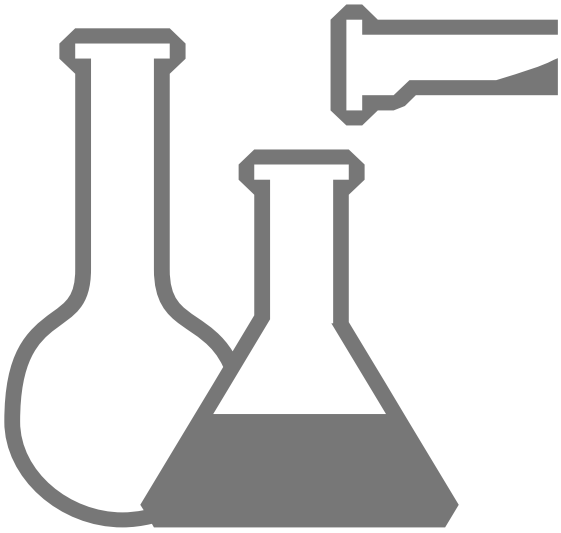
Laboratório Java com Testes, JSF e Design Patterns

Curso FJ-22



caelum

ensino e inovação

Caelum
Ensino e Inovação

Conheça mais da Caelum.

Cursos Online

www.caelum.com.br/online

Casa do Código

Livros para o programador

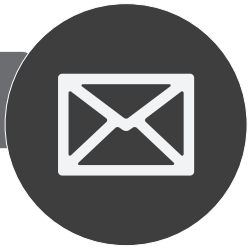
www.casadocodigo.com.br

Blog Caelum

blog.caelum.com.br

Newsletter

www.caelum.com.br/newsletter



Facebook

www.facebook.com/caelumbr

Twitter

twitter.com/caelum

Sobre esta apostila

Esta apostila da Caelum visa ensinar de uma maneira elegante, mostrando apenas o que é necessário e quando é necessário, no momento certo, poupando o leitor de assuntos que não costumam ser de seu interesse em determinadas fases do aprendizado.

A Caelum espera que você aproveite esse material. Todos os comentários, críticas e sugestões serão muito bem-vindos.

Essa apostila é constantemente atualizada e disponibilizada no site da Caelum. Sempre consulte o site para novas versões e, ao invés de anexar o PDF para enviar a um amigo, indique o site para que ele possa sempre baixar as últimas versões. Você pode conferir o código de versão da apostila logo no final do índice.

Baixe sempre a versão mais nova em: www.caelum.com.br/apostilas

www.caelum.com.br

Sumário

Versão: 18.3.13

Capítulo 1

Tornando-se um desenvolvedor pragmático

"Na maioria dos casos, as pessoas, inclusive os facínoras, são muito mais ingênuas e simples do que costumamos achar. Aliás, nós também."
– Fiodór Dostoiévski, em Irmãos Karamazov

Por que fazer esse curso?

1.1 O que é realmente importante?

Você já passou pelo FJ-11 e, quem sabe, até pelo FJ-21. Agora chegou a hora de codificar bastante para pegar os truques e hábitos que são os grandes diferenciais do programador Java experiente.

Pragmático é aquele que se preocupa com as questões práticas, menos focado em ideologias e tentando colocar a teoria pra andar.

Esse curso tem como objetivo trazer uma visão mais prática do desenvolvimento Java através de uma experiência rica em código, onde exercitaremos diversas APIS e recursos do Java. Vale salientar que as bibliotecas em si não são os pontos mais importantes do aprendizado neste momento, mas sim as boas práticas, a cultura e um olhar mais amplo sobre o design da sua aplicação.

Os *design patterns*, as boas práticas, a refatoração, a preocupação com o baixo acoplamento, os testes de unidade (também conhecidos como testes unitários) e as técnicas de programação (idiomismos) são passados com afinco.

Para atingir tal objetivo, esse curso baseia-se fortemente em artigos, blogs e, em especial, na literatura que se consagrou como fundamental para os desenvolvedores Java. Aqui citamos alguns desses livros:

http://blog.caelum.com.br/2006/09/22/livros-escolhendo-a-trindade-do-desenvolvedor-java/

Somamos a esses mais dois livros, que serão citados no decorrer do curso, e influenciaram muito na elaboração do conteúdo que queremos transmitir a vocês. Todos os cinco são:

- **Effective Java, Joshua Bloch**

  Livro de um dos principais autores das maiores bibliotecas do Java SE (como o java.io e o java.util), arquiteto chefe Java na Google atualmente. Aqui ele mostra como enfrentar os principais problemas e limitações da linguagem. Uma excelente leitura, dividido em mais de 70 tópicos de 2 a 4 páginas cada, em média. Entre os casos interessantes está o uso de *factory methods*, os problemas da herança e do `protected`, uso de coleções, objetos imutáveis e serialização, muitos desses abordados e citados aqui no curso.

- **Design Patterns, Erich Gamma et al**

  Livro de Erich Gamma, por muito tempo líder do projeto Eclipse na IBM, e mais outros três autores, o que justifica terem o apelido de *Gang of Four* (GoF). Uma excelente leitura, mas cuidado: não saia lendo o catálogo dos patterns decorando-os, mas concentre-se especialmente em ler toda a primeira parte, onde eles revelam um dos princípios fundamentais da programação orientada a objetos:

  "*Evite herança, prefira composição*" e "*Programe voltado às interfaces e não à implementação*".

- **Refactoring, Martin Fowler**

  Livro do cientista chefe da ThoughtWorks. Um excelente catálogo de como consertar pequenas falhas do seu código de maneira sensata. Exemplos clássicos são o uso de herança apenas por preguiça, uso do `switch` em vez de polimorfismo, entre dezenas de outros. Durante o curso, faremos diversos refactoring clássicos utilizando do Eclipse, muito mais que o básico *rename*.

- **Pragmatic Programmer, Andrew Hunt**

  As melhores práticas para ser um bom desenvolvedor: desde o uso de versionamento, ao bom uso do logging, debug, nomenclaturas, como consertar bugs, etc.

- **The mythical man-month, Frederick Brooks**

  Um livro que fala dos problemas que encontramos no dia a dia do desenvolvimento de software, numa abordagem mais gerencial. Aqui há, inclusive, o clássico artigo "No Silver Bullet", que afirma que nunca haverá uma solução única (uma linguagem, um método de desenvolvimento, um sistema operacional) que se adeque sempre a todos os tipos de problema.

1.2 A importância dos exercícios

É um tanto desnecessário debater sobre a importância de fazer exercícios, porém neste curso específico eles são vitais: como ele é focado em boas práticas, alguma parte da teoria não está no texto - e é passado no decorrer de exercícios.

Não se assuste, há muito código aqui nesse curso, onde vamos construir uma pequena aplicação que lê um XML com dados da bolsa de valores e plota o gráfico de *candlesticks*, utilizando diversas APIs do Java SE e até mesmo bibliotecas externas.

1.3 Tirando dúvidas e referências

Para tirar dúvidas dos exercícios, ou de Java em geral, recomendamos o fórum do site do GUJ (http://www.guj.com.br/) , onde sua dúvida será respondida prontamente.

Fora isso, sinta-se à vontade para entrar em contato com seu instrutor e tirar todas as dúvidas que tiver durante o curso.

Você pode estar interessado no livro TDD no mundo real, da editora Casa do Código:

http://www.tddnomundoreal.com.br/

1.4 Para onde ir depois?

Se você se interessou pelos testes, design e automação, recomendamos os cursos online de testes da Caelum:

http://www.caelum.com.br/curso/online/testes-automatizados/

O FJ-21 é indicado para ser feito antes ou depois deste curso, dependendo das suas necessidades e do seu conhecimento. Ele é o curso que apresenta o desenvolvimento Web com Java e seus principais ferramentas e frameworks.

Depois destes cursos, que constituem a Formação Java da Caelum, indicamos dois outros cursos, da Formação Avançada:

http://www.caelum.com.br/formacao-java-avancada/

O FJ-25 aborda Hibernate e JPA 2 e o FJ-26 envolve JSF 2, Facelets e CDI. Ambos vão passar por tecnologias hoje bastante utilizadas no desenvolvimento server side para web, e já na versão do Java EE 6.

Capítulo 2

O modelo da bolsa de valores, datas e objetos imutáveis

"Primeiro aprenda ciência da computação e toda a teoria. Depois desenvolva um estilo de programação. E aí esqueça tudo e apenas 'hackeie'."
– George Carrette

O objetivo do FJ-22 é aprender boas práticas da orientação a objetos, do design de classes, uso correto dos design patterns, princípios de práticas ágeis de programação e a importância dos testes de unidade.

Dois livros que são seminais na área serão referenciados por diversas vezes pelo instrutor e pelo material: *Effective Java*, do Joshua Bloch, e *Design Patterns: Elements of Reusable Object-Oriented Software*, de Erich Gamma e outros (conhecido Gang of Four).

2.1 A bolsa de valores

Poucas atividades humanas exercem tanto fascínio quanto o mercado de ações, assunto abordado exaustivamente em filmes, livros e em toda a cultura contemporânea. Somente em novembro de 2007, o total movimentado pela BOVESPA foi de R$ 128,7 bilhões. Destes, o volume movimentado por aplicações home *broker* foi de R$ 22,2 bilhões.

Neste curso, abordaremos esse assunto que, hoje em dia, chega a ser cotidiano desenvolvendo uma aplicação que interpreta os dados de um XML, trata e modela eles em Java e mostra gráficos pertinentes.

2.2 Candlesticks: O Japão e o arroz

Yodoya Keian era um mercador japonês do século 17. Ele se tornou rapidamente muito rico, dadas as suas habilidades de transporte e precificação do arroz, uma mercadoria em crescente produção em consumo no país. Sua situação social de mercador não permitia que ele fosse tão rico dado o sistema de castas da época e, logo, o governo confiscou todo seu dinheiro e suas posses. Depois dele, outros vieram e tentaram esconder suas origens como mercadores: muitos tiveram seus filhos executados e seu dinheiro confiscado.

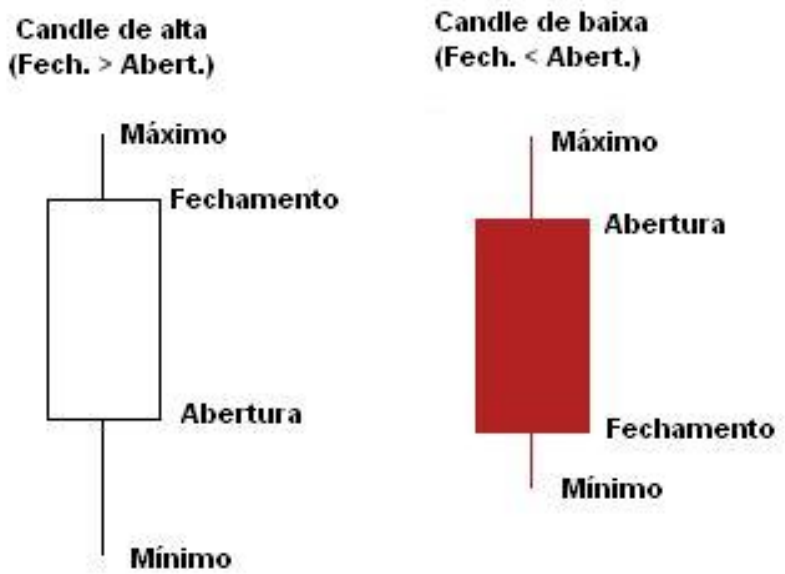
Apesar da triste história, foi em Dojima, no jardim do próprio Yodoya Keian, que nasceu a bolsa de arroz do Japão. Lá eram negociados, precificados e categorizados vários tipos de arroz. Para anotar os preços do arroz, desenhava-se figuras no papel. Essas figuras parecem muito com velas -- daí a analogia **candlestick**.

Esses desenhos eram feitos em um papel feito de... arroz! Apesar de usado a séculos, o mercado ocidental só se interessou pela técnica dos candlesticks recentemente, no último quarto de século.

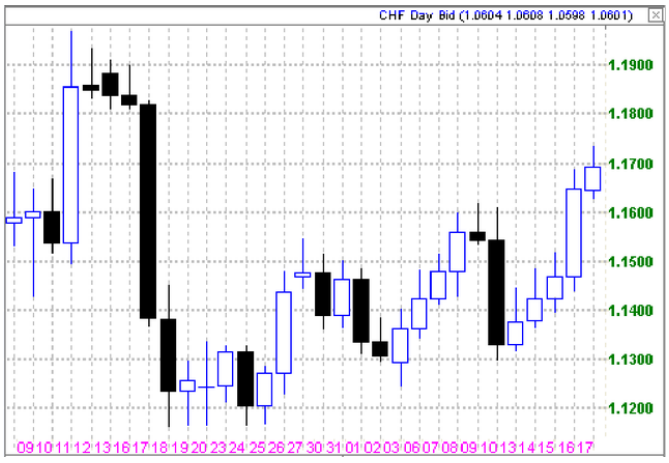
Um candlestick indica 4 valores: o maior preço do dia, o menor preço do dia (as pontas), o primeiro preço do dia e o último preço do dia (conhecidos como abertura e fechamento, respectivamente).

Os preços de abertura e fechamento são as linhas horizontais e dependem do tipo de candle: se for de alta, o preço de abertura é embaixo; se for de baixa, é em cima. Um candle de alta costuma ter cor azul ou branca e os de baixa costumam ser vermelhos ou pretos. Caso o preço não tenha se movimentado, o candle tem a mesma cor que a do dia anterior.

Para calcular as informações necessárias para a construção de um `Candlestick`, são necessários os dados de todos as **negociações** (*trades*) de um dia. Uma **Negociação** possui três informações: o **preço** pelo qual foi comprado, a **quantidade** de ativos e a **data** em que ele foi executado.



Você pode ler mais sobre a história dos candles em: http://www.candlestickforum.com/PPF/Parameters/1_279_/candlestick.asp



Apesar de falarmos que o `Candlestick` representa os principais valores de *um dia*, ele pode ser usado para os mais variados intervalos de tempo: um candlestick pode representar 15 minutos, ou uma semana, dependendo se você está analisando o ativo para curto, médio ou longo prazo.

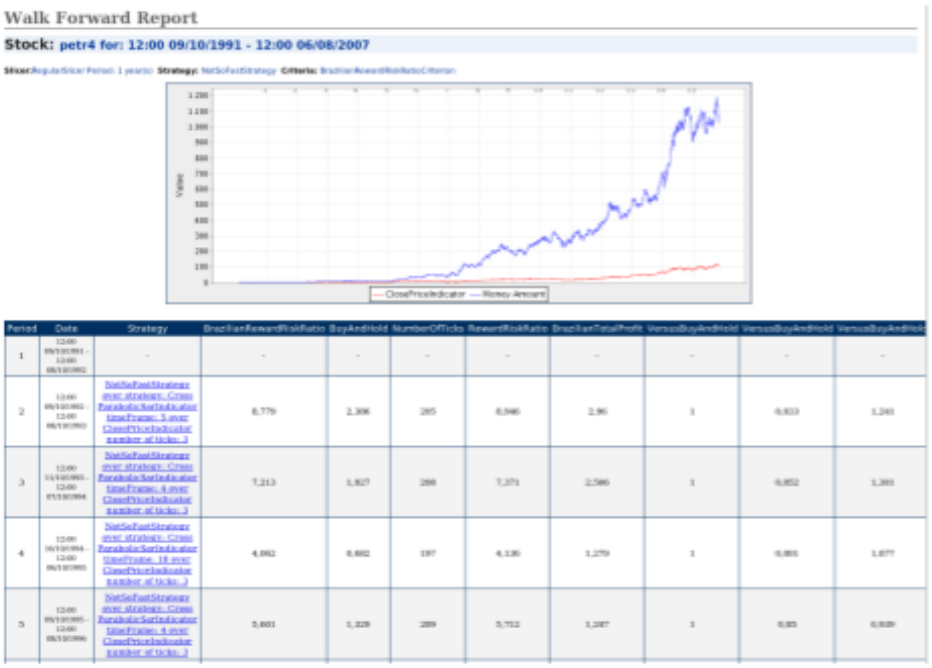
2.3 O projeto Tail

A ideia do projeto **Tail** (**T**echnical **A**nalysis **I**ndicator **L**ibrary) nasceu quando um grupo de alunos da Universidade de São Paulo procurou o professor doutor Alfredo Goldman para orientá-los no desenvolvimento de um software para o projeto de conclusão de curso.

Ele então teve a ideia de juntar ao grupo alguns alunos do mestrado através de um sistema de coorientação, onde os mestrandos auxiliariam os graduandos na implementação, modelagem e metodologia do projeto. Somente então o grupo definiu o tema: o desenvolvimento de um software *open source* de análise técnica grafista (veremos o que é a análise técnica em capítulos posteriores).

O software está disponível no SourceForge:

```
http://sourceforge.net/projects/tail/
```



Essa ideia, ainda vaga, foi gradativamente tomando a forma do projeto desenvolvido. O grupo se reunia semanalmente adaptando o projeto, atribuindo novas tarefas e objetivos. Os graduandos tiveram a oportunidade de trabalhar em conjunto com os mestrandos, que compartilharam suas experiências anteriores.

Objetivos do projeto Tail:

- Implementar os componentes básicos da análise técnica grafista: série temporal, operações de compra e venda e indicadores técnicos;
- Implementar as estratégias de compra e venda mais utilizadas no mercado, assim como permitir o rápido desenvolvimento de novas estratégias;
- Implementar um algoritmo genérico para determinar um momento apropriado de compra e venda de um ativo, através da escolha da melhor estratégia aplicada a uma série temporal;
- Permitir que o critério de escolha da melhor estratégia seja trocado e desenvolvido facilmente;
- Criar relatórios que facilitem o estudo e a compreensão dos resultados obtidos pelo algoritmo;
- Criar uma interface gráfica, permitindo o uso das ferramentas implementadas de forma fácil, rápida e de simples entendimento, mas que não limite os recursos da biblioteca;
- Arquitetura orientada a objetos, com o objetivo de ser facilmente escalável e de simples entendimento;
- Utilizar práticas de XP, adaptando-as conforme as necessidades do grupo.
- Manter a cobertura de testes superior a 90%;
- Analisar o funcionamento do sistema de coorientação, com o objetivo estendê-lo para projetos futuros.

O Tail foi desenvolvido por Alexandre Oki Takinami, Carlos Eduardo Mansur, Márcio Vinicius dos Santos, Thiago Garutti Thies, Paulo Silveira (mestre em Geometria Computacional pela USP e diretor da Caelum), Julian Monteiro (mestre em sistemas distribuídos pela USP e doutor pelo INRIA, em Sophia Antipolis, França) e Danilo Sato (mestre em Metodologias Ágeis pela USP e Lead Consultant na ThoughtWorks).

Esse projeto foi a primeira parceria entre a Caelum e a USP, onde a Caelum patrocinou o trabalho de conclusão de curso dos 4 graduandos, hoje todos formados.

Caso tenha curiosidade você pode acessar o CVS do projeto, utilizando o seguinte repositório:

```
http://tail.cvs.sourceforge.net/viewvc/tail/
```

2.4 O projeto Argentum: modelando o sistema

O projeto Tail é bastante ambicioso. Tem centenas de recursos, em especial o de sugestão de quando comprar e de quando vender ações. O interessante durante o desenvolvimento do projeto Tail foi que muitos dos bons princípios de orientação a objetos, engenharia de software, design patterns e Programação eXtrema se encaixaram muito bem - por isso, nos inspiramos fortemente nele como base para o FJ-22.

Queremos modelar diversos objetos do nosso sistema, entre eles teremos:

- `Negociação` - guardando preço, quantidade e data;
- `Candlestick` - guardando as informações do Candle, além do volume de dinheiro negociado;
- `SerieTemporal` - que guarda um conjunto de candles.

Essas classes formarão a base do projeto que criaremos durante o treinamento, o **Argentum** (do latim, dinheiro ou prata). As funcionalidades do sistema serão as seguintes:

- Resumir `Negociacoes` em `Candlesticks`.

  Nossa base serão as negociações. Precisamos converter uma lista de negociações em uma lista de Candles.

- Converter `Candlesticks` em `SerieTemporal`.

  Dada uma lista de Candle, precisamos criar uma série temporal.

- Utilizar indicadores técnicos

  Para isso, implementar um pequeno *framework* de indicadores e criar alguns deles de forma a facilitar o desenvolvimento de novos.

- Gerar gráficos

  Embutíveis e interativos na interface gráfica em Java, dos indicadores que criamos.

Para começar a modelar nosso sistema, precisamos entender alguns recursos de design de classes que ainda não foram discutidos no FJ-11. Entre eles podemos citar o uso da imutabilidade de objetos, uso de anotações e aprender a trabalhar e manipular datas usando a API do Java.

2.5 Trabalhando com dinheiro

Até agora, não paramos muito para pensar nos tipos das nossas variáveis e já ganhamos o costume de automaticamente atribuir valores a variáveis `double`. Essa é, contudo, uma prática bastante perigosa!

O problema do `double` é que não é possível especificar a precisão mínima que ele vai guardar e, dessa forma, estamos sujeitos a problemas de arredondamento ao fracionar valores e voltar a somá-los. Por exemplo:

```
double cem = 100.0;
double tres = 3.0;
double resultado = cem / tres;

System.out.println(resultado);
        //        33.333?
        //        33.333333?
        //         33.3?
```

Se não queremos correr o risco de acontecer um arredondamento sem que percebamos, a alternativa é usar a classe `BigDecimal`, que lança exceção quando tentamos fazer uma operação cujo resultado é inexato.

Leia mais sobre ela na própria documentação do Java.

2.6 Palavra chave final

A palavra chave `final` tem várias utilidades. Em uma classe, define que a classe nunca poderá ter uma filha, isso é, não pode ser estendida. A classe `String`, por exemplo, é `final`.

Como modificador de método, `final` indica que aquele método não pode ser reescrito. Métodos muito importantes costumam ser definidos assim. Claro que isso não é necessário declarar caso sua classe já seja `final`.

Ao usarmos como modificador na declaração de variável, indica que o valor daquela variável nunca poderá ser alterado, uma vez atribuído. Se a variável for um atributo, você tem que inicializar seu valor durante a construção do objeto - caso contrário, ocorre um erro de compilação, pois atributos `final` não são inicializados com valores default.

Imagine que, quando criamos um objeto `Negociacao`, não queremos que seu `valor` seja modificado:

```
class Negociacao {

    private final double valor;

    // getters e setters?

}
```

Esse código não compila, nem mesmo com um setter, pois o valor final deveria já ter sido inicializado. Para resolver isso, ou declaramos o `valor` da `Negociacao` direto na declaração do atributo (o que não faz muito sentido nesse caso), ou então populamos pelo construtor:

```
class Negociacao {

    private final double valor;

    public Negociacao(double valor) {
        this.valor = valor;
    }

    // podemos ter um getter, mas nao um setter aqui!

}
```

Uma variável `static final` tem uma cara de constante daquela classe e, se for `public static final`, aí parece uma constante global! Por exemplo, na classe `Collections` do `java.util` existe uma constante `public static final` chamada `EMPTY_LIST`. É convenção que constantes sejam declaradas letras maiúsculas e separadas por travessão (*underscore*) em vez de usar o padrão *camel case*. Outros bons exemplos são o `PI` e o `E`, dentro da `java.lang.Math`.

Isso é muito utilizado, mas hoje no java 5 para criarmos constantes costuma ser muito mais interessante utilizarmos o recurso de enumerações que, além de tipadas, já possuem diversos métodos auxiliares.

No caso da classe `Negociacao`, no entanto, bastará usarmos atributos finais e também marcarmos a própria classe como final para que ela crie apenas objetos imutáveis.

2.7 Imutabilidade de objetos

> **Effective Java**
>
> Item 15: Minimize mutabilidade

Para que uma classe seja imutável, ela precisa ter algumas características:

- Nenhum método pode modificar seu estado;
- A classe deve ser `final`;
- Os atributos devem ser privados;
- Os atributos devem ser `final`, apenas para legibilidade de código, já que não há métodos que modificam o estado do objeto;
- Caso sua classe tenha composições com objetos mutáveis, eles devem ter acesso exclusivo pela sua classe.

Diversas classes no Java são imutáveis, como a `String` e todas as classes *wrapper*. Outro excelente exemplo de imutabilidade são as classes `BigInteger` e `BigDecimal`:

Qual seria a motivação de criar uma classe de tal maneira?

Objetos podem compartilhar suas composições

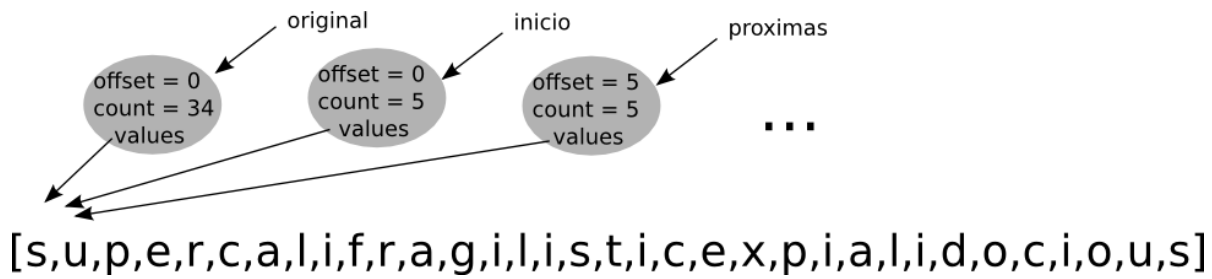
Como o objeto é imutável, a composição interna de cada um pode ser compartilhada entre eles, já que não há chances de algum deles mudar tais atributos. Esse compartilhamento educado possibilita fazer *cache* de suas partes internas, além de facilitar a manipulação desses objetos.

Isso pode ser encarado como o famoso design pattern **Flyweight**.

É fácil entender os benefícios dessa prática quando olhamos para o caso da `String`: objetos do tipo `String` que contêm exatamente o mesmo texto ou partes exatas do texto original (como no caso de usarmos o `substring`) compartilham a *array* privada de `chars`!

Na prática, o que isso quer dizer é que se você tem uma String muito longa e cria várias outras com trechos da original, você não terá que armazenar os caracteres de novo para cada trecho: eles utilizarão o `array` de `chars` da `String` original!

```
String palavra = supercalifragilisticexpialidocious;
String inicio = palavra.substring(0, 5);
String proximas = palavra.substring(5, 10);
String outras = palavra.substring(10, 15);
String resto = palavra.substring(15);
```



Esses objetos também são ideais para usar como chave de tabelas de hash.

Thread safety

Uma das principais vantagens da imutabilidade é em relação a concorrência. Simplesmente não precisamos nos preocupar em relação a isso: como não há método que mude o estado do objeto, então não há como fazer duas modificações acontecerem concorrentemente!

Objetos mais simples

Uma classe imutável é mais simples de dar manutenção. Como não há chances de seu objeto ser modificado, você tem uma série de garantias sobre o uso daquela classe.

Se os construtores já abrangem todas as regras necessárias para validar o estado do objeto, não há preocupação em relação a manter o estado consistente, já que não há chances de modificação.

Uma boa prática de programação é evitar tocar em variáveis parâmetros de um método. Com objetos imutáveis nem existe esse risco quando você os recebe como parâmetro.

Se nossa classe `Negociacao` é imutável, isso remove muitas dúvidas e medos que poderíamos ter durante o desenvolvimento do nosso projeto: saberemos em todos os pontos que os valores da negociação são sempre os mesmos, não corremos o risco de um método que constrói o `candlestick` mexer nos nossos atributos (deixando ou não num estado inconsistente), além de a imutabilidade também garantir que não haverá problemas no caso de acesso concorrente ao objeto.

2.8 Trabalhando com datas: Date e Calendar

Se você fez o FJ-21 conosco, já teve que lidar com as conversões entre `Date` e `Calendar` para pegar a entrada de data de um texto digitado pelo usuário e convertê-lo para um objeto que representa datas em Java.

A classe mais antiga que representa uma data dentro do Java é a `Date`. Ela armazena a data de forma cada momento do tempo seja representado por um número - isso quer dizer, que o `Date` guarda todas as datas como milissegundos que se passaram desde 01/01/1970.

O armazenamento dessa forma não é de todo ruim, mas o problema é que a API não traz métodos que ajudem muito a lidar com situações do dia como, por exemplo, adicionar dias ou meses a uma data.

A classe `Date` não mais é recomendada porque a maior parte de seus métodos estão marcados como `deprecated`, porém ela tem amplo uso legado nas bibliotecas do java. Ela foi substituída no Java 1.1 pelo `Calendar`, para haver suporte correto à internacionalização e à localização do sistema de datas.

Calendar: evolução do Date

A classe abstrata `Calendar` também encapsula um instante em milissegundos, como a `Date`, mas ela provê métodos para manipulação desse momento em termos mais cotidianos como dias, meses e anos. Por ser abstrata, no entanto, não podemos criar objetos que são simplesmente `Calendar`s.

A subclasse concreta de `Calendar` mais usada é a `GregorianCalendar`, que representa o calendário usado pela maior parte dos países -- outras implementações existem, como a do calendário budista `BuddhistCalendar`, mas estas são bem menos usadas e devolvidas de acordo com seu `Locale`.

Há ainda a API nova do Java 8, chamada `java.time`, desenvolvida com base no Joda Time. Trabalharemos aqui com Calendar por ser ainda muito mais difundida, dado que o Java 8 é muito recente. Caso você tenha a possibilidade de trabalhar com Java 8, favoreça o uso da API nova.

Para obter um `Calendar` que encapsula o instante atual (data e hora), usamos o método estático `getInstance()` de `Calendar`.

```
Calendar agora = Calendar.getInstance();
```

Porque não damos `new` diretamente em `GregorianCalendar`? A API do Java fornece esse método estático que **fabrica** um objeto `Calendar` de acordo com uma série de regras que estão encapsuladas dentro de `getInstance`. Esse é o padrão de projeto *factory*, que utilizamos quando queremos esconder a maneira em que um objeto é instanciado. Dessa maneira podemos trocar implementações devolvidas como retorno a medida que nossas necessidades mudem.

Nesse caso algum país que use calendários diferente do gregoriano pode implementar esse método de maneira adequada, retornando o que for necessário de acordo com o `Locale` configurado na máquina.

> **Effective Java**
>
> Item 1: Considere utilizar Factory com métodos estáticos em vez de construtores

Repare ainda que há uma sobrecarga desse método que recebe `Locale` ou `Timezone` como argumento, caso você queira que essa *factory* trabalhe com valores diferentes dos valores que a JVM descobrir em relação ao seu ambiente.

Um outro excelente exemplo de *factory* é o `DriverManager` do `java.sql` que fabrica `Connection` de acordo com os argumentos passados.

A partir de um `Calendar`, podemos saber o valor de seus campos, como ano, mês, dia, hora, minuto, etc. Para isso, usamos o método `get` que recebe um inteiro representando o campo; os valores possíveis estão em constantes na classe `Calendar`.

No exemplo abaixo, imprimimos o dia de hoje e o dia da semana correspondente. Note que o dia da semana devolvido é um inteiro que representa o dia da semana (`Calendar.MONDAY` etc):

```
Calendar c = Calendar.getInstance();
System.out.println(Dia do Mês:  + c.get(Calendar.DAY_OF_MONTH));
System.out.println(Dia da Semana:  + c.get(Calendar.DAY_OF_WEEK));
```

Um possível resultado é:

```
Dia do Mês: 4
Dia da Semana: 5
```

No exemplo acima, o dia da semana **5** representa a **quinta-feira**.

Da mesma forma que podemos pegar os valores dos campos, podemos atribuir novos valores a esses campos por meio dos métodos `set`.

Há diversos métodos `set` em `Calendar`. O mais geral é o que recebe dois argumentos: o primeiro indica qual é o campo (usando aquelas constantes de `Calendar`) e, o segundo, o novo valor. Além desse método, outros métodos `set` recebem valores de determinados campos; o `set` de três argumentos, por exemplo, recebe ano, mês e dia. Vejamos um exemplo de como alterar a data de hoje:

```
Calendar c = Calendar.getInstance();
c.set(2011, Calendar.DECEMBER, 25, 10, 30);
// mudamos a data para as 10:30am do Natal
```

Outro método bastante usado é `add`, que adiciona uma certa quantidade a qualquer campo do `Calendar`. Por exemplo, para uma aplicação de agenda, queremos adicionar um ano à data de hoje:

```
Calendar c = Calendar.getInstance();
c.add(Calendar.YEAR, 1); // adiciona 1 ao ano
```

Note que, embora o método se chame `add`, você pode usá-lo para subtrair valores também; basta colocar uma quantidade negativa no segundo argumento.

Os métodos `after` e `before` são usados para comparar o objeto `Calendar` em questão a outro `Calendar`. O método `after` devolverá `true` quando o objeto atual do `Calendar` representar um momento posterior ao do `Calendar` passado como argumento. Por exemplo, `after` devolverá `false` se compararmos o dia das crianças com o Natal, pois o dia das crianças não vem depois do Natal:

```
Calendar c1 = new GregorianCalendar(2005, Calendar.OCTOBER, 12);
Calendar c2 = new GregorianCalendar(2005, Calendar.DECEMBER, 25);
System.out.println(c1.after(c2));
```

Analogamente, o método `before` verifica se o momento em questão vem antes do momento do `Calendar` que foi passado como argumento. No exemplo acima, `c1.before(c2)` devolverá `true`, pois o dia das crianças vem antes do Natal.

Note que `Calendar` implementa `Comparable`. Isso quer dizer que você pode usar o método `compareTo` para comparar dois calendários. No fundo, `after` e `before` usam o `compareTo` para dar suas respostas - apenas, fazem tal comparação de uma forma mais elegante e encapsulada.

Por último, um dos problemas mais comuns quando lidamos com datas é verificar o intervalo de dias entre duas datas que podem ser até de anos diferentes. O método abaixo devolve o número de dias entre dois objetos Calendar. O cálculo é feito pegando a diferença entre as datas em milissegundos e dividindo esse valor pelo número de milissegundos em um dia:

```
public int diferencaEmDias(Calendar c1, Calendar c2) {
    long m1 = c1.getTimeInMillis();
    long m2 = c2.getTimeInMillis();
    return (int) ((m2  m1) / (24*60*60*1000));
}
```

Relacionando Date e Calendar

Você pode pegar um `Date` de um `Calendar` e vice-versa através dos métodos `getTime` e `setTime` presentes na classe `Calendar`:

```
Calendar c = new GregorianCalendar(2005, Calendar.OCTOBER, 12);
Date d = c.getTime();
c.setTime(d);
```

Isso faz com que você possa operar com datas da maneira nova, mesmo que as APIs ainda usem objetos do tipo `Date` (como é o caso de `java.sql`).

Para saber mais: Classes Deprecated e o JodaTime

O que fazer quando descobrimos que algum método ou alguma classe não saiu bem do jeito que deveria? Simplesmente apagá-la e criar uma nova?

Essa é uma alternativa possível quando apenas o seu programa usa tal classe, mas definitivamente não é uma boa alternativa se sua classe já foi usada por milhões de pessoas no mundo todo.

É o caso das classes do Java. Algumas delas (`Date`, por exemplo) são repensadas anos depois de serem lançadas e soluções melhores aparecem (`Calendar`). Mas, para não quebrar compatibilidade com códigos existentes, o Java mantém as funcionalidades problemáticas ainda na plataforma, mesmo que uma solução melhor exista.

Mas como desencorajar códigos novos a usarem funcionalidades antigas e não mais recomendadas? A prática no Java para isso é marcá-las como **deprecated**. Isso indica aos programadores que não devemos mais usá-las e que futuramente, em uma versão mais nova do Java, podem sair da API (embora isso nunca tenha ocorrido na prática).

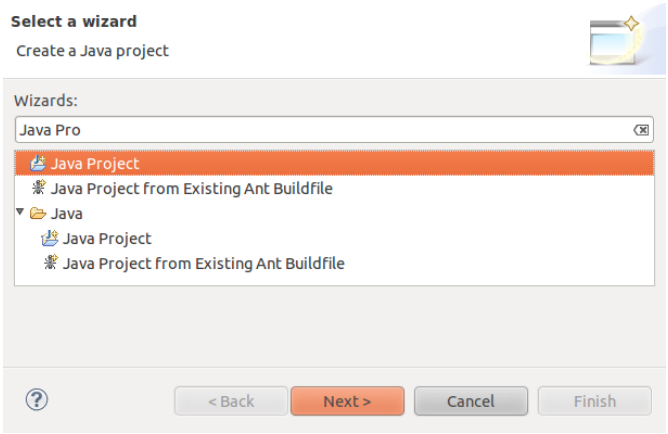
Antes do Java 5, para falar que algo era deprecated, usava-se um comentário especial no Javadoc. A partir do Java 5, a anotação `@Deprecated` foi adicionada à plataforma e garante verificações do próprio compilador (que gera um warning). Olhe o Javadoc da classe `Date` para ver tudo que foi deprecated.

A API de datas do Java, mesmo considerando algumas melhorias da `Calendar` em relação a `Date`, ainda é muito pobre. Numa próxima versão novas classes para facilitar ainda mais o trabalho com datas e horários devem entrar na especificação do Java, baseadas na excelente biblioteca **JodaTime**.

