá-lo para que a busca pelo erro não seja feita manualmente, e sim com uma ferramenta específica de *debug*. E acredite, este conteúdo é muito poderoso para o seu futuro como programador!

Para realizar os testes utilizando o *debug*, este conhecimento o ajudará a simular um cenário de uma aplicação de cadastro de pessoas utilizando o NetBeans e o MySQL. O seguinte *script* é usado para a criação da base de dados:

```
CREATE DATABASE principal;

USE principal;

CREATE TABLE pessoa
(
    id_pessoa INT NOT NULL PRIMARY KEY auto_increment,
    nome      VARCHAR(100),
    telefone  VARCHAR(15),
    idade     INT,
    peso      DECIMAL(10, 0),
    cpf       VARCHAR(11)
);
```

Para testes, é possível popular a tabela com algumas informações.

```
INSERT INTO pessoa (nome, telefone, idade, peso, cpf)
VALUES ('JOSE', '32356', 30, 90, '00000000000'),
       ('THIAGO', '1560', 20, 80, '00000000000'),
       ('JESSICA', '35698', 30, 60, '00000000000'),
       ('PEDRO', '35198', 50, 75, '00000000000');
```

Serão simulados alguns erros em aplicação para se descobrir a melhor forma de utilizar o *debug* nessa situação.

Agora, crie uma aplicação Java Ant chamada TesteDebug e inclua a biblioteca de conexão com o MySQL (se necessário, retorne ao conteúdo **Conexão com banco de dados** desta unidade curricular para rever detalhes de como preparar um projeto).

O projeto ficará com a seguinte estrutura:

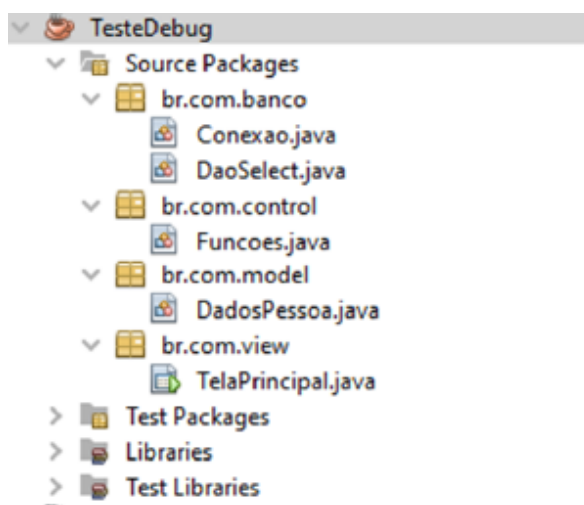


Figura 3 – Projeto TesteDebug

Fonte: NetBeans (2022)

Depois de criar o projeto, crie também três pacotes Java: **br.com.banco**, **br.com.control** e **br.com.model**. No projeto, uma classe de conexão chamada **ConexãoMySQL** foi incluída no pacote **br.com.banco** utilizando-se a JDBC (Java Database Connectivity):

```
package br.com.banco;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.JOptionPane;

public class ConexaoMySQL {

    public Connection getConnection() {
        try {
            Connection conexao = DriverManager.getConnection("jdbc:mysql://localhost/principal", "root", "masterkey");
            return conexao;
        } catch (SQLException ex) {
            Logger.getLogger(ConexaoMySQL.class.getName()).log(Level.SEVERE, null, ex);
            JOptionPane.showMessageDialog(null, ex.getMessage());
        }
        return null;
    }
}
```

Nessa classe, você fará uma conexão apontando diretamente a URL (*uniform resource locator*) na qual está o banco de dados; passará usuário e senha para a conexão e criará o método para realizar a conexão (não é uma boa prática passar usuário e senha sem criptografia para conexão com o banco de dados). Depois disso, você criará o objeto correspondente à tabela do banco de dados e o método para trazer o **ResultSet** (interface utilizada para guardar dados vindos de um banco de dados), e então populará uma lista de objeto **PESSOA**.

A classe a seguir deve ser criada em **br.com.model**, com o nome **DadosPessoa**:

```
package br.com.model;

public class DadosPessoa {
    private String nome, telefone, cpf;
    private int id, idade;
    private Double peso;

    public DadosPessoa(String nome, String telefone, String cpf, int id, int idade,
Double peso) {
        this.nome = nome;
        this.telefone = telefone;
        this.cpf = cpf;
        this.id = id;
        this.idade = idade;
        this.peso = peso;
    }

    public DadosPessoa() {
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getTelefone() {
        return telefone;
    }

    public void setTelefone(String telefone) {
        this.telefone = telefone;
    }

    public String getCpf() {
        return cpf;
    }

    public void setCpf(String cpf) {
        this.cpf = cpf;
    }

    public int getId() {
        return id;
    }
}
```

```
public void setId(int id) {  
    this.id = id;  
}  
  
public int getIdade() {  
    return idade;  
}  
  
public void setIdade(int idade) {  
    this.idade = idade;  
}  
  
public Double getPeso() {  
    return peso;  
}  
  
public void setPeso(Double peso) {  
    this.peso = peso;  
}  
}
```

A classe corresponde ao objeto **pessoa**, seguindo os mesmos atributos do banco, com seus respectivos métodos de acesso (**Getters** e **Setters**).

O projeto também conta com uma classe DAO para pessoas: **DaoSelect**:


```
package br.com.banco;

import br.com.model.DadosPessoa;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.JOptionPane;

public class DaoSelect {

    private Conexao conexao;
    private Statement stm;
    private Connection con;
    private ResultSet rs;

    public void conect() {
        try {
            conexao = new Conexao();
            con = conexao.getConnection();
            stm = con.createStatement();
        } catch (SQLException ex) {
            Logger.getLogger(DaoSelect.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

    public void desconect() {
        try {
            stm.close();
            con.close();
        } catch (SQLException ex) {
            Logger.getLogger(DaoSelect.class.getName()).log(Level.SEVERE, null, ex);
            JOptionPane.showMessageDialog(null, ex.getMessage());
        }
    }

    public ResultSet selectPessoa() {
        try {
            return stm.executeQuery("SELECT * FROM PESSOA");
        } catch (SQLException ex) {
            Logger.getLogger(DaoSelect.class.getName()).log(Level.SEVERE, null, ex);
            JOptionPane.showMessageDialog(null, ex.getMessage());
        }
    }
}
```

```
        return null;
    }

    public void insertPessoa(DadosPessoa d){
        try {
            stm.execute("INSERT INTO PESSOA (NOME, TELEFONE, IDADE, PESO, CPF) VALU
ES ('"+d.getNome()+"', '"+d.getTelefone()+"', "+d.getIdade()+"', "+d.getPeso()+"',
'"+d.getCpf()+"'");");
        } catch (SQLException ex) {
            Logger.getLogger(DaoSelect.class.getName()).log(Level.SEVERE, null, e
x);
            JOptionPane.showMessageDialog(null, ex.getMessage());
        }
    }
}
```

A classe **Funcoes** deve ser criada no pacote **br.com.control**:

```
package br.com.control;

import br.com.banco.DaoSelect;
import br.com.model.DadosPessoa;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.JOptionPane;

public class Funcoes {

    public void buscaPessoa(){
        DaoSelect daoSelect = new DaoSelect();
        daoSelect.conect();
        ResultSet rs = daoSelect.selectPessoa();

        List<DadosPessoa> listPessoa = new ArrayList<>();

        try {
            while(rs.next()){
                DadosPessoa dadosPessoa = new DadosPessoa();
                dadosPessoa.setCpf(rs.getString("CPF"));
                dadosPessoa.setId(rs.getInt("ID"));
                dadosPessoa.setIdade(rs.getInt("IDADE"));
                dadosPessoa.setNome(rs.getString("NOME"));
                dadosPessoa.setPeso(rs.getDouble("PESO"));
                dadosPessoa.setTelefone(rs.getString("TELEFONE"));
                listPessoa.add(dadosPessoa);
            }
        } catch (SQLException ex) {
            Logger.getLogger(Funcoes.class.getName()).log(Level.SEVERE, null, ex);
            JOptionPane.showMessageDialog(null, ex.getMessage());
        }
        daoSelect.desconect();

        System.out.println(listPessoa.size());
    }

    public void inserePessoa(){
        DaoSelect daoSelect = new DaoSelect();
        daoSelect.conect();

        DadosPessoa dadosPessoa = new DadosPessoa();
        dadosPessoa.setCpf("12345678911");
        dadosPessoa.setIdade(11);
        dadosPessoa.setNome("teste");
    }
}
```

```
        dadosPessoa.setPeso(10.0);  
        dadosPessoa.setTelefone("12345678911151515");  
  
        daoSelect.insertPessoa(dadosPessoa);  
  
        daoSelect.desconnect();  
    }  
}
```

Você deve ter atenção especial ao método para **buscaPessoa()**. Por fim, você criou uma tela somente para ilustração com um botão para buscar as pessoas no banco e outro para sair do sistema. Essa classe se chama **TelaPrincipal** e tem o seguinte *layout*:

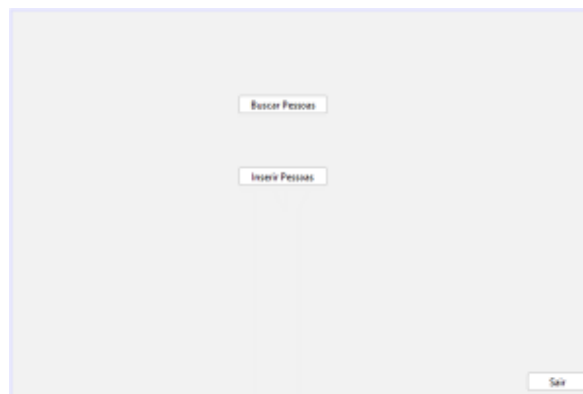


Figura 4 – Tela para testes

Fonte: NetBeans (2022)

Os eventos de clique para os botões são os seguintes: **btnBuscarPessoa**, **btnSair** e **btnInserirPessoas**:

```
private void btnBuscarPessoaActionPerformed(java.awt.event.ActionEvent evt) {  
    new Funcoes().buscaPessoa();  
}  
  
private void btnSairActionPerformed(java.awt.event.ActionEvent evt) {  
    System.exit(0);  
}  
  
private void btnInserirPessoasActionPerformed(java.awt.event.ActionEvent evt) {  
    new Funcoes().inserePessoa();  
}
```

Com isso, seu projeto está pronto para ser testado.

Detecção e correção de erros

Ao se executar o sistema e clicar no botão **Buscar Pessoas**, já é lançada uma exceção. Primeiramente, executa-se a rotina apertando o botão de *play*, pois ele funciona como a execução normal, como se a pessoa que usará o sistema desse dois cliques para abri-lo. A diferença é que, na IDE, há um terminal no qual se consegue ler possível parte do erro, porém, nem sempre essa parte é suficiente para que o problema seja encontrado, pois, muitas vezes, o erro vem de forma genérica, com termos técnicos ou confusos.

