03

Aplicando Padrões de projeto

Começando deste ponto? Você pode fazer o DOWNLOAD (https://s3.amazonaws.com/caelum-online-public/laboratorio-java2/cap4.zip) do projeto completo do capítulo anterior e continuar seus estudos a partir deste capítulo.

Aplicando Padrões de projeto

Nossos indicadores e o design pattern Strategy

Nosso gerador de gráficos já está interessante, mas no momento ele só consegue plotar a Média Móvel Simples do fechamento da série. Nosso cliente certamente precisará de outros indicadores técnicos como o de Média Móvel Ponderada ou ainda indicadores mais simples como os de Abertura ou de Fechamento.

Esses indicadores, similarmente à `MediaMovelSimples`, devem calcular o valor de uma posição do gráfico baseado na `SerieTemporal` que ele atende.

Se tivermos esses indicadores todos, precisaremos que o `GeradorModeloGrafico` consiga plotar cada um desses gráficos. Terminaremos com uma crescente classe `GeradorModeloGrafico` com métodos **extremamente** parecidos:

```
public class GeradorModeloGrafico {
    //...

    public void plotaMediaMovelSimples() {
            MediaMovelSimples indicador = new MediaMovelSimples();
            LineChartSeries linha = new LineChartSeries();
            linha.setLabel("MMS - Fechamento");

            for (int i = comeco; i <= fim; i++) {
                double valor = indicador.calcula(i, serie);
                linha.set(i, valor);
            }
            this.modeloGrafico.addSeries(linha);

    }

    public void plotaMediaMovelPonderada() {
            MediaMovelPonderada indicador = new MediaMovelPonderada();
            LineChartSeries linha = new LineChartSeries();
            linha.setLabel("MMP - Fechamento")


            for (int i = comeco; i <= fim; i++) {
                double valor = indicador.calcula(i, serie);
                linha.set(i, valor);
            }
            this.modeloGrafico.addSeries(linha);

    }
```

```
    //...
}
```

O problema é que cada vez que criarmos um indicador técnico diferente, um novo método deverá ser criado na classe `GeradorModeloGrafico`. Isso é uma indicação clássica de acoplamento no sistema.

Como resolver esses problemas de acoplamento e de código parecidíssimo nos métodos? Será que conseguimos criar um único método para plotar e passar como argumento qual *indicador técnico* queremos plotar naquele momento?

Note que a diferença entre os métodos está apenas no `new` do indicador escolhido e na legenda do gráfico. O restante é precisamente igual.

```
public void plotaMediaMovelSimples() {
    LineChartSeries linha = new LineChartSeries();
    linha.setLabel(linha.setLabel("MMS - Fechamento");
    MediaMovelSimples indicador = new MediaMovelSimples();
    for (int i = comeco; i <= fim; i++) {
        double valor = indicador.calcula(i, serie);
        linha.set(i, valor);
    }
    this.modeloGrafico.addSeries(linha);
}
public void plotaMediaMovelPonderada() {
    LineChartSeries linha = new LineChartSeries();
    linha.setLabel(linha.setLabel("MMP - Fechamento");
    MediaMovelPonderada indicador = new MediaMovelPonderada();
    for (int i = comeco; i <= fim; i++) {
        double valor = indicador.calcula(i, serie);
        linha.set(i, valor);
    }
    this.modeloGrafico.addSeries(linha);
}
```

A orientação a objetos nos dá a resposta: **polimorfismo!** Repare que nossos dois indicadores possuem a mesma assinatura de método, parece até que eles assinaram o mesmo *contrato*. Vamos definir então a *interface* `Indicador`:

```
public interface Indicador {
    public abstract double calcula(int posicao, SerieTemporal serie);
}
```

Podemos fazer as classes `MediaMovelSimples` e `MediaMovelPonderada` implementarem a interface `Indicador`. Com isso, podemos criar apenas um método na classe do gráfico que recebe um `Indicador` qualquer. O objeto do indicador será responsável por calcular o valor no ponto pedido (método `calcula`) e, também, pela legenda do gráfico (método `toString`)

```
public class GeradorModeloGrafico {

    public void plotaIndicador(Indicador indicador) {
        LineChartSeries linha = new LineChartSeries(x);
        linha.setLabel(linha.setLabel("MMP - Fechamento");
        for (int i = comeco; i <= fim; i++) {
            double valor = indicador.calcula(i, serie);
            linha.set(i, valor);
        }
        this.modeloGrafico.addSeries(linha);
```

```
    }
}
```

Na hora de desenhar os gráficos, chamaremos sempre o `plotaIndicador`, passando como parâmetro qualquer classe que **seja um** `Indicador`:

```
GeradorModeloGrafico gerador = new GeradorModeloGrafico(serie, 2, 40);
gerador.plotaIndicador(new MediaMovelSimples());
gerador.plotaIndicador(new MediaMovelPonderada());
```

A ideia de usar uma *interface comum* é ganhar *polimorfismo* e poder trocar os indicadores. Nosso método plota qualquer `Indicador`, isto é, quando criarmos ou removermos uma implementação de indicador, não precisaremos mexer no `GeradorModeloGrafico`: ganhamos flexibilidade e aumentamos a coesão. Podemos ainda criar novos indicadores que implementem a interface e passá-los para o gráfico sem que nunca mais mexamos na classe `GeradorModeloGrafico`.