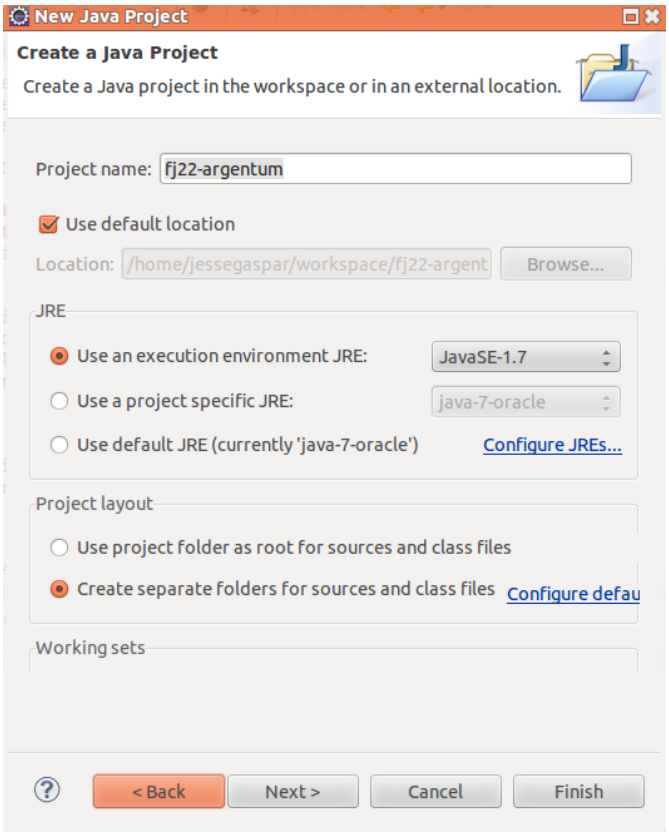
Para mais informações: http://blog.caelum.com.br/2007/03/15/jsr-310-date-and-time-api/ http://jcp.org/en/jsr/detail?id=310

2.9 Exercícios: o modelo do Argentum

1) Vamos criar o projeto `fj22-argentum` no Eclipse, já com o foco em usar a IDE melhor: use o atalho **ctrl + N**, que *cria novo...* e comece a digitar *Java Project*:



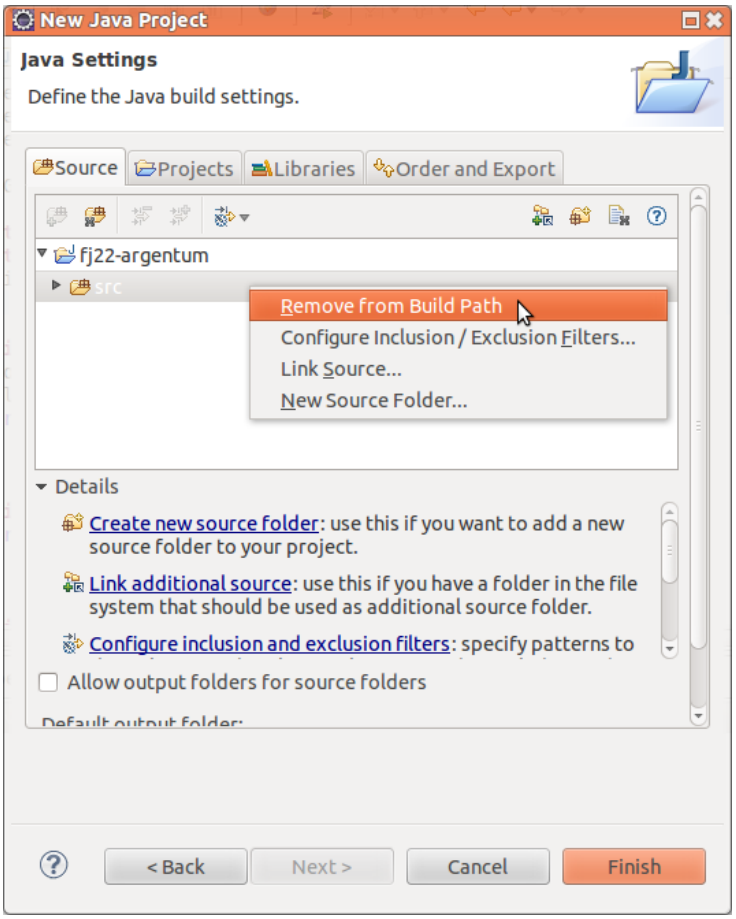
2) Na janela que abrirá em sequência, preencha o nome do projeto como **fj22-argentum** e clique em **Next**:



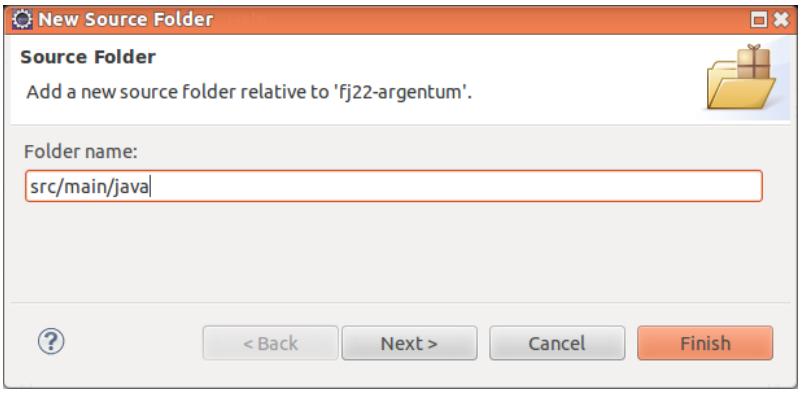
3) Na próxima tela, podemos definir uma série de configurações do projeto (que também podem ser feitas depois, através do menu *Build path -> Configure build path*, clicando com o botão da direita no projeto.

   Queremos mudar o diretório que conterá nosso código fonte. Faremos isso para organizar melhor nosso projeto e utilizar convenções amplamente utilizadas no mercado.

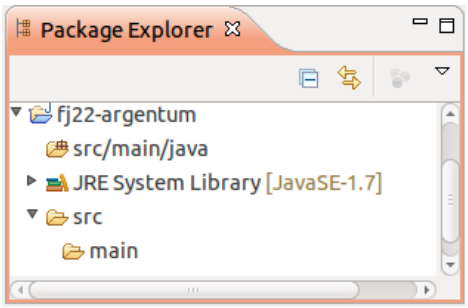
   Nessa tela, **remova** o diretório `src` da lista de diretórios fonte:



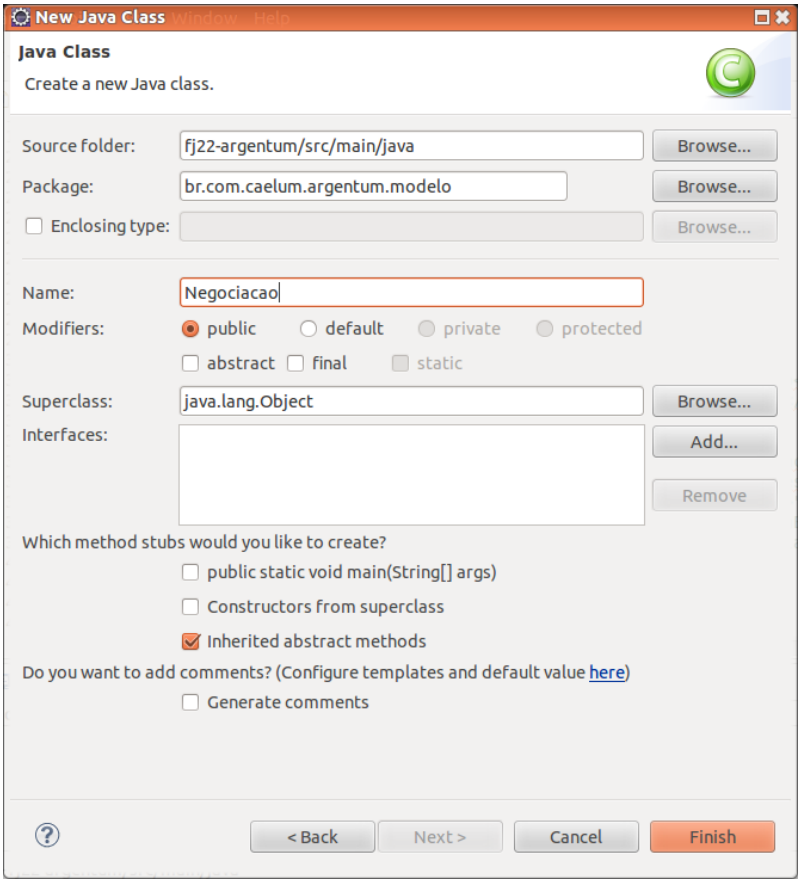
Agora, na mesma tela, adicione um novo diretório fonte, chamado **src/main/java**. Para isso, clique em **Create new source folder** e preencha com **src/main/java**:



4) Agora basta clicar em **Finish**. A estrutura final de seu projeto deve estar parecida com isso:



5) Crie a classe usando **ctrl** + **N** `Class`, chamada `Negociacao` e dentro do pacote `br.com.caelum.argentum.modelo`:



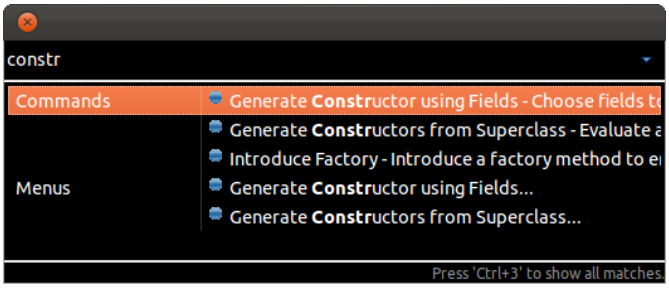
6) Transforme a classe em **final** e já declare os três atributos que fazem parte de uma negociação da bolsa de valores (também como final):

```
public final class Negociacao {
    private final double preco;
    private final int quantidade;
    private final Calendar data;

}
```

Não esqueça de importar o `Calendar`!

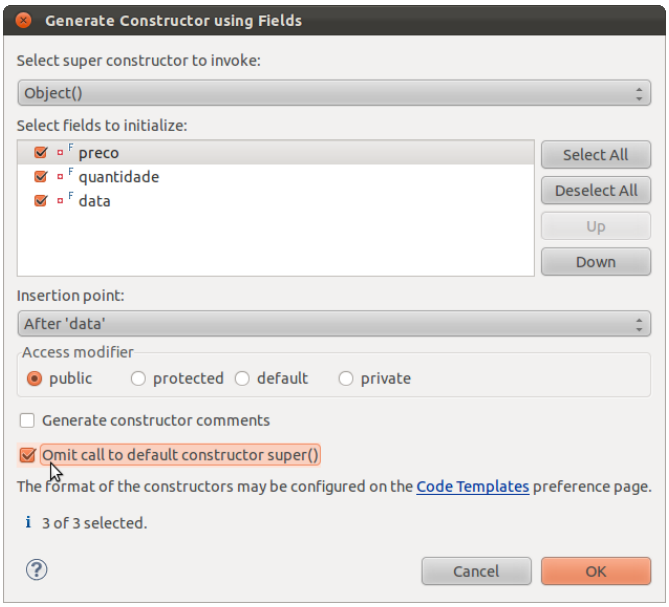
7) Vamos criar um construtor que recebe esses dados, já que são obrigatórios para nosso domínio. Em vez de fazer isso na mão, na edição da classe, use o atalho **ctrl + 3** e comece a digitar *constructor*. Ele vai mostrar uma lista das opções que contêm essa palavra: escolha a *Generate constructor using fields*.

   Alternativamente, tecle **ctrl + 3** e digite *GCUF*, que são as iniciais do menu que queremos acessar.



   Agora, selecione todos os campos e marque para omitir a invocação ao `super`, como na tela abaixo.

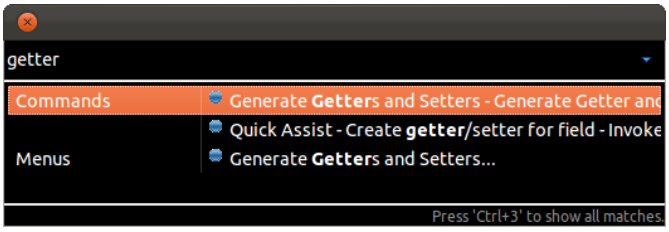
   Atenção para deixar os campos na ordem 'preco, quantidade, data'. Você pode usar os botões *Up* e *Down* para mudar a ordem.



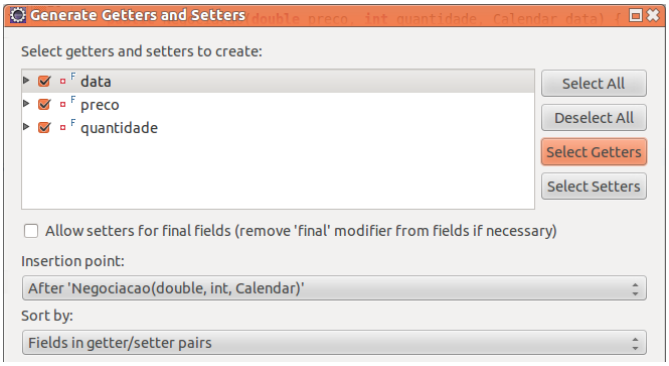
   Pronto! Mande gerar. O seguinte código que será gerado:

```
public Negociacao(double preco, int quantidade, Calendar data) {
    this.preco = preco;
    this.quantidade = quantidade;
    this.data = data;
}
```

8) Agora, vamos gerar os getters dessa classe. Faça **ctrl + 3** e comece a digitar *getter*, as opções aparecerão e basta você escolher *generate getters and setters*. É sempre bom praticar os atalhos do **ctrl + 3**.



Selecione os getters e depois **Finish**:



9) Verifique sua classe. Ela deve estar assim:

```
public final class Negociacao {

    private final double preco;
    private final int quantidade;
    private final Calendar data;

    public Negociacao(double preco, int quantidade, Calendar data) {
        this.preco = preco;
        this.quantidade = quantidade;
        this.data = data;
    }

    public double getPreco() {
        return preco;
    }

    public int getQuantidade() {
        return quantidade;
    }

    public Calendar getData() {
```

```
            return data;
        }
    }
```

10) Um dado importante para termos noção da estabilidade de uma ação na bolsa de valores é o volume de dinheiro negociado em um período.

    Vamos fazer nossa classe `Negociacao` devolver o volume de dinheiro transferido naquela negociação. Na prática, é só multiplicar o `preco` pago pela `quantidade` de ações negociadas, resultando no total de dinheiro que aquela negociação realizou.

    Adicione o método `getVolume` na classe `Negociacao`:

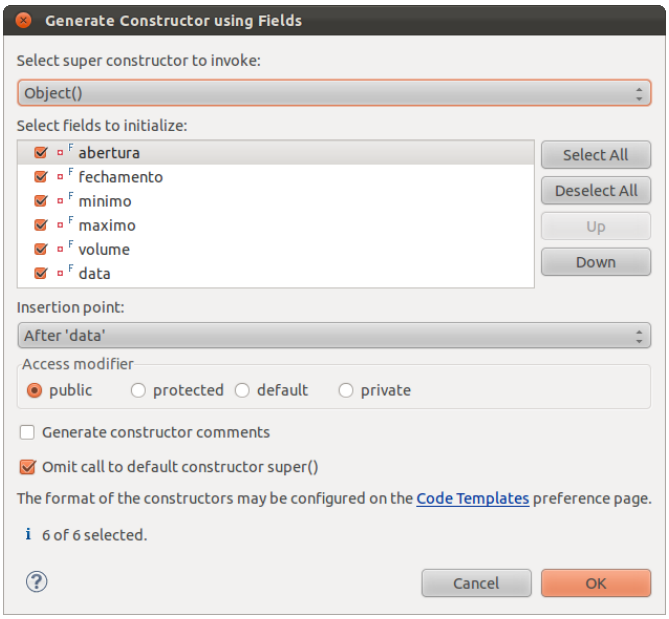
```
public double getVolume() {
    return preco * quantidade;
}
```

    Repare que um método que parece ser um simples *getter* pode (e deve muitas vezes) encapsular regras de negócio e não necessariamente refletem um atributo da classe.

11) Siga o mesmo procedimento para criar a classe `Candlestick`. Use o **ctrl + N** *Class* para isso, marque-a como **final** e adicione os seguintes atributos **finais**, nessa ordem:

```
public final class Candlestick {
    private final double abertura;
    private final double fechamento;
    private final double minimo;
    private final double maximo;
    private final double volume;
    private final Calendar data;

}
```

12) Use o **ctrl + 3** para gerar o construtor com os seis atributos. Atenção à ordem dos parâmetros no construtor:



13) Gere também os seis respectivos getters, usando o **ctrl + 3**.

A classe final deve ficar parecida com a que segue:

```
public final class Candlestick {
    private final double abertura;
    private final double fechamento;
    private final double minimo;
    private final double maximo;
    private final double volume;
    private final Calendar data;

    public Candlestick(double abertura,double fechamento,double minimo,
            double maximo, double volume, Calendar data) {
        this.abertura = abertura;
        this.fechamento = fechamento;
        this.minimo = minimo;
        this.maximo = maximo;
        this.volume = volume;
        this.data = data;
    }

    public double getAbertura() {
        return abertura;
    }
    public double getFechamento() {
        return fechamento;
    }
    public double getMinimo() {
```

```
        return minimo;
    }
    public double getMaximo() {
        return maximo;
    }
    public double getVolume() {
        return volume;
    }
    public Calendar getData() {
        return data;
    }
}
```

14) (opcional) Vamos *adicionar* dois métodos de negócio, para que o `Candlestick` possa nos dizer se ele é do tipo de alta, ou se é de baixa:

```
public boolean isAlta() {
    return this.abertura < this.fechamento;
}

public boolean isBaixa() {
    return this.abertura > this.fechamento;
}
```

2.10 Resumo diário das Negociações

Agora que temos as classes que representam negociações na bolsa de valores (`Negociacao`) e resumos diários dessas negociações (`Candlestick`), falta apenas fazer a ação de resumir as negociações de um dia em uma candle.

A regra é um tanto simples: dentre uma lista de negociações, precisamos descobrir quais são os valores a preencher na `Candlestick`:

- **Abertura**: preço da primeira negociação do dia;
- **Fechamento**: preço da última negociação do dia;
- **Mínimo**: preço da negociação mais barata do dia;
- **Máximo**: preço da negociação mais cara do dia;
- **Volume**: quantidade de dinheiro que passou em todas as negociações nesse dia;
- **Data**: a qual dia o resumo se refere.

Algumas dessas informações são fáceis de encontrar por que temos uma **convenção** no sistema: quando vamos criar a candle, a lista de negociações já vem ordenada por tempo. Dessa forma, a abertura e o fechamento

são triviais: basta recuperar o preço , respectivamente, da primeira e da última negociações do dia!

Já mínimo, máximo e volume precisam que todos os valores sejam verificados. Dessa forma, precisamos passar por cada negociação da lista verificando se aquele valor é menor do que todos os outros que já vimos, maior que nosso máximo atual. Aproveitando esse processo de passar por cada negociação, já vamos somando o volume de cada negociação.

O algoritmo, agora, está completamente especificado! Basta passarmos essas ideias para código. Para isso, lembremos, você pode usar alguns atalhos que já vimos antes:

- **Ctrl + N**: cria novo(a)...
- **Ctrl + espaço**: autocompleta
- **Ctrl + 1**: resolve pequenos problemas de compilação e atribui objetos a variáveis.

Em que classe colocar?

Falta apenas, antes de pôr em prática o que aprendemos, decidirmos onde vai esse código de criação de Candlestick. Pense bem a respeito disso: será que uma negociação deveria saber resumir vários de si em uma candle? Ou será que uma Candlestick deveria saber gerar um objeto do próprio tipo Candlestick a partir de uma lista de negociações.

Em ambos os cenários, nossos modelos têm que ter informações a mais que, na realidade, são responsabilidades que não cabem a eles!

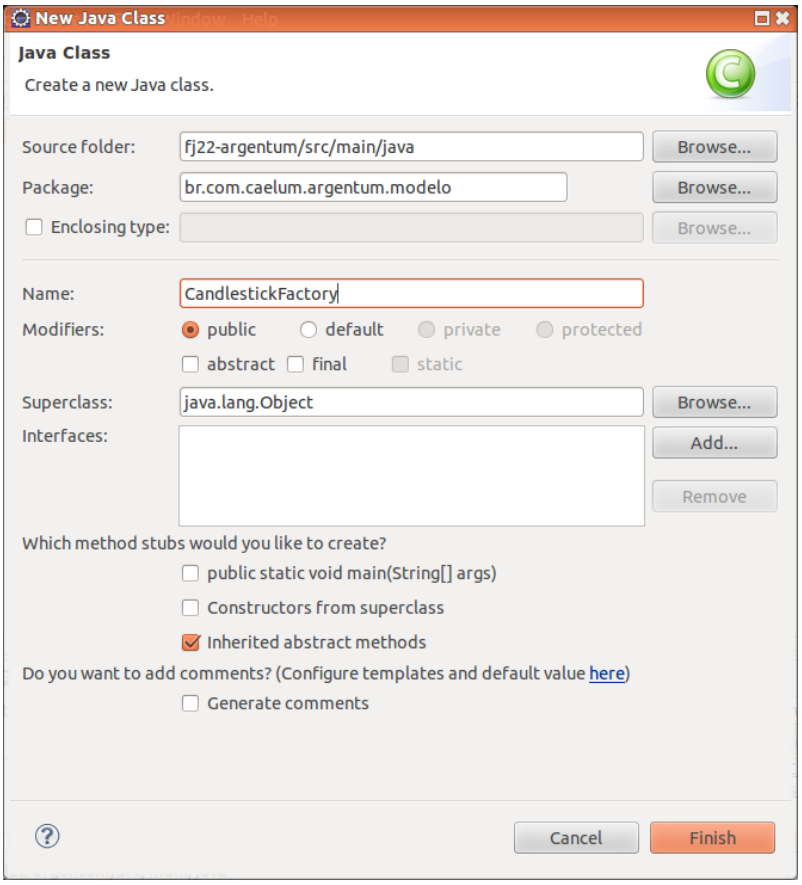
Criaremos, então, uma classe que: *dado a matéria-prima, nos constrói uma candle*. E uma classe com esse comportamento, que recebem o necessário para criar um objeto e **encapsulam** o algoritmo para tal criação, costuma ser chamadas de `Factory`.

No nosso caso particular, essa é uma fábrica que cria `Candlesticks`, então, seu nome fica `CandlestickFactory`.

Perceba que esse nome, apesar de ser um tal *Design Pattern* nada mais faz do que encapsular uma lógica um pouco mais complexa, isto é, apenas aplica boas práticas de orientação a objetos que você já vem estudando desde o FJ-11.

2.11 Exercícios: fábrica de Candlestick

1) Como o resumo de `Negociacoes` em um `Candlestick` é um processo complicado, vamos encapsular sua construção através de uma fábrica, assim como vimos a classe `Calendar`, porém o método de fabricação ficará numa classe a parte, o que também é muito comum. Vamos criar a classe `CandlestickFactory` dentro do pacote `br.com.caelum.argentum.modelo`:



Depois de criá-la, adicione a assinatura do método `constroiCandleParaData` como abaixo:

```
public class CandlestickFactory {
    // ctrl + 1 para adicionar o return automaticamente
    public Candlestick constroiCandleParaData(Calendar data,
                                        List<Negociacao> negociacoes) {

    }
}
```

2) Procuraremos os preços máximo e mínimo percorrendo todas as negociações. Para isso usaremos variáveis auxiliares `maximo` e `minimo` e, dentro do for, verificaremos se o preço da negociação atual é maior que o valor da variável maximo. Se não for, veremos se ele é menor que o `minimo`. Calcularemos o volume somando o volume de cada negociação em uma variável auxiliar chamada `volume`. Poderemos pegar o preço de abertura através de `negociacoes.get(0)` e o de fechamento por `negociacoes.get(negociacoes.size() - 1)`.

```
public class CandlestickFactory {
    public Candlestick constroiCandleParaData(Calendar data,
                                        List<Negociacao> negociacoes) {
        double maximo = negociacoes.get(0).getPreco();
        double minimo = negociacoes.get(0).getPreco();
```

```
        double volume = 0;

        // digite foreach e dê um ctrl + espaço para ajudar a
        // criar o bloco abaixo!
        for (Negociacao negociacao : negociacoes) {
            volume += negociacao.getVolume();

            if (negociacao.getPreco() > maximo) {
                maximo = negociacao.getPreco();
            } else if (negociacao.getPreco() < minimo) {
                minimo = negociacao.getPreco();
            }
        }

        double abertura = negociacoes.get(0).getPreco();
        double fechamento = negociacoes.get(negociacoes.size()1).getPreco();

        return new Candlestick(abertura, fechamento, minimo, maximo,
                                                        volume, data);
    }
}
```

3) Vamos testar nosso código, criando 4 negociações e calculando o `Candlestick`, finalmente. Crie a classe `TestaCandlestickFactory` no pacote `br.com.caelum.argentum.testes`

```
public class TestaCandlestickFactory {

    public static void main(String[] args) {
        Calendar hoje = Calendar.getInstance();

        Negociacao negociacao1 = new Negociacao(40.5, 100, hoje);
        Negociacao negociacao2 = new Negociacao(45.0, 100, hoje);
        Negociacao negociacao3 = new Negociacao(39.8, 100, hoje);
        Negociacao negociacao4 = new Negociacao(42.3, 100, hoje);

        List<Negociacao> negociacoes = Arrays.asList(negociacao1, negociacao2,
                            negociacao3, negociacao4);

        CandlestickFactory fabrica = new CandlestickFactory();
        Candlestick candle = fabrica.constroiCandleParaData(
                                                hoje, negociacoes);

        System.out.println(candle.getAbertura());
        System.out.println(candle.getFechamento());
        System.out.println(candle.getMinimo());
        System.out.println(candle.getMaximo());
```

```
        System.out.println(candle.getVolume());
    }
}
```

O método `asList` da classe `java.util.Arrays` cria uma lista dada uma array. Mas não passamos nenhuma array como argumento! Isso acontece porque esse método aceita `varargs`, possibilitando que invoquemos esse método **separando a array por vírgula**. Algo parecido com um *autoboxing* de arrays.

> **Effective Java**
>
> Item 47: Conheça e use as bibliotecas!

A saída deve ser parecida com:

```
40.5
42.3
39.8
45.0
16760.0
```

2.12 Exercícios opcionais

1)

> **Effective Java**
>
> Item 10: Sempre reescreva o `toString`

Reescreva o `toString` da classe `Candlestick`. Como o `toString` da classe `Calendar` retorna uma `String` bastante complexa, faça com que a data seja corretamente visualizada, usando para isso o `SimpleDateFormat`. Procure sobre essa classe na API do Java.

Ao imprimir um candlestick, por exemplo, a saída deve ser algo como segue:

```
[Abertura 40.5, Fechamento 42.3, Mínima 39.8, Máxima 45.0,
Volume 145234.20, Data 12/07/2008]
```

Para reescrever um método e tirar proveito do Eclipse, a maneira mais direta é de dentro da classe `Candlestick`, fora de qualquer método, pressionar **ctrl + espaço**.

Uma lista com todas as opções de métodos que você pode reescrever vai aparecer. Escolha o `toString`, e ao pressionar *enter* o esqueleto da reescrita será montado.

2) Um `double` segue uma regra bem definida em relação a contas e arredondamento, e para ser rápido e caber em 64 bits, não tem precisão infinita. A classe `BigDecimal` pode oferecer recursos mais interessantes em um ambiente onde as casas decimais são valiosas, como um sistema financeiro. Pesquise a respeito.

3) O construtor da classe `Candlestick` é simplesmente **muito** grande. Poderíamos usar uma *factory*, porém continuaríamos passando muitos argumentos para um determinado método.

Quando construir um objeto é complicado, ou confuso, costumamos usar o padrão **Builder** para resolver isso. Builder é uma classe que ajuda você a construir um determinado objeto em uma série de passos, independente da ordem.

> **Effective Java**
>
> Item 2: Considere usar um builder se o construtor tiver muitos parâmetros!

A ideia é que possamos criar um *candle* da seguinte maneira:

```
CandleBuilder builder = new CandleBuilder();

builder.comAbertura(40.5);
builder.comFechamento(42.3);
builder.comMinimo(39.8);
builder.comMaximo(45.0);
builder.comVolume(145234.20);
builder.comData(new GregorianCalendar(2012, 8, 12, 0, 0, 0));

Candlestick candle = builder.geraCandle();
```

Os *setters* aqui possuem nomes mais curtos e expressivos. Mais ainda: utilizando o padrão de projeto **fluent interface**, podemos tornar o código acima mais conciso, sem perder a legibilidade:

```
Candlestick candle = new CandleBuilder().comAbertura(40.5)
        .comFechamento(42.3).comMinimo(39.8).comMaximo(45.0)
        .comVolume(145234.20).comData(
        new GregorianCalendar(2008, 8, 12, 0, 0, 0)).geraCandle();
```

Para isso, a classe `CandleBuilder` deve usar o seguinte idiomismo:

```
public class CandleBuilder {

    private double abertura;
    // outros 5 atributos

    public CandleBuilder comAbertura(double abertura) {
        this.abertura = abertura;
        return this;
    }

    // outros 5 setters que retornam this
```

```
    public Candlestick geraCandle() {
        return new Candlestick(abertura, fechamento, minimo, maximo,
                        volume, data);
    }
}
```

Escreva um código com `main` que teste essa sua nova classe. Repare como o builder parece bastante com a `StringBuilder`, que é uma classe builder que ajuda a construir `Strings` através de *fluent interface* e métodos auxiliares.

Usos famosos de Fluent Interface e DSLs

Fluent interfaces são muito usadas no Hibernate, por exemplo. O jQuery, uma famosa biblioteca de efeitos javascript, popularizou-se por causa de sua *fluent interface*. O JodaTime e o JMock são dois excelentes exemplos.

São muito usadas (e recomendadas) na construção de DSLs, Domain Specific Languages. Martin Fowler fala bastante sobre fluent interfaces nesse ótimo artigo:

http://martinfowler.com/bliki/FluentInterface.html

Capítulo 3

Testes Automatizados

"Apenas duas coisas são infinitas: o universo e a estupidez humana. E eu não tenho certeza do primeiro."
– Albert Einstein

3.1 Nosso código está funcionando corretamente?

Escrevemos uma quantidade razoável de código no capítulo anterior, meia dúzia de classes. Elas funcionam corretamente? Tudo indica que sim, até criamos um pequeno `main` para verificar isso e fazer as perguntas corretas.

Pode parecer que o código funciona, mas ele tem **muitas** falhas. Olhemos com mais cuidado.

3.2 Exercícios: testando nosso modelo sem frameworks

1) Será que nosso programa funciona para um determinado dia que ocorrer apenas uma única negociação? Vamos escrever o teste e ver o que acontece:

```
public class TestaCandlestickFactoryComUmaNegociacaoApenas {

    public static void main(String[] args) {
        Calendar hoje = Calendar.getInstance();

        Negociacao negociacao1 = new Negociacao(40.5, 100, hoje);

        List<Negociacao> negociacoes = Arrays.asList(negociacao1);

        CandlestickFactory fabrica = new CandlestickFactory();
```

```
        Candlestick candle = fabrica.constroiCandleParaData(hoje, negociacoes);

        System.out.println(candle.getAbertura());
        System.out.println(candle.getFechamento());
        System.out.println(candle.getMinimo());
        System.out.println(candle.getMaximo());
        System.out.println(candle.getVolume());
    }
}
```

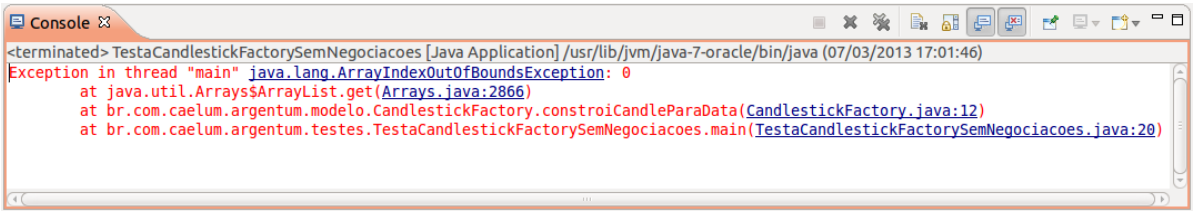
A saída deve indicar `40.5` como todos os valores, e `4050.0` como volume. Tudo parece bem?

2) Mais um teste: as ações menos negociadas podem ficar dias sem nenhuma operação acontecer. O que nosso sistema gera nesse caso?

   Vamos ao teste:

```
public class TestaCandlestickFactorySemNegociacoes {

    public static void main(String[] args) {
        Calendar hoje = Calendar.getInstance();

        List<Negociacao> negociacoes = Arrays.asList();

        CandlestickFactory fabrica = new CandlestickFactory();
        Candlestick candle = fabrica.constroiCandleParaData(hoje, negociacoes);

        System.out.println(candle.getAbertura());
        System.out.println(candle.getFechamento());
        System.out.println(candle.getMinimo());
        System.out.println(candle.getMaximo());
        System.out.println(candle.getVolume());
    }
}
```

   Rodando o que acontece? Você acha essa saída satisfatória? Indica bem o problema?



3) `ArrayIndexOutOfBoundsException` certamente é uma péssima exceção para indicar que não teremos *Candle*.

Qual decisão vamos tomar? Podemos lançar nossa própria *exception*, podemos retornar `null` ou ainda podemos devolver um `Candlestick` que possui um significado especial. Devolver `null` deve ser sempre a última opção.

Vamos retornar um `Candlestick` que possui um volume zero. Para corrigir o erro, vamos alterar o código do nosso `CandlestickFactory`.

Poderíamos usar um `if` logo de cara para verificar se `negociacoes.isEmpty()`, porém podemos tentar algo mais sutil, sem ter que criar vários pontos de `return`.

Vamos então iniciar os valores de `minimo` e `maximo` sem usar a lista, que pode estar vazia. Mas, para nosso algoritmo funcionar, precisaríamos iniciar o `maximo` com um valor **bem pequeno**, isto é, menor do que o da ação mais barata. Assim, quando percorrermos o `for`, qualquer valor que encontremos vai substituir o `maximo`.

No caso do máximo, é fácil pensar nesse valor! Qual é o limite inferior para o preço de uma ação? Será que ele chega a ser negativo? A resposta é não. Ninguém vai vender uma ação por um preço negativo! Portanto, se inicializarmos o máximo com **zero**, é certeza que ele será substituido na primeira iteração do `for`.

Similarmente, queremos inicializar o `minimo` com um valor **bem grande**, maior do que o maior valor possível para o valor de uma ação. Esse é um problema mais complexo, já que não existe um limitante superior tão claro para o preço de uma ação!

Qual valor colocaremos, então? Quanto é um número grande o suficiente? Podemos apelar para a limitação do tipo que estamos usando! Se o preço é um double, certamente não poderemos colocar nenhum valor que estoure o tamanho do double. Felizmente, a classe `Double` conta com a constante que representa o maior double válido! É o `Double.MAX_VALUE`.

Altere o método `constroiCandleParaData` da classe `CandlestickFactory`:

```
double maximo = 0;
double minimo = Double.MAX_VALUE;
```

Além disso, devemos verificar se `negociacoes` está vazia na hora de calcular o preço de `abertura` e `fechamento`. **Altere** novamente o método:

```
double abertura = negociacoes.isEmpty() ? 0 : negociacoes.get(0).getPreco();
double fechamento = negociacoes.isEmpty() ? 0 :
                    negociacoes.get(negociacoes.size()  1).getPreco();
```

Pronto! Rode o teste, deve vir tudo zero e números estranhos para máximo e mínimo!

4) Será que tudo está bem? Rode novamente os outros dois testes, o que acontece?

Incrível! Consertamos um bug, mas adicionamos outro. A situação lhe parece familiar? Nós desenvolvedores vivemos com isso o tempo todo: tentando fugir dos velhos bugs que continuam a reaparecer!

O teste com apenas uma negociação retorna 1.7976931348623157E308 como valor mínimo agora! Mas deveria ser 40.5. Ainda bem que *lembramos* de rodar essa classe, e que percebemos que esse número está diferente do que deveria ser.

Vamos sempre confiar em nossa memória?

5) (opcional) Será que esse erro está ligado a ter apenas uma negociação? Vamos tentar com mais negociações? Crie e rode um teste com as seguintes negociações:

```
Negociacao negociacao1 = new Negociacao(40.5, 100, hoje);
Negociacao negociacao2 = new Negociacao(45.0, 100, hoje);
Negociacao negociacao3 = new Negociacao(49.8, 100, hoje);
Negociacao negociacao4 = new Negociacao(53.3, 100, hoje);
```

E com uma sequência decrescente, funciona? Por quê?

3.3 Definindo melhor o sistema e descobrindo mais bugs

Segue uma lista de dúvidas pertinentes ao Argentum. Algumas dessas perguntas você não saberá responder, porque não definimos muito bem o comportamento de alguns métodos e classes. Outras você saberá responder.

De qualquer maneira, crie um código curto para testar cada uma das situações, em um `main` apropriado.

1) Uma negociação da Petrobras a 30 reais, com uma quantidade negativa de negociações é válida? E com número zero de negociações?

   Em outras palavras, posso dar `new` em uma `Negociacao` com esses dados?

2) Uma negociação com data nula é válida? Posso dar `new Negociacao(10, 5, null)`? Deveria poder?

3) Um candle é realmente imutável? Não podemos mudar a data de um candle de maneira alguma?

4) Um candle em que o preço de abertura é igual ao preço de fechamento, é um candle de alta ou de baixa? O que o sistema diz? O que o sistema deveria dizer?

5) Como geramos um candle de um dia que não houve negociação? O que acontece?

6) E se a ordem das negociações passadas ao `CandlestickFactory` não estiver na ordem crescente das datas? Devemos aceitar? Não devemos?

7) E se essas `Negociações` forem de dias diferentes que a data passada como argumento para a factory?

3.4 Testes de Unidade

Testes de unidade são testes que testam apenas uma classe ou método, verificando se seu comportamento está de acordo com o desejado. Em testes de unidade, verificamos a funcionalidade da classe e/ou método

em questão passando o mínimo possível por outras classes ou dependências do nosso sistema.

> **Unidade**
>
> Unidade é a menor parte testável de uma aplicação. Em uma linguagem de programação orientada a objetos como o Java, a menor unidade é um método.
>
> O termo correto para esses testes é **testes de unidade**, porém o termo *teste unitário* propagou-se e é o mais encontrado nas traduções.

Em testes de unidade, não estamos interessados no comportamento real das dependências da classe, mas em como a classe em questão se comporta diante das possíveis respostas das dependências, ou então se a classe modificou as dependências da maneira esperada.

Para isso, quando criamos um teste de unidade, simulamos a execução de métodos da classe a ser testada. Fazemos isso passando parâmetros (no caso de ser necessário) ao método testado e definimos o resultado que esperamos. Se o resultado for igual ao que definimos como esperado, o teste passa. Caso contrário, falha.

> **Atenção**
>
> Muitas vezes, principalmente quando estamos iniciando no mundo dos testes, é comum criarmos alguns testes que testam muito mais do que o necessário, mais do que apenas a unidade. Tais testes se transformam em verdadeiros **testes de integração** (esses sim são responsáveis por testar o sistemas como um todo).
>
> Portanto, lembre-se sempre: testes de unidade testam **apenas** unidades!

3.5 JUnit

O **JUnit** (junit.org) é um framework muito simples para facilitar a criação destes testes de unidade e em especial sua execução. Ele possui alguns métodos que tornam seu código de teste bem legível e fácil de fazer as **asserções**.

Uma **asserção** é uma afirmação: alguma invariante que em determinado ponto de execução você quer garantir que é verdadeira. Se aquilo não for verdade, o teste deve indicar uma falha, a ser reportada para o programador, indicando um possível bug.

À medida que você mexe no seu código, você roda novamente toda aquela bateria de testes com um comando apenas. Com isso você ganha a confiança de que novos bugs não estão sendo introduzidos (ou reintroduzidos) conforme você cria novas funcionalidades e conserta antigos bugs. Mais fácil do que ocorre quando fazemos os testes dentro do `main`, executando um por vez.

O JUnit possui integração com todas as grandes IDEs, além das ferramentas de build, que vamos conhecer

mais a frente. Vamos agora entender um pouco mais sobre anotações e o `import` estático, que vão facilitar muito o nosso trabalho com o JUnit.

Usando o JUnit - configurando Classpath e seu JAR no Eclipse

O JUnit é uma biblioteca escrita por terceiros que vamos usar no nosso projeto. Precisamos das classes do JUnit para escrever nossos testes. E, como sabemos, o formato de distribuição de bibliotecas Java é o JAR, muito similar a um ZIP com as classes daquela biblioteca.

Precisamos então do JAR do JUnit no nosso projeto. Mas quando rodarmos nossa aplicação, como o Java vai saber que deve incluir as classes daquele determinado JAR junto com nosso programa? (dependência)

É aqui que o **Classpath** entra história: é nele que definimos qual o "*caminho para buscar as classes que vamos usar*". Temos que indicar onde a JVM deve buscar as classes para compilar e rodar nossa aplicação.

Há algumas formas de configurarmos o *classpath*:

- Configurando uma variável de ambiente (**desaconselhado**);
- Passando como argumento em linha de comando (**trabalhoso**);
- Utilizando ferramentas como Ant e Maven (veremos mais a frente);
- Deixando o eclipse configurar por você.

No Eclipse, é muito simples:

1) Clique com o botão direito em cima do nome do seu projeto.
2) Escolha a opção *Properties*.
3) Na parte esquerda da tela, selecione a opção "*Java Build Path*".

E, nessa tela:

1) "*Java Build Path*" é onde você configura o *classpath* do seu projeto: lista de locais definidos que, por padrão, só vêm com a máquina virtual configurada;
2) Opções para adicionar mais caminhos, "Add JARs..." adiciona Jar's que estejam no seu projeto; "Add External JARs" adiciona Jar's que estejam em qualquer outro lugar da máquina, porém guardará uma referência para aquele caminho (então seu projeto poderá não funcionar corretamente quando colocado em outro micro, mas existe como utilizar variáveis para isso);

No caso do JUnit, por existir integração direta com Eclipse, o processo é ainda mais fácil, como veremos no exercício. Mas para todas as outras bibliotecas que formos usar, basta copiar o JAR e adicioná-lo ao *Build Path*. Vamos ver esse procedimento com detalhes quando usarmos as bibliotecas que trabalham com XML e gráficos em capítulos posteriores.