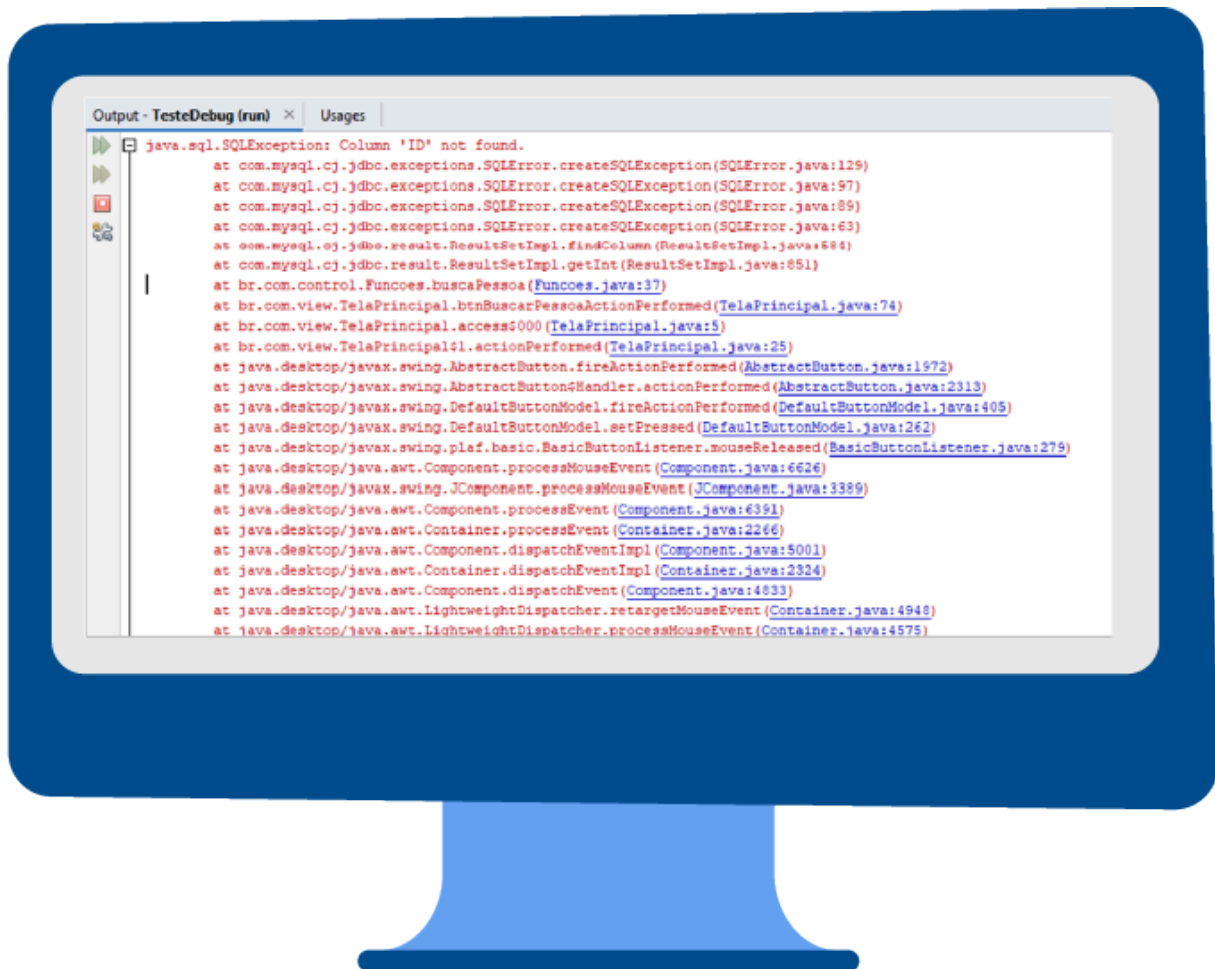


Figura 5 – Falha gerada pela aplicação (seleccionando dados de pessoa)

Fonte: NetBeans (2022)

Essa é a exceção lançada pelo sistema. Com o tempo e a experiência, ficará mais fácil entender o erro devido à familiaridade com algumas *exceptions*. Mesmo assim, ainda poderá ser difícil encontrar alguns erros apenas lendo o console. Nesse contexto, utiliza-se o *debug*, pois com ele se consegue entender o funcionamento do código e assim ver onde e por que o erro está acontecendo.

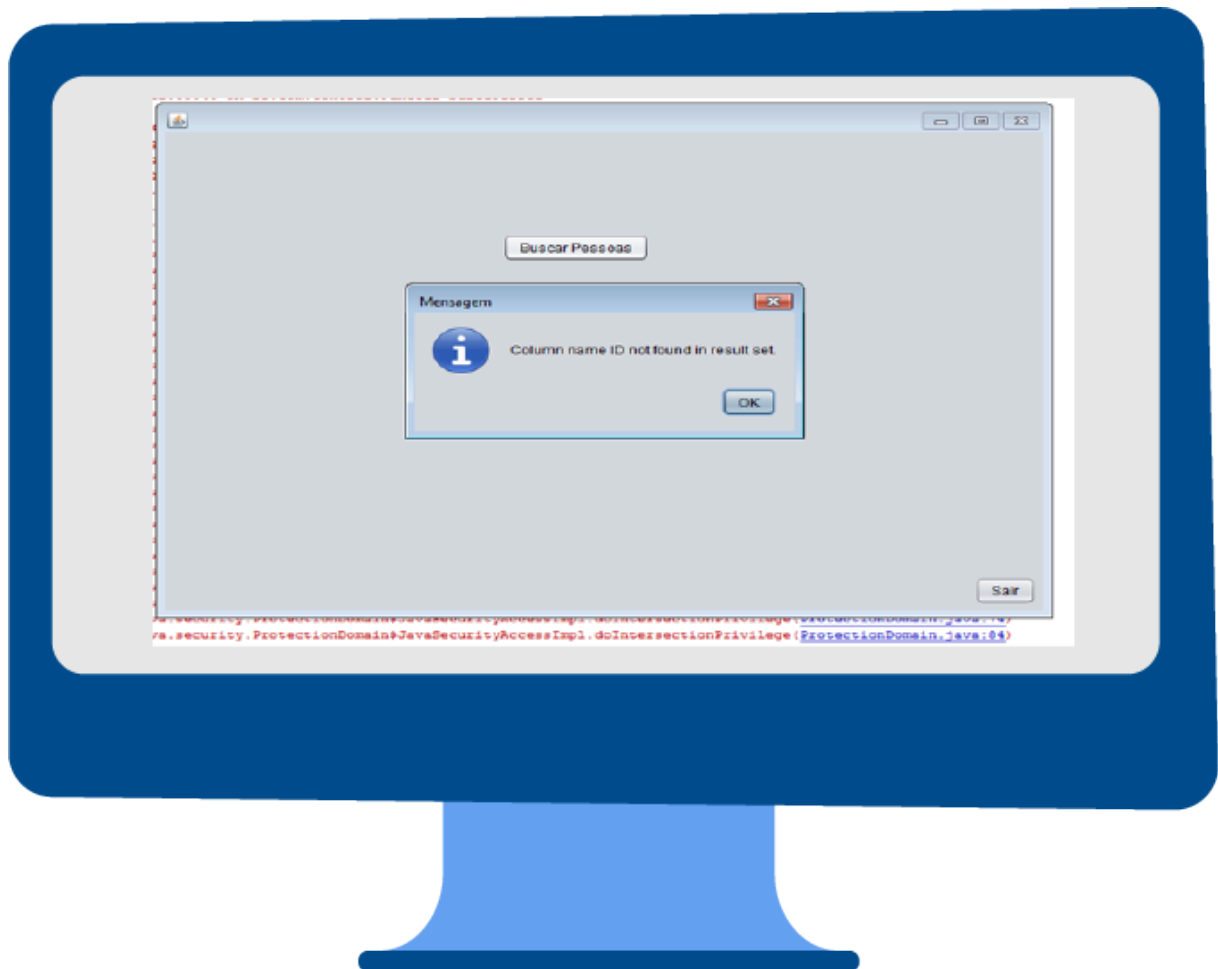


Figura 6 – Mensagem de erro em janela de diálogo

Fonte: NetBeans (2022)

Essa é uma exceção tratada pelo *try-catch*. Geralmente, essas são bem mais enxutas e são utilizadas para os programadores terem um norte de qual pode ser o problema em tempo de execução. Então, como a exceção lançada pelo sistema não possibilita uma visão tão clara do erro, comece pelo *debug*. O primeiro passo é executar o sistema em modo *debug*:

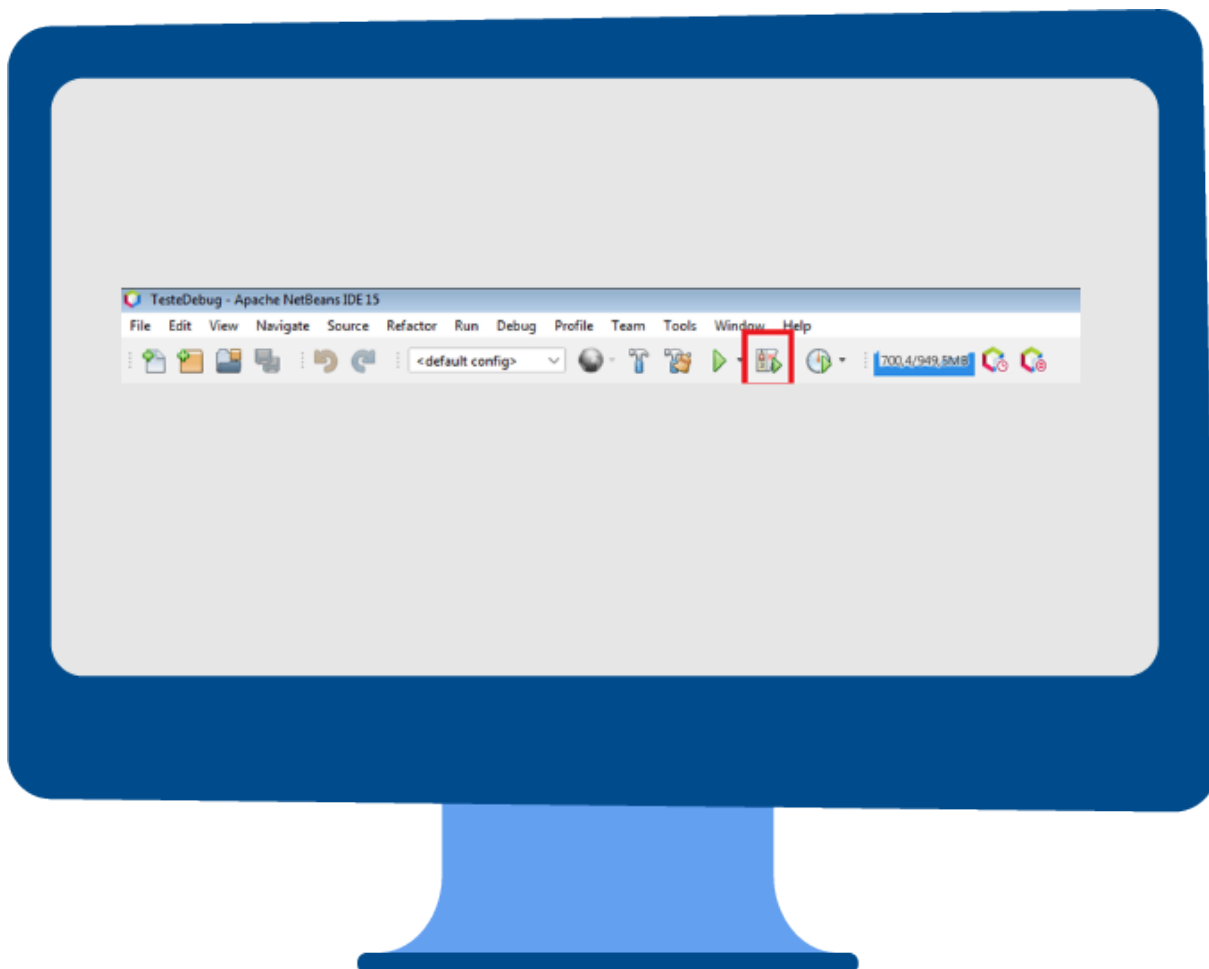


Figura 7 – Botão **Debug Project**

Fonte: NetBeans (2022)

