**Roteiro 01**

Os roteiros a serem desenvolvidos a seguir visam trazer a percepção da evolução no processo de desenvolvimento de software. Por isso, iremos criar um projeto com vários pacotes, onde cada pacote representa a evolução da implementação deste projeto.

Neste caso iremos explorar os Padrões de Projeto do GoF (Gang of Four), e neste roteiro iremos trabalhar com o **Padrão Strategy**.

Agora que já conhecemos algumas técnicas da OO, alguns princípios e boas práticas, iremos evoluir um novo cenário até chegar a um dos padrões, o Strategy.

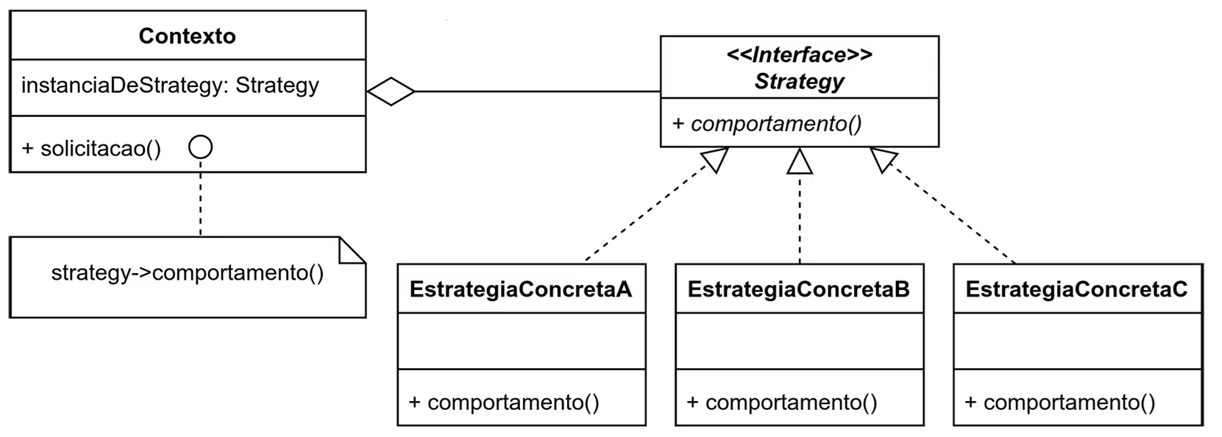
O **Padrão Strategy** define uma família de algoritmos, encapsula cada um deles e os torna intercambiáveis, deixando assim o algoritmo variar independente dos clientes que o utilizam.

Abaixo temos o modelo formal do padrão Strategy

Diagrama

Descrição gerada automaticamente

Abaixo temos uma outra representação do Strategy um pouco mais intuitiva.



**Cenário:**

**Suponha um projeto para um jogo de esportes. Inicialmente após a entrevista com nosso cliente, identificamos a necessidade de criar dois tipos de jogadores. Um jogador de tênis e outro de futebol. Todo jogador possui um nome que o identifica, e o jogo permite que os jogadores treinem, compitam e definam suas próprias táticas para uso do jogo.**

**Início do projeto – Pacote : roteiro1.parte1**

**OBS.: Colocaremos os roteiros sobre padrões de projeto em um projeto diferente do projeto criado para o SOLID**

1 – Criar um projeto no NetBeans chamado **PADROESroteiros**

2 – Inicialmente temos a primeira classe chamada **Player**. Esta classe deve ser criada no pacote **roteiro1.parte1.**

package roteiro1.parte1;

public abstract class Player {

private String nome;

public Player(String nome) {

this.nome = nome;

}

public void treinar(){

System.out.println(this.nome + " Executando o treino !");

}

public void estiloCompetidor(){

System.out.println(this.nome + " Muito competitivo !");

}

public abstract void definirTatica();

}

Observe que a classe tem um atributo **nome** que pode ser informado no construtor da classe. Além disso tem 2 métodos concretos e um abstrato :

- treinar() – permite executar o treino do jogador.

- estiloCompetidor() – permite informar o quanto o jogador é competitivo.

- definirTatica() – método abstrato para definir a tática do jogador. Como cada tipo de jogador terá diferentes táticas, definimos este método como abstrato e por consequência a classe também precisa ser abstrata.

