CAPÍTULO 11

A API de Reflection

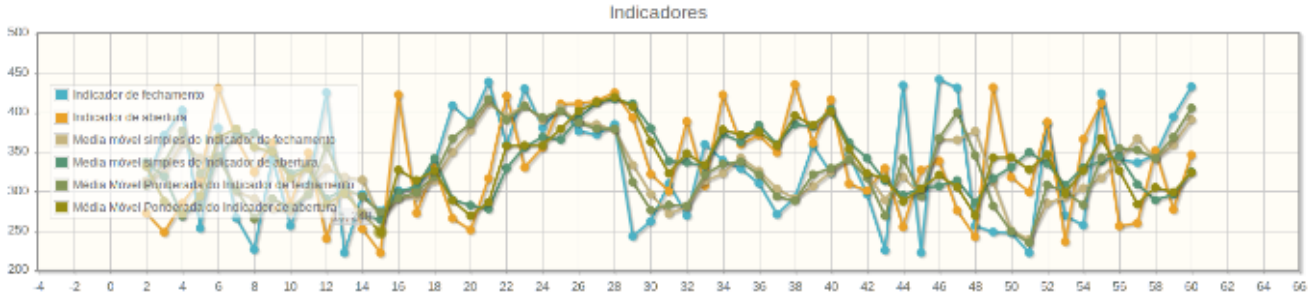
Até agora, ao acessar a aplicação pelo navegador, o gráfico e a tabela de negociações serão gerados automaticamente. Não há nenhuma forma de personalizar o gráfico, como por exemplo a opção de ver apenas um dos indicadores por vez.

Vamos melhorar a interface e adicionar um formulário que oferece para o usuário as seguintes opções:

- Tipo de indicador (abertura ou fechamento)
- Tipo de média (móvel simples ou móvel ponderada)

11.1 – Escolhendo qual gráfico plotar

Nesse momento, apesar de conseguirmos compor diversos indicadores e plotar seus gráficos, o Argentum apenas consegue mostrar o gráfico da Média Móvel Simples do Fechamento. No entanto, nós já preparamos diversos outros indicadores e poderíamos plotar todos eles naquele mesmo gráfico. O resultado disso seria algo assim:



Há informação demais nesse gráfico e isso o torna menos útil. Seria muito mais interessante se o usuário pudesse escolher quais indicadores ele quer ver e mandar plotar apenas estes no gráfico. Uma das formas possíveis para isso seria colocarmos as opções para que ele decida e um botão que disparará a geração do

gráfico. Nossa tela ficará parecida com:

Parece bom? Felizmente, colocar tais botões na tela é uma tarefa bastante simples! Como diversas aplicações têm a necessidade desses botões, esses componentes já existem e usá-los será bem semelhante ao que já vínhamos fazendo desde nossos primeiros exemplos com JSF.

Tais componentes são, inclusive, tão comuns que existem tanto na implementação do JSF quanto no Primefaces e nas outras bibliotecas semelhantes. Esse é um dilema comum quando trabalhamos com JSF: usar a implementação padrão ou a do Primefaces.

Essa escolha usualmente cai para a utilização da biblioteca, já que seus componentes já vêm com um estilo uniforme e, como já vimos antes, frequentemente adicionam funcionalidades interessantes aos componentes.

Sabendo disso, daremos preferência para os componentes do Primefaces, embora isso traga alguns efeitos colaterais. Fique atento às particularidades em destaque no decorrer desse capítulo.

Componentes para escolher o gráfico

Para que o usuário escolha qual gráfico ele quer plotar, será necessário que ele consiga interagir com a tela, isto é, precisamos de um componente de *input* de dados. Há diversos deles disponíveis no Primefaces: http://www.primefaces.org/showcase/index.xhtml

Como temos opções pré-definidas para as médias e para os indicadores básicos, usaremos um componente de `select`. Você pode descobrí-los até de dentro do Eclipse, no `index.xhtml`: basta abrir a tag `<p:select` e usar o **ctrl + espaço**!