É só clicar no botão **Debug Project** que está presente na barra de ferramentas do NetBeans que o sistema já será executado em modo *debug*. Mas lembre-se de que o modo *debug* precisa de um *breakpoint*, ou seja, um “ponto de parada”. Então, antes de executar em modo *debug*, você precisa mostrar ao sistema onde ele deve parar para você encontrar o erro. Nesse momento, marque esse *breakpoint* no início do sistema, ou seja, quando o usuário apertar o botão. Para colocar o *breakpoint*, basta clicar na lateral esquerda da linha desejada. Aqui, você quer depurar a linha que realiza o comando **Funcoes().buscaPessoa()** (no caso do exemplo, esse comando está na linha 74).

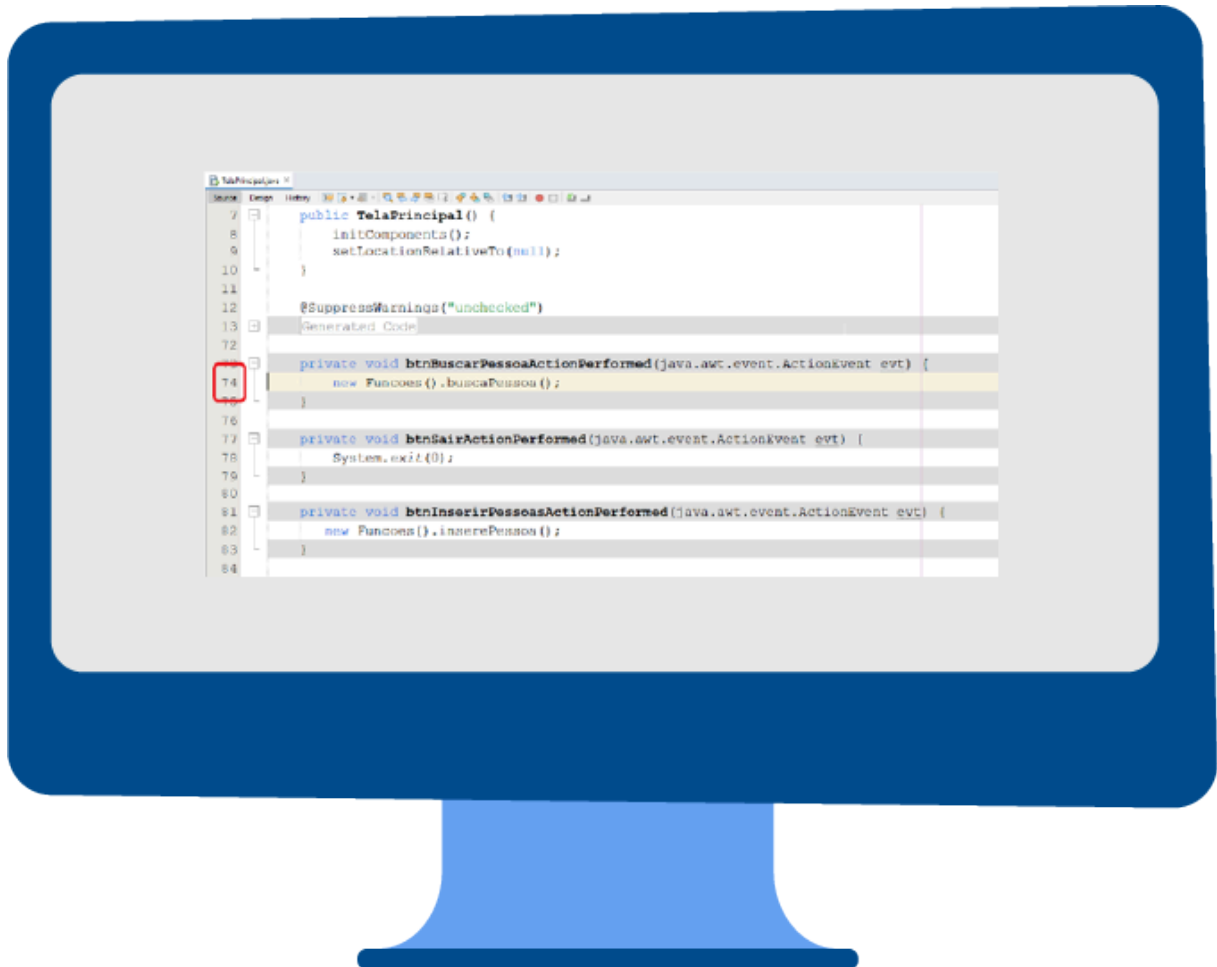


Figura 8 – Localizando onde incluir *breakpoint*

Fonte: NetBeans (2022)

Depois de clicar-se na linha, como apontado pelo destaque na imagem, seu número ficará com um quadrado vermelho e toda a linha ficará com fundo em outra cor, sinalizando que o *breakpoint* foi colocado.

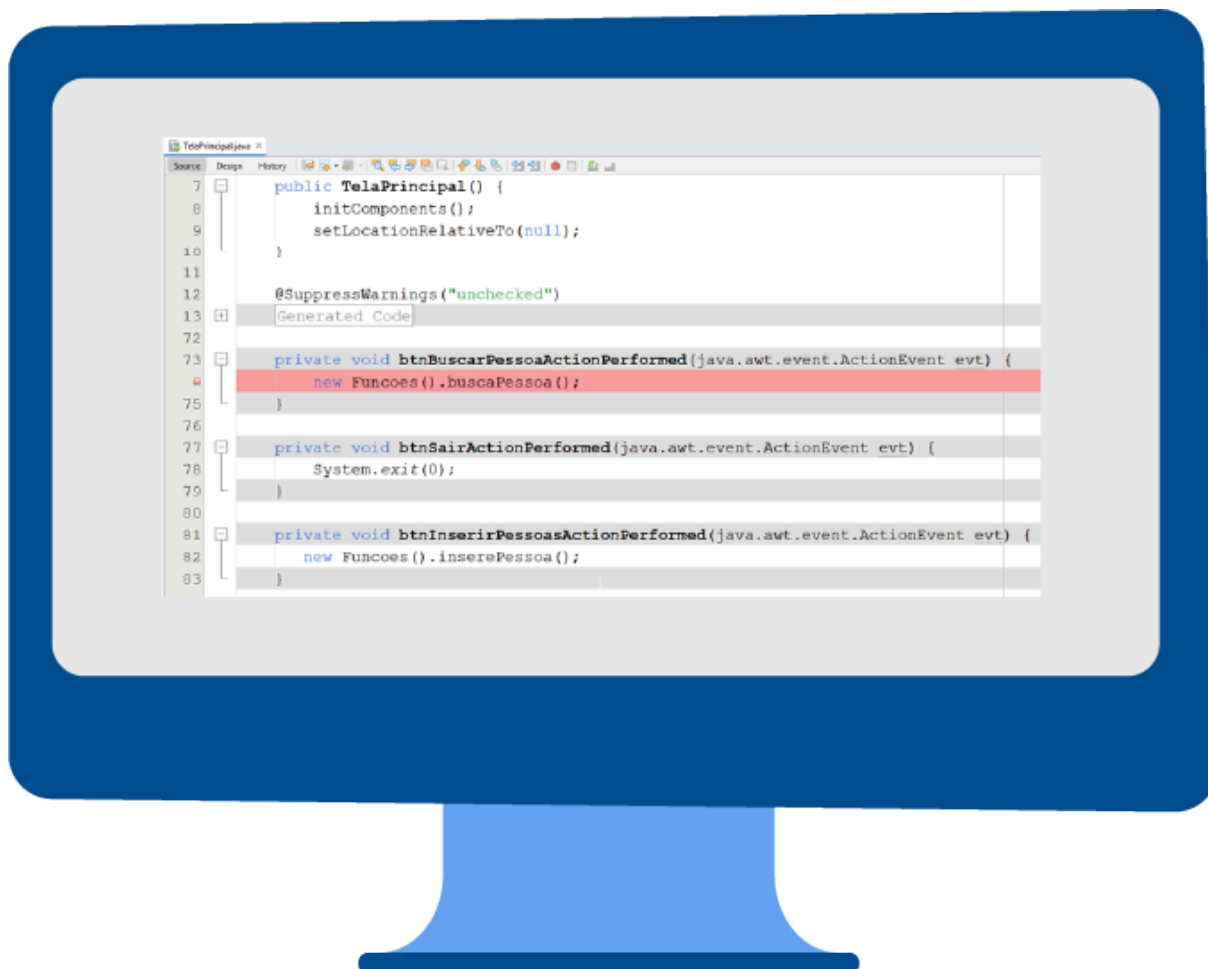


Figura 9 – *Breakpoint* incluído

Fonte: NetBeans (2022)

Agora, com o *breakpoint* devidamente inserido, você **executará em modo *debug***, clicando no ícone **Debug Project** ou usando o atalho **Ctrl + F5**. O NetBeans abrirá um menu de *debug* depois de executado, conforme esta imagem:

