Princípios e Padrões de Projeto
Lista de Exercícios 1º Bim.

Prof. Igor Avila Pereira

Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Sul (IFRS)
Divisão de Computação

igor.pereira@riogrande.ifrs.edu.br

1. Strategy

1.1. Um pouco de teoria

O Strategy é um padrão de **projeto comportamental** que transforma um conjunto de comportamentos em objetos e os torna intercambiáveis dentro do objeto de contexto original.

O objeto original, chamado **contexto**, mantém uma referência a um objeto Strategy e o delega a execução do comportamento. Para alterar a maneira como o **contexto** executa seu trabalho, outros objetos podem substituir o objeto Strategy atualmente vinculado por outro.

O Strategy é um padrão de projeto comportamental que permite que você defina uma família de algoritmos, coloque-os em classes separadas, e faça os objetos deles intercambiáveis.

1.2. Exercícios

1.2.1. Loja Virtual

Uma loja virtual possuiu alguns produtos a venda. Os produtos são livros, DVDs e brinquedos. Cada produto possui nome e preço. A mesma loja oferece promoções diferentes a cada mês. Uma promoção regular desconta cada produto em 10% mais um desconto extra varia de 5% a 10% dependendo do mês. Uma liquidação desconta 30% ao preço de cada produto. Há meses que não há promoção descrita. **Implemente o Padrão Strategy.**

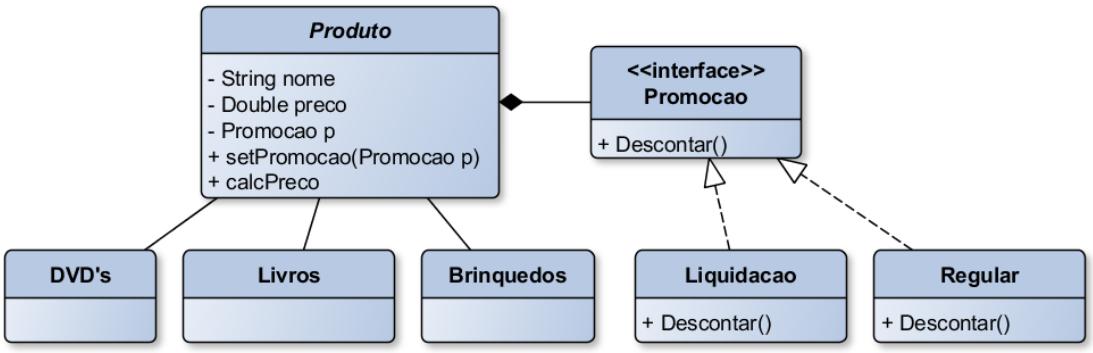

Figure 1. UML: Loja Virtual

2. Observer

2.1. Um pouco de teoria

O Observer é um padrão de **projeto comportamental** que permite que um **objeto notifique outros objetos sobre alterações em seu estado**.

O Observer é um padrão de projeto comportamental que permite que você defina um **mecanismo de assinatura** para **notificar múltiplos objetos sobre quaisquer eventos que aconteçam com o objeto que eles estão observando**.

O padrão Observer fornece uma maneira de assinar e cancelar a assinatura desses eventos para qualquer objeto que implemente uma interface de assinante.

2.2. Exercícios

2.2.1. Telefone

Como projetar um sistema que modele um telefone e todos os objetos que poderiam estar interessados quando ele toca? **Implemente o padrão Observer para o problema proposto.**

Os objetos interessados poderiam ser:

- Pessoas que estejam perto (na mesma sala)
- Uma secretária eletrônica
- Um FAX
- Até um dispositivo de escuta clandestina

Os objetos interessados podem mudar dinamicamente:

- Pessoas entram e saem da sala onde o telefone está.
- Secretárias eletrônicas, FAX, etc. podem ser adicionados ou removidos durante a execução do programa
- Novos dispositivos poderão ser inventados e adicionados em versões futuras do programa

