Strategy:

É um padrão comportamental que delega as responsabilidades adquiridas pelas entidades, atribuindo ao comportamento de cada uma, porque assim existe a distribuição das responsabilidades.

O padrão Strategy permite definir novas operações sem alterar as classes dos elementos sobre os quais opera, devido a criação de uma hierarquia de classes partindo de uma interface, o que gera classes filhas em um mesmo contexto. Então assim a classe que fizer o contrato com a Interface não tem a necessidade de saber qual das classes que foram estendidas está sendo passada como parâmetro, pois ela só precisa do comportamento.

**package** padroesDeProjeto;

**public** **class** main {

**public** **static** **void** main(String[] args) {

Calabresa calabresa = **new** Calabresa();

Bacon bacon = **new** Bacon();

Pedido pedido1 = **new** Pedido();

pedido1.addPizza(bacon);

pedido1.addPizza(calabresa);

System.***out***.println(pedido1.retornaTotal());

}

}

**package** padroesDeProjeto;

**public** **interface** Pizza {

**double** retornarPreco();

}

**package** padroesDeProjeto;

**public** **class** Calabresa **implements** Pizza{

@Override

**public** **double** retornarPreco() {

**return** 38.50;

}

}

**package** padroesDeProjeto;

**public** **class** Bacon **implements** Pizza{

@Override

**public** **double** retornarPreco() {

**return** 34.00;

}

}

**package** padroesDeProjeto;

**public** **class** FrangoComCatupiry **implements** Pizza{

@Override

**public** **double** retornarPreco() {

**return** 42.00;

}

}

**package** padroesDeProjeto;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Pedido {

**private** **double** total = 0;

**private** List<Pizza> pizzas = **new** ArrayList<>();

**public** **void** addPizza(Pizza pizza) {

pizzas.add(pizza);

}

**public** **double** retornaTotal() {

**for** (Pizza pizza : pizzas) {

total += pizza.retornarPreco();

}

**return** total;

}

}

Builder:

É um padrão de projeto de software criacional, ou seja, que permite a separação da construção de um objeto, de forma que o mesmo processo de construção possa criar diferentes representações.

A classe precisa ter uma ligação de composição com a classe do objeto mais complexo (está sendo abstrata), e dependendo da classe filha que for chamada o objeto é construído daquela forma especifica. Quem está chamando para a construção do objeto não precisa saber qual classe foi implementada, apenas precisa do objeto construído. Dessa forma cada classe filha é responsável pelo seu tipo de construção e seus métodos e atributos permanecem encapsulados.

