

# Instrução prática 24

## Etapa 1: Individual

**1.** Defina, de acordo com o que vimos nas aulas de Herança, Composição e Agregação, qual estratégia é mais adequada para a implementação de cada uma das classes acima. Explique o porquê de suas escolhas.

Herança - não é necessária para o caso descrito, pois não há uma relação clara entre as classes.

Composição e agregação - são mais apropriadas, pois permitem criar uma estrutura mais flexível e de baixo acoplamento, facilitando a manutenção e extensibilidade do sistema.

Para determinar a estratégia mais adequada (herança, composição ou agregação) para a implementação das classes mencionadas, é importante considerar a relação entre essas classes e como elas se relacionam no contexto do sistema que está sendo desenvolvido.

Segue abaixo, um análise geral das estratégias mais apropriadas com base nas informações fornecidas.

Classe Cliente:

- Estratégia: Herança.
- Justificação: A classe Cliente pode ser a classe base de várias subclasses que representam tipos específicos de clientes, como ClientePessoaFisica e ClientePessoaJuridica. A herança permite criar subclasses que compartilham características comuns, como nome, endereço e número de telefone, ao mesmo tempo em que podem adicionar atributos específicos de cada tipo de cliente.

Classe Pacote:

- Estratégia: Composição.
- Justificação: Um Pacote é composto por diversos elementos, como voos, hotéis, atividades, etc. A composição é a estratégia mais adequada, pois um Pacote pode conter muitos objetos de outras classes e controlar a vida útil desses objetos.

Classe Dependente:

- Estratégia: Herança (se os dependentes compartilham atributos e comportamentos comuns) ou Composição (se os dependentes não têm um relacionamento "é um" com a classe Cliente).
- Justificação: Se Dependente compartilha atributos e comportamentos comuns com Cliente, a herança pode ser adequada. Se Dependente é uma entidade independente com uma relação "tem um" com Cliente, a composição é mais apropriada.

#### Classe Evento:

- Estratégia: Composição.
- Justificação: Eventos podem conter detalhes, como data, hora, local e descrição. A composição é apropriada, pois um Evento pode ser composto por vários atributos, e o evento em si é uma parte de outra entidade, como um Pacote.

#### Classe Roteiro:

- Estratégia: Agregação.
- Justificação: Um Roteiro pode ser uma coleção de eventos, locais, atividades, etc. A agregação é uma relação de "tem muitos" ou "contém muitos", onde um Roteiro pode conter diversos elementos, mas esses elementos também podem existir independentemente do Roteiro.

