Aluno: Arthur Nunes de Castro Oliveira

Detalhes do refatoramento

PRINCIPAIS CASOS:

- DUPLICATED CODE:

Na classe Histórico, percebe-se que os métodos **listarHistoricoGeral** e **listarHistoricoCliente** realizam operações idênticas, como pode ser observado na figura abaixo:

```
29    //printar histórico geral
30    public static void listarHistoricoGeral() {
31        for (int i = listaPedidoGeral.size() - 1; i >= 0; i--) {
32            System.out.println("\n-------\nCliente:  " + listaPedidoGeral.get(i).getNomeCliente() + "\nData: "
33                    + listaPedidoGeral.get(i).getData() + "\nComida: " + listaPedidoGeral.get(i).getComida().getNome()
34                    + "\nBebida: " + listaPedidoGeral.get(i).getBebida().getNome() + "\nValor: R$"
35                    + listaPedidoGeral.get(i).getCusto() + "\n-------\n");
36        }
37    }
38
39    // printar histórico do cliente
40    public void listarHistoricoCliente() {
41        for (int i = listaPedidoCliente.size() - 1; i >= 0; i--) {
42            System.out.println("\n-------\nCliente:  " + this.listaPedidoCliente.get(i).getNomeCliente() + "\nData: "
43                    + this.listaPedidoCliente.get(i).getData() + "\nComida: "
44                    + this.listaPedidoCliente.get(i).getComida().getNome() + "\nBebida: "
45                    + this.listaPedidoCliente.get(i).getBebida().getNome() + "\nValor: R$"
46                    + this.listaPedidoCliente.get(i).getCusto() + "\n-------\n");
47        }
48    }
49 }
```

Como solução para isso, foi elaborado um método geral que pudesse imprimir tanto o histórico específico de um determinado cliente, como também, listar o histórico de todos os pedidos atendidos pela lanchonete.

```
19    // listar historico
20    public static void listarHistorico(ArrayList<Pedido> historico) {
21        for (int i = historico.size() - 1; i >= 0; i--) {
22            System.out.println("\n-------\nCliente:  " + historico.get(i).getNomeCliente() + "\nData: "
23                    + historico.get(i).getData() + "\nComida: " + historico.get(i).getComida().getNome() + "\nBebida: "
24                    + historico.get(i).getBebida().getNome() + "\nValor: R$" + historico.get(i).getCusto()
25                    + "\n-------\n");
26        }
27    }
```

-**DUPLICATED CODE E EXCESSO DE ESTRUTURAS CONDICIONAIS:**

Nas classes **AcoesAdm** e **AcoesFuncionario** existe um método em comum que é exatamente idêntico, que é o **método exibirCardapio** como mostra a figura abaixo:

```
86  public void exibirCardapio() {
87     System.out.println("\n>>COMIDAS DISPONÍVEIS:");
88
89     if(BancoDados.listaComidas.isEmpty())
90        System.out.println(">>Nenhuma comida cadastrada no momento!");
91     else {
92        for(int i=0; i<BancoDados.listaComidas.size(); i++) {
93           System.out.println("-------\nNome: " + BancoDados.listaComidas.get(i).getNome() +
94                 "\nValor: R$ " + BancoDados.listaComidas.get(i).getValor()+
95                 "\nFrequência: " + BancoDados.listaComidas.get(i).getFrequencia());
96        }
97     }
98
99     System.out.println("\n>>BEBIDAS DISPONÍVEIS:");
100
101    if(BancoDados.listaBebidas.isEmpty())
102       System.out.println(">>Nenhuma bebida cadastrada no momento!");
103    else {
104       for(int i=0; i<BancoDados.listaBebidas.size(); i++) {
105          System.out.println("-------\nNome: " + BancoDados.listaBebidas.get(i).getNome() +
106                "\nValor: R$ " + BancoDados.listaBebidas.get(i).getValor() +
107             "\nFrequência: " + BancoDados.listaBebidas.get(i).getFrequencia());
108       }
109    }
110 }
111 }
```

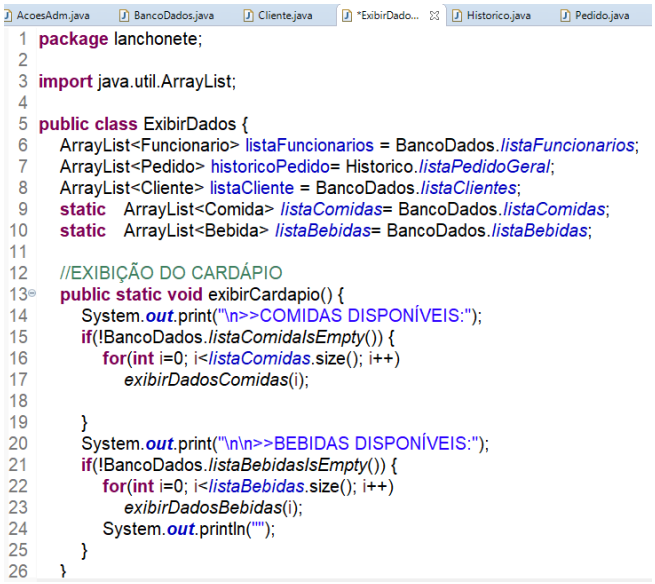
A lógica de execução do método foi alterada pensando em reduzir a quantidade de (if e else), como também foi utilizado o extrair método pensando em torna-lo mais enxuto. Esse método foi implementado em uma nova classe chamada ExibirDados.

