CAPÍTULO 10

Aplicando Padrões de projeto

"Estamos todos na sarjeta, mas alguns de nós estão olhando para as estrelas."
— Oscar Wilde

10.1 – 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 chartSerie = new LineChartSeries("MMS - Fechamento");

      for (int i = comeco; i <= fim; i++) {
        double valor = indicador.calcula(i, serie);
        chartSerie.set(i, valor);
      }
      this.modeloGrafico.addSeries(chartSerie);
      this.modeloGrafico.setLegendPosition("w");
      this.modeloGrafico.setTitle("Indicadores");
  }

  public void plotaMediaMovelPonderada() {
```

```
        MediaMovelPonderada indicador = new MediaMovelPonderada();
        LineChartSeries chartSerie = new LineChartSeries("MMP - Fechamento");

        for (int i = comeco; i <= fim; i++) {
          double valor = indicador.calcula(i, serie);
          chartSerie.set(i, valor);
        }
        this.modeloGrafico.addSeries(chartSerie);
        this.modeloGrafico.setLegendPosition("w");
        this.modeloGrafico.setTitle("Indicadores");
    }

    //...
}
```

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 chartSeries = new LineChartSeries("MMS - Fechamento");
    MediaMovelSimples indicador = new MediaMovelSimples();
    for (int i = comeco; i <= fim; i++) {
        double valor = indicador.calcula(i, serie);
        chartSeries.set(i, valor);
    }
    modeloGrafico.addSeries(chartSeries);
}

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

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 chartSerie = new LineChartSeries(indicador.toString());

    for (int i = comeco; i <= fim; i++) {
      double valor = indicador.calcula(i, serie);
      chartSeries.set(i, valor);
    }
    this.modeloGrafico.addSeries(chartSeries);
    this.modeloGrafico.setLegendPosition("w");
    this.modeloGrafico.setTitle("Indicadores");
  }
}
```

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.

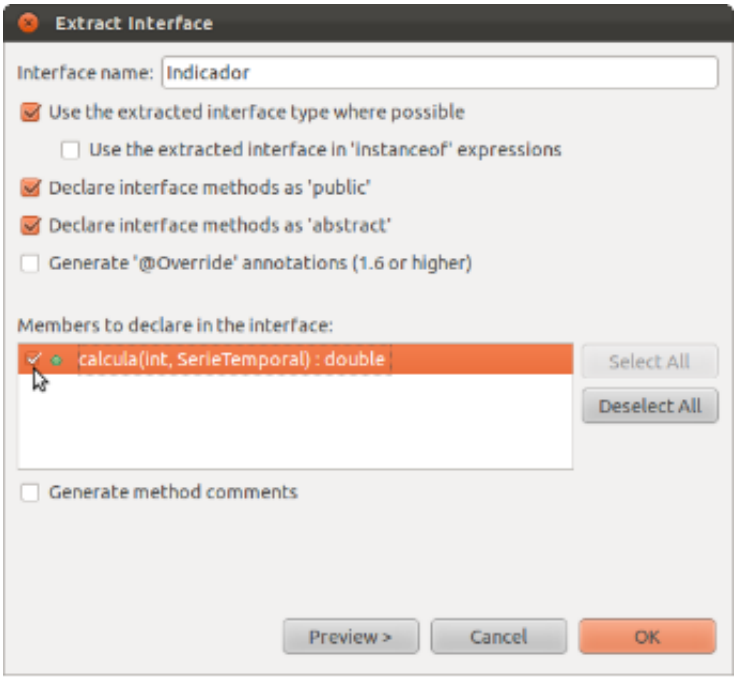
A fábrica de `Candles` apresentada em outro capítulo e o método que nos auxilia a criar séries para testes na `GeradorDeSerie` são exemplos, respectivamente, dos padrões *Abstract Factory* e *Factory Method*. No caso que estamos estudando agora, o *Strategy* está nos ajudando a deixar a classe `GeradorModeloGrafico`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

10.2 – Exercícios: refatorando para uma interface e usando bem os testes

1. Já que nossas classes de médias móveis são indicadores técnicos, começaremos extraindo a interface de um `Indicador` a partir dessas classes.

Abra a classe `MediaMovelSimples` e use o **ctrl + 3** *Extract interface*. Selecione o método `calcula` e dê o nome da interface de `Indicador`:

Atenção: A interface deve ficar dentro do pacote `br.com.caelum.argentum.indicadores`:



> **Effective Java**
>
> Item 52: Refira a objetos pelas suas interfaces

2. Na classe `MediaMovelPonderada` coloque agora o `implements Indicador` nela.

```
public class MediaMovelPonderada implements Indicador {
  ...
}
```

3. Vamos criar também uma classe para, por exemplo, ser o indicador do preço de fechamento, o `IndicadorFechamento`, no pacote `br.com.caelum.argentum.indicadores`, que implementa a interface `Indicador`.

