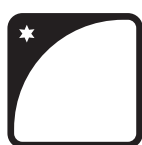


FJ-16

Laboratório Java com Testes,
XML e Design Patterns



Caelum
Ensino e Inovação

www.caelum.com.br

Conheça mais da Caelum.



Cursos Online

www.caelum.com.br/online



Casa do Código

Livros para o programador
www.casadocodigo.com.br



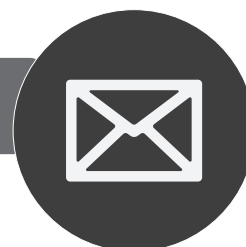
Blog Caelum

blog.caelum.com.br



Newsletter

www.caelum.com.br/newsletter



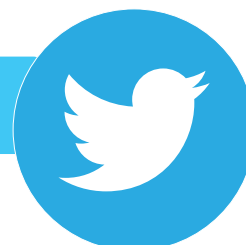
Facebook

www.facebook.com/caelumbr



Twitter

twitter.com/caelum



Conheça alguns de nossos cursos



FJ-11:

Java e Orientação a objetos



FJ-25:

Persistência com JPA2 e Hibernate



FJ-16:

Laboratório Java com Testes, XML e Design Patterns



FJ-26:

Laboratório Web com JSF2 e CDI



FJ-19:

Preparatório para Certificação de Programador Java



FJ-31:

Java EE avançado e Web Services



FJ-21:

Java para Desenvolvimento Web



FJ-91:

Arquitetura e Design de Projetos Java



RR-71:

Desenvolvimento Ágil para Web 2.0 com Ruby on Rails



RR-75:

Ruby e Rails avançados: lidando com problemas do dia a dia

Para mais informações e outros cursos, visite: caelum.com.br/cursos

- ✓ Mais de 8000 alunos treinados;
- ✓ Reconhecida nacionalmente;
- ✓ Conteúdos atualizados para o mercado e para sua carreira;
- ✓ Aulas com metodologia e didática cuidadosamente preparadas;
- ✓ Ativa participação nas comunidades Java, Rails e Scrum;
- ✓ Salas de aula bem equipadas;
- ✓ Instrutores qualificados e experientes;
- ✓ Apostilas disponíveis no site.



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

Esse material é parte integrante do treinamento Laboratório Java com Testes, XML e Design Patterns e distribuído gratuitamente exclusivamente pelo site da Caelum. Todos os direitos são reservados à Caelum. A distribuição, cópia, revenda e utilização para ministrar treinamentos são absolutamente vedadas. Para uso comercial deste material, por favor, consulte a Caelum previamente.

www.caelum.com.br

Sumário

1	Tornando-se um desenvolvedor pragmático	1
1.1	O que é realmente importante?	1
1.2	A importância dos exercícios	2
1.3	Tirando dúvidas e referências	3
1.4	Para onde ir depois?	3
2	O modelo da bolsa de valores, datas e objetos imutáveis	4
2.1	A bolsa de valores	4
2.2	Candlesticks: O Japão e o arroz	5
2.3	O projeto Tail	6
2.4	O projeto Argentum: modelando o sistema	8
2.5	Trabalhando com dinheiro	9
2.6	Palavra chave final	9
2.7	Imutabilidade de objetos	10
2.8	Trabalhando com datas: Date e Calendar	12
2.9	Exercícios: o modelo do Argentum	15
2.10	Resumo diário dos negócios	22
2.11	Exercícios: fábrica de Candlestick	24
2.12	Exercícios opcionais	26
3	Testes Automatizados	29
3.1	Nosso código está funcionando corretamente?	29
3.2	Exercícios: testando nosso modelo sem frameworks	29
3.3	Definindo melhor o sistema e descobrindo mais bugs	32
3.4	Testes de Unidade	32
3.5	JUnit	33
3.6	Anotações	35
3.7	JUnit4, convenções e anotação	35
3.8	Exercícios: migrando os testes do main para JUnit	37
3.9	Vale a pena testar classes de modelo?	42
3.10	Exercícios: novos testes	43
3.11	Para saber mais: Import Estático	48
3.12	Mais exercícios opcionais	49
3.13	Discussão em aula: testes são importantes?	49
4	Trabalhando com XML	50
4.1	Os dados da bolsa de valores	50
4.2	XML	51
4.3	Lendo XML com Java de maneira difícil, o SAX	52

4.4	XStream	55
4.5	Exercícios: Lendo o XML	58
4.6	Separando as candles	61
4.7	Test Driven Design - TDD	62
4.8	Exercícios: Separando os candles	63
4.9	Exercícios opcionais	67
4.10	Discussão em aula: Onde usar XML e o abuso do mesmo	69
5	Interfaces gráficas com Swing	70
5.1	Interfaces gráficas em Java	70
5.2	Portabilidade	71
5.3	Look And Feel	71
5.4	Componentes	72
5.5	Começando com Swing - Mensagens	72
5.6	Exercícios: Escolhendo o XML com JFileChooser	73
5.7	Componentes: JFrame, JPanel e JButton	74
5.8	O design pattern Composite: Component e Container	75
5.9	Tratando eventos	76
5.10	Classes internas e anônimas	77
5.11	Exercícios: nossa primeira tela	78
5.12	JTable	82
5.13	Implementando um TableModel	82
5.14	Exercícios: Tabela	84
5.15	Exercícios opcionais: melhorando a apresentação	88
5.16	Para saber mais	89
5.17	Discussão em sala de aula: Listeners como classes top level, internas ou anônimas?	90
6	Refatoração: os Indicadores da bolsa	91
6.1	Análise Técnica da bolsa de valores	91
6.2	Indicadores Técnicos	92
6.3	As médias móveis	93
6.4	Exercícios: criando indicadores	96
6.5	Refatoração	100
6.6	Exercícios: Primeiras refatorações	101
6.7	Refatorações maiores	103
6.8	Discussão em aula: quando refatorar?	104
7	Gráficos com JFreeChart	105
7.1	JFreeChart	105
7.2	Utilizando o JFreeChart	106
7.3	Isolando a API do JFreeChart: baixo acoplamento	108
7.4	Para saber mais: Design Patterns Factory Method e Builder	110
7.5	Exercícios: JFreeChart	111

7.6	Exercícios opcionais	113
7.7	Nossos indicadores e o design pattern Strategy	114
7.8	Exercícios: refatorando para uma interface e usando bem os testes	116
7.9	Exercícios opcionais	119
8	Mais Swing: layout managers, mais componentes e detalhes	121
8.1	Gerenciadores de Layout	121
8.2	Layout managers mais famosos	123
8.3	Exercícios: usando layout managers	124
8.4	Integrando JFreeChart	127
8.5	Exercícios: completando a tela da nossa aplicação	127
8.6	Indicadores mais Elaborados e o Design Pattern Decorator	130
8.7	Exercícios: Indicadores mais espertos e o Design Pattern Decorator	132
8.8	Discussão em sala de aula: uso de IDEs para montar a tela	134
9	Reflection e Annotations	135
9.1	Por que Reflection?	135
9.2	Class, Field e Method	136
9.3	Usando anotações	137
9.4	Usar JTables é difícil	138
9.5	Usando bem anotações	139
9.6	Criando sua própria anotação	141
9.7	Exercícios: ArgentumTableModel	142
9.8	Exercícios opcionais: nomes das colunas	144
9.9	Para saber mais: Formatter, printf e String.format	145
9.10	Para saber mais: parâmetros opcionais	146
9.11	Exercícios opcionais: formatações na tabela	147
9.12	Discussão em sala de aula: quando usar reflection, anotações e interfaces	148
10	Apêndice: O processo de Build: Ant e Maven	149
10.1	O processo de build	149
10.2	O Ant	149
10.3	Exercícios com Ant	150
10.4	O Maven	151
10.5	O Project Object Model	152
10.6	Plugins, goals e phases	153
10.7	Exercícios: build com o Maven	153
10.8	Uso dentro do Eclipse	157
10.9	Discussão em sala de aula: IDE, ant ou Maven?	157
11	Apêndice - Mais swing e recursos avançados	158
11.1	Input de dados formatados: Datas	158
11.2	Exercícios opcionais: filtrando por data	159

11.3	Para saber mais: barra de menu	162
11.4	Exercício: escolhendo indicadores para o gráfico	163
11.5	Dificuldades com Threads e concorrência	165
11.6	SwingWorker	166
11.7	Exercícios: pesquisando mais na API	167
12	Apêndice - Logging com Log4j	168
12.1	Usando logs - LOG4J	168
12.2	Níveis de logs	169
12.3	Appenders e layout	169
12.4	Exercícios: Adicionando logging com Log4J	170
12.5	O sl4j	171
	Índice Remissivo	171

Versão: 15.3.22

CAPÍTULO 1

Tornando-se um desenvolvedor pragmático

“Na maioria dos casos, as pessoas, inclusive os fascínoras, são muito mais ingênuas e simples do que costumamos achar. Aliás, nós também.”

– Fiodór Dostoievski, 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 GUV (<http://www.guv.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/curso/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-16 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 e 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.

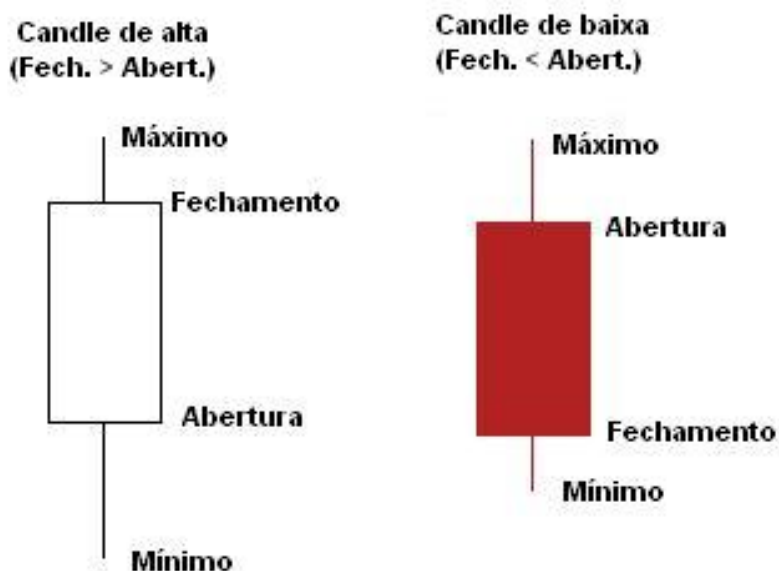
Apesar da triste história, foi em Dojima, no jardim do próprio Yodoya Keian, que nasceu a bolsa de arroz do Japão. Lá eram negociados, precificados e categorizados vários tipos de arroz. Para anotar os preços do arroz, desenhava-se figuras no papel. Essas figuras parecem muito com velas -- daí a analogia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

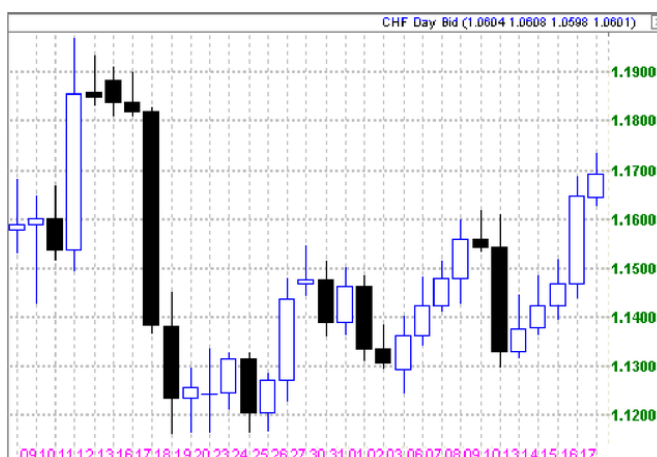
Um candlestick indica 4 valores: o maior preço do dia, o menor preço do dia (as pontas), o primeiro preço do dia e o último preço do dia (conhecidos como abertura e fechamento, respectivamente).

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 os **negócios** (*trades*) de um dia. Um **negócio** 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.

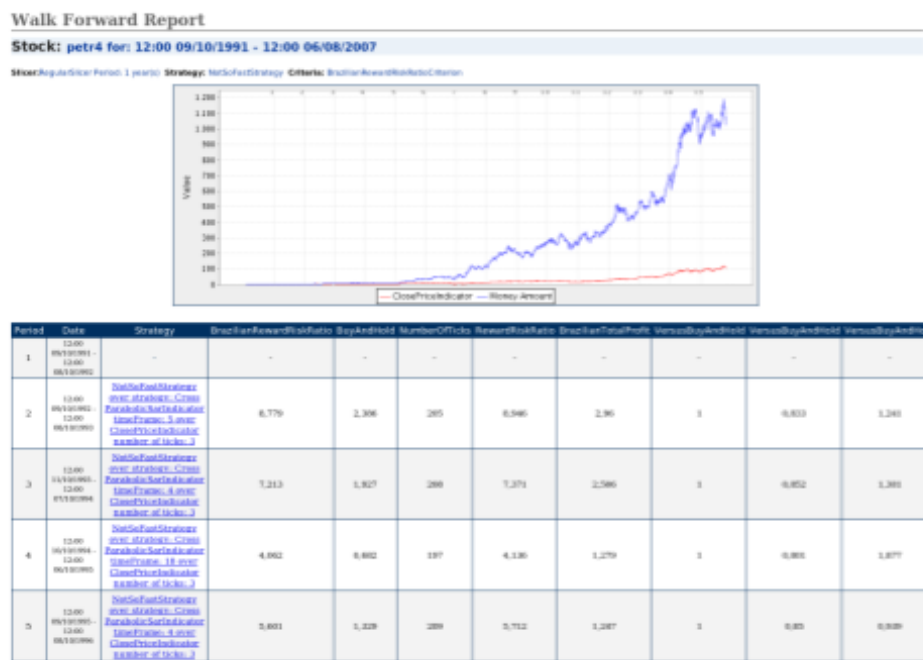
2.3 O PROJETO TAIL

A idéia do projeto **Tail** (Technical Analysis Indicator Library) 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 idéia de juntar ao grupo alguns alunos do mestrado através de um sistema de co-orientação, 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 co-orientação, 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-16.

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

- **Negocio** - 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 Negocios em Candlesticks.** Nossa base serão os Negócios. Precisamos converter uma lista de negócios 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 `Negocio`, não queremos que seu valor seja modificado:

```
class Negocio {

    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 do `Negocio` direto na declaração do atributo (o que não faz muito sentido nesse caso), ou então populamos pelo construtor:

```
class Negocio {  
  
    private final double valor;  
  
    public Negocio(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 `Negocio`, 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 modifiquem 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

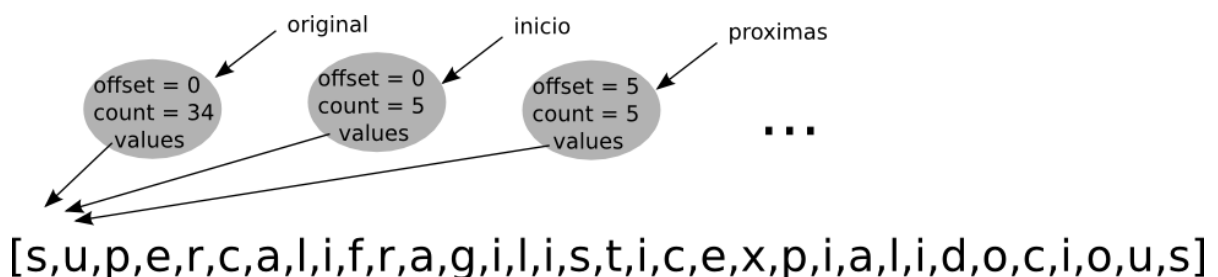
Como o objeto é imutável, a composição interna de cada um pode ser compartilhada entre eles, já que não há chances de algum deles mudar tais atributos. Esse compartilhamento educado possibilita fazer *cache* de suas partes internas, além de facilitar a manipulação desses objetos.

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 `Negocio` 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 do negócio 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 `Calendars`.

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`.

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).