3.6 Anotações

Anotação é a maneira de escrever metadados na própria classe, isto é, configurações ou outras informações pertinentes a essa classe. Esse recurso foi introduzido no Java 5.0. Algumas anotações podem ser mantidas (*retained*) no `.class`, permitindo que possamos reaver essas informações, se necessário.

É utilizada, por exemplo, para indicar que determinada classe deve ser processada por um framework de uma certa maneira, evitando assim as clássicas configurações através de centenas de linhas de XML.

Apesar dessa propriedade interessante, algumas anotações servem apenas para indicar algo ao compilador. `@Override` é o exemplo disso. Caso você use essa anotação em um método que não foi reescrito, vai haver um erro de compilação! A vantagem de usá-la é apenas para facilitar a legibilidade.

`@Deprecated` indica que um método não deve ser mais utilizado por algum motivo e decidiram não retirá-lo da API para não quebrar programas que já funcionavam anteriormente.

`@SurpressWarnings` indica para o compilador que ele não deve dar warnings a respeito de determinado problema, indicando que o programador sabe o que está fazendo. Um exemplo é o warning que o compilador do Eclipse dá quando você não usa determinada variável. Você vai ver que um dos quick fixes é a sugestão de usar o `@SupressWarnings`.

Anotações podem receber parâmetros. Existem muitas delas na API do java 5, mas realmente é ainda mais utilizada em frameworks, como o Hibernate 3, o EJB 3 e o JUnit4.

3.7 JUnit4, convenções e anotação

Para cada classe, teremos uma classe correspondente (por convenção, com o sufixo `Test`) que contará todos os testes relativos aos métodos dessa classe. Essa classe ficará no pacote de mesmo nome, mas na *Source Folder* de testes (`src/test/java`).

Por exemplo, para a nossa `CandlestickFactory`, teremos a `CandlestickFactoryTest`:

```
package br.com.caelum.argentum.modelo;

public class CandlestickFactoryTest {

    public void sequenciaSimplesDeNegociacoes() {
        Calendar hoje = Calendar.getInstance();

        Negociacao negociacao1 = new Negociacao(40.5, 100, hoje);
        Negociacao negociacao2 = new Negociacao(45.0, 100, hoje);
        Negociacao negociacao3 = new Negociacao(39.8, 100, hoje);
        Negociacao negociacao4 = new Negociacao(42.3, 100, hoje);

        List<Negociacao> negociacoes = Arrays.asList(negociacao1, negociacao2,
```

```
                                        negociacao3, negociacao4);

        CandlestickFactory fabrica = new CandlestickFactory();
        Candlestick candle = fabrica.constroiCandleParaData(hoje, negociacoes);
    }
}
```

Em vez de um `main`, criamos um método com nome expressivo para descrever a situação que ele está testando. Mas... como o JUnit saberá que deve executar aquele método? Para isso **anotamos** este método com `@Test`, que fará com que o JUnit saiba no momento de execução, por reflection, de que aquele método deva ser executado:

```
public class CandlestickFactoryTest {

    @Test
    public void sequenciaSimplesDeNegociacoes() {
        // ...
    }
}
```

Pronto! Quando rodarmos essa classe como sendo um teste do JUnit, esse método será executado e a View do JUnit no Eclipse mostrará se tudo ocorreu bem. Tudo ocorre bem quando o método é executado sem lançar exceções inesperadas e se todas as **asserções** passarem.

Uma asserção é uma verificação. Ela é realizada através dos métodos estáticos da classe `Assert`, importada do `org.junit`. Por exemplo, podemos verificar se o valor de abertura desse candle é 40.5:

```
Assert.assertEquals(40.5, candle.getAbertura(), 0.00001);
```

O primeiro argumento é o que chamamos de *expected*, e ele representa o valor que esperamos para argumento seguinte (chamado de *actual*). Se o valor real for diferente do esperado, o teste não passará e uma barrinha vermelha será mostrada, juntamente com uma mensagem que diz:

expected <valor esperado> but was <o que realmente deu>

Double é inexato

Logo na primeira discussão desse curso, conversamos sobre a inexatidão do `double` ao trabalhar com arredondamentos. Porém, diversas vezes, gostaríamos de comparar o double esperado e o valor real, sem nos preocupar com diferenças de arredondamento quando elas são **muito** pequenas.

O JUnit trata esse caso adicionando um terceiro argumento, que só é necessário quando comparamos valores **double** ou **float**. Ele é um delta que se aceita para o erro de comparação entre o valor esperado e o real.

Por exemplo, quando lidamos com dinheiro, o que nos importa são as duas primeiras casas decimais e, portanto, não há problemas se o erro for na quinta casa decimal. Em softwares de engenharia, no entanto, um erro na quarta casa decimal pode ser um grande problema e, portanto, o delta deve ser ainda menor.

Nosso código final do teste, agora com as devidas asserções, ficará assim:

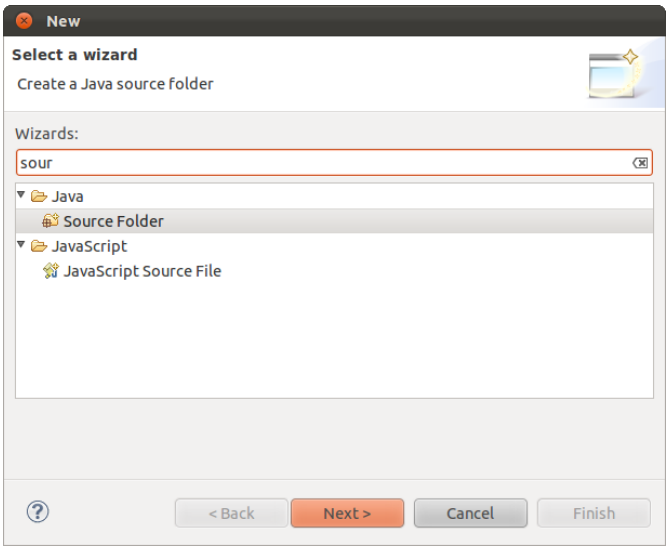
```
public class CandlestickFactoryTest {

    @Test
    public void sequenciaSimplesDeNegociacoes() {
        Calendar hoje = Calendar.getInstance();

        Negociacao negociacao1 = new Negociacao(40.5, 100, hoje);
        Negociacao negociacao2 = new Negociacao(45.0, 100, hoje);
        Negociacao negociacao3 = new Negociacao(39.8, 100, hoje);
        Negociacao negociacao4 = new Negociacao(42.3, 100, hoje);

        List<Negociacao> negociacoes = Arrays.asList(negociacao1, negociacao2,
                                        negociacao3, negociacao4);

        CandlestickFactory fabrica = new CandlestickFactory();
        Candlestick candle = fabrica.constroiCandleParaData(hoje, negociacoes);

        Assert.assertEquals(40.5, candle.getAbertura(), 0.00001);
        Assert.assertEquals(42.3, candle.getFechamento(), 0.00001);
        Assert.assertEquals(39.8, candle.getMinimo(), 0.00001);
        Assert.assertEquals(45.0, candle.getMaximo(), 0.00001);
        Assert.assertEquals(1676.0, candle.getVolume(), 0.00001);
    }
}
```

Existem ainda outras anotações principais e métodos importantes da classe `Assert`, que conheceremos no decorrer da construção do projeto.

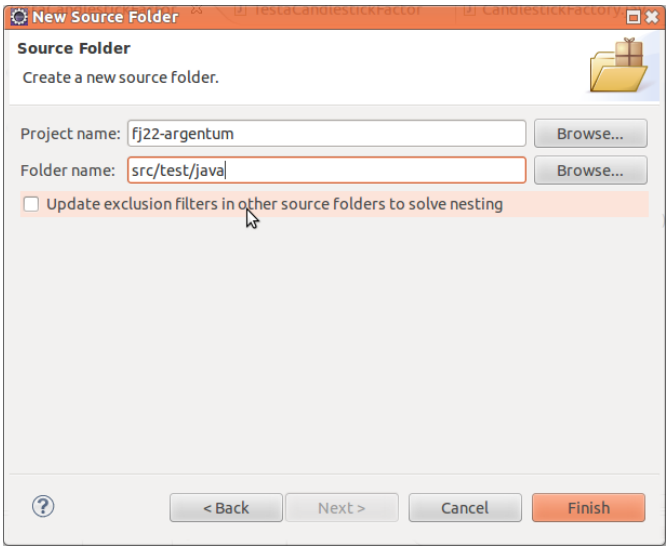
3.8 Exercícios: migrando os testes do main para JUnit

1) É considerada boa prática separar as classes de testes das classes principais. Para isso, normalmente se cria um novo *source folder* apenas para os testes. Vamos fazer isso:

   a) **Ctrl + N** e comece a digitar "Source Folder" até que o filtro a encontre:



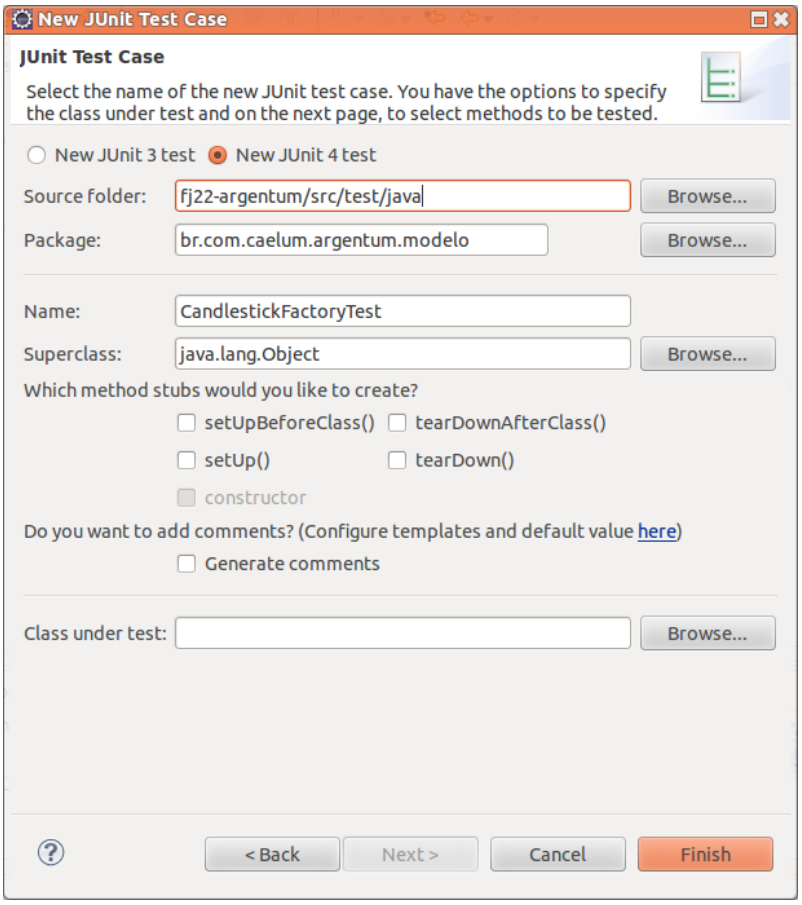
b) Preencha com **src/test/java** e clique **Finish**:



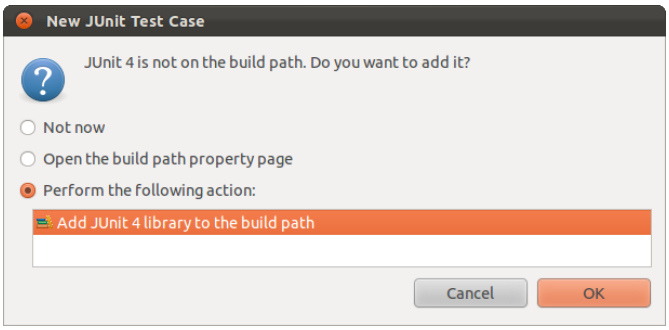
É nesse novo diretório em que você colocará todos seus testes de unidade.

2) Vamos criar um novo *unit test* em cima da classe `CandlestickFactory`. O Eclipse já ajuda bastante: com o editor na CandlestickFactory, crie um novo (**ctrl + N**) JUnit Test Case.

   Na janela seguinte, selecione o *source folder* como **src/test/java**. Não esqueça, também, de **selecionar JUnit4**.



Ao clicar em *Finish*, o Eclipse te perguntará se pode adicionar os jars do JUnit no projeto.



A anotação `@Test` indica que aquele método deve ser executado na bateria de testes, e a classe `Assert` possui uma série de métodos estáticos que realizam comparações, e no caso de algum problema um exceção é lançada.

Vamos colocar primeiro o teste inicial:

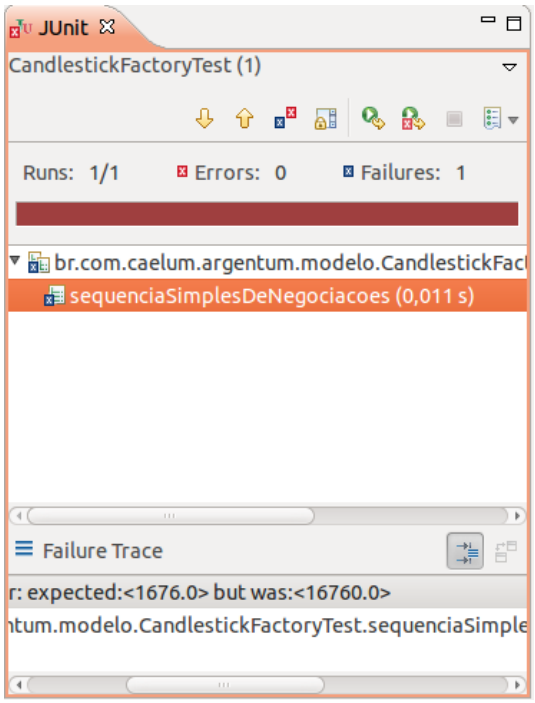
```
public class CandlestickFactoryTest {

    @Test
    public void sequenciaSimplesDeNegociacoes() {
```

```
        Calendar hoje = Calendar.getInstance();

        Negociacao negociacao1 = new Negociacao(40.5, 100, hoje);
        Negociacao negociacao2 = new Negociacao(45.0, 100, hoje);
        Negociacao negociacao3 = new Negociacao(39.8, 100, hoje);
        Negociacao negociacao4 = new Negociacao(42.3, 100, hoje);

        List<Negociacao> negociacoes = Arrays.asList(negociacao1, negociacao2,
                                        negociacao3, negociacao4);

        CandlestickFactory fabrica = new CandlestickFactory();
        Candlestick candle = fabrica.constroiCandleParaData(hoje, negociacoes);

        Assert.assertEquals(40.5, candle.getAbertura(), 0.00001);
        Assert.assertEquals(42.3, candle.getFechamento(), 0.00001);
        Assert.assertEquals(39.8, candle.getMinimo(), 0.00001);
        Assert.assertEquals(45.0, candle.getMaximo(), 0.00001);
        Assert.assertEquals(1676.0, candle.getVolume(), 0.00001);
    }
}
```

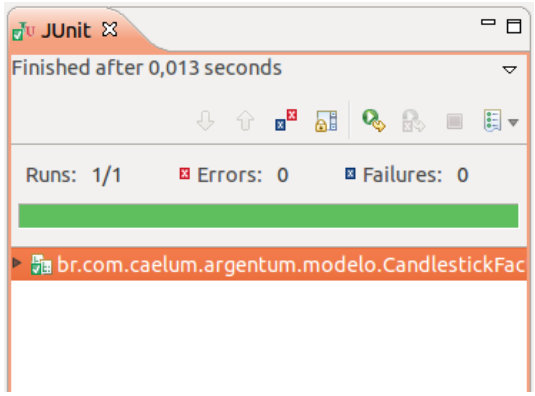
Para rodar, use qualquer um dos seguintes atalhos:

- **ctrl + F11**: roda o que estiver aberto no editor;
- **alt + shift + X** (solte) **T**: roda testes do JUnit.

Não se assuste! **Houve a falha porque o número esperado do volume está errado no teste.** Repare que o Eclipse já associa a falha para a linha exata da asserção e explica porque falhou:



O número correto é mesmo **16760.0**. Adicione esse zero na classe de teste e rode-o novamente:



É comum digitarmos errado no teste e o teste falhar, por isso, é importante sempre verificar a corretude do teste, também!

3) Vamos **adicionar** outro método de teste à mesma classe `CandlestickFactoryTest`, dessa vez para testar o método no caso de não haver nenhuma negociação:

```
@Test
public void semNegociacoesGeraCandleComZeros() {
    Calendar hoje = Calendar.getInstance();

    List<Negociacao> negociacoes = Arrays.asList();

    CandlestickFactory fabrica = new CandlestickFactory();
```

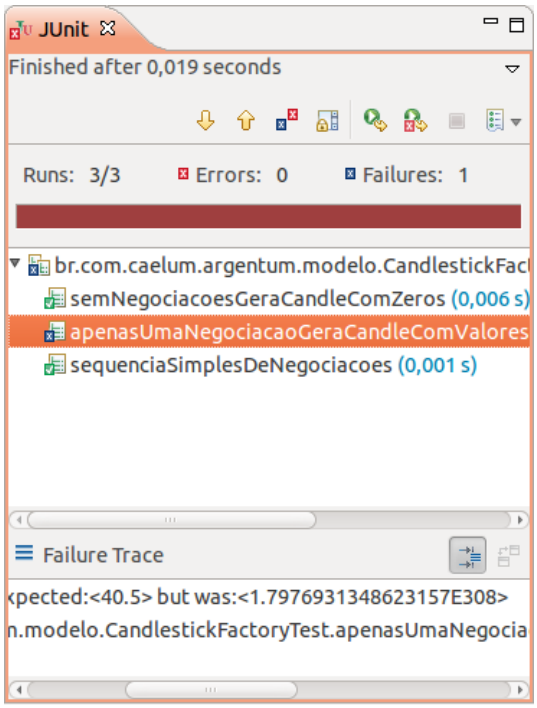
```
    Candlestick candle = fabrica.constroiCandleParaData(hoje, negociacoes);

    Assert.assertEquals(0.0, candle.getVolume(), 0.00001);
}
```

Rode o teste com o mesmo atalho.

4) E, agora, vamos para o que tem apenas uma negociação e estava falhando. Ainda na classe `CandlestickFactoryTest` **adicione** o método: (repare que cada classe de teste possui vários métodos com vários casos diferentes)

```
@Test
public void apenasUmaNegociacaoGeraCandleComValoresIguais() {
    Calendar hoje = Calendar.getInstance();

    Negociacao negociacao1 = new Negociacao(40.5, 100, hoje);

    List<Negociacao> negociacoes = Arrays.asList(negociacao1);

    CandlestickFactory fabrica = new CandlestickFactory();
    Candlestick candle = fabrica.constroiCandleParaData(hoje, negociacoes);

    Assert.assertEquals(40.5, candle.getAbertura(), 0.00001);
    Assert.assertEquals(40.5, candle.getFechamento(), 0.00001);
    Assert.assertEquals(40.5, candle.getMinimo(), 0.00001);
    Assert.assertEquals(40.5, candle.getMaximo(), 0.00001);
    Assert.assertEquals(4050.0, candle.getVolume(), 0.00001);
}
```

Rode o teste. Repare no erro:



Como consertar?

3.9 Vale a pena testar classes de modelo?

Faz todo sentido testar classes como a `CandlestickFactory`, já que existe um algoritmo nela, alguma lógica que deve ser executada e há uma grande chance de termos esquecido algum comportamento para casos incomuns - como vimos nos testes anteriores.

Mas as classes de modelo, Negociacao e Candlestick, também precisam ser testadas?

A resposta para essa pergunta é um grande e sonoro **sim**! Apesar de serem classes mais simples elas também têm comportamentos específicos como:

1) as classes `Negociacao` e `Candlestick` devem ser **imutáveis**, isto é, não devemos ser capazes de alterar nenhuma de suas informações depois que o objeto é criado;

2) valores negativos também não deveriam estar presentes nas negociações e candles;

3) se você fez o opcional CandleBuilder, ele não deveria gerar a candle se os valores não tiverem sido preenchidos;

4) etc...

Por essa razão, ainda que sejam classes mais simples, elas merecem ter sua integridade testada - mesmo porque são os objetos que representam nosso modelo de negócios, o coração do sistema que estamos desenvolvendo.

3.10 Exercícios: novos testes

1) A classe `Negociacao` é realmente imutável?

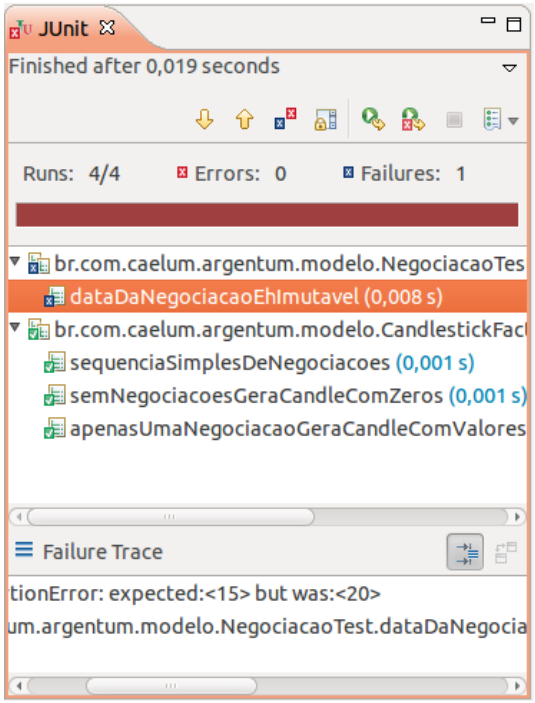
Vamos criar um novo *Unit Test* para a classe `Negociacao`. O processo é o mesmo que fizemos para o teste da `CandlestickFactory`: abra a classe `Negociacao` no editor e faça **Ctrl + N** JUnit Test Case.

Lembre-se de alterar a Source Folder para `src/test/java` e selecionar o JUnit 4.

```
public class NegociacaoTest {

    @Test
    public void dataDaNegociacaoEhImutavel() {
        // se criar um negocio no dia 15...
        Calendar c = Calendar.getInstance();
        c.set(Calendar.DAY_OF_MONTH, 15);
        Negociacao n = new Negociacao(10, 5, c);


        // ainda que eu tente mudar a data para 20...
        n.getData().set(Calendar.DAY_OF_MONTH, 20);

        // ele continua no dia 15.
        Assert.assertEquals(15, n.getData().get(Calendar.DAY_OF_MONTH));
    }
}
```

Você pode rodar esse teste apenas, usando o atalho (alt + shift + X T) ou pode fazer melhor e o que é mais comum, rodar **todos** os testes de unidade de um projeto.

Basta selecionar o projeto na *View* Package Explorer e mandar rodar os testes: para essa ação, o único atalho possível é o `alt + shift + X T`.



Esse teste falha porque devolvemos um objeto mutável através de um *getter*. Deveríamos ter retornado uma cópia desse objeto para nos assegurarmos que o original permanece intacto.

> **Effective Java**
>
> Item 39: Faça cópias defensivas quando necessário.

Basta **alterar** a classe `Negociacao` e utilizar o método `clone` que todos os objetos têm (mas só quem implementa `Cloneable` executará com êxito):

```
public Calendar getData() {
    return (Calendar) this.data.clone();
}
```

Sem `clone`, precisaríamos fazer esse processo na mão. Com `Calendar` é relativamente fácil:

```
public Calendar getData() {
    Calendar copia = Calendar.getInstance();
    copia.setTimeInMillis(this.data.getTimeInMillis());
    return copia;
}
```

Com outras classes, em especial as que tem vários objetos conectados, isso pode ser mais complicado.

> **Listas e arrays**
>
> Esse também é um problema que ocorre muito com coleções e arrays: se você retorna uma *List* que é um atributo seu, qualquer um pode adicionar ou remover um elemento de lá, causando estrago nos seus atributos internos.
>
> Os métodos `Collections.unmodifiableList(List)` e outros ajudam bastante nesse trabalho.

2) Podemos criar uma `Negociacao` com data nula? Por enquanto, podemos, mas não deveríamos. Para que outras partes do meu sistema não se surpreendam mais tarde, vamos impedir que a Negociacao seja criada se sua data estiver nula, isto é, vamos lançar uma exceção.

   Mas qual exceção? Vale a pena criar uma nova exceção para isso?

   A exceção padrão no Java para cuidar de parâmetros indesejáveis é a `IllegalArgumentException` então, em vez de criar uma nova com a mesma semântica, vamos usá-la para isso.

> **Effective Java**
>
> Item 60: Favoreça o uso das exceções padrões!

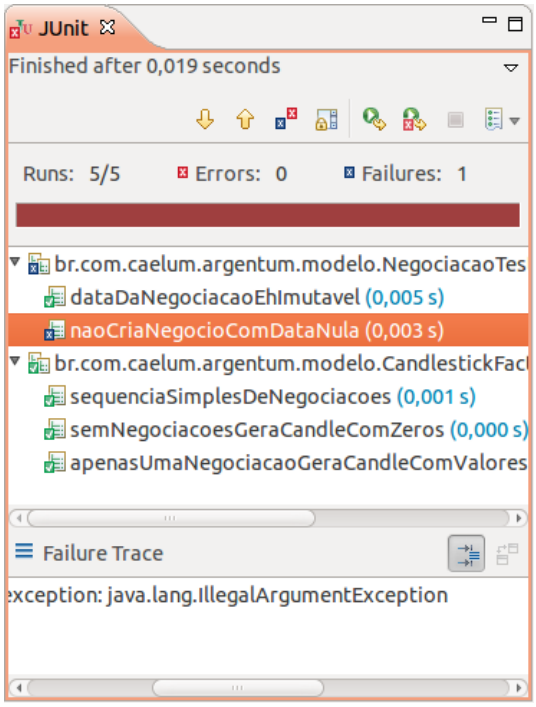
   Antes de fazer a modificação na classe, vamos preparar o teste. Mas o que queremos testar? Queremos saber se nossa classe `Negociacao` não permite a criação do objeto e lança uma `IllegalArgumentException` quando passamos `null` no construtor.

   Ou seja, *esperamos* que uma exceção aconteça! Para o teste *passar*, ele precisa dar a exceção (parece meio contraditório). É fácil fazer isso com JUnit.

   Adicione um novo método `naoCriaNegociacaoComDataNula` na classe `NegociacaoTest`. Repare que agora temos um argumento na anotação `expected=IllegalArgumentException.class`. Isso indica que, para esse teste ser considerado um sucesso, uma exceção deve ser lançada daquele tipo. Caso contrário será uma falha:

```
@Test(expected=IllegalArgumentException.class)
public void naoCriaNegociacaoComDataNula() {
    new Negociacao(10, 5, null);
}
```

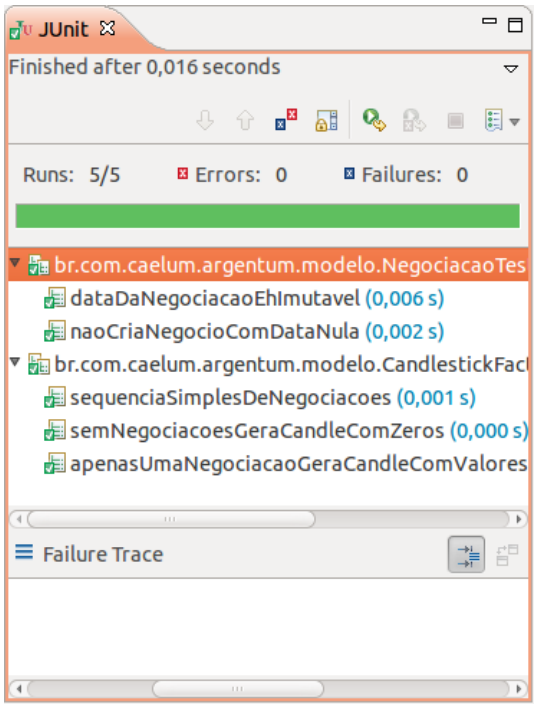
   Rode os testes. Barrinha vermelha! Já que ainda não verificamos o argumento na classe `Negociacao` e ainda não lançamos a exceção:



Vamos **alterar** a classe `Negociacao`, para que ela lance a exceção no caso de data nula. No construtor, adicione o seguinte `if`:

```
public Negociacao(double preco, int quantidade, Calendar data) {
    if (data == null) {
        throw new IllegalArgumentException(data nao pode ser nula);
    }
    this.preco = preco;
    this.quantidade = quantidade;
    this.data = data;
}
```

Rode novamente os testes.



3) (opcional) Nosso teste para quando não há negociações na `CandlestickFactory` está verificando apenas se o volume é zero. Ele também poderia verificar que os outros valores dessa candle são zero.

   Modifique o método `semNegociacoesGeraCandleComZeros` e adicione os asserts faltantes de abertura, fechamento, mínimo e máximo.

   O teste vai parar de passar!

   Corrija ele da mesma forma que resolvemos o problema para as variáveis `abertura` e `fechamento`.

4) (opcional) Um Candlestick pode ter preço máximo menor que o preço mínimo? Não deveria.

   Crie um novo teste, o `CandlestickTest`, da maneira que fizemos com o `Negociacao`. É boa prática que todos os testes da classe `X` se encontrem em `XTest`.

   Dentro dele, crie o `precoMaximoNaoPodeSerMenorQueMinimo` e faça um `new` passando argumentos que quebrem isso. O teste deve esperar pela `IllegalArgumentException`.

   A ideia é testar se o construtor de Candlestick faz as validações necessárias. Lembre-se que o construtor recebe como argumento `Candlestick(abertura, fechamento, minimo, maximo, volume, data)`, portanto queremos testar se algo assim gera uma exceção (e deveria gerar):

   ```
   new Candlestick(10, 20, 20, 10, 10000, Calendar.getInstance());
   ```

5) (opcional) Um `Candlestick` pode ter data nula? Pode ter algum valor negativo?

   Teste, verifique o que está errado, altere código para que os testes passem! Pegue o ritmo, essa será sua rotina daqui para a frente.

6) (opcional) Crie mais dois testes na `CandlestickFactoryTest`: o `negociacoesEmOrdemCrescenteDeValor` e `negociacoesEmOrdemDecrescenteDeValor`, que devem fazer o que o próprio nome diz.

   Agora eles funcionam?

3.11 Para saber mais: Import Estático

Algumas vezes, escrevemos classes que contém muitos métodos e atributos estáticos (finais, como constantes). Essas classes são classes utilitárias e precisamos sempre nos referir a elas antes de chamar um método ou utilizar um atributo:

```
import pacote.ClasseComMetodosEstaticos;
class UsandoMetodosEstaticos {
    void metodo() {
        ClasseComMetodosEstaticos.metodo1();
        ClasseComMetodosEstaticos.metodo2();
    }
}
```

Começa a ficar muito chato escrever, toda hora, o nome da classe. Para resolver esse problema, no Java 5.0 foi introduzido o `static import`, que importa métodos e atributos estáticos de qualquer classe. Usando essa nova técnica, você pode importar os métodos do exemplo anterior e usá-los diretamente:

```
import static pacote.ClasseComMetodosEstaticos.*;
class UsandoMetodosEstaticos {
    void metodo() {
        metodo1();
        metodo2();
    }
}
```

Apesar de você ter importado todos os métodos e atributos estáticos da classe `ClasseComMetodosEstaticos`, a classe em si não foi importada e, se você tentasse dar `new`, por exemplo, ele não conseguiria encontrá-la, precisando de um `import` normal à parte.

Um bom exemplo de uso são os métodos e atributos estáticos da classe `Assert` do JUnit:

```
import static org.junit.Assert.*;

class TesteMatematico {

    @Test
    void doisMaisDois() {
```

```
        assertEquals(4, 2 + 2);
    }
}
```

Use os imports estáticos dos métodos de `Assert` nos testes de unidade que você escreveu.

3.12 Mais exercícios opcionais

1) Crie um teste para o `CandleBuilder`. Ele possui um grande erro: se só chamarmos alguns dos métodos, e não todos, ele construirá um Candle inválido, com data nula, ou algum número zerado.

   Faça um teste `geracaoDeCandleDeveTerTodosOsDadosNecessarios` que tente isso. O método `geraCandle` deveria lançar outra exception conhecida da biblioteca Java, a `IllegalStateException`, quando invocado antes dos seus outros seis métodos já terem sido.

   O teste deve falhar. Corrija-o criando booleans que indicam se cada método *setter* foi invocado, ou utilizando alguma outra forma de verificação.

2) Se você fez os opcionais do primeiro exercício do capítulo anterior (criação do projeto e dos modelos) você tem os métodos `isAlta` e `isBaixa` na classe `Candlestick`. Contudo, temos um comportamento não especificado nesses métodos: e quando o preço de abertura for igual ao de fechamento?

   Perguntando para nosso cliente, ele nos informou que, nesse caso, o candle deve ser considerado de alta.

   Crie o teste `quandoAberturaIgualFechamentoEhAlta` dentro de `CandlestickTest`, verifique se isso está ocorrendo. Se o teste falhar, faça mudanças no seu código para que a barra volte a ficar verde!

3) O que mais pode ser testado? Testar é viciante, e aumentar o número de testes do nosso sistema começa a virar um hábito divertido e contagioso. Isso não ocorre de imediato, é necessário um tempo para se apaixonar por testes.

3.13 Discussão em aula: testes são importantes?

Capítulo 4

Trabalhando com XML

"Se eu enxerguei longe, foi por ter subido nos ombros de gigantes."
– Isaac Newton

4.1 Os dados da bolsa de valores

Como vamos puxar os dados da bolsa de valores para popular nossos *candles*?

Existem inúmeros formatos para se trabalhar com diversas bolsas. Sem dúvida XML é um formato comumente encontrado em diversas indústrias, inclusive na bolsa de valores.

Utilizaremos esse tal de XML. Para isso, precisamos conhecê-lo mais a fundo, seus objetivos, e como manipulá-lo. Considere que vamos consumir um arquivo XML como o que segue:

```
<list>
    <negociacao>
        <preco>43.5</preco>
        <quantidade>1000</quantidade>
        <data>
            <time>1222333777999</time>
        </data>
    </negociacao>
    <negociacao>
        <preco>44.1</preco>
        <quantidade>700</quantidade>
        <data>
            <time>1222444777999</time>
        </data>
```

```
    </negociacao>
    <negociacao>
        <preco>42.3</preco>
        <quantidade>1200</quantidade>
        <data>
            <time>1222333999777</time>
        </data>
    </negociacao>
</list>
```

Uma lista de negociações! Cada negociação informa o preço, quantidade e uma data. Essa data é composta por um horário dado no formato de `Timestamp`, e opcionalmente um `Timezone`.

4.2 O formato XML

XML (eXtensible Markup Language) é uma formalização da W3C para gerar linguagens de marcação que podem se adaptar a quase qualquer tipo de necessidade. Algo bem extensível, flexível, de fácil leitura e hierarquização. Sua definição formal pode ser encontrada em:

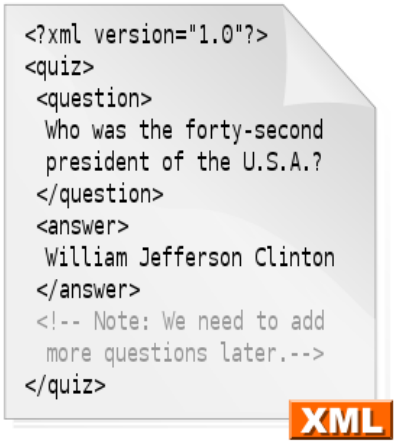
http://www.w3.org/XML/

Exemplo de dados que são armazenados em XMLs e que não conhecemos tão bem, é o formato aberto de gráficos vetoriais, o SVG (usado pelo Corel Draw, Firefox, Inkscape, etc), e o Open Document Format (ODF), formato usado pelo OpenOffice, e hoje em dia um padrão ISO de extrema importância. (na verdade o ODF é um ZIP que contém XMLs internamente).

A ideia era criar uma linguagem de marcação que fosse muito fácil de ser lida e gerada por softwares, e pudesse ser integrada as outras linguagens. Entre seus princípios básicos, definidos pelo W3C:

- Separação do conteúdo da formatação
- Simplicidade e Legibilidade
- Possibilidade de criação de tags novas
- Criação de arquivos para validação (DTDs e schemas)

O XML é uma excelente opção para documentos que precisam ter seus dados organizados com uma certa hierarquia (uma árvore), com relação de pai-filho entre seus elementos. Esse tipo de arquivo é dificilmente organizado com CSVs ou properties. Como a própria imagem do wikipedia nos trás e mostra o uso estruturado e encadeado de maneira hierárquica do XML:



O cabeçalho opcional de todo XML é o que se segue:

```
<?xml version=1.0 encoding=ISO88591?>
```

Esses caracteres não devem ser usados como elemento, e devem ser “escapados":

- &, use &
- ‘, use '
- “, use "
- <, use <
- >, use >

Você pode, em Java, utilizar a classe `String` e as regex do pacote `java.util.regex` para criar um programa que lê um arquivo XML. Isso é uma grande perda de tempo, visto que o Java, assim como quase toda e qualquer linguagem existente, possui uma ou mais formas de ler um XML. O Java possui diversas, vamos ver algumas delas, suas vantagens e suas desvantagens.

4.3 Lendo XML com Java de maneira difícil, o SAX

O SAX (**Simple API for XML**) é uma API para ler dados em XML, também conhecido como **Parser de XML**. Um parser serve para analisar uma estrutura de dados e geralmente o que fazemos é transformá-la em uma outra.

Neste processo de análise também podemos ler o arquivo XML para procurar algum determinado elemento e manipular seu conteúdo.

O parser lê os dados XML como um fluxo ou uma sequência de dados. Baseado no conteúdo lido, o parser vai disparando eventos. É o mesmo que dizer que o parser SAX funciona orientado a eventos.

Existem vários tipos de eventos, por exemplo:

- início do documento XML;
- início de um novo elemento;
- novo atributo;
- início do conteúdo dentro de um elemento.

Para tratar estes eventos, o programador deve passar um objeto **listener** ao parser que será notificado automaticamente pelo parser quando um desses eventos ocorrer. Comumente, este objeto é chamado de **Handler**, **Observer**, ou **Listener** e é quem faz o trabalho necessário de processamento do XML.

```
public class NegociacaoHandler extends DefaultHandler {

      @Override
      public void startDocument() throws SAXException {

      }

      @Override
      public void startElement(String uri, String localName,
                  String name, Attributes attributes) throws SAXException {
        // aqui você é avisado, por exemplo
        // do inicio da tag <preco>
      }

      @Override
      public void characters(char[] chars, int offset, int len)
                                    throws SAXException {
          // aqui você seria avisado do inicio
          // do conteúdo que fica entre as tags, como por exemplo 30
          // de dentro de <preco>30</preco>

          // para saber o que fazer com esses dados, você precisa antes ter
          // guardado em algum atributo qual era a negociação que estava
          // sendo percorrida
      }

      @Override
      public void endElement(String uri, String localName, String name)
                                     throws SAXException {
        // aviso de fechamento de tag
      }
  }
```

A classe `DefaultHandler` permite que você reescreva métodos que vão te notificar sobre quando um elemento (tag) está sendo aberto, quando está sendo fechado, quando caracteres estão sendo parseados (conteúdo de

uma tag), etc.. Você é o responsável por saber em que posição do *object model* (árvore) está, e que atitude deve ser tomada. A interface `ContentHandler` define mais alguns outros métodos.

Curiosidade sobre o SAX

Originalmente o SAX foi escrito só para Java e vem de um projeto da comunidade (http://www.saxproject.org) , mas existem outras implementações em C++, Perl e Python.

O SAX está atualmente na versão 2 e faz parte do JAX-P (Java API for XML Processing).

O SAX somente sabe ler dados e nunca modificá-los e só consegue ler para frente, nunca para trás. Quando passou um determinado pedaço do XML que já foi lido, não há mais como voltar. O parser SAX não guarda a estrutura do documento XML na memória.

Também não há como fazer acesso aleatório aos itens do documento XML, somente sequencial.