2.2.2. Revista de Fofoca Online

Pense em uma site de uma revista de fofocas online. Nela há uma lista de assinantes. Quando uma nova notícia é cadastrada, todos os assinantes são notificados. Há vários tipos de assinantes: pessoa física, pessoa jurídica, idosos e etc. **Implemente o padrão Observer para o problema proposto.**

3. Command

3.1. Um pouco de teoria

O Command é um padrão que consiste na representação de uma operação como uma classe. A abstração que representa o comando, no formato de uma superclasse ou interface, define um método que deve ser utilizado para executar a operação. Os comandos concretos, que implementam ou estendem a abstração, devem definir como atributo a estrutura de dados necessária para a sua execução. Dessa forma, o comando acaba sendo uma classe que pode representar uma operação do sistema independente de todo o contexto. É essa característica que permite diversas aplicações desse padrão.

3.2. Exercícios

3.2.1. Bloco de Notas

Bloco de notas é uma aplicação que pode editar **documentos** de texto. No bloco de notas há também tem um **menu** responsável por **copiar** e **colar** textos. Um texto pode ser selecionado a partir de qualquer documento e pode ser colado a qualquer outro documento.

> **Implemente o padrão Command para o problema proposto.**
>
> **Dicas:**
>
> - As operações de *copiar* e *colar* devem ser vistas como comandos.
> - Crie uma classe que representa o *Documento*
> - Crie uma classe que representa o item do *Menu*. A classe funciona como *invoker* e deve ser possível "setar" os comandos da aplicação e chamar a execução dos mesmos.

4. State

4.1. Um pouco de teoria

State é um padrão de projeto de software usado quando o comportamento de um objeto muda, dependendo do seu estado.

State pertence ao grupo de padrões comportamentais. Utilizamos ele quando queremos justamente que o comportamento de um objeto mude, de acordo com o seu estado atual.

Quando temos um objeto, onde além de lidarmos com a mudança de estado do mesmo ainda precisamos definir comportamentos diferentes para estados diferentes, tudo no mesmo código, tornamos nosso código mais complicado de entender e manter. Além disto, testes acabam ficam inviáveis.

- **Objetivo/Intenção:** Tornar nosso código mais organizado e coeso quando temos comportamentos diferentes baseados em estados diferentes;
- **Motivação:** Quando temos diversos estados diferentes em nosso domínio e o comportamento dele varia de acordo com o estado atual, tendemos a colocar IF's e outros condicionais em nosso código, tornando ele extenso e difícil de entender/-manter. A ideia do *pattern* e fornecer um modelo para organizarmos e separarmos o que é código relacionado a transição de estado e o que é de comportamento baseado no estado atual;
- **Aplicabilidade:** Utilizamos principalmente em domínios que possuem estados diferentes onde o comportamento pode mudar de acordo com cada estado, construindo basicamente uma máquina de estados, mas mantendo uma organização e separação de responsabilidades em classes concretas para cada estado;

4.2. Exercícios

4.2.1. Status de Uma Compra Online

Implemente o padrão State para simular o andamento (status) de uma compra feita em site de e-commerce.

Figure 2. State - Status de uma Compra via e-commcerce

5. Template Method

5.1. Um pouco de teoria

O Padrão Template Method é um padrão de projeto comportamental que define o esqueleto de um algoritmo na superclasse mas deixa as subclasses sobrescreverem etapas específicas do algoritmo sem modificar sua estrutura.

Um Template Method auxilia na definição de um algoritmo com partes do mesmo definidos por métodos abstratos. As subclasses devem se responsabilizar por estas partes abstratas, deste algoritmo, que serão implementadas, possivelmente de várias formas, ou seja, cada subclasse irá implementar à sua necessidade e oferecer um comportamento concreto construindo todo o algoritmo.