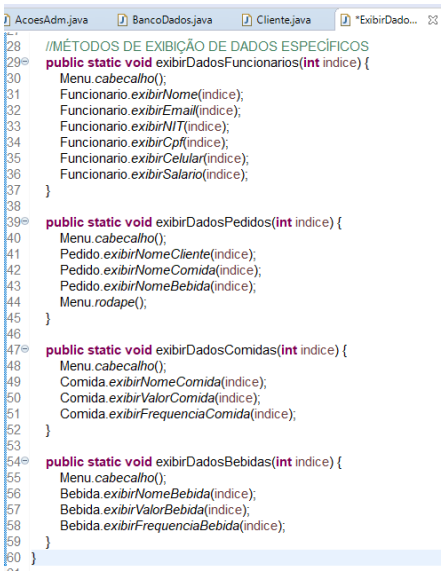
```
113  public void exibirCardapio() {
114     System.out.print("\n>>COMIDAS DISPONÍVEIS:");
115     if(!BancoDados.listaComidaIsEmpty()) {
116        for(int i=0; i<listaComidas.size(); i++)
117           BancoDados.exibirDadosComidas(i);
118
119     }
120     System.out.print("\n\n>>BEBIDAS DISPONÍVEIS:");
121     if(!BancoDados.listaBebidasIsEmpty()) {
122        for(int i=0; i<listaBebidas.size(); i++)
123           BancoDados.exibirDadosBebidas(i);
124        System.out.println("");
125     }
126  }
127 }
```

- LARGER CLASS

Nas classes **AcoesAdm** e **AcoesFuncionario** percebe-se também diversos métodos que fazem dentro de sua estrutura a exibição de dados relativos ao cardápio, funcionários, pedidos, comidas, bebidas etc. Pensando melhor, esses tipos de ações são auxiliares, não sendo de responsabilidade das classes **AcoesAdm e AcoesFuncionario**. Pensando em diminuir a responsabilidade dessas classes com relação a isso, foi criada uma nova classe chamada **ExibirDados** nos quais constam os métodos: exibirCardápios, exibirDadosPedidos, exibirDadosFuncionarios, exibirDadosComidas, exibirDadosBebidas.

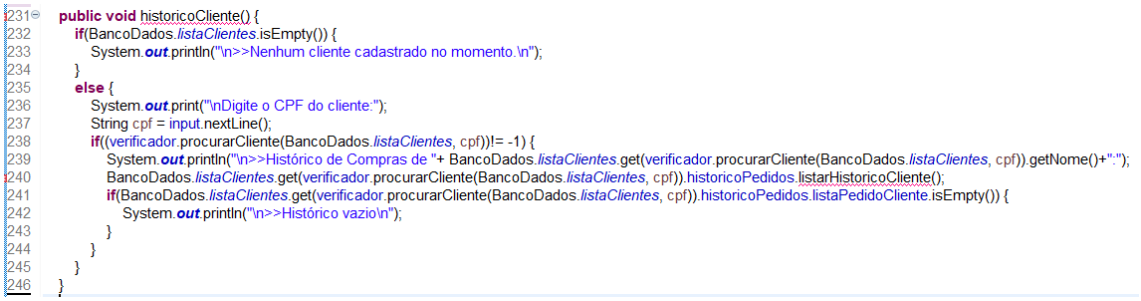
Obs: Não coloquei imagem antes do refatoramento desse smell porque as ações de exibição de dados ocorrem em vários pontos do código nas classes AcoesAdm e AcoesFuncionarios, desta forma, para melhor observação é melhor comparar as classes diretamente pelo código.