Por outro lado, como os dados vão sendo analisados e transformados (pelo Handler) na hora da leitura, o SAX ocupa pouca memória, principalmente porque nunca vai conhecer o documento inteiro e sim somente um pequeno pedaço. Devido também a leitura sequencial, ele é muito rápido comparado com os parsers que analisam a árvore do documento XML completo.

Quando for necessário ler um documento em partes ou só determinado pedaço e apenas uma vez, o SAX parser é uma excelente opção.

StAX - Streaming API for XML

StAX é um projeto que foi desenvolvido pela empresa BEA e padronizado pela JSR-173. Ele é mais novo do que o SAX e foi criado para facilitar o trabalho com XML. StAX faz parte do Java SE 6 e JAX-P.

Como o SAX, o StAX também lê os dados de maneira sequencial. A diferença entre os dois é a forma como é notificada a ocorrência de um evento.

No SAX temos que registrar um `Handler`. É o SAX que avisa o `Handler` e chama os métodos dele. Ele empurra os dados para o `Handler` e por isso ele é um parser do tipo `push`, .

O StAX, ao contrário, não precisa deste Handler. Podemos usar a API do StAX para chamar seus métodos, quando e onde é preciso. O cliente decide, e não o parser. É ele quem pega/tira os dados do StAX e por isso é um parser do tipo `pull`.

O site http://www.xmlpull.org fornece mais informações sobre a técnica de **Pull Parsing**, que tem sido considerada por muitos como a forma mais eficiente de processar documentos xml.

A biblioteca XPP3 é a implementação em Java mais conhecida deste conceito. É usada por outras bibliotecas de processamento de xml, como o CodeHaus XStream.

4.4 XStream

O **XStream** é uma alternativa perfeita para os casos de uso de XML em persistência, transmissão de dados e configuração. Sua facilidade de uso e performance elevada são os seus principais atrativos.

É um projeto hospedado na Codehaus, um repositório de código open source focado em Java, que foi formado por desenvolvedores de famosos projetos como o XDoclet, PicoContainer e Maven. O grupo é patrocinado por empresas como a ThoughWorks, BEA e Atlassian. Entre os desenvolvedores do projeto, Guilherme Silveira da Caelum está também presente.

http://xstream.codehaus.org

Diversos projetos opensource, como o container de inversão de controle NanoContainer, o framework de redes neurais Joone, dentre outros, passaram a usar XStream depois de experiências com outras bibliotecas. O XStream é conhecido pela sua extrema facilidade de uso. Repare que raramente precisaremos fazer configurações ou mapeamentos, como é extremamente comum nas outras bibliotecas mesmo para os casos mais básicos.

Como gerar o XML de uma negociação? Primeiramente devemos ter uma referência para o objeto. Podemos simplesmente criá-lo e populá-lo ou então deixar que o Hibernate faça isso.

Com a referência `negociacao` em mãos, basta agora pedirmos ao XStream que gera o XML correspondente:

```
Negociacao negociacao = new Negociacao(42.3, 100, Calendar.getInstance());

XStream stream = new XStream(new DomDriver());
System.out.println(stream.toXML(negociacao));
```

E o resultado é:

```
<br.com.caelum.argentum.Negociacao>
    <preco>42.3</preco>
    <quantidade>100</quantidade>
    <data>
        <time>1220009639873</time>
        <timezone>America/Sao_Paulo</timezone>
    </data>
</br.com.caelum.argentum.Negociacao>
```

A classe XStream atua como **façade** de acesso para os principais recursos da biblioteca. O construtor da classe XStream recebe como argumento um `Driver`, que é a engine que vai gerar/consumir o XML. Aqui você pode definir se quer usar SAX, DOM, DOM4J dentre outros, e com isso o XStream será mais rápido, mais lento, usar mais ou menos memória, etc.

O default do XStream é usar um driver chamado XPP3, desenvolvido na universidade de Indiana e conhecido por ser extremamente rápido (leia mais no box de pull parsers). Para usá-lo você precisa de um outro JAR no classpath do seu projeto.

O método `toXML` retorna uma `String`. Isso pode gastar muita memória no caso de você serializar uma lista grande de objetos. Ainda existe um overload do toXML, que além de um `Object` recebe um `OutputStream` como argumento para você poder gravar diretamente num arquivo, socket, etc.

Diferentemente de outros parsers do Java, o XStream serializa por default os objetos através de seus atributos (sejam privados ou não), e não através de getters e setters.

Repare que o XStream gerou a tag raiz com o nome de `br.com.caelum.argentum.Negociacao`. Isso porque não existe um conversor para ela, então ele usa o próprio nome da classe e gera o XML recursivamente para cada atributo não transiente daquela classe.

Porém, muitas vezes temos um esquema de XML já muito bem definido, ou simplesmente não queremos gerar um XML com cara de java. Para isso podemos utilizar um `alias`. Vamos modificar nosso código que gera o XML:

```
XStream stream = new XStream(new DomDriver());
stream.alias(negociacao, Negociacao.class);
```

Essa configuração também pode ser feita através da anotação `@XStreamAlias("negociacao")` em cima da classe `Negociacao`.

Podemos agora fazer o processo inverso. Dado um XML que representa um bean da nossa classe `Negociacao`, queremos popular esse bean. O código é novamente extremamente simples:

```
XStream stream = new XStream(new DomDriver());
stream.alias(negociacao, Negociacao.class);
Negociacao n = (Negociacao) stream.fromXML(<negociacao> +
                                        <preco>42.3</preco> +
                                        <quantidade>100</quantidade> +
                                    </negociacao>);
System.out.println(negociacao.getPreco());
```

Obviamente não teremos um XML dentro de um código Java. O exemplo aqui é meramente ilustrativo (útil em um teste!). Os atributos não existentes ficarão como `null` no objeto, como é o caso aqui do atributo `data`, ausente no XML.

O XStream possui uma sobrecarga do método `fromXML` que recebe um `InputStream` como argumento, outro que recebe um `Reader`.

JAXB ou XStream?

A vantagem do JAXB é ser uma especificação do Java, e a do XStream é ser mais flexível e permitir trabalhar com classes imutáveis.

@XStreamAlias

Em vez de chamar `stream.alias("negociacao", Negociacao.class);`, podemos fazer essa configuração direto na classe `Negociacao` com uma anotação:

```
@XStreamAlias(negociacao)
public class Negociacao {
}
```

Para habilitar o suporte a anotações, precisamos chamar no `xstream`:

```
stream.autodetectAnnotations(true);
```

Ou então, se precisarmos processar as anotações de apenas uma única classe, basta indicá-la, como abaixo:

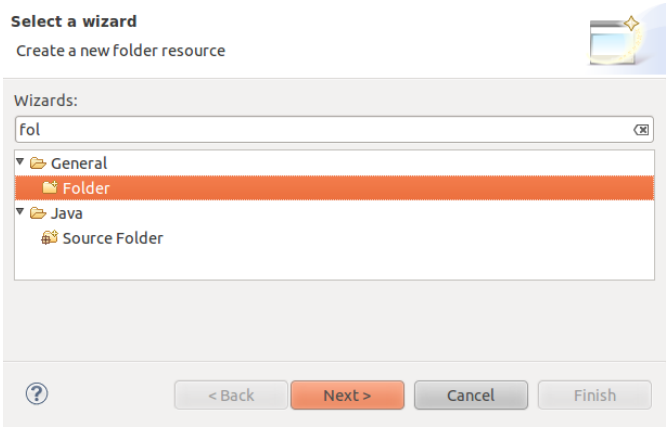
```
stream.processAnnotations(Negociacao.class);
```

Note que trabalhar com as anotações, portanto, não nos economiza linhas de código. Sua principal vantagem é manter as configurações centralizadas e, assim, se houver mais de uma parte na sua aplicação responsável por gerar XMLs de um mesmo modelo, não corremos o risco de ter XMLs incompatíveis.

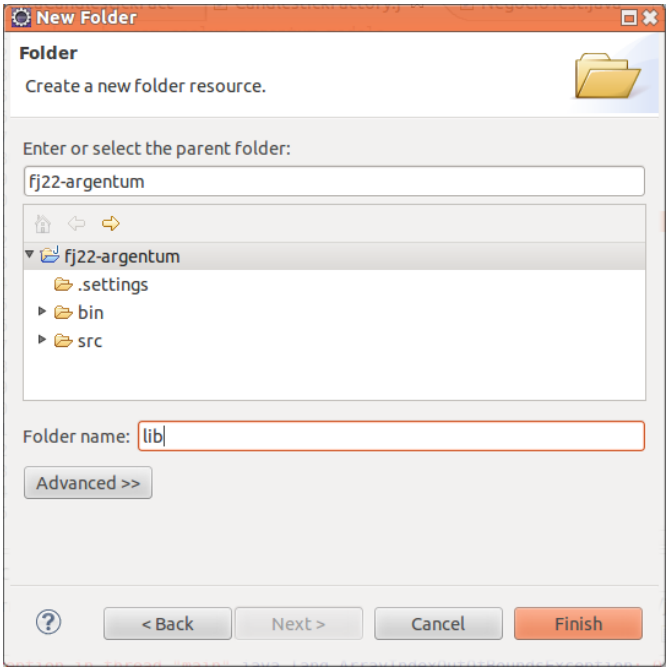
4.5 Exercícios: Lendo o XML

1) Para usarmos o XStream, precisamos copiar seus JARs para o nosso projeto e adicioná-los ao *Build Path*. Para facilitar, vamos criar uma pasta **lib** para colocar todos os JARs que necessitarmos.

   Crie uma nova pasta usando **ctrl + N** e começando a digitar *Folder*:



Coloque o nome de **lib** e clique OK:



2) Vamos pôr o XStream no nosso projeto. Vá na pasta **Caelum** no seu Desktop e entre em **22**. Localize o arquivo do XStream:

Esse é o mesmo arquivo que você encontra para download no site do XStream, na versão minimal.

3) Copie o JAR do XStream 1.4 para a pasta `lib/` do Argentum e, pelo Eclipse, entre na pasta **lib** e dê refresh (F5) nela.

Então, selecione o JAR, clique com o botão direito e vá em **Build Path**, **Add to build path**. A partir de agora o Eclipse considerará as classes do XStream para esse nosso projeto.

4) Vamos, finalmente, implementar a leitura do XML, delegando o trabalho para o XStream. Criamos a classe `LeitorXML` dentro do pacote `br.com.caelum.argentum.reader`:

```
package br.com.caelum.argentum.reader;

// imports...

public class LeitorXML {

    public List<Negociacao> carrega(InputStream inputStream) {
        XStream stream = new XStream(new DomDriver());
        stream.alias(negociacao, Negociacao.class);
        return (List<Negociacao>) stream.fromXML(inputStream);
    }
}
```

5) Crie um teste de unidade `LeitorXMLTest` pelo Eclipse para testarmos a leitura do XML. Com o cursor na classe `LeitorXML`, faça **Ctrl + N** e digite *JUnit Test Case*:

Lembre-se de colocá-lo na *source folder* **src/test/java**.

Para não ter de criar um arquivo XML no sistema de arquivos, podemos usar um truque interessante: coloque o trecho do XML em uma String Java, e passe um `ByteArrayInputStream`, convertendo nossa String para byte através do método `getBytes()`:

```
@Test
public void carregaXmlComUmaNegociacaoEmListaUnitaria() {
    String xmlDeTeste = ...; // o XML vai aqui!

    LeitorXML leitor = new LeitorXML();

    InputStream xml = new ByteArrayInputStream(xmlDeTeste.getBytes());

    List<Negociacao> negociacoes = leitor.carrega(xml);
}
```

Use o seguinte XML de teste, **substituindo a linha em negrito** acima:

```
String xmlDeTeste = <list> +
                        <negociacao> +
                            <preco>43.5</preco> +
                            <quantidade>1000</quantidade> +
                            <data> +
                                <time>1322233344455</time> +
                            </data> +
                        </negociacao> +
                    </list>;
```

6) Um teste de nada serve se não tiver suas verificações. Assim, não esqueça de verificar valores esperados como:

   - a lista devolvida deve ter tamanho 1;
   - a negociação deve ter preço 43.5;
   - a quantidade deve ser 1000.

7) (Opcional) Crie mais alguns testes para casos excepcionais, como:

   - Zero negociações;
   - Preço ou quantidade faltando;
   - Outras quantidades de negociações (3, por exemplo).

8) (importante, conceitual) E o que falta agora? Testar nosso código com um XML real?

   É muito comum sentirmos a vontade de fazer um teste "maior": um teste que realmente abre um `InputStreamReader`, passa o XML para o `LeitorXML` e depois chama a `CandlestickFactory` para quebrar as negociações em candles.

   Esse teste seria um chamado *teste de integração* - não de unidade. Se criássemos esse teste e ele falhasse, seria muito mais difícil detectar o ponto de falha!

   Pensar em sempre testar as menores unidades possíveis nos força a pensar em classes menos dependentes entre si, isto é, com baixo acoplamento. Por exemplo: poderíamos ter criado um `LeitorXML` que internamente chamasse a fábrica e devolvesse diretamente uma `List<Candlestick>`. Mas isso faria com que o nosso teste do leitor de XML testasse muito mais que apenas a leitura de XML (já que estaria passando pela `CandlestickFactory`).

4.6 Discussão em aula: Onde usar XML e o abuso do mesmo

Capítulo 5

Test Driven Design - TDD

"Experiência sem teoria é cegueira, mas teoria sem experiência é mero jogo intelectual."
– Immanuel Kant

5.1 Separando as candles

Agora que temos nosso leitor de XML que cria uma lista com as negociações representadas no arquivo passado, um novo problema surge: a BOVESPA permite fazer download de um arquivo XML contendo **todas** as negociações de um ativo desde a data especificada. Entretanto, nossa `CandlestickFactory` está preparada apenas para *construir candles de uma data específica*.

Dessa forma, precisamos ainda quebrar a lista que contém todas as negociações em partes menores, com negociações de um dia apenas, e usar o outro método para gerar cada `Candlestick`. Essas, devem ser armazenadas em uma nova lista para serem devolvidas.

Para fazer tal lógica, então, precisamos:

- passar por cada negociações da lista original;
- verificar **se continua no mesmo dia** e...
- ...se sim, adiciona na lista do dia;
- ...caso contrário:
  - gera a candle;
  - guarda numa lista de `Candlesticks`;
  - zera a lista de negociações do dia;

  - indica que vai olhar o próximo dia, agora;

- ao final, devolver a lista de candles;

O algoritmo não é trivial e, ainda, ele depende de uma verificação que o Java não nos dá prontamente: *se continua no mesmo dia*. Isto é, dado que eu sei qual a `dataAtual`, quero verificar se a negociação pertence a esse mesmo dia.

Verificar uma negociação é do mesmo dia que um `Calendar` qualquer exige algumas linhas de código, mas veja que, mesmo antes de implementá-lo, já sabemos como o método `isMesmoDia` deverá se comportar em diversas situações:

- se for exatamente o mesmo milissegundo => true;
- se for no mesmo dia, mas em horários diferentes => true;
- se for no mesmo dia, mas em meses diferentes => false;
- se for no mesmo dia e mês, mas em anos diferentes => false.

Sempre que vamos começar a desenvolver uma lógica, intuitivamente, já pensamos em seu comportamento. Fazer os testes automatizados para tais casos é, portanto, apenas colocar nosso pensamento em forma de código. Mas fazê-lo incrementalmente, mesmo antes de seguir com a implementação é o princípio do que chamamos de *Test Driven Design* (TDD).

5.2 Vantagens do TDD

TDD é uma técnica que consiste em pequenas iterações, em que novos casos de testes de funcionalidades desejadas são criados antes mesmo da implementação. Nesse momento, o teste escrito deve falhar, já que a funcionalidade implementada não existe. Então, o código necessário para que os testes passem, deve ser escrito e o teste deve passar. O ciclo se repete para o próximo teste mais simples que ainda não passa.

Um dos principais benefício dessa técnica é que, como os testes são escritos antes da implementação do trecho a ser testado, o programador não é influenciado pelo código já feito - assim, ele tende a escrever testes melhores, pensando no comportamento em vez da implementação.

Lembremos: os testes devem mostrar (e documentar) o comportamento do sistema, e não o que uma implementação faz.

Além disso, nota-se que TDD traz baixo acoplamento, o que é ótimo já que classes muito acopladas são difíceis de testar. Como criaremos os testes antes, desenvolveremos classes menos acopladas, isto é, menos dependentes de outras muitas, separando melhor as responsabilidades.

O TDD também é uma espécie de guia: como o teste é escrito antes, nenhum código do sistema é escrito por "acharmos" que vamos precisar dele. Em sistemas sem testes, é comum encontrarmos centenas de linhas que jamais serão invocadas, simplesmente porque o desenvolvedor "achou" que alguém um dia precisaria daquele determinado método.

Imagine que você já tenha um sistema com muitas classes e nenhum teste: provavelmente, para iniciar a criação de testes, muitas refatorações terão de ser feitas, mas como modificar seu sistema garantindo o funcionamento dele após as mudanças quando não existem testes que garantam que seu sistema tenha o comportamento desejado? Por isso, crie testes sempre e, de preferência, antes da implementação da funcionalidade.

TDD é uma disciplina difícil de se implantar, mas depois que você pega o jeito e o hábito é adquirido, podemos ver claramente as diversas vantagens dessa técnica.

5.3 Exercícios: Identificando negociações do mesmo dia

Poderíamos criar uma classe `LeitorXML` que pega todo o XML e converte em candles, mas ela teria muita responsabilidade. Vamos cuidar da lógica que separa as negociações em vários candles por datas em outro lugar.

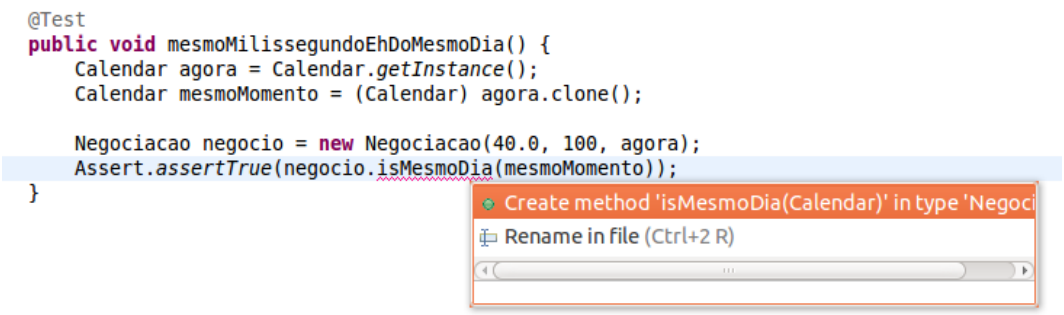
1) Queremos então, em nossa classe de factory, pegar uma série de negociações e transformar em uma lista de candles. Para isso vamos precisar que uma negociação saiba identificar se é do mesmo dia que a `dataAtual`.

   Para saber, conforme percorremos todas as negociações, se a negociação atual ainda aconteceu na mesma data que estamos procurando, vamos usar um método na classe `Negociacao` que faz tal verificação.

   Seguindo os princípios do TDD, começamos escrevendo um teste na classe `NegociacaoTest`:

```
@Test
public void mesmoMilissegundoEhDoMesmoDia() {
    Calendar agora = Calendar.getInstance();
    Calendar mesmoMomento = (Calendar) agora.clone();

    Negociacao negociacao = new Negociacao(40.0, 100, agora);
    Assert.assertTrue(negociacao.isMesmoDia(mesmoMomento));
}
```

   Esse código não vai compilar de imediato, já que não temos esse método na nossa classe. No Eclipse, aperte **Ctrl + 1** em cima do erro e escolha **Create method isMesmoDia**.



   E qual será uma implementação interessante? Que tal simplificar usando o método `equals` de `Calendar`?

```
public boolean isMesmoDia(Calendar outraData) {
    return this.data.equals(outraData);
}
```

Rode o teste! Passa?

2) Nosso teste passou de primeira! Vamos tentar mais algum teste? Vamos testar datas iguais em horas diferentes, crie o método a seguir na classe `NegociacaoTest`:

```
@Test
public void comHorariosDiferentesEhNoMesmoDia() {
    // usando GregorianCalendar(ano, mes, dia, hora, minuto)
    Calendar manha = new GregorianCalendar(2011, 10, 20, 8, 30);
    Calendar tarde = new GregorianCalendar(2011, 10, 20, 15, 30);

    Negociacao negociacao = new Negociacao(40.0, 100, manha);
    Assert.assertTrue(negociacao.isMesmoDia(tarde));
}
```

Rode o teste. Não passa!

Infelizmente, usar o `equals` não resolve nosso problema de comparação.

Lembre que um `Calendar` possui um *timestamp*, isso quer dizer que além do dia, do mês e do ano, há também informações de hora, segundos etc. A implementação que compara os dias será:

```
public boolean isMesmoDia(Calendar outraData) {
    return
        data.get(Calendar.DAY_OF_MONTH) == outraData.get(Calendar.DAY_OF_MONTH);
}
```

Altere o método `isMesmoDia` na classe `Negociacao` e rode os testes anteriores. Passamos agora?

3) O próximo teste a implementarmos será o que garante que para dia igual, mas mês diferente, a data não é a mesma. Quer dizer: não basta comparar o campo referente ao dia do mês, ainda é necessário que seja o mesmo mês!

Crie o `mesmoDiaMasMesesDiferentesNaoSaoDoMesmoDia` na classe de testes do `Negociacao`, veja o teste falhar e, então, implemente o necessário para que ele passe. Note que, dessa vez, o valor esperado é o `false` e, portanto, utilizaremos o `Assert.assertFalse`.

4) Finalmente, o último teste a implementarmos será o que garante que para dia e meses iguais, mas anos diferentes, a data não é a mesma. Siga o mesmo procedimento para desenvolver com TDD:

- Escreva o teste `mesmoDiaEMesMasAnosDiferentesNaoSaoDoMesmoDia`;
- Rode e veja que falhou;
- Implemente o necessário para fazê-lo passar.

Feito esse processo, seu método `isMesmoDia` na classe `Negociacao` deve ter ficado bem parecido com isso:

```
public boolean isMesmoDia(Calendar outraData) {
    return
      this.data.get(Calendar.DAY_OF_MONTH) == outraData.get(Calendar.DAY_OF_MONTH)
      && this.data.get(Calendar.MONTH) == outraData.get(Calendar.MONTH)
      && this.data.get(Calendar.YEAR) == outraData.get(Calendar.YEAR);
}
```

5.4 Exercícios: Separando os candles

1) Próximo passo: dada uma lista de negociações de várias datas diferentes mas ordenada por data, quebrar em uma lista de candles, uma para cada data.

   Seguindo a disciplina do TDD: começamos pelo teste!

   Adicione o método `paraNegociacoesDeTresDiasDistintosGeraTresCandles` na classe `CandlestickFactoryTest`:

```
@Test
public void paraNegociacoesDeTresDiasDistintosGeraTresCandles() {
    Calendar hoje = Calendar.getInstance();

    Negociacao negociacao1 = new Negociacao(40.5, 100, hoje);
    Negociacao negociacao2 = new Negociacao(45.0, 100, hoje);
    Negociacao negociacao3 = new Negociacao(39.8, 100, hoje);
    Negociacao negociacao4 = new Negociacao(42.3, 100, hoje);

    Calendar amanha = (Calendar) hoje.clone();
    amanha.add(Calendar.DAY_OF_MONTH, 1);

    Negociacao negociacao5 = new Negociacao(48.8, 100, amanha);
    Negociacao negociacao6 = new Negociacao(49.3, 100, amanha);

    Calendar depois = (Calendar) amanha.clone();
    depois.add(Calendar.DAY_OF_MONTH, 1);

    Negociacao negociacao7 = new Negociacao(51.8, 100, depois);
    Negociacao negociacao8 = new Negociacao(52.3, 100, depois);

    List<Negociacao> negociacoes = Arrays.asList(negociacao1, negociacao2,
        negociacao3, negociacao4, negociacao5, negociacao6, negociacao7,
        negociacao8);

    CandlestickFactory fabrica = new CandlestickFactory();
```

```
    List<Candlestick> candles = fabrica.constroiCandles(negociacoes);

    Assert.assertEquals(3, candles.size());
    Assert.assertEquals(40.5, candles.get(0).getAbertura(), 0.00001);
    Assert.assertEquals(42.3, candles.get(0).getFechamento(), 0.00001);
    Assert.assertEquals(48.8, candles.get(1).getAbertura(), 0.00001);
    Assert.assertEquals(49.3, candles.get(1).getFechamento(), 0.00001);
    Assert.assertEquals(51.8, candles.get(2).getAbertura(), 0.00001);
    Assert.assertEquals(52.3, candles.get(2).getFechamento(), 0.00001);
}
```

A chamada ao método `constroiCandles` não compila pois o método não existe ainda. **Ctrl + 1** e **Create method**.

Como implementamos? Precisamos:

- Criar a List<Candlestick>;
- Percorrer a List<Negociacao> adicionando cada negociacão no Candlestick atual;
- Quando achar uma negociação de um novo dia, cria um Candlestick novo e adiciona;
- Devolve a lista de candles;

O código talvez fique um pouco grande. Ainda bem que temos nosso teste!

```
public List<Candlestick> constroiCandles(List<Negociacao> todasNegociacoes) {
    List<Candlestick> candles = new ArrayList<Candlestick>();

    List<Negociacao> negociacoesDoDia = new ArrayList<Negociacao>();
    Calendar dataAtual = todasNegociacoes.get(0).getData();

    for (Negociacao negociacao : todasNegociacoes) {
        // se não for mesmo dia, fecha candle e reinicia variáveis
        if (!negociacao.isMesmoDia(dataAtual)) {
            Candlestick candleDoDia = constroiCandleParaData(dataAtual,
                                                    negociacoesDoDia);
            candles.add(candleDoDia);
            negociacoesDoDia = new ArrayList<Negociacao>();
            dataAtual = negociacao.getData();
        }
        negociacoesDoDia.add(negociacao);
    }
    // adiciona último candle
    Candlestick candleDoDia = constroiCandleParaData(dataAtual,
                negociacoesDoDia);
    candles.add(candleDoDia);
```

```
    return candles;
}
```

Rode o teste!

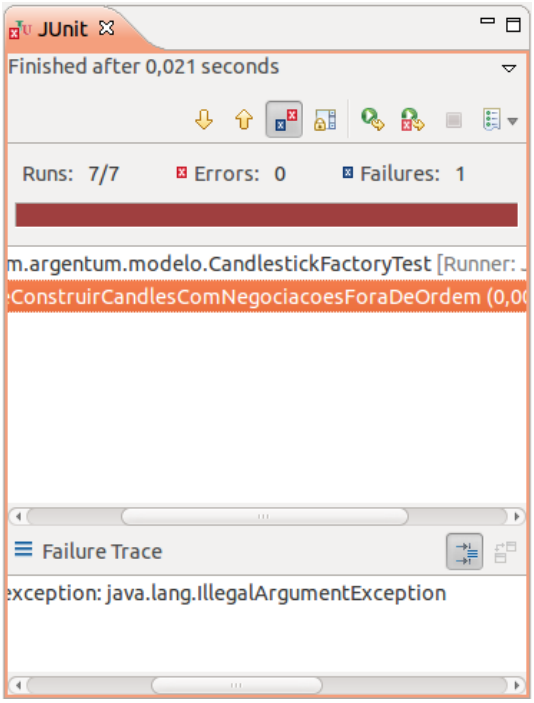
5.5 Exercícios opcionais

1) E se passarmos para o método `constroiCandles` da fábrica uma lista de negociações que não está na ordem crescente? O resultado vai ser candles em ordem diferentes, e provavelmente com valores errados. Apesar da especificação dizer que os negociações vem ordenados pela data, é boa prática programar defensivamente em relação aos parâmetros recebidos.

   Aqui temos diversas opções. Uma delas é, caso alguma `Negociacao` venha em ordem diferente da crescente, lançamos uma exception, a `IllegalStateException`.

   Crie o `naoPermiteConstruirCandlesComNegociacoesForaDeOrdem` e configure o teste para verificar que uma `IllegalStateException` foi lançada. Basta usar como base o mesmo teste que tínhamos antes, mas adicionar as negociações com datas não crescentes.

   Rode o teste e o veja falhar.



   Pra isso, modificamos o código adicionando as linhas em negrito ao método `constroiCandles`:

```
for (Negociacao negociacao : todasNegociacoes) {
    if (negociacao.getData().before(dataAtual)) {
```

```
            throw new IllegalStateException(negociações em ordem errada);
        }
        // se não for mesmo dia, fecha candle e reinicia variáveis
        ...
```

2) Vamos criar um gerador automático de arquivos para testes da bolsa. Ele vai gerar 30 dias de candle e cada candle pode ser composto de 0 a 19 negociações. Esses preços podem variar.

```
public class GeradorAleatorioDeXML {
    public static void main(String[] args) throws IOException {
        Calendar data = Calendar.getInstance();
        Random random = new Random(123);
        List<Negociacao> negociacoes = new ArrayList<Negociacao>();

        double valor = 40;
        int quantidade = 1000;

        for (int dias = 0; dias < 30; dias++) {
            int quantidadeNegociacoesDoDia = random.nextInt(20);

            for (int negociacao = 0; negociacao < quantidadeNegociacoesDoDia;
                negociacao++){

                // no máximo sobe ou cai R$1,00 e nao baixa além de R$5,00
                valor += (random.nextInt(200)  100) / 100.0;
                if (valor < 5.0) {
                    valor = 5.0;
                }

                // quantidade: entre 500 e 1500
                quantidade += 1000  random.nextInt(500);

                Negociacao n = new Negociacao(valor, quantidade, data);
                negociacoes.add(n);
            }
            data = (Calendar) data.clone();
            data.add(Calendar.DAY_OF_YEAR, 1);
        }

        XStream stream = new XStream(new DomDriver());
        stream.alias(negociacao, Negociacao.class);
        stream.setMode(XStream.NO_REFERENCES);

        PrintStream out = new PrintStream(new File(negociacao.xml));
        out.println(stream.toXML(negociacoes));
    }
```

```
}
```

Se você olhar o resultado do XML, verá que, por usarmos o mesmo objeto Calendar em vários lugares, o XStream coloca referências no próprio XML evitando a cópia do mesmo dado. Mas talvez isso não seja tão interessante na prática, pois é mais comum na hora de integrar sistemas, passar um XML simples com todos os dados.

A opção `XStream.NO_REFERENCES` serve para indicar ao XStream que não queremos que ele crie *referências* a tags que já foram serializadas iguaizinhas. Você pode passar esse argumento para o método `setMode` do XStream. Faça o teste sem e com essa opção para entender a diferença.

Desafio - Ordene a lista on demand

1) Faça com que uma lista de `Negociacao` seja ordenável pela data das negociações.

   Então poderemos, logo no início do método, ordenar todas as negociações com `Collections.sort` e não precisamos mais verificar se os negociações estão vindo em ordem crescente!

   Perceba que mudamos uma regra de negócio, então teremos de refletir isso no nosso teste unitário que estava com `expected=IllegalStateException.class` no caso de vir em ordem errada. O resultado agora com essa modificação tem de dar o mesmo que com as datas crescentes.

Capítulo 6

Acessando um Web Service

"Nenhum homem é uma ilha isolada; cada homem é uma partícula do continente, uma parte da terra"
– John Donne

6.1 Integração entre sistemas

No capítulo anterior resolvemos o problema de como interpretar os dados oriundos de um arquivo XML, apesar disso, no mundo real, dados são gerados dinamicamente a todo instante das mais diversas fontes.

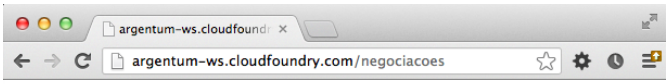
No mercado de bolsa de valores é comum o uso de aplicações que permitem aos seus usuários analisar o mercado, e até mesmo comprar e vender ações em tempo real. Mas como é possível analisar o mercado se não temos acesso aos dados da Bovespa?

A integração e a comunicação com o sistema da Bovespa se faz necessária para que possamos receber dados sempre atualizados. Geralmente essa comunicação se dá pelo próprio protocolo da web, o HTTP, com o formato difundido e já estudado XML. Essa integração e comunicação entre aplicações possui o nome de **Web Service**.

6.2 Consumindo dados de um Web Service

Para consumir dados vindos de outra aplicação, a primeira coisa importante é saber onde essa aplicação se encontra. Em termos técnicos, qual a URL desse **web service**. Em nosso projeto a URL específica da aplicação será **http://argentumws.caelum.com.br/negociacoes**.

Podemos testar essa URL facilmente dentro do navegador, basta copiar e colar na barra de endereço. Ao executar, o navegador recebe como resposta o XML de negócios que já conhecemos, por exemplo:




```
<list>
  <negociacao>
    <preco>250.64</preco>
    <quantidade>14</quantidade>
    <data>
      <time>1357516800000</time>
      <timezone>Etc/UTC</timezone>
    </data>
  </negociacao>
  <negociacao>
    <preco>415.37</preco>
    <quantidade>24</quantidade>
    <data>
      <time>1357516800000</time>
      <timezone>Etc/UTC</timezone>
    </data>
  </negociacao>
  <negociacao>
    <preco>336.48</preco>
    <quantidade>19</quantidade>
    <data>
      <time>1357516800000</time>
      <timezone>Etc/UTC</timezone>
    </data>
  </negociacao>
  <negociacao>
```

6.3 Criando o cliente Java

Já sabemos de onde consumir, resta saber como consumir esses dados pela web. No mundo web trabalhamos com o conceito de requisição e resposta. Se queremos os dados precisamos realizar uma requisição para aquela URL, mas como?

Na própria *API* do Java temos classes que tornam possível essa tarefa. Como é o caso da classe URL que nos permite referenciar um recurso na Web, seja ele um arquivo ou até mesmo um diretório.

```
URL url = new URL(http://argentumws.caelum.com.br/negociacoes);
```

Conhecendo a URL falta agora que uma requisição HTTP seja feita para ela. Faremos isso através do método `openConnection` que nos devolve um `URLConnection`. Entretanto, como uma requisição HTTP se faz necessária, usaremos uma subclasse de `URLConnection` que é a `HttpURLConnection`.

```
URL url = new URL(http://argentumws.caelum.com.br/negociacoes);
HttpURLConnection connection = (HttpURLConnection) url.openConnection();
```

Dessa conexão pediremos um `InputStream` que será usado pelo nosso `LeitorXML`.

```
URL url = new URL(http://argentumws.caelum.com.br/negociacoes);
HttpURLConnection connection = (HttpURLConnection) url.openConnection();
InputStream content = connection.getInputStream();
List<Negociacao> negociacoes = new LeitorXML().carrega(content);
```

Dessa forma já temos em mãos a lista de negociações que era o nosso objetivo principal. Resta agora encapsularmos este código em alguma classe, para que não seja necessário repeti-lo toda vez que precisamos receber os dados da negociação. Vamos implementar a classe `ClienteWebService` e nela deixar explícito qual o caminho da aplicação que a conexão será feita.

```
public class ClienteWebService {

    private static final String URL_WEBSERVICE  =
        http://argentumws.caelum.com.br/negociacoes;
}
```

Por último, e não menos importante, declararemos um método que retorna uma lista de negociações, justamente o que usaremos no projeto.

```
public class ClienteWebService {

    private static final String URL_WEBSERVICE  =
        http://argentumws.caelum.com.br/negociacoes;

    public List<Negociacao> getNegociacoes() {

        URL url = new URL(URL_WEBSERVICE);
        HttpURLConnection connection = (HttpURLConnection)url.openConnection();
        InputStream content = connection.getInputStream();
        return new LeitorXML().carrega(content);
    }
}
```

Nossa classe ainda não compila, pois tanto o construtor de `URL` quanto os métodos de `HttpURLConnection` lançam exceções que são do tipo `IOException`. Vamos tratar o erro e fechar a conexão que foi aberta pelo método `getInputStream`, em um bloco `finally`.

```
...
public List<Negociacao> getNegociacoes() {

    HttpURLConnection connection = null;

    try {
            URL url = new URL(URL_WEBSERVICE);
            connection = (HttpURLConnection)url.openConnection();
            InputStream content = connection.getInputStream();
            return new LeitorXML().carrega(content);
    } catch (IOException e) {
        throw new RuntimeException(e);
```

```
    } finally {
        connection.disconnect();
    }
}
...
```

Dessa forma conseguimos nos comunicar com um **Web Service** e consumir os dados disponibilizados por ele através de um XML. Esta é uma prática bastante utilizada pelo mercado e estudada com mais aprofundamento no curso **FJ-31 | Curso Java EE avançado e Web Services**.

HttpClient

Existe uma biblioteca capaz de lidar com dados de uma forma simples e mais específica do que utilizar diretamente a API do Java. Para trabalhar com o protocolo HTTP há a biblioteca **HttpClient** que faz parte do Apache Software Foundation:

http://hc.apache.org/httpcomponents-client-ga/index.html

Com ela ganhamos uma API que fornece toda funcionalidade do protocolo HTTP e poderíamos usá-la para chamar o Web Service. Segue um pequeno exemplo usando o HttpClient para executar uma requisição do tipo *GET*:

```
HttpClient client = new DefaultHttpClient();
HttpGet request = new HttpGet(URL_DO_WEBSERVICE);
HttpResponse response = client.execute(request);
InputStream content = response.getEntity().getContent();
```

Web Service - SOAP, JSON e outros

Por definição um Web Service é alguma lógica de negócio acessível usando padrões da Internet. O mais comum é usar HTTP como protocolo de comunicação e XML para o formato que apresenta os dados - justamente o que praticaremos aqui. Mas nada impede o uso de outros formatos.

Uma tentativa de especificar mais ainda o XML dos Web Services são os padrões SOAP e WSDL. Junto com o protocolo HTTP, eles definem a base para comunicação de vários serviços no mundo de aplicações *Enterprise*. O SOAP e WSDL tentam esconder toda comunicação e geração do XML, facilitando assim o uso para quem não conhece os padrões Web. No treinamento FJ-31 veremos os detalhes sobre publicação e a criação de clientes baseados no Web Services SOAP/WSDL.

Outro formato bastante popular nos Web Services é o JSON. JSON é parecido com XML, mas um pouco menos verboso e fácil de usar com JavaScript. JSON ganhou popularidade através das requisições AJAX e conquistou o seu espaço nos Web Services também. Além de ser bastante difundido no desenvolvimento mobile por ser mais leve no tráfego via rede.

6.4 Exercícios: Nosso cliente Web Service

1) Vamos agora implementar o cliente do Web Service, primeiramente criaremos a classe `ClienteWebService`, dentro do pacote `br.com.caelum.argentum.ws` na pasta `src/main/java`. Vamos criar também uma constante com a URL para onde será feita a requisição.

```
public class ClienteWebService {

    private static final String URL_WEBSERVICE =
        http://argentumws.caelum.com.br/negociacoes;
}
```

2) Agora vamos criar o método `getNegociacoes()`, que retorna a nossa lista de negociações:

```
public class ClienteWebService {

    private static final String URL_WEBSERVICE =
        http://argentumws.caelum.com.br/negociacoes;

    public List<Negociacao> getNegociacoes() {

        HttpURLConnection connection = null;

        URL url = new URL(URL_WEBSERVICE);
```

```
            connection = (HttpURLConnection)url.openConnection();

            InputStream content = connection.getInputStream();

            return new LeitorXML().carrega(content);
        }
    }
```

3) Não podemos esquecer de colocar o `try/catch` para tratar possíveis erros e logo em seguida fechar a conexão:

```
public class ClienteWebService {

    private static final String URL_WEBSERVICE =
        http://argentumws.caelum.com.br/negociacoes;

    public List<Negociacao> getNegociacoes() {

        HttpURLConnection connection = null;

        try {
            URL url = new URL(URL_WEBSERVICE);

            connection = (HttpURLConnection)url.openConnection();

            InputStream content = connection.getInputStream();

            return new LeitorXML().carrega(content);

        } catch (IOException e) {
            throw new RuntimeException(e);
        } finally {
            connection.disconnect();
        }
    }
}
```

6.5 Discussão em aula: Como testar o cliente do web service?

Capítulo 7

Introdução ao JSF e Primefaces

"Eu não temo computadores, eu temo é a falta deles"
– Isaac Asimov

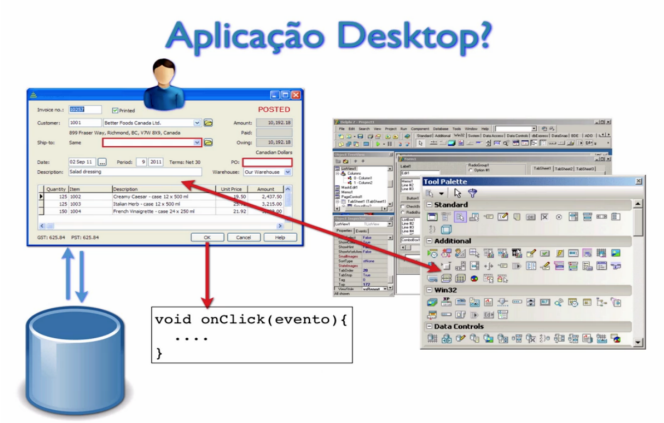
Durante muitos anos, os usuários se habituaram com aplicações Desktop. Este tipo de aplicação é instalada no computador local e acessa diretamente um banco de dados ou gerenciador de arquivos. As tecnologias típicas para criar uma aplicação Desktop são Delphi, VB (Visual Basic) ou, no mundo Java, Swing.