Para replicar a tela do screenshot acima, usaremos o componente com cara de botão que permite escolher apenas uma opção, o `p:selectOneButton`. E, como qualquer `select`, precisamos também indicar quais são os itens que servirão de

opção.

```
<h:outputLabel value="Media Móvel: "/>
<p:selectOneButton value="#{argentumBean.nomeMedia}">
  <f:selectItem itemLabel="Simples" itemValue="MediaMovelSimples" />
  <f:selectItem itemLabel="Ponderada" itemValue="MediaMovelPonderada" />
</p:selectOneButton>
```

Marcamos, através do `value`, que esse componente está ligado ao atributo `nomeMedia` da classe `ArgentumBean`. Quando o usuário escolher o botão desejado, o `itemValue` será atribuído a esse atributo.

Semelhantemente, queremos outra listagem de indicadores possíveis, mas agora com nossos indicadores básicos:

```
<h:outputLabel value="Indicador base: "/>
<p:selectOneButton value="#{argentumBean.indicadorBase}">
  <f:selectItem itemLabel="Abertura" itemValue="IndicadorAbertura" />
  <f:selectItem itemLabel="Fechamento" itemValue="IndicadorFechamento" />
</p:selectOneButton>
```

Finalmente, também precisamos que o usuário indique que terminou de escolher e mande efetivamente gerar tal gráfico. E a forma mais natural de fazer isso é através de um botão com uma determinada ação -- em outras palavras, a `action` desse componente indica o método que será chamado quando o usuário apertar o botão:

```
<p:commandButton value="Gerar gráfico"
        action="#{argentumBean.geraGrafico}"/>
```

Note que esse botão do Primefaces é muito mais atraente do que o botão padrão que usamos no `olaMundo.xhtml`. A folha de estilos do Primefaces é realmente superior, mas não é só isso que muda!

Ponto de atenção: Primefaces e o AJAX

Os botões do Primefaces não são apenas bonitos. Eles também trabalham por padrão com chamadas via AJAX, em vez de recarregar a tela toda. Isso é interessante quando temos telas complexas, com diversas informações e o apertar de um botão altera somente um pedaço dela.

Contudo, exatamente porque nossa chamada será executada via AJAX, não haverá navegação e os outros componentes da tela não serão recarregados, a menos que explicitamente indiquemos que tal botão deve recarregar um componente.

No nosso caso, queremos que apertar botão *Gerar gráfico* regere o modelo do gráfico e recarregue tal componente na tela, então ainda é necessário alterar o `p:commandButton` para que ele atualize o gráfico. Para isso, precisamos indicar ao botão que ele será também responsável por atualizar o componente do gráfico. Para ligar um componente ao outro, usaremos um `id`.

```
<p:commandButton value="Gerar gráfico" update=":grafico"
        action="#{argentumBean.geraGrafico}"/>
```

Note o ":" (dois pontos). Isso indica para o botão que ele partirá da tag raiz da página procurando algum componente com id `grafico`. E, para que isso funcione, também é necessário que o componente do gráfico tenha tal id.

```
<p:chart type="line" model="#{argentumBean.modeloGrafico}" />
```

Lembre-se também que sempre que temos botões, precisamos que eles estejam dentro de um componente `h:form`. Fora deste componente, os botões não funcionam. Outro detalhe aqui é que, por padrão, o Primefaces coloca cada *label*, *select* e *botão* em uma linha. Esse comportamento é ótimo para formulários mais padrão, mas se quisermos mostrar tudo em uma linha só ainda podemos utilizar um panelGrid com 5 colunas para acomodar os 5 componentes desse menu.

```
<h:form>
  <h:panelGrid columns="5">
    <h:outputLabel value="Media Móvel: "/>
    <p:selectOneButton value="#{argentumBean.nomeMedia}">
      <f:selectItem itemLabel="Simples" itemValue="MediaMovelSimples" />
      <f:selectItem itemLabel="Ponderada" itemValue="MediaMovelPonderada" />
    </p:selectOneButton>

    <h:outputLabel value="Indicador base: "/>
    <p:selectOneButton value="#{argentumBean.nomeIndicadorBase}">
      <f:selectItem itemLabel="Abertura" itemValue="IndicadorAbertura" />
      <f:selectItem itemLabel="Fechamento" itemValue="IndicadorFechamento"/>
    </p:selectOneButton>

    <p:commandButton value="Gerar gráfico" update=":grafico"
            action="#{argentumBean.geraGrafico}"/>
  </h:panelGrid>
</h:form>
```

E o ManagedBean?