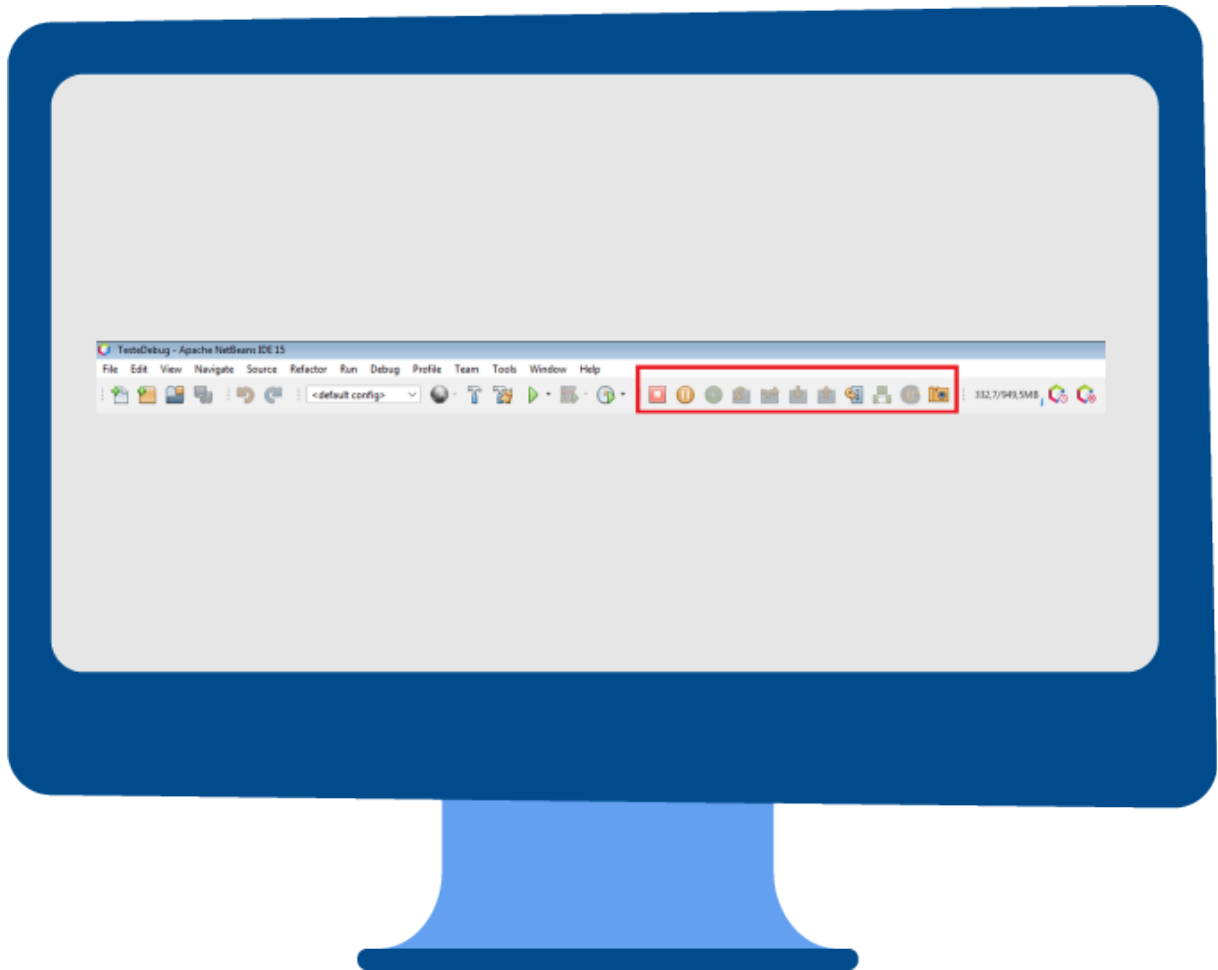


Figura 10 – Funções de *debug*

Fonte: NetBeans (2022)

Esse é o menu de *debug* depois de o sistema ser executado em modo *debug*. Por enquanto, ele está quase todo desabilitado, porque o sistema ainda não chegou no *breakpoint* definido. Primeiro, é preciso clicar no **Buscar Pessoas** para ele parar no *breakpoint* e começar o *debug*, pois foi assim que você indicou para o sistema buscar pessoas.

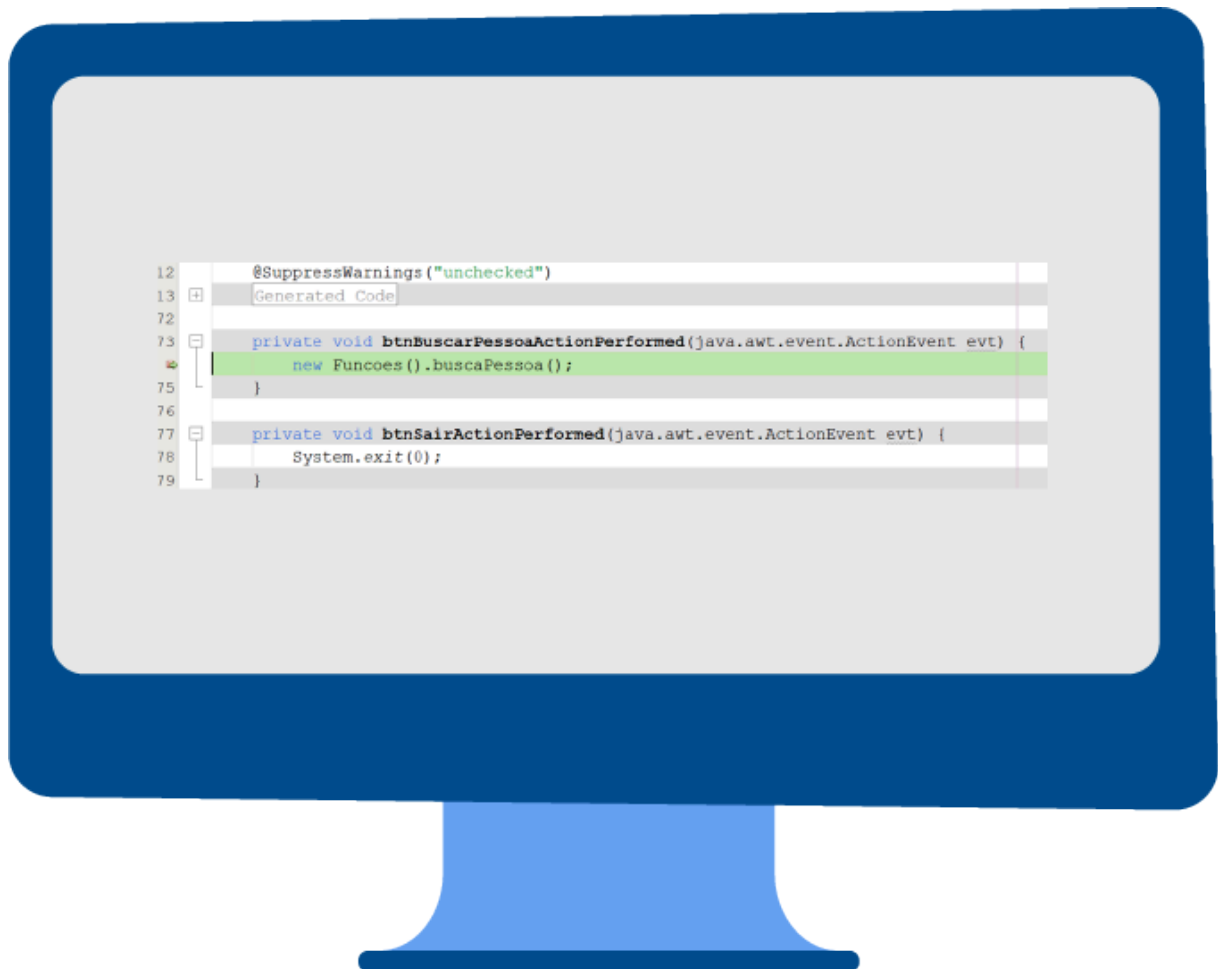


Figura 11 – Executando o *debug* e parando no *breakpoint*

Fonte: NetBeans (2022)

O primeiro comportamento observado é que a linha na qual foi colocado o *breakpoint* fica toda em verde, sinalizando que da aplicação até aqui não resultou erro e ela está esperando a sua ação para continuar de acordo com sua escolha.

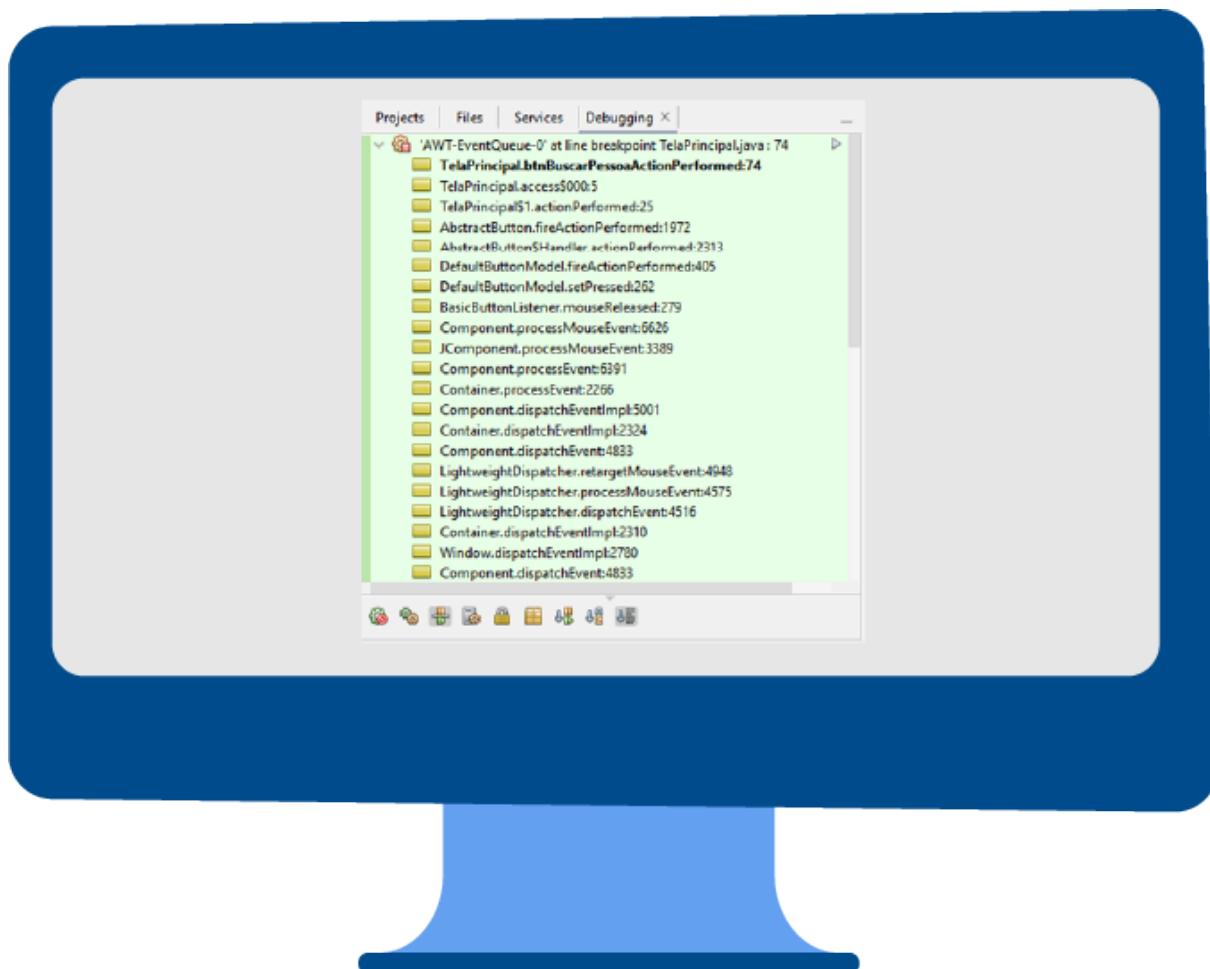


Figura 12 – Painel de depuração

Fonte: NetBeans (2022)

Em seguida, um painel será aberto na lateral esquerda da aba **Debugging**, com as informações de tudo que foi executado até aqui. Desde a abertura da tela, mostrando tamanho, até a linha que fica em negrito, mostrando onde parou, ou seja no breakpoint, a ação de apertar o botão na tela principal linha 74.