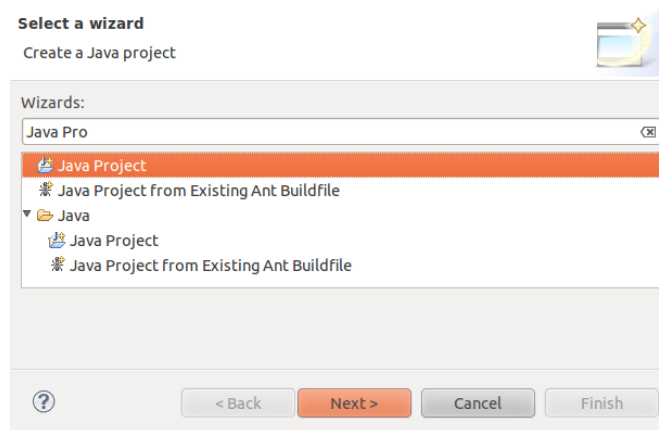
Antes do Java 5, para falar que algo era deprecated, usava-se um comentário especial no Javadoc. A partir do Java 5, a anotação `@Deprecated` foi adicionada à plataforma e garante verificações do próprio compilador (que gera um warning). Olhe o Javadoc da classe `Date` para ver tudo que foi deprecated.

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**.

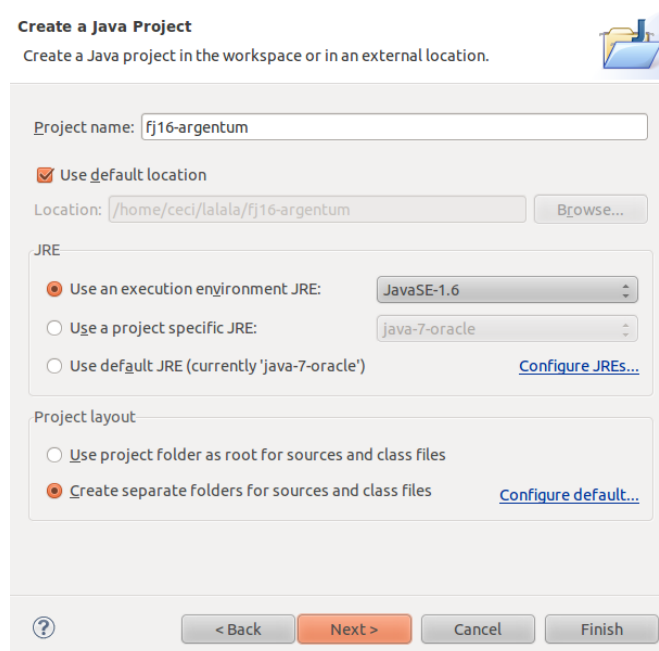
Para mais informações: <http://blog.caelum.com.br/2007/03/15/jsr-310-date-and-time-api/> <http://jcp.org/en/jsr/detail?id=310>

2.9 EXERCÍCIOS: O MODELO DO ARGENTUM

- 1) Vamos criar o projeto `fj-16-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*:



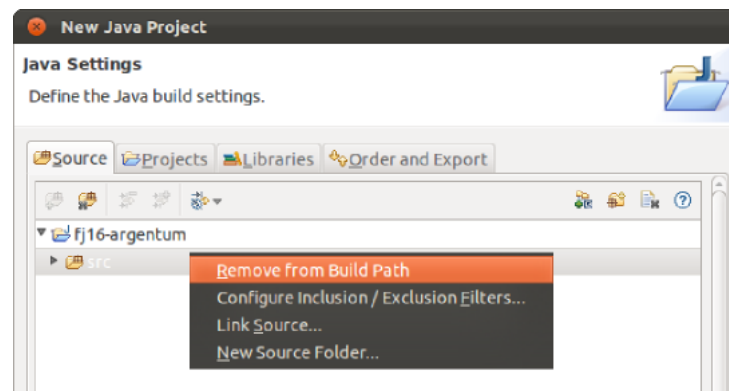
- 2) Na janela que abrirá em sequência, preencha o nome do projeto como **fj16-argentum** e clique em **Next**:



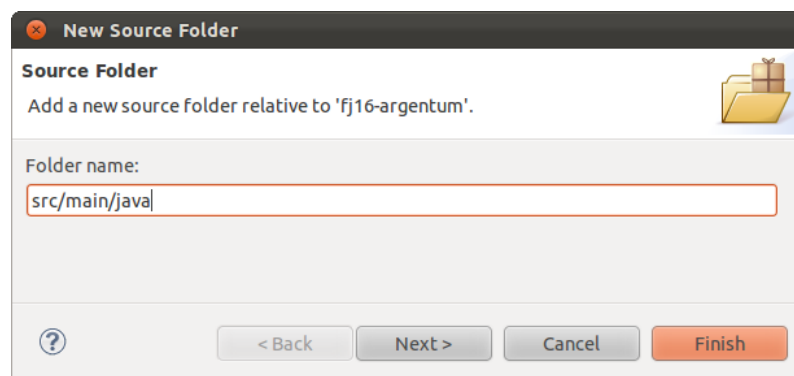
- 3) Na próxima tela, podemos definir uma série de configurações do projeto (que também podem ser feitas depois, através do menu *Build path -> Configure build path*, clicando com o botão da direita no projeto).

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

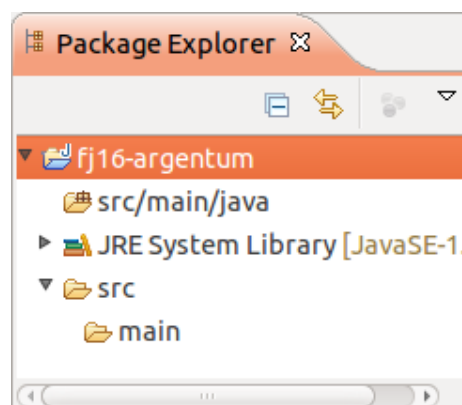
Nessa tela, **remova** o diretório `src` da lista de diretórios fonte:



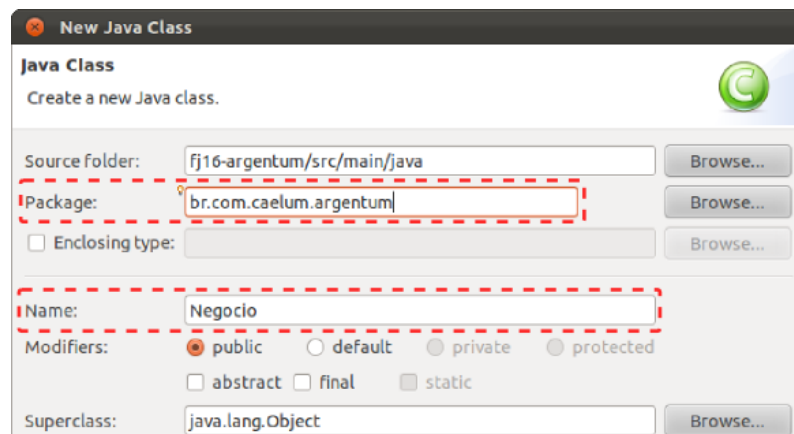
Agora, na mesma tela, adicione um novo diretório fonte, chamado **src/main/java**. Para isso, clique em **Create new source folder** e preencha com **src/main/java**:



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 **Negocio** e dentro do pacote **br.com.caelum.argentum**:



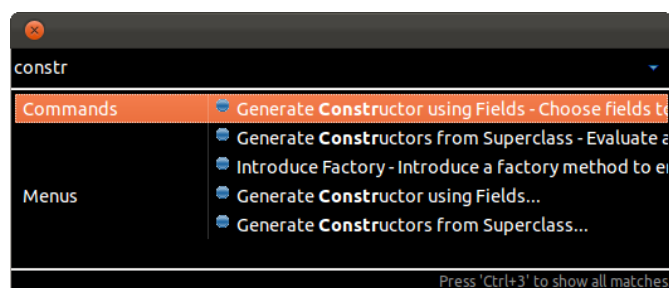
- 6) Transforme a classe em **final** e já declare os três atributos que fazem parte de uma negociação da bolsa de valores (também como final):

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

Não esqueça de importar o Calendar!

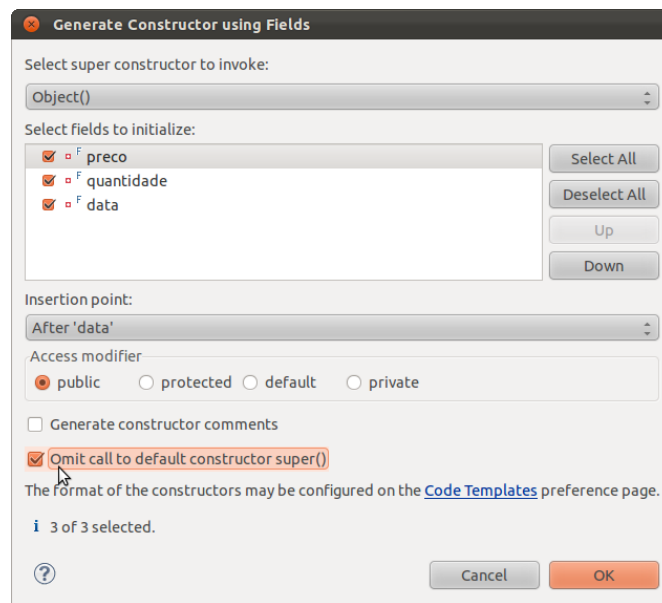
- 7) Vamos criar um construtor que recebe esses dados, já que são obrigatórios para nosso domínio. Em vez de fazer isso na mão, na edição da classe, use o atalho **ctrl + 3** e comece a digitar *constructor*. Ele vai mostrar uma lista das opções que contêm essa palavra: escolha a *Generate constructor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

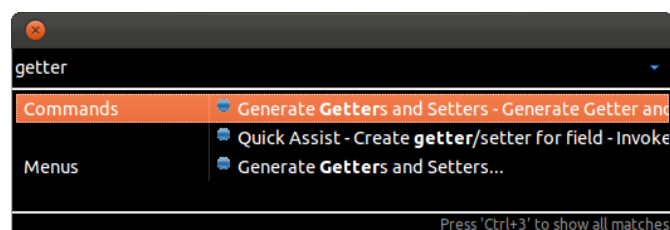
Atenção para deixar os campos na ordem 'preco, quantidade, data'. Você pode usar os botões *Up* e *Down* para mudar a ordem.



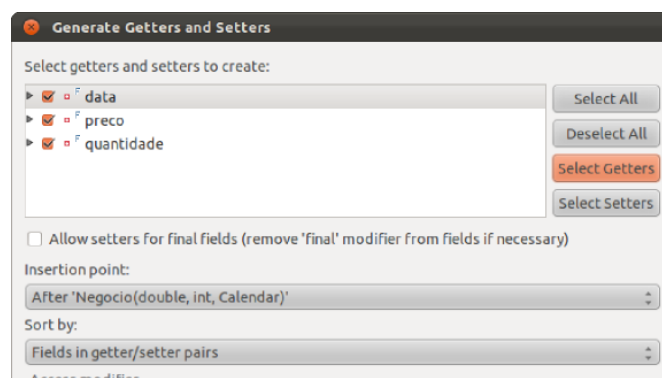
Pronto! Mande gerar. O seguinte código que será gerado:

```
public Negocio(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 Negocio {  
  
    private final double preco;  
    private final int quantidade;  
    private final Calendar data;  
  
    public Negocio(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 `Negocio` devolver o volume de dinheiro transferido naquela negociação. Na prática, é só multiplicar o preço pago pela quantidade de ações negociadas, resultando no total de dinheiro que aquela negociação realizou.

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

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

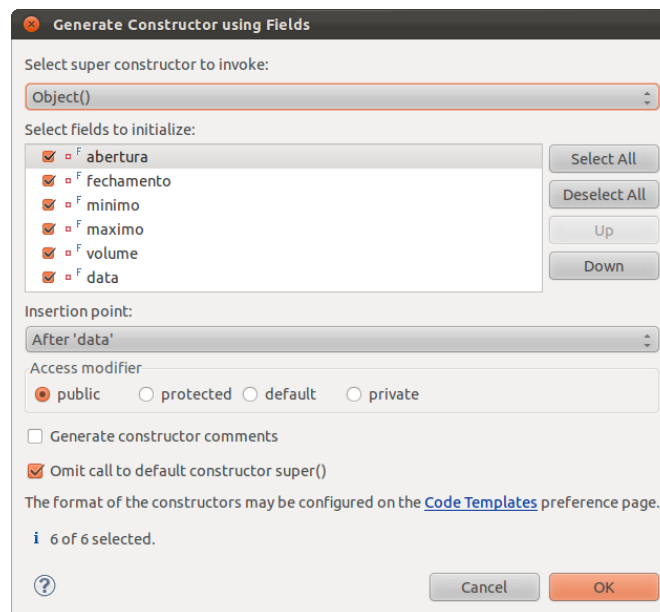
Repare que um método que parece ser um simples *getter* pode (e deve muitas vezes) encapsular regras de negócio e não necessariamente refletem um atributo da classe.

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:

```
1 public class Candlestick {  
2     private final double abertura;  
3     private final double fechamento;  
4     private final double minimo;  
5     private final double maximo;  
6     private final double volume;  
7     private final Calendar data;  
8  
9     public Candlestick(double abertura, double fechamento, double minimo,  
10         double maximo, double volume, Calendar data) {  
11         this.abertura = abertura;  
12         this.fechamento = fechamento;  
13         this.minimo = minimo;  
14         this.maximo = maximo;  
15         this.volume = volume;
```

```
16         this.data = data;
17     }
18
19     public double getAbertura() {
20         return abertura;
21     }
22     public double getFechamento() {
23         return fechamento;
24     }
25     public double getMinimo() {
26         return minimo;
27     }
28     public double getMaximo() {
29         return maximo;
30     }
31     public double getVolume() {
32         return volume;
33     }
34     public Calendar getData() {
35         return data;
36     }
37 }
```

- 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 DOS NEGÓCIOS

Agora que temos as classes que representam negociações na bolsa de valores (Negocio) e resumos diários dessas negociações (Candlestick), falta apenas fazer a ação de resumir os negócios 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 negócios 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 negócio 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 negócio, já vamos somando o volume de cada negociação.

O algoritmo, agora, está completamente especificado! Basta passarmos essas idéias 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 um negócio 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 negócios.

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 Negócios 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.reader`:



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<Negocio> negocios) {  
  
    }  
}
```

- 2) Calcularemos o preço máximo e mínimo percorrendo todos os negócios e achando os valores corretos. Para isso, usaremos variáveis temporárias e, dentro do for, verificaremos se o preço do negócio atual bate ou máximo. Se não bater, veremos se ele é menor que o mínimo. Calculamos o volume somando o volume de cada negócio em uma variável:

```
volume += negocio.getVolume();  
  
if (negocio.getPreco() > maximo) {  
    maximo = negocio.getPreco();  
} else if (negocio.getPreco() < minimo) {  
    minimo = negocio.getPreco();  
}
```

Podemos pegar o preço de abertura através de `negocios.get(0)` e o de fechamento por `negocios.get(negocios.size() - 1)`.

```
1 public class CandlestickFactory {  
2     public Candlestick constroiCandleParaData(Calendar data,  
3                                                List<Negocio> negocios) {
```



```
4     double maximo = negocios.get(0).getPreco();
5     double minimo = negocios.get(0).getPreco();
6     double volume = 0;
7
8     // digite foreach e dê um ctrl + espaço para ajudar a
9     // criar o bloco abaixo!
10    for (Negocio negocio : negocios) {
11        volume += negocio.getVolume();
12
13        if (negocio.getPreco() > maximo) {
14            maximo = negocio.getPreco();
15        } else if (negocio.getPreco() < minimo) {
16            minimo = negocio.getPreco();
17        }
18    }
19
20    double abertura = negocios.get(0).getPreco();
21    double fechamento = negocios.get(negocios.size()-1).getPreco();
22
23    return new Candlestick(abertura, fechamento, minimo, maximo,
24                           volume, data);
25 }
26 }
```

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

```
1 public class TestaCandlestickFactory {
2
3     public static void main(String[] args) {
4         Calendar hoje = Calendar.getInstance();
5
6         Negocio negocio1 = new Negocio(40.5, 100, hoje);
7         Negocio negocio2 = new Negocio(45.0, 100, hoje);
8         Negocio negocio3 = new Negocio(39.8, 100, hoje);
9         Negocio negocio4 = new Negocio(42.3, 100, hoje);
10
11        List<Negocio> negocios = Arrays.asList(negocio1, negocio2,
12                                              negocio3, negocio4);
13
14        CandlestickFactory fabrica = new CandlestickFactory();
15        Candlestick candle = fabrica.constroiCandleParaData(
16                                hoje, negocios);
17
18        System.out.println(candle.getAbertura());
19        System.out.println(candle.getFechamento());
20    }
21 }
```

```
20      System.out.println(candle.getMinimo());  
21      System.out.println(candle.getMaximo());  
22      System.out.println(candle.getVolume());  
23  }  
24 }
```

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

EFFECTIVE JAVA

1)

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 ajudar você a construir um determinado objeto em uma série de passos, independente de 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 um único negócio? Vamos escrever o teste, e ver o que acontece:

```
1 public class TestaCandlestickFactoryComUmNegocioApenas {  
2  
3     public static void main(String[] args) {  
4         Calendar hoje = Calendar.getInstance();
```

```
5
6     Negocio negocio1 = new Negocio(40.5, 100, hoje);
7
8     List<Negocio> negocios = Arrays.asList(negocio1);
9
10    CandlestickFactory fabrica = new CandlestickFactory();
11    Candlestick candle = fabrica.constroiCandleParaData(hoje, negocios);
12
13    System.out.println(candle.getAbertura());
14    System.out.println(candle.getFechamento());
15    System.out.println(candle.getMinimo());
16    System.out.println(candle.getMaximo());
17    System.out.println(candle.getVolume());
18 }
19 }
```

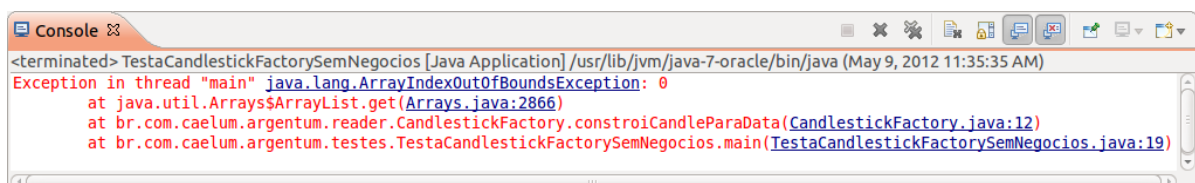
A saída deve indicar 40.5 como todos os valores, e 4050.0 como volume. Tudo parece bem?

