Universidade Federal de Campina Grande
Centro de Engenharia Eletrica e Informática
Departamento de Sistemas e Computação
Graduação em Ciência da Computação

Exercício sobre listas ligadas

Objetivo: Conhecer a noção de padrões de projeto, padrão Strategy e sua relação com Comparator.

Padrões de Projeto

Desde sua criação e uso, a orientação a objetos estabeleceu um paradigma focado nas estruturas ao invés das operações. Dessa forma, ao conceber/modelar um sistema, o projetista pensa primeiro nas estruturas a serem manipuladas (entidades) e depois nas operações sobre ela (o paradigma imperativo inverte essa ordem). A justificativa pra isso é que os dados mudam menos do que as operações e, dessa forma, devem ser prioridade no design.

Apesar de trazer recursos muito poderosos de abstração, encapsulamento, reuso, modularidade e coesão, a Orientação a Objetos possui um limite de aplicação quando o problema requer diversos componentes trabalhando em conjunto com um determinado propósito. Para ir mais além, os Padrões de Projeto OO, fazem uso da própria OO para combinar recursos e alcançar soluções que aumentem ainda mais o reuso.

Padrões de projeto são soluções construídas ao longo dos anos como fruto da experiência prática do desenvolvimento de software. Problemas/situações recorrentes aparecem continuamente no desenvolvimento e isso motivou a busca por soluções para essas situações corriqueiras.

Foi com o desenvolvimento OO que os padrões de projeto foram estudados, agrupados e difundidos na comunidade, publicado no livro *Design Patterns – Elements of Reusable Object-Oriented Software*. Um padrão é caracterizado por:

- *Intenção* – a motivação/contexto para o uso do padrão, pois padrões distintos podem ter a mesma estrutura mas diferentes intenções.
- *Terminologia e estrutura* – funcionalidade definida a partir de métodos e tipos (interfaces e classes) envolvidos no padrão.
- *Vantagens e desvantagens* – cada padrão apresenta vantagens e desvantagens que devem ser levados em consideração quando de seu uso.

Procure ler um pouco mais sobre padrões de projeto nestas referências: referência 1 e referência 2.

Padrão de Projeto Strategy

Intenção

A intenção do padrão de projeto Strategy é facilitar a mudança de comportamento de uma classe através da mudança do algoritmo interno sem modificação na classe original.

Estrutura

A estrutura do padrão Strategy envolve uma definição abstrata da estratégia (*interface* com serviço ou método), um *contexto* de uso (classe cliente que vai usar o método da estratégia) e um a implementação concreta (*classe*) da estratégia.

Vantagens e desvantagens

O principal benefício do padrão Strategy é a flexibilidade, pois clientes podem escolher um algoritmo específico (estratégia), ou também novas implementações da estratégia podem ser usadas sem causar mudanças no cliente.

Para implementar uma nova estratégia/algoritmo é preciso escrever uma nova classe implementando a interface da estratégia. Isso gera um componente a mais no cenário.

Comparator e Design Pattern Strategy

Um cenário de uso do Comparator é uma exemplo clássico do padrão Strategy. Imagine que se deseja implementar uma lista encadeada ordenada, onde a ordenação é estabelecida por um componente externo (um comparator). A lista em si seria o cliente do comparator, usando-o para qualquer operação e comparação entre os elementos manipulados. O Comparator em si seria a interface definindo a estratégia. E a classe implementando o Comparator seria o algoritmo/estratégia a ser aplicado.

Esse cenário traz duas vantagens de imediato:

- Todo o código relativa a comparação de elementos é designada para um componente externo. Isso "limpa" mais o código do cliente em si.
- Uso de uma mesma lista com diferentes propósitos. Por exemplo, uma mesma lista ordenada (classe) pode ser utilizada para guardar dados em ordem ascendente ou descendente.

Atividade

Procure responder aos seguintes questionamentos:

- Seria possível a existência de uma coleção (lista simplesmente encadeada mesmo) ser capaz de guardar seus elementos de forma ordenada, tendo essa propriedade sempre válida após a execução de qualquer operação? Se sim, como?
- E se desejarmos que a ordem em que essa lista guarde seus elementos seja flexível (crescente ou decrescente), qual o esforço de sua solução no item anterior?
- Caso você ache que é possível, proponha um modelo/forma de resolver o problema de forma elegante.