3 – Devemos agora criar as classes **SoccerPlayer** e **TennisPlayer** como subclasses da classe **Player** através de herança.

package roteiro1.parte1;

public class SoccerPlayer extends Player{

public SoccerPlayer (String nome) {

super(nome);

}

@Override

public void definirTatica() {

System.out.println(super.nome + " Trabalha em equipe !");

}

}

Atenção : Como precisamos usar o atributo nome dentro da subclasse, precisaremos alterar a cláusula de acesso deste atributo de **private** para **protected** dentro da superclasse.

4 – O mesmo deve ser feito ao criar a classe **TennisPlayer**, e a tática deste jogador é definida com a mensagem abaixo.

System.out.println(super.nome + " Rebate a bola por cima do oponente");

5 – Agora crie uma classe para teste chamada **TesteJogo**.

package roteiro1.parte1;

public class TesteJogo {

public static void main(String[] args) {

System.out.println("Detalhes do Jogador de Tenis");

TennisPlayer guga = new TennisPlayer("Gustavo Kuerten");

guga.treinar();

guga.estiloCompetidor();

guga.definirTatica();

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

System.out.println("Detalhes do Jogador de Futebol");

SoccerPlayer ronaldo = new SoccerPlayer("Ronaldinho Gaucho");

ronaldo.treinar();

ronaldo.estiloCompetidor();

ronaldo.definirTatica();

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"); }

}

6 – Utilize uma ferramenta de software qualquer para geração do diagrama de classes para esta etapa do projeto (Sugestão : Astah Community). Obs.: Adicione aqui o diagrama para que seja disponibilizado no teams

Obs: Outra possibilidade para geração dos diagramas é o StarUML : <https://staruml.io/> . Para fazer a engenharia reversa do código siga as orientações conforme o vídeo a seguir : <https://www.youtube.com/watch?v=z6DDuZQArro>

**Pacote : roteiro1.parte2**

Imagine agora que o nosso projeto precise evoluir e precisamos de 3 novos tipos de jogadores. Um de xadrez, um de golf e outro de cartas. Teremos as classes : **ChessPlayer, GolfPlayer, CardPlayer** . Acontece também que precisaremos incluir uma nova habilidade para os jogadores. Trata-se da habilidade de correr, e já podemos perceber que nem todos os tipos de jogadores precisarão desta habilidade.

1 – No mesmo projeto crie o pacote roteiro3.parte2

2 – Copie todas as classes criadas na parte1 para o novo pacote.

3 – Inicialmente vamos criar o método correr() na classe Player como forma de implementar a nova habilidade de um jogador.

package roteiro1.parte2;

public abstract class Player {

private String nome;

public Player(String nome) {

this.nome = nome;

}

public void treinar(){

System.out.println(this.nome + " Executando o treino !");

}

public void estiloCompetidor(){

System.out.println(this.nome + " Muito competitivo !");

}

public void correr(){

System.out.println(this.nome + " Corre muito durante o jogo !");

}

public abstract void definirTatica();

}

4 – Crie as classes **ChessPlayer, GolfPlayer, CardPlayer** que também herdam da classe Player, semelhante as classes **SoccerPlayer** e **TennisPlayer**. Segue abaixo a sugestão de tática para cada jogador

Jogador de xadrez

System.out.println(super.nome + " - Domina o centro do tabuleiro");

Jogador de golf

System.out.println(super.nome + " - Pontua com o nº de tacadas previstas");

System.out.println(super.nome + " - É um jogador muito calmo");

Jogador de cartas

5 – Depois de implementadas as classes do item 4, perceba que temos um problema de modelagem. Os jogadores de Xadrez, Carta e Golf também irão herdar o método **correr**, e isso não faz muito sentido dada a especificidades destes tipos de jogadores.

Uma possível solução seria sobrescrever o método **correr** nestas classes, usando o conceito de sobrecarga de método da OO. Certamente existem outras soluções, piores ou melhores, mas ainda assim esta é uma solução viável para testes.