Para o desenvolvedor, a aplicação Desktop é construída com uma série de componentes que a plataforma de desenvolvimento oferece para cada sistema operacional. Esses componentes ricos e muitas vezes sofisticados estão associados a eventos ou procedimentos que executam lógicas de negócio.

Problemas de validação de dados são indicados na própria tela sem que qualquer informação do formulário seja perdida. De uma forma natural, esses componentes lembram-se dos dados do usuário, inclusive entre telas e ações diferentes.

Nesse tipo de desenvolvimento são utilizados diversos **componentes ricos**, como por exemplo, calendários, menus diversos ou componentes *drag and drop* (arrastar e soltar). Eles ficam associados a eventos, ou ações, e guardam automaticamente seu estado, já que mantêm os valores digitados pelo usuário.

Esses componentes não estão, contudo, associados exclusivamente ao desenvolvimento de aplicações Desktop. Podemos criar a mesma sensação confortável para o cliente em uma aplicação web, também usando componentes ricos e reaproveitáveis.



7.1 Desenvolvimento desktop ou web?

Existem algumas desvantagens no desenvolvimento desktop. Como cada usuário tem uma cópia integral da aplicação, qualquer alteração precisaria ser propagada para todas as outras máquinas. Estamos usando um *cliente gordo*, isto é, com muita responsabilidade no lado do cliente.

Note que, aqui, estamos chamando de **cliente** a aplicação que está rodando na máquina do usuário.

Para piorar, as regras de negócio rodam no computador do usuário. Isso faz com que seja muito mais difícil depurar a aplicação, já que não costumamos ter acesso tão fácil à maquina onde a aplicação está instalada. Em geral, enfrentamos **problemas de manutenção e gerenciabilidade**.

O desenvolvimento Web e o protocolo HTTP

Para resolver problemas como esse, surgiram as aplicações baseadas na web. Nessa abordagem há um servidor central onde a aplicação é executada e processada e todos os usuários podem acessá-la através de um cliente simples e do protocolo HTTP.

Um navegador web, como Firefox ou Chrome, que fará o papel da aplicação cliente, interpretando HTML, CSS e JavaScript -- que são as tecnologias que ele entende.

Enquanto o usuário usa o sistema, o navegador envia requisições (*requests*) para o lado do servidor (*server side*), que responde para o computador do cliente (*client side*). Em nenhum momento a aplicação está salva no cliente: todas as regras da aplicação estão no lado do servidor. Por isso, essa abordagem também foi chamada de **cliente magro** (*thin client*).

Isso facilita bastante a manutenção e a gerenciabilidade, pois temos um lugar central e acessível onde a aplicação é executada. Contudo, note que será preciso conhecer HTML, CSS e JavaScript, para fazer a interface com o usuário, e o protocolo **HTTP** para entender a comunicação pela web. E, mais importante ainda, não há mais eventos, mas sim um modelo bem diferente **orientado a requisições e respostas**. Toda essa base precisará ser conhecida pelo desenvolvedor.

Comparando as duas abordagens, podemos ver vantagens e desvantagens em ambas. No lado da aplicação puramente Desktop, temos um estilo de desenvolvimento orientado a eventos, usando componentes ricos, porém com problemas de manutenção e gerenciamento. Do outro lado, as aplicações web são mais fáceis de gerenciar e manter, mas precisamos lidar com HTML, conhecer o protocolo HTTP e seguir o modelo requisição/resposta.

Mesclando desenvolvimento Desktop e Web

Em vez de desenvolver puramente para desktop, é uma tendência mesclar os dois estilos, aproveitando as vantagens de cada um. Seria um desenvolvimento Desktop para a web, tanto central quanto com componentes ricos, aproveitando o melhor dos dois mundos e abstraindo o protocolo de comunicação. Essa é justamente a ideia dos **frameworks web baseados em componentes**.

No mundo Java há algumas opções como **JavaServer Faces** (JSF), Apache Wicket, Vaadin, Tapestry ou GWT da Google. Todos eles são *frameworks* web baseados em componentes.

7.2 Características do JSF

JSF é uma tecnologia que nos permite criar aplicações Java para Web utilizando componentes visuais prés-prontos, de forma que o desenvolvedor não se preocupe com Javascript e HTML. Basta adicionarmos os componentes (calendários, tabelas, formulários) e eles serão renderizados e exibidos em formato html.

Guarda o estado dos componentes

Além disso o estado dos componentes é sempre guardado automaticamente (como veremos mais à frente), criando a característica Stateful. Isso nos permite, por exemplo, criar formulários de várias páginas e navegar nos vários passos dele com o estado das telas sendo mantidos.

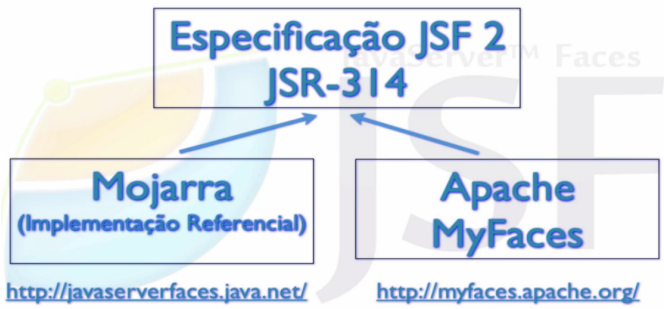
Separa as camadas

Outra característica marcante na arquitetura do JSF é a separação que fazemos entre as camadas de apresentação e de aplicação. Pensando no modelo MVC, o JSF possui uma camada de visualização bem separada do conjunto de classes de modelo.

Especificação: várias implementações

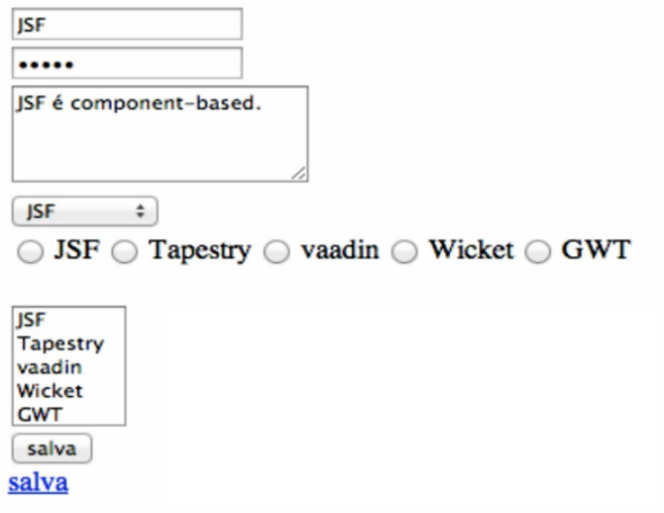
O JSF ainda tem a vantagem de ser uma especificação do Java EE, isto é, todo servidor de aplicações Java tem que vir com uma implementação dela e há diversas outras disponíveis.

A implementação mais famosa do JSF e também a implementação de referência, é a Oracle Mojarra disponível em http://javaserverfaces.java.net/. Outra implementação famosa é a MyFaces da *Apache Software Foundation* em http://myfaces.apache.org/.



Primeiros passos com JSF

Nosso projeto utilizará a implementação Mojarra do JSF. Ela já define o modelo de desenvolvimento e oferece alguns componentes bem básicos. Nada além de `inputs`, botões e `ComboBoxes` simples.

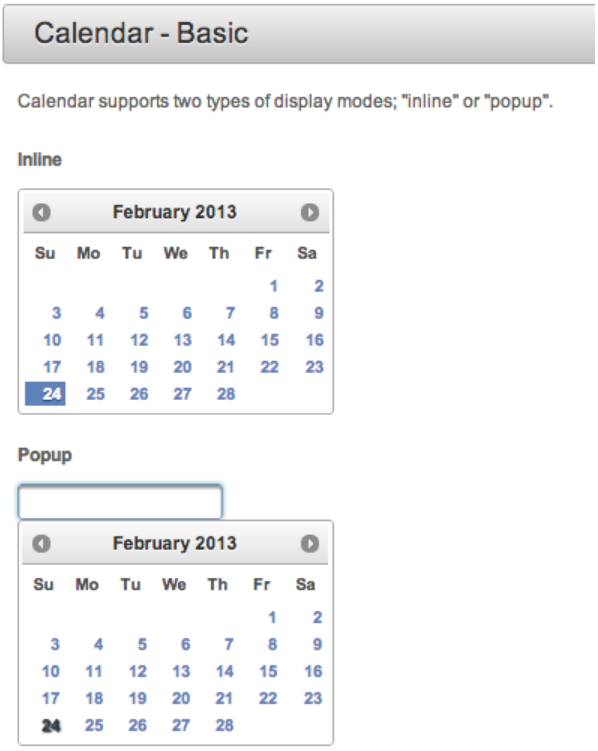


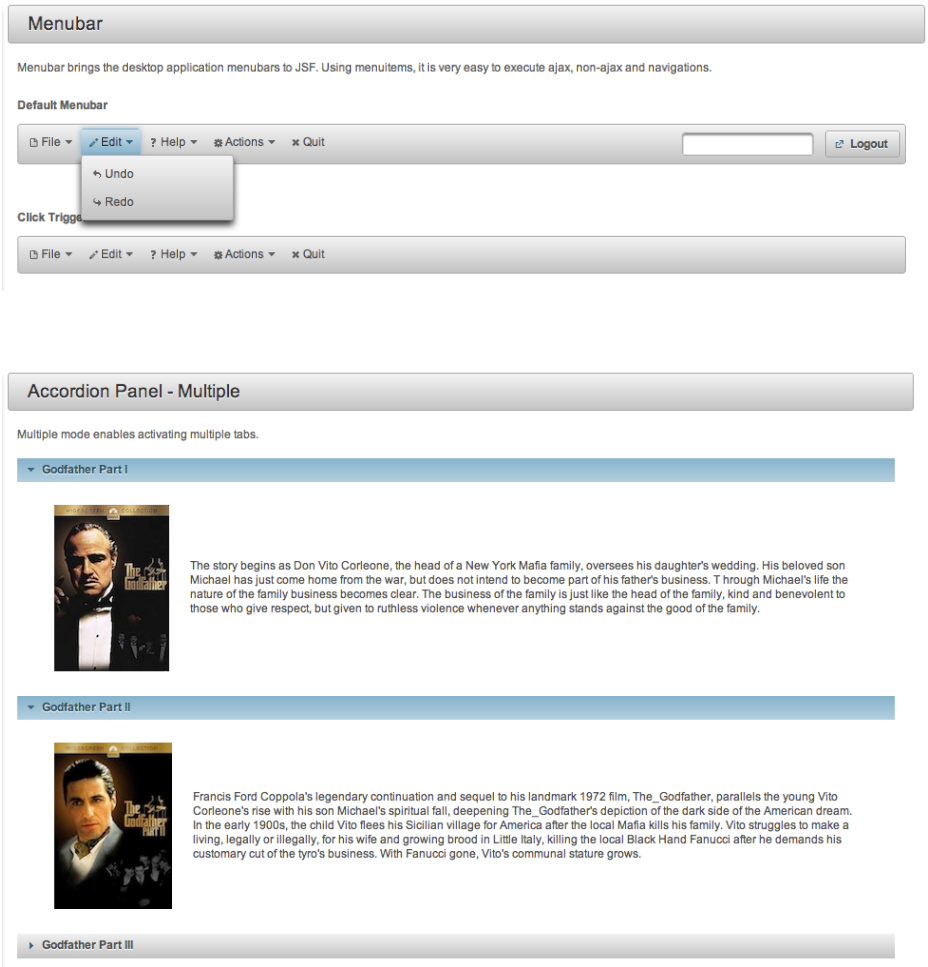
Não há componentes sofisticados dentro da especificação e isso é proposital: uma especificação tem que ser estável e as possibilidades das interfaces com o usuário crescem muito rapidamente. A especificação trata do que é fundamental, mas outros projetos suprem o que falta.

Para atender a demanda dos desenvolvedores por componentes mais sofisticados, há várias extensões do JSF que seguem o mesmo ciclo e modelo da especificação. Exemplos dessas bibliotecas são **PrimeFaces**, **RichFaces** e **IceFaces**. Todas elas definem componentes JSF que vão muito além da especificação.

Cada biblioteca oferece *ShowCases* na web para mostrar seus componentes e suas funcionalidades. Você pode ver o *showcase* do **PrimeFaces** no endereço http://www.primefaces.org.

Na sua *demo online*, podemos ver uma lista de componentes disponíveis, como inputs, painéis, botões diversos, menus, gráficos e componentes *drag & drop*, que vão muito além das especificações, ainda mantendo a facilidade de uso:





Para a definição da interface do projeto *Argentum* usaremos **Oracle Mojarra** com **PrimeFaces**, uma combinação muito comum no mercado.

Preparação do ambiente

Nossa aplicação *Argentum* precisa de uma interface web. Para isso vamos preparar uma aplicação web comum que roda dentro de um *Servlet Container*. Qualquer implementação de servlet container seria válida e, no curso, usaremos o *Apache Tomcat 7*. Uma outra boa opção seria o *Jetty*.

Configuração do controlador do JSF

O JSF segue o padrão arquitetural MVC (*Model-View-Controller*) e faz o papel do *Controller* da aplicação. Para começar a usá-lo, é preciso configurar a servlet do JSF no `web.xml` da aplicação. Esse `Servlet` é responsável por receber as requisições e delegá-las ao JSF. Para configurá-lo basta adicionar as seguintes configurações no `web.xml`:

```
<servlet>
    <servletname>FacesServlet</servletname>
```

```
    <servletclass>javax.faces.webapp.FacesServlet</servletclass>
    <loadonstartup>1</loadonstartup>
</servlet>
<servletmapping>
    <servletname>FacesServlet</servletname>
    <urlpattern>*.xhtml</urlpattern>
</servletmapping>
```

Ao usar o Eclipse com suporte a JSF 2 essa configuração no web.xml já é feita automaticamente durante a criação de um projeto.

Faces-config: o arquivo de configuração do mundo JSF

Além disso, há um segundo XML que é o arquivo de configuração relacionado com o mundo JSF, o `faces-config.xml`.

Como o JSF na versão dois encoraja o uso de anotações em vez de configurações no XML, este arquivo torna-se pouco usado. Ele era muito mais importante na primeira versão do JSF. Neste treinamento, deixaremos ele vazio:

```
<facesconfig
     xmlns=http://java.sun.com/xml/ns/javaee
     xmlns:xsi=http://www.w3.org/2001/XMLSchemainstance
     xsi:schemaLocation=http://java.sun.com/xml/ns/javaee
     http://java.sun.com/xml/ns/javaee/webfacesconfig_2_0.xsd
     version=2.0>
</facesconfig>
```

Agora já temos as informações necessárias para criar nosso primeiro projeto utilizando JSF.

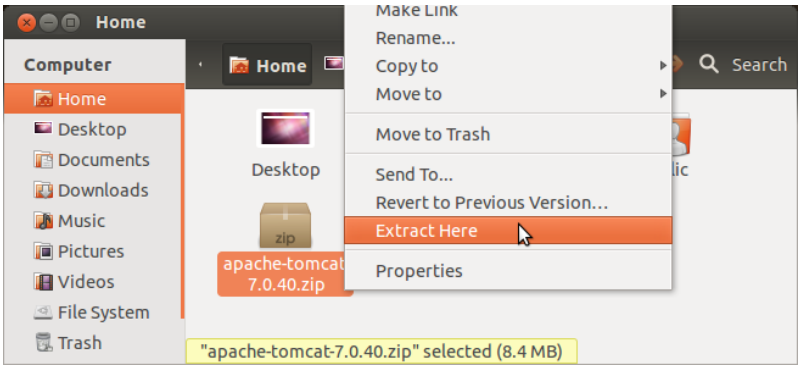
7.3 Exercícios: Instalando o Tomcat e criando o projeto

1) Primeiramente, precisamos instalar o Tomcat. Usaremos a versão 7.x:

   a) Vá no Desktop e entre na pasta *Caelum* e em seguida na pasta 22.

   b) Copie o arquivo zip do TomCat e cole ele na sua pasta Home.

   c) Clique com o botão direito e escolha *Extract here*.



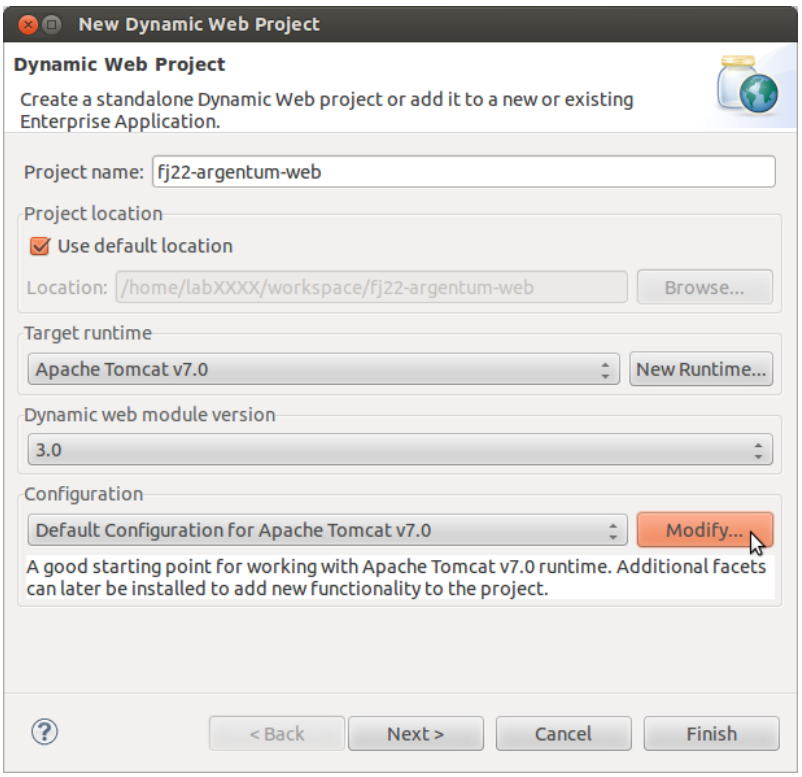
d) O *Tomcat* já está pronto para o uso!

2) O próximo passo é criar o nosso projeto no Eclipse.

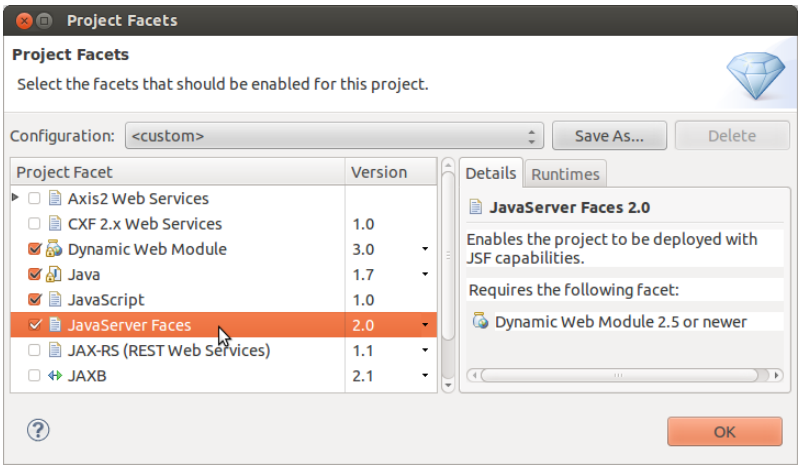
a) Crie um novo projeto web usando o **ctrl + 3** *Dynamic Web Project*.

b) Em *Project name* coloque `fj22-argentum-web`.

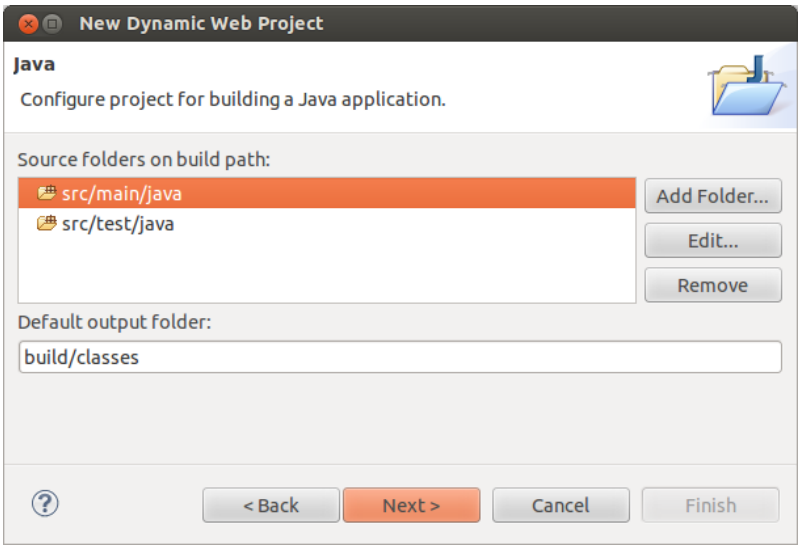
c) Na seção *Configuration* clique em *Modify* para acrescentarmos suporte ao JSF.



d) Na tela que abre, marque o checkbox com **JavaServer Faces 2.0** e clique dê ok:

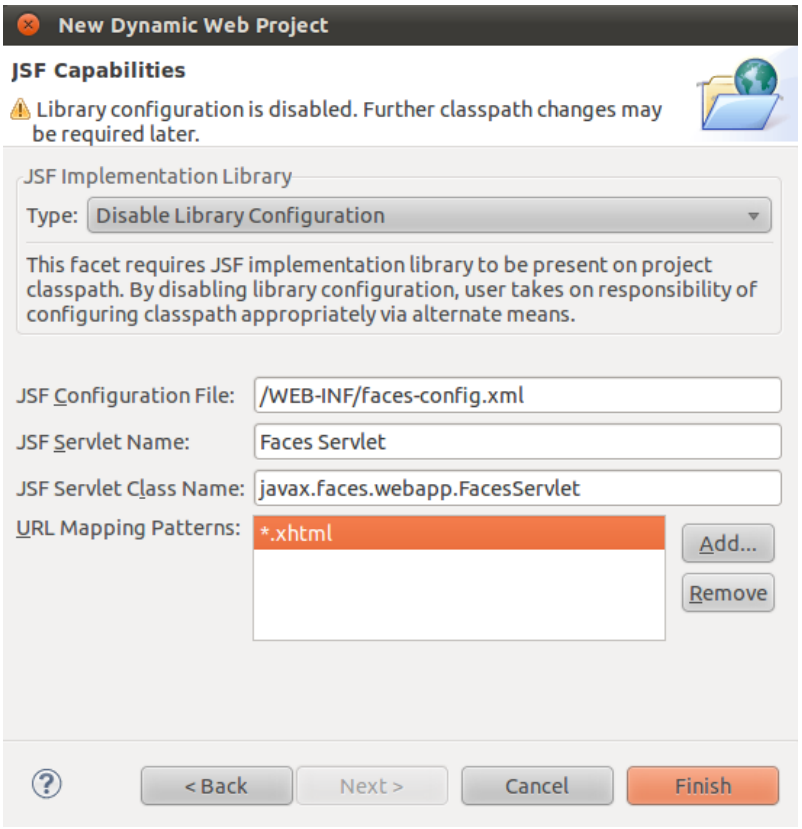


e) De volta à tela de criação do projeto, clique em **Next**. Nessa tela, faremos como no início do curso: removeremos a *source folder* padrão (`src`) e adicione as source folders `src/main/java` e `src/test/java`.



f) Dê *Next* mais duas vezes até chegar à tela de **JSF Capabilities**. Nessa tela, escolha a opção **Disable Library Configuration** para indicarmos para o Eclipse que nós mesmos copiaremos os JARs do JSF.

Ainda nessa tela, na parte *URL Mapping Patterns*, **remova** o mapeamento */faces/** e **adicione** um novo mapeamento como ***.xhtml**

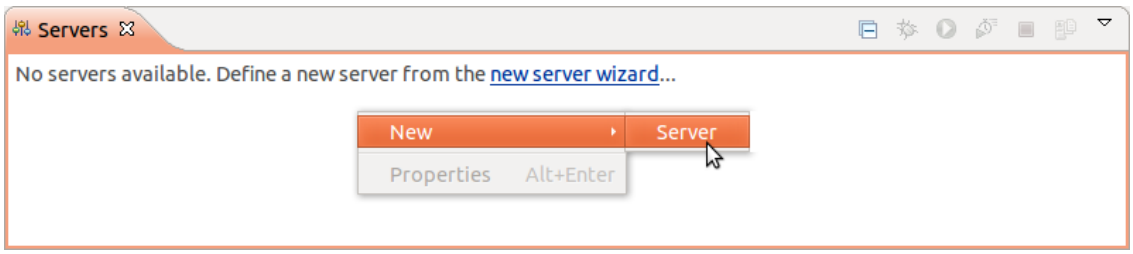


g) Clique em *Finish* e o projeto está criado.

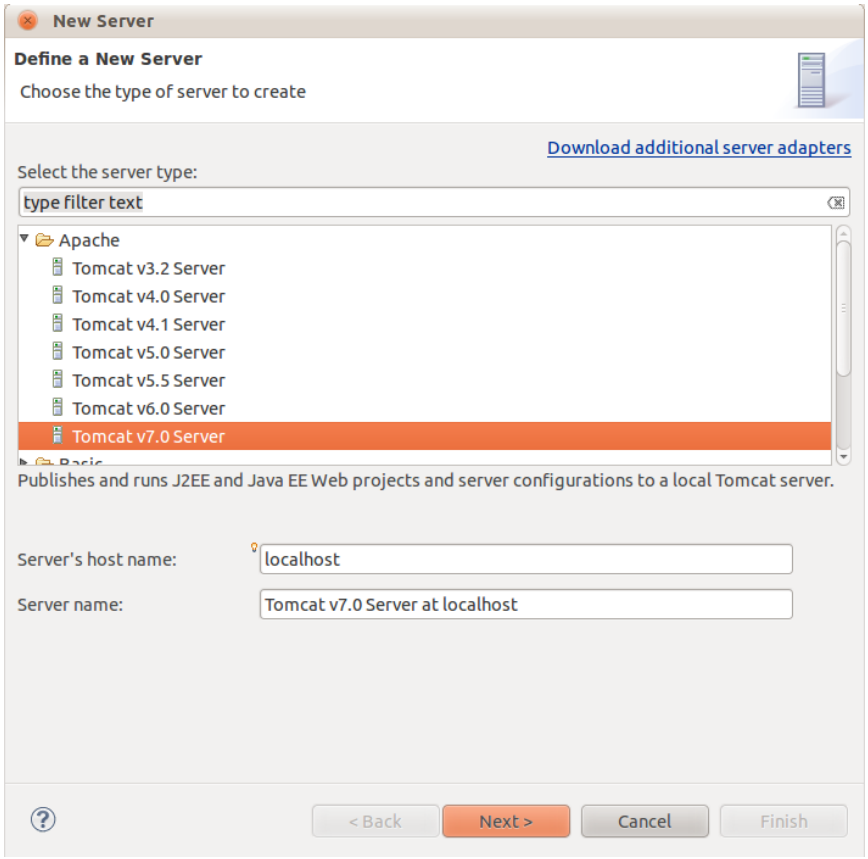
3) O próximo passo é configurar o Tomcat no Eclipse, para que possamos controlá-lo mais facilmente.

a) Dentro do Eclipse, abra a view *Servers*. Para isso, pressione **ctrl + 3**, digite *Servers* e escolha a view. Ela será aberta na parte inferior do seu Eclipse.

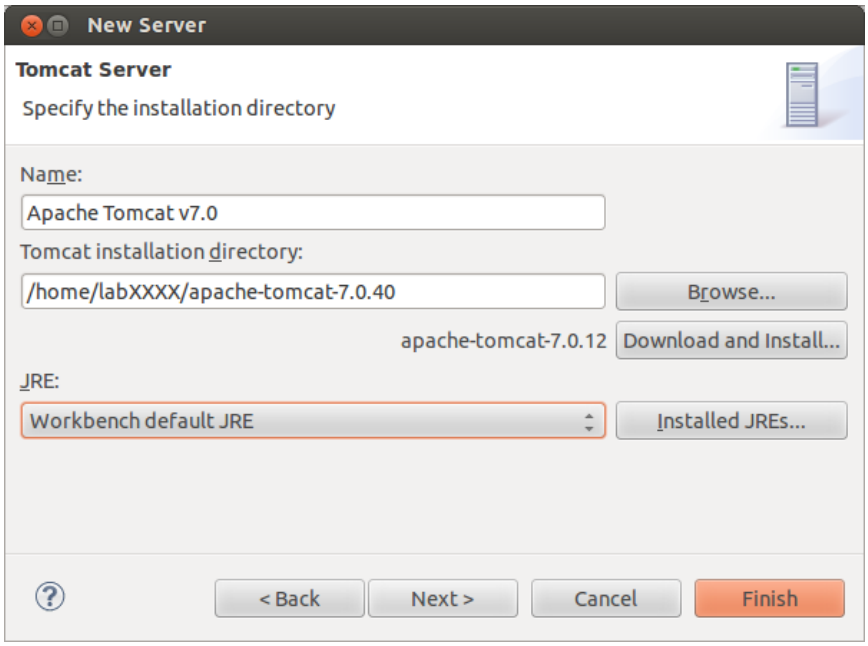
b) Dentro da aba *Servers* clique com o botão direito do mouse e escolha *New -> Server*. Se não quiser usar o mouse, você pode fazer **ctrl+3** *New server*.



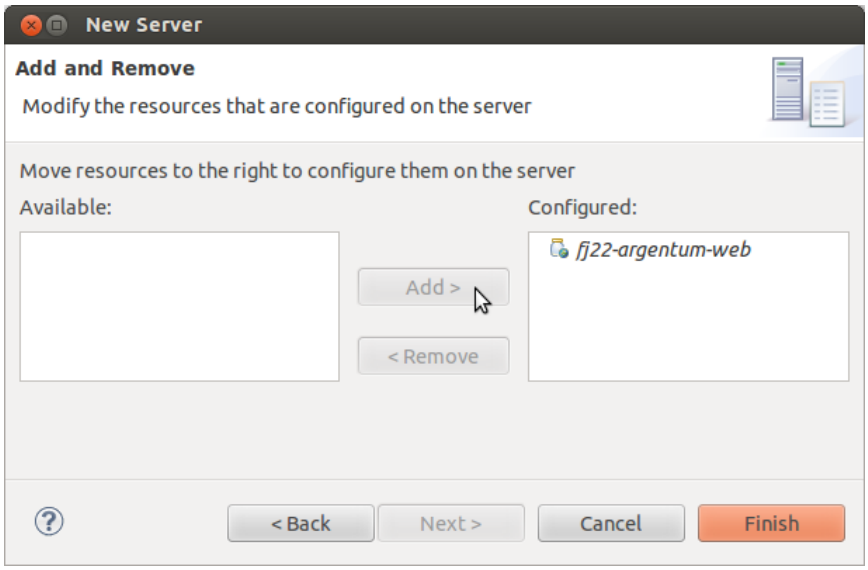
c) Dentro da Janela *New Server* escolha *Apache Tomcat v7.0 Server* e clique em *Next*.



d) O próximo passo é dizermos ao Eclipse em qual diretório instalamos o Tomcat. Clique no botão *Browse...* e escolha a pasta na qual você descompactou o *Tomcat*.



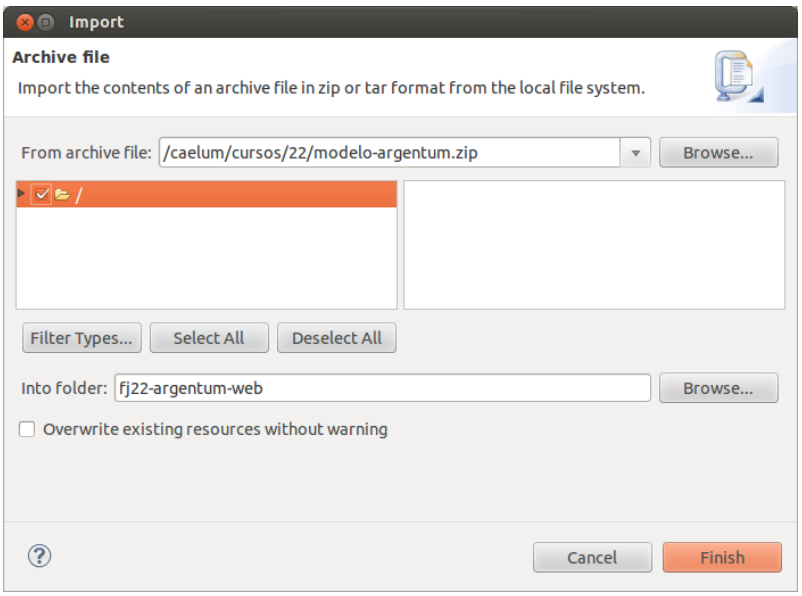
e) Clique em *Next* e, na próxima tela, selecione o projeto *fj22-argentum-web* no box *Available* (da esquerda), pressione o botão *Add >* (moverá para o box *Configured* da direita) e depois *Finish*.



f) Clique em *Finish*.

4) Por fim, precisamos importar do projeto anterior as classes como `Negociacao` ou `Candlestick`. Já o temos pronto na pasta `/caelum/cursos/22/`, com o nome de `modelo-argentum.zip`. Precisamos apenas importá-lo:

- Para importá-lo, use **ctrl + 3** *Archive File* e escolha a opção **Import (Archive file)**.
- Em *Browse...*, selecione o nosso arquivo **modelo-argentum.zip** e finalize-o.



- Note que, com esse import, trouxemos também os jars da implementação Mojarra do JSF e do Primefaces, que usaremos daqui pra frente.

Para casa...

Se você está fazendo esse exercício em casa, certifique-se que seu projeto anterior está funcionando corretamente e simplesmente copie os pacotes dele para o novo.

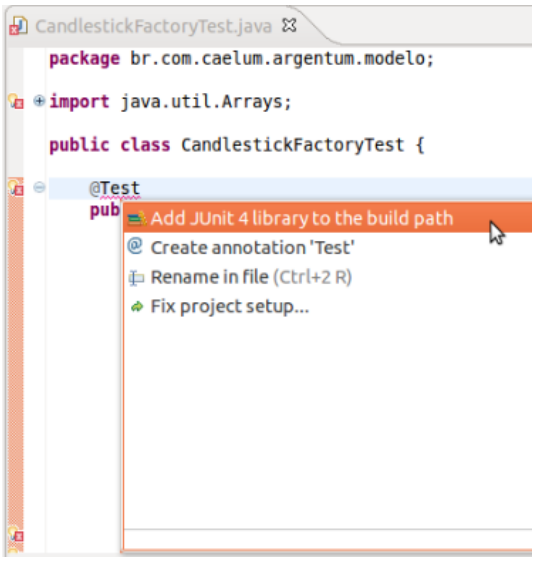
Não esqueça de copiar também o jar do XStream para a pasta `WebContent/WEB-INF/lib/`.

Além disso, no zip da aula ainda há os jars do JSF e do PrimeFaces, que usaremos a seguir. Nesta versão da apostila estamos usando as versões 2.x.x e 3.5.x, respectivamente. Links para o download:

- JSF: https://javaserverfaces.java.net/download.html
- Primefaces: http://primefaces.org/downloads.html

5) Nossas classes de teste (*src/test/java*) ainda apresentam problemas relacionados ao JUnit. Falta adicioná-lo ao *Build Path*.

   Abra a classe **CandlestickFactoryTest** e dê **ctrl + 1** na anotação *@Test*. Escolha a opção *Add JUnit 4 library to the build path*.



6) Finalmente, para evitar confusões mais para a frente, feche o projeto que fizemos nos outros dias de curso. Clique com o botão direito no fj22-argentum-**base** e escolha a opção **Close project**

7.4 A primeira página com JSF

Como configuramos, na criação do projeto, que o JSF será responsável por responder às requisições com extensão `.xhtml`. Dessa forma, tabalharemos com arquivos xhtml no restante do curso.

Vale relembrar uma diferença fundamental entre as duas formas de desenvolvimento para a web. A abordagem *action based*, como no SpringMVC e no VRaptor, focam seu funcionamento nas classes que contêm as lógicas. A view é meramente uma camada de apresentação do que foi processado no modelo.

Enquanto isso, o pensamento *component based* adotado pelo JSF leva a view como a peça mais importante -- é a partir das necessidades apontadas pelos componentes da view que o modelo é chamado e populado com dados.

As tags que representam os componentes do JSF estão em duas *taglibs* principais (bibliotecas de tags): a **core** e a **html**.

A taglib *html* contém os componentes necessários para montarmos nossa tela gerando o HTML adequado. Já a *core* possui diversos componentes não visuais, como tratadores de eventos ou validadores. Por ora, usaremos apenas os componentes da *h:html*

Importando as tags em nossa página

Diferente da forma importação de taglibs em JSPs que vimos no curso de Java para a web (FJ-21), para importar as tags no JSF basta declararmos seus namespaces no arquivo `.xhtml`. Dessa forma, teremos:

```
<!DOCTYPE html PUBLIC //W3C//DTD XHTML 1.0 Transitional//EN
    http://www.w3.org/TR/xhtml1/DTD/xhtml1transitional.dtd>

<html xmlns=http://www.w3.org/1999/xhtml
        xmlns:h=http://java.sun.com/jsf/html>

        <! aqui usaremos as tags do JSF >

</html>
```

Definindo a interface da aplicação

Como qualquer outro aprendizado de tecnologia, vamos começar a explorar o JSF criando nossa primeira tela com uma mensagem de boas vindas para o usuário.

Como todo arquivo HTML, todo o cabeçalho deve estar dentro da tag `head` e o que será renderizado no navegador deve ficar dentro da tag `body`. Uma página padrão para nós seria algo como:

```
<html ...>
    <head>
        <! cabeçalho aqui >
    </head>
    <body>
        <! informações a serem mostradas >
    </body>
</html>
```

Quando estamos lidando com o JSF, no entanto, precisamos nos lembrar de utilizar preferencialmente as tags do próprio framework, já que, à medida que utilizarmos componentes mais avançados, o JSF precisará gerenciar os próprios *body* e *head* para, por exemplo, adicionar CSS e javascript que um componente requisitar.

Assim, usando JSF preferiremos utilizar as tags estruturais do HTML que vêm da taglib http://java.sun.com/jsf/html, nosso html vai ficar mais parecido com esse:

```
<html ...>
    <h:head>
        <! cabeçalho aqui >
    </h:head>
    <h:body>
        <! informações a serem mostradas >
    </h:body>
</html>
```

Mostrando informações com h:outputText

Como queremos mostrar uma saudação para o visitante da nossa página, podemos usar a tag `h:outputText`. É através do seu atributo `value` que definimos o texto que será apresentado na página.