Note que, ao adicionar o trecho `implements Indicador` à classe, o Eclipse mostra um erro de compilação. Usando o **ctrl + 1**, escolha a opção *Add unimplemented methods* para que ele crie toda a assinatura do método `calcula`.

No final, faltará preencher apenas a linha destacada:

```
public class IndicadorFechamento implements Indicador {

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

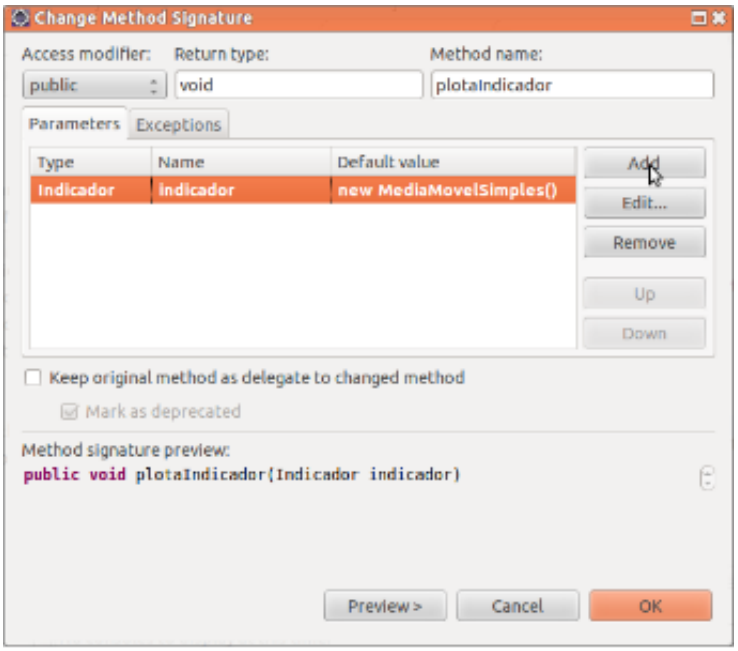
4. De maneira análoga vamos criar o `IndicadorAbertura`, também no pacote

`br.com.caelum.argentum.indicadores`, que implementa a interface `Indicador`:

```
public class IndicadorAbertura implements Indicador {

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

5. Volte para a classe `GeradorModeloGrafico` e altere o método `plotaMediaMovelSimples` usando o atalho **alt + shift + C**. Você precisará alterar o nome para `plotaIndicador` e adicionar um parâmetro. Veja com atenção o *screenshot* abaixo:



Ignore o erro e, quando essa refatoração terminar, remova a linha que declarava o indicador. Seu método terminará assim:

```
public void plotaIndicador(Indicador indicador) {
  LineChartSeries chartSeries = new LineChartSeries(indicador.toString());

  for (int i = comeco; i <= fim; i++) {
    double valor = indicador.calcula(i, serie);
    chartSeries.set(i, valor);
  }
  this.modeloGrafico.addSeries(chartSeries);
  this.modeloGrafico.setLegendPosition("w");
  this.modeloGrafico.setTitle("Indicadores");
}
```

6. Repare que estamos chamando o método `toString()` do indicador no método `plotaIndicador()`. Vamos sobrescrevê-lo em todos os indicadores criados.

Sobrescreva na classe `MediaMovelSimples`:

```
public class MediaMovelSimples implements Indicador{
  // método calcula

  public String toString() {
    return "MMS de Fechamento";
  }
}
```

Sobrescreva na classe `MediaMovelPonderada`:

```
public class MediaMovelPonderada implements Indicador{
  // método calcula

  public String toString() {
    return "MMP de Fechamento";
  }
}
```

Também na classe `IndicadorFechamento`:

```
public class IndicadorFechamento implements Indicador {
  // método calcula

  public String toString() {
    return "Fechamento";
  }
}
```

E por último na classe `IndicadorAbertura`:

```
public class IndicadorAbertura implements Indicador {
  // método calcula

  public String toString() {
    return "Abertura";
  }
}
```

10.3 - Exercícios opcionais

1. Nossos cálculos de médias móveis são sempre para o intervalo de 3 dias. Faça com que o intervalo seja parametrizável. As classes devem receber o tamanho desse intervalo no construtor e usar esse valor no algoritmo de cálculo.

Não esqueça de fazer os testes para essa nova versão e alterar os testes já existentes para usar esse cálculo novo. Os testes já existentes que ficarem desatualizados aparecerão com erros de compilação.