- 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:

```
1 public class TestaCandlestickFactorySemNegocios {
2
3     public static void main(String[] args) {
4         Calendar hoje = Calendar.getInstance();
5
6         List<Negocio> negocios = Arrays.asList();
7
8         CandlestickFactory fabrica = new CandlestickFactory();
9         Candlestick candle = fabrica.constroiCandleParaData(hoje, negocios);
10
11        System.out.println(candle.getAbertura());
12        System.out.println(candle.getFechamento());
13        System.out.println(candle.getMinimo());
14        System.out.println(candle.getMaximo());
15        System.out.println(candle.getVolume());
16    }
17 }
```

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 `negocios.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 `minimo` com um valor **bem grande**, assim quando percorrermos o `for` qualquer valor já vai ser logo `minimo`. Mesmo com `maximo`, que deve ter um valor **bem pequeno**.

Mas quais valores colocar? Quanto é um número pequeno o suficiente? Ou um número grande o suficiente? Na classe `Double`, encontramos constantes que representam os maiores e menores números existentes.

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

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

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

```
double abertura = negocios.isEmpty() ? 0 : negocios.get(0).getPreco();  
double fechamento = negocios.isEmpty() ? 0 :  
    negocios.get(negocios.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 um negócio 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 um negócio? Vamos tentar com mais negócios? Crie e rode um teste com os seguintes negócios:

```
Negocio negocio1 = new Negocio(40.5, 100, hoje);  
Negocio negocio2 = new Negocio(45.0, 100, hoje);  
Negocio negocio3 = new Negocio(49.8, 100, hoje);  
Negocio negocio4 = new Negocio(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 Petrobrás a 30 reais, com uma quantidade negativa de negócios é válida? E com número zero de negócios?

Em outras palavras, posso dar `new` em um `Negocio` com esses dados?

- 2) Uma negociação com data nula é válida? Posso dar `new Negocio(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 negócios? O que acontece?
- 6) E se a ordem dos negócios passadas ao `CandlestickFactory` não estiver na ordem crescente das datas? Devemos aceitar? Não devemos?
- 7) E se esses `Negocios` 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.

@SuppressWarnings 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 @SuppressWarnings.

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.reader

public class CandlestickFactoryTest {

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

        Negocio negocio1 = new Negocio(40.5, 100, hoje);
        Negocio negocio2 = new Negocio(45.0, 100, hoje);
        Negocio negocio3 = new Negocio(39.8, 100, hoje);
        Negocio negocio4 = new Negocio(42.3, 100, hoje);

        List<Negocio> negocios = Arrays.asList(negocio1, negocio2, negocio3,
```

```
negocio4);  
  
CandlestickFactory fabrica = new CandlestickFactory();  
Candlestick candle = fabrica.constroiCandleParaData(hoje, negocios);  
}  
}
```

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 sequenciaSimplesDeNegocios() {  
        // ...  
    }  
}
```

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.

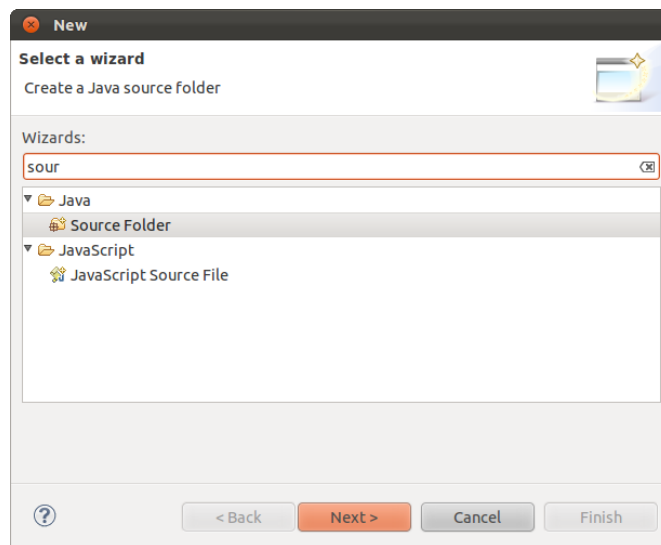
Nosso código final do teste, agora com as devidas asserções, ficará assim:

```
1 public class CandlestickFactoryTest {
2
3     @Test
4     public void sequenciaSimplesDeNegocios() {
5         Calendar hoje = Calendar.getInstance();
6
7         Negocio negocio1 = new Negocio(40.5, 100, hoje);
8         Negocio negocio2 = new Negocio(45.0, 100, hoje);
9         Negocio negocio3 = new Negocio(39.8, 100, hoje);
10        Negocio negocio4 = new Negocio(42.3, 100, hoje);
11
12        List<Negocio> negocios = Arrays.asList(negocio1, negocio2, negocio3,
13                                              negocio4);
14
15        CandlestickFactory fabrica = new CandlestickFactory();
16        Candlestick candle = fabrica.constroiCandleParaData(hoje, negocios);
17
18        Assert.assertEquals(40.5, candle.getAbertura(), 0.00001);
19        Assert.assertEquals(42.3, candle.getFechamento(), 0.00001);
20        Assert.assertEquals(39.8, candle.getMinimo(), 0.00001);
21        Assert.assertEquals(45.0, candle.getMaximo(), 0.00001);
22        Assert.assertEquals(1676.0, candle.getVolume(), 0.00001);
23    }
24 }
```

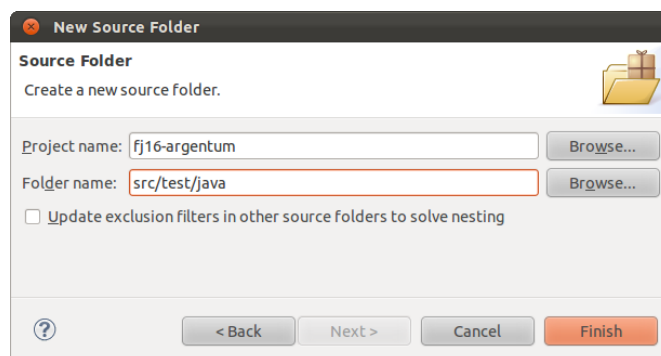
Existem ainda outras anotações principais e métodos importantes da classe `Assert`, que conheceremos no decorrer da construção do projeto.

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:



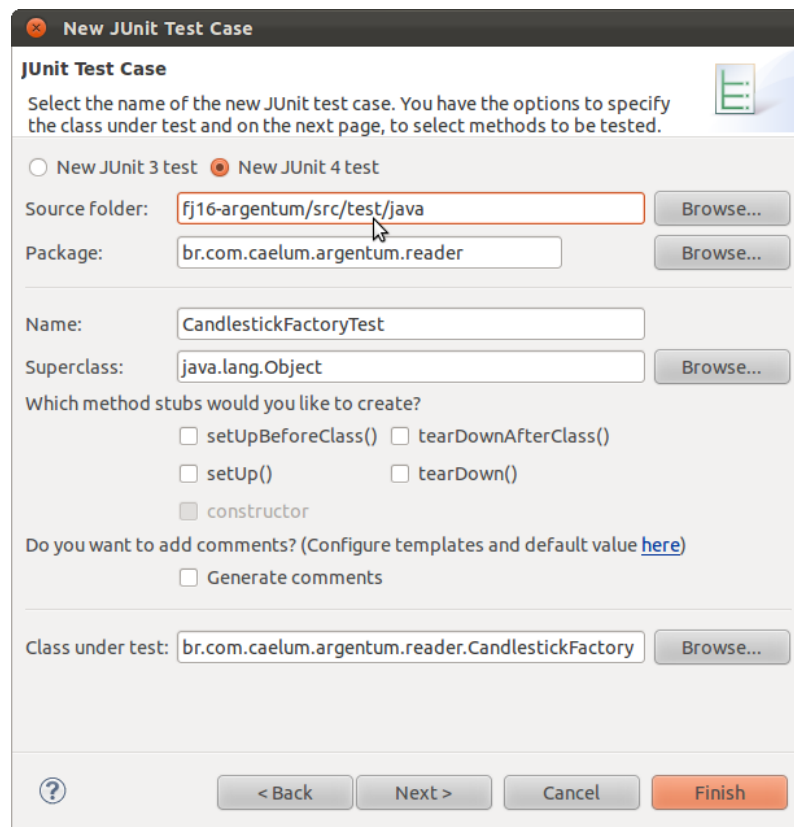
b) Preencha com **src/test/java** e clique **Finish**:



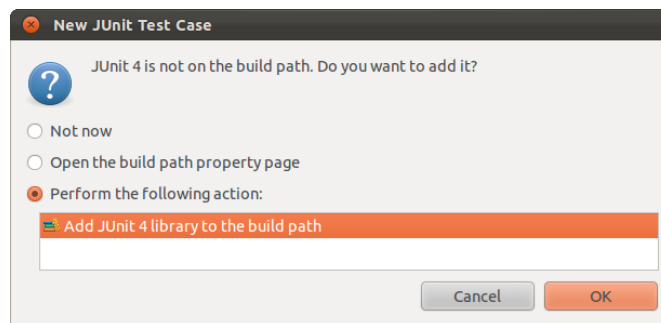
É nesse novo diretório em que você colocará todos seus testes de unidade.

- 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 uma exceção é lançada.

Vamos colocar primeiro o teste inicial:

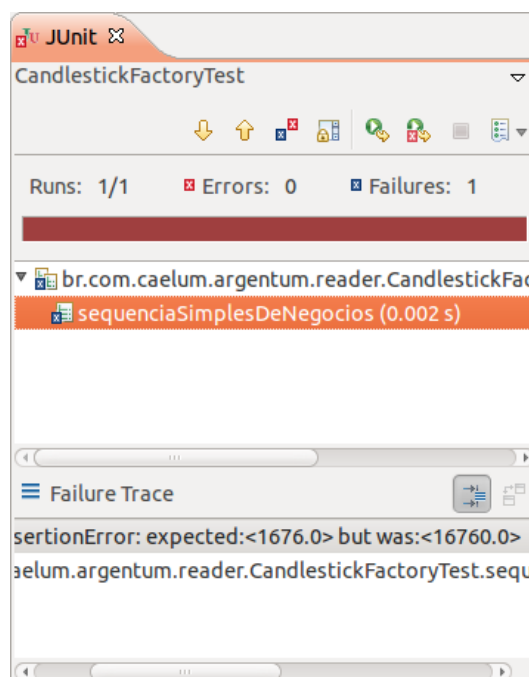
```
1 public class CandlestickFactoryTest {  
2  
3     @Test  
4     public void sequenciaSimplesDeNegocios() {  
5         Calendar hoje = Calendar.getInstance();  
6     }
```

```
7      Negocio negocio1 = new Negocio(40.5, 100, hoje);
8      Negocio negocio2 = new Negocio(45.0, 100, hoje);
9      Negocio negocio3 = new Negocio(39.8, 100, hoje);
10     Negocio negocio4 = new Negocio(42.3, 100, hoje);
11
12     List<Negocio> negocios = Arrays.asList(negocio1, negocio2, negocio3,
13                                           negocio4);
14
15     CandlestickFactory fabrica = new CandlestickFactory();
16     Candlestick candle = fabrica.constroiCandleParaData(hoje, negocios);
17
18     Assert.assertEquals(40.5, candle.getAbertura(), 0.00001);
19     Assert.assertEquals(42.3, candle.getFechamento(), 0.00001);
20     Assert.assertEquals(39.8, candle.getMinimo(), 0.00001);
21     Assert.assertEquals(45.0, candle.getMaximo(), 0.00001);
22     Assert.assertEquals(1676.0, candle.getVolume(), 0.00001);
23 }
24 }
```

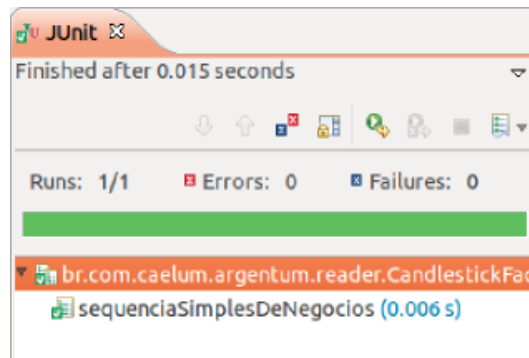
Para rodar, use qualquer um dos seguintes atalhos:

- **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 nenhum negócio:

```
1 @Test
2 public void semNegociosGeraCandleComZeros() {
3     Calendar hoje = Calendar.getInstance();
4
5     List<Negocio> negocios = Arrays.asList();
6
7     CandlestickFactory fabrica = new CandlestickFactory();
8     Candlestick candle = fabrica.constroiCandleParaData(hoje, negocios);
9
10    Assert.assertEquals(0.0, candle.getVolume(), 0.00001);
11 }
```

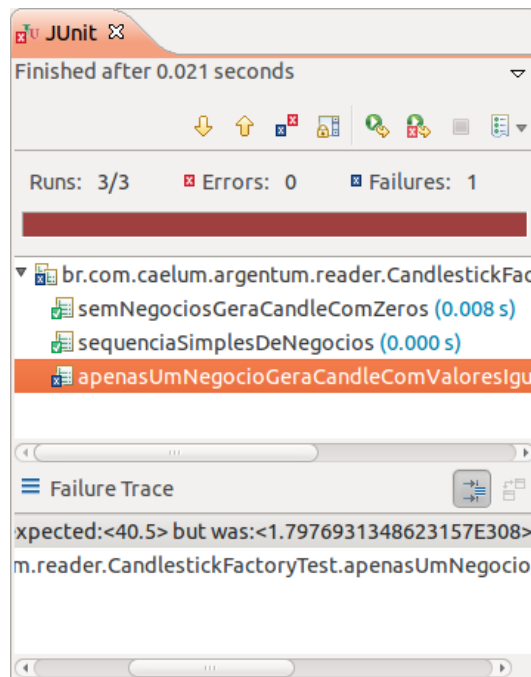
Rode o teste com o mesmo atalho.

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

```
1 @Test
2 public void apenasUmNegocioGeraCandleComValoresIguais() {
3     Calendar hoje = Calendar.getInstance();
4
5     Negocio negocio1 = new Negocio(40.5, 100, hoje);
6
7     List<Negocio> negocios = Arrays.asList(negocio1);
8
9     CandlestickFactory fabrica = new CandlestickFactory();
10    Candlestick candle = fabrica.constroiCandleParaData(hoje, negocios);
11 }
```

```
12    Assert.assertEquals(40.5, candle.getAbertura(), 0.00001);
13    Assert.assertEquals(40.5, candle.getFechamento(), 0.00001);
14    Assert.assertEquals(40.5, candle.getMinimo(), 0.00001);
15    Assert.assertEquals(40.5, candle.getMaximo(), 0.00001);
16    Assert.assertEquals(4050.0, candle.getVolume(), 0.00001);
17 }
```

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, `Negocio` 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 `Negocio` 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 nos negócios 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 `Negocio` é realmente imutável?