O Template Method fornece uma estrutura fixa, de um algoritmo, esta parte fixa deve estar presente na superclasse, sendo obrigatório uma *classeAbstrata* que possa conter um método concreto, pois em uma interface só é possível conter métodos abstratos que definem um comportamento, esta é a vantagem de ser uma Classe Abstrata porque também irá fornecer métodos abstratos às suas subclasses, que por sua vez herdam este método, por Herança (programação), e devem implementar os métodos abstratos fornecendo um comportamento concreto aos métodos que foram definidos como abstratos. Com isso certas partes do algoritmo serão preenchidos por implementações que irão variar, ou seja, implementar um algoritmo em um método, postergando a definição de alguns passos do algoritmo, para que outras classes possam redefini-los.

5.2. Exercícios

5.2.1. Frete

Uma empresa criou uma classe responsável pelo cálculo do frete. Na medida que a empresa cresce normas formas de venda surgem e, consequentemente, novas modalidades para o cálculo do frete também começam a surgir. Entretanto, a empresa percebeu que o cálculo do frete, independente da modalidade, pode ser dividido em 3 etapas:

1. taxa do frete devido a modalidade e a distância;
2. valor do frete devido ao peso do produto e,
3. o desconto devido aos cupons promocionais (caso haja algum).

A única etapa que muda é a primeira pois para cada modalidade de frete (Sedex, convencional ou por Transportadora) temos uma regra de negócio diferente e um preço base diferente.

Implemente o Template Method para cada uma das modalidades de frete.

6. Iterator

6.1. Um pouco de teoria

"Fornecer um meio de acessar, sequencialmente, os elementos de um objeto agregado sem expor sua representação subjacente".

Então utilizando o padrão Iterator nós poderemos acessar os elementos de um conjunto de dados sem conhecer sua implementação, ou seja, sem a necessidade de saber se será utilizado ArrayList ou Matriz.

6.2. Exercícios

6.2.1. Canais de TV a Cabo

Imagine que você trabalha em uma empresa de Tv a Cabo.Você recebeu a tarefa de mostrar a lista de canais que a empresa oferece. Ao procurar os desenvolvedores dos canais você descobre que existe uma separação entre os desenvolvedores que cuidam dos **canais de esportes** e os que cuidam dos **canais de filmes**. O problema começa quando você percebe que, apesar de ambos utilizarem uma **lista de canais**, os desenvolvedores dos canais de filmes utilizaram Matriz (Array convencional) para representar a lista de canais e os desenvolvedores dos canais de esportes utilizaram ArrayList.

Você não quer padronizar as listas pois todo o resto do código dos sistema que cada equipe fez utiliza sua própria implementação (ArrayList ou Matriz). Como construir o programa que vai exibir o nome dos canais?

A solução mais simples é pegar a lista de canais e fazer dois loops, um para percorrer o ArrayList e outro para percorrer a Matriz, e em cada *loop* exibir o nome dos canais. A impressão dos canais seria algo desse tipo:

```
1 ArrayList<Canal> arrayListDeCanais = new ArrayList<Canal>();
2 Canal[] matrizDeCanais = new Canal[5];
3 for (Canal canal : arrayListDeCanais) {
4     System.out.println(canal.nome);
5 }
6 for (int i = 0; i < matrizDeCanais.length; i++) {
7     System.out.println(matrizDeCanais[i].nome);
8 }
```

No entanto é fácil perceber os problemas desta implementação sem ao menos ver o código, pois, caso outra equipe utilize outra estrutura para armazenar a lista de canais você deverá utilizar outro loop para imprimir. Pior ainda é se você precisar realizar outra operação com as listas de canais terá que implementar o mesmo método para cada uma das listas.

Implemente o padrão Iterator para resolver este problema.