Abaixo temos a sobrecarga do método na classe **ChessPlayer** e o mesmo pode ser feito nas outras 2 classes.

package roteiro1.parte2;

public class ChessPlayer extends Player{

public ChessPlayer(String nome) {

super(nome);

}

@Override

public void definirTatica() {

System.out.println(super.nome + " - Domina o centro do tabuleiro");

}

@Override

public void correr() {

System.out.println(super.nome + " - Não Corre ! Ele pensa !");

}

}

Outra possibilidade é sobrescrever o método simplesmente sem implementação como segue abaixo. Não é a solução mais elegante, mas funciona.

@Override

public void correr() {

// Sem implementação

}

A seguir faremos o teste das novas implementações

6 – Faça os devidos testes usando a classe **TesteJogo** acrescentando os novos jogadores e a habilidade de correr. Verifique se aconteceu o resultado esperado.

package roteiro1.parte2;

public class TesteJogo {

public static void main(String[] args) {

System.out.println("Detalhes do Jogador de Tenis");

TennisPlayer guga = new TennisPlayer("Gustavo Kuerten");

guga.treinar();

guga.estiloCompetidor();

guga.definirTatica();

guga.correr();

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

System.out.println("Detalhes do Jogador de Futebol");

SoccerPlayer ronaldo = new SoccerPlayer("Ronaldinho Gaucho");

ronaldo.treinar();

ronaldo.estiloCompetidor();

ronaldo.definirTatica();

ronaldo.correr();

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

System.out.println("Detalhes do Jogador de Cartas");

CardPlayer joseCartas = new CardPlayer("Jose das Cartas");

joseCartas.treinar();

joseCartas.estiloCompetidor();

joseCartas.definirTatica();

joseCartas.correr();

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

System.out.println("Detalhes do Jogador de Xadrez");

ChessPlayer kasparov = new ChessPlayer("Kasparov");

kasparov.treinar();

kasparov.estiloCompetidor();

kasparov.definirTatica();

kasparov.correr();

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

System.out.println("Detalhes do Jogador de Golf");

GolfPlayer tigerwoods = new GolfPlayer("Tiger Woods");

tigerwoods.treinar();

tigerwoods.estiloCompetidor();

tigerwoods.definirTatica();

tigerwoods.correr();

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

}

}

7 – Utilize uma ferramenta de software qualquer para geração do diagrama de classes para esta etapa do projeto (Sugestão : Astah Community). Obs.: Adicione aqui o diagrama para que seja disponibilizado no teams

Obs: Outra possibilidade para geração dos diagramas é o StarUML : <https://staruml.io/> . Para fazer a engenharia reversa do código siga as orientações conforme o vídeo a seguir : <https://www.youtube.com/watch?v=z6DDuZQArro>

8 – Esta solução atendeu plenamente aos novos requisitos do projeto ? Quais são as suas críticas ?

**Pacote : roteiro1.parte3**

Faremos agora uma tentativa de melhoria neste modelo, no que diz respeito a duplicidade de código e talvez uma forma mais elegante de implementar métodos que não desejáveis em subclasses quando trabalhamos com herança. Neste caso, estamos falando do método **correr()**.

Faremos novamente o uso de uma interface para declarar o método correr(). Chamaremos esta interface de **Runnable** como forma de indicar que algumas classes podem implementar os métodos existentes nela e outras classes não.

1 - No mesmo projeto crie o pacote roteiro1.parte3

2 – Copie todas as classes criadas na parte2 para o novo pacote

3 – Crie a interface **Runnable** com a declaração do método **correr()** conforme segue abaixo.

package roteiro1.parte3;

public interface Runnable {

public void correr();

}

4 – Uma vez criada a interface com o método **correr()** devemos retirar este método da classe Player, pois não faz mais sentido.

package roteiro1.parte3;

public abstract class Player {

protected String nome;

public Player(String nome) {

this.nome = nome;

}

public void treinar(){

System.out.println(this.nome + " Executando o treino !");

}

public void estiloCompetidor(){

System.out.println(this.nome + " Muito competitivo !");

}

public void correr(){

System.out.println(this.nome + " Corre muito durante o jogo !");

}

public abstract void definirTatica();