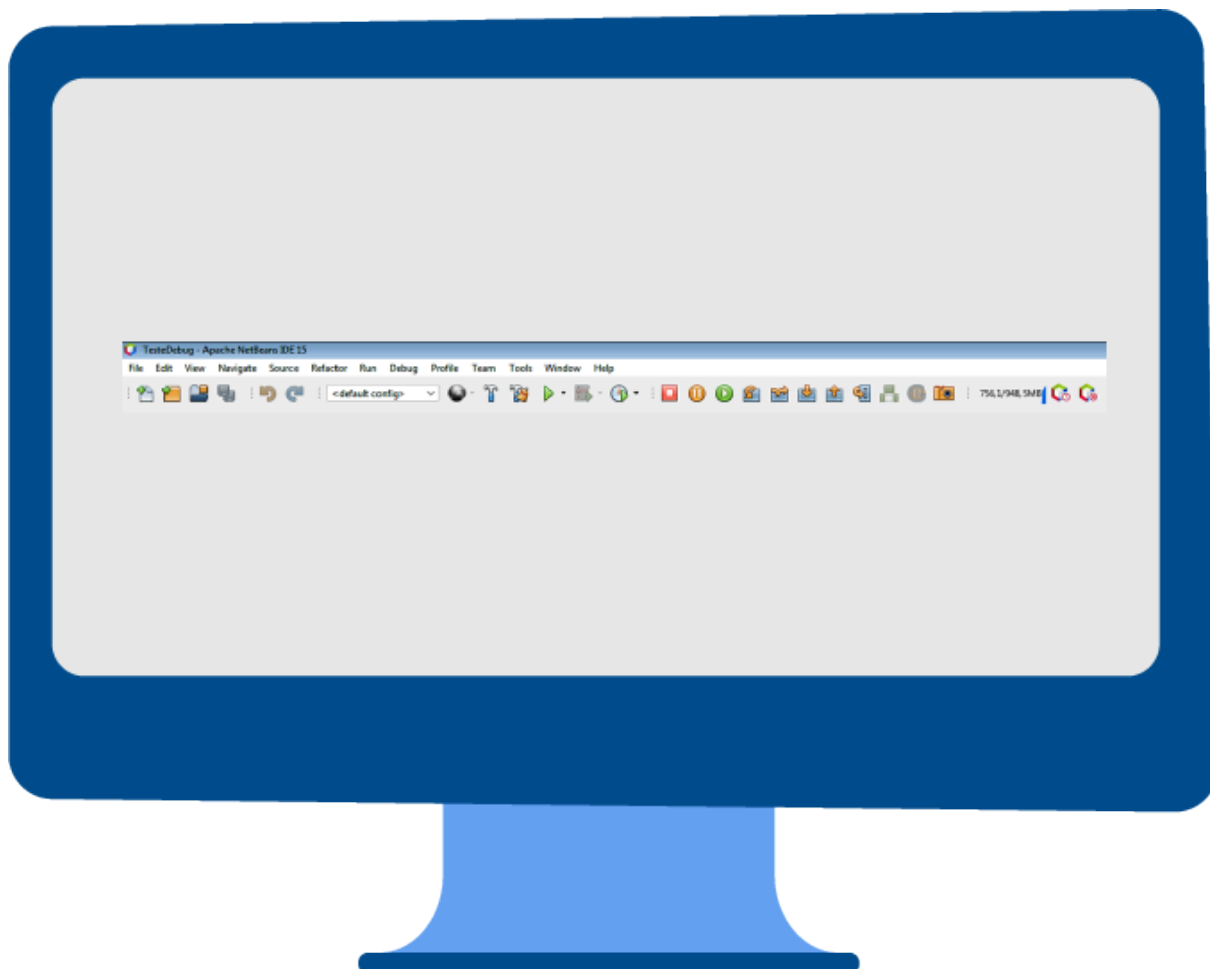


Figura 13 – Ferramentas de depuração em execução

Fonte: NetBeans (2022)

Note também que por causa da execução ter parado no *breakpoint*, o menu de *debug* foi liberado com as opções de depuração e você poderá escolher o que fazer a seguir -- encerrar a aplicação, entrar no código, continuar a execução entre outras. Conheça a seguir as opções mais importantes e suas funcionalidades:

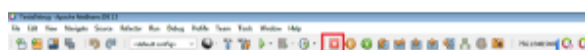


Figura 14 – Botão **Finish Debugger Session** para encerrar depuração

Fonte: NetBeans (2022)

- ◆ **Finish Debugger Session (stop):** com ele, pode-se sair do modo *debug* e encerrar a aplicação.

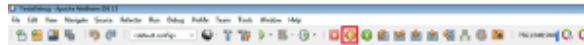


Figura 15 – Botão **Pause** para pausar depuração

Fonte: NetBeans (2022)

- ◆ **Pause:** com ele, pausa-se a aplicação e, caso haja algo rodando por trás, ele para tudo e concentra-se no *breakpoint*.



Figura 16 – Botão **Continue** para continuar a depuração

Fonte: NetBeans (2022)

- ◆ **Continue (play):** com ele, pode-se continuar a aplicação, ignorando o *breakpoint* atual (ele para no próximo, caso haja mais de um).



Figura 17 – Botão **Step Over** para continuar a depuração

Fonte: NetBeans (2022)

- ◆ **Step Over (ignorar):** um dos mais utilizados, esse botão serve para o usuário ir para a linha de baixo, conseguindo assim descobrir qual a linha está com erro e analisar as linhas posteriores, descobrindo o que está causando o erro.

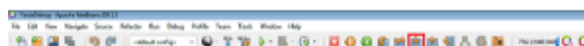


Figura 18 – Botão **Step Into** para continuar a depuração

Fonte: NetBeans (2022)

- ◆ **Step Into (entrar):** esse botão complementa a opção anterior. Quando você entrar em um método, não conseguirá pular de linha em linha, então usará esse botão, que entra no método que está no *breakpoint*, e, dentro do método, você consegue analisar as linhas. Nesse caso, primeiro se utiliza o botão **Step Into** para entrar no método e depois passar de linha em linha.

Quando você clica nele pela primeira vez, o botão grifa o método presente no *breakpoint*, e se for esse o método que você quer “debugar”, é só clicar novamente para ir para a primeira linha do método. Lembre-se de que é preciso avaliar sempre a necessidade ou não de entrar nesse método, pois você pode não precisar entrar em todos os métodos. Deve-se avaliar caso a caso.

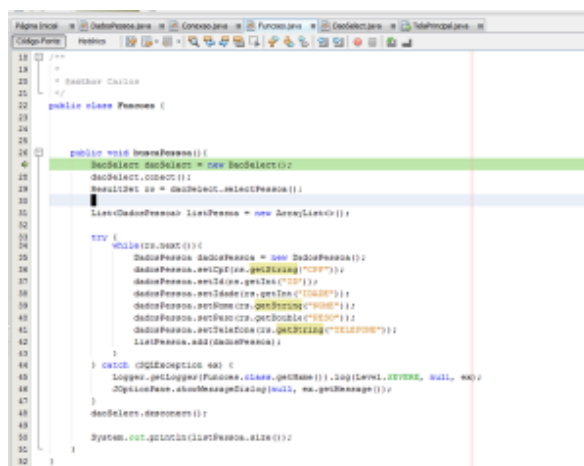


Figura 19 –Depurando o código do método **buscaPessoa()**

Fonte: NetBeans (2022)

Agora sim, o *breakpoint* veio para a primeira linha do método, como mencionado anteriormente, e daqui você passará de linha em linha no botão **Step Over** (ignorar) até encontrar a linha com erro e verificar qual é a possibilidade deste.

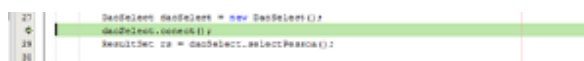


Figura 20 – **Step Over** mandando para a linha seguinte

Fonte: NetBeans (2022)

O *breakpoint* passou para a linha de baixo, linha 28, na qual está sendo chamado o método de conexão com o banco de dados. Clique em **Step Over** novamente:



Figura 21 – Seguindo o código com **Step Over**

Fonte: NetBeans (2022)

Agora, você foi para a linha na qual está sendo atribuído um **ResultSet** a uma variável chamada **rs**, para ser tratada dentro do método. Siga com **Step Over**:

```

28      daSelect.close();
29      ResultSet rs = daSelect.selectPessoa();
30
31      List<DadosPessoa> listPessoa = new ArrayList<>();
32

```

Figura 22 – Depurando a linha 31

Fonte: NetBeans (2022)

Declarando a variável de lista de pessoas para receber os dados do banco. Siga com **Step Over**:

```

33      try {
34          while(rs.next()){
35              DadosPessoa dadosPessoa = new DadosPessoa();
36              dadosPessoa.setCpf(rs.getString("CPF"));
37              dadosPessoa.setId(rs.getInt("ID"));
38              dadosPessoa.setIdade(rs.getInt("IDADE"));
39              dadosPessoa.setNome(rs.getString("NOME"));
40              dadosPessoa.setPeso(rs.getDouble("PESO"));
41              dadosPessoa.setTelefone(rs.getString("TELEFONE"));
42          }
43      }
44

```

Figura 23 – Depurando a linha 34

Fonte: NetBeans (2022)

Agora teste se o **ResultSet** tem resultados. Continue com **Step Over**:

```

33      try {
34          while(rs.next()){
35              DadosPessoa dadosPessoa = new DadosPessoa();
36              dadosPessoa.setCpf(rs.getString("CPF"));
37              dadosPessoa.setId(rs.getInt("ID"));
38              dadosPessoa.setIdade(rs.getInt("IDADE"));
39              dadosPessoa.setNome(rs.getString("NOME"));
40              dadosPessoa.setPeso(rs.getDouble("PESO"));
41              dadosPessoa.setTelefone(rs.getString("TELEFONE"));
42              listPessoa.add(dadosPessoa);
43          }
44      }
45

```

Figura 24 – Depurando a linha 36

Fonte: NetBeans (2022)

Agora será modificado o valor do campo “CPF” do banco de dados, no objeto “Pessoa”, no atributo “cpf”. Siga com **Step Over**:

```

33      try {
34          while(rs.next()){
35              DadosPessoa dadosPessoa = new DadosPessoa();
36              dadosPessoa.setCpf(rs.getString("CPF"));
37              dadosPessoa.setId(rs.getInt("ID"));
38              dadosPessoa.setIdade(rs.getInt("IDADE"));
39              dadosPessoa.setNome(rs.getString("NOME"));
40              dadosPessoa.setPeso(rs.getDouble("PESO"));
41              dadosPessoa.setTelefone(rs.getString("TELEFONE"));
42              listPessoa.add(dadosPessoa);
43          }
44      }
45

```

Figura 25 – Depurando a linha 37

Fonte: NetBeans (2022)

Alterando o valor do campo “ID” do banco de dados, no objeto “pessoa”, no atributo “id”.