Juntando tudo, nosso primeiro exemplo é uma tela simples com um texto:

```
<!DOCTYPE html PUBLIC //W3C//DTD XHTML 1.0 Transitional//EN
    http://www.w3.org/TR/xhtml1/DTD/xhtml1transitional.dtd>
<html xmlns=http://www.w3.org/1999/xhtml
        xmlns:h=http://java.sun.com/jsf/html>

    <h:head>
        <title>Argentum Web</title>
    </h:head>
    <h:body>
        <h:outputText value =Olá JSF! />
     </h:body>
</html>
```

7.5 Interagindo com o modelo: Managed Beans

O `h:outputText` é uma tag com um propósito aparentemente muito bobo e, no exemplo acima, é exatamente equivalente a simplesmente escrevermos "Olá JSF!" diretamente. E, de fato, para textos fixos, não há problema em escrevê-lo diretamente!

Contudo, se um pedaço de texto tiver que interagir com o modelo, uma lógica ou mesmo com outros componentes visuais, será necessário que ele também esteja guardado em um componente.

Exemplos dessas interações, no caso do `h:outputText`: mostrar informações vindas de um banco de dados, informações do sistema, horário de acesso, etc.

Para mostrar tais informações, precisaremos executar um código Java e certamente não faremos isso na camada de visualização: esse código ficará separado da *view*, em uma classe de modelo. Essas classes de modelo que interagem com os componentes do JSF são os **Managed Beans**.

Estes, são apenas classezinhas simples que com as quais o JSF consegue interagir através do acesso a seus métodos. Nada mais são do que POJOs anotados com `@ManagedBean`.

> **POJO (Plain Old Java Object)**
>
> *POJO* é um termo criado por Martin Fowler, Rebecca Parsons e Josh Mackenzie que serve para definir um objeto simples. Segundo eles, o termo foi criado pois ninguém usaria objetos simples nos seus projetos pois não existia um nome extravagante para ele.

Se quisermos, por exemplo, mostrar quando foi o acesso do usuário a essa página, podemos criar a seguinte classe:

```
@ManagedBean
public class OlaMundoBean {

    public String getHorario() {
        SimpleDateFormat sdf = new SimpleDateFormat(hh:mm:ss);
        return Atualizado em  + sdf.format(new Date());
    }
}
```

E, bem semelhantemente à forma padrão nas JSPs vistas no treinamento de Java para a Web, acessaremos o *getter* através da *Expression Language*. Existe apenas uma pequena diferença: para chamar os métodos no JSF, em vez do cifrão (`$`), usaremos a cerquilha (`#`).

```
<h:outputText value=#{olaMundoBean.horario} />
```

Ao fazer colocar o código acima, estamos dizendo que há uma classe gerenciada pelo JSF chamada **OlaMundoBean** que tem um método `getHorario` -- e que o retorno desse método será mostrado na página. É uma forma extremamente simples e elegante de ligar a *view* a métodos do *model*.

7.6 Recebendo informações do usuário

Agora que já sabemos conectar a página à camada de modelo, fica fácil obter dados do usuário! Por nossa vivência com aplicações web, até mesmo como usuários, sabemos que a forma mais comum de trazer tais dados para dentro da aplicação é através de formulários.

A boa notícia é que no JSF não será muito diferente! Se para mostrar dados na página usamos a tag `h:outputText`, para trazer dados do usuário para dentro da aplicação, usaremos a tag `h:inputText`. Ela fará a ligação entre o atributo do seu bean e o valor digitado no campo.

Note que a ideia é a mesma de antes: como o JSF precisará interagir com os dados desse componente, não podemos usar a tag HTML que faria o mesmo trabalho. Em vez disso, usaremos a taglib de HTML provida pelo próprio JSF, indicando como a informação digitada será guardada no bean.

```
<h:outputLabel value=Digite seu nome:/>
<h:inputText value=#{olaMundoBean.nome}/>
```

Apenas com esse código, já podemos ver o texto *Digite seu nome* e o campo de texto onde o usuário digitará. Sabemos, no entanto, que não faz sentido ter apenas um campo de texto! É preciso ter também um botão para o usuário confirmar que acabou de digitar o nome e um formulário para agrupar todas essas tags.

Botão e o formulário em JSF

Esse é um pequeno ponto de divergência entre o HTML puro e o JSF. Em um simples formulário HTML, configuramos a *action* dele na própria tag `form` e o papel do botão é apenas o de mandar executar a ação já configurada.

Para formulários extremamente simples, isso é o bastante. Mas quando queremos colocar dois botões com ações diferentes dentro de um mesmo formulário, temos que recorrer a um JavaScript que fará a chamada correta.

Como dito antes, no entanto, o JSF tem a proposta de abstrair todo o protocolo HTTP, o JavaScript e o CSS. Para ter uma estrutura em que o formulário é marcado apenas como um agregador de campos e cada um dos botões internos pode ter funções diferentes, a estratégia do JSF foi a de deixar seu form como uma tag simples e adicionar a configuração da ação ao próprio botão.

```
<h:form>
    <h:outputLabel for=nome value=Digite seu nome:/>
    <h:inputText id=nome value=#{olaMundoBean.nome}/>
    <h:commandButton value=Ok action=#{olaMundoBean.digaOi}/>
</h:form>
```

Quando o usuário clica no botão *Ok*, o JSF chama o setter do atributo nome do `OlaMundoBean` e, logo em seguida, chama o método `digaOi`. Repare que esta ordem é importante: o método provavelmente dependerá dos dados inseridos pelo usuário.

Note, também, que teremos um novo método no *managed bean* chamado `digaOi`. Os botões sempre estão atrelados a métodos porque, na maior parte dos casos, realmente queremos executar alguma ação além da chamada do setter. Essa ação pode ser a de disparar um processo interno, salvar no banco ou qualquer outra necessidade.

O que fazer enquanto não ainda houver informação?

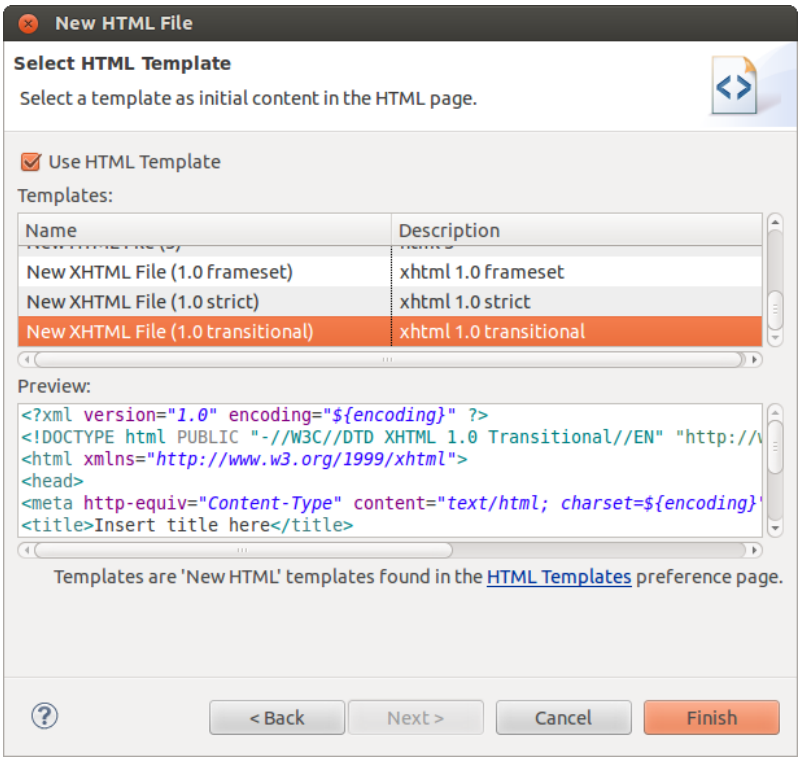
Sabendo que, antes de chamar o método correspondente à ação do botão, o JSF preenche os atributos através dos setters, sabemos que teremos a informação a ser mostrada para o usuário.

No entanto, muitas vezes não gostaríamos de mostrar um campo enquanto ele não estiver preenchido e, felizmente, o JSF tem uma forma bastante simples de só mostrar um `h:outputText` na tela apenas se a informação estiver preenchida! Basta usar o atributo `rendered`:

```
<h:outputText value=Oi #{olaMundoBean.nome}
              rendered=#{not empty olaMundoBean.nome}/>
```

7.7 Exercícios: Os primeiros componentes JSF

1) Use **ctrl + N** *HTML* para criar o arquivo `olaMundo.xhtml` na pasta `WebContent` da sua aplicação. Escolha *Next* e, na próxima tela, escolha o template *xhtml 1.0 transitional*, usualmente a última opção da lista:



Selecione a mesma opção da imagem acima e pressione *Finish*.

Implemente nosso primeiro código JSF com apenas uma saída de texto:

```
<!DOCTYPE html PUBLIC //W3C//DTD XHTML 1.0 Transitional//EN
        http://www.w3.org/TR/xhtml1/DTD/xhtml1transitional.dtd>
<html xmlns=http://www.w3.org/1999/xhtml
      xmlns:h=http://java.sun.com/jsf/html>
    <h:head>
        <title>Argentum</title>
    </h:head>

    <h:body>
        <h:outputText value=Ola Mundo />
    </h:body>
</html>
```

2) Inicie o Tomcat e acesse a URL: http://localhost:8080/fj22-argentum-web/olaMundo.xhtml

3) Verifique o código fonte gerado pela página. Repare que ele não é nada mais que simples HTML. Para isso, na maior parte dos navegadores, use **ctrl + U**.

   Repare no uso das tags `<h:head>`, `<h:body>` e `<h:outputText>`: elas não aparecem no html gerado! Sua função é apenas indicar para o JSF como gerar o código HTML necessário para o exemplo funcionar.

4) Além de usar mensagens fixas, poderíamos fazer com que a mensagem seja devolvida de uma classe responsável por prover objetos para uma view: um dos chamados `ManagedBeans`. Vamos começar criando essa classe contendo apenas a mensagem inicial.

   Crie uma classe chamada `OlaMundoBean`, com apenas o atributo `mensagem` já inicializada, seu getter e não esqueça de **anotar a classe** com `@ManagedBean`

```
@ManagedBean
public class OlaMundoBean {

    private String mensagem = Quem é você?;

    public String getMensagem() {
        return mensagem;
    }
}
```

5) Alteremos o arquivo xhtml, então, para que ele use a mensagem *Quem é você?* que escrevemos *hard-coded* na classe `OlaMundoBean`. Usaremos a *Expression Language* específica do JSF para isso, que é capaz de pegar informações de qualquer classe configurada como um `ManagedBean`.

   Basta alterar o *value* da tag `h:outputText`:

```
...
<h:body>
    <h:outputText value=#{olaMundoBean.mensagem} />
</h:body>
```

6) Agora, se quisermos pegar a resposta do usuário e cumprimentá-lo propriamente, podemos adicionar à nossa página um campo de texto para que o usuário digite seu nome. Então, trocaremos a mensagem cumprimentando ele. Comecemos pelas alterações no olaMundo.xhtml, adicionando um h:inputText e um botão para o usuário enviar seu nome, como abaixo.

   Atenção! Não esqueça da tag **h:form** em volta do formulário. Lembre-se que, sem ela, os botões não funcionam.

```
...
<h:body>
    <h:form>
        <h:outputText value=#{olaMundoBean.mensagem} /><br />
        <h:inputText value=#{olaMundoBean.nome} />
        <h:commandButton action=#{olaMundoBean.nomeFoiDigitado}
                         value=Ok/>
    </h:form>
</h:body>
```

7) Essa alteração, no entanto, não é suficiente. Se você rodar o servidor agora, notará que a página, que antes funcionava, agora lança uma `ServletException` informando que *Property 'nome' not found on type br.com.caelum.argentum.bean.OlaMundoBean*.

   Isto é, falta adicionarmos o atributo nome e seu getter à página, como fizemos com a mensagem, no outro exercício. Adicione à classe `OlaMundoBean` o atributo e seu getter.

```
@ManagedBean
public class OlaMundoBean {
    ...
    private String nome;

    public String getNome() {
        return nome;
    }
    ...
}
```

8) Agora sim podemos ver a mensagem, o campo de texto e o botão. Contudo, ao apertar o botão, levamos uma `javax.el.PropertyNotFoundException` informando que nome é um atributo não alterável.

   Faltou adicionarmos o setter do atributo à `OlaMundoBean`, para que o JSF possa preenchê-lo! Além disso, o botão chamará o método `nomeFoiDigitado`, que também não existe ainda.

   Complete a classe com o setter faltante e o método `nomeFoiDigitado`, reinicie o servidor e teste!

```
@ManagedBean
public class OlaMundoBean {
    // ...tudo o que já existia aqui
```

```
    public void setNome(String nome) {
        this.nome = nome;
    }

    public void nomeFoiDigitado() {
        System.out.println(\nChamou o botão);
    }
}
```

9) (Opcional) Para entender melhor o ciclo de execução de cada chamada ao JSF, adicione `System.out.println("nome do método")` a cada um dos métodos da sua aplicação e veja a ordem das chamadas pelo console do Eclipse.

7.8 A lista de negociações

Agora que já aprendemos o básico do JSF, nosso objetivo é listar em uma página as negociações do web service que o Argentum consome. Nessa listagem, queremos mostrar as informações das negociações carregadas -- isto é, queremos uma forma de mostrar preço, quantidade e data de cada negociação. E a forma mais natural de apresentar dados desse tipo é, certamente, uma tabela.

Até poderíamos usar a tabela que vem na taglib padrão do JSF, mas ela é bastante limitada e não tem pré-definições de estilo. Isto é, usando a taglib padrão, teremos sim uma tabela no HTML, mas ela será mostrada da forma mais feia e simples possível.

Já falamos, contudo, que a proposta do JSF é abstrair toda a complexidade relativa à web -- e isso inclui CSS, formatações, JavaScript e tudo o mais. Então, em apoio às tags básicas, algumas bibliotecas mais sofisticadas surgiram. As mais conhecidas delas são PrimeFaces, RichFaces e IceFaces.

Taglibs como essas oferecem um visual mais bacana já pré-pronto e, também, diversas outras facilidades. Por exemplo, uma tabela que utilize as tags do Primefaces já vem com um estilo bonito, possibilidade de colocar cabeçalhos nas colunas e até recursos mais avançados como paginação dos registros.

O componente responsável por produzir uma tabela baseada em um modelo se chama `dataTable`. Ele funciona de forma bem semelhante ao `for` do Java 5 ou o `forEach` da JSTL: itera em uma lista de elementos atribuindo cada item na variável definida.

```
<html xmlns=http://www.w3.org/1999/xhtml
      xmlns:h=http://java.sun.com/jsf/html
      xmlns:p=http://primefaces.org/ui>
    <h:head>
        <title>Argentum</title>
    </h:head>
    <h:body>
        <p:dataTable var=negociacao value=#{argentumBean.negociacoes}>
```

```
        </p:dataTable>
    </h:body>
</html>
```

O código acima chamará o método `getNegociacoes` da classe `ArgentumBean` e iterará pela lista devolvida atribuindo o objeto à variável `negociacao`. Então, para cada coluna que quisermos mostrar, será necessário apenas manipular a negociação do momento.

E, intuitivamente o bastante, cada coluna da tabela será representada pela tag `p:column`. Para mostrar o valor, você pode usar a tag que já vimos antes, o `h:outputText`. Note que as tags do Primefaces se integram perfeitamente com as básicas do JSF.

```
<p:dataTable var=negociacao value=#{argentumBean.negociacoes}>
    <p:column headerText=Preço>
        <h:outputText value=#{negociacao.preco}/>
    </p:column>
    ... outras colunas
</p:dataTable>
```

Falta ainda implementar a classe que cuidará de devolver essa lista de negociações. O código acima sugere que tenhamos uma classe chamada `ArgentumBean`, gerenciada pelo JSF, que tenha um getter de negociações que pode, por exemplo, trazer essa lista direto do `ClienteWebService` que fizemos anteriormente:

```
@ManagedBean
public class ArgentumBean {

    public List<Negociacao> getNegociacoes() {
        return new ClienteWebService().getNegociacoes();
    }
}
```

Da forma acima, o exemplo já funciona e você verá a lista na página. No entanto, nesse exemplo simples o JSF chamará o método `getNegociacoes` duas vezes durante uma mesma requisição. Isso não seria um problema se ele fosse um getter padrão, que devolve uma referência local, mas note como nosso `getNegociacoes` vai buscar a lista diretamente no web service. Isso faz com que, para construir uma simples página, tenhamos que esperar a resposta do serviço... duas vezes!

Esse comportamento não é interessante. Nós gostaríamos que o Argentum batesse no serviço em busca dos dados apenas uma vez por requisição, e não a cada vez que o JSF chame o *getter*. Isso significa que o acesso ao serviço não pode estar diretamente no método `getNegociacoes`, que deve apenas devolver a lista pré-carregada.

No JSF, o comportamento padrão diz que um objeto do `ManagedBean` dura por uma requisição. Em outras palavras, o escopo padrão dos *beans* no JSF é o de requisição. Isso significa que um novo `ArgentumBean`

será criado a cada vez que um usuário chamar a página da listagem. E, para cada chamada a essa página, precisamos buscar a lista de negociações no serviço apenas uma vez. A resposta para esse problema, então, é bastante simples e apareceu logo no início do aprendizado do Java orientado a objetos.

Basta colocar a chamada do web service naquele bloco de código que é chamado apenas na criação do objeto, isto é, no construtor. Ao armazenar a listagem em um atributo, o *getter* de negociações passa a simplesmente devolver a referência, evitando as múltiplas chamadas a cada requisição.

```
@ManagedBean
public class ArgentumBean {

    private List<Negociacao> negociacoes;

    public ArgentumBean() {
        ClienteWebService cliente = new ClienteWebService();
        this.negociacoes = cliente.getNegociacoes();
    }

    public List<Negociacao> getNegociacoes() {
        return this.negociacoes;
    }
}
```

Juntando as informações dessa seção, já conseguimos montar a listagem de negociações com os dados vindos do *web service*. E o processo será muito frequentemente o mesmo para as diversas outras telas: criamos a página usando as tags do Primefaces em complemento às básicas do JSF, implementamos a classe que cuidará da lógica por trás da tela e a anotamos com `@ManagedBean`.

7.9 Formatação de Data com JSF

A tabela já é funcional, mas com a data mal formatada. O componente não sabe como gostaríamos de formatar a data e chama por de baixo dos planos o método `toString` da data para receber uma apresentação como `String`.

Preço	Quantidade	Volume	
321.65	18	5789.7	java.util.GregorianCalendar[time=1380758400000,areFieldsSet=true,areAllFie
384.39	22	8456.58	java.util.GregorianCalendar[time=1380758400000,areFieldsSet=true,areAllFie
298.99	17	5082.83	java.util.GregorianCalendar[time=1380758400000,areFieldsSet=true,areAllFie
309.49	18	5570.82	java.util.GregorianCalendar[time=1380758400000,areFieldsSet=true,areAllFie
392.33	23	9023.59	java.util.GregorianCalendar[time=1380844800000,areFieldsSet=true,areAllFie
425.02	25	10625.5	java.util.GregorianCalendar[time=1380844800000,areFieldsSet=true,areAllFie

A forma clássica de resolver esse problema seria através de um getter que traria a data formatada por um `SimpleDateFormat`. Mas, assim como a JSTL vista no curso de Java para a Web, o JSF também tem uma tag

para formatar valores, números e, claro, datas. Essas tags e muitas outras, são parte da biblioteca fundamental de tags lógicas do JSF e, para usá-las, será necessário importar tal taglib.

Assim como as bibliotecas de tags de HTML e do Primefaces, para utilizar essas será necessário declará-las no namespace da sua página.

Daí, podemos facilmente mudar a forma padrão de exibição usando o componente de formatação `f:convertDateTime` que define um *pattern* para a data. É importante lembrar que, internamente, o `f:convertDateTime` acaba fazendo uma chamada ao `SimpleDateFormat` e, assim, só podemos formatar objetos do tipo `java.util.Date` com ele. Por essa razão, chamaremos o método *getTime* que devolve a representação em `Date` do `Calendar` em questão. Mais uma vez podemos omitir a palavra "get" com expression language. Segue a tabela completa:

```
<html xmlns=http://www.w3.org/1999/xhtml
        xmlns:f=http://java.sun.com/jsf/core
        xmlns:h=http://java.sun.com/jsf/html
        xmlns:p=http://primefaces.org/ui>

    <h:body>
        <p:dataTable var=negociacao value=#{argentumBean.negociacoes}>

            ... outras colunas, e então:
            <p:column headerText=Data>
                <h:outputText value=#{negociacao.data.time}>
                    <f:convertDateTime pattern=dd/MM/yyyy/>
                </h:outputText>
            </p:column>
        </p:dataTable>
    </h:body>
</html>
```

7.10 Exercícios: p:dataTable para listar as Negociações do Web Service

1) Use **ctrl + N** *HTML* para criar um novo arquivo na pasta `WebContent` chamado `index.xhtml`. Como já fizemos antes, clique em *Next* e, na tela seguinte, escolha o template *xhtml 1.0 transitional*.

   O Eclipse vai gerar um arquivo com um pouco de informações a mais, mas ainda muito parecido com o seguinte, onde mudamos o title:

   ```
   <html xmlns=http://www.w3.org/1999/xhtml>
       <head>
           <title>Argentum Web</title>
       </head>
   ```

```
    <body>

    </body>
</html>
```

2) Como vamos usar o JSF nesse arquivo e já temos até mesmo o JAR primefaces-3.x.jar adicionado ao projeto (veja em `WebContent/WEB-INF/lib`) basta declarar os namespaces das taglibs do JSF e do Primefaces, que usaremos no exercício.

   Além disso, para que os componentes consigam incluir seu CSS à nossa página, altere as tags head e body de forma a usar suas versões gerenciadas pelo JSF:

```
<html xmlns=http://www.w3.org/1999/xhtml
      xmlns:h=http://java.sun.com/jsf/html
      xmlns:f=http://java.sun.com/jsf/core
      xmlns:p=http://primefaces.org/ui>
    <h:head>
        <title>Argentum Web</title>
    </h:head>
    <h:body>

    </h:body>
</html>
```

3) Agora, **dentro do h:body**, vamos começar a montar nossa tabela de negociações. O componente que usaremos para isso é o `p:datatable`, do Primefaces. Ele precisará da lista de negociações e, assim como um `forEach`, uma variável para que cada coluna seja preenchida.

```
<p:dataTable var=negociacao value=#{argentumBean.negociacoes}>

</p:dataTable>
```

4) Esse código acima diz que o componente `dataTable` do Primefaces chamará o método `getNegociacoes()` da classe `ArgentumBean` e, para cada linha da tabela, disponibilizará a negociação da vez na variável `negociacao`.

   O problema é que o managed bean `ArgentumBean` ainda não existe e, claro, nem o método `getNegociacoes()` dela. E como cada vez que a página `index.xhtml` for requisitada ela fará algumas chamadas ao `getNegociacoes`, faremos a chamada ao *webservice* no construtor e, a cada chamada ao getter, apenas devolveremos a referência à mesma lista.

   a) Crie a classe `ArgentumBean` com **ctrl + N** *Class*, no pacote `br.com.caelum.argentum.bean` e anote ela com `@ManagedBean`.

```
@ManagedBean
public class ArgentumBean {

}
```

b) Adicione o construtor que faça a chamada ao webservice através do `ClienteWebservice`, guarde a lista em um atributo e crie o getter que o componente chamará.

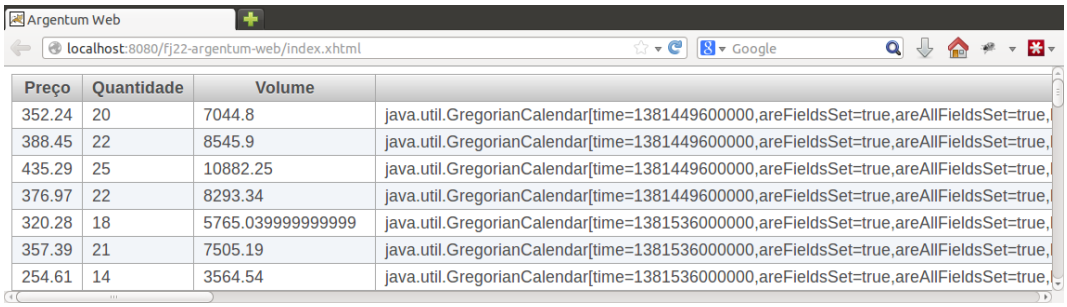
```
@ManagedBean
public class ArgentumBean {

    private List<Negociacao> negociacoes;

    public ArgentumBean() {
        negociacoes = new ClienteWebService().getNegociacoes();
    }

    public List<Negociacao> getNegociacoes() {
        return negociacoes;
    }
}
```

5) Agora, nossa página já não dá erro, mas nada é mostrado na tela, quando a acessamos. Falta indicarmos quais colunas queremos na nossa tabela -- no nosso caso: preço, quantidade, volume e data. Em cada coluna, adicionaremos um título e cada uma delas também mostrará o valor de texto.

Para criar a coluna com o título, usaremos o componente `p:column` e, como já fizemos antes, para mostrar o valor necessário, usaremos a `h:outputText`.

```
<p:dataTable var=negociacao value=#{argentumBean.negociacoes}>
    <p:column headerText=Preço>
        <h:outputText value=#{negociacao.preco}/>
    </p:column>
    <p:column headerText=Quantidade>
        <h:outputText value=#{negociacao.quantidade}/>
    </p:column>
    <p:column headerText=Volume>
        <h:outputText value=#{negociacao.volume}/>
    </p:column>
    <p:column headerText=Data>
        <h:outputText value=#{negociacao.data}/>
    </p:column>
</p:dataTable>
```

6) Reinicie o Tomcat e acesse em seu navegador o endereço http://localhost:8080/fj22-argentum-web/index.xhtml . O resultado deve ser algo parecido com:



Preço	Quantidade	Volume	
352.24	20	7044.8	java.util.GregorianCalendar[time=1381449600000,areFieldsSet=true,areAllFieldsSet=true,
388.45	22	8545.9	java.util.GregorianCalendar[time=1381449600000,areFieldsSet=true,areAllFieldsSet=true,
435.29	25	10882.25	java.util.GregorianCalendar[time=1381449600000,areFieldsSet=true,areAllFieldsSet=true,
376.97	22	8293.34	java.util.GregorianCalendar[time=1381449600000,areFieldsSet=true,areAllFieldsSet=true,
320.28	18	5765.039999999999	java.util.GregorianCalendar[time=1381536000000,areFieldsSet=true,areAllFieldsSet=true,
357.39	21	7505.19	java.util.GregorianCalendar[time=1381536000000,areFieldsSet=true,areAllFieldsSet=true,
254.61	14	3564.54	java.util.GregorianCalendar[time=1381536000000,areFieldsSet=true,areAllFieldsSet=true,

7) As informações de preço, quantidade e volume estão legíveis, mas a data das negociações está mostrando um monte de informações que não nos interessam. Na verdades, o que precisamos na coluna data é de informações de dia, mês, ano e, no máximo, horário de cada movimentação.

   Adicione a tag `f:convertDateTime` à coluna da data. Essa tag modificará o comportamento da `h:outputText` para mostrá-lo formatado de acordo com o padrão passado. Note que a tag h:outputText passará a ser fechada depois da formatação da data:

```
...
<p:column headerText=Data>
    <h:outputText value=#{negociacao.data.time}>
        <f:convertDateTime pattern=dd/MM/yyyy/>
    </h:outputText>
</p:column>
...
```

7.11 Para saber mais: paginação e ordenação

O componente `p:dataTable` sabe listar items, mas não pára por aí. Ele já vem com várias outras funcionalidades frequentemente necessárias em tabelas já prontas e fáceis de usar.

Muitos dados

Por exemplo, quando um programa traz uma quantidade muito grande de dados, isso pode causar uma página pesada demais para o usuário que provavelmente nem olhará com atenção todos esses dados.

Uma solução clássica para resultados demais é mostrá-los aos poucos, apenas conforme o usuário indicar que quer ver os próximos resultados. Estamos, é claro, falando da paginação dos resultados e o componente de tabelas do Primefaces já a disponibiliza!

Para habilitar a paginação automática, basta adicionar o atributo `paginator="true"` à sua `p:dataTable` e definir a quantidade de linhas por página pelo atributo `rows`. A definição da tabela de negociações para paginação de 15 em 15 resultados ficará assim:

```
<p:dataTable var=negociacao value=#{argentumBean.negociacoes}
             paginator=true rows=15>
```

```
    <! colunas omitidas >
</p:dataTable>
```

Essa pequena mudança já traz uma visualização mais legal para o usuário, mas estamos causando um problema silencioso no servidor. A cada vez que você chama uma página de resultados, a cada requisição, o `ArgentumBean` é recriado e perdemos a lista anterior. Assim, na criação da nova instância de `ArgentumBean`, seu construtor é chamado e acessamos novamente o webservice.

Como recebemos a lista completa do webservice, podíamos aproveitar a mesma lista para todas as páginas de resultado e, felizmente, isso também é bastante simples.

O comportamento padrão de um `ManagedBean` é durar apenas uma requisição. Em outras palavras, o escopo padrão de um `ManagedBean` é de *request*. Com apenas uma anotação podemos alterar essa duração. Os três principais escopos do JSF são:

- **RequestScoped:** é o escopo padrão. A cada requisição um novo objeto do bean será criado;
- **ViewScoped:** escopo da página. Enquanto o usuário estiver na mesma página, o bean é mantido. Ele só é recriado quando acontece uma navegação em sí, isto é, um botão abre uma página diferente ou ainda quando acessamos novamente a página atual.
- **SessionScoped:** escopo de sessão. Enquanto a sessão com o servidor não expirar, o mesmo objeto do `ArgentumBean` atenderá o mesmo cliente. Esse escopo é bastante usado, por exemplo, para manter o usuário logado em aplicações.

No nosso caso, o escopo da página resolve plenamente o problema: enquanto o usuário não recarregar a página usaremos a mesma listagem. Para utilizá-lo, basta adicionar ao *bean* a anotação `@ViewScoped`. No exemplo do Argentum:

```
@ManagedBean
@ViewScoped
public class ArgentumBean {
    ...
```

Sempre que um ManagedBean possuir o escopo maior que o escopo de requisição, ele deverá implementar a interface `Serializable`:

```
@ManagedBean
@ViewScoped
public class ArgentumBean implementes Serializable {
    ...
```

Tirando informações mais facilmente

Outra situação clássica que aparece quando lidamos com diversos dados é precisarmos vê-los de diferentes formas em situações diversas.

Considere um sistema que apresenta uma tabela de contatos. Se quisermos encontrar um contato específico nela, é melhor que ela esteja ordenada pelo nome. Mas caso precisemos pegar os contatos de todas as pessoas de uma região, é melhor que a tabela esteja ordenada, por exemplo, pelo DDD.

Essa ideia de ordenação é extremamente útil e muito presente em aplicações. Como tal, essa funcionalidade também está disponível para tabelas do Primefaces. Apenas, como podemos tornar diversar colunas ordenáveis, essa configuração fica na tag da coluna.

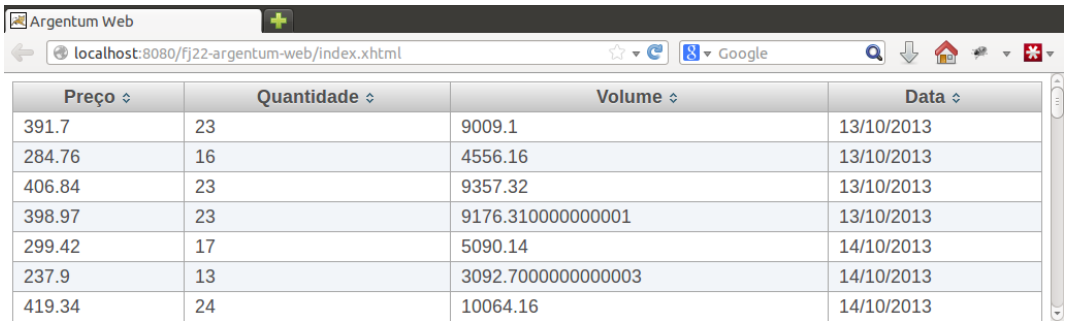
Para tornar uma coluna ordenável, é preciso adicionar um simples atributo `sortBy` à tag `h:column` correspondente. Esse atributo torna o cabeçalho dessa coluna em um elemento clicável e, quando clicarmos nele, chamará a ordenação.

Contudo, exatamente pela presença de elementos clicáveis, será necessário colocar a tabela dentro de uma estrutura que comporte botões em HTML: um formulário. E, como quem configurará o que cada clique vai disparar é o JSF, será necessário usar o formulário da taglib de HTML dele. Resumidamente, precisamos colocar a tabela inteira dentro do componente `h:form`.

Se quiséssemos tornar ordenáveis as colunas da tabela de negociações, o resultado final seria algo como:

```
<h:form id=listaNegociacao>
    <p:dataTable var=negociacao value=#{argentumBean.negociacoes}>

        <p:column sortBy=#{negociacao.preco} headerText=Preço >
             <h:outputText value=#{negociacao.preco} />
        </p:column>

        <! outras colunas omitidas >
    </p:dataTable>
<h:form>
```

Se permitirmos ordenar por qualquer coluna do modelo `Negociacao`, teremos um resultado bem atraente:



Preço	Quantidade	Volume	Data
391.7	23	9009.1	13/10/2013
284.76	16	4556.16	13/10/2013
406.84	23	9357.32	13/10/2013
398.97	23	9176.310000000001	13/10/2013
299.42	17	5090.14	14/10/2013
237.9	13	3092.7000000000003	14/10/2013
419.34	24	10064.16	14/10/2013

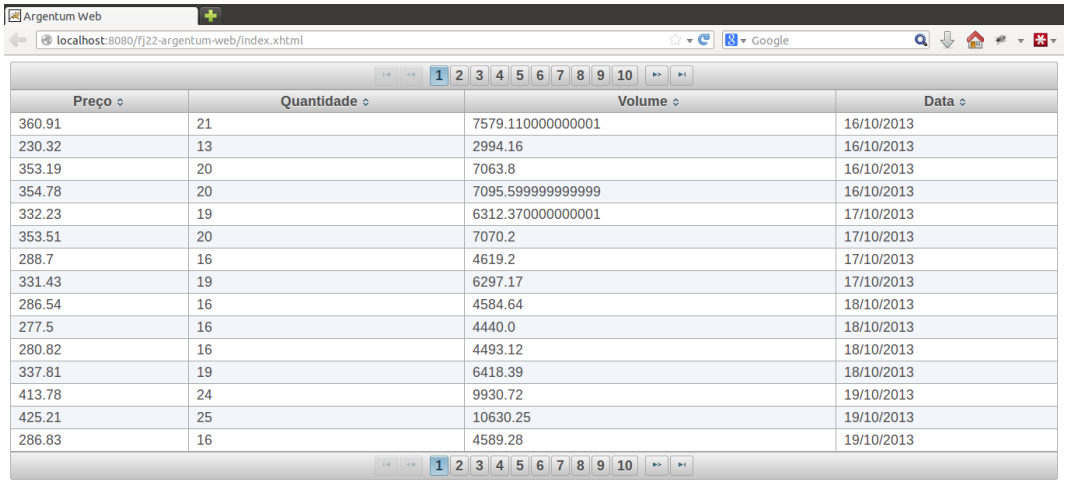
Note que não foi necessário adicionar código algum à classe ArgentumBean! Note também que é até possível usar ambas as funcionalidades na mesma tabela. E essas são apenas algumas das muitas facilidades que o `p:dataTable` oferece. Vale a pena verificar o *showcase* e documentação no site do Primefaces.

7.12 Exercício opcional: adicione paginação e ordenação à tabela

1) Vamos colocar paginação na tabela. Adicione os atributos `paginator="true"` e `rows="15"`. Adicione os atributos `paginator` e `rows`:

```
<p:dataTable var=negociacao value=#{argentumBean.negociacoes}
    paginator=true rows=15>
```

Salve a página, suba o servidor e acesse no seu navegador o endereço http://localhost:8080/fj22-argentum-web/index.xhtml . Agora você já consegue ver resultados paginados de 15 em 15 negociações:



2) Para evitar chamar o webservice a cada vez que pedimos os próximos resultados paginados, **adicione** a anotação `@ViewScoped` à classe `ArgentumBean`:

```
@ManagedBean
@ViewScoped
public class ArgentumBean {
    ...
}
```

3) Como estamos utilizando um escopo maior do que o escopo de requisição, nosso `ManagedBean` precisa implementar a interface `Serializable`. Adicione a implementação da interface `Serializable` à classe `ArgentumBean`:

```
@ManagedBean
@ViewScoped
```

```
public class ArgentumBean implements Serializable {
    ...
}
```

4) Também será necessário implementar a interface `Serializable` na classe `Negociacao`, pois ela é utilizada pela classe `ArgentumBean`:

```
public class Negociacao implements Serializable {
    ...
}
```

5) Deixe as colunas ordenáveis, use o atributo `sortBy` em cada atributo. Por exemplo, para a coluna que mostra o preço da negociação:

```
<p:column sortBy=#{negociacao.preco} headerText=Preço >
     <h:outputText value=#{negociacao.preco} />
</p:column>
```

Repare que usamos a *expression language* `#{negociacao.preco}` do JSF dentro do `sortBy` para definir o valor a ordenar.

6) Como estamos permitindo a ordenação das colunas da nossa tabela, temos que colocar nossa `p:dataTable` dentro de um `h:form`:

```
<h:form id=listaNegociacao>
    <p:dataTable var=negociacao value=#{argentumBean.negociacoes}>

        <p:column sortBy=#{negociacao.preco} headerText=Preço >
             <h:outputText value=#{negociacao.preco} />
        </p:column>

        <! outras colunas omitidas >
    </p:dataTable>
<h:form>
```

Salve a página e veja o resultado recarregando a página (F5) no seu navegador.

H:FORM SEMPRE USARÁ HTTP POST

É importante saber que diferente da tag `form` no HTML, o `h:form` sempre envia uma requisição HTTP do tipo *POST*. Ele nem nos dá a possibilidade de escolher usar requisições GET.

Isso ocorre porque o JSF tenta abstrair o mundo HTTP e assim fica mais perto do desenvolvimento Desktop tradicional. Ele esconde do desenvolvedor o fato de que uma URL está sendo chamada. Em vez disso, para o desenvolvedor, é como se botões efetivamente chamassem métodos ou eventos dentro de um *Managed Bean*.

A decisão automática pelo POST foi a forma encontrada para abstrair o HTTP.

Capítulo 8

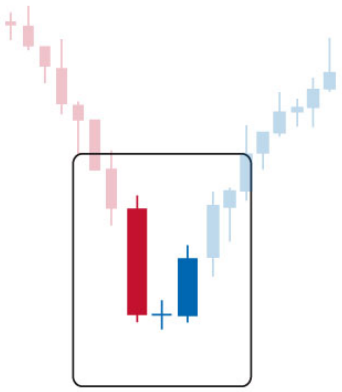
Refatoração: os Indicadores da bolsa

"Nunca confie em um computador que você não pode jogar pela janela."
– Steve Wozniak

8.1 Análise Técnica da bolsa de valores

Munehisa Homna, no século 18, foi quem começou a pesquisar os preços antigos do arroz para reconhecer padrões. Ele fez isso e começou a criar um catálogo grande de figuras que se repetiam.

A estrela da manhã, *Doji*, da figura abaixo, é um exemplo de figura sempre muito buscada pelos analistas:



Ela indica um padrão de reversão. Dizem que quando o preço de abertura e fechamento é praticamente igual (a estrela), essa é uma forte indicação de que o mercado se inverta, isto é, se estava em uma grande **baixa**, tenderá a **subir** e, se estava em uma grande **alta**, tenderá a **cair**.

Baseada nessas ideias, surgiu a **Análise Técnica Grafista**: uma escola econômica que tem como objetivo avaliar o melhor momento para compra e venda de ações através da análise histórica e comportamental do

ativo na bolsa.