#### Classe Deslocamento:

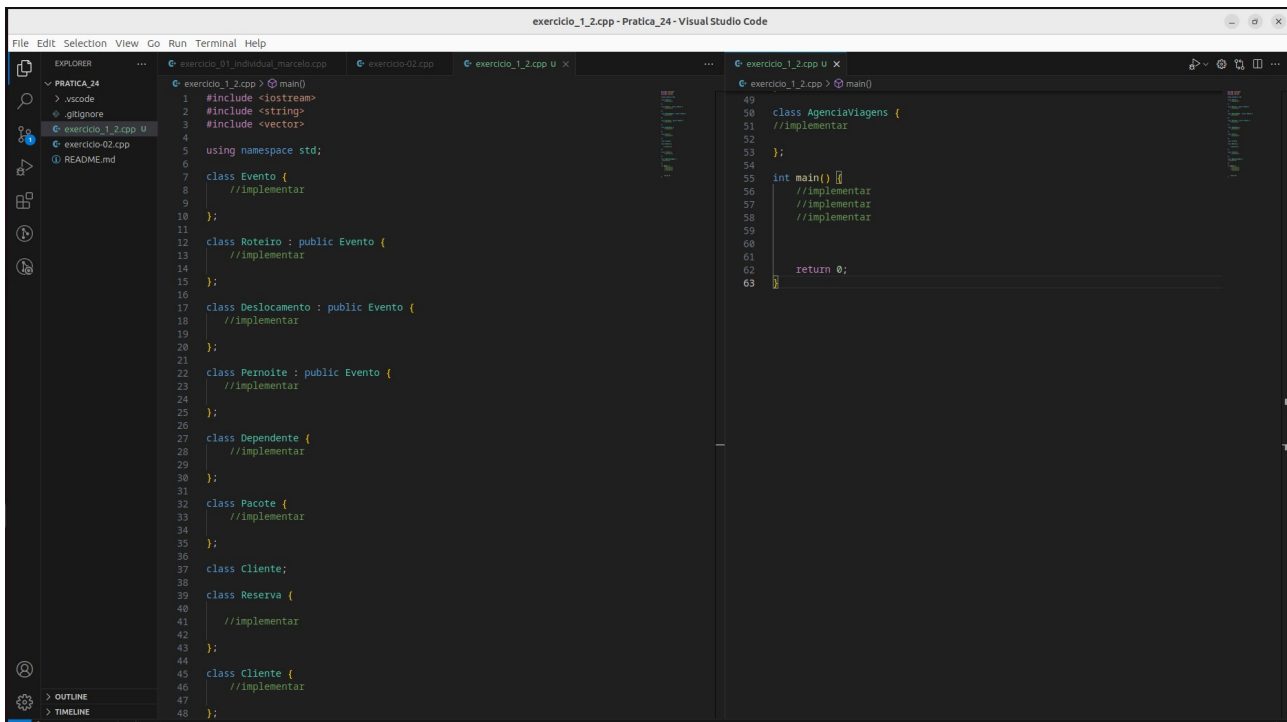
- Estratégia: Composição.
- Justificação: Um Deslocamento pode conter informações sobre um meio de transporte, horários, origem e destino. A composição é apropriada, pois um Deslocamento é uma parte de um Pacote ou Roteiro e é responsável por controlar a vida útil dos objetos de transporte.

#### Classe Pernoite:

- Estratégia: Composição.
- Justificação: A classe Pernoite representa acomodações durante uma viagem, e pode conter informações sobre o hotel, datas, custos, etc. A composição é a estratégia apropriada, pois as instâncias de Pernoite são uma parte de Pacotes ou Roteiros.

É importante lembrar que a escolha da estratégia de modelagem (herança, composição, agregação) depende das necessidades específicas do sistema, da relação entre as classes e das funcionalidades que precisam ser suportadas. Portanto, essas sugestões são baseadas em uma análise geral, e a implementação real pode variar dependendo dos requisitos detalhados do sistema.

2. Esboce a implementação das classes de acordo com a estratégia que você julgou adequada.



The image shows a Visual Studio Code editor window titled "exercício\_1\_2.cpp - Pratica\_24 - Visual Studio Code". The editor displays two C++ source files. The left file, "exercício\_1\_2.cpp", contains the main logic and class definitions. The right file, "exercício\_1\_2.cpp u", contains the implementation of the "AgenciaViagens" class and the "main" function.

```
1 #include <iostream>
2 #include <string>
3 #include <vector>
4
5 using namespace std;
6
7 class Evento {
8     //implementar
9 };
10
11 class Roteiro : public Evento {
12     //implementar
13 };
14
15
16 class Deslocamento : public Evento {
17     //implementar
18 };
19
20
21 class Pernoite : public Evento {
22     //implementar
23 };
24
25
26 class Dependente {
27     //implementar
28 };
29
30
31 class Pacote {
32     //implementar
33 };
34
35
36 class Cliente;
37
38 class Reserva {
39     //implementar
40 };
41
42
43
44
45 class cliente {
46     //implementar
47 };
48
```

```
49
50 class AgenciaViagens {
51     //implementar
52 };
53
54
55 int main() {
56     //implementar
57     //implementar
58     //implementar
59
60
61     return 0;
62 }
63
```