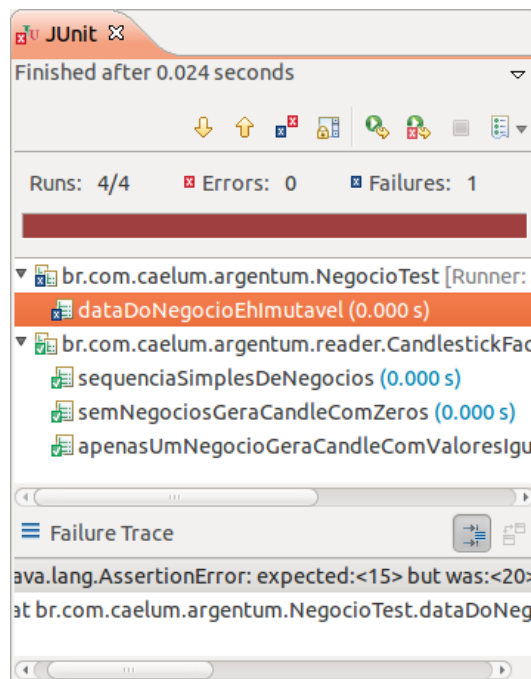
Vamos criar um novo *Unit Test* para a classe `Negocio`. O processo é o mesmo que fizemos para o teste da `CandlestickFactory`: abra a classe `Negocio` no editor e faça **Ctrl + N** JUnit Test Case.

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

```
1 public class NegocioTest {
2
3     @Test
4     public void dataDoNegocioEhImutavel() {
5         // se criar um negocio no dia 15...
6         Calendar c = Calendar.getInstance();
7         c.set(Calendar.DAY_OF_MONTH, 15);
8         Negocio n = new Negocio(10, 5, c);
9
10
11         // ainda que eu tente mudar a data para 20...
12         n.getData().set(Calendar.DAY_OF_MONTH, 20);
13
14         // ele continua no dia 15.
15         Assert.assertEquals(15, n.getData().get(Calendar.DAY_OF_MONTH));
16     }
17 }
```

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 `Negocio` 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 um `Negocio` com data nula? Por enquanto, podemos, mas não deveríamos. Para que outras partes do meu sistema não se surpreendam mais tarde, vamos impedir que o `Negocio` seja criado 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!

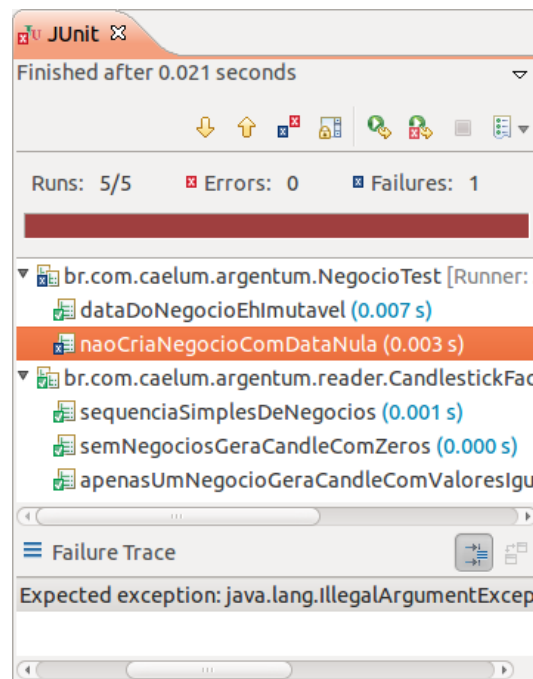
Antes de fazer a modificação na classe, vamos preparar o teste. Mas o que queremos testar? Queremos saber se nossa classe `Negocio` não permite a criação do objeto e lança uma `IllegalArgumentException` quando passamos `null` no construtor.

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 `naoCriaNegocioComDataNula` na classe `NegocioTest`. 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 naoCriaNegocioComDataNula() {
    new Negocio(10, 5, null);
}
```

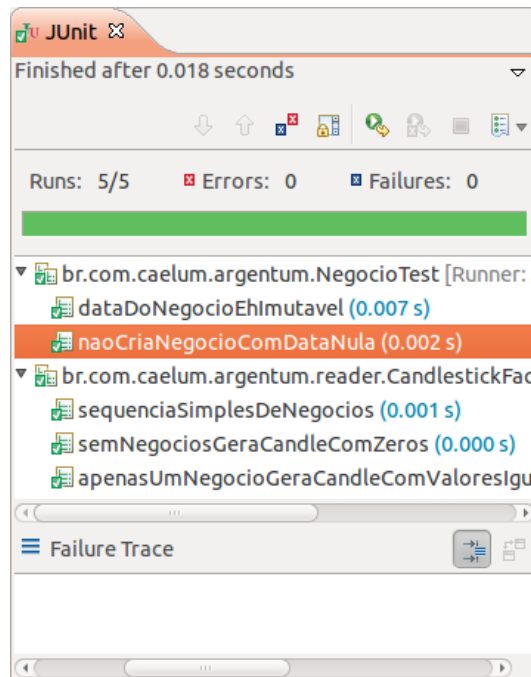
Rode os testes. Barrinha vermelha! Já que ainda não verificamos o argumento na classe `Negocio` e ainda não lançamos a exceção:



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

```
public Negocio(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 negócios 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 `semNegociosGeraCandleComZeros` 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 `Negocio`. 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 `negociosEmOrdemCrescenteDeValor` e `negociosEmOrdemDecrescenteDeValor`, que devem fazer o que o próprio nome diz.

Agora eles funcionam?

3.11 PARA SABER MAIS: IMPORT ESTÁTICO

Algumas vezes, escrevemos classes que contêm 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>
  <negocio>
    <preco>43.5</preco>
    <quantidade>1000</quantidade>
    <data>
      <time>1222333777999</time>
    </data>
  </negocio>
</negocio>
```

```
<preco>44.1</preco>
<quantidade>700</quantidade>
<data>
  <time>1222444777999</time>
</data>
</negocio>
<negocio>
  <preco>42.3</preco>
  <quantidade>1200</quantidade>
  <data>
    <time>1222333999777</time>
  </data>
</negocio>
</list>
```

Uma lista de negócios! Cada negócio 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 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:

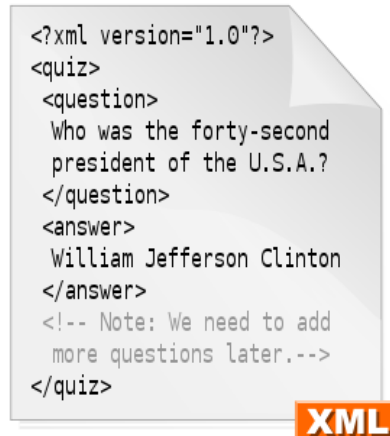
<http://www.w3.org/XML/>

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="ISO-8859-1"?>
```

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 voce é avisado, por exemplo
        // do inicio da tag "<preco>"
    }

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

        // para saber o que fazer com esses dados, voce 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 ThoughtWorks, 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 `negocio` em mãos, basta agora pedirmos ao `XStream` que gera o XML correspondente:

```
Negocio negocio = new Negocio(42.3, 100, Calendar.getInstance());
```

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

E o resultado é:

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

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.Negocio`. 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("negocio", Negocio.class);
```

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

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

```
XStream stream = new XStream(new DomDriver());
stream.alias("negocio", Negocio.class);
Negocio n = (Negocio) stream.fromXML("<negocio>" +
                                     "<preco>42.3</preco>" +
                                     "<quantidade>100</quantidade>" +
                                     "</negocio>");
System.out.println(negocio.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("negocio", Negocio.class);`, podemos fazer essa configuração direto na classe `Negocio` com uma anotação:

```
@XStreamAlias("negocio")
public class Negocio {
}
```

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:

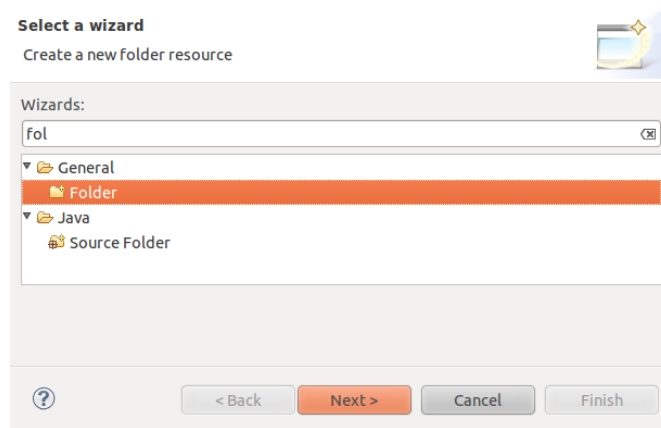
```
stream.processAnnotations(Negocio.class);
```

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.

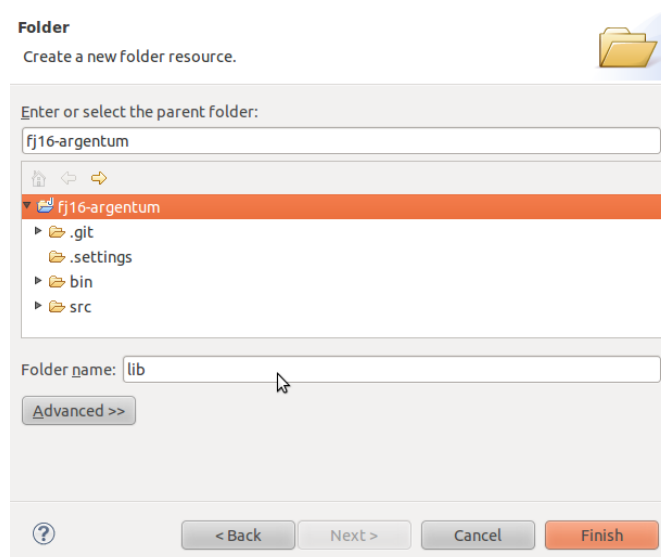
4.5 EXERCÍCIOS: LENDO O XML

- 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.

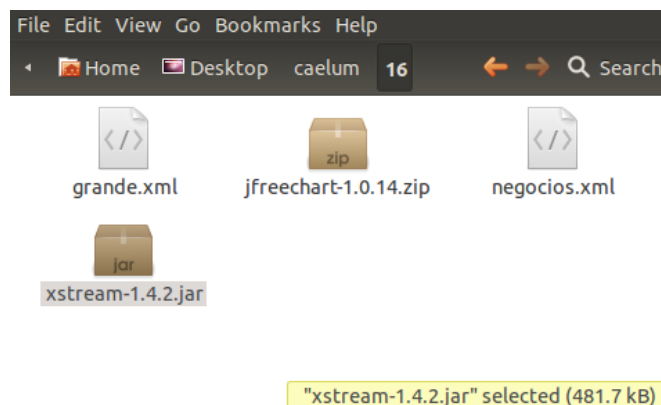
Crie uma nova pasta usando **ctrl + N** e começando a digitar *Folder*:



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



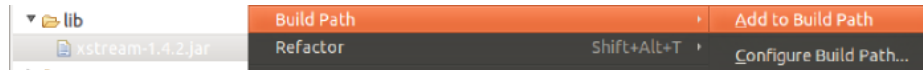
- 2) Vamos pôr o XStream no nosso projeto. Vá na pasta **Caelum** no seu Desktop e entre em **16**. Localize o arquivo do XStream:



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`:

```
1 package br.com.caelum.argentum.reader;
2
3 // imports...
4
5 public class LeitorXML {
6
7     public List<Negocio> carrega(Reader fonte) {
8         XStream stream = new XStream(new DomDriver());
9         stream.alias("negocio", Negocio.class);
10        return (List<Negocio>) stream.fromXML(fonte);
11    }
12 }
```

- 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 `StringReader`:

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

    LeitorXML leitor = new LeitorXML();
    List<Negocio> negocios = leitor.carrega(new StringReader(xmlDeTeste));
}
```

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

```
String xmlDeTeste = "<list>" +
    "    <negocio>" +
    "        <preco>43.5</preco>" +
    "        <quantidade>1000</quantidade>" +
    "        <data>" +
```

```
"          <time>1322233344455</time>" +  
"      </data>" +  
"  </negocio>" +  
"</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;
- o negócio deve ter preço 43.5;
- a quantidade deve ser 1000.

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

- Zero negócios;
- Preço ou quantidade faltando;
- Outras quantidades de negócios (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 `FileReader`, passa o XML para o `LeitorXML` e depois chama a `CandlestickFactory` para quebrar os negócios 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 SEPARANDO AS CANDLES

Agora que temos nosso leitor de XML que cria uma lista com os negócios representados 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 negócios 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 negócio 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 negócios 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 um negócio 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).

4.7 TEST DRIVEN DESIGN - 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 benefícios 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.

4.8 EXERCÍCIOS: SEPARANDO OS CANDLES

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 os negócios em vários candles por datas em outro lugar.

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

Para saber, conforme percorremos todos os negócios, se a negociação atual ainda aconteceu na mesma data que estamos procurando, vamos usar um método na classe `Negocio` que faz tal verificação.

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

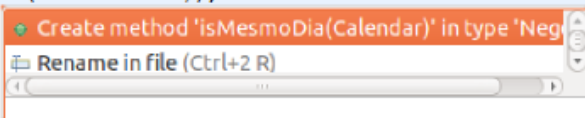
```
@Test
public void mesmoMilissegundoEhDoMesmoDia() {
    Calendar agora = Calendar.getInstance();
    Calendar mesmoMomento = (Calendar) agora.clone();

    Negocio negocio = new Negocio(40.0, 100, agora);
    Assert.assertTrue(negocio.isMesmoDia(memoMomento));
}
```

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**.

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

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



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 `NegocioTest`:

```
@Test
public void mesmoDiaHorasDiferentesEhDoMesmoDia() {
    // 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);

    Negocio negocio = new Negocio(40.0, 100, manha);
    Assert.assertTrue(negocio.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.DATE) == outraData.get(Calendar.DATE);
}
```

Altere o método `isMesmoDia` na classe `Negocio` 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 `Negocio`, 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 `Negocio` deve ter ficado bem parecido com isso:

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

- 5) Próximo passo: dada uma lista de negócios 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 `paraNegociosDeTresDiasDistintosGeraTresCandles` **na classe** `CandlestickFactoryTest`:

```
1 @Test
2 public void paraNegociosDeTresDiasDistintosGeraTresCandles() {
3     Calendar hoje = Calendar.getInstance();
4
5     Negocio negocio1 = new Negocio(40.5, 100, hoje);
6     Negocio negocio2 = new Negocio(45.0, 100, hoje);
7     Negocio negocio3 = new Negocio(39.8, 100, hoje);
8     Negocio negocio4 = new Negocio(42.3, 100, hoje);
9
10    Calendar amanha = (Calendar) hoje.clone();
11    amanha.add(Calendar.DAY_OF_MONTH, 1);
12
13    Negocio negocio5 = new Negocio(48.8, 100, amanha);
14    Negocio negocio6 = new Negocio(49.3, 100, amanha);
15
16    Calendar depois = (Calendar) amanha.clone();
```

```
17    depois.add(Calendar.DAY_OF_MONTH, 1);
18
19    Negocio negocio7 = new Negocio(51.8, 100, depois);
20    Negocio negocio8 = new Negocio(52.3, 100, depois);
21
22    List<Negocio> negocios = Arrays.asList(negocio1, negocio2, negocio3,
23        negocio4, negocio5, negocio6, negocio7, negocio8);
24
25    CandlestickFactory fabrica = new CandlestickFactory();
26
27    List<Candlestick> candles = fabrica.constroiCandles(negocios);
28
29    Assert.assertEquals(3, candles.size());
30    Assert.assertEquals(40.5, candles.get(0).getAbertura(), 0.00001);
31    Assert.assertEquals(42.3, candles.get(0).getFechamento(), 0.00001);
32    Assert.assertEquals(48.8, candles.get(1).getAbertura(), 0.00001);
33    Assert.assertEquals(49.3, candles.get(1).getFechamento(), 0.00001);
34    Assert.assertEquals(51.8, candles.get(2).getAbertura(), 0.00001);
35    Assert.assertEquals(52.3, candles.get(2).getFechamento(), 0.00001);
36 }
```

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<Negocio>` adicionando cada `negocio` no `Candlestick` atual;
- Quando achar um negócio 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!

```
1 public List<Candlestick> constroiCandles(List<Negocio> todosNegocios) {
2     List<Candlestick> candles = new ArrayList<Candlestick>();
3
4     List<Negocio> negociosDoDia = new ArrayList<Negocio>();
5     Calendar dataAtual = todosNegocios.get(0).getData();
6
7     for (Negocio negocio : todosNegocios) {
8         // se não for mesmo dia, fecha candle e reinicia variáveis
9         if (!negocio.isMesmoDia(dataAtual)) {
10             Candlestick candleDoDia = constroiCandleParaData(dataAtual, negociosDoDia);
11             candles.add(candleDoDia);
12             negociosDoDia = new ArrayList<Negocio>();
13         }
14     }
15     return candles;
16 }
```

```
13         dataAtual = negocio.getData();
14     }
15     negociosDoDia.add(negocio);
16 }
17 // adiciona último candle
18 Candlestick candleDoDia = constroiCandleParaData(dataAtual, negociosDoDia);
19 candles.add(candleDoDia);
20
21 return candles;
22 }
```

Rode o teste!