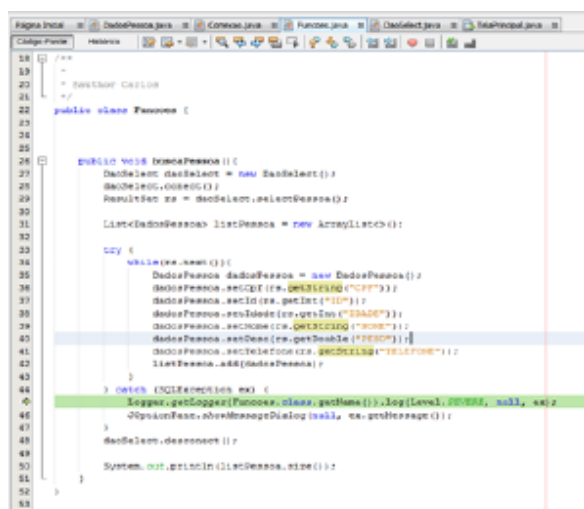


Figura 26 – **Step Over** pulando para linha de exceção – Erro detectado

Fonte: NetBeans (2022)

Perfeito! Ao invés de o sistema ir para a próxima linha, ele está pulando para o *catch*, ou seja, ele encontrou uma exceção na linha 37. Agora, pare o *debug* (botão **Finish Debugger Session**). Você pode mudar seu *breakpoint* diretamente para a linha 37 da classe **Funcoes** para facilitar a depuração – fica muito mais fácil a depuração com um *breakpoint* direto onde é necessário analisar o que o programa está fazendo ao invés de sempre percorrer várias linhas de código até chegar ao ponto em que o problema acontece. Só não esqueça de desativar o *breakpoint* incluído anteriormente, para que não haja dois pontos de parada desnecessariamente.

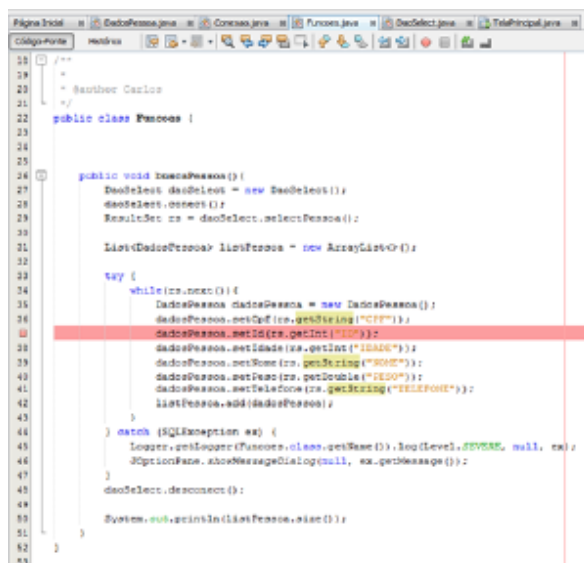


Figura 27 – Novo *breakpoint*, agora no ponto de erro

Fonte: NetBeans (2022)

Como o *breakpoint* colocado no lugar necessário e aquele da tela principal já retirado – pois você sabe onde está o problema –, execute o modo *debug* de novo, não esquecendo de apertar o botão de buscar pessoas novamente para que ele tenha o mesmo comportamento.

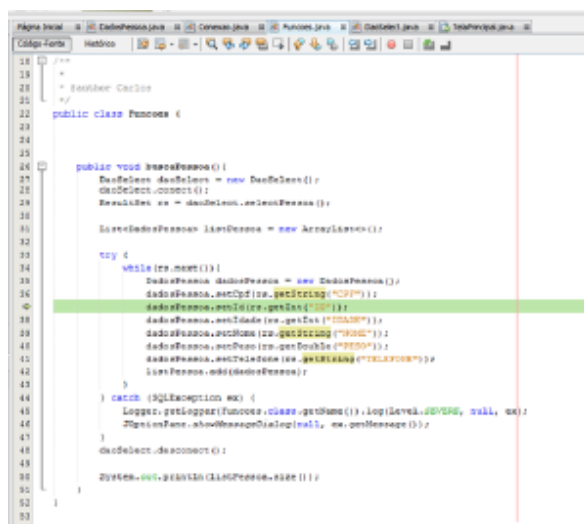


Figura 28 – Depuração parada na linha que gera o problema

Fonte: NetBeans (2022)

Pronto, agora o sistema parou na linha com o problema. Analise o que pode estar acontecendo:

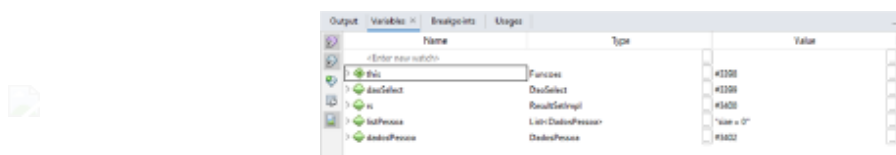


Figura 29 – Painel de observação de variáveis

Fonte: NetBeans (2022)

Na parte inferior são mostradas as variáveis em tempo de execução do sistema, desde a classe que está sendo executada até a lista, por exemplo, de pessoas que foi criada, além do objeto “pessoa” declarado para popular a lista.

No caso da lista, na coluna “valor” consta “size = 0”, ou seja, a lista está vazia e isso quer dizer que já há erro na primeira pessoa trazida do banco. Continue para verificar mais eventos que possam estar ocasionando o erro.

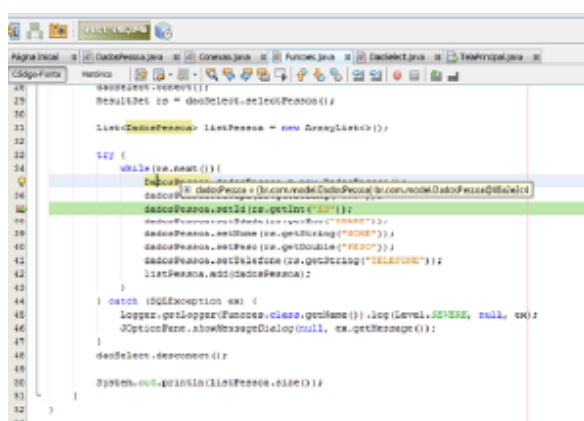


Figura 30 – Observação de variáveis ao posicionar *mouse* sobre o código

Fonte: NetBeans (2022)

Quando se deixa o *mouse* parado sobre **dadosPessoa**, aparece o endereço na memória do objeto, possibilitando-se expandir esse objeto para ver o que tem nele, o que, na maioria dos casos, ajuda muito o programador.



Figura 31 – Detalhes sobre o objeto **dadosPessoa**

Fonte: NetBeans (2022)

Assim, é possível ver os atributos e verificar que, até agora, somente o atributo “CPF” foi populado, ou seja, ao verificar a sequência do **ResultSet**, nota-se que o primeiro atributo é o CPF e depois está o ID. Então, verifique se existe algo errado no *set* do ID. Basta selecionar com o cursor o que está dentro do *set* que este mostrará o que pode estar errado.

```

List<DadosPessoa> listPessoa = new ArrayList<>();

try {
    while(rs.next()){
        DadosPessoa dadosPessoa = new DadosPessoa();
        dadosPessoa.setCpf(rs.getString("CPF"));
        dadosPessoa.setId(rs.getInt("ID"));
        dadosPessoa.setNome(rs.getString("NOME"));
        dadosPessoa.setPeso(rs.getDouble("PESO"));
        dadosPessoa.setTelefone(rs.getString("TELEFONE"));
        listPessoa.add(dadosPessoa);
    }
} catch (SQLException ex) {
    Logger.getLogger(Funcoes.class.getName()).log(Level.SEVERE, null, ex);
    JOptionPane.showMessageDialog(null, ex.getMessage());
}

```

Figura 32 – Verificando detalhes dos objetos

Fonte: NetBeans (2022)

O erro indica que a coluna com o nome de ID não foi encontrada no **ResultSet**. Verificando no banco de dados pode-se notar que realmente há um erro. A coluna no banco está nomeada como “ID_PESSOA”, então, quando procurado no **ResultSet** um nome de coluna que esteja diferente do nome da coluna no banco de dados, o Java retorna uma exceção indicando que não encontra. Corrigindo esse problema, você conseguirá rodar a aplicação sem nenhum erro; basta parar o *debug* no botão **Stop**, ajustar o nome da coluna e executar o sistema em modo *debug* novamente. Assim, quando você clicar no botão de **ignorar**, o sistema não pulará para o *catch* para dar uma exceção, e sim irá para a próxima linha, como na imagem a seguir:

```

ResultSet rs = daoSelect.selectPessoa();

List<DadosPessoa> listPessoa = new ArrayList<>();

try {
    while(rs.next()){
        DadosPessoa dadosPessoa = new DadosPessoa();
        dadosPessoa.setCpf(rs.getString("CPF"));
        dadosPessoa.setId(rs.getInt("ID_PESSOA"));
        dadosPessoa.setNome(rs.getString("NOME"));
        dadosPessoa.setPeso(rs.getDouble("PESO"));
        dadosPessoa.setTelefone(rs.getString("TELEFONE"));
        listPessoa.add(dadosPessoa);
    }
} catch (SQLException ex) {
    Logger.getLogger(Funcoes.class.getName()).log(Level.SEVERE, null, ex);
    JOptionPane.showMessageDialog(null, ex.getMessage());
}

daoSelect.disconnect();

System.out.println(listPessoa.size());

```


Foi lançada uma exceção que está cheia de informações e é muito genérica, dificultando chegar-se ao real erro. Por isso, inicie o processo de *debug* adicionando o *breakpoint* em um local estratégico, ou seja, na chamada do botão de inserir pessoas:



Figura 35 – *Breakpoint* na linha que inicia cadastramento de pessoas

Fonte: NetBeans (2022)

