

Diagrama de Classes

Exercício: Resolução

Vinicius Takeo Friedrich Kuwaki

Universidade do Estado de Santa Catarina

Seções

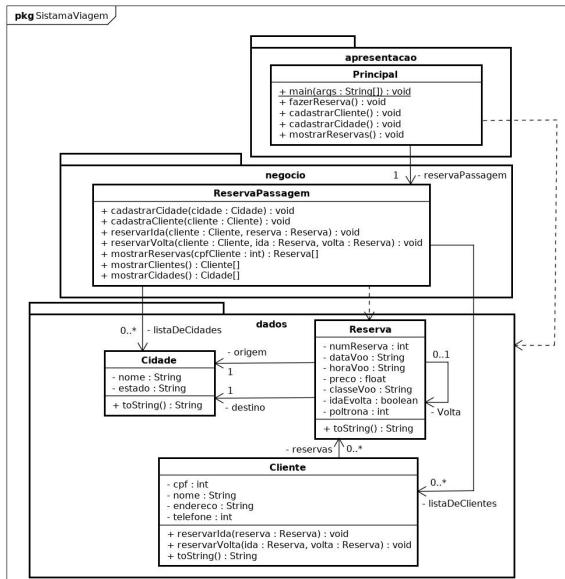
Exercício

Resolução

Exercício

Implemente o Diagrama UML a seguir:

Exercício



Exercício

- Todas as classes do pacote de dados devem implementar seus métodos **toString()**. Esse método retorna uma String contendo todas as informações relacionadas ao objeto.
- Os métodos de cadastrar Cidade e Cliente, adicionam objetos ao array (ou lista);
- Os métodos de realizar reservas adicionam a reserva dentro do array de reservas do cliente;
- O método de mostrar reservas busca as reservas relacionadas ao cliente cujo cpf foi passado como parâmetro e retorna um array contendo todas as reservas;
- Os métodos de mostrar clientes e cidades também retornam seus respectivos arrays (ou listas);
- Toda a entrada e saída de dados deve ser feita na classe Principal, isto é, métodos dos pacotes dados e negócios não podem exibir ou requisitar dados do usuário!

Seções

Exercício

Resolução

Resolução

- O diagrama UML de classes apresentado anteriormente é de um sistema em três camadas, logo, teremos uma camada de dados, negócio e apresentação;
- Iremos primeiro definir as classes da camada de dados;
- A primeira delas é a classe Cidade;
- Ela pertence ao pacote *dados* e possui dois atributos: nome e estado;

```
package dados;  
  
public class Cidade {  
    private String nome;  
    private String estado;  
}
```

- A classe possui seus métodos getters e setters, mas não irei detalha-los aqui;
- Apenas será apresentada a implementação do método **toString()**. Esse método retorna todos os atributos da Cidade concatenados em uma String:

```
public String toString() {  
    return this.nome + " – " + this.estado + "\n";  
}
```

- Agora será definido a classe Reserva, esta possui vários atributos;
- Também pertence ao pacote dados:

```
package dados;  
  
public class Reserva {  
  
    private int numReserva;  
    private String dataVoo;  
    private String horaVoo;  
    private float preco;  
    private String classeVoo;  
    private boolean idaVolta;  
    private int poltrona;  
    private Reserva volta;  
    private Cidade origem;  
    private Cidade destino;  
}
```


Resolução

- Vale destacar apenas o atributo volta. Esse atributo é um objeto do mesmo tipo da classe ao qual ele pertence.
- A classe Reserva também possui seu método **toString()**. Esse método apenas exibe os dados relacionados a reserva de volta, caso ela seja diferente de **null**:

```
public String toString() {  
  
    String reserva = "";  
    reserva += "Numero Reserva: " + this.numReserva + "\n";  
    reserva += "Origem: " + this.origem.toString();  
    reserva += "Destino: " + this.destino.toString();  
    reserva += "Data do voo: " + this.dataVoo + "\n";  
    reserva += "Hora do voo: " + this.horaVoo + "\n";  
    reserva += "Preco: " + this.preco + "\n";  
    reserva += "Classe do voo: " + this.classeVoo + "\n";  
    reserva += "Poltrona: " + this.poltrona + "\n";  
  
    if (this.volta != null) {  
        reserva += this.volta.toString();  
    }  
  
    return reserva;  
}
```