Semelhantemente à proposta do JSF, focamos apenas nos novos componentes e todas as alterações propostas nesse capítulo estão limitadas a alterações no `index.xhtml`.

Se repararmos bem, contudo, os novos componentes exigirão mudanças consideráveis ao `ArgentumBean`:

- Atributo `nomeMedia`, seu getter e setter;
- Atributo `nomeIndicadorBase`, seu getter e setter;
- Método `geraGrafico`, que será o novo responsável pelo modelo do gráfico.

Faremos tais alterações no decorrer do próximo exercício.

11.2 - Exercícios: permitindo que o usuário escolha o gráfico

1. No `index.html` e logo após a tag `<h:body>`, adicione o formulário que permitirá que o usuário escolha qual indicador ele gostaria de ver plotado e também o `panelGrid` que fará com que seu formulário ocupe apenas uma linha.

```
<h:form>
  <h:panelGrid columns="5">

  </h:panelGrid>
</h:form>
```

2. Agora, **dentro do `h:panelGrid`**, precisamos colocar nossos selects das médias e dos indicadores básicos, juntamente com as respectivas legendas dos campos:

```
<h:outputLabel value="Media Móvel: "/>
<p:selectOneButton value="#{argentumBean.nomeMedia}">
  <f:selectItem itemLabel="Simples" itemValue="MediaMovelSimples" />
  <f:selectItem itemLabel="Ponderada" itemValue="MediaMovelPonderada" />
</p:selectOneButton>

<h:outputLabel value="Indicador base: "/>
<p:selectOneButton value="#{argentumBean.nomeIndicadorBase}">
  <f:selectItem itemLabel="Abertura" itemValue="IndicadorAbertura" />
  <f:selectItem itemLabel="Fechamento" itemValue="IndicadorFechamento"/>
</p:selectOneButton>
```

3. Suba o TomCat agora e verifique que recebemos uma ServletException. Ela nos informa que ainda não temos a propriedade `nomeMedia` no `ArgentumBean`. Precisamos criar os atributos necessários para que esses componentes funcionem corretamente.

Na classe `ArgentumBean` crie os atributos `nomeMedia` e `nomeIndicadorBase` e seus respectivos *getters* e *setters*. Lembre-se de usar o **ctrl + 3** *getter* para isso!

```
@ManagedBean
@ViewScoped
```

```
public class ArgentumBean implements Serializable {

  private List<Negociacao> negociacoes;
  private ChartModel modeloGrafico;
  private String nomeMedia;
  private String nomeIndicadorBase;

  // agora crie os getters e setters com o ctrl+3
```

Atenção: se seu `ArgentumBean` não estiver anotado com `@ViewScoped`, anote-o. A explicação dos escopos está no exercício opcional de paginação e ordenação, no capítulo de Introdução ao JSF.

4. Reinicie o TomCat e veja os selects na tela. Falta pouco! De volta ao `index.xhtml` coloque o botão no mesmo formulário, logo após os selects. Coloque também o `id=grafico` no componente `p:chart` que já existe, para fazer com que o clique no botão atualize também esse componente.

```
<p:commandButton value="Gerar gráfico" update=":grafico"
            action="#{argentumBean.geraGrafico}"/>
  </h:panelGrid>
</h:form>

<p:chart id="grafico" type="line" model="#{argentumBean.modeloGrafico}" />
```