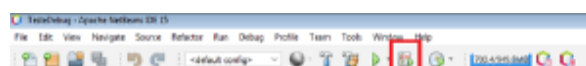


Figura 36 – Botão **Debug Project**

Fonte: NetBeans (2022)

Neste momento, basta iniciar o sistema em modo *debug* para você começar a analisar o caso:

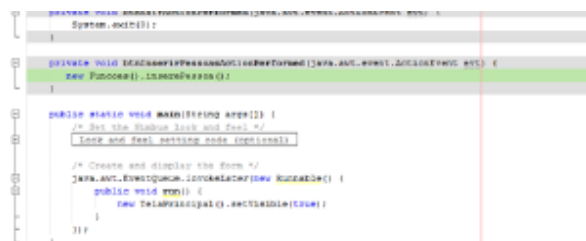


Figura 37 – Iniciando a depuração

Fonte: NetBeans (2022)

Agora que o *breakpoint* já parou na linha, quando clicada para inserir as pessoas, clique no botão de **entrar** para entrar no método, como na imagem a seguir:



Figura 38 – Entrando no método

Fonte: NetBeans (2022)

```

public void inserePessoa() {
    DaoSelect daoSelect = new DaoSelect();
    daoSelect.conect();

    DadosPessoa dadosPessoa = new DadosPessoa();
    dadosPessoa.setCpf("12345678911");
    dadosPessoa.setIdade(11);
    dadosPessoa.setNome("teste");
    dadosPessoa.setPeso(10.0);
    dadosPessoa.setTelefone("12345678911151515");

    daoSelect.insertPessoa(dadosPessoa);

    daoSelect.desconect();
}

```

Figura 39 – Depurando **inserePessoa()**

Fonte: NetBeans (2022)

O sistema foi para a primeira linha do método. Passe então as linhas com **Step Over** até encontrar o erro:

```

public void inserePessoa() {
    DaoSelect daoSelect = new DaoSelect();
    daoSelect.conect();

    DadosPessoa dadosPessoa = new DadosPessoa();
    dadosPessoa.setCpf("12345678911");
    dadosPessoa.setIdade(11);
    dadosPessoa.setNome("teste");
    dadosPessoa.setPeso(10.0);
    dadosPessoa.setTelefone("12345678911151515");

    daoSelect.insertPessoa(dadosPessoa);
}

```

Figura 40 – Depurando **inserePessoa()**

Fonte: NetBeans (2022)

```

53 public void inserePessoa() {
54     DaoSelect daoSelect = new DaoSelect();
55     daoSelect.conect();
56
57     DadosPessoa dadosPessoa = new DadosPessoa();
58     dadosPessoa.setCpf("12345678911");
59     dadosPessoa.setIdade(11);
60     dadosPessoa.setNome("teste");
61     dadosPessoa.setPeso(10.0);
62     dadosPessoa.setTelefone("12345678911151515");
63
64     daoSelect.insertPessoa(dadosPessoa);
65
66     daoSelect.desconect();
67
68 }
69
70

```

Figura 41 – Depurando **inserePessoa()**

Fonte: NetBeans (2022)

Passando **Step Over** até a linha na qual os dados são inseridos no banco de dados, nenhum erro foi encontrado, pois passou-se por todas as linhas sem ocorrer erro ou pular alguma. Então, siga adiante para a próxima busca do erro, pois ele só pode estar dentro do método de inserir. Para isso, confirme se o erro aparecerá na linha do método de inserção: se sim, analise os dados e compare-os com o banco para ver se são compatíveis.

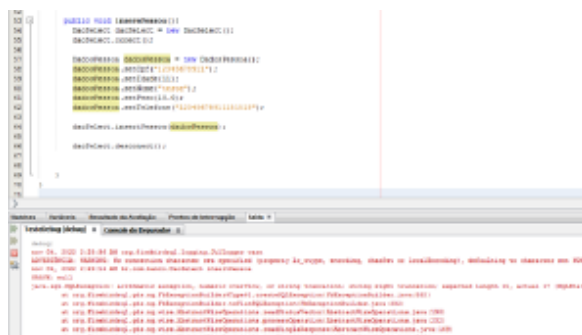


Figura 42 – Exceção lançada no console

Fonte: NetBeans (2022)

Isso mesmo, o erro está na inserção. Agora, altere o *breakpoint* para a linha da inserção, retire-o da tela principal, clique no botão **Stop** e verifique os dados presentes no objeto.

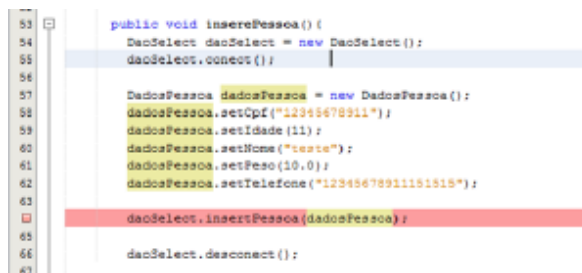


Figura 43 – Novo *breakpoint* para *inserePessoa()*

Fonte: NetBeans (2022)

Agora, com o *breakpoint* colocado no lugar correto, clique no botão **Stop** e entre em modo *debug* novamente.

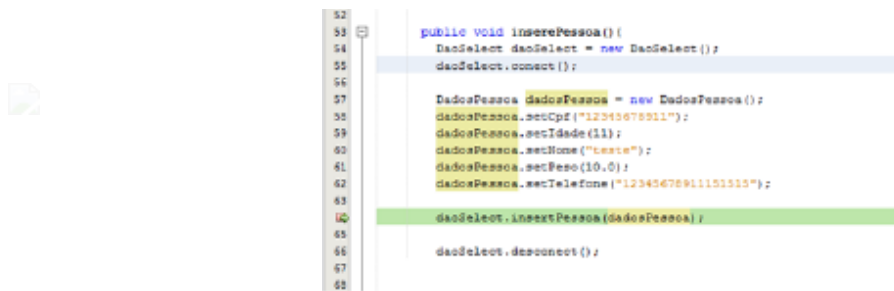


Figura 44 – Reiniciando a depuração

Fonte: NetBeans (2022)

Pronto, agora confira os valores, deixando o *mouse* sobre o objeto “pessoa” para conferir os valores atribuídos ao objeto e ver se correspondem aos do banco.



Figura 45 – Detalhes do objeto **dadosPessoa**

Fonte: NetBeans (2022)

Abra o objeto para ver os valores clicando no ícone de mais, localizado no canto esquerdo do objeto:

Nome	Tipo	Valor
dadosPessoa	DadosPessoa	br.com.model.DadosPessoa@4786202b
nome	String	teste
idade	Integer	11
cpf	String	12345678911
peso	Double	10.0
telefone	String	12345678911151515

Figura 46 – Detalhes do objeto **dadosPessoa**

Fonte: NetBeans (2022)

Analisando os dados com seus valores na coluna “valor”, perceba que o valor atribuído para o atributo telefone aparentemente está muito grande. Compare com o campo que está no banco para ver se há inconsistência.

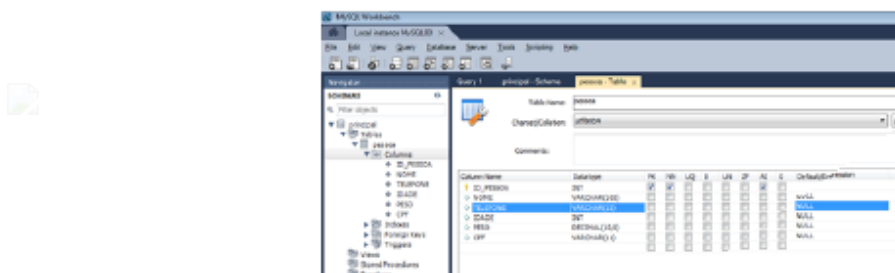


Figura 47 – Detalhes da tabela “pessoa”

Fonte: MySQL Workbench (2022)

Realmente há um problema com o tamanho do telefone, pois o campo está definido como **varchar** com tamanho máximo de 15, ou seja, qualquer valor acima disso não será aceito pelo banco, que retornará o erro *truncation*.

Auxílio à depuração em JPA

Quando se trabalha com JPA (Java Persistence API), frequentemente algumas falhas podem ser difíceis de identificar por conta da natureza do *framework*, que forma uma camada intermediária entre a aplicação e o banco de dados. Nos bastidores, cada operação de dados da JPA é uma operação em SQL, e conhecer quais *scripts* estão de fato sendo executados no banco pode ajudar você em vários problemas.

Existe uma configuração que permite que essa informação seja mostrada ao desenvolvedor. No arquivo de configuração **src/main/resources/META-INF/persistence.xml**, você pode incluir as seguintes linhas entre as tags **<properties>** e **</properties>** para ativar essa depuração:

```
<property name="hibernate.show_sql" value="true" />
<property name="hibernate.format_sql" value="true" />
```

Uma vez configurada, ao executar uma operação de banco de dados, a JPA mostrará na aba **Output** qual comando SQL foi usado.

Por exemplo, considere a seguinte consulta JPQL (Java Persistence Query Language):

Query consulta = em.createQuery("SELECT d FROM Despesa d WHERE d.descricao LIKE '%me%'");

Essa consulta, assim que for executada, gerará a seguinte saída na aba **Output**:

```
Hibernate:
select
  d1_0.id,
  d1_0.data,
  d1_0.descricao,
  d1_0.valor
from
  Despesa d1_0
where
  d1_0.descricao like '%me'
```

Essa informação, aliada ao uso da depuração do NetBeans, pode ser bastante valiosa para detecção e correção de erros.