2. Toda refatoração deve ser acompanhada dos testes para garantir que não quebramos nada! Rode os testes e veja se as mudanças feitas até agora mudaram o comportamento do programa.

Se você julgar necessário, acrescente mais testes à sua aplicação refatorada.

10.4 - 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**.

> **Também é um composite!**
>
> Note que, agora, nossa `MediaMovelSimples` **é um** `Indicador` e também **tem**

um outro `Indicador`. Já vimos antes outro tipo que se comporta da mesma forma, você se lembra?

Assim como os componentes do Swing, nossa `MediaMovelSimples` se tornou **também** um exemplo de *Composite*.

10.5 – Exercícios: Indicadores mais espertos e o Design Pattern Decorator

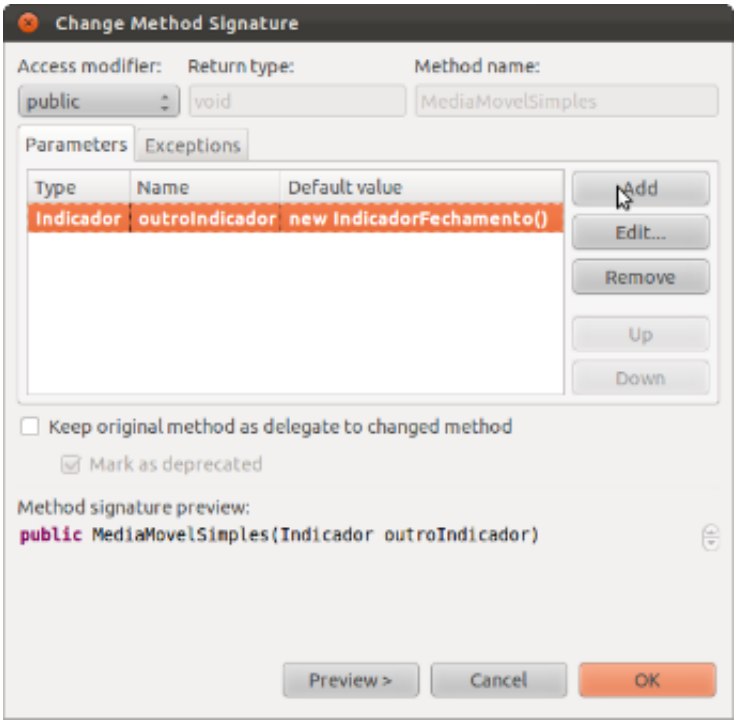
1. Faremos uma grande mudança agora: nossas médias devem receber como argumento um outro indicador, formando o *design pattern* Decorator, como visto na explicação.

Para isso, crie o construtor padrão (sem parâmetros) da `MediaMovelSimples` usando o atalho de sua preferência:

a. Comece a escrever o nome da classe e mande autocompletar: `Med`**<ctrl + espaço>**;

b. Use o criador automático de construtores: **ctrl + 3** *Constructor*.

2. Modifique o construtor usando o atalho **alt + shift + C** para adicionar o parâmetro do tipo `Indicador` chamado `outroIndicador` e com valor padrão `new IndicadorFechamento()`.



Agora, com o cursor sobre o parâmetro `outroIndicador`, faça **ctrl + 1** e guarde esse valor em um novo atributo, selecionando *assign parameter to new field*.

3. Troque a implementação do método `calcula` para chamar o `calcula` do `outroIndicador`:

```
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;
}
```

4. Lembre que toda refatoração **deve** ser acompanhada dos testes correspondentes. Mas ao usar a refatoração do Eclipse no construtor da nossa classe `MediaMovelSimples`, a IDE evitou que quebrássemos os testes já passando o `IndicadorFechamento` como parâmetro padrão para todos eles!

Agora, rode os testes novamente e tudo **deve** continuar se comportando exatamente como antes da refatoração. Caso contrário, nossa refatoração não foi bem sucedida e seria bom reverter o processo todo.

5. Modifique também a classe `MediaMovelPonderada` para também ter um *Decorator*.

Isto é, faça ela também receber um `outroIndicador` no construtor e delegar a chamada a esse indicador no seu método `calcula`, assim como fizemos com a `MediaMovelSimples`.

6. (opcional)Faça um teste de unidade na classe `MediaMovelSimplesTest` que use receba um `IndicadorAbertura` vez do de fechamento. Faça também para quaisquer outros indicadores que você crie.

Conheça os cursos online Alura.

CAPÍTULO ANTERIOR:

Gráficos interativos com Primefaces

PRÓXIMO CAPÍTULO:

A API de Reflection

Casa do Código

Twitter