Resolução

- Agora será definida a classe Cliente;
- Ela possui um atributo reservas, que é um array de objetos do tipo Reserva;
- Entretanto em vez de utilizar um array, irei utilizar uma LinkedList, que é uma estrutura que automatiza o armazenamento de objetos em um array;
- Para isso é necessário importa-la:

```
package dados;  
  
import java.util.LinkedList;
```

- Logo a classe fica com os seguintes atributos:

```
public class Cliente {  
  
    private int cpf;  
    private String nome;  
    private String endereco;  
    private int telefone;  
    private LinkedList<Reserva> reservas = new LinkedList<Reserva>();  
}
```

- O unico método get que vale destacar é o das reservas, entretanto ele não é diferente dos demais:

```
public LinkedList<Reserva> getReservas() {  
    return this.reservas;  
}
```

- O diagrama pede o método **reservarIda()**. No qual recebe uma reserva e adiciona ao array (em nosso caso a lista de reservas):

```
public void reservarIda(Reserva reserva) {  
    this.reservas.add(reserva);  
}
```

- Já a volta não precisa necessariamente estar dentro da lista, pois ela será associada a uma ida, logo basta setar o atributo volta de uma ida:

```
public void reservarVolta(Reserva ida, Reserva volta) {  
    ida.setIdaEvolta(true);  
    ida.setVolta(volta);  
}
```

- A classe Cliente também possui um método **toString()** que segue os mesmos princípios dos métodos toString das demais classes:

```
public String toString() {  
    String cliente = "";  
    cliente += "CPF: " + this.cpf + "\n";  
    cliente += "Nome: " + this.nome + "\n";  
    cliente += "Endereco: " + this.endereco + "\n";  
    cliente += "Telefone: " + this.telefone + "\n";  
    return cliente;  
}
```

- Agora será implementado a classe ReservaPassagem, que pertence ao pacote de negócios;

Resolução

- Novamente iremos utilizar um LinkedList para substituir os arrays;
- E iremos importar as classes do pacote de dados:

```
package negocio;  
  
import java.util.LinkedList;  
  
import dados.Cidade;  
import dados.Cliente;  
import dados.Reserva;
```

- Essa classe terá uma lista de clientes e uma de cidades. Além de possuir um atributo para enumerar as reservas:

```
public class ReservaPassagem {  
  
    private LinkedList<Cidade> listaDeCidades = new LinkedList<Cidade>();  
    private LinkedList<Cliente> listaDeClientes = new LinkedList<Cliente>();  
  
    private int numeroReservas = 0;
```

- Os métodos de cadastrar cidades e clientes são exatamente iguais. Basicamente recebem um objeto e adicionam a lista:

```
public void cadastrarCidade(Cidade cidade) {  
    this.listaDeCidades.add(cidade);  
}
```

```
public void cadastrarCliente(Cliente cliente) {  
    this.listaDeClientes.add(cliente);  
}
```

- Já o método de reservar ida adiciona uma reserva a um cliente, além de adicionar um código a reserva (o atributo destacado na declaração dessa classe). Esse código precisa ser incrementado em um a cada nova reserva:

```
public void reservarIda(Cliente cliente , Reserva reserva) {  
  
    reserva.setNumReserva(this.numeroReservas);  
    this.numeroReservas++;  
    cliente.reservarIda(reserva);  
  
}
```

- O método reservar volta realiza um processo semelhante, visto que os métodos de reserva pertencem a classe Cliente:

```
public void reservarVolta(Cliente cliente , Reserva ida , Reserva volta) {  
  
    ida.setNumReserva(this.numeroReservas);  
    this.numeroReservas++;  
  
    volta.setNumReserva(this.numeroReservas);  
    this.numeroReservas++;  
  
    cliente.reservarVolta(ida , volta);  
  
}
```

- O próximo método recebe um cpf como parâmetro e retorna o cliente que possui esse cpf. O método retorna a lista de clientes e verifica um a uma se possui o cpf desejado. Caso positivo retorna o cliente:

```
public Cliente buscarCliente(int cpf) {  
    for (Cliente c : this.listaDeClientes) {  
        if (c.getCpf() == cpf) {  
            return c;  
        }  
    }  
    return null;  
}
```

- O método mostrar reservas utiliza do método de buscar clientes. Esse método retorna uma lista de reservas pertencentes ao cliente que possui o cpf passado como parâmetro. Caso o cliente não exista, o método retorna uma lista vazia:


```
public LinkedList<Reserva> mostrarReservas(int cpfCliente) {  
    Cliente c = buscarCliente(cpfCliente);  
  
    if (c != null) {  
        return c.getReservas();  
    }  
    return new LinkedList<Reserva>();  
}
```