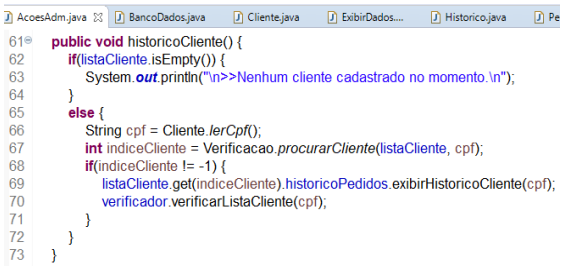


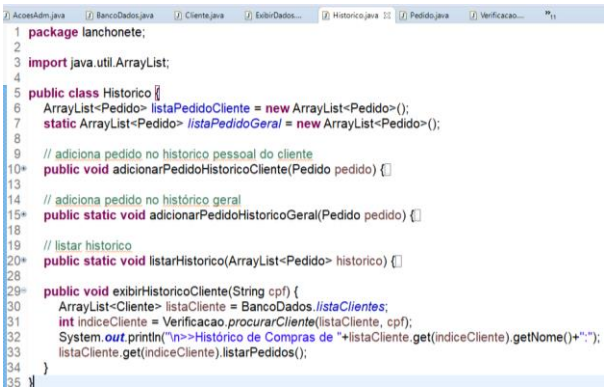
-LONG PARAMETER LIST:

Na classe **acoesAdm**, no método **historicoCliente** percebe-se a utilização de vários ifs e else como também longos parâmetros



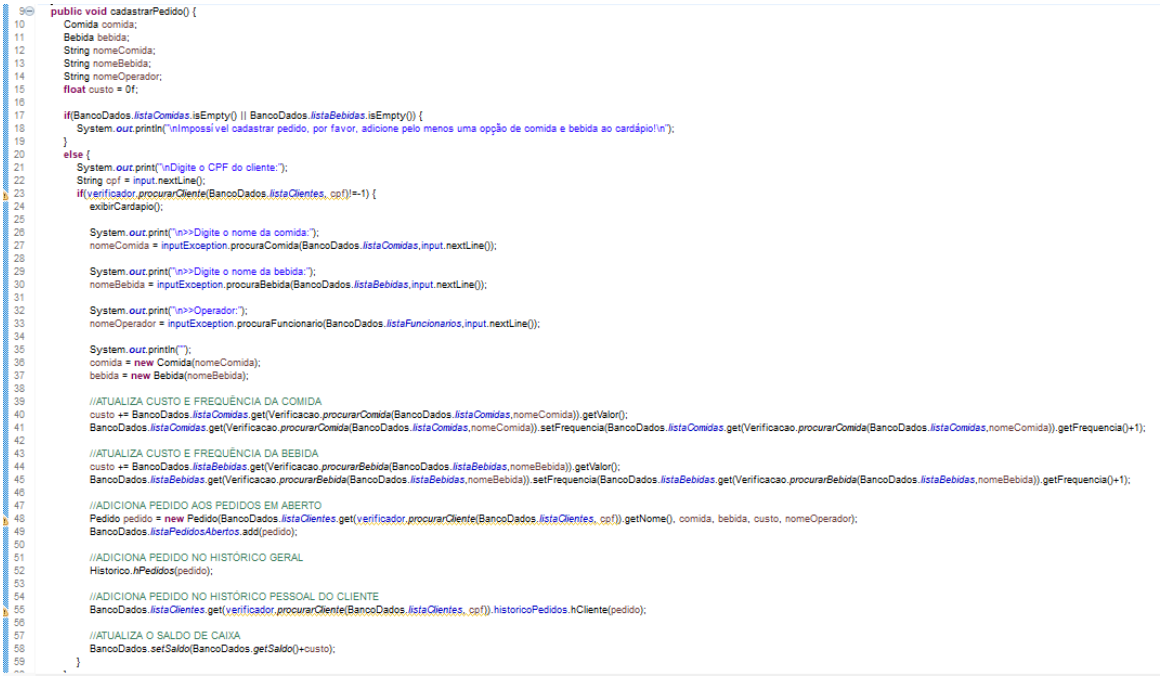
Foi utilizado o extrair método e parte das responsabilidades desse método **hitoricoCliente** foi passada para a **classe Cliente** e um novo método criado na classe **Histórico** chamado **exibirHistóricoCliente** que é responsável de procurar o cliente em específico, printar o cabeçalho e chamar o método **listarHistórico** por meio da classe cliente.





-LONG METHOD E LONG PARAMETER LIST:

Na classe **acoesFuncionario** percebe-se que o **método cadastrarPedido** está muito complexo, com longos parâmetros e fazendo mais o que deveria, uma vez que o método cadastrar pedido está obtendo os nomes da comida e bebida, atualizando o saldo de caixa, atualizando a frequência da comida e bebida, adicionando o pedido na lista de pedidos em aberto, adicionando pedido aos históricos etc..... como pode ser observado na imagem abaixo:



Para resolver esse problema, foi criada a classe **Cadastro** que é responsável por conter os métodos responsáveis por adicionar ou alterar informações relativos aos pedidos, funcionários ou cardápio. Parte da responsabilidade do método foram passadas para outros métodos localizados na **classe BancoDados**. O método mesmo depois da refatoração continuou um pouco extenso, porém, ficou muito mais intuitivo, como pode ser observado na imagem a seguir:

```
80    public void cadastrarPedido() {
81        Comida comida;
82        Bebida bebida;
83
84        if(!BancoDados.lanchesIsEmpty()) {
85            String cpf = Cliente.lerCpf();
86            int indiceCliente = Cliente.getIndice(cpf);
87            if(indiceCliente!=-1) {
88                AcoesAdm.exibirCardapio();
89                String nomeComida =  Comida.lerNomeComida(listaComidas);
90                int indiceComida = Verificacao.procurarComida(listaComidas,nomeComida);
91                String nomeBebida = Bebida.lerNomeBebida(listaBebidas);
92                int indiceBebida = Verificacao.procurarBebida(listaBebidas,nomeBebida);
93                String nomeOperador = Funcionario.lerNomeFuncionario(listaFuncionarios);
94                comida = new Comida(nomeComida);
95                bebida = new Bebida(nomeBebida);
96                System.out.println("");
97                BancoDados.atualizarFrequenciaComida(indiceComida);
98                BancoDados.atualizarFrequenciaBebida(indiceBebida);
99                float custo = BancoDados.atualizarSaldo(indiceComida, indiceBebida);
100               Pedido pedido = new Pedido(listaClientes.get(indiceCliente).getNome(), comida, bebida, custo, nomeOperador);
101               BancoDados.adicionarPedido(pedido);
102               Historico.adicionarPedidoHistoricoGeral(pedido);
103               BancoDados.adicionarPedidoHistoricoCliente(pedido,indiceCliente);
104           }
105
106       }
107   }
108 }
```

- EXCESSO DE ESTRUTURAS CONDICIONAIS E (LAZY CLASS DA CLASSE FUNCIONARIO)

Na classe **AcoesAdm** o **método alterarDadosFuncionario** possui um uso excessivo de estruturas condicionais como ifs e switch case tornando o código menos intuitivo.

```
47    public void alterarDadosFuncionario() {
48        if(BancoDados.listaFuncionarios.isEmpty()) {
49            System.out.println("\n>>Não possui funcionários\n");
50        }
51        else{
52            int opcao;
53            System.out.println("\n(1) Alterar Salário");
54            System.out.println("(2) Alterar E-mail");
55            System.out.println("(3) Alterar Número de Contato");
56            opcao = inputException.inputMenu(1,3);
57
58            switch (opcao) {
59            case 1:
60                System.out.print("\nDigite o nome do funcionário:");
61                String nome = input.nextLine();
62                if((verificador.procurarFuncionario(BancoDados.listaFuncionarios, nome))!= -1) {
63                    System.out.print("Digite o novo salário: R$");
64                    BancoDados.listaFuncionarios.get(verificador.procurarFuncionario(BancoDados.listaFuncionarios, nome)).setSalario(inputExceptionexceptionFloat());
65                    System.out.print(">>Salário alterado com sucesso!\n");
66                }
67                break;
68            case 2:
69                System.out.print("\nDigite o nome do funcionário:");
70                String nome2 = input.nextLine();
71                if((verificador.procurarFuncionario(BancoDados.listaFuncionarios, nome2))!= -1) {
72                    System.out.print("Digite o novo E-mail: ");
73                    BancoDados.listaFuncionarios.get(verificador.procurarFuncionario(BancoDados.listaFuncionarios, nome2)).setEmail(input.nextLine());
74                    System.out.print(">>E-mail alterado com sucesso!\n");
75                }
76                break;
77            case 3:
78                System.out.print("\nDigite o nome do funcionário:");
79                String nome3 = input.nextLine();
80                if((verificador.procurarFuncionario(BancoDados.listaFuncionarios, nome3))!= -1) {
81                    System.out.print("Digite um número de contato: ");
82                    BancoDados.listaFuncionarios.get(verificador.procurarFuncionario(BancoDados.listaFuncionarios, nome3)).setCelular(input.nextLine());
83                    System.out.print(">>Número alterado com sucesso!\n");
84                }
85                break;
86            }
87
88        }
89    }
```

O método alterarDadosFuncionario é responsável por alterar o **salário, email ou número de contato** a depender da entrada fornecida. Pensando nisso, foram extraídos os códigos de cada case do switch e foram criados os métodos: **alterarSalario, alterarEmail, alterarCelular**, **que foram implementados na classe Funcionário** que por sua vez, esta classe possuía nada além de atributos e métodos getter e setters, ou seja, **ela possuía o**