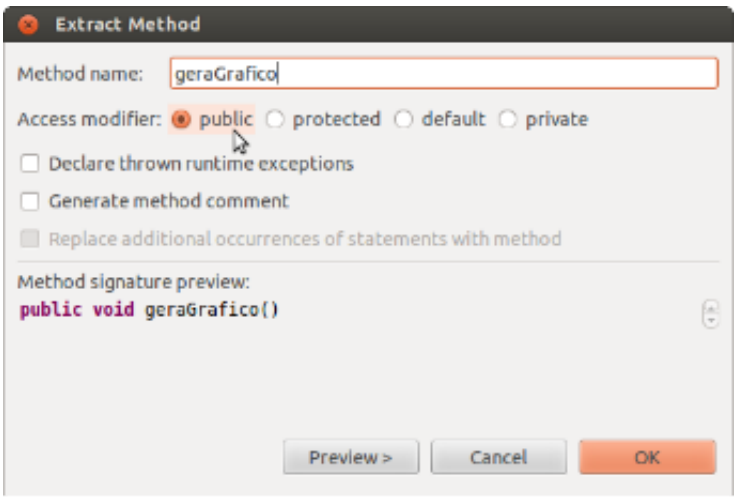
5. Já é possível ver a tela, mas ao clicar no botão de gerar o gráfico, nada acontece. Na verdade, no Console do Eclipse é possível ver que uma exceção foi lançada porque o método que o botão chama, o `geraGrafico`, ainda não existe.

Esse método terá que, a partir da lista de negociações, criar as Candles, a Série Temporal, mandar plotar um indicador ao gráfico e disponibilizar o modelo atualizado para o gráfico. Note, no entanto, que nosso construtor já faz todo esse trabalho!

```
public ArgentumBean() {
    this.negociacoes = new ClienteWebService().getNegociacoes();
    List<Candle> candles = new CandleFactory().constroiCandles(negociacoes);
    SerieTemporal serie = new SerieTemporal(candles);

    GeradorModeloGrafico geradorGrafico =
                new GeradorModeloGrafico(serie, 2, serie.getUltimaPosicao());
    geradorGrafico.plotaIndicador(new MediaMovelSimples(new IndicadorFechamento()));
    this.modeloGrafico = geradorGrafico.getModeloGrafico();
}
```

Basta fazermos uma simples refatoração *Extract method* nessas linhas! Selecione tais linhas do construtor da `ArgentumBean` e use **ctrl + 3** *extract method* indicando que o novo método deve se chamar `geraGrafico` e deve ser público.



6. Finalmente, para verificarmos que estamos recebendo informações corretas sobre qual indicador plotar no momento que o método `geraGrafico` é chamado, vamos imprimir essas informações no console com um simples `syso`.

Altere o recém-criado `geraGrafico`, adicionando o `syso`:

```
public void geraGrafico() {
  System.out.println("PLOTANDO: " + nomeMedia + " de " + nomeIndicadorBase);
  List<Candle> candles = new CandleFactory().constroiCandles(negociacoes);
  //...
```

7. Reinicie o TomCat e volte a acessar a URL do projeto:
http://localhost:8080/fj22-argentum-web/index.xhtml

Teste os botões e mande gerar o gráfico. **Atenção! O gráfico ainda não será gerado corretamente!** Falta implementarmos essa parte no *bean*. Olhando no Console do Eclipse, contudo, note que nosso `syso` já mostra que os atributos que escolherão qual indicador plotar já foi escolhido corretamente!

Faça o teste para algumas combinações de média e indicador, observando o console.

11.3 - Montando os indicadores dinamicamente

No nosso formulário criamos dois *select buttons*, um para a média e outro para o indicador base. Nós até já verificamos que essas informações estão sendo preenchidas corretamente quando o usuário clica no botão.