- Os métodos de mostrar clientes e cidades atua como sendo o get das listas de clientes e cidades:

```
public LinkedList<Cliente> mostrarClientes() {  
    return this.listaDeClientes;  
}  
  
public LinkedList<Cidade> mostrarCidades() {  
    return this.listaDeCidades;  
}
```

Resolução

- Agora a última classe a ser implementada é a do pacote de apresentação;
- Utilizaremos a classe Scanner para ler os dados via console. Além de utilizar todas as classes do pacote de dados (relacionamento de dependência exemplificado aqui):
- Como teremos uma instância direta da classe ReservaPassagem, isso justifica o relacionamento de associação;

```
package apresentacao;  
  
import java.util.Scanner;  
  
import dados.Cidade;  
import dados.Cliente;  
import dados.Reserva;  
import negocio.ReservaPassagem;
```

- A classe Principal então terá dois atributos estáticos, pois serão utilizados na **main**:

```
public class Principal {  
    private static Scanner s = new Scanner(System.in);  
    private static ReservaPassagem sistema = new ReservaPassagem();
```

- O método **main** terá um menu de opções:

```
public static void exibeMenuPrincipal() {  
    System.out.println("Escolha uma opção:");  
    System.out.println("0 - Encerrar");  
    System.out.println("1 - Realizar Reserva");  
    System.out.println("2 - Cadastrar Cliente");  
    System.out.println("3 - Mostrar Reservas");  
    System.out.println("4 - Cadastrar Cidade");  
}
```

- O método **main** também possui um laço de repetição que só é quebrado quando o usuário digita 0:

```
public static void menuPrincipal() {  
    int opcao = -1;  
    while (opcao != 0) {  
        exibeMenuPrincipal();  
        opcao = s.nextInt();  
    }
```

- Após o usuário escolher uma opção, um *switch case* pula para o método em específico:

```
switch (opcao) {  
    case 1:  
        System.out.println("Realizar Reserva");  
        realizarReserva();  
        break;  
    case 2:  
        System.out.println("Cadastrar Cliente");  
        cadastrarCliente();  
        break;  
    case 3:  
        System.out.println("Mostrar Reservas");  
        mostrarReservas();  
        break;  
}
```

```
case 4:
    System.out.println("Cadastrar Cidade");
    cadastrarCidade();
    break;
default:
    System.out.println("Numero invalido");
    break;
}
```

- Para implementar o método **cadastrarCliente()** primeiro iremos implementar um método para ler os dados do console e retornar um cliente:

```
public static Cliente novoCliente() {

    System.out.println("Digite o cpf:");
    int cpf = s.nextInt();

    System.out.println("Digite o nome:");
    String nome = s.nextLine();
    nome = s.nextLine();

    System.out.println("Digite o endereco:");
    String endereco = s.nextLine();
}
```

```
System.out.println("Digite o telefone:");  
int telefone = s.nextInt();  
  
Cliente c = new Cliente();  
  
c.setCpf(cpf);  
c.setNome(nome);  
c.setEndereco(endereco);  
c.setTelefone(telefone);  
  
return c;  
}
```

- Esse método é então chamado pelo **cadastrarCliente()**, que envia ao sistema o cliente retornado pelo método **novoCliente()**

```
public static void cadastrarCliente() {  
    sistema.cadastrarCliente(novoCliente());  
}
```

- Após os clientes estarem cadastrados no sistema é possível lista-los. O método **mostrarClientes()** percorre a lista retornada pelo pacote de negócio e a exibe para o usuário:

```
public static void mostrarClientes() {  
    for (Cliente c : sistema.mostrarClientes()) {  
        System.out.println(c.toString());  
    }  
}
```

- O método **escolherCliente()** então faz uso do método de mostrar os clientes. A partir do cpf o usuário escolhe um cliente, que é então retornado pelo método:

```
public static Cliente escolherCliente() {  
    mostrarClientes();  
    System.out.println("Digite o CPF do cliente escolhido:");  
  
    Cliente c = sistema.buscarCliente(s.nextInt());  
  
    if (c != null) {  
        return c;  
    }  
  
    return null;  
}
```

- O mesmo será feito para as cidades;
- Primeiro é criado um método **novaCidade()**, que requisita ao usuário os dados da cidade e então retorna o objeto com seus atributos setados:


```
public static Cidade novaCidade() {  
  
    System.out.println("Digite o nome da cidade:");  
    String nome = s.nextLine();  
    nome = s.nextLine();  
  
    System.out.println("Digite o estado da cidade");  
    String estado = s.nextLine();  
  
    Cidade c = new Cidade();  
    c.setNome(nome);  
    c.setEstado(estado);  
  
    return c;  
  
}
```