7. UML

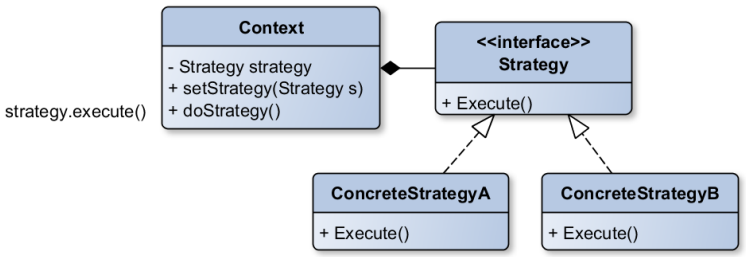

Figure 3. Strategy

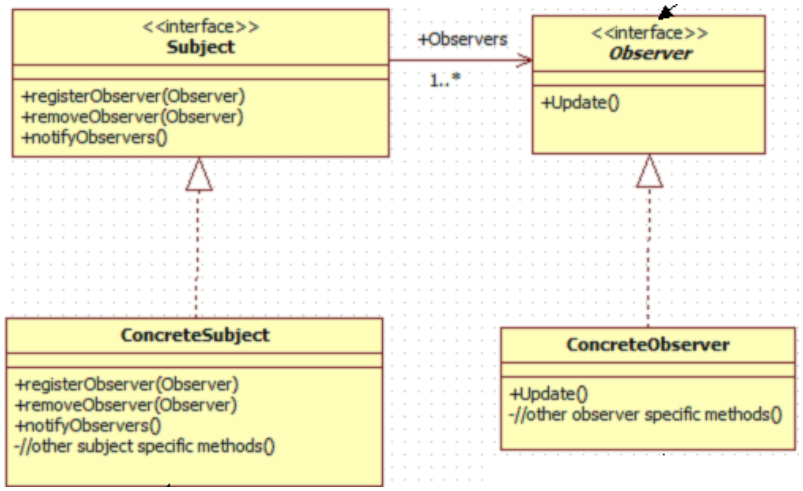

Figure 4. Observer

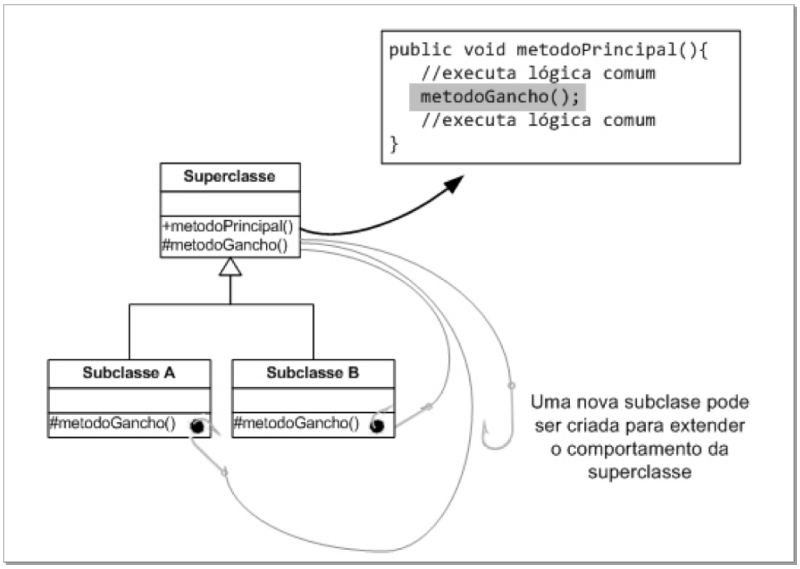

Figure 5. Template Method.

Obs: o **metodoPrincipal** geralmente é definido como *final* e o(s) **método(s) gancho(s)** são definidos como *abstract*.