smell LAZY CLASS, desta forma ela ganha um pouco mais de responsabilidades. Também ainda na classe Funcionário, foi criado um método chamado **alterarDados**, ele é responsável, a depender do parâmetro de entrada, de chamar os métodos **alterarSalario, alterarEmail, alterarCelular,** sendo um pouco semelhante ao padrão **factory method**, onde eu tenho um método, onde a depender do parâmetro passado, ele chama automaticamente o método responsável por aquela ação desejada, enxugando ainda mais meu "método cliente" da classe AcaoAdm. A seguir, segue o código depois do refatoramento:

```
49  public  void alterarDados(int opcao) {
50     int indiceFuncionario = getIndice();
51     if(indiceFuncionario!= -1) {
52        switch(opcao) {
53        case 1:
54           alterarSalario(indiceFuncionario);
55           break;
56        case 2:
57           alterarEmail(indiceFuncionario);
58           break;
59        case 3:
60           alterarCelular(indiceFuncionario);
61           break;
62        }
63     }
64  }
65
66  private int getIndice() {
67     return verificador.procurarFuncionario(listaFuncionarios, lerNomeFuncionario());
68  }
```

```
28  private void alterarSalario(int indiceFuncionario) {
29     System.out.print("Digite o novo salário: R$");
30     float salario = inputException.exceptionFloat();
31     listaFuncionarios.get(indiceFuncionario).setSalario(salario);
32     System.out.print(">>Salário alterado com sucesso!\n");
33  }
34
35  private  void alterarEmail(int indiceFuncionario) {
36     System.out.print("Digite o novo E-mail: ");
37     String email = input.nextLine();
38     listaFuncionarios.get(indiceFuncionario).setEmail(email);
39     System.out.print(">>E-mail alterado com sucesso!\n");
40  }
41
42  private  void alterarCelular(int indiceFuncionario) {
43     System.out.print("Digite um número de contato: ");
44     String celular = input.nextLine() ;
45     listaFuncionarios.get(indiceFuncionario).setCelular(celular);
46     System.out.print(">>Número alterado com sucesso!\n");
47  }
```

Os métodos **removerLanche e alterarValorLanche** da classe AcoesAdm também foram refatorados seguindo os passos descritos anteriormente.

OUTROS REFATORAMENTOS:

- UTILIZAÇÃO DO PADRÃO EXTRAIR MÉTODO E ENCAPSULAMENTO DE COLEÇÃO

Na classe **AcoesAdm** percebe-se que o método **addFuncionario** poderia ser simplificado:

```
10  public void addFuncionario() {
11
12     System.out.print("\nNome:");
13     String nome = input.nextLine();
14
15     System.out.print("CPF:");
16     String  cpf = input.nextLine();
17
18     System.out.print("NIT: ");
19     String  nit = input.nextLine();
20
21     System.out.print("Celular:");
22     String  celular = input.nextLine();
23
24     System.out.print("E-mail:");
25     String  email = input.nextLine();
26
27     System.out.print("Endereço:");
28     String endereco = input.nextLine();
29
30     System.out.print("Salário: R$");
31     float salario = inputException.exceptionFloat();
32
33     Funcionario funcionario = new Funcionario(nome,cpf,celular,email,endereco,salario,nit);
34     BancoDados.listaFuncionarios.add(funcionario);
35
36     System.out.println("\n>>Funcionário cadastrado com sucesso!\n");
37  }
```

Foi criada uma nova classe chamda Cadastro, nela foi implementado um método para ser usado para obtenção de dados de novos funcionarios:

```
11  public void addFuncionario() {
12      Funcionario funcionario = cadastro.cadastrarFuncionario();
13      BancoDados.listaFuncionarios.add(funcionario);
14      System.out.println("\n>>Funcionário cadastrado com sucesso!\n");
15  }
```

Ainda sobre o método acima, podemos realizar outro padrão de refatoramento que é o **encapsular coleção**, então na classe **BancoDados** foi criado o método **adicionarFuncionarios ().**

```
14  public void addFuncionario() {
15      Funcionario funcionario = cadastro.cadastrarFuncionario();
16      BancoDados.adicionarFuncionarios(funcionario);
17  }
18
```

O método ***remover funcionario*** da classe ***acoesAdm*** também foi refatorado de modo a torna-lo menor e mais intuitivo, com menores parâmetros e menos estruturas condicionais, como pode ser observado nos códigos abaixo:

Antes:

```
33  public void removeFuncionario() {
34      if(BancoDados.listaFuncionarios.isEmpty()) {
35          System.out.println("\n>>Não possui funcionários\n");
36      }
37      else{
38          System.out.print("\nDigite o nome do funcionário que irá ser demitido:");
39          String nome = input.nextLine();
40          if((verificador.procurarFuncionario(BancoDados.listaFuncionarios, nome))!= -1) {
41              System.out.println("Funcionário demitido com sucesso!");
42              BancoDados.listaFuncionarios.remove(verificador.procurarFuncionario(BancoDados.listaFuncionarios, nome));
43              System.out.println("Total de funcionários: " + BancoDados.listaFuncionarios.size() + "\n");
44          }
45      }
46  }
```

Depois:

```
23  public void removeFuncionario() {
24      if(!BancoDados.listaFuncionarioIsEmpty()) {
25          BancoDados.removerFuncionarios();
26          BancoDados.totalFuncionarios();
27      }
28  }
```

- UTILIZAÇÃO DO EXTRAIR MÉTODO E FACTORY METHOD

Nos métodos adicionarLanche, removerLanche e alterarValorLanche da classe AcoesAdm e cadastrarCliente da classe AcoesFuncionario:

Método adicionarLanche():

Antes: Na classe AcoesAdm

```
97  public void addLanche() {
98      System.out.println("\n(1) Adicionar Comida");
99      System.out.println("(2) Adicionar Bebida");
100     int opcao = inputException.inputMenu(1,2);
101
102     switch(opcao) {
103     case 1:
104         System.out.print(">>Nome da comida: ");
105         String nomeComida = input.nextLine();
106         System.out.print(">>Valor: R$");
107         float valor = inputException.exceptionFloat();
108         Comida lanche = new Comida(nomeComida, valor);
109         BancoDados.listaComidas.add(lanche);
110         System.out.println("");
111         break;
112     case 2:
113         System.out.print(">>Nome da bebida: ");
114         String nomeBebida = input.nextLine();
115         System.out.print(">>Valor: R$");
116         float valorBebida = inputException.exceptionFloat();
117         Bebida bebida = new Bebida(nomeBebida, valorBebida);
118         BancoDados.listaBebidas.add(bebida);
119         System.out.println("");
120         break;
121     }
122 }
```

Depois:

Na classe AcoesAdm:

```
37  public void addLanche() {
38      int opcao = exibir.menuCadastroLanche();
39      cadastro.cadastrarLanche(opcao);
40  }
```

Na classe Cadastro:

```
76  public void cadastrarLanche(int opcao) {
77      switch(opcao) {
78      case 1:
79          cadastrarComida();
80          break;
81      case 2:
82          cadastrarBebida();
83          break;
84      }
85  }
```

```
58  private void cadastrarComida() {
59      System.out.print(">>Nome da comida: ");
60      String nomeComida = input.nextLine();
61
62      System.out.print(">>Valor: R$");
63      float valor = inputException.exceptionFloat();
64      BancoDados.addComida(new Comida(nomeComida, valor));
65  }
66
67  private void cadastrarBebida() {
68      System.out.print(">>Nome da bebida: ");
69      String nomeBebida = input.nextLine();
70
71      System.out.print(">>Valor: R$");
72      float valorBebida = inputException.exceptionFloat();
73      BancoDados.addBebida(new Bebida(nomeBebida, valorBebida));
74  }
75
```

Remover lanche:

Antes: Na classe AcoesAdm

```
24  public void removeLanche() {
25     System.out.println("\n(1) Remover Comida");
26     System.out.println("(2) Remover Bebida");
27     int opcao = inputException.inputMenu(1,2);
28
29     switch(opcao) {
30     case 1:
31        if(BancoDados.listaComidas.isEmpty()) {
32           System.out.println("\n>>Não possui comidas cadastradas\n");
33        }
34        else{
35           System.out.print("Digite o nome da comida que irá ser removida:");
36           String nomeComida = input.nextLine();
37           if((verificador.procurarComida(BancoDados.listaComidas, nomeComida))!= -1) {
38              BancoDados.listaComidas.remove(verificador.procurarComida(BancoDados.listaComidas, nomeComida));
39              System.out.println(">>Comida removida com sucesso!");
40              System.out.println(">>Total de opções de comida: " + BancoDados.listaComidas.size() + "\n");
41           }
42        }
43        break;
44     case 2:
45        if(BancoDados.listaBebidas.isEmpty()) {
46           System.out.println("\n>>Não possui bebidas cadastradas\n");
47        }
48        else{
49           System.out.print(">>Digite o nome da bebida que irá ser removida:");
50           String nomeBebida = input.nextLine();
51           if((verificador.procurarBebida(BancoDados.listaBebidas, nomeBebida))!= -1) {
52              BancoDados.listaBebidas.remove(verificador.procurarBebida(BancoDados.listaBebidas, nomeBebida));
53              System.out.println(">>Bebida removida com sucesso!");
54              System.out.println(">>Total de opções de bebidas: " + BancoDados.listaBebidas.size() + "\n");
55           }
56        }
57        break;
58     }
59  }
60
```

Depois:

```
43     public void removeLanche() {
44        int opcao = exibir.menuCadastroLanche("Remover");
45        BancoDados.removerLanche(opcao);
46     }
47
```

Alterar valor do lanche:

Antes: Na classe AcoesAdm

```
161 public void valorLanche() {
162    System.out.println("\n(1) Alterar valor da Comida");
163    System.out.println("(2) Alterar valor da Bebida");
164    int opcao = inputException.inputMenu(1,2);
165
166    switch(opcao) {
167    case 1:
168       if(BancoDados.listaComidas.isEmpty()) {
169          System.out.println("\n>>Não possui comidas cadastradas\n");
170       }
171       else{
172          System.out.print(">>Digite o nome da comida:");
173          String nomeComida = input.nextLine();
174          if((verificador.procurarComida(BancoDados.listaComidas, nomeComida))!= -1) {
175             System.out.print(">>Digite o novo valor: R$");
176             BancoDados.listaComidas.get(verificador.procurarComida(BancoDados.listaComidas, nomeComida)).setValor(inputException.exceptionFloat());
177             System.out.println(">>Valor alterado com sucesso!\n");
178          }
179       }
180       break;
181    case 2:
182       if(BancoDados.listaBebidas.isEmpty()) {
183          System.out.println(">>Não possui bebidas cadastradas\n");
184       }
185       else{
186          System.out.print("Digite o nome da bebida:");
187          String nomeBebida = input.nextLine();
188          if((verificador.procurarBebida(BancoDados.listaBebidas, nomeBebida))!= -1) {
189             System.out.print("Digite o novo valor: R$");
190             BancoDados.listaBebidas.get(verificador.procurarBebida(BancoDados.listaBebidas, nomeBebida)).setValor(inputException.exceptionFloat());
191             System.out.println("Valor alterado com sucesso!\n");
192          }
193       }
194       break;
195    }
196 }
197
```

Depois:

-Na classe AcoesAdm

```
48    public void alterarValorLanche() {
49        int opcao = exibir.menuAlteracaoValorLanche();
50        cadastro.alterarValorLanche(opcao);
51    }
```

Na classe Cadastro:

```
27    public static void alterarValorBebida() {
28        if(!BancoDados.listaBebidasIsEmpty()) {
29            int indiceBebida = getIndice();
30            if(indiceBebida!= -1) {
31                System.out.print(">>Digite o novo valor: R$");
32                listaBebidas.get(indiceBebida).setValor(inputException.exceptionFloat());
33                System.out.println(">>Valor alterado com sucesso!\n");
34            }
35        }
36    }
```

```
21    public static void alterarValorComida() {
22        if(!BancoDados.listaComidaIsEmpty()) {
23            int indiceComida = getIndice();
24            if(indiceComida!= -1) {
25                System.out.print(">>Digite o novo valor: R$");
26                listaComida.get(indiceComida).setValor(inputException.exceptionFloat());
27                System.out.println(">>Valor alterado com sucesso!\n");
28            }
29        }
30    }
```

```
64    public void alterarValorLanche(int opcao) {
65        switch(opcao) {
66        case 1:
67            Comida.alterarValorComida();
68            break;
69        case 2:
70            Bebida.alterarValorBebida();
71            break;
72        }
73    }
74 }
```

Na método cadastrarCliente da classe acoesFuncionario:

Antes:

```
20    public void cadastrarCliente() {
21        System.out.print("\nNome:");
22        String nome = input.nextLine();
23
24        System.out.print("CPF:");
25        String  cpf = input.nextLine();
26
27        System.out.print("Celular:");
28        String  celular = input.nextLine();
29
30        System.out.print("E-mail:");
31        String  email = input.nextLine();
32
33        System.out.print("Endereço:");
34        String endereco = input.nextLine();
35
36        Cliente cliente = new Cliente (nome, cpf, celular, email, endereco);
37        BancoDados.listaClientes.add(cliente);
38
39        System.out.println("\n>>Cliente cadastrado com sucesso!\n");
40    }
```

Depois:

```
20    public void cadastrarCliente() {
21        Cliente cliente = cadastro.cadastrarCliente();
22        BancoDados.adicionarCliente(cliente);
23    }
```

- EXTRAÇÃO DE MÉTODOS E TREIXOS DE CÓDIGOS DE PRINTAR DADOS (COMIDA, BEBIDAS, PEDIDOS, FUNCIONARIOS) PARA DIMINUIR A RESPONSABILIDADE DA CLASSE ACOESADM E ACOESFUNCIONARIO

No método **listarFuncionarios()** da classe AcoesAdm

ANTES:

```
73  public void listarFuncionarios() {
74     if(BancoDados.listaFuncionarios.isEmpty())
75        System.out.println("\n>>Não possui funcionários cadastrados no momento!\n");
76     else {
77        for(int i=0; i<BancoDados.listaFuncionarios.size(); i++) {
78           System.out.println("-------\nNome: " + BancoDados.listaFuncionarios.get(i).getNome() +
79                    "\nE-mail: " + BancoDados.listaFuncionarios.get(i).getEmail()+
80                    "\nNIT: " + BancoDados.listaFuncionarios.get(i).getNit()+
81                    "\nCpf: " + BancoDados.listaFuncionarios.get(i).getCpf()+
82                    "\nCelular: " + BancoDados.listaFuncionarios.get(i).getCelular()+
83                    "\nSalário:R$ " + BancoDados.listaFuncionarios.get(i).getSalario());
84        }
85        System.out.println("------");
86        System.out.println(">>Total de funcionários: " + BancoDados.listaFuncionarios.size() + "\n");
87     }
88  }
```

DEPOIS:

Na classe AcoesAdm:

```
73  public void listarFuncionarios() {
74     if(!BancoDados.listaFuncionarioIsEmpty()) {
75        for(int i=0; i<listaFuncionarios.size(); i++)      {
76           BancoDados.exibirDadosFuncionarios(i);
77        }
78        BancoDados.totalFuncionarios();
79     }
80  }
```

Na classe Cadastro

```
28  //MÉTODOS DE EXIBIÇÃO DE DADOS ESPECÍFICOS
29  public static void exibirDadosFuncionarios(int indice) {
30     Menu.cabecalho();
31     Funcionario.exibirNome(indice);
32     Funcionario.exibirEmail(indice);
33     Funcionario.exibirNIT(indice);
34     Funcionario.exibirCpf(indice);
35     Funcionario.exibirCelular(indice);
36     Funcionario.exibirSalario(indice);
37  }
```

No método pedidosAbertos() da classe AcoesAdm:

Antes:

```
public void pedidosAbertos() {
   if(BancoDados.listaPedidosAbertos.isEmpty()) {
      System.out.println("\n>>Não temos nenhum pedido em aberto no momento\n");
   }
   else {
      for(int i=0; i<BancoDados.listaPedidosAbertos.size(); i++) {
         System.out.println("-------\nNome: " + BancoDados.listaPedidosAbertos.get(i).getNomeCliente() +
               "\nComida: " + BancoDados.listaPedidosAbertos.get(i).getComida().getNome()+
               "\nBebida: " + BancoDados.listaPedidosAbertos.get(i).getBebida().getNome());
      }
      System.out.println("------");
      System.out.println(">>Total de pedidos em aberto: " + BancoDados.listaPedidosAbertos.size() + "\n");
   }
}
```

Depois:

- Na classe AcoesAdm

```
26    public void pedidosAbertos() {
27       if(!BancoDados.listaPedidosAbertosIsEmpty()) {
28          for(int i=0; i<listaPedidosAbertos.size(); i++) {
29             BancoDados.exibirDadosPedidos(i);
30          }
31       BancoDados.totalPedidosAbertos();
32       }
33    }
```

-Na classe ExibirDados

```
39    public static void exibirDadosPedidos(int indice) {
40       Menu.cabecalho();
41       Pedido.exibirNomeCliente(indice);
42       Pedido.exibirNomeComida(indice);
43       Pedido.exibirNomeBebida(indice);
44       Menu.rodape();
45    }
```

- CRIAÇÃO DE CLASSE NOTA FISCAL, UMA VEZ QUE ELA ESTÁ PROPENSA A MAIOR NÚMERO DE ALTERAÇÕES, RESOLUÇÃO DE LOGO PARÂMETRO

ANTES:

NA CLASSE ACOESADM

```
public void notaFiscal() {
   if(BancoDados.listaPedidosAbertos.isEmpty()) {
      System.out.println("\n>>Não temos nenhum pedido em aberto no momento\n");
   }
   else {
      System.out.print("\n>>Digite o nome do cliente:");
      String nome = input.nextLine();
      if(verificador.procurarPedido(BancoDados.listaPedidosAbertos,nome)!=-1) {
         System.out.println("---------------------");
         System.out.println("COMPROVANTE DE VENDA");
         System.out.println("---------------------");
         System.out.println(">>Produtos:");
         System.out.println("Comida: "+BancoDados.listaPedidosAbertos.get(verificador.procurarPedido(BancoDados.listaPedidosAbertos,nome)).getComida().getNome());
         System.out.println("Bebida: "+BancoDados.listaPedidosAbertos.get(verificador.procurarPedido(BancoDados.listaPedidosAbertos,nome)).getBebida().getNome());
         System.out.println("(Data/Horário) de pagamento: "+BancoDados.listaPedidosAbertos.get(verificador.procurarPedido(BancoDados.listaPedidosAbertos,nome)).getData());
         System.out.println("Custo: R$"+BancoDados.listaPedidosAbertos.get(verificador.procurarPedido(BancoDados.listaPedidosAbertos,nome)).getCusto());
         System.out.println("Operador: "+BancoDados.listaPedidosAbertos.get(verificador.procurarPedido(BancoDados.listaPedidosAbertos,nome)).getNomeFuncionario());
         System.out.println("---------------------\n");
      }
   }
}
```

DEPOIS NA CLASSE NOTAFISCAL:

```
6  public class NotaFiscal {
7     static ArrayList <Pedido>  listaPedidosAbertos = BancoDados. listaPedidosAbertos;
8     Scanner input = new Scanner(System.in);
9
10    public void exibirNotaFiscal() {
11       if(!BancoDados.listaPedidosAbertosIsEmpty()) {
12          int indicePedido = Pedido.getIndice();
13          if(indicePedido!=-1) {
14             exibirCabecalho();
15             exibirProdutos(indicePedido);
16             exibirData(indicePedido);
17             exibirCusto(indicePedido);
18             exibirOperador(indicePedido);
19             exibirRodape();
20          }
21       }
22    }
```