- Após isso é possível cadastrar uma cidade:

```
public static void cadastrarCidade() {  
    sistema.cadastrarCidade(novaCidade());  
}
```

- Com as cidades já cadastradas é possível lista-las;
- Diferentemente dos clientes que possuem cpf, as cidades não possuem atributos que as diferenciam umas das outras, por isso é preciso exibi-las com a sua posição na lista:

```
public static void mostrarCidades() {  
    for (int i = 0; i < sistema.mostrarCidades().size(); i++) {  
        System.out.println("Cidade " + i);  
        System.out.println(sistema.mostrarCidades().get(i).toString());  
    }  
}
```

- Agora então é possível implementar o método de escolher cidades;

Resolução

- Esse método exibe ao usuário as cidades cadastradas no sistema e então requisita ao usuário para escolher uma:

```
public static Cidade escolherCidade() {  
  
    mostrarCidades();  
    System.out.println("Escolha uma cidade:");  
    int codigo = s.nextInt();  
  
    if (codigo > sistema.mostrarCidades().size()) {  
        System.out.println("Cidade inválida");  
        return null;  
    } else {  
        return sistema.mostrarCidades().get(codigo);  
    }  
  
}
```

- Agora é possível implementar o método **realizarReserva()**, primeiro é necessário implementar um método para fazer a leitura de dados do console;
- Esse método lê os dados e retorna uma reserva:

```
public static Reserva novaReserva() {  
  
    System.out.println("Digite a data do voo:");  
    String data = s.nextLine();  
    data = s.nextLine();  
  
    System.out.println("Digite a hora do voo:");  
    String hora = s.nextLine();  
  
    System.out.println("Digite o pre o do voo:");  
    float preco = s.nextFloat();  
  
    System.out.println("Digite a classe do voo:");  
    String classe = s.nextLine();  
    classe = s.nextLine();  
  
    System.out.println("Digite a poltrona no voo:");  
    int poltrona = s.nextInt();  
  
    Cidade origem = escolherCidade();  
    Cidade destino = escolherCidade();  
  
    Reserva r = new Reserva();  
    r.setDataVoo(data);  
}
```

```
        r.setHoraVoo(hora);  
        r.setPreco(preco);  
        r.setClasseVoo(classe);  
        r.setPoltrona(poltrona);  
        r.setOrigem(origem);  
        r.setDestino(destino);  
  
        return r;  
    }
```

- Após isso um cliente é escolhido para realizar a reserva:

```
    public static void realizarReserva() {  
        Cliente c = escolherCliente();
```

- Caso o cliente exista, é possível então associar uma reserva a ele;
- O usuário pode cadastrar somente a ida ou a ida e a volta:

```
if (c != null) {  
  
    int opcao = -1;  
  
    while (opcao != 0) {  
  
        System.out.println("Digite 0 para Sair");  
        System.out.println("Digite 1 para reservar somente a Ida");  
        System.out.println("Digite 2 para reservar Ida e Volta");  
        opcao = s.nextInt();  
    }  
}
```

- Um *switch case* determina o que deve ser feito:


```
switch (opcao) {  
case 1:  
    Reserva r = novaReserva();  
    sistema.reservarIda(c, r);  
    break;  
case 2:  
    Reserva r1 = novaReserva();  
    Reserva volta = novaReserva();  
    sistema.reservarVolta(c, r1, volta);  
    break;  
}
```

```
default:  
    System.out.println("Numero invalido");  
    break;  
}
```

- Após existirem reservas cadastradas no sistema é possível lista-las:

```
public static void mostrarReservas() {  
    Cliente c = escolherCliente();  
  
    for (Reserva r : sistema.mostrarReservas(c.getCpf())) {  
        System.out.println(r.toString());  
    }  
}
```

- Os códigos-fonte dessa resolução estarão disponíveis nesse link.

 KUWAKI, V. T. F. Modelo de slides udesc lattex. In: . [S.l.]: Disponível em: <<https://github.com/takeofriedrich/slidesUdescLattex>>. Acesso em: 24 jan. 2020.

Duvidas:
Vinicius Takeo Friedrich Kuwaki
vinicius.kuwaki@edu.udesc.br
github.com/takeofriedrich



UDESC
UNIVERSIDADE
DO ESTADO DE
SANTA CATARINA