Build e dependências

Depois que você passou por todo o processo de *debug*, encontrou as possíveis falhas sistêmicas e corrigiu tudo o que precisava ser corrigido, chegou a hora de construir o seu projeto, ou seja, tornar ele um arquivo executável para que possa ser usado em outros computadores/servidores, a fim de que sua aplicação rode como um sistema de fato. Tendo certeza de que seu sistema está com tudo funcionando, de que não há nenhum erro e de que todos os testes foram realizados, use a ferramenta de *build* do projeto, que fica no menu superior do NetBeans, conforme a seguinte imagem:

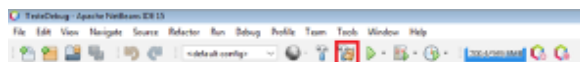
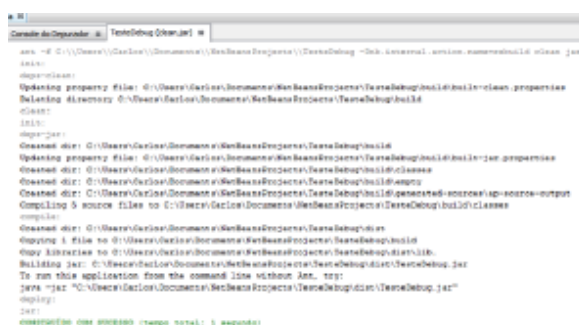


Figura 48 – Ícone **Clean and Build** para limpar e reconstruir a aplicação
Fonte: NetBeans (2022)

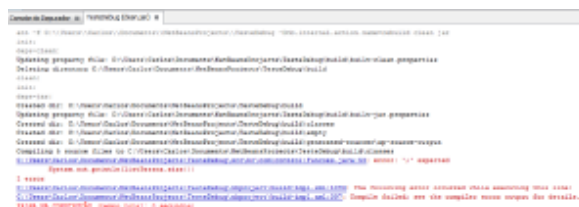
Depois de clicar nesse botão de **limpar e construir**, o Java validará seu código-fonte para ver se não existe nenhuma inconsistência. Estando tudo correto, será gerado um arquivo **.jar**, que é um executável do Java. Esse seria o sistema propriamente dito que será executado. Todo código-fonte está encapsulado dentro dele e pronto para ser usado por qualquer computador, já que o Java é multiplataforma e não fica preso a sistema operacional (SO).



```
ant -f C:\Users\Gustavo\Documents\NetBeansProjects\IwreDebug\build\build.xml clean -jar
[OK]
[INFO]
[INFO]
[INFO] Updating property file: C:\Users\Gustavo\Documents\NetBeansProjects\IwreDebug\build\build.xml
[INFO] Building directory: C:\Users\Gustavo\Documents\NetBeansProjects\IwreDebug\build
[INFO]
[INFO]
[INFO]
[INFO] Created dir: C:\Users\Gustavo\Documents\NetBeansProjects\IwreDebug\build
[INFO] Updating property file: C:\Users\Gustavo\Documents\NetBeansProjects\IwreDebug\build\build.xml
[INFO] Created dir: C:\Users\Gustavo\Documents\NetBeansProjects\IwreDebug\build\classes
[INFO] Created dir: C:\Users\Gustavo\Documents\NetBeansProjects\IwreDebug\build\empty
[INFO] Created dir: C:\Users\Gustavo\Documents\NetBeansProjects\IwreDebug\build\generated-sources\api-source-output
[INFO] Compiling 5 source files to C:\Users\Gustavo\Documents\NetBeansProjects\IwreDebug\build\classes
[INFO]
[INFO] Created dir: C:\Users\Gustavo\Documents\NetBeansProjects\IwreDebug\build
[INFO] Copying 5 files to C:\Users\Gustavo\Documents\NetBeansProjects\IwreDebug\build
[INFO] Copy 5 files to C:\Users\Gustavo\Documents\NetBeansProjects\IwreDebug\build
[INFO] Building jar: C:\Users\Gustavo\Documents\NetBeansProjects\IwreDebug\build\IwreDebug.jar
[INFO] To run this application from the command line without Ant, try:
[INFO] java -jar "C:\Users\Gustavo\Documents\NetBeansProjects\IwreDebug\build\IwreDebug.jar"
[INFO]
[INFO]
[INFO] BUILD SUCCESSFUL (total time: 1 second)
```

Figura 49 – Log do processo da construção
Fonte: NetBeans (2022)

Esse código que aparece no console funciona como se fosse um *log*. Depois que você clica no botão para construir o projeto, são mostradas as etapas do que está sendo feito e, se tudo estiver correto, aparecerá a mensagem “Construído com sucesso” em verde. Se houver algum erro, aparecerá uma mensagem em vermelho, informando que não se conseguiu construir o projeto e indicando o motivo, como no exemplo a seguir:



```
ant -f C:\Users\Gustavo\Documents\NetBeansProjects\IwreDebug\build\build.xml clean -jar
[OK]
[INFO]
[INFO]
[INFO] Updating property file: C:\Users\Gustavo\Documents\NetBeansProjects\IwreDebug\build\build.xml
[INFO] Building directory: C:\Users\Gustavo\Documents\NetBeansProjects\IwreDebug\build
[INFO]
[INFO]
[INFO]
[INFO] Created dir: C:\Users\Gustavo\Documents\NetBeansProjects\IwreDebug\build
[INFO] Updating property file: C:\Users\Gustavo\Documents\NetBeansProjects\IwreDebug\build\build.xml
[INFO] Created dir: C:\Users\Gustavo\Documents\NetBeansProjects\IwreDebug\build\classes
[INFO] Created dir: C:\Users\Gustavo\Documents\NetBeansProjects\IwreDebug\build\empty
[INFO] Created dir: C:\Users\Gustavo\Documents\NetBeansProjects\IwreDebug\build\generated-sources\api-source-output
[INFO] Compiling 5 source files to C:\Users\Gustavo\Documents\NetBeansProjects\IwreDebug\build\classes
[INFO]
[INFO] Created dir: C:\Users\Gustavo\Documents\NetBeansProjects\IwreDebug\build
[INFO] Copying 5 files to C:\Users\Gustavo\Documents\NetBeansProjects\IwreDebug\build
[INFO] Copy 5 files to C:\Users\Gustavo\Documents\NetBeansProjects\IwreDebug\build
[INFO] Building jar: C:\Users\Gustavo\Documents\NetBeansProjects\IwreDebug\build\IwreDebug.jar
[INFO] To run this application from the command line without Ant, try:
[INFO] java -jar "C:\Users\Gustavo\Documents\NetBeansProjects\IwreDebug\build\IwreDebug.jar"
[INFO]
[INFO]
[INFO] BUILD FAILED (total time: 1 second)
[ERROR] Error: Main method not found in class IwreDebug, please define the main method as static in IwreDebug.java
```

Figura 50 – Log do processo da construção
Fonte: NetBeans (2022)

Nesse caso, a JVM (Java Virtual Machine) encontrou inconsistências no código-fonte e não permitiu a construção do projeto. É preciso verificar o que pode estar acontecendo de errado e corrigir o problema, para depois construir novamente. O *debug* pode ser um forte aliado neste momento, pois ele pode procurar calmamente o erro até encontrá-lo. Depois que tudo estiver correto e o projeto foi construído com sucesso, pode-se localizar o arquivo JAR construído dentro da pasta **dist** do projeto – use um navegador de arquivos para localizar essa pasta.

A pasta **dist** é o local no qual a IDE constrói o projeto e cria o arquivo **.jar**. Toda vez que o programa é construído, um novo arquivo **.jar** é gerado com as respectivas alterações incluídas, pois geralmente, depois de qualquer alteração, o sistema é construído novamente para aplicar as alterações feitas. Note que também é criada uma pasta **lib** para conter as dependências do projeto (como a biblioteca MySQL Connector J, por exemplo).

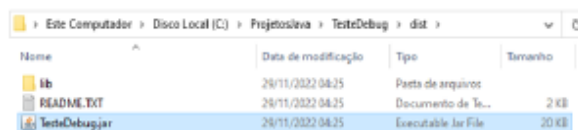


Figura 51 – Pasta de construção do arquivo executável

Fonte: NetBeans (2022)

Logo, sempre que for instalar um programa em outro computador, basta você levar a pasta do projeto atualizada, colocar em um local seguro do computador e puxar um atalho do arquivo **.jar** na raiz da pasta **dist**. Assim, seu sistema já estará funcionando.

Documentação



Algumas pessoas dizem que escrever a documentação do programa é como escrever muitos comentários no código. Isso está errado! Uma boa documentação não sobrecarrega o código com comentários. Uma boa documentação limita-se a explicar o que cada função do programa faz. Uma boa documentação não perde tempo tentando explicar como funcionam as funções, pois os leitores interessados no problema podem ler o código.

A empresa de *courier* promete pegar um pacote em São Paulo e entregá-lo em Manaus. É isso que as empresas fazem. Como o serviço será executado, por exemplo, se o transporte será por terra, ar ou mar, é um assunto interno da empresa.

Resumindo, a documentação de uma função é um minimanual que fornece instruções completas sobre como usar a função corretamente (nesse sentido, o conceito de documentação se confunde com o conceito de API (*application programming interface*)). Esse minimanual deve explicar o que a função recebe e o que ela retorna. Então, tem que dizer com muita precisão que efeito a função produz, ou seja, qual é a relação entre o que a função recebe e o que ela retorna. A documentação adequada é uma questão de honestidade intelectual, pois capacita os leitores/usuários a verificarem e provarem que a funcionalidade está errada (desde que esteja).



O NetBeans conta também com um recurso relacionado à documentação, porém, antes de usá-lo, é necessário comentar o código primeiro, não somente citando as partes mais importantes do código, mas classificando essas partes.

Veja o exemplo de um comentário simples em um programa em Java:

```
/** Comentário simples em Java – TDS */
```

Coloque esse comentário nos pontos mais importantes do seu código e lembre-se de que qualquer pessoa deve ter facilidade para ler seu código.

Porém, também existem os comentários com *tags*, que vão ao final de tudo, contribuindo para a documentação padrão do NetBeans. A *tag* referencia a informação contida no comentário. Confira um exemplo:

```
/**  
 * Comentário simples em Java – TDS  
 * @author Aluno  
 * @version 1.0  
 * @since Primeira versão  
 */
```

Recomenda-se que você atribua comentários por classe, como na imagem anterior, mas, para cada classe, deixe uma especificação sobre ela. Isso, além de facilitar a leitura do código, fará o NetBeans entender melhor a documentação do seu código para que, ao final, você use o seguinte recurso:

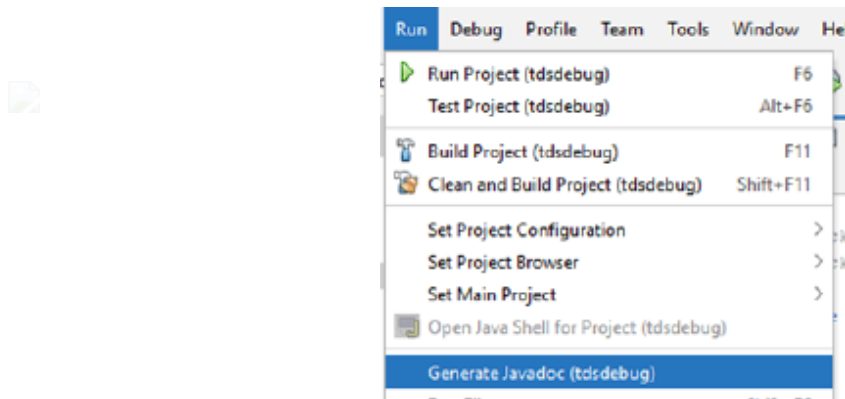


Figura 52 – Funcionalidade **Generate Javadoc** do NetBeans

Fonte: NetBeans (2022)

Com o **Generate Javadoc**, seu NetBeans gerará uma página simples de documentação do seu programa (o que pode ser demorado).

Encerramento



Com este conteúdo, você aprendeu maneiras preciosas de descobrir os erros do seu programa e gerar *build* dele para que possa ser instalado em algum cliente. Além disso, você também entendeu um pouco a documentação de *software* e documentação com NetBeans. Utilize este conhecimento durante seu curso, pois ele lhe será extremamente útil.