}

5 – Agora faremos com que as classes de jogadores específicos implementem a interface **Runnable** quando for adequado. Observe que agora os nossos jogadores herdam da classe **Player** e podem implementar ou não a classe **Runnable**. Para efeito de teste, segue abaixo 1 classe implementada como exemplo. Vamos então implementar as classes : **SoccerPlayer**, **TennisPlayer**

package roteiro1.parte3;

public class SoccerPlayer extends Player implements Runnable{

public SoccerPlayer(String nome) {

super(nome);

}

@Override

public void definirTatica() {

System.out.println(super.nome + " Trabalha em equipe !");

}

@Override

public void correr() {

System.out.println(this.nome + " Corre muito durante o jogo !");

}

}

6 – Se neste momento todas as refatorações foram concluídas, já podemos fazer o teste na classe **TesteJogo**. Obs.: Lembre-se que agora os jogadores de golf e de cartas não possuem o método **correr()**, e dessa forma a chamada deste método na classe de teste deve ser retirada, caso contrário teremos um erro de compilação.

7 – Consegue identificar princípios do SOLID aplicados nesta refatoração ?

8 – Utilize uma ferramenta de software qualquer para geração do diagrama de classes para esta etapa do projeto (Sugestão : Astah Community). Obs.: Adicione aqui o diagrama para que seja disponibilizado no teams

Obs: Outra possibilidade para geração dos diagramas é o StarUML : <https://staruml.io/> . Para fazer a engenharia reversa do código siga as orientações conforme o vídeo a seguir : <https://www.youtube.com/watch?v=z6DDuZQArro>

**Seguimos para o pacote2.parte4 !**

**Pacote : roteiro1.parte4**

O uso de uma interface para implementar a habilidade de correr nos jogadores é uma boa ideia e pode ser melhorada. Entendemos que a ação de correr além de ser uma habilidade do jogador, é também um comportamento que o jogador pode assumir ou não. Em outras palavras, podemos atribuir o comportamento de um rápido corredor a um objeto jogador em um dado momento. Em outro momento ele pode ser lento ou simplesmente não correr. Assim como um jogador de cartas não corre em momento algum, o jogador de Tennis pode mudar o ritmo de jogo de rápido para lento.

Criaremos a seguir **uma interface** e **duas classes** que implementam esta interface.

- Interface **RunBehavior** – representa um comportamento para o jogador que precisa correr.

- Classe **RunFast** – representa uma classe que implementa **RunBehavior** e será usada para um jogador que precisa correr rápido.

- Classe **RunNoWay** - representa uma classe que implementa **RunBehavior** e será usada para um jogador que não precisa correr.

Perceba que poderíamos criar outras classes que implementam **RunBehavior,** como por exemplo um comportamento para um jogador lento que seria algo do tipo **RunSlow**.

ATENÇÃO : Desta vez faremos um pouco diferente. Vamos aproveitar o código feito no pacote **roteiro3.parte2** para gerar este novo pacote **roteiro3.parte4**. A solução da parte3 não ficou bom o suficiente para evoluirmos a partir dela.

1 - No mesmo projeto crie o pacote roteiro1.parte4

2 – Copie todas as classes criadas na parte2 para o novo pacote

3 - Crie a interface **RunBehavior** com a declaração do método **correr()** conforme segue abaixo.

package roteiro1.parte4;

public interface RunBehavior {

public void correr();

}

4 – Crie as classes **RunFast** , **RunNoWay e RunSlow** de forma que implementem a interface **RunBehavior**, conforme segue abaixo.

package roteiro1.parte4;

public class RunFast implements RunBehavior{

@Override

public void correr() {

System.out.println("Corre muito rápido !");

}

}

package roteiro1.parte4;

public class RunNoWay implements RunBehavior{

@Override

public void correr() {

// Sem implementação

}

}

Com esta estratégia de modelagem criamos uma hierarquia em cima da ação/comportamento de um jogador que corre. Temos então 2 hierarquias de classes. Uma em cima de **Player** e outra em cima de **RunBehavior**.

Assim, podemos isolar o comportamento de quem corre da nossa classe **Player**.