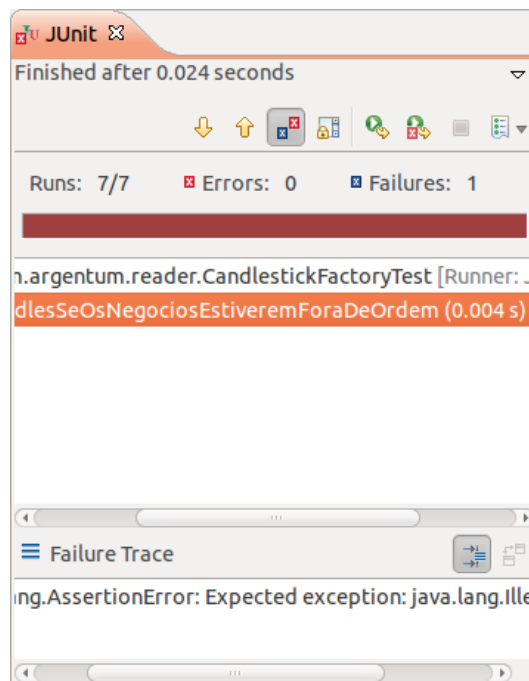
4.9 EXERCÍCIOS OPCIONAIS

- 1) E se passarmos para o método `constroiCandles` da fábrica uma lista de negócios 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 negócios vem ordenados pela data, é boa prática programar defensivamente em relação aos parâmetros recebidos.

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

Crie o `naoPermiteConstruirCandlesComNegociosForaDeOrdem` e configure o teste para verificar que uma `IllegalStateException` foi lançada. Basta usar como base o mesmo teste que tínhamos antes, mas adicionar os negócios 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 (Negocio negocio : todosNegocios) {
    if (negocio.getData().before(dataAtual)) {
        throw new IllegalStateException("negocios 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 negócios. Esses preços podem variar.

```
1 public class GeradorAleatorioDeXML {
2     public static void main(String[] args) throws IOException {
3         Calendar data = Calendar.getInstance();
4         Random random = new Random(123);
5         List<Negocio> negocios = new ArrayList<Negocio>();
6
7         double valor = 40;
8         int quantidade = 1000;
9
10        for (int dias = 0; dias < 30; dias++) {
11            int quantidadeNegociosDoDia = random.nextInt(20);
12
13            for (int negocio = 0; negocio < quantidadeNegociosDoDia; negocio++){
14                // no máximo sobe ou cai R$1,00 e não baixa além de R$5,00
15                valor += (random.nextInt(200) - 100) / 100.0;
16                if (valor < 5.0) {
```

```
17         valor = 5.0;
18     }
19
20     // quantidade: entre 500 e 1500
21     quantidade += 1000 - random.nextInt(500);
22
23     Negocio n = new Negocio(valor, quantidade, data);
24     negocios.add(n);
25 }
26 data = (Calendar) data.clone();
27 data.add(Calendar.DAY_OF_YEAR, 1);
28 }
29
30 XStream stream = new XStream(new DomDriver());
31 stream.alias("negocio", Negocio.class);
32 stream.setMode(XStream.NO_REFERENCES);
33
34 PrintStream out = new PrintStream(new File("negocios.xml"));
35 out.println(stream.toXML(negocios));
36 }
37 }
```

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 igualzinhas. 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 `Negocio` seja ordenável pela data dos negócios.

Então poderemos, logo no início do método, ordenar todos os negócios com `Collections.sort` e não precisamos mais verificar se os negócios 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.

4.10 DISCUSSÃO EM AULA: ONDE USAR XML E O ABUSO DO MESMO

Interfaces gráficas com Swing

“Não tenho medo dos computadores. Temo a falta deles.”

– Isaac Asimov

5.1 INTERFACES GRÁFICAS EM JAVA

Atualmente, o Java suporta, oficialmente, dois tipos de bibliotecas gráficas: **AWT** e **Swing**. A AWT foi a primeira API para interfaces gráficas a surgir no Java e foi, mais tarde, superada pelo Swing (a partir do Java 1.2), que possui diversos benefícios em relação a seu antecessor.

As bibliotecas gráficas são bastante simples no que diz respeito a conceitos necessários para usá-las. A complexidade no aprendizado de interfaces gráficas em Java reside no tamanho das bibliotecas e no enorme mundo de possibilidades; isso pode assustar, em um primeiro momento.

AWT e Swing são bibliotecas gráficas oficiais incluídas em qualquer JRE ou JDK. Além destas, existem algumas outras bibliotecas de terceiros, sendo a mais famosa, o SWT - desenvolvida pela IBM e utilizada no Eclipse e em vários outros produtos.

5.2 PORTABILIDADE

As APIs de interface gráfica do Java favorecem, ao máximo, o lema de portabilidade da plataforma Java. O **look-and-feel** do Swing é único em todas as plataformas onde roda, seja ela Windows, Linux, ou qualquer outra. Isso implica que a aplicação terá exatamente a mesma interface (cores, tamanhos etc) em qualquer sistema operacional.

Grande parte da complexidade das classes e métodos do Swing está no fato da API ter sido desenvolvida tendo em mente o máximo de portabilidade possível. Favorece-se, por exemplo, o posicionamento relativo de componentes, em detrimento do uso de posicionamento fixo, que poderia prejudicar usuários com resoluções de tela diferentes da prevista.

Com Swing, não importa qual sistema operacional, qual resolução de tela, ou qual profundidade de cores: sua aplicação se comportará da mesma forma em todos os ambientes.

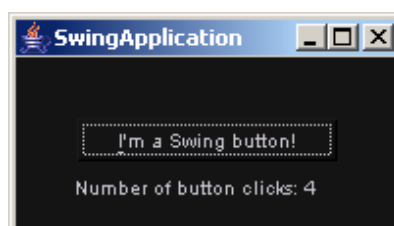
5.3 LOOK AND FEEL

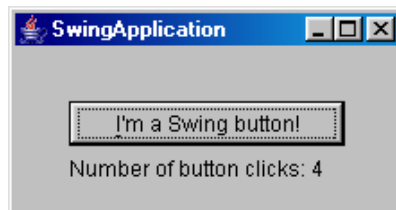
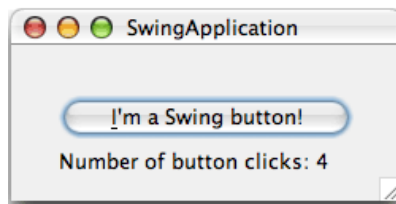
Look-and-Feel (ou LaF) é o nome que se dá à “cara” da aplicação (suas cores, formatos e etc). Por padrão, o Java vem com um look-and-feel próprio, que se comporta exatamente da mesma forma em todas as plataformas suportadas.

Mas, às vezes, esse não é o resultado desejado. Quando rodamos nossa aplicação no Windows, por exemplo, é bastante gritante a diferença em relação ao visual das aplicações nativas. Por isso é possível alterar qual o look-and-feel a ser usado em nossa aplicação.

Além do padrão do Java, o JRE 5 da Sun ainda traz LaF nativos para Windows e Mac OS, além do Motif e GTK. E, fora esses, você ainda pode baixar diversos LaF na Internet ou até desenvolver o seu próprio.

Veja esses screenshots da documentação do Swing mostrando a mesma aplicação rodando com 4 LaF diferentes:





5.4 COMPONENTES

O Swing traz muitos componentes para usarmos: botões, entradas de texto, tabelas, janelas, abas, scroll, árvores de arquivos e muitos outros. Durante o treinamento, veremos alguns componentes que nos ajudarão a fazer as telas do Argentum.

A biblioteca do Swing está no pacote `javax.swing` (inteira, exceto a parte de acessibilidade, que está em `javax.accessibility`).

5.5 COMEÇANDO COM SWING - MENSAGENS

A classe mais simples do Swing é a `JOptionPane` que mostra janelinhas de mensagens, confirmação e erros, entre outras.

Podemos mostrar uma mensagem para o usuário com a seguinte linha:

```
JOptionPane.showMessageDialog(null, "Minha mensagem!");
```

A classe `JFileChooser` é a responsável por mostrar uma janela de escolha de arquivos. É possível indicar o diretório inicial, os tipos de arquivos a serem mostrados, selecionar um ou vários e muitas outras opções.

Para mostrar a mensagem:

```
JFileChooser fileChooser = new JFileChooser();  
fileChooser.showOpenDialog(null);
```

O argumento do `showOpenDialog` indica qual o componente pai da janela de mensagem (pensando em algum frame aberto, por exemplo, que não é nosso caso). Esse método retorna um `int` indicando se o usuário escolheu um arquivo ou cancelou. Se ele tiver escolhido um, podemos obter o `File` com `getSelectedFile`:


```
JFileChooser fileChooser = new JFileChooser();
int retorno = fileChooser.showOpenDialog(null);

if (retorno == JFileChooser.APPROVE_OPTION) {
    File file = fileChooser.getSelectedFile();
    // faz alguma coisa com arquivo
} else {
    // dialogo cancelado
}
```

5.6 EXERCÍCIOS: ESCOLHENDO O XML COM JFileChooser

- 1) Vamos escrever um programa simples que permita ao usuário escolher qual XML ele quer abrir e exibe uma mensagem com o preço do primeiro negócio.

Usaremos um JFileChooser e um JOptionPane para mostrar o resultado.

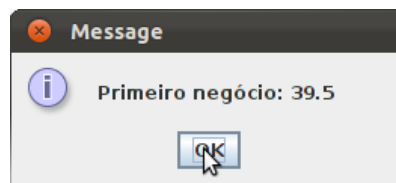
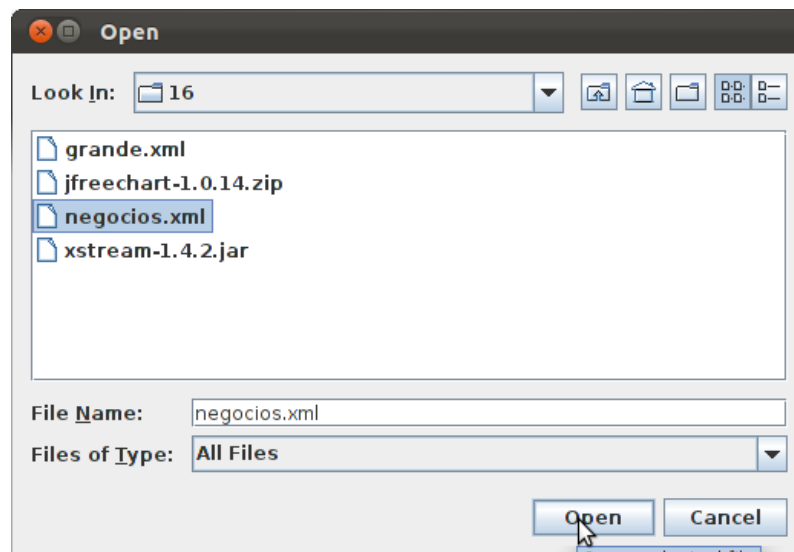
Crie a classe EscolhedorDeXML no pacote `br.com.caelum.argentum.ui`. Escreva o método `escolhe`:

```
1 public void escolhe() {
2     try {
3         JFileChooser chooser = new JFileChooser();
4         int retorno = chooser.showOpenDialog(null);
5
6         if (retorno == JFileChooser.APPROVE_OPTION) {
7             FileReader reader = new FileReader(chooser.getSelectedFile());
8             List<Negocio> negocios = new LeitorXML().carrega(reader);
9
10            Negocio primeiroNegocio = negocios.get(0);
11            String mensagem = "Primeiro negócio: " + primeiroNegocio.getPreco();
12            JOptionPane.showMessageDialog(null, mensagem);
13        }
14    } catch (FileNotFoundException e) {
15        e.printStackTrace();
16    }
17 }
```

Adicione o método `main` nessa mesma classe dando `new` e chamando o método que acabamos de criar:

```
public static void main(String[] args) {
    new EscolhedorDeXML().escolhe();
}
```

Rode a classe e escolha o arquivo **negocios.xml** (ele está na pasta `caelum/16` no seu Desktop).



- 2) Por padrão, o JFileChooser abre na pasta do usuário. Para abrir inicialmente em outra pasta, podemos usar o argumento no construtor. Para abrir na pasta do curso nas máquinas da Caelum, por exemplo, adicionaríamos o parâmetro:

```
public void escolhe() {  
    try {  
        JFileChooser chooser = new JFileChooser("/caelum/cursos/16");  
        int retorno = chooser.showOpenDialog(null);  
        //...
```

- 3) (opcional) Do jeito que fizemos, o usuário pode selecionar qualquer arquivo e não apenas os XMLs. Podemos filtrar por extensão de arquivo adicionando a seguinte linha **antes** da chamada ao showOpenDialog:

```
chooser.setFileFilter(new FileNameExtensionFilter("Apenas XML", "xml"));
```

5.7 COMPONENTES: JFRAME, JPANEL E JBUTTON

Os componentes mais comuns já estão frequentemente prontos e presentes na API do Swing. Contudo, para montar as telas que são específicas do seu projeto, será necessário compor alguns componentes mais básicos, como JFrames, JPanels, JButtons, etc.

A tela que vamos para o Argentum terá vários componentes. Para começarmos, faremos dois botões importantes: um dispara o carregamento do XML e outro permite ao usuário sair da aplicação.

Criar um componente do Swing é bastante simples! Por exemplo, para criar um botão:

```
JButton botao = new JButton("Carregar XML");
```

Contudo, esse botão apenas não faz nossa tela ainda. É preciso adicionar o botão para sair da aplicação. O problema é que, para exibirmos vários componentes organizadamente, é preciso usar um painel para agrupar esses componentes: o JPanel:

```
JButton botaoCarregar = new JButton("Carregar XML");  
JButton botaoSair = new JButton("Sair");  
  
JPanel painel = new JPanel();  
painel.add(botaoCarregar);  
painel.add(botaoSair);
```

Repare que a ordem em que os componentes são adicionados **importa**! Repare também que não definimos posicionamento algum para os nossos componentes: estamos usando o comportamento padrão dos painéis - falaremos mais sobre os layout managers logo mais.

Por fim, temos apenas objetos na memória. Ainda é preciso mostrar esses objetos na tela do usuário. O componente responsável por isso é o JFrame, a moldura da janela aberta no sistema operacional:

```
JFrame janela = new JFrame("Argentum");  
janela.add(painel);  
janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
janela.pack();  
janela.setVisible(true);
```

O método `pack()` de `JFrame`, chamado acima, serve para organizar os componentes do *frame* para que eles ocupem o menor espaço possível. E o `setVisible` recebe um boolean indicando se queremos que a janela esteja visível ou não.

Adicionamos também um comando que indica ao nosso *frame* que a aplicação deve ser terminada quando o usuário fechar a janela (caso contrário a aplicação continuaria rodando).

5.8 O DESIGN PATTERN COMPOSITE: COMPONENT E CONTAINER

Toda a API do Swing é feita usando os mais variados design patterns, procurando deixar sua arquitetura bastante flexível, extensível e modularizada.

Uma prática comum em tutoriais e explicações sobre Swing é estender os componentes para *simplificar* o trabalho com a API. O problema dessa abordagem é que as classes `Component` e `Container` têm um ciclo de vida complicado e é fácil cometer um engano ao reescrever seus métodos.

Além disso, com uma observação mais orientada a objetos do Swing fica claro que, assim como já vimos em cursos anteriores, a API foi construída para promover o uso de composição em vez de herança.

EFFECTIVE JAVA

Item 16: Favoreça composição à herança

Um dos patterns base usados pelo Swing é o **Composite** aplicado aos componentes e containers. A idéia é que todo componente é também um container de outros componentes. E, dessa forma, vamos *compondo* uma hierarquia de componentes uns dentro dos outros.

No nosso exemplo anterior, o JPanel é um componente adicionado ao JFrame. Mas o JPanel também é um container onde adicionamos outros componentes, os JButtons. E, poderíamos ir mais além, mostrando como JButtons podem ser containers de outros componentes como por exemplo algum ícone usado na exibição.