Essa forma de análise dos dados gerados sobre dados das negociações (preço, volume, etc), usa gráficos na busca de padrões e faz análise de tendências para tentar prever o comportamento futuro de uma ação.

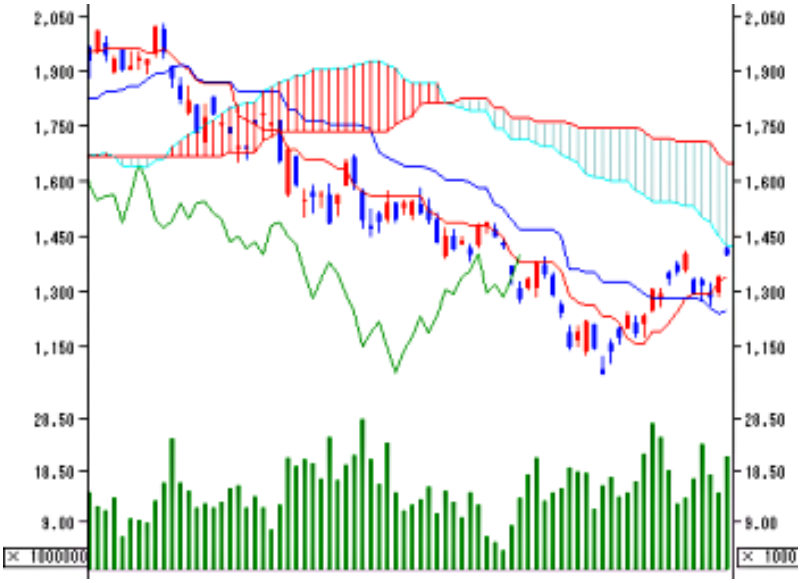
A análise técnica surgiu no início do século 20, com o trabalho de Charles Dow e Edward Jones. Eles criaram a empresa Dow Jones & Company e foram os primeiros a inventarem índices para tentar prever o comportamento do mercado de ações. O primeiro índice era simplesmente uma média ponderada de 11 ativos famosos da época, que deu origem ao que hoje é conhecido como Dow-Jones.

A busca de padrões nos candlesticks é uma arte. Através de critérios subjetivos e formação de figuras, analistas podem determinar, com algum grau de acerto, como o mercado se comportará dali para a frente.

8.2 Indicadores Técnicos

Uma das várias formas de aplicar as premissas da análise técnica grafista é através do uso de indicadores técnicos. **Indicadores** são fórmulas que manipulam dados das negociações e tiram valores deles em busca de informações interessantes para recomendar as próximas ações para um ativo. Esse novo número, é determinístico e de fácil cálculo por um computador. É até de praxe que analistas financeiros programem diversas dessas fórmulas em macros VBScript, para vê-las dentro do Excel.

É comum, na mesma visualização, termos uma combinação de gráficos, indicadores e até dos *candles*:



Diversos livros são publicados sobre o assunto e os principais *homebrokers* fornecem softwares que traçam esses indicadores e muitos outros. Além disso, você encontra uma lista com os indicadores mais usados e como calculá-los em: http://stockcharts.com/school/doku.php?id=chart_school:technical_indicators

8.3 As médias móveis

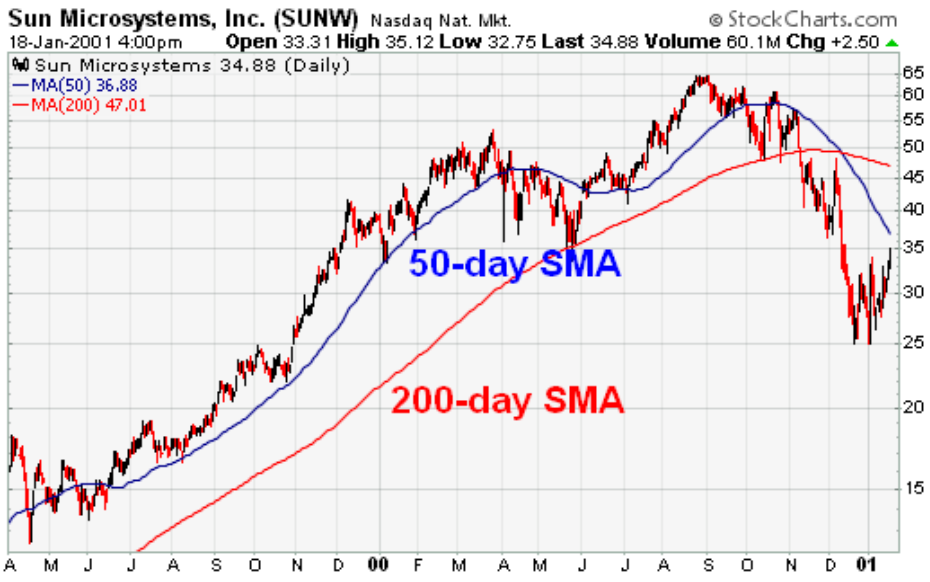
Há diversos tipos de médias móveis usadas em análises técnicas e elas são frequentemente usadas para investidores que fazem compras/vendas em intervalos muito maiores do que o intervalo de recolhimento de dados para as *candles*. As médias mais famosas são a simples, a ponderada, a exponencial e a Welles Wilder.

Vamos ver as duas primeiras, a média móvel simples e a média móvel ponderada.

Média móvel simples

A média móvel simples calcula a média aritmética de algum dos valores das candlesticks do papel para um determinado intervalo de tempo -- em geral, o valor de fechamento. Basta pegar todos os valores, somar e dividir pelo número de dias.

A figura a seguir mostra duas médias móveis simples: uma calculando a média dos últimos 50 dias e outra dos últimos 200 dias. O gráfico é do valor das ações da antiga Sun Microsystems em 2001.



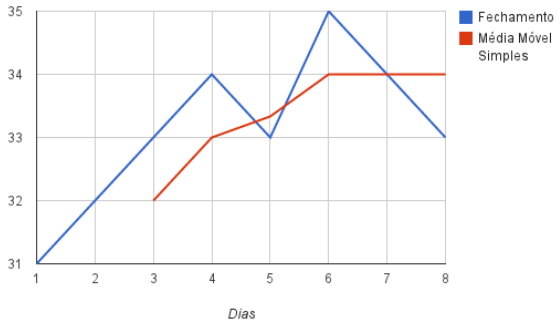
Repare que a média móvel mais 'curta', a de 50 dias, responde mais rápido aos movimentos atuais da ação, mas pode gerar sinais pouco relevantes a médio prazo.

Usualmente, estamos interessados na média móvel dos **últimos** `N` dias e queremos definir esse dia inicial. Por exemplo, para os dados de fechamento abaixo:

```
DIA     FECHAMENTO
dia 1:      31
dia 2:      32
dia 3:      33
dia 4:      34
dia 5:      33
```

```
dia 6:      35
dia 7:      34
dia 8:      33
```

Vamos fazer as contas para que o indicador calcule a média para os 3 dias anteriores ao dia que estamos interessados. Por exemplo: se pegamos o **dia 6**, a média móvel simples para os últimos 3 dias é a soma do dia 4 ao dia 6: (34 + 33 + 35) / 3 = **34**. A média móvel do dia 3 para os últimos 3 dias é 2: (31 + 32 + 33) / 3. E assim por diante.



O gráfico anterior das médias móveis da Sun pega, para cada dia do gráfico, a média dos 50 dias anteriores.

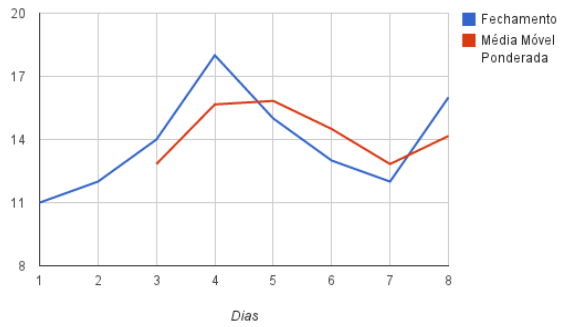
Média móvel ponderada

Outra média móvel muito famosa é a ponderada. Ela também leva em conta os últimos `N` dias a partir da data a ser calculada. Mas, em vez de uma média aritmética simples, faz-se uma média ponderada onde damos mais *peso* para o valor mais recente e vamos diminuindo o peso dos valores conforme movemos para valores mais antigos.

Por exemplo, para os dias a seguir:

```
DIA     FECHAMENTO
dia 1:      11
dia 2:      12
dia 3:      14
dia 4:      18
dia 5:      15
dia 6:      13
dia 7:      12
dia 8:      16
```

Vamos calcular a média móvel para os últimos 3 dias, onde hoje tem peso 3, ontem tem peso 2 e anteontem tem peso 1. Se calcularmos a média móvel ponderada para o dia 6 temos: (13*3 + 15*2 + 18*1) / 6 = **14.50**.



A média ponderada nos dá uma visão melhor do que está acontecendo no momento com a cotação da minha ação, mostrando com menos importância os resultados "atrasados", portanto menos relevantes para minhas decisões de compra e venda atuais. Essa média, contudo, não é tão eficiente quando estudamos uma série a longo prazo.

8.4 Exercícios: criando indicadores

1) O cálculo de uma média móvel é feito a partir de uma lista de resumos do papel na bolsa. No nosso caso, vamos pegar vários Candlesticks, um para cada dia, e usar seus valores de fechamento.

   Para encapsular a lista de candles e aproximar a nomenclatura do código à utilizada pelo cliente no dia a dia, vamos criar a classe `SerieTemporal` no pacote `br.com.caelum.argentum.modelo`:

```
public class SerieTemporal {

    private final List<Candlestick> candles;

    public SerieTemporal(List<Candlestick> candles) {
        this.candles = candles;
    }

    public Candlestick getCandle(int i) {
        return this.candles.get(i);
    }

    public int getUltimaPosicao() {
        return this.candles.size()  1;
    }
}
```

2) Vamos criar a classe `MediaMovelSimples`, dentro do novo pacote `br.com.caelum.argentum.indicadores`. Essa classe terá o método `calcula` que recebe a posição

a ser calculada e a `SerieTemporal` que proverá os candles. Então, o método devolverá a média simples dos fechamentos dos dois dias anteriores e o atual.

Comece fazendo apenas o cabeçalho desse método:

```
public class MediaMovelSimples {

    public double calcula(int posicao, SerieTemporal serie) {
        return 0;
    }

}
```

A ideia é passarmos para o método calcula a `SerieTemporal` e o dia para o qual queremos calcular a média móvel simples. Por exemplo, se passarmos que queremos a média do dia 6 da série, ele deve calcular a média dos valores de fechamento dos dias 6, 5 e 4 (já que nosso intervalo é de 3 dias).

Como essa é uma lógica um pouco mais complexa, **começaremos essa implementação pelos testes**.

3) Seguindo a ideia do TDD, faremos o teste antes mesmo de implementar a lógica da média. Assim como você já vem fazendo, use o `ctrl + N` -> *JUnit Test Case* para criar a classe de teste `MediaMovelSimplesTest` na *source folder* `src/test/java`, pacote `br.com.caelum.argentum.indicadores`.

   Então, crie um teste para verificar que a média é calculada corretamente para a sequência de fechamentos `1, 2, 3, 4, 3, 4, 5, 4, 3`.

   Note que, para fazer tal teste, será necessário criar uma série temporal com candles cujo fechamento tenha tais valores. Criaremos uma outra classe para auxiliar nesses testes logo em seguida. Por hora, não se preocupe com o erro de compilação da 5a. linha do código abaixo:

```
1  public class MediaMovelSimplesTest {
2
3      @Test
4      public void sequenciaSimplesDeCandles() throws Exception {
5          SerieTemporal serie =
6                  GeradorDeSerie.criaSerie(1, 2, 3, 4, 3, 4, 5, 4, 3);
7          MediaMovelSimples mms = new MediaMovelSimples();
8
9          Assert.assertEquals(2.0, mms.calcula(2, serie), 0.00001);
10         Assert.assertEquals(3.0, mms.calcula(3, serie), 0.00001);
11         Assert.assertEquals(10.0/3, mms.calcula(4, serie), 0.00001);
12         Assert.assertEquals(11.0/3, mms.calcula(5, serie), 0.00001);
13         Assert.assertEquals(4.0, mms.calcula(6, serie), 0.00001);
14         Assert.assertEquals(13.0/3, mms.calcula(7, serie), 0.00001);
15         Assert.assertEquals(4.0, mms.calcula(8, serie), 0.00001);
16     }
17 }
```

4) Ainda é necessário fazer esse código compilar! Note que, pelo que escrevemos, queremos chamar um método estático na classe `GeradorDeSerie` que receberá diversos valores e devolverá a série com Candles respectivas.

   Deixe que o Eclipse o ajude a criar essa classe: use o `ctrl + 1` e deixe que ele crie a classe para você e, então, adicione a ela o método `criaSerie`, usando o recurso de *varargs* do Java para receber diversos `doubles`.

   > **Varargs**
   >
   > A notação `double... valores` (com os três pontinhos mesmo!) que usaremos no método a seguir é indicação do uso de *varargs*. Esse recurso está presente desde o Java 5 e permite que chamemos o método passando de zero a quantos doubles quisermos! Dentro do método, esses argumentos serão interpretados como um array.
   >
   > Varargs vieram para oferecer uma sintaxe mais amigável nesses casos. Antigamente, quando queríamos passar um número variável de parâmetros de um mesmo tipo para um método, era necessário construir um array com esses parâmetros e passá-lo como parâmetro.
   >
   > Leia mais sobre esse recurso em: http://docs.oracle.com/javase/1.5.0/docs/guide/language/varargs.html

   Na classe `GeradorDeSerie`, faça o seguinte método:

   Atenção: não é necessário copiar o JavaDoc.

```
/**
 * Serve para ajudar a fazer os testes.
 *
 * Recebe uma sequência de valores e cria candles com abertura, fechamento,
 * minimo e maximo iguais, mil de volume e data de hoje. Finalmente, devolve
 * tais candles encapsuladas em uma Serie Temporal.
 **/
public static SerieTemporal criaSerie(double... valores) {
    List<Candlestick> candles = new ArrayList<Candlestick>();
    for (double d : valores) {
        candles.add(new Candlestick(d, d, d, d, 1000,
                                    Calendar.getInstance()));
    }
    return new SerieTemporal(candles);
}
```

   Agora que ele compila, rode a classe de teste. **Ele falha**, já que a implementação padrão simplesmente devolve zero!

5) **Volte** à classe principal `MediaMovelSimples` e **implemente** agora a lógica de negócio do método `calcula`, que já existe. O método deve ficar parecido com o que segue:

```
public class MediaMovelSimples {

    public double calcula(int posicao, SerieTemporal serie) {
        double soma = 0.0;
        for (int i = posicao; i > posicao  3; i) {
            Candlestick c = serie.getCandle(i);
            soma += c.getFechamento();
        }
        return soma / 3;
    }
}
```

Repare que iniciamos o `for` com `posicao` e iteramos com `i--`, **retrocedendo** nas posições enquanto elas forem **maiores** que os três dias de intervalo. Isso significa que estamos calculando a média móvel apenas dos últimos 3 dias.

Mais para frente, existe um exercício opcional para parametrizar esse valor.

6) Crie a classe `MediaMovelPonderada` análoga a `MediaMovelSimples`. Essa classe dá peso 3 para o dia atual, peso 2 para o dia anterior e o peso 1 para o dia antes desse. O código interno é muito parecido com o da média móvel simples, só precisamos multiplicar sempre pela quantidade de dias passados.

A implementação do método `calcula` deve ficar bem parecida com isso:

```
public class MediaMovelPonderada {

    public double calcula(int posicao, SerieTemporal serie) {
        double soma = 0.0;
        int peso = 3;

        for (int i = posicao; i > posicao  3; i) {
            Candlestick c = serie.getCandle(i);
            soma += c.getFechamento() * peso;
            peso;
        }
        return soma / 6;
    }
}
```

Repare que o `peso` começa valendo 3, o tamanho do nosso intervalo, para o dia atual e vai reduzindo conforme nos afastamos do dia atual, demonstrando a maior importância dos valores mais recentes.

A divisão por 6 no final é a **soma dos pesos** para o intervalo de 3 dias: (3 + 2 + 1 = 6).

7) Depois a classe `MediaMovelPonderada` deve passar pelo seguinte teste:

```
public class MediaMovelPonderadaTest {
```

```
    @Test
    public void sequenciaSimplesDeCandles() {
        SerieTemporal serie =
                        GeradorDeSerie.criaSerie(1, 2, 3, 4, 5, 6);
        MediaMovelPonderada mmp = new MediaMovelPonderada();

        //ex: calcula(2): 1*1 + 2*2 +3*3 = 14. Divide por 6, da 14/6
        Assert.assertEquals(14.0/6, mmp.calcula(2, serie), 0.00001);
        Assert.assertEquals(20.0/6, mmp.calcula(3, serie), 0.00001);
        Assert.assertEquals(26.0/6, mmp.calcula(4, serie), 0.00001);
        Assert.assertEquals(32.0/6, mmp.calcula(5, serie), 0.00001);
    }
}
```

Rode o teste e veja se passamos!

8) (opcional) Crie um teste de unidade em uma nova classe `SerieTemporalTest`, que verifique se essa classe pode receber uma lista nula. O que não deveria poder.

   Aproveite o momento para pensar quais outros testes poderiam ser feitos para essa classe.

8.5 Refatoração

"Refatoração é uma técnica controlada para reestruturar um trecho de código existente, alterando sua estrutura interna sem modificar seu comportamento externo. Consiste em uma série de pequenas transformações que preservam o comportamento inicial. Cada transformação (chamada de refatoração) reflete em uma pequena mudança, mas uma sequência de transformações pode produzir uma significante reestruturação. Como cada refatoração é pequena, é menos provável que se introduza um erro. Além disso, o sistema continua em pleno funcionamento depois de cada pequena refatoração, reduzindo as chances do sistema ser seriamente danificado durante a reestruturação. -- **Martin Fowler**

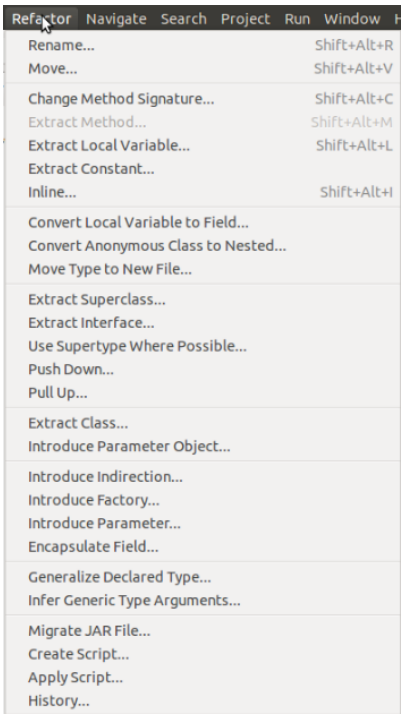
Em outras palavras, refatoração é o processo de modificar um trecho de código já escrito, executando pequenos passos (**baby-steps**) sem modificar o comportamento do sistema. É uma técnica utilizada para melhorar a clareza do código, facilitando a leitura ou melhorando o design do sistema.

Note que para garantir que erros não serão introduzidos nas refatorações, bem como para ter certeza de que o sistema continua se comportando da mesma maneira que antes, a presença de testes é fundamental. Com eles, qualquer erro introduzido será imediatamente apontado, facilitando a correção a cada passo da refatoração imediatamente.

Algumas das refatorações mais recorrentes ganharam nomes que identificam sua utilidade (veremos algumas nas próximas seções). Além disso, **Martin Fowler** escreveu o livro **Refactoring: Improving the Design of Existing Code**, onde descreve em detalhes as principais.

Algumas são tão corriqueiras, que o próprio Eclipse inclui um menu com diversas refatorações que ele é

capaz de fazer por você:

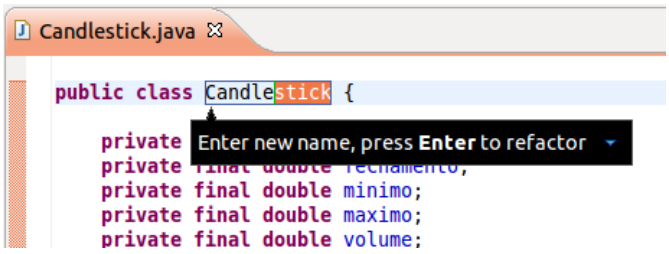


8.6 Exercícios: Primeiras refatorações

1) Temos usado no texto sempre o termo **candle** em vez de **Candlestick**. Essa nomenclatura se propagou no nosso dia-a-dia, tornando-se parte do nosso modelo. Refatore o nome da classe `Candlestick` para `Candle` no pacote `br.com.caelum.argentum.modelo`.

   Use `ctrl + shift + T` para localizar e abrir a classe `Candlestick`.

   Seja no *Package Explorer* ou na classe aberta no editor, coloque o cursor sobre o nome da classe e use o atalho `alt + shift + R`, que renomeia. Esse atalho funciona para classes e também para métodos, variáveis, etc.



2) No método `calcula` da classe `MediaMovelSimples`, temos uma variável do tipo `Candle` que só serve para que peguemos o fechamento desse objeto. Podemos, então, deixar esse método uma linha menor fazendo o *inline* dessa variável!

Na linha 4 do método abaixo, coloque o cursor na variável `c` e use o `alt + shift + I`. (Alternativamente, use `ctrl + 1` *Inline local variable*)

```
1 public double calcula(int posicao, SerieTemporal serie) {
2     double soma = 0.0;
3     for (int i = posicao; i > posicao  3; i) {
4         Candle c = serie.getCandle(i);
5         soma += c.getFechamento();
6     }
7     return soma / 3;
8 }
```

O novo código, então ficará assim:

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

Não esqueça de tirar os imports desnecessários (`ctrl + shift + O`)!

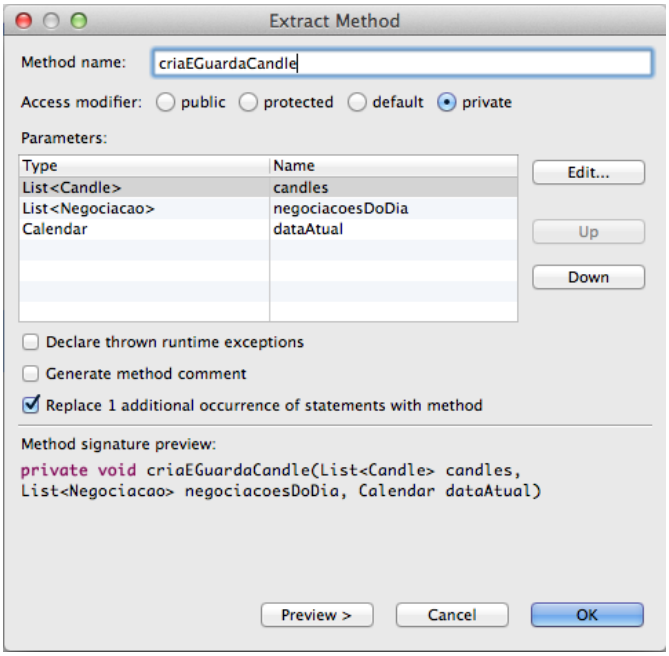
3) Finalmente, abra a classe `CandlestickFactory` e observe o método `constroiCandles`. Escrevemos esse método no capítulo de XML com um algoritmo para separar todos os negócios em vários candles.

   No meio desse código, contudo, há um pequeno bloco de código que se repete duas vezes, dentro e fora do `for`:

   ```
   Candle candleDoDia = constroiCandleParaData(dataAtual, negociacoesDoDia);
   candles.add(candleDoDia);
   ```

   Se encontramos códigos iguais pelo nosso código, as boas práticas de orientação a objetos nos dizem para isolar então essa parte repetida em um novo método que, além de poder ser chamado várias vezes, ainda tem um nome que ajuda a compreender o algoritmo final.

   No Eclipse, podemos aplicar a refatoração **Extract Method**. Basta ir até a classe `CandlestickFactory` e selecionar essas linhas de código dentro do método `constroiCandles` e usar o atalho `alt + shift + M`, nomeando o novo método de `criaEGuardaCandle` e clique em OK.



Repare como a IDE resolve os parâmetros e ainda substitui as chamadas ao código repetido pela chamada ao novo método.

4) No método `constroiCandleParaData`, observe que no bloco de código dentro do `for` invocamos `negociacao.getPreco()` quatro vezes.

```
// ...
  for (Negociacao negociacao : negociacoes) {
      volume += negociacao.getVolume();

      if (negociacao.getPreco() > maximo) {
          maximo = negociacao.getPreco();
      }
      if (negociacao.getPreco() < minimo) {
          minimo = negociacao.getPreco();
      }
  }
  // ...
```

Uma forma de deixar o código mais limpo e evitar chamadas desnecessárias seria extrair para uma variável local.

Para isso, usaremos a refatoração **Extract Local Variable** através do Eclipse. Selecione a primeira chamada para `negociacao.getPreco()` e pressione `alt + shift + L`.

Um box vai aparecer perguntando o nome da variável que será criada, mantenha o nome aconselhado pelo Eclipse e pressione OK.

O Eclipse vai alterar o código de forma que fique parecido com:

```
// ...
  for (Negociacao negociacao : negociacoes) {
      volume += negociacao.getVolume();

      double preco = negociacao.getPreco();
      if (preco > maximo) {
          maximo = preco;
      }
      if (preco < minimo) {
          minimo = preco;
      }
  }
  // ...
```

Observe que o Eclipse automaticamente substituiu as outras chamadas à `negociacoes.getPreco()` por `preco`.

5) (Opcional) Aproveite e mude os nomes das outras classes que usam a palavra `Candlestick`: a Factory e os Testes.

> **Refatorações disponíveis**
>
> Para quem está começando com a usar o Eclipse, a quantidade de atalhos pode assustar. Note, contudo, que os atalhos de refatoração começam sempre com `alt + shift`!
>
> Para ajudar um pouco, comece memorizando apenas o atalho que mostra as refatorações disponíveis dado o trecho de código selecionado: `alt + shift + T`.

8.7 Refatorações maiores

As refatorações que fizemos até agora são bastante simples e, por conta disso, o Eclipse pôde fazer todo o trabalho para nós!

Há, contudo, refatorações bem mais complexas que afetam diversas classes e podem mudar o design da aplicação. Algumas refatorações ainda mais complexas chegam a modificar até a arquitetura do sistema! Mas, quanto mais complexa a mudança para chegar ao resultado final, mais importante é **quebrar essa refatoração em pedaços pequenos e rodar os testes a cada passo**.

Nos próximos capítulos faremos refatorações que flexibilizam nosso sistema e tornam muito mais fácil trabalhar com os indicadores técnicos que vimos mais cedo.

Exemplos de refatorações maiores são:

- Adicionar uma camada a um sistema;
- Tirar uma camada do sistema;

- Trocar uma implementação doméstica por uma biblioteca;
- Extrair uma biblioteca de dentro de um projeto;
- etc...

8.8 Discussão em aula: quando refatorar?

Capítulo 9

Gráficos interativos com Primefaces

"A única pessoa educada é aquela que aprendeu a aprender e a mudar."
– Carl Rogers

9.1 Por que usar gráficos?

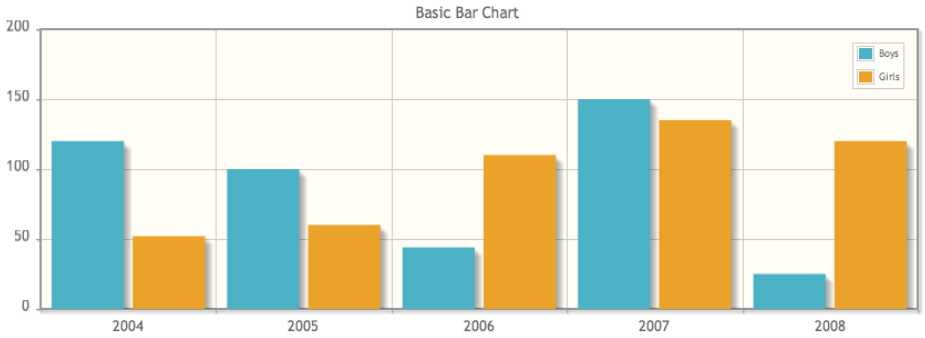
Nossa aplicação apresenta os dados das negociações tabularmente através do componente `p:dataTable`. Essa forma de mostrar informações é interessante para analisarmos dados um a um, mas não ajudam muito quando queremos ter uma ideia do que acontece com dados coletivamente.

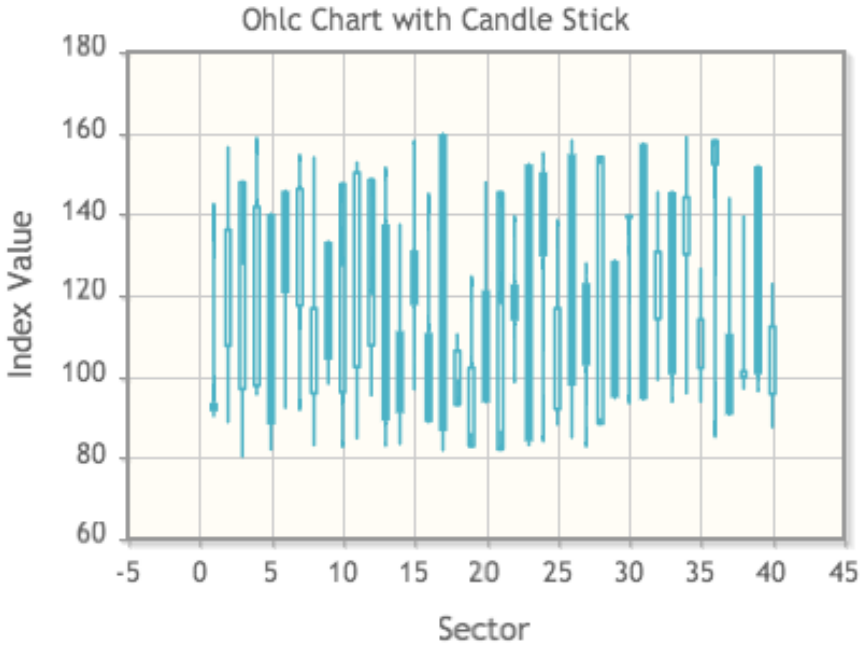
Gráficos comprovadamente ajudam no entendimento mais abrangente dos dados e são mais fáceis de analisar do que números dentro de uma tabela. É simples reconhecer padrões de imagens, por exemplo na análise técnica de valores da bolsa.

Para o projeto `Argentum`, apresentaremos os valores da `SerieTemporal` em um gráfico de linha, aplicando algum indicador como abertura ou fechamento. Continuaremos com a biblioteca Primefaces que já vem com suporte para vários tipos de gráficos.

Exemplos de gráficos

O Primefaces já possui diversos componentes para gráficos: é possível utilizá-lo para desenhar gráficos de linha, de barra, de pizza, de área, gráficos para atualização dinâmica e até para Candles, entre outros. Também é possível exportar e animar gráficos.





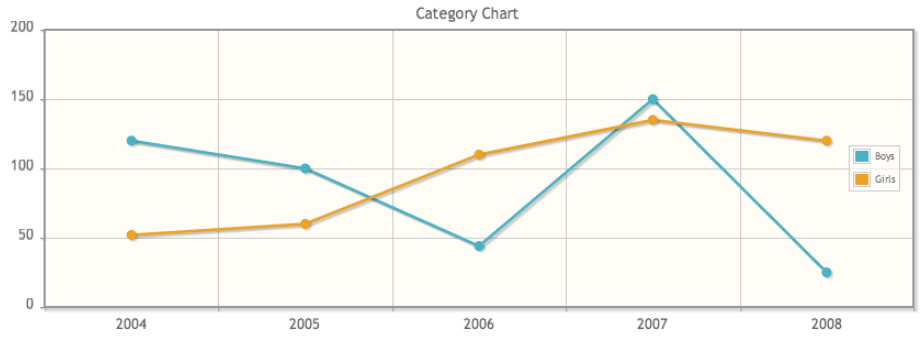
9.2 Gráficos com o Primefaces

Vamos usar o Primefaces para gerar um gráfico que mostra a evolução dos valores da série.

Nosso projeto está configurado e já podemos decidir qual gráfico utilizar. Para facilitar a decisão e ao mesmo tempo ver as possibilidades e tipos de gráficos disponíveis, o showcase do Primefaces nos ajudará muito:

http://www.primefaces.org/showcase/index.xhtml

Nele encontramos o resultado final e também o código utilizado para a renderização. Vamos programar usando o componente `p:chart` que deve mostrar os valores de abertura ou de fechamento da `SerieTemporal`.



O uso de componente é simples, veja o código de exemplo do showcase:

```
<p:chart type=line model=#{chartView.lineModel2} style=height:300px;/>
```

Podemos ver que o componente recebe os dados (`model`) através da *Expression Language* que chama o *Managed Bean* `#{chartView.lineModel2}`.

9.3 Documentação

A documentação do Primefaces está disponível na internet e na forma de um guia do usuário em PDF. Ela fará parte do dia-a-dia do desenvolvedor, que a consultará sempre que necessário:

http://www.primefaces.org/documentation.html

Também há o tradicional Javadoc disponível em: http://www.primefaces.org/docs/api/5.1/. Devemos usar ambos para descobrir funcionalidades e propriedades dos componentes.

No showcase também há um exemplo do uso do *Managed Bean*, porém para maiores informações é fundamental ter acesso ao Javadoc e à documentação da biblioteca para saber quais classes, atributos e métodos utilizar.

JSF e CSS

O Primefaces ajuda o desenvolvedor através de seu suporte a temas, mas nem sempre ele é suficiente, inclusive quando há uma equipe de designers responsável em determinar a apresentação da aplicação.

Apesar de trabalharmos com componentes, no final o que é enviado para o cliente é puro HTML. Assim, quem conhece CSS pode aplicar estilos para sobrescrever o visual da aplicação.

A Caelum oferece o curso **Desenvolvimento Web com HTML, CSS e JavaScript**, para aqueles que querem aprender a criar interfaces Web com experiência rica do usuário, estruturação correta e otimizações SEO.

Aplicando ao nosso projeto

Para mostrarmos o gráfico no Argentum, usaremos a tag do Primefaces vista acima:

```
<p:chart type=line />
```

Da forma como está, no entanto, não há nada a ser mostrado no gráfico. Ainda falta indicarmos para o componente que os dados, o modelo do gráfico, será disponibilizado por nosso `ManagedBean`. Em outras palavras, faltou indicarmos que o `model` desse gráfico será produzido em `argentumBean.modeloGrafico`.

Nossa adição ao `index.xhtml` será, portanto:

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

9.4 Definição do modelo do gráfico

Já temos uma noção de como renderizar gráficos através de componentes do Primefaces. O desenvolvedor não precisa se preocupar como detalhes de JavaScript, imagem ou animações. Tudo isso é encapsulado no próprio componente, seguindo boas práticas do mundo orientado a objetos.

Apenas, será necessário informar ao componente quais são os dados a serem plotados no gráfico em questão. Esses dados representam o modelo do gráfico e devem ser disponibilizados como o `model` para o `p:chart` em um objeto do tipo `org.primefaces.model.chart.ChartModel`.

Essa é a classe principal do modelo e há classes filhas especializadas dela como `LineChartModel`, `PieChartModel` ou `BubbleChartModel`. No JavaDoc podemos ver todas as filhas da `ChartModel` e ainda no *showcase* é possível ver o `ManagedBean` responsável por cada modelo de gráfico. Assim, saberemos qual delas devemos usar de acordo com o gráfico escolhido:

Javadoc: http://www.primefaces.org/docs/api/5.1/org/primefaces/model/chart/ChartModel.html

No nosso projeto, utilizaremos o `LineChartModel`, já que queremos plotar pontos em um gráfico de linha. Um `LineChartModel` recebe uma ou mais `ChartSeries` e cada `ChartSeries` representa uma linha no gráfico do componente `p:chart`. Uma `ChartSeries`, por sua vez, contém todos os pontos de uma linha do gráfico, isto é, os valores X e Y que serão ligados pela linha do gráfico.

Vejo como fica o código fácil de usar:

```
ChartSeries serieGrafico = new ChartSeries(Abertura);
serieGrafico.set(dia 1, 20.9);
serieGrafico.set(dia 2, 25.1);
serieGrafico.set(dia 3, 22.6);
serieGrafico.set(dia 4, 24.6);
```

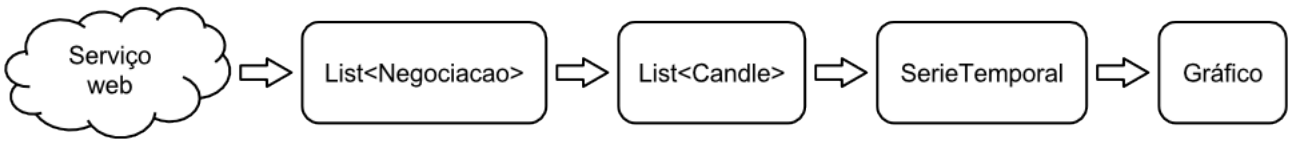
```
LineChartModel modeloGrafico = new LineChartModel();
modeloGrafico.addSeries(serieGrafico);
```

Uma `ChartSeries` recebe no construtor a legenda da linha que ela representa (*label*) e, através do método `set`, passamos os valores de cada ponto nos eixos horizontal e vertical, respectivamente. No exemplo acima, os valores colocados são fixos, mas na nossa implementação do Argentum, é claro, iteraremos pela `SeriaTemporal` calculando os indicadores sobre ela.

Isto é, uma vez que temos a `SerieTemporal`, nosso código para plotar a média móvel simples do fechamento em uma `ChartSeries` será semelhante a este:

```
SerieTemporal serie = ...
ChartSeries chartSeries = new ChartSeries(MMS do Fechamento);
MediaMovelSimples indicador = new MediaMovelSimples();
for (int i = 2; i < serie.getUltimaPosicao(); i++) {
    double valor = indicador.calcula(i, serie);
    chartSeries.set(i, valor);
}

LineChartModel modeloGrafico = new LineChartModel();
modeloGrafico.addSeries(chartSeries);
```

Conseguir a série temporal é um problema pelo qual já passamos antes. Relembre que a `SerieTemporal` é apenas um *wrapper* de uma lista de `Candles`. E a lista de `Candles` é gerada pelo `CandlestickFactory` resumindo uma lista com muitas `Negociacoes`.



Se procurarmos o local no nosso código onde pegamos a lista de negociações do web service, vamos notar que isso acontece no construtor da `ArgentumBean`. Seu código completo ficaria assim;

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

    ChartSeries chartSeries = new ChartSeries(MMS do Fechamento);
    MediaMovelSimples indicador = new MediaMovelSimples();
    for (int i = 2; i < serie.getUltimaPosicao(); i++) {
        double valor = indicador.calcula(i, serie);
        chartSeries.set(i, valor);
```

```
    }

    this.modeloGrafico = new LineChartModel();
    modeloGrafico.addSeries(chartSeries);
}
```

Note, no entanto, que esse trecho de código está com responsabilidades demais: ele busca as negociações no *web service*, cria a série temporal, plota um indicador e disponibiliza o modelo do gráfico. E, pior ainda, esse código ainda está no construtor do `ArgentumBean`!

Com todo esse código no construtor do bean, teríamos um código mais sujo, pouco coeso e muito difícil de testar. O que acontece aqui é que não estamos separando responsabilidades o bastante e nem encapsulando a lógica de geração de gráfico corretamente.

9.5 Isolando a API do Primefaces: baixo acoplamento

O que acontecerá se precisarmos criar dois gráficos de indicadores diferentes? Vamos copiar e colar todo aquele código e modificar apenas as partes que mudam? E se precisarmos alterar algo na geração do modelo? Essas mudanças não serão fáceis se tivermos o código todo espalhado pelo nosso programa.

Os princípios de orientação a objetos e as boas práticas de programação vêm ao nosso socorro aqui. Vamos **encapsular** a maneira como o modelo do gráfico é criado na classe `GeradorModeloGrafico`.

Essa classe deve ser capaz de gerar o `ChartModel` para nosso gráfico de linhas, com os pontos plotados pelos indicadores com base nos valores de uma série temporal. Isto é, nosso `GeradorModeloGrafico` precisa receber a `SerieTemporal` sobre a qual plotar os indicadores.

Se quisermos restringir o gráfico a um período menor do que o devolvido pelo *web service*, é uma boa recebermos as posições de início e fim que devem ser plotadas. Esse intervalo também serve para que não tentemos calcular a média da posição zero, por exemplo -- note que, como a média é dos três últimos valores, tentaríamos pegar as posições `0`, `-1` e `-2`, o que causaria uma `IndexOutOfBoundsException`.

Como essas informações são fundamentais para qualquer gráfico que plotemos, o gerador do modelo do gráfico receberá tais informações no construtor. Além delas, teremos também o objeto do `LineChartModel`, que guardará as informações do gráfico.