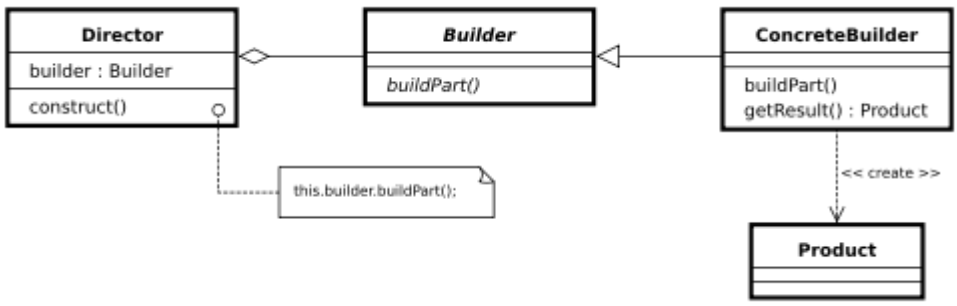
Padrão de Projeto Builder

Intenção

A intenção do padrão de projeto Builder é eliminar a necessidade de definição de diversos construtores para uma mesma classe no sentido de flexibilizar as diferentes formas de como objetos podem ser construídos com seus atributos, evitando a necessidade de diversos construtores combinando os atributos. Dessa forma, a intenção do padrão Builder é de separar a construção de objetos complexos de sua representação e assim possibilitar que o processo de criação (delegado a um outro componente chamado Builder) possa criar diferentes representações.

Estrutura

A estrutura do padrão Builder envolve a representação (classe que deve ser criada) a interface de criação (*o Builder*) e a implementação concreta do Builder. A classe a ser criada possui um Builder interno que é usado para criar objetos de seu tipo.



Vantagens e desvantagens

- Permite variar a representação interna dos produtos a serem criados
- Encapsula o código de construção e representação
- Provê controle sobre os passos do processo de construção
- (Desvantagem) Requer a criação de um Builder separado para cada tipo de produto que precisa ser instanciado.

Exemplo de implementação em Java

Imagine uma classe cliente como seguinte código:

```
public class Cliente {
	String id;
	String cpf;
	String nome;
	String tipoSanguineo;
	int idade;
}
```

Caso se deseje ter diferentes possibilidades de se instanciar um cliente (passando apenas um, dois, três ou quatro atributos, por exemplo), precisaríamos de diversos construtores na classe. Com o uso de um builder, nós conseguimos construir um objeto que contém os campos preenchidos de acordo com a demanda e daí pegamos esse objeto e usamos para construir um Cliente com o que foi preenchido. O código dessa classe fica dentro da própria classe Cliente, que passa a ter um construtor recebendo o Builder e preenchendo os campos do objeto com o que existe no Builder.

```
public static class Builder{
            String id;
            String cpf;
            String nome;
            String tipoSanguineo;
            int idade;

            public Cliente.Builder id(String id){
                  this.id = id;
                  return this;
            }

            public Cliente.Builder cpf(String cpf){
                  this.cpf = cpf;
                  return this;
            }

            public Cliente.Builder nome(String nome){
                  this.nome = nome;
                  return this;
            }
            public Cliente.Builder tipoSanguineo(String tipoSanguineo){
                  this.tipoSanguineo = tipoSanguineo;
                  return this;
            }

            public Builder idade(int idade){
                  this.idade = idade;
                  return this;
            }

            public Cliente build(){
                  return new Cliente(this);
            }
      }

  private Cliente(Builder builder){
            this.id = builder.id;
            this.cpf = builder.cpf;
            this.nome = builder.nome;
            this.tipoSanguineo = builder.tipoSanguineo;
            this.idade = builder.idade;
   }

}
```

Agora a criação de objetos da classe Cliente fica a cargo do seu Builder, onde os atributos podem ser inicializados sob demanda, sem precisar definir construtores para combinar as diversas formas de como objetos são criados. Um código exemplo do padrão builder seria:

```
Cliente vazio = new Cliente.Builder().build();
            Cliente clienteComId = new Cliente.Builder()
                                        .id("123")
                                        .build();
Cliente clienteComIdCpf = new Cliente.Builder()
                                        .id("123")
                                        .cpf("11111111-11")
                                        .build();
Cliente clienteComCpfNome = new Cliente.Builder()
                                        .cpf("11111111-11")
                                        .nome("Ted Bear")
                                        .build();
```

Note que para isntanciar os clientes agora precisa-se instanciar o builder, preencher os atributos necessários e depois chamar o método build() no mesmo para que o cliente seja preenchido com os campos iniciados no Builder.