5.9 TRATANDO EVENTOS

Até agora os botões que fizemos não têm efeito algum, já que não estamos tratando os **eventos** que são disparados no componente. E quando falamos de botões, em geral, estamos interessados em saber quando ele foi *disparado* (clicado). No linguajar do Swing, queremos saber quando uma ação (action) aconteceu com o botão.

Mas como chamar um método quando o botão for clicado? Como saber esse momento?

O Swing, como a maioria dos toolkits gráficos, nos traz o conceito de **Listeners (ouvintes)**, que são interfaces que implementamos para sobrescrever métodos que serão disparados pelas ações sobre um botão, por exemplo. Os eventos do usuário são capturados pelo Swing que chama o método que implementamos.

No nosso caso, para fazer um método disparar ao clique do botão, usamos a interface `ActionListener`. Essa interface nos dá um método `actionPerformed`:

```
public void actionPerformed(ActionEvent e) {  
    // ... tratamento do evento aqui...  
}
```

Agora, precisamos indicar que essa ação (esse `ActionListener`) deve ser disparada quando o botão for clicado. Fazemos isso através do método `addActionListener`, chamado no botão. Ele recebe como argumento um objeto que implementa `ActionListener`:

```
ActionListener nossoListener = ???;  
botao.addActionListener(nossoListener);
```

A dúvida que fica é: onde implemento meu `ActionListener`? A forma mais simples e direta seria criar uma nova classe normal que implemente a interface `ActionListener` e tenha o método `actionPerformed`. Depois basta dar `new` nessa classe e passar para o botão.

Essa forma funciona consideravelmente bem para botões que têm o mesmo comportamento em diversas telas do sistema, mas torna-se um problema para aqueles botões que devem chamar métodos diferentes de acordo com o contexto da página onde estão inseridos.

Pense, por exemplo em um botão de *Ok*: os métodos que precisam ser chamados quando um botão com esse texto é clicado podem ser os mais variados! Nesses casos, que são a maioria, o mais comum é encontrar a implementação dos `ActionListeners` através de classes internas e anônimas.

5.10 CLASSES INTERNAS E ANÔNIMAS

Em uma tela grande frequentemente temos muitos componentes que disparam eventos e muitos eventos diferentes. Temos que ter um objeto de listener para cada evento e temos que ter classes diferentes para cada tipo de evento disparado.

Só que é muito comum que nossos listeners sejam bem curtos, em geral chamando algum método da lógica de negócios ou atualizando algum componente. E, nesses casos, seria um grande problema de manutenção criarmos uma classe *top level* para cada evento.

Voltando ao exemplo do botão *Ok*, seria necessário criar uma classe para cada um deles e, aí, como diferenciar os nomes desses botões: nomes de classes diferentes? Pacotes diferentes? Ambas essas soluções causam problemas de manutenibilidade.

O mais comum para tais casos é criarmos **classes internas** que manipula os componentes que desejamos tratar junto à classe principal.

Classes internas são classes declaradas dentro de outras classes:

```
public class Externa {  
    public class Interna {  
  
    }  
}
```

Uma classe interna tem nome `Externa`. `Interna` pois faz parte do objeto da classe externa. A vantagem é não precisar de um arquivo separado e que classes internas podem acessar tudo que a externa possui (métodos, atributos etc). É possível até encapsular essa classe interna marcando-a como `private`. Dessa forma, apenas a externa pode enxergar.

Fora isso, são classes normais, que podem implementar interfaces, ter métodos, ser instanciadas etc.

Uma forma mais específica de classe interna é a chamada **classe anônima** que é muito vista em códigos com Swing. De forma simples, cria-se uma classe sem mesmo declarar seu nome em momento algum, isto é, o código `public class SairListener implements ActionListener{` não existirá em lugar algum.

Em um primeiro momento, a sintaxe das classes anônimas pode assustar - tente não criar preconceitos. Vamos usar uma classe anônima para tratar o evento de clique no botão que desliga a aplicação:

```
JButton botaoSair = new JButton("Sair");

ActionListener sairListener = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }
};

botaoSair.addActionListener(sairListener);
```

Repare que precisamos de um objeto do tipo `ActionListener` para passar para nosso botão. Normalmente criaríamos uma nova classe para isso e daríamos `new` nela. Usando classes anônimas damos `new` e implementamos a classe ao mesmo tempo, usando a sintaxe que vimos acima.

E podemos simplificar mais ainda sem a variável local `sairListener`:

```
JButton botaoSair = new JButton("Sair");
botaoSair.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }
});
```

Cuidado para não entender errado. Embora a sintaxe diga `new ActionListener()` sabemos que é impossível criar um objeto a partir de uma interface Java. O que essa sintaxe indica (e as chaves logo após o parêntesis indicam isso) é que estamos dando `new` em uma *nova classe* que implementa a interface `ActionListener` e possui o método `actionPerformed` cuja implementação chama o `System.exit(0)`.

Como criamos uma classe no momento que precisamos, é comum nos perguntarmos qual é o nome dessa classe criada. A resposta para essa pergunta é exatamente a esperada: não sabemos! Ninguém definiu o nome para essa implementação particular de um `ActionListener`, então, por não ter nome definido, classes como essas são chamadas de **classes anônimas**.

5.11 EXERCÍCIOS: NOSSA PRIMEIRA TELA

- 1) Vamos começar nossa interface gráfica pela classe chamada `ArgentumUI` que estará no pacote `br.com.caelum.argentum.ui` terá um método `montaTela` que desenha a tela em Swing e um método `main` que apenas dispara a montagem da tela:

```
1 public class ArgentumUI {  
2  
3     public static void main(String[] args) {  
4         new ArgentumUI().montaTela();  
5     }  
6  
7     private void montaTela() {  
8         // TODO Auto-generated method stub  
9  
10    }  
11 }
```

- 2) É boa prática quando usamos Swing quebrar a declaração dos componentes em pequenos métodos que fazem tarefas simples. Vamos usar os componentes e conceitos vistos até aqui para criar nossa tela organizada.

Para começar, vá ao método `montaTela()` e preencha-o com as seguintes chamadas aos métodos que formarão a tela do Argentum.

```
public void montaTela() {  
    preparaJanela();  
    preparaPainelPrincipal();  
    preparaBotaoCarregar();  
    preparaBotaoSair();  
    mostraJanela();  
}
```

Volte, de baixo para cima, e, com ajuda do **ctrl + 1**, faça com que o Eclipse crie os esqueletos dos métodos para nós.

- 3) Preencha o método `preparaJanela` com a criação do `JFrame` e, como essa é a principal janela da aplicação, configure-a para terminar o programa Java quando a fecharem. Aproveite para **preencher** também o método `mostraJanela`, organizando os componentes (`pack`), configurando o tamanho (`setSize`) e mostrando-a (`setVisible`).

Como é necessário utilizar o `JFrame` no método `mostraJanela`, será preciso que a janela seja um **atributo** em vez de uma variável local. Corrija os erros de compilação que o Eclipse vai acusar usando o **ctrl + 1** *Create field janela*.

Ao fim, seu código deve estar parecido com o seguinte:

```
public class ArgentumUI {  
    private JFrame janela;  
  
    // main e montaTela  
  
    private void preparaJanela() {  
        janela = new JFrame("Argentum");
```

```
janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
}  
  
// outros metodos prepara...  
  
private void mostraJanela() {  
    janela.pack();  
    janela.setSize(540, 540);  
    janela.setVisible(true);  
}  
}
```

- 4) Agora, só falta preencher os métodos que preparam os componentes da tela: o `painelPrincipal`, que conterá os botões, e cada um dos botões com seus `ActionListeners`. Lembre-se, também, que em cada um desses métodos será necessário dar `new` no componente e adicioná-lo ao seu componente pai.

Note que, como será necessário adicionar os botões ao `painelPrincipal`, este também deve ser criado como um atributo. Novamente, use o **ctrl + 1** *Create field painelPrincipal*.

Comece **preenchendo** o método `preparaPainelPrincipal`:

```
private void preparaPainelPrincipal() {  
    painelPrincipal = new JPanel();  
    janela.add(painelPrincipal);  
}
```

- 5) O código dos métodos que criam os botões, seus `ActionListeners` e adicionam ao `painelPrincipal`. Ao fim, eles devem estar assim:

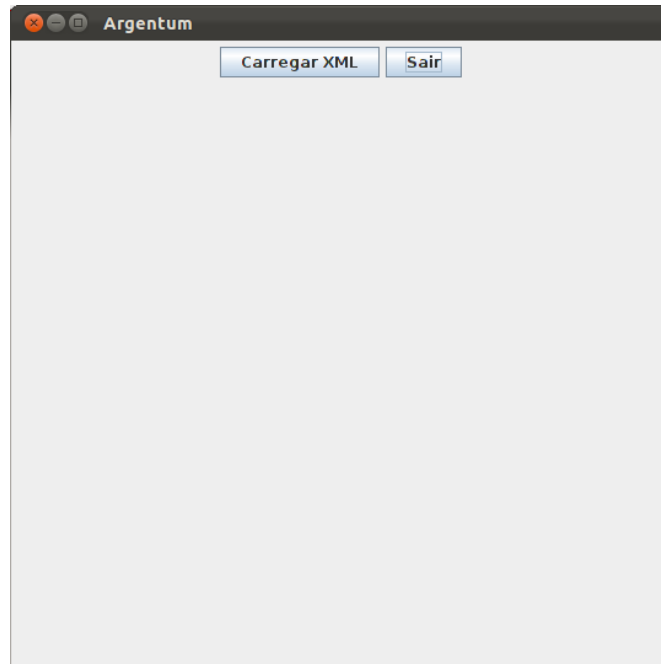
```
private void preparaBotaoCarregar() {  
    JButton botaoCarregar = new JButton("Carregar XML");  
    botaoCarregar.addActionListener(new ActionListener() {  
        @Override  
        public void actionPerformed(ActionEvent e) {  
            new EscolhedorDeXML().escolhe();  
        }  
    });  
    painelPrincipal.add(botaoCarregar);  
}
```

```
private void preparaBotaoSair() {  
    JButton botaoSair = new JButton("Sair");  
    botaoSair.addActionListener(new ActionListener() {  
        @Override  
        public void actionPerformed(ActionEvent e) {  
            System.exit(0);  
        }  
    });  
}
```



```
});  
painelPrincipal.add(botaoSair);  
}
```

- 6) O resultado final tem bastante código, mas temos tudo bem separado em pequenos métodos. Rode a classe acima e você deve ter o seguinte resultado:



Teste os botões e o disparo dos eventos.

- 7) (opcional) Diversos usuários preferem acessar a aplicação somente através de teclado. Para esses, é essencial que configuremos o atalho que chamará cada botão. Esse caracter que, somado ao *alt*, equivale a um clique do mouse é chamado de *mnemonic* do botão.

Explore os objetos do tipo `JButton` para que **alt + s** saia da aplicação e **alt + c** abra o escolhedor de XML.

- 8) (opcional) Procure sobre a forma de trocar a aparência da sua aplicação em Swing. A classe responsável por isso é a `UIManager` . Procure sobre como mudar o *Look and Feel* do Argentum.

Aproveite para conhecer um projeto *opensource* chamado **NapkinLaF**, que provê uma aparência bem diferente ao seu projeto!

LAYOUT MANAGERS E SEPARAÇÃO

No próximo capítulo, veremos porque os componentes ficaram dispostos dessa maneira e como mudar isso. Veremos também como organizar melhor esse nosso código.

5.12 JTable

Para completar a primeira fase da tela, ainda queremos mostrar os resultados das negociações em uma tabela de 3 colunas: preço, quantidade e data do negócio.

Para mostrarmos tabelas em Swing, usamos o **JTable** que é um dos componentes mais complexos de todo o Java. Tanto que tem um pacote `javax.swing.table` só para ele.

Um `JTable` é extremamente flexível. Ele serve para exibição e edição de dados, pode ter outros componentes dentro, reorganizar as colunas, fazer drag and drop, ordenação, entre outros.

Basicamente, o que precisamos é o nosso `JTable`, que representa a tabela, e um `TableModel`, que representa os dados que queremos exibir na tabela. Podemos ainda definir muitas outras configurações, como o comportamento das colunas com `TableColumn`.

Começamos criando o `JTable`:

```
JTable table = new JTable();

// por padrão, vem sem bordas, então colocamos:
table.setBorder(new LineBorder(Color.black));
table.setGridColor(Color.black);
table.setShowGrid(true);
```

Um `JTable` não tem comportamento de rolagem para tabelas muito grandes. Mas podemos adicionar esse comportamento compondo com um `JScrollPane`:

```
JScrollPane scroll = new JScrollPane();
scroll.getViewport().setBorder(null);
scroll.getViewport().add(table);
scroll.setSize(450, 450);
```

Ou seja, adicionamos a tabela ao *scrollpane*. Depois esse painel com barra de rolagem será adicionado ao `painelPrincipal`.

5.13 IMPLEMENTANDO UM TABLEMODEL

Um table model é responsável por devolver para a tabela os dados necessários para exibição. Há implementações com `Vectors` ou `Object[][]`. Ou o que é mais comum, podemos criar nosso próprio estendendo da classe `AbstractTableModel`.

Essa classe tem três métodos abstratos que somos obrigados a implementar:

- `getColumnCount` - devolve a quantidade de colunas

- `getRowCount` - devolve a quantidade de linhas
- `getValueAt(row, column)` - dada uma linha e uma coluna devolve o valor correspondente.

Podemos, então, criar uma `NegociosTableModel` simples que responde essas 3 perguntas baseadas em uma `List<Negocio>` que lemos do XML:

```
1 public class NegociosTableModel extends AbstractTableModel {
2
3     private final List<Negocio> negocios;
4
5     public NegociosTableModel(List<Negocio> negocios) {
6         this.negocios = negocios;
7     }
8
9     @Override
10    public int getColumnCount() {
11        return 3;
12    }
13
14    @Override
15    public int getRowCount() {
16        return negocios.size();
17    }
18
19    @Override
20    public Object getValueAt(int rowIndex, int columnIndex) {
21        Negocio n = negocios.get(rowIndex);
22
23        switch (columnIndex) {
24            case 0:
25                return n.getPreco();
26            case 1:
27                return n.getQuantidade();
28            case 2:
29                return n.getData();
30        }
31        return null;
32    }
33 }
```

Só isso já seria suficiente, mas queremos fazer o botão funcionar: ao clicar em carregar XML, precisamos pegar os dados do arquivo e popular a tabela, ou melhor, popular o `TableModel` responsável por representar os dados da tabela.

Primeiro, vamos fazer o método `escolhe` da classe `EscolhedorDeXML` devolver a `List<Negocio>` de que precisamos (em vez de manipulá-la para mostrar apenas o preço do primeiro negócio).

```
public class EscolhedorDeXML {

    public List<Negocio> escolhe() {
        try {
            JFileChooser chooser = new JFileChooser("/caelum/cursos/16");
            chooser.setFileFilter(new FileNameExtensionFilter("Apenas XML", "xml"));
            int retorno = chooser.showOpenDialog(null);

            if (retorno == JFileChooser.APPROVE_OPTION) {
                FileReader reader = new FileReader(chooser.getSelectedFile());
                return new LeitorXML().carrega(reader);
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        return Collections.emptyList();
    }
}
```

Agora, basta criar um objeto do `NegociosTableModel` na classe anônima que trata o evento:

```
botaoCarregar.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        List<Negocio> negocios = new EscolheXML().escolher();
        NegociosTableModel ntm = new NegociosTableModel(negocios);
        table.setModel(ntm);
    }
});
```

FINAL DE CLASSE ANÔNIMAS

Apenas um detalhe: para que nossa classe anônima acesse variáveis locais do método que a cria, essas variáveis **precisam** ser **final**. Portanto, vamos precisar marcar a variável `table` como **final** para que o código acima funcione.

5.14 EXERCÍCIOS: TABELA

- 1) Antes de colocarmos a tabela, vamos **alterar** a classe `EscolhedorDeXML` para devolver a lista de negócios em vez de mostrar o pop-up de mensagem. **Altere** o método `escolhe` para devolver uma lista de negócios. Note que tanto a linha 9 quanto a 14 são alterações!

```
1 public List<Negocio> escolhe() {
2     try {
```

```
3      JFileChooser chooser = new JFileChooser("/caelum/cursos/16");
4      int retorno = chooser.showOpenDialog(null);
5
6      if (retorno == JFileChooser.APPROVE_OPTION) {
7          FileReader reader =
8              new FileReader(chooser.getSelectedFile());
9          return new LeitorXML().carrega(reader);
10     }
11 } catch (FileNotFoundException e) {
12     e.printStackTrace();
13 }
14 return Collections.emptyList();
15 }
```

COLLECTIONS.EMPTYLIST()

A classe Collections tem ótimos métodos para trabalharmos com diversas coleções. No exemplo acima, usamos o `emptyList()` para devolver uma lista vazia genérica - no caso, achamos essa uma opção melhor do que devolver `null`.