No entanto, ainda não estamos usando essa informação para plotar o indicador escolhido pelo usuário - ainda estamos plotando a média móvel simples do fechamento em todos os casos. Essa informação está *hard-coded* no `geraGrafico`

```
public void geraGrafico() {
  System.out.println("PLOTANDO: " + nomeMedia + " de " + nomeIndicadorBase);
  List<Candle> candles = new CandleFactory().constroiCandles(negociacoes);
  SerieTemporal serie = new SerieTemporal(candles);

  GeradorModeloGrafico geradorGrafico =
        new GeradorModeloGrafico(serie, 2, serie.getUltimaPosicao());
  geradorGrafico.plotaIndicador(
              new MediaMovelSimples(new IndicadorFechamento()));
  this.modeloGrafico = geradorGrafico.getModeloGrafico();
}
```

Note que as informações de qual indicador o usuário quer plotar já estão preenchidas nos atributos `nomeMedia` e `nomeIndicadorBase`, mas esses atributos são meramente Strings passadas pelo componente de select.

Poderíamos, então, usar um conjunto de `if/else` para decidir qual indicador usar:

```
private Indicador defineIndicador() {
  if ("MediaMovelSimples".equals(nomeMedia)) {
    if ("IndicadorAbertura".equals(nomeIndicadorBase))
      return new MediaMovelSimples(new IndicadorAbertura());
    if ("IndicadorFechamento".equals(nomeIndicadorBase))
      return new MediaMovelSimples(new IndicadorFechamento());
  } else if ("MediaMovelPonderada".equals(nomeMedia)) {
    // varios outros ifs
```

É fácil ver que essa é uma solução extremamente deselegante. Nesse pequeno trecho escrevemos quatro `if`s e isso é um sinal de mal design. Pense agora no teste para esse trecho de código: esse método sozinho tem quatro caminhos possíveis de se percorrer. Esse número também é chamado de **complexidade ciclomática**.

Quatro não é um número ruim, mas note que, ao adicionar um novo indicador básico ou uma nova média, esse número aumentará de forma multiplicativa. O número de caminhos será sempre igual à `quantidadeDeMedias` *

`quantidadeDeIndicadoresBase`.

> **Complexidade Ciclomática**
>
> Essa métrica expõe o número de caminhos diferentes que uma execução pode percorrer em um método. O problema de uma complexidade ciclomática alta é que: quanto maior esse número for, mais testes serão necessários para garantir o funcionamento esperado.
>
> Para maiores detalhes sobre essa métrica consulte no blog da caelum esse artigo: http://blog.caelum.com.br/medindo-a-complexidade-do-seu-codigo/

Cada vez mais indicadores

Por vantagem competitiva, nosso cliente pode pedir mais indicadores (mínimo e máximo) e outras médias (exponencial e adaptativa). É sensível que o código nunca parará de crescer e estará sempre sujeito a alteração.

Embora seja aceitável para um número bem pequeno de indicadores, conforme nosso sistema evolui é fundamental que encontremos um jeito melhor de conseguir um objeto de Indicador a partir das Strings que já temos.

Nesse caso a API de reflection cai com uma luva. Com ela podemos escrever o código que cria objetos ou chamar métodos sem conhecer as classes antecipadamente, justamente o que precisamos para montar as indicadores.

11.4 - Introdução a Reflection

Por ser uma linguagem compilada, Java permite que, enquanto escrevemos nosso código, tenhamos total controle sobre o que será executado, de tudo que faz parte do nosso sistema. Em tempo de desenvolvimento, olhando nosso código, sabemos quantos atributos uma classe tem, quais construtores e métodos ela possui, qual chamada de método está sendo feita e assim por diante.

Mas existem algumas raras situações onde essa garantia do compilador não nos ajuda. Isto é, existem situações em que precisamos de características dinâmicas. Por exemplo, imagine permitir que, dado um nome de método que o usuário passar, nós o invocaremos em um objeto. Ou ainda, que, ao recebermos o nome de

uma classe enquanto executando o programa, possamos criar um objeto dela!

Nas próximas seções, você vai conhecer um pouco desse recurso avançado e muito poderoso que, embora possua diversas qualidades, aumenta bastante a complexidade do nosso código e por essa razão deve ser usado de forma sensata.

11.5 - Por que reflection?