Por exemplo, imagine que queremos um gráfico simples que mostre apenas os preços de fechamento. Podemos considerar a evolução dos preços de fechamento como um `Indicador`:

```
public class IndicadorFechamento implements Indicador {

    public double calcula(int posicao, SerieTemporal serie) {
        return serie.getCandle(posicao).getFechamento();
    }
}
```

Ou criar ainda classes como `IndicadorAbertura`, `IndicadorMaximo`, etc.

Temos agora vários **indicadores** diferentes, cada um com sua própria **estratégia** de cálculo do valor, mas todos obedecendo a mesma interface: dada uma série temporal e a posição a ser calculada, eles devolvem o valor do indicador. Note que o `plotaIndicador` não recebe dados, mas sim a forma de manipular esses dados, a estratégia para tratá-los. Esse é o design pattern chamado de **Strategy**.

Design Patterns

Design Patterns são aquelas soluções catalogadas para problemas clássicos de orientação a objetos, como este que temos no momento: encapsular e ter flexibilidade.

A fábrica de `Candles` apresentada em outro capítulo e o método que nos auxilia a criar séries para testes na `GeradorDeSerie` são exemplos, respectivamente, dos padrões *Abstract Factory* e *Factory Method*. No caso que estamos estudando agora, o *Strategy* está nos ajudando a deixar a classe `GeradorModeloGrafico` isolada das diferentes formas de cálculo dos indicadores.

Indicadores mais elaborados e o Design Pattern Decorator

Vimos no capítulo de refatoração que os analistas financeiros fazem suas análises sobre indicadores mais elaborados, como por exemplo *Médias Móveis*, que são *calculadas* a partir de outros indicadores. No momento, nossos algoritmos de médias móveis sempre calculam seus valores sobre o preço de fechamento. Mas, e se quisermos calculá-las a partir

de outros indicadores? Por exemplo, o que faríamos se precisássemos da *média móvel simples do preço máximo*, da abertura ou de outro indicador qualquer?

Criaríamos classes como `MediaMovelSimplesAbertura` e `MediaMovelSimplesMaximo` ? Que código colocaríamos lá? Provavelmente, copiaríamos o código que já temos e apenas trocaríamos a chamada do `getFechamento` pelo `getAbertura` e `getMaximo` .

A maior parte do código seria a mesma e não estamos reaproveitando código - copiar e colar código não é reaproveitamento, é uma forma de nos dar dor de cabeça no futuro ao ter que manter 2 códigos idênticos em lugares diferentes.

Queremos calcular médias móveis de fechamento, abertura, volume, etc, sem precisar copiar essas classes de média. Na verdade, o que queremos é calcular a média móvel baseado em algum *outro indicador*. Já temos a classe `IndicadorFechamento` e é trivial implementar outros como `IndicadorAbertura` , `IndicadorMinimo` , etc.

A `MediaMovelSimples` é um `Indicador` que vai depender de algum *outro* `Indicador` para ser calculada (por exemplo o `IndicadorFechamento`). Queremos chegar em algo assim:

```
MediaMovelSimples mms = new MediaMovelSimples(new IndicadorFechamento());
// ou...
MediaMovelSimples mms = new MediaMovelPonderada(new IndicadorFechamento());
```

Repare na flexibilidade desse código. O cálculo de média fica totalmente independente do dado usado e, toda vez que criarmos um novo indicador, já ganhamos a média móvel desse novo indicador de brinde. Vamos fazer então nossa classe de média receber algum outro `Indicador` :

```
public class MediaMovelSimples implements Indicador {

    private final Indicador outroIndicador;

    public MediaMovelSimples(Indicador outroIndicador) {
        this.outroIndicador = outroIndicador;
    }

    // ... calcula ...
}
```

E, dentro do método `calcula` , em vez de chamarmos o `getFechamento` , delegamos a chamada para o `outroIndicador` :

```
@Override
public double calcula(int posicao, SerieTemporal serie) {
    double soma = 0.0;
    for (int i = posicao; i > posicao - 3; i--) {
        soma += outroIndicador.calcula(i, serie);
    }
    return soma / 3;
}
```

Nossa classe `MediaMovelSimples` recebe um outro indicador que **modifica um pouco** os valores de saída - ele complementa o algoritmo da média! Passar um objeto que modifica um pouco o comportamento do seu é uma solução

clássica para ganhar em **flexibilidade** e, como muitas soluções clássicas, ganhou um nome nos design patterns de **Decorator**.

O que aprendemos?

- Utilizar Interfaces para generalizar indicadores.
  - O que é um Design Pattern.
  - Um exemplo de refatoração mais prática.
  - O Design Pattern Decorator.
  - O Design Patter Strategy.
  - Aprendendo outros casos para aplicar Design Patterns