- 2) Na classe `ArgentumUI`, localize o método `montaTela()` (use o `ctrl + O` para isso) e **adicione a chamada** a esse método *dentro* do método `montaTela` logo acima das chamadas aos botões:

```
private void montaTela() {
    preparaJanela();
    preparaPainelPrincipal();
    preparaTabela();           // linha adicionada!
    preparaBotaoCarregar();
    preparaBotaoSair();
    mostraJanela();
}
```

- 3) Essa nova linha não compila porque o método ainda não existe. Use o `ctrl + 1` para que o Eclipse crie o método para você. Então, preencha o método com a criação da tabela e o painel com barra de rolagem.

Atente para o detalhe de que a tabela é um atributo (*field*) e não uma variável local (*local variable*): uma das opções que o `ctrl + 1` oferece é para transformar essa variável em atributo.

```
1 private void preparaTabela() {
2     tabela = new JTable();
3
4     JScrollPane scroll = new JScrollPane();
5     scroll.setViewportView().add(tabela);
6 }
```

```
7    painelPrincipal.add(scroll);
8 }
```

- 4) Crie a classe `NegociosTableModel` que é um `AbstractTableModel` e sobrescreva os métodos obrigatórios. Para isso, faça `ctrl + N` para criar a classe, indique o pacote `br.com.caelum.argentum.ui` e indique a `AbstractTableModel` como superclasse. **Altere** a implementação dos métodos para lidar com o tipo `Negocio`, adicione o atributo que falta e gere o construtor de acordo:

```
1 public class NegociosTableModel extends AbstractTableModel {
2
3     private final List<Negocio> lista;
4
5     // ctrl + 3 constructor
6
7     @Override
8     public int getRowCount() {
9         return lista.size();
10    }
11
12    @Override
13    public int getColumnCount() {
14        return 3;
15    }
16
17    @Override
18    public Object getValueAt(int linha, int coluna) {
19        Negocio negocio = lista.get(linha);
20        switch(coluna) {
21            case 0:
22                return negocio.getPreco();
23            case 1:
24                return negocio.getQuantidade();
25            case 2:
26                return negocio.getData();
27        }
28        return "";
29    }
30 }
```

- 5) Agora, precisamos **alterar** a classe anônima que trata o evento do botão de carregar XML para pegar a lista de negócios e passá-la para um `NegociosTableModel`, que será passado para a tabela:

```
JButton botaoCarregar = new JButton("Carregar XML");
botaoCarregar.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        List<Negocio> lista = new EscolhedorDeXML().escolhe();
```

```

    NegociosTableModel ntm = new NegociosTableModel(lista);
    tabela.setModel(ntm);
  }
});

```

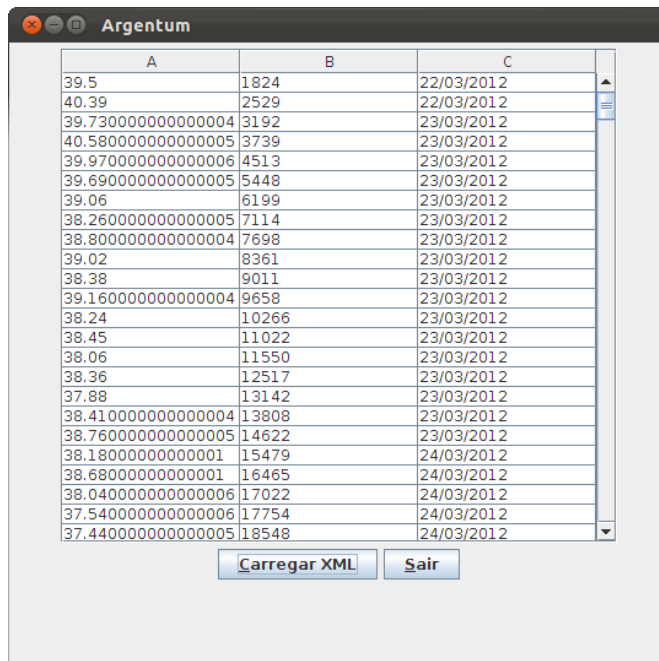
- 6) Se você rodar a aplicação agora, verá que a data ainda não está devidamente formatada! Modifique o método `getValueAt` para devolver a data formatada utilizando o `SimpleDateFormat`:

```

@Override
public Object getValueAt(int linha, int coluna) {
    Negocio negocio = lista.get(linha);
    switch(coluna) {
        case 0:
            return negocio.getPreco();
        case 1:
            return negocio.getQuantidade();
        case 2:
            SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
            return sdf.format(negocio.getData().getTime());
    }
    return "";
}

```

- 7) Rode a aplicação e veja o resultado:



A	B	C
39.5	1824	22/03/2012
40.39	2529	22/03/2012
39.7300000000000004	3192	23/03/2012
40.5800000000000005	3739	23/03/2012
39.9700000000000006	4513	23/03/2012
39.6900000000000005	5448	23/03/2012
39.06	6199	23/03/2012
38.2600000000000005	7114	23/03/2012
38.8000000000000004	7698	23/03/2012
39.02	8361	23/03/2012
38.38	9011	23/03/2012
39.1600000000000004	9658	23/03/2012
38.24	10266	23/03/2012
38.45	11022	23/03/2012
38.06	11550	23/03/2012
38.36	12517	23/03/2012
37.88	13142	23/03/2012
38.4100000000000004	13808	23/03/2012
38.7600000000000005	14622	23/03/2012
38.1800000000000001	15479	24/03/2012
38.6800000000000001	16465	24/03/2012
38.0400000000000006	17022	24/03/2012
37.5400000000000006	17754	24/03/2012
37.4400000000000005	18548	24/03/2012

5.15 EXERCÍCIOS OPCIONAIS: MELHORANDO A APRESENTAÇÃO

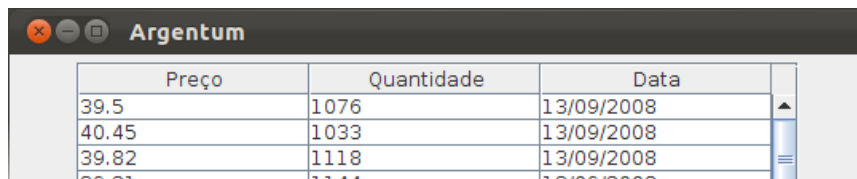
- 1) (Opcional) Há alguns detalhes que podemos ainda melhorar. Por exemplo, podemos mudar os nomes das colunas para algo que faça mais sentido. Vá à classe `NegociosTableModel` e **sobrescreva** o método `getColumnName`:

```

1 @Override
2 public String getColumnName(int column) {
3     switch (column) {
4         case 0:
5             return "Preço";
6         case 1:
7             return "Quantidade";
8         case 2:
9             return "Data";
10    }
11    return "";
12 }

```

Rode a aplicação e veja o resultado:



Preço	Quantidade	Data
39.5	1076	13/09/2008
40.45	1033	13/09/2008
39.82	1118	13/09/2008

- 2) (opcional) Vamos adicionar um título à nossa janela, usando um `JLabel`. Na classe `ArgentumUI`, crie o componente como indicado abaixo e **não esqueça** de adicionar a chamada desse método ao `montaTela`:

```

private void preparaTitulo() {
    JLabel titulo = new JLabel("Lista de Negócios", SwingConstants.CENTER);
    titulo.setFont(new Font("Verdana", Font.BOLD, 25));
    painelPrincipal.add(titulo);
}

```

- 3) (opcional) Façamos com que a tabela formate também os valores monetários. Para isso, altere novamente o método `getValueAt` na classe `NegociosTableModel`:

```

@Override
public Object getValueAt(int linha, int coluna) {
    Negocio negocio = lista.get(linha);
    switch (coluna) {
        case 0:
            Locale brasil = new Locale("pt", "BR");
            NumberFormat formatadorMoeda =

```



```
        NumberFormat.getCurrencyInstance(brasil);
        return formatadorMoeda.format(negocio.getPreco());
    case 1:
        return negocio.getQuantidade();
    case 2:
        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
        return sdf.format(negocio.getData().getTime());
    }
    return "";
}
```

Rode novamente a aplicação e veja o resultado.

LOCALE

Se não passarmos o `Locale` para o `getCurrencyInstance`, ele vai usar o padrão da máquina virtual, o que pode ser bastante desejável se nossa aplicação for internacionalizada.

Contudo, o modo mais comum de forçar um `Locale` é alterá-lo em todo o sistema de uma vez só, usando seu método estático `setLocale` logo no início da aplicação. Você pode mudar isso colocando no método `main` da classe `ArgentumUI`:

```
public static void main(String[] args) {
    Locale.setDefault(new Locale("pt", "BR"));
    new ArgentumUI().montaTela();
}
```

5.16 PARA SABER MAIS

- Consultar o javadoc do Swing pode não ser muito simples. Por isso, a Sun disponibiliza um ótimo tutorial online sobre Swing, que hoje está hospedado no site da Oracle: <http://docs.oracle.com/javase/tutorial/uiswing/>
- Aplicações grandes com Swing podem ganhar uma complexidade enorme e ficar difíceis de manter. Alguns projetos tentam minimizar esses problemas; há, por exemplo, o famoso projeto *Thinlet*, onde um XML substitui classes como a nossa `ArgentumUI`.
- Existem diversos frameworks, como o JGoodies, que auxiliam na criação de telas através de APIs teoricamente mais simples que o Swing. Não são editores, mas sim bibliotecas mais amigáveis. Apesar da existência de tais bibliotecas, é importante conhecer as principais classes e interfaces e seu funcionamento no Swing.

5.17 DISCUSSÃO EM SALA DE AULA: LISTENERS COMO CLASSES TOP LEVEL, INTERNAS OU ANÔNIMAS?

CAPÍTULO 6

Refatoração: os Indicadores da bolsa

“Nunca confie em um computador que você não pode jogar pela janela.”

– Steve Wozniak

6.1 ANÁLISE TÉCNICA DA BOLSA DE VALORES

A **Análise Técnica Grafista** é 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.

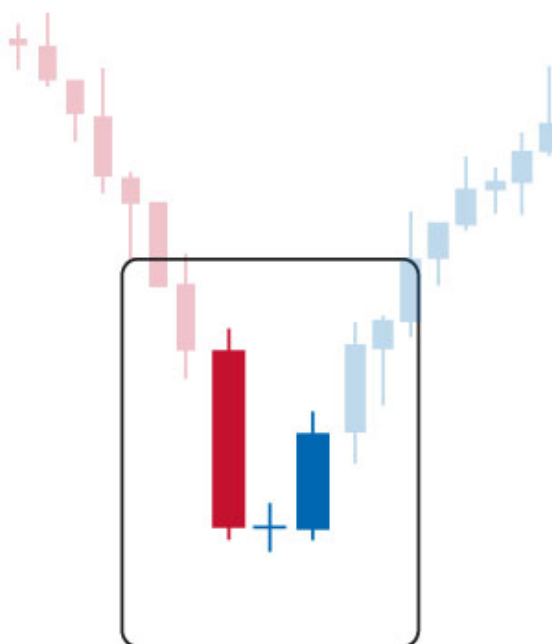
Essa forma de análise dos dados gerados sobre dados das negociações (preço, volume, etc), usa gráficos na busca de padrões e faz análise de tendências para tentar prever o comportamento futuro de uma ação.

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.

Munehisa Homma, no século 18, foi o primeiro 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:

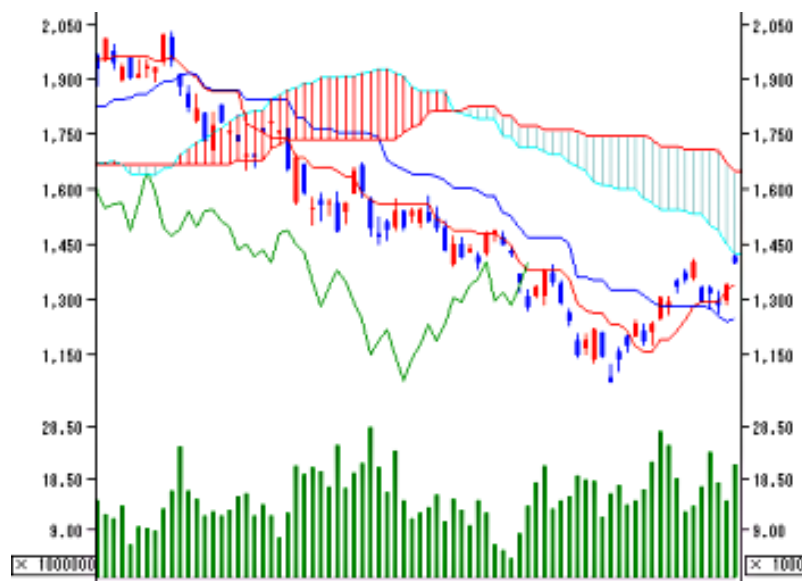


Ela indica um padrão de reversão. Dizem que quando o preço de abertura e fechamento é praticamente igual (a estrela), essa é uma forte indicação de que o mercado se inverta, isto é, se estava em uma grande **baixa**, tenderá a **subir** e, se estava em uma grande **alta**, tenderá a **cair**.

6.2 INDICADORES TÉCNICOS

Uma das várias formas de se 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*:



Uma lista com os indicadores mais usados e como calculá-los pode ser encontrada em: http://stockcharts.com/school/doku.php?id=chart_school:technical_indicators

Diversos livros também são publicados sobre o assunto. Os principais *homebrokers* fornecem softwares que traçam esses indicadores e muitos outros.

6.3 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 indicador do papel (em geral o valor de fechamento) para um determinado intervalo de tempo. 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.

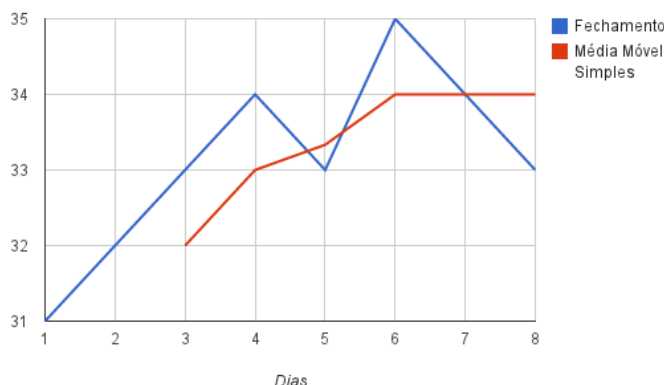


Repare que a média móvel mais ‘curta’, a de 50 dias, responde mais rápido aos movimentos atuais da ação, mas pode gerar sinais errados a médio prazo.

Em geral, 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: $(5 + 3 + 4) / 3 = 4$. A média móvel do dia 3 para os últimos 3 dias é 2: $(3 + 2 + 1) / 3$. E assim por diante.



O gráfico anterior das médias móveis da Sun pega, para cada dia do gráfico, a média dos 50 dias anteriores.

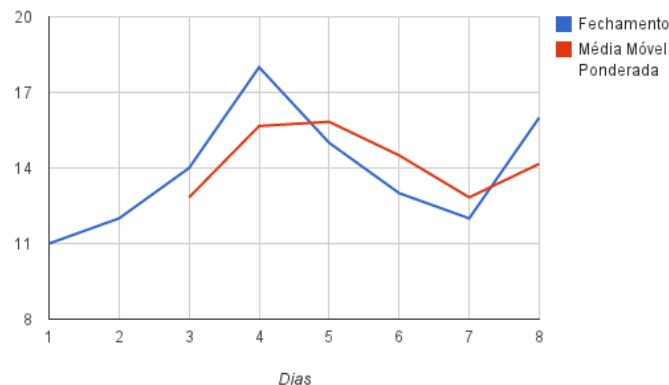
Média móvel ponderada

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: $(13 \cdot 3 + 15 \cdot 2 + 18 \cdot 1) / 6 = 14.50$.



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.

6.4 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`:

```
1 public class SerieTemporal {
2
3     private final List<Candlestick> candles;
4
5     public SerieTemporal(List<Candlestick> candles) {
6         this.candles = candles;
7     }
8
9     public Candlestick getCandle(int i) {
10         return this.candles.get(i);
11     }
12
13     public int getTotal() {
14         return this.candles.size();
15     }
16 }
```


- 2) Vamos criar a média móvel, dentro do pacote `br.com.caelum.argentum.indicadores`. Essa classe terá o método `calcula` que recebe a posição a ser calculada e a `SerieTemporal` que proverá as 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 idéia 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 idéia 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 da `MediaMovelSimples` na source folder `src/test/java`.