5 - Precisamos agora refatorar a classe **Player**. Lembre-se que agora o método correr() é de responsabilidade da hierarquia montada em cima de **RunBehavior.** Precisamos então criar uma associação entre Player e **RunBehavior.** Crie um atributo na classe **Player** do tipo **RunBehavior** como o nome **habilidadeCorrer**.

package roteiro1.parte4;

public abstract class Player {

protected String nome;

protected RunBehavior habilidadeCorrer;

public Player(String nome, RunBehavior habilidadeCorrer) {

this.nome = nome;

this.habilidadeCorrer = habilidadeCorrer;

}

public void treinar(){

System.out.println(this.nome + " Executando o treino !");

}

public void estiloCompetidor(){

System.out.println(this.nome + " Muito competitivo !");

}

public void correr(){

this.habilidadeCorrer.correr();

}

public abstract void definirTatica();

}

Para facilitar a manipulação do atributo **habilidadeCorrer** , observe que colocamos como parâmetro no construtor da classe.

Observe também que o método **correr()**, também precisa de uma refatoração. Então, por meio de delegação acessamos o método correr que na verdade está em **RunBehavior**.

6 – Como consequência da refatoração feita no item 5, devemos também acrescentar a passagem de parâmetro do atributo **habilidadeCorrer** nas subclasses de Player (**SoccerPlayer**, **TennisPlayer**, **GolfPlayer**, **ChessPlayer**, **CardPlayer**) .

Atenção : Nenhuma destas subclasses deve ter o método correr, pois isolamos esta funcionalidade (Itens 4 e 5). Assim sendo, qualquer destas classes que ainda tenha o método correr, deve ser retirado.

Segue abaixo a implementação de 1 das classes como exemplo.

package roteiro1.parte4;

public class SoccerPlayer extends Player{

public SoccerPlayer(String nome, RunBehavior habilidadeCorrer) {

super(nome, habilidadeCorrer);

}

@Override

public void definirTatica() {

System.out.println(super.nome + " Trabalha em equipe !");

}

}

**Façamos os testes a seguir !!**

7 – Faça os devidos testes usando a classe **TesteJogo** , e na criação de cada objeto jogador passamos como parâmetro a habilidade de correr adequada.

package roteiro1.parte4;

public class TesteJogo {

public static void main(String[] args) {

RunFast correRapido = new RunFast();

RunNoWay naoCorre = new RunNoWay();

System.out.println("Detalhes do Jogador de Tenis");

TennisPlayer guga = new TennisPlayer("Gustavo Kuerten", correRapido);

guga.treinar();

guga.estiloCompetidor();

guga.definirTatica();

guga.correr();

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

System.out.println("Detalhes do Jogador de Futebol");

SoccerPlayer ronaldo = new SoccerPlayer("Ronaldinho Gaucho", correRapido);

ronaldo.treinar();

ronaldo.estiloCompetidor();

ronaldo.definirTatica();

ronaldo.correr();

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

System.out.println("Detalhes do Jogador de Cartas");

CardPlayer joseCartas = new CardPlayer("Jose das Cartas", naoCorre);

joseCartas.treinar();

joseCartas.estiloCompetidor();

joseCartas.definirTatica();

joseCartas.correr();

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

System.out.println("Detalhes do Jogador de Xadrez");

ChessPlayer kasparov = new ChessPlayer("Kasparov", naoCorre);

kasparov.treinar();

kasparov.estiloCompetidor();

kasparov.definirTatica();

kasparov.correr();

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

System.out.println("Detalhes do Jogador de Golf");

GolfPlayer tigerwoods = new GolfPlayer("Tiger Woods", naoCorre);

tigerwoods.treinar();

tigerwoods.estiloCompetidor();

tigerwoods.definirTatica();

tigerwoods.correr();

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

}

}

10 – Utilize uma ferramenta de software qualquer para geração do diagrama de classes para esta etapa do projeto (Sugestão : Astah Community). Obs.: Adicione aqui o diagrama para que seja disponibilizado no teams

Obs: Outra possibilidade para geração dos diagramas é o StarUML : <https://staruml.io/> . Para fazer a engenharia reversa do código siga as orientações conforme o vídeo a seguir : <https://www.youtube.com/watch?v=z6DDuZQArro>

11 – Faça a correspondência entre as classes, comparando o diagrama formal do padrão Strategy e as classes que acabamos de modelar neste projeto.