Um aluno que já cursou o FJ-21 pode notar uma incrível semelhança com a discussão em aula sobre MVC, onde, dado um parâmetro da requisição, criamos um objeto das classes de lógica. Outro exemplo, já visto, é o do XStream: dado um XML, ele consegue criar objetos para nós e colocar os dados nos atributos dele. Como ele faz isso? Será que no código-fonte do XStream acharíamos algo assim:

```
Negociacao n = new Negociacao(...);
```

O XStream foi construído para funcionar com qualquer tipo de XML e objeto. Com um pouco de ponderação fica óbvio que **não há** um `new` para cada objeto possível e imaginável dentro do código do XStream. Mas como ele consegue instanciar um objeto da minha classe e popular os atributos, tudo sem precisar ter `new Negociacao()` escrito dentro dele?

O **java.lang.reflect** é um pacote do Java que permite criar instâncias e chamadas em tempo de execução, sem precisar conhecer as classes e objetos envolvidos no momento em que escrevemos nosso código (tempo de compilação). Esse dinamismo é necessário para resolvermos determinadas tarefas que só descobrimos serem necessárias ao receber dados, isto é, em tempo de execução.

De volta ao exemplo do XStream, ele só descobre o nome da nossa classe `Negociacao` quando rodamos o programa e selecionamos o XML a ser lido. Enquanto escreviam essa biblioteca, os desenvolvedores do XStream não tinham a menor ideia de que um dia o usaríamos com a classe `Negociacao`.

Apenas para citar algumas possibilidades com reflection:

- Listar todos os atributos de uma classe e pegar seus valores em um objeto;
- Instanciar classes cujo nome só vamos conhecer em tempo de execução;
- Invocar um construtor especifico baseado no tipo de atributo
- Invocar métodos dinamicamente baseado no nome do método como String;

- Descobrir se determinados pedaços do código têm annotations.

11.6 - Constructor, Field e Method

O ponto de partida da reflection é a classe `Class`. Esta, é uma classe da própria API do Java que representa cada modelo presente no sistema: nossas classes, as que estão em JARs e também as do próprio Java. Através da `Class` conseguimos obter informações sobre qualquer classe do sistema, como seus atributos, métodos, construtores, etc.

Todo objeto tem um jeito fácil pegar o `Class` dele:

```
Negociacao n = new Negociacao();

// chamamos o getClass de Object
Class<Negociacao> classe = n.getClass();
```

Mas nem mesmo precisamos de um objeto para conseguir as informações da sua classe. Diretamente com o nome da classe também podemos fazer o seguinte:

```
Class<Negociacao> classe = Negociacao.class;
```

Ou ainda, se tivermos o caminho completo da classe, conseguimos recuperar tal classe até através de uma String:

```
Class classe = Class.forName("br.com.caelum.argentum.model.Negociacao");
```

Java 5 e Generics

> A partir do Java 5, a classe `Class` é tipada e recebe o tipo da classe que estamos trabalhando. Isso melhora alguns métodos, que antes recebiam `Object` e agora trabalham com um tipo `T` qualquer, parametrizado pela classe.

A partir de um `Class` podemos listar, por exemplo, os nomes e valores dos seus atributos:

```
Class<Negociacao> classe = Negociacao.class;
for (Field atributo : classe.getDeclaredFields()) {
  System.out.println(atributo.getName());
}
```

A saída será:

```
preco
quantidade
data
```

Fazendo similarmente para os métodos é possível conseguir a lista:

```
Class<Negociacao> classe = Negociacao.class;
for (Method metodo : classe.getDeclaredMethods()) {
  System.out.println(metodo.getName());
}
```

Cuja saída será:

```
getPreco
getQuantidade
getData
getVolume
isMesmoDia
```

Assim como podemos descobrir métodos e atributos, o mesmo aplica para construtores. Para, por exemplo, descobrir o construtor da classe `MediaMovelSimples`:

```
Class<MediaMovelSimples> classe = MediaMovelSimples.class;
Constructor<?>[] construtores = classe.getConstructors();

for (Constructor<?> constructor : construtores) {
  System.out.println(Arrays.toString(constructor.getParameterTypes()));
}
```

Imprime o tipo do parâmetro do único construtor:

```
interface br.com.caelum.argentum.indicadores.Indicador
```

É possível fazer muito mais. Investigue a API de reflection usando **ctrl + espaço** a partir das classes Method, Constructor e Fields no Eclipse e pelo JavaDoc.