```
public class GeradorModeloGrafico {

    private SerieTemporal serie;
    private int comeco;
    private int fim;
    private LineChartModel modeloGrafico;

    public GeradorModeloGrafico(SerieTemporal serie, int comeco, int fim) {
```

```
        this.serie = serie;
        this.comeco = comeco;
        this.fim = fim;
        this.modeloGrafico = new LineChartModel();
    }

}
```

E, quem for usar essa classe, fará:

```
SerieTemporal serie = //...
GeradorModeloGrafico g = new GeradorModeloGrafico(serie, inicio, fim);
```

Repare como o código que usa o `GeradorModeloGrafico` não possui nada que o ligue ao `ChartModel` especificamente. O dia em que precisarmos mudar o gráfico a ser plotado ou mesmo mudar a tecnologia que gerará o gráfico, só precisaremos alterar a classe `GeradorModeloGrafico`. Relembre o conceito: esse é o poder do **encapsulamento**!

E nossa classe não se limitará a isso: ela encapsulará tudo o que for relacionado ao gráfico. Por exemplo, para criar o gráfico, precisamos ainda de um método que plote os pontos calculados por um indicador, como o `plotaMediaMovelSimples` abaixo. Ele passa por cada posição do `comeco` ao `fim` da `serie`, chamando o cálculo da média móvel simples.

```
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);
}
```

O método `plotaMediaMovelSimples` cria um objeto da `MediaMovelSimples` e varre a `SerieTemporal` recebida para calcular o conjunto de dados para o modelo do gráfico.

Por fim, ainda precisaremos de um método `getModeloGrafico` que devolverá o modelo para o `ArgentumBean`, já que o componente `p:chart` é que precisará desse objeto preenchido. Repare que o retorno é do tipo `ChartModel`, super classe do `LineChartModel`. É boa prática deixar nossa classe a mais genérica possível para funcionar com qualquer tipo de método.

Effective Java: refira-se a objetos pela sua interface

O item 34 do *Effective Java* discorre sobre preferir referenciar objetos pela sua interface, em vez de pelo seu próprio exato, sempre que este for relevante para o restante do sistema.

O livro também menciona que, na falta de uma interface adequada, uma superclasse pode ser utilizada, o que é nosso caso com a superclasse abstrata `ChartModel`. Referir-se a um objeto da forma mais genérica possível é uma boa ideia, já que isso faz com que futuras mudanças limitem-se ao máximo à classe que as encapsula.

Veja como fica o programa dentro do *ArgentumBean* e note que essa classe apenas delega para a outra toda a parte de criação do gráfico:

```
public class ArgentumBean {

    private List<Negociacao> negociacoes;
    private String indicadorBase;
    private String media;
    private ChartModel modeloGrafico;

    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.plotaMediaMovelSimples();
        this.modeloGrafico = geradorGrafico.getModeloGrafico();
    }

    public List<Negociacao> getNegociacoes() {
        return negociacoes;
    }

    public ChartModel getModeloGrafico() {
        return modeloGrafico;
    }
}
```

Note que, para quem usa o `GeradorModeloGrafico` nem dá para saber como ele gera o modelo. É um código **encapsulado**, **flexível**, **pouco acoplado** e **elegante**: usa boas práticas da Orientação a Objetos.

9.6 Para saber mais: Design Patterns Factory Method e Builder

Dois famosos design patterns do GoF são o **Factory Method** e o **Builder** -- e estamos usando ambos. Eles são chamados de **padrões de criação** (**creational patterns**), pois nos ajudam a criar objetos complicados.

A *factory* é usada pelo `GeradorDeSerie` dos testes. A ideia é que criar um objeto `SerieTemporal` diretamente é complicado. Então criaram um *método de fábrica* que encapsula essas complicações e já devolve o objeto prontinho para uso.

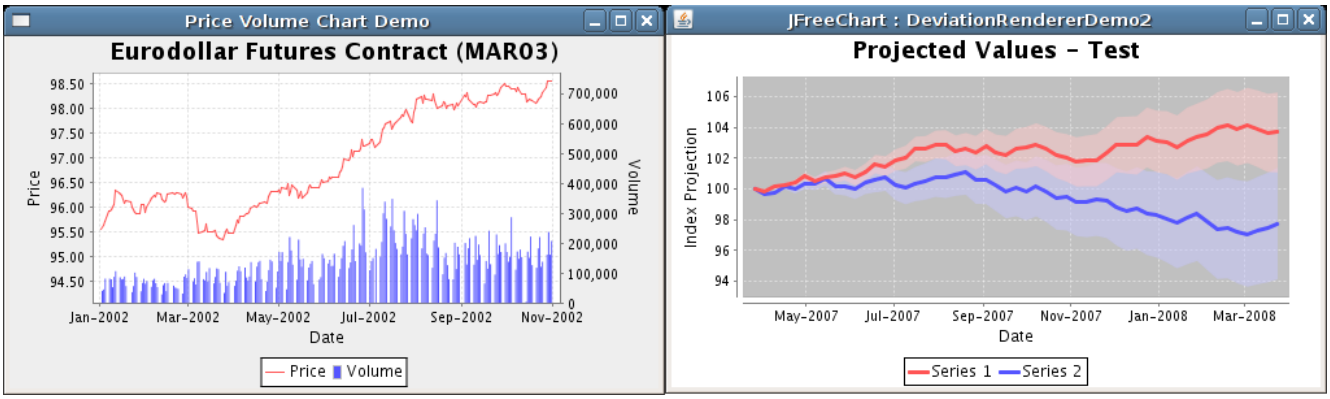
O padrão Builder é o que estamos usando na classe `GeradorModeloGrafico`. Queremos encapsular a criação complicada do modelo e que pode mudar depois com o tempo. Entra aí o *objeto construtor* da nossa classe Builder: seu único objetivo é descrever os passos para criação do nosso objeto final (o gráfico) e encapsular a complexidade disso.

Leia mais sobre esses e outros Design Patterns no livro do GoF.

JFreeChart

O **JFreeChart** é uma biblioteca famosa para desenho de gráficos, independente da tecnologia JSF. É um projeto de software livre iniciado em 2000 e que tem ampla aceitação pelo mercado, funcionando até em ambientes antigos que rodam Java 1.3.

Além do fato de ser livre, possui a vantagem de ser bastante robusta e flexível. É possível usá-la para desenhar gráficos de pontos, de barra, de torta, de linha, gráficos financeiros, gantt charts, em 2D ou 3D e muitos outros. Consegue dar saída em JPG, PNG, SVG, EPS e até mesmo exibir em componentes Swing.



O site oficial possui links para download, demos e documentação:

http://www.jfree.org/jfreechart/

Existe um livro oficial do JFreeChart escrito pelos desenvolvedores com exemplos e explicações detalhadas de vários gráficos diferentes. Ele é pago e pode ser obtido no site oficial. Além disso, há muitos tutoriais gratuitos na internet.

9.7 Exercícios: Gráficos com Primefaces

1) Dentro da pasta `src/main/java` crie a classe `GeradorModeloGrafico` no pacote `br.com.caelum.argentum.grafico` para encapsular a criação do modelo do gráfico.

   Use os recursos do Eclipse para escrever esse código! Abuse do `ctrl + espaço`, do `ctrl + 1` e do `ctrl + shift + O`.

```
public class GeradorModeloGrafico {
    // atributos: serie, comeco, fim e grafico
    // (todos gerados com ctrl + 1, conforme você criar o construtor)
    // importe as classes com ctrl + shift + O
```

```
    public GeradorModeloGrafico(SerieTemporal serie, int comeco, int fim) {
        this.serie = serie;
        this.comeco = comeco;
        this.fim = fim;
        this.modeloGrafico = new LineChartModel();
    }

    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 ChartModel getModeloGrafico() {
        return this.modeloGrafico;
    }
}
```

2) Agora que já temos uma classe que criará o modelo do gráfico, falta usá-la para renderizar o gráfico na tela usando a tag do Primefaces. Abra a página `index.xhtml` que se encontra na pasta `WebContent`.

   Na página procure o componente `h:body`. Logo depois da abertura da tag `h:body` e antes do `h:form` da lista de negociações, adicione o componente `p:chart`:

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

3) Na classe `ArgentumBean`, procure o construtor. É nele que criaremos uma `SerieTemporal` baseada na lista de negociações do Web Service. Depois usaremos o `GeradorModeloGrafico` para criar o modelo gráfico:

```
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.plotaMediaMovelSimples();
    this.modeloGrafico = geradorGrafico.getModeloGrafico();
}
```

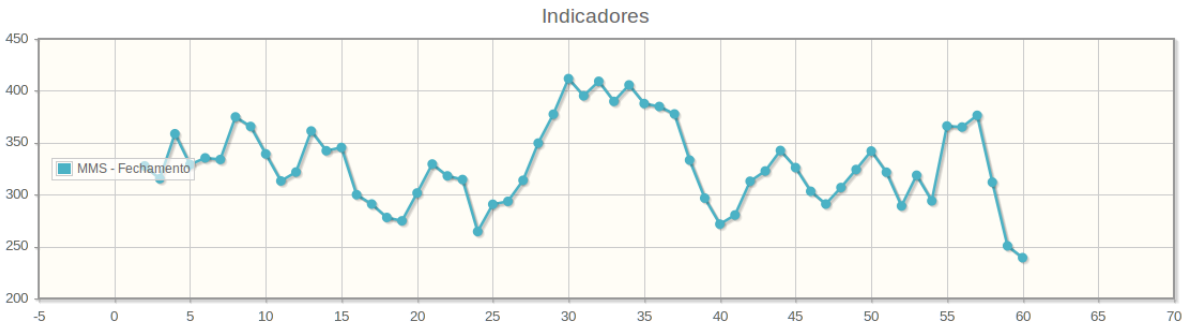
4) Você deve estar vendo um erro de compilação na última linha do seu construtor. Ele acontece porque faltou criarmos um novo atributo que representa o modelo do gráfico. Use o **ctrl + 1** para criar o atributo e, ao mesmo tempo, fazer o import da classe.

```
private ChartModel modeloGrafico;
```

Gere também o *getter* para este atributo, para que o componente `p:chart` consiga pegar o modelo do gráfico quando necessitar.

Para isso, na classe `ArgentumBean`, escreva `getM` e aperte **ctrl + espaço**. O Eclipse sugerirá o getter para o atributo `modeloGrafico`.

Salve o projeto. Reinicie o Tomcat e acesse em seu navegador: http://localhost:8080/fj22-argentum-web/index.xhtml



1 2 3 4 5 6 7 8 9 10

Preço	Quantidade	Volume	Data
246.93	14	3457.02	21/11/2013
376.8	22	8289.6	21/11/2013
411.78	24	9882.72	21/11/2013

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**.

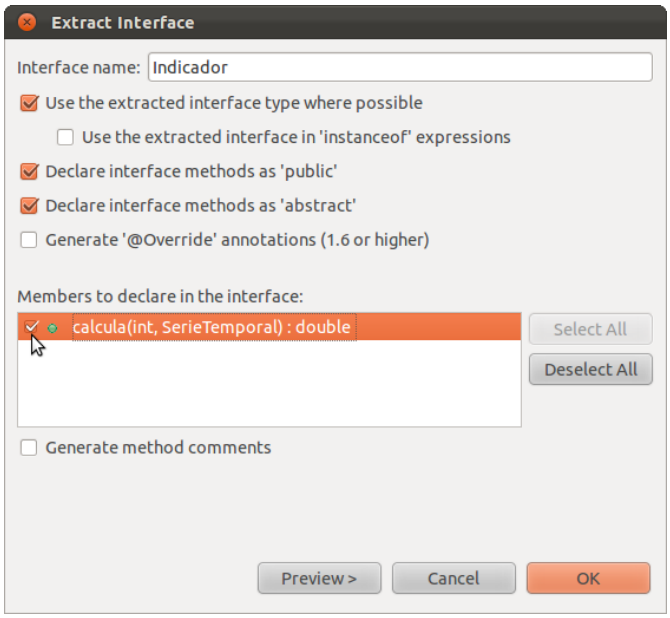
> **Design Patterns**
>
> **Design Patterns** são aquelas soluções catalogadas para problemas clássicos de orientação a objetos, como este que temos no momento: encapsular e ter flexibilidade.
>
> A fábrica de `Candles` apresentada em outro capítulo e o método que nos auxilia a criar séries para testes 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`.

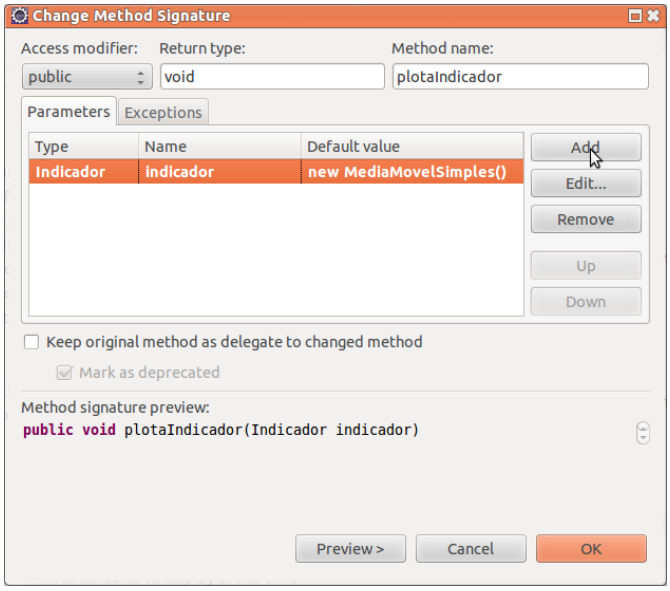
   No final, faltará preencher apenas a linha destacada:

```
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

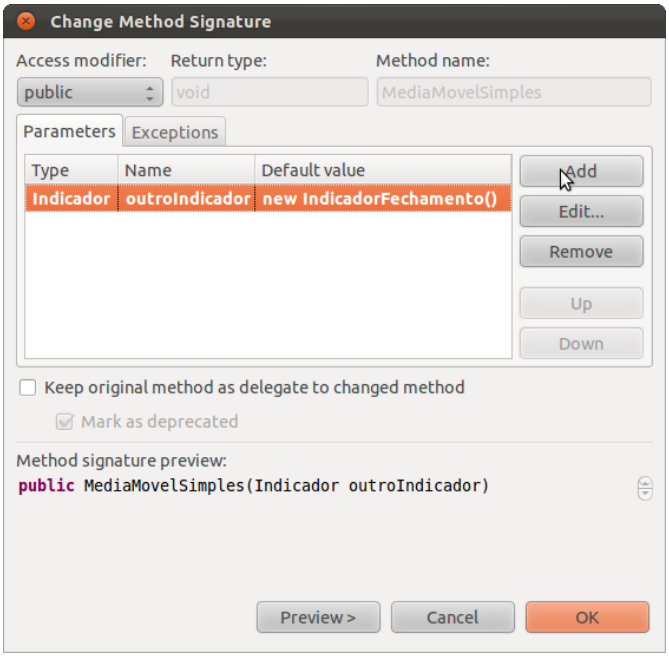
1) Faremos uma grande mudança agora: nossas médias devem receber como argumento um outro indicador, formando o *design pattern* Decorator, como visto na explicação.

   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.

Capítulo 11

A API de Reflection

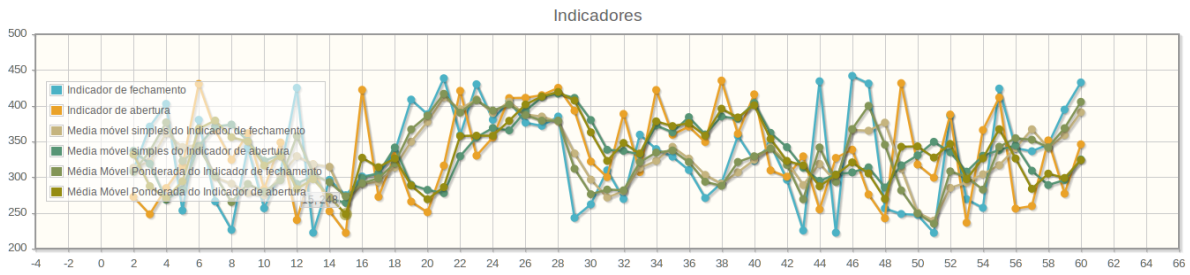
Até agora, ao acessar a aplicação pelo navegador, o gráfico e a tabela de negociações serão gerados automaticamente. Não há nenhuma forma de personalizar o gráfico, como por exemplo a opção de ver apenas um dos indicadores por vez.

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.

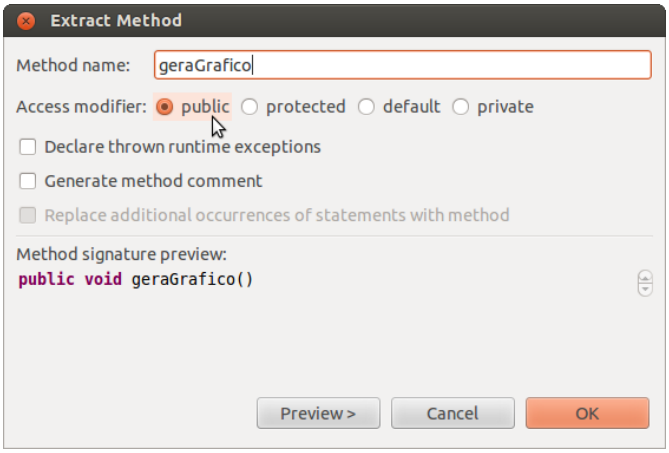
```
<p:commandButton value=Gerar gráfico update=:grafico
                        action=#{argentumBean.geraGrafico}/>
    </h:panelGrid>
</h:form>

<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 `ifs` 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 12

Apêndice Testes de interface com Selenium

"Além da magia negra, há apenas automação e mecanização"
– Federico Garcia Lorca

12.1 Alterando o título do gráfico

Olhando para o nosso gráfico reparamos que ele possui como título o texto *Indicadores*, sendo que este não pode ser alterado. Seria interessante permitir que o usuário pudesse alterar esse título, colocando outro texto de sua preferência.

Podemos fazer isso adicionando mais um campo no formulário, sendo que este será um campo de texto, e para isso devemos utilizar a tag `<h:inputText>` do JSF, ou então a tag `<p:inputText>` do PrimeFaces:

```
<h:outputLabel value=Título Gráfico: />
<p:inputText id=titulo value=#{argentumBean.titulo} />
```

Também será necessário adicionar mais um atributo do tipo `String` à classe `ArgentumBean`, bem como seu getter e setter.

Outra mudança que precisamos fazer é passar o título digitado pelo usuário para o `GeradorModeloGrafico`. Podemos passar esta informação pelo construtor, ao instanciarmos o objeto `GeradorModeloGrafico`:

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

    GeradorModeloGrafico geradorGrafico = new GeradorModeloGrafico(
            serie, 2, serie.getUltimaPosicao(), titulo);
```

```
    geradorGrafico.plotaIndicador(defineIndicador());
    this.modeloGrafico = geradorGrafico.getModeloGrafico();
}
```

E no construtor da classe `GeradorModeloGrafico` recebemos o título e o guardamos em um novo atributo:

```
public class GeradorModeloGrafico {

    private SerieTemporal serie;
    private int comeco;
    private int fim;
    private LineChartModel modeloGrafico;
    private String tituloGrafico;

    public GeradorModeloGrafico(SerieTemporal serie, int comeco, int fim,
            String tituloGrafico) {
        this.serie = serie;
        this.comeco = comeco;
        this.fim = fim;
        this.tituloGrafico = tituloGrafico;
        this.modeloGrafico = new LineChartModel();
    }

    //restante do código
```

E por fim no método `plotaIndicador` devemos atribuir o título do gráfico ao objeto `LineChartModel`:

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

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

12.2 Validação com JSF

Podemos agora definir um título para cada geração de gráfico, mas temos um problema: nada impede que o usuário deixe o título em branco, o que não faz sentido para nossa aplicação.

Preenchimento obrigatório através do atributo required

O JSF vem nos socorrer. Se quisermos tornar o preenchimento do título obrigatório, basta adicionarmos o atributo `required="true"` no componente `p:inputText`. A mesma coisa é válida para o `h:inputText` da especificação.

```
<p:inputText id=titulo value=#{argentumBean.titulo} required=true/>
```

Quando o formulário for submetido com o título em branco, o JSF automaticamente gerará uma mensagem avisando ao usuário que o campo deve ser preenchido. O problema é que ainda não definimos qual componente será utilizado para mostrar essa mensagem.

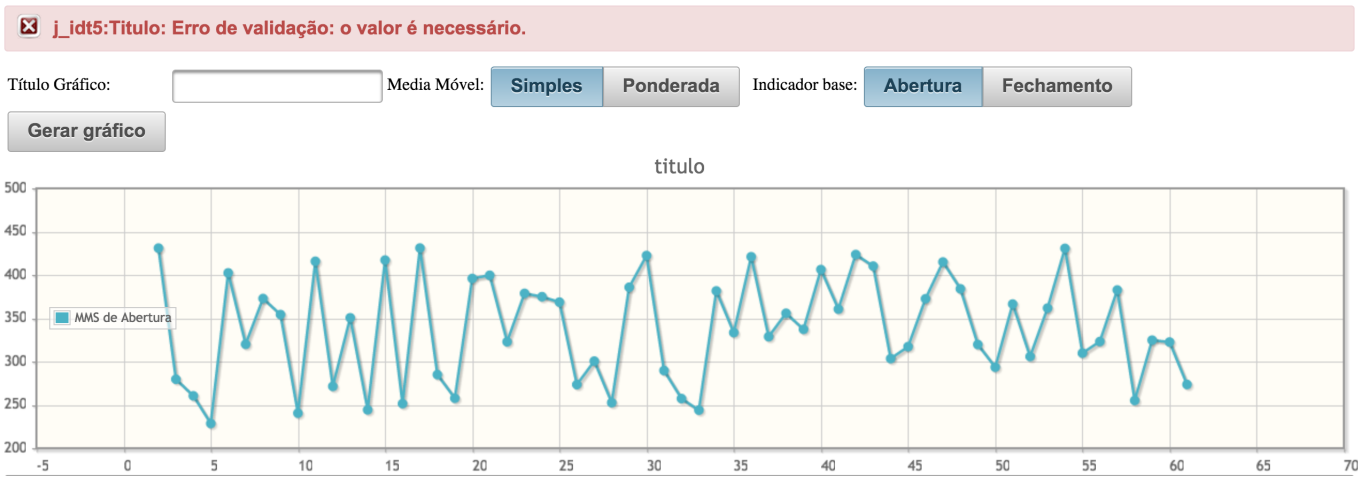
Exibindo mensagens com p:messages

O PrimeFaces possui o componente `p:messages` que exibe para o usuário as mensagens geradas pelo JSF, em nosso caso, as de validação. O objetivo aqui é fazer com que a mensagem de preenchimento obrigatório, inclusive outras mensagens que possam ser geradas, seja exibida para o usuário:

```
....
<h:form>
  <p:messages autoUpdate=true />
....
```

O atibuto `autoUpdate="true"` indica ao PrimeFaces que este componente deve ser atualizado automaticamente em caso de requisições `AJAX`.

Pronto, quando o formulário for submetido com o campo título em branco o JSF exibirá automaticamente a mensagem de validação:



12.3 Introdução aos testes de aceitação

Nos capítulos anteriores fizemos uso dos testes de unidade, será que eles são suficientes para garantir que a aplicação funcione da maneira esperada? Em um projeto web, uma das partes mais importantes para o funcionamento desejado é a parte de apresentação, já que é a partir dela que toda a comunicação do cliente com o nosso sistema será feita.

Verificar se as funcionalidades de um sistema estão se comportando corretamente sem que tenhamos de testar manualmente a aplicação, abrindo um navegador, navegando por ele e visualizando os resultados são tarefas que são realizadas pelos *Acceptance Testing* (ou Testes de Aceitação).

Temos que testar o nosso código, mas em uma aplicação web fica difícil testar a camada de apresentação, o resultado final. Como testar a compatibilidade entre browsers diferentes de maneira automatizada? Em geral, como testar se a página renderizada tem o resultado desejado?

Para isso ser possível, existem algumas ferramentas que permitem tal trabalho, simulando a interação de um usuário com a aplicação. Esse é o caso do **Selenium** que, com a assistência do **JUnit**, pode facilitar o trabalho de escrever tais testes.

12.4 Como funciona?

Para demonstrar como o Selenium funciona, testaremos se a validação do campo *titulo* funciona corretamente, isto é, o formulário não poderá ser submetido caso ele esteja em branco.

A primeira coisa que uma pessoa faz para testar a aplicação é abrir um browser. Com Selenium, precisamos apenas da linha abaixo:

```
WebDriver driver = new FirefoxDriver();
```

Nesse caso, abrimos o Firefox. A ideia aqui é igual ao mundo JDBC onde o Driver abstrai detalhes sobre o funcionamento do banco de dados. Em nosso caso o driver se preocupa com a comunicação com o navegador.

Há outros drivers disponíveis para Chrome, Internet Explorer, Safari e até para Android ou iPhone. O que precisa mudar é apenas a implementação concreta da interface `WebDriver`, por exemplo:

```
WebDriver driver = new InternetExplorerDriver();
```

HtmlUnitDriver

Há uma outra implementação do `WebDriver` disponível que nem precisa abrir um navegador. Ela se chama `HtmlUnitDriver` e simula o navegador em memória. Isso é muito mais rápido mas não testa a compatibilidade do navegador fidedignamente:

```
WebDriver driver = new HtmlUnitDriver();
```

Essa alternativa é bastante utilizada em ambientes de integração que fazem uso de um cloud, já que não temos disponíveis um browser para teste.

O próximo passo é indicar qual página queremos abrir através do navegador. O driver possui um método `navigate().to()` que recebe a URL:

```
WebDriver driver = new FirefoxDriver();

driver.navigate().to(http://localhost:8080/fj22argentumweb/index.xhtml);
```

Com a página aberta, faremos a inserção de um valor em branco no campo `titulo`. Para isso, precisamos da referência deste elemento em nosso código, o que é feito pelo método `findElement`.

Este método recebe como parâmetro um critério de busca. A classe `By` possui uma série de métodos estáticos. Pesquisaremos pelo `id`:

```
WebDriver driver = new FirefoxDriver();

driver.navigate().to(http://localhost:8080/fj22argentumweb/index.xhtml);

WebElement titulo = driver.findElement(By.id(titulo));
```

Agora que temos o elemento, podemos preenchê-lo com o texto que quisermos através do método *sendKeys*. Em nosso caso, deixaremos o campo em branco passando uma string vazia:

```
WebDriver driver = new FirefoxDriver();

driver.navigate().to(http://localhost:8080/fj22argentumweb/index.xhtml);

WebElement titulo = driver.findElement(By.id(titulo));

titulo.sendKeys();
```

Quando o formulário for submetido e o campo `titulo` estiver em branco, esperamos que o sistema mostre a mensagem *Erro de Validação*.

Antes de testarmos se tudo esta correto, precisamos submeter o formulário. O objeto `WebElement` possui o método `submit` que faz exatamente isso:

```
WebDriver driver = new FirefoxDriver();

driver.navigate().to(http://localhost:8080/fj22argentumweb/index.xhtml);

WebElement titulo = driver.findElement(By.id(titulo));

titulo.sendKeys();
titulo.submit();
```

Para termos certeza de que o sistema esta validando, pediremos ao Selenium que procure pelo texto "Erro de Validação". Primeiro, precisamos evocar o método `getPageSource`, que nos permite procurar por algo no código da página, e logo em seguida o método `contains`, que retorna *true* ou *false*:

```
WebDriver driver = new FirefoxDriver();

driver.navigate().to(http://localhost:8080/fj22argentumweb/index.xhtml);

WebElement titulo = driver.findElement(By.id(titulo));

titulo.sendKeys();
titulo.submit();

boolean existeMensagem = driver.getPageSource().contains(Erro de validação);
```

Por fim, após o formulário ter sido submetido, precisamos saber se tudo saiu conforme o planejado, o que pode ser feito através do método estático `assertTrue` da classe `Assert`, que recebe como parâmetro um boolean que indica a presença ou não do texto procurado pelo método `contains`:

```
WebDriver driver = new FirefoxDriver();

driver.navigate().to(http://localhost:8080/fj22argentumweb/index.xhtml);

WebElement titulo = driver.findElement(By.id(titulo));

titulo.sendKeys();
titulo.submit();

boolean existeMensagem = driver.getPageSource().contains(Erro de validação);

Assert.assertTrue(existeMensagem);
```

Pronto, podemos verificar no próprio Eclipse se o teste passou se ele estiver verde. Se algo der errado, como de costume, a cor vermelha será utilizada.

Podemos usar agora o método `driver.close()` para fechar a janela do navegador.

```
WebDriver driver = new FirefoxDriver();

driver.navigate().to(http://localhost:8080/fj22argentumweb/index.xhtml);

WebElement titulo = driver.findElement(By.id(titulo));

titulo.sendKeys();
titulo.submit();

boolean existeMensagem = driver.getPageSource().contains(Erro de validação);

Assert.assertTrue(existeMensagem);

driver.close();
```

12.5 Trabalhando com diversos testes de aceitação

É comum termos dois ou mais testes de aceitação em nossa bateria de testes. Teríamos então que abrir uma janela do navegador em cada método, e, após os testes, fechá-la. Com isso, estaríamos repetindo muito código! O **Selenium** nos permite fazer isso de uma forma mais fácil. Primeiro, criaremos o método `setUp`, e o anotaremos com `@Before`.

```
@Before
public void setUp() {
    driver = new FirefoxDriver();
}
```

Este método será invocado antes de cada teste, sempre abrindo uma nova janela. Analogamente, existe a anotação `@After`, que indica que o método será invocado após cada teste. Agora, precisamos fechar essas janelas:

```
@After
public void tearDown() {
    driver.close();
}
```

12.6 Para saber mais: Configurando o Selenium em casa

Caso você esteja fazendo esse passo de casa, é preciso baixar algumas JARs para o funcionamento dessa aplicação, usaremos as seguintes versões:

- commons-exec-1.1.jar
- commons-logging-1.1.3.jar
- gson-2.3.jar
- guava-18.0.jar
- httpclient-4.3.4.jar
- httpcore-4.3.2.jar
- selenium-java-2.44.0.jar

Para mais informações, você pode consultar o site http://docs.seleniumhq.org/

12.7 Exercícios: Testes de aceitação com Selenium

Vamos criar nosso primeiro teste com Selenium:

1) Primeiramente devemos alterar nossa página adicionando o campo para o título. Adicione o campo de texto logo após a abertura da tag `<h:panelGrid>` no arquivo `index.xhtml`:

```
<h:panelGrid columns=5>
    <h:outputLabel value=Título Gráfico: />
    <p:inputText id=titulo value=#{argentumBean.titulo} required=true/>
```

2) Como adicionamos mais um campo no formulário, altere a quantidade de colunas para 6 no componente `<h:panelGrid>`:

```
<h:panelGrid columns=6>
```

3) Para facilitar a localização do campo título nos testes, vamos atribuir um `id` ao componente `<h:form>`

```
<h:form id=dadosGrafico>
    <h:panelGrid columns=6>
```

4) Devemos agora alterar a classe `ArgentumBean` adicionando o atributo `titulo`, juntamente como seu getter e setter:

```
@ManagedBean
@ViewScoped
public class ArgentumBean implements Serializable {
```

```
    private String titulo;

    public String getTitulo() {
        return titulo;
    }
    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }

    //restante do código
```

5) Ainda na classe `ArgentumBean` vamos alterar o método `geraGrafico`, passando o título como parâmetro para o construtor do `GeradorModeloGrafico`:

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

    GeradorModeloGrafico geradorGrafico = new GeradorModeloGrafico(serie, 2,
        serie.getUltimaPosicao(), titulo);
    geradorGrafico.plotaIndicador(defineIndicador());
    this.modeloGrafico = geradorGrafico.getModeloGrafico();
}

//restante do código
```

6) Agora na classe `GeradorModeloGrafico` vamos alterar o construtor para receber o título como parâmetro, e atribuí-lo a um novo atributo:

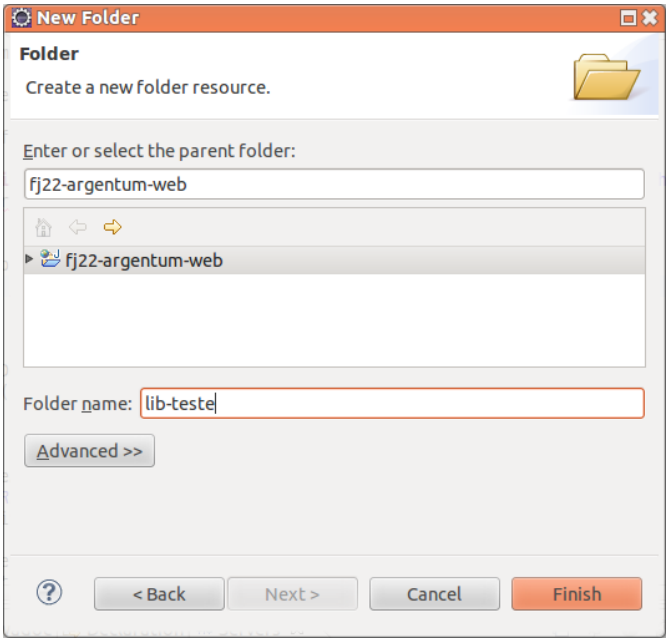
```
public class GeradorModeloGrafico {

private SerieTemporal serie;
private int comeco;
private int fim;
private LineChartModel modeloGrafico;
private String tituloGrafico;

public GeradorModeloGrafico(SerieTemporal serie, int comeco, int fim,
        String tituloGrafico) {
    this.serie = serie;
    this.comeco = comeco;
    this.fim = fim;
    this.tituloGrafico = tituloGrafico;
    this.modeloGrafico = new LineChartModel();
}
```

7) E no método `plotaIndicador` passamos o título como parâmetro no método `setTitle` do objeto

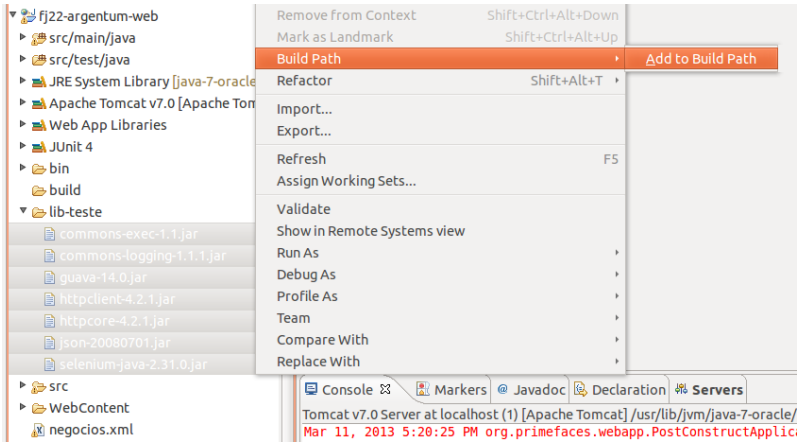
```
modeloGrafico:

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

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

8) Agora vamos colocar os jars do Selenium no nosso projeto, eles se encontram na pasta `Desktop/caelum/22/selenium-jars/`:

Para adicioná-los ao projeto, crie uma nova pasta chamada `lib-teste` na raiz do projeto e copie os jars para dentro dela.



Logo em seguida adicione-os ao *Build Path*:



9) Crie a classe usando **ctrl + N** `Class`, chamada `GeraGraficoTest` no *source folder src/test/java* e dentro do pacote `br.com.caelum.argentum.aceitacao`:

10) A classe terá dois atributos. O primeiro com a URL da página que queremos testar e o segundo o WebDriver, o objeto que nos permite manipular o navegador.

```
public class GeraGraficoTest {

    private static final String URL =
        http://localhost:8080/fj22argentumweb/index.xhtml;

    private WebDriver driver;
}
```

11) Vamos criar o método `testeAoGerarGraficoSemTituloUmaMensagemEhApresentada` e anotá-lo com `@Test` para indicar que ele deve ser chamado quando o teste for executado:

```
public class GeraGraficoTest {

    private static final String URL =
        http://localhost:8080/fj22argentumweb/index.xhtml;

    private WebDriver driver;

    @Test
    public void testeAoGerarGraficoSemTituloUmaMensagemEhApresentada() {
    }
}
```

12) Usaremos o WebDriver para abrir uma nova janela do Firefox e acessar a URL do projeto, e usaremos o método `driver.close()` para fechar a janela

```
public class GeraGraficoTest {
```

```
    private static final String URL =
        http://localhost:8080/fj22argentumweb/index.xhtml;

    private WebDriver driver;

    @Test
    public void testeAoGerarGraficoSemTituloUmaMensagemEhApresentada() {
        driver = new FirefoxDriver();
        driver.navigate().to(URL);

        driver.close();
    }
}
```

13) Estamos testando se a mensagem de erro aparece quando submetemos um formulário com o título. Para isso, primeiro precisamos capturar o elemento do título em um `WebElement`. Como estamos trabalhando com **JSF**, devemos lembrar que ele concatena o *id* do formulário com o id dos inputs. Por conseguinte, devemos procurar o elemento pelo id `dadosGrafico:titulo`

```
public class GeraGraficoTest {

    private static final String URL =
        http://localhost:8080/fj22argentumweb/index.xhtml;

    private WebDriver driver;

    @Test
    public void testeAoGerarGraficoSemTituloUmaMensagemEhApresentada() {
        driver = new FirefoxDriver();
        driver.navigate().to(URL);
        WebElement titulo = driver.findElement(By.id(dadosGrafico:titulo));

        titulo.sendKeys();
        titulo.submit();

        boolean existeMensagem = driver.getPageSource().contains(
            Erro de validação);

        Assert.assertTrue(existeMensagem);

        driver.close();
    }
}
```

14) Vemos que usamos `driver = new FirefoxDriver()` para abrir uma janela (um WebDriver) do navegador, e `driver.close()` para fechar a janela. Caso formos escrever mais testes, precisaremos abrir e fechar

o navegador novamente. Para podermos reaproveitar esse código, podemos colocá-los em blocos separados e usar as anotações `@Before`, para executá-lo antes de cada método, e `@After`, para executá-lo após cada método.

```
public class GeraGraficoTest {

    private static final String URL =
        http://localhost:8080/fj22argentumweb/index.xhtml;

    private WebDriver driver;

    @Before
    public void setUp() {
        driver = new FirefoxDriver();
    }

    @After
    public void tearDown() {
        driver.close();
    }

    @Test
    public void testeAoGerarGraficoSemTituloUmaMensagemEhApresentada() {
        driver.navigate().to(URL);
        WebElement titulo = driver.findElement(By.id(dadosGrafico:titulo));

        titulo.sendKeys();
        titulo.submit();

        boolean existeMensagem = driver.getPageSource().contains(
            Erro de validação);

        Assert.assertTrue(existeMensagem);
    }
}
```

Índice Remissivo