12 – Neste modelo final em que chegamos, seria possível modificar a habilidade de correr de algum jogador em tempo de execução. Ex.: Um jogador que inicialmente não corre, podemos fazer com que ele passe a correr ? Ou passe a correr lentamente ? Se sim, como isso poderia ser feito ?

13 – Neste último teste de modelagem preservamos o princípio da substituição de Liskov (Um dos princípios do SOLID) ? Ou seja, conseguimos ter os objetos “jogadores” apenas do tipo Player, e facilmente substituir um jogador por outro sem comprometer o funcionamento do sistema ?

13 – Consegue identificar alguma semelhança desta modelagem final com o roteiro 7 do SOLID ?

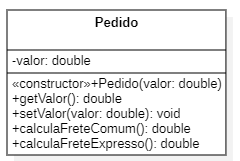
**Pacote : roteiro1.parte5**

Vamos exercitar um pouco a aplicação do padrão Strategy

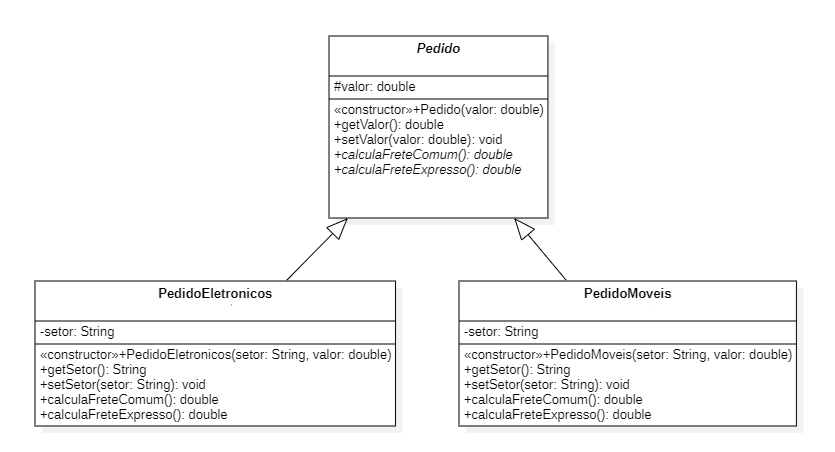
Imagine um novo cenário em que tenha sido implementado um sistema de e-commerce. Neste sistema o cliente pode fazer um pedido de compras que está sujeito a dois tipos de frete :

* Frete Comum – em que é cobrado 5% do valor do pedido
* Frete Expresso – em que é cobrado 10% do valor do pedido

Vamos abstrair alguns detalhes do pedido, mas em um primeiro momento podemos imaginar que uma única classe é capaz de resolver esta questão conforme o exemplo abaixo. Observe que temos 2 métodos para calcular o frete. Um para frete comum e outro para frete expresso.



Acontece que o cenário é um pouco mais complexo. A empresa é dividida em dois departamentos (Eletrônicos e Móveis). Cada um destes departamentos tem especificidades na forma de fazer o pedido, e mais uma vez vamos abstrair detalhes e atributos de cada um dos departamentos, apenas para simplificar a modelagem. Vamos considerar neste modelo que temos apenas o nome do setor nas classes PedidoEletronicos e PedidoMoveis.



Classe Abstrata - Pedido

Métodos abstratos :

- calculaFreteComum

- calculaFreteExpresso

Estes métodos abstratos são implementados nas classes concretas :

- PedidoEletronicos

- PedidoMoveis

Outra questão que devemos considerar é que por questões estratégicas da empresa, nem sempre é possível disponibilizar o frete expresso para um dos departamentos. Ou seja, em algum momento um determinado pedido de um dos departamentos pode ofertar ou não um certo tipo de frete. Além disso, podem surgir outras regras para cálculo do frete no futuro.

Considere que esta modelagem acima é exatamente a versão atual do sistema implementada. Pense como podemos evoluir este projeto e de que forma o padrão strategy pode ajudar neste contexto.

Apresente a solução implementada nesta etapa do roteiro, e construa os diagramas para representar a solução.