11.7 – Melhorando nosso ArgentumBean

Já temos os conhecimentos necessários para melhorar nosso método `defineIndicador()`, que será chamado pelo `geraGrafico`. A primeira necessidade é instanciar um Indicador a partir da String recebida. Mas antes é preciso descobrir a classe pela String e através do método `newInstance()` instanciar um objeto da classe.

Isso se torna bastante fácil se convencionarmos que todo indicador estará no pacote `br.com.caelum.argentum.indicadores`. Assim, instanciar o indicador base é muito simples.

```
String pacote = "br.com.caelum.argentum.indicadores.";
Class<?> classeIndicadorBase = Class.forName(pacote + nomeIndicadorBase);
Indicador indicadorBase = (Indicador) classeIndicadorBase.newInstance();
```

Para a média o trabalho é um pouco maior, já que precisamos passar um Indicador como parâmetro na instanciação do objeto. Isso pode ser até mais complexo se houver outros indicadores na classe em questão. Nesse caso, não basta fazer a chamada ao método `newInstance()`, já que este só consegue instanciar objetos quando o construtor padrão existe.

Por outro lado, já sabemos como descobrir o construtor a partir de um `Class`. A boa notícia é que existe como criarmos uma instância a partir de um `Constructor`, pelo mesmo método `newInstance()` que passa a receber parâmetros.

Veja o código:

```
Class<?> classeMedia = Class.forName(pacote + nomeMedia);
Constructor<?> construtorMedia = classeMedia.getConstructor(Indicador.class);
Indicador indicador = (Indicador) construtorMedia.newInstance(indicadorBase);
```

Repare que esse código funciona com qualquer indicador ou média que siga a convenção de passar o nome da classe no `itemValue` do *select* e a convenção de pacote.

Se errarmos alguma dessas informações, no entanto, diversos erros podem acontecer e é por isso que o código acima pode lançar diversas exceções: `InstantiationException, IllegalAccessException, ClassNotFoundException, NoSuchMethodException, InvocationTargetException`.

Como essas exceções são todas *checked*, seria necessário declará-las todas com um `throws` na assinatura do método `defineIndicador`, o que pode não ser desejável. Podemos, então, apelar para um `try/catch` que encapsule tais exceções em uma *unchecked* para que ela continue sendo lançada, mas não suje a assinatura do método com as tantas exceções. Essa é uma prática um tanto comum, hoje em dia, e linguagens mais novas chegam a nem ter mais as *checked exceptions*.

Nosso método, portanto, ficará assim:

```
private Indicador defineIndicador() {
  try {
    String pacote = "br.com.caelum.argentum.indicadores.";
    // instancia indicador base
    // instancia a média, passando o indicador base
    return indicador;
  } catch (Exception e) {
    throw new RuntimeException(e);
  }
}
```

11.8 – Exercícios: indicadores em tempo de execução

1. Na classe `ArgentumBean`, ache o método `geraGrafico` e identifique a linha que plota o indicador. Ela deve ser algo como:

```
geradorGrafico.plotaIndicador(new MediaMovelSimples(new
IndicadorFechamento()));
```

Troque o parâmetro do `plotaIndicador` por uma chamada a um novo método `defineIndicador()`:

```
geradorGrafico.plotaIndicador(defineIndicador());
```

2. Use o **ctrl + 1** e escolha a opção *Create method defineIndicador*. O Eclipse gerará o esqueleto do método para nós e, agora, falta preenchê-lo.

```
private Indicador defineIndicador() {
  String pacote = "br.com.caelum.argentum.indicadores.";
  Class<?> classeIndicadorBase = Class.forName(pacote + nomeIndicadorBase);
  Indicador indicadorBase = (Indicador) classeIndicadorBase.newInstance();

  Class<?> classeMedia = Class.forName(pacote + nomeMedia);
  Constructor<?> construtorMedia =
classeMedia.getConstructor(Indicador.class);
  Indicador indicador = (Indicador)
construtorMedia.newInstance(indicadorBase);
  return indicador;
}
```