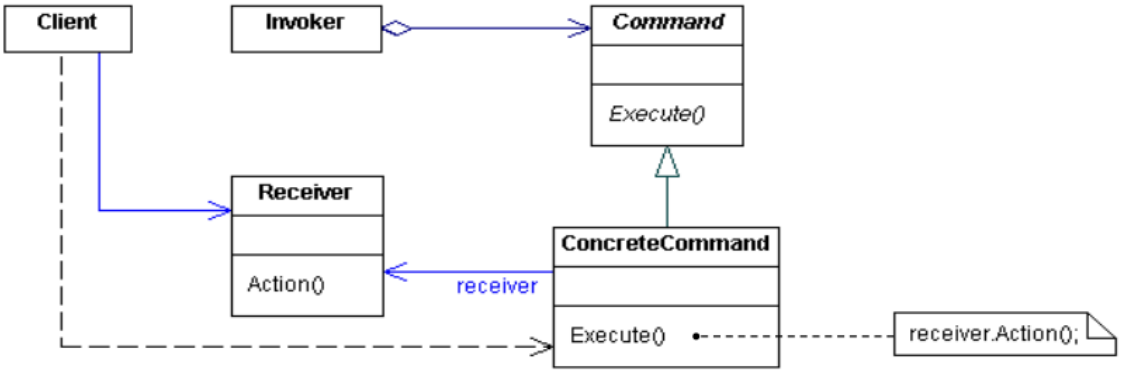

Figure 6. Command

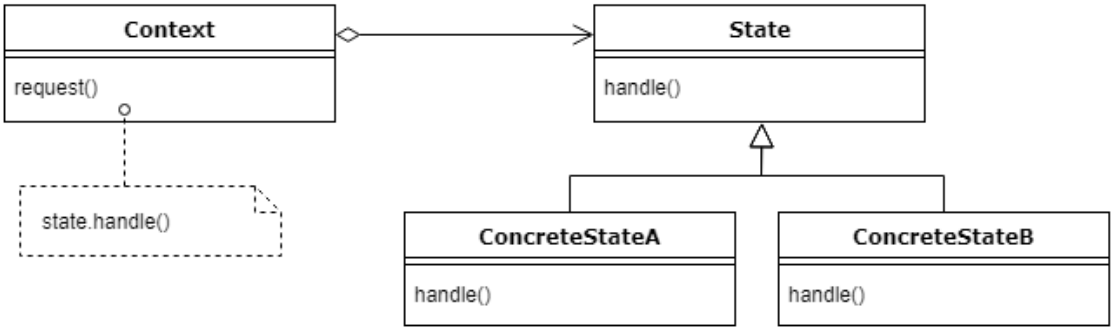

Figure 7. State

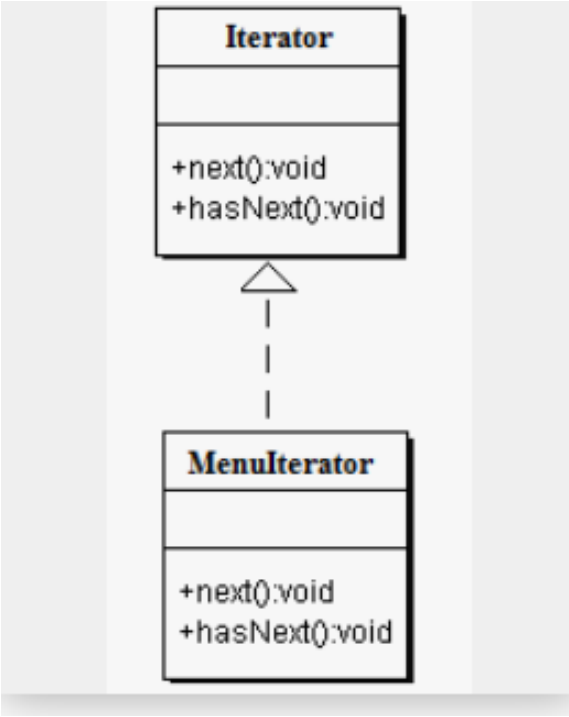

Figure 8. Iterator