3. **O código acima ainda não compila!** As diversas exceções que a API de reflection lança ainda precisam ser tratadas. Com a ajuda do Eclipse, envolva todo esse trecho de código com um `try/catch` que pega qualquer `Exception` e a encapsula em uma `RuntimeException`.

```
private Indicador defineIndicador() {
  try {
    String pacote = "br.com.caelum.argentum.indicadores.";
    Class<?> classeIndicadorBase = Class.forName(pacote + nomeIndicadorBase);
    Indicador indicadorBase = (Indicador) classeIndicadorBase.newInstance();

    Class<?> classeMedia = Class.forName(pacote + nomeMedia);
    Constructor<?> construtorMedia =
classeMedia.getConstructor(Indicador.class);
    Indicador indicador = (Indicador)
construtorMedia.newInstance(indicadorBase);
    return indicador;
  } catch (Exception e) {
    throw new RuntimeException(e);
  }
}
```

4. Vamos aproveitar e adicionar um comportamento padrão para quando o usuário não tiver escolhido qual gráfico ele quer ainda. Antes mesmo do bloco de `try/catch`, podemos adicionar o comportamento padrão de, se os indicadores não estiverem escolhidos, devolvemos por padrão a Média Móvel Simples do Fechamento.

```
private Indicador defineIndicador() {
  if (nomeIndicadorBase == null || nomeMedia == null)
    return new MediaMovelSimples(new IndicadorFechamento());
  try {
    //...
```

5. Reinicie o TomCat e acesse o Argentum para testar o funcionamento dessa nova modificação no sistema. Tudo deve estar funcionando!

11.9 - Melhorando a orientação a objetos

O design do código do exercício anterior tem muito espaço para melhorias. Note que acabamos colocando muitas responsabilidades diferentes na classe `ArgentumBean` - justamente nela, que deveria servir apenas para prover objetos para os componentes da nossa página.

Além disso, nosso método `defineIndicador` é um método privado na `ArgentumBean` e isso o torna extremamente difícil de testar.

Podemos, por exemplo, extrair essa lógica de definição do indicador para uma classe completamente nova, que tivesse essa única responsabilidade. Assim, nosso método `geraGrafico` instanciaria um `IndicadorFactory`, que devolveria o indicador correto:

```
public void geraGrafico() {
  // outras linhas de código

  IndicadorFactory fabrica = new IndicadorFactory(nomeMedia,
nomeIndicadorBase);
  geradorGrafico.plotaIndicador(fabrica.defineIndicador());
  this.modeloGrafico = geradorGrafico.getModeloGrafico();
}
```

Assim, o método `defineIndicador()` passará a ser público e, portanto, muito mais facilmente testável e muito mais coeso - seguindo um dos princípios **SOLID**: mais especificamente, o **princípio da responsabilidade única**

Exercícios Opcionais: refatoração para melhorar a coesão

1. (Opcional) Crie a classe `IndicadorFactory` e faça com que ela receba no construtor os parâmetros `nomeMedia` e `nomeIndicadorBase`. Esses valores são obrigatórios e não serão alterados, portanto eles podem ser `final`.

2. (Opcional) Traga para essa classe o método `defineIndicador`, alterando ele para público, agora que ele está em outra classe. Lembre-se, também, de arrumar erros de compilação que surgirem na classe `ArgentumBean`.

3. (Opcional) Crie testes para a `IndicadorFactory`. Um desses testes deve verificar o comportamento quando não passamos o nome de uma das classes. Outro, poderia motivar a melhoria do tratamento das exceções para algo mais descritivo.

CAPÍTULO ANTERIOR:

Aplicando Padrões de projeto

PRÓXIMO CAPÍTULO:

Apêndice Testes de interface com Selenium

Twitter