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 tem tais valores. Criaremos uma classe para auxiliar nesses testes na sequência. Por ora, 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` para que ele crie a classe para você e, então, adicione a ela o método `criaSerie`, usando o recurso de *varargs* 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):

```
1 /**
2  * Serve para ajudar a fazer os testes.
3  *
4  * Recebe uma sequência de valores e cria candles com abertura, fechamento,
5  * minimo e maximo iguais, mil de volume e data de hoje. Finalmente, devolve
6  * tais candles encapsuladas em uma Serie Temporal.
7  */
8 public static SerieTemporal criaSerie(double... valores) {
9     List<Candlestick> candles = new ArrayList<Candlestick>();
10    for (double d : valores) {
11        candles.add(new Candlestick(d, d, d, d, 1000,
12                                   Calendar.getInstance()));
13    }
14    return new SerieTemporal(candles);
15 }
```

Agora que ele compila, rode o teste. **Ele falha**, já que a implementação padrão simplesmente devolve zero!

- 5) **Volte** à classe principal `MediaMoveISimples` e **implemente** agora a lógica de negócio do método `calcula`,

que já existe. O método deve ficar parecido com o que segue:

```
1 public class MediaMovelSimples {
2
3     public double calcula(int posicao, SerieTemporal serie) {
4         double soma = 0.0;
5         for (int i = posicao - 2; i <= posicao; i++) {
6             Candlestick c = serie.getCandle(i);
7             soma += c.getFechamento();
8         }
9         return soma / 3;
10    }
11 }
```

Repare que iniciamos o for com `posicao - 2`. Isso significa que estamos calculando a média móvel dos últimos 3 dias. Mais para frente, existe um exercício opcional para parametrizar esse valor.

- 6) (opcional) Crie um teste de unidade em uma nova classe `SerieTemporalTest`, que verifica que essa classe não pode receber uma lista nula.

Aproveite o momento para pensar quais outros testes poderiam ser feitos para essa classe.

- 7) (opcional) 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. Ela 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);
    }
}
```

Dica: o código interno é muito parecido com o da média simples, só precisamos multiplicar sempre pela quantidade de dias passados. Se precisar de ajuda nessa implementação, veja a resposta no final da apostila.

6.5 REFATORAÇÃO

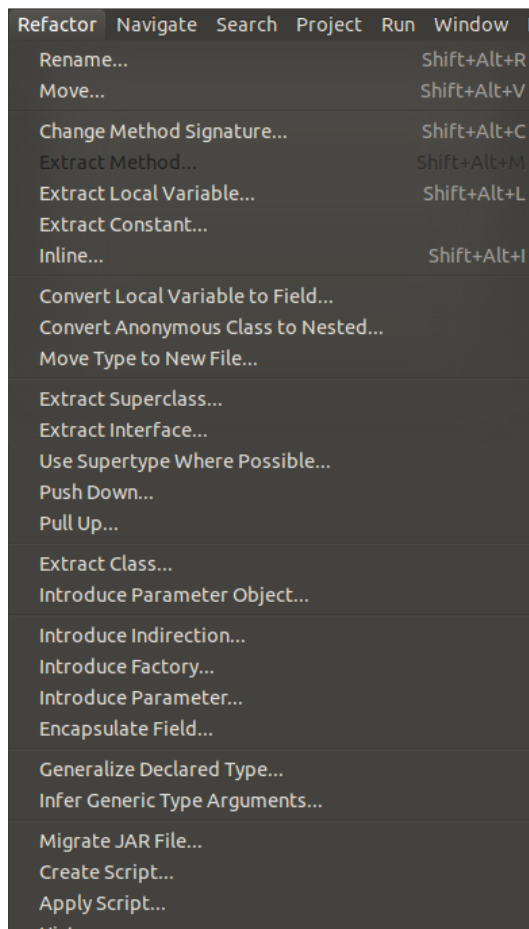
*Refatoração é uma técnica controlada para reestruturar um trecho de código existente, alterando sua estrutura interna sem modificar seu comportamento externo. Consiste em uma série de pequenas transformações que preservam o comportamento inicial. Cada transformação (chamada de refatoração) reflete em uma pequena mudança, mas uma sequência de transformações pode produzir uma significativa 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.

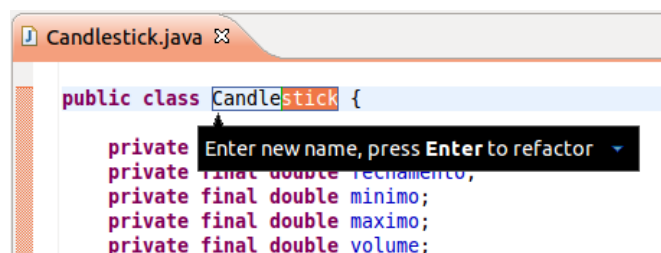
Algumas são tão corriqueiras, que o próprio Eclipse inclui um menu com diversas refatorações que ele é capaz de fazer por você:



6.6 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`.

Há várias formas de se fazer isso no Eclipse. Pelo *Package Explorer*, podemos selecionar a classe e apertar F2. Ou, se estivermos dentro da classe, podemos colocar o cursor sobre o nome dela e usar o atalho `alt + shift + R`, que renomeia classes, 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 - 2; i <= posicao; i++) {  
4         Candle c = serie.getCandle(i);  
5         soma += c.getFechamento();  
6     }  
7     return soma / 3;  
8 }
```

O código, então ficará assim:

```
public double calcula(int posicao, SerieTemporal serie) {  
    double soma = 0.0;  
    for (int i = posicao - 2; i <= posicao; i++) {  
        soma += serie.getCandle(i).getFechamento();  
    }  
    return soma / 3;  
}
```

Não esqueça de tirar os imports desnecessários (`ctrl + shift + O`)!

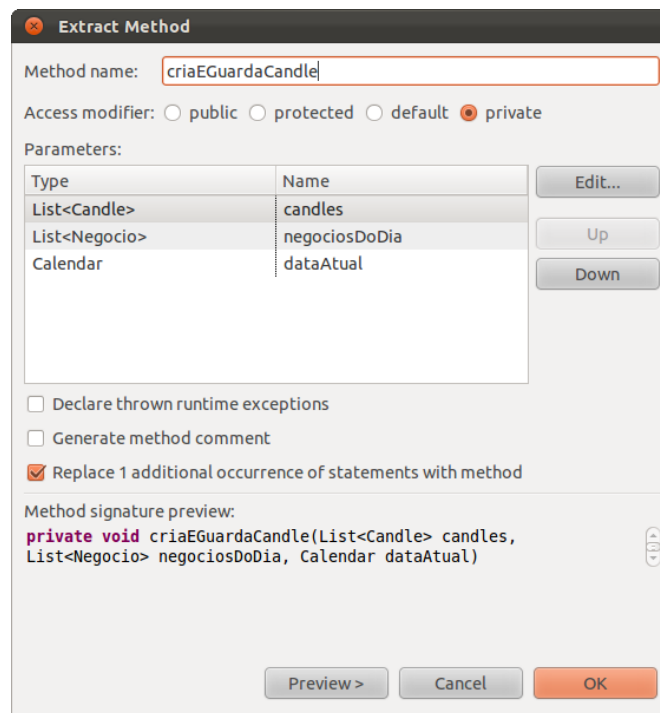
- 3) Finalmente, abra a classe `CandlestickFactory` e observe o método `constroiCandles`. Escrevemos esse método no capítulo de XML com um algoritmo para separar todos os negócios em vários candles.

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, negociosDoDia);  
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) (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`.

6.7 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 complexos as mudanças para chegar ao resultado final, mais importante é **quebrar essa refatoração em pedaços pequenos e rodar os testes a cada passo**.

No final dos 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...

6.8 DISCUSSÃO EM AULA: QUANDO REFATORAR?

CAPÍTULO 7

Gráficos com JFreeChart

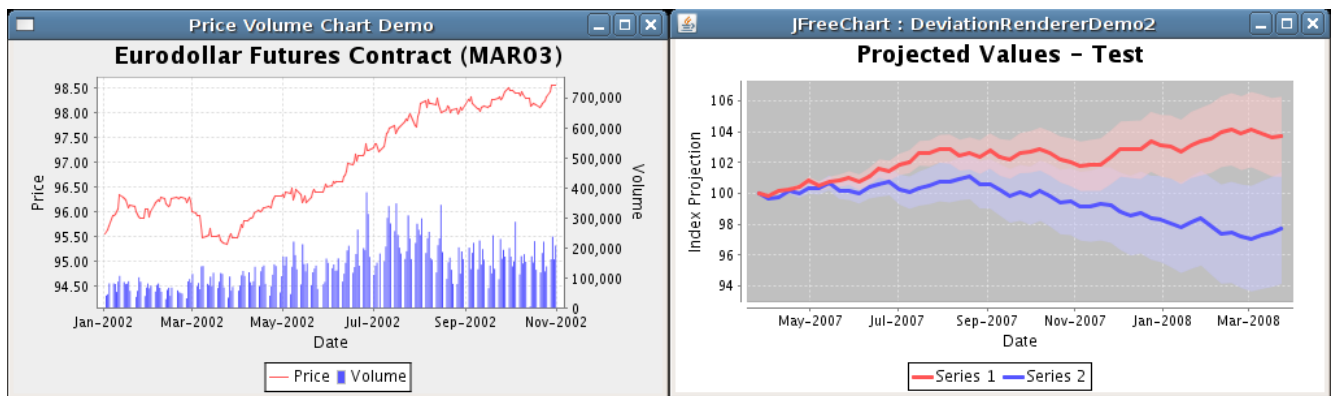
É comum nos depararmos com necessidades de geração de relatórios que, frequentemente traz informações na forma de gráficos, entre outras. E, como qualquer outra necessidade comum a desenvolvedores Java, já existe uma biblioteca capaz de lidar com dados de uma forma muito mais simples do que utilizando diretamente a API do Java.

7.1 JFREECHART



O **JFreeChart** é hoje a biblioteca mais famosa para desenho de gráficos. É um projeto de software livre iniciado em 2000 e que tem ampla aceitação pelo mercado, funcionando até em ambientes 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.



Sua licença é LGPL o que permite ser usada em projetos de código fechado. 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.

Nós vamos usar o JFreeChart para gerar os gráficos da nossa análise gráfica.

7.2 UTILIZANDO O JFREECHART

Com nosso projeto todo configurado, vamos agora programar usando a API do JFreeChart. É fundamental quando programamos com alguma biblioteca, ter acesso ao Javadoc da mesma para saber quais classes e métodos usar, pra que servem os parâmetros e etc.

O Javadoc do JFreeChart pode ser acessado aqui: <http://www.jfree.org/jfreechart/api/javadoc/>

A classe principal é a `org.jfree.chart.JFreeChart`, ela representa um gráfico e pode assumir vários formatos (torta, barra, pontos, linhas etc). Trabalhar diretamente com essa classe é um pouco trabalhoso e a biblioteca optou por criar uma classe com uma série de métodos estáticos para facilitar o trabalho de criação dos gráficos: são métodos que se comportam como uma fábrica! Esses, também são um design pattern, chamado *Factory method*. A classe que tem tais métodos no JFreeChart é a `org.jfree.chart.ChartFactory`.

Queremos criar um gráfico de linha. Consultando o Javadoc da `ChartFactory`, descobrimos o método `createLineChart` que devolve um objeto do tipo `JFreeChart` prontinho.

Este método recebe alguns argumentos:

- um título (`String`);
- duas `Strings` com os labels dos eixos X e Y;
- os dados a serem exibidos (do tipo `CategoryDataset`);

- orientação da linha (PlotOrientation.VERTICAL ou HORIZONTAL);
- um boolean indicando se queremos legenda ou não;
- um boolean dizendo se queremos tooltips;
- um boolean para exibir URLs no gráfico ou não.

Repare que poderíamos descobrir tudo isso pelo Javadoc!

O principal aqui é o `CategoryDataset`, justamente o conjunto de dados que queremos exibir no gráfico. `CategoryDataset` é uma interface e há várias implementações: por exemplo, uma simples onde adicionamos os elementos manualmente, outra para trabalhar direto com banco de dados (JDBC) e outras.

Vamos usar a `DefaultCategoryDataset`, a implementação padrão e mais fácil de usar:

```
// cria o conjunto de dados
DefaultCategoryDataset ds = new DefaultCategoryDataset();
ds.addValue(40.5, "maximo", "dia 1");
ds.addValue(38.2, "maximo", "dia 2");
ds.addValue(37.3, "maximo", "dia 3");
ds.addValue(31.5, "maximo", "dia 4");
ds.addValue(35.7, "maximo", "dia 5");
ds.addValue(42.5, "maximo", "dia 6");

// cria o gráfico
JFreeChart grafico = ChartFactory.createLineChart("Meu Grafico", "Dia",
    "Valor", ds, PlotOrientation.VERTICAL, true, true, false);
```

Repare que a classe `DefaultCategoryDataset` possui o método `addValue` que recebe o valor, o nome dessa linha (porque um gráfico pode ter várias) e a posição desse valor no eixo X.

Depois de ter criado o gráfico e preenchido o conjunto de dados, queremos salvá-lo em um arquivo, enviar via rede, exibir na tela, mandar pra impressora ou algo semelhante. A classe `org.jfree.chart.ChartUtilities` tem uma série de métodos para isso.

Talvez o método mais interessante, para nós, da `CharUtilities` seja o `writeChartAsPNG`. Veja o que diz o javadoc dele: Writes a chart to an output stream in PNG format.

E a lista de parâmetros ainda segundo o javadoc:

- *out* - the output stream (null not permitted).
- *chart* - the chart (null not permitted).
- *width* - the image width.
- *height* - the image height.

Ou seja: passamos o gráfico, o tamanho (altura e largura) e onde queremos que seja feita a saída dos dados do gráfico através de qualquer objeto que **seja um** `java.io.OutputStream`.

Assim o JFreeChart consegue escrever nosso gráfico em qualquer fluxo de saída de dados: seja um arquivo (`FileOutputStream`), seja enviando via rede (pela `Socket`), seja usando dinamicamente numa página web (usando `Servlets`) ou em qualquer outro lugar que suporte o padrão `OutputStream` genérico de envio de dados.

É o uso da API do `java.io` novamente que, através do polimorfismo, garante esse poder todo sem que o JFreeChart precise saber onde exatamente o gráfico será salvo.

Vamos usar esse método para salvar em um arquivo PNG:

```
OutputStream arquivo = new FileOutputStream("grafico.png");
ChartUtilities.writeChartAsPNG(arquivo, grafico, 550, 400);
fos.close();
```

Agora, se juntarmos tudo, teremos um programa que gera os gráficos em um arquivo.

7.3 ISOLANDO A API DO JFREECHART: 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 deixar o JFreeChart de lado e usar outra forma de gerar gráficos? Essas mudanças são fáceis se tivermos o código todo *espalhado* pelo nosso programa?

Os princípios de orientação a objetos e as boas práticas de programação podem nos ajudar nesses casos. Vamos **encapsular** a forma como o gráfico é criado dentro de uma classe, a `GeradorDeGrafico`.

Essa classe deve ser capaz de gerar um gráfico com os dados que quisermos e salvá-lo na saída que quisermos. Mas... se ela vai receber os dados para gerar o gráfico, como receberemos esses dados?

É fácil notar que não é uma boa idéia pedir um `CategoryDataset` para quem quiser usar o `GeradorDeGrafico`! Se tivesse que passar um `CategoryDataset`, no dia que precisássemos trocar o JFreeChart por uma alternativa o impacto dessa mudança será muito maior: afetaremos não somente essa classe, mas todas as que a utilizam.

Vamos encapsular inclusive a passagem dos dados para o gráfico: a partir de uma `SerieTemporal` e do intervalo que desejamos plotar, a própria classe `GeradorDeGrafico` se encarregará de *traduzir* essas informações em um `CategoryDataset`:

```
1 public class GeradorDeGrafico {
2
3     private SerieTemporal serie;
4     private int comeco;
5     private int fim;
```

```
6
7     public GeradorDeGrafico(SerieTemporal serie, int comeco, int fim) {
8         this.serie = serie;
9         this.comeco = comeco;
10        this.fim = fim;
11    }
12 }
```

E, quem for usar essa classe, fará:

```
SerieTemporal serie = criaSerie(1,2,3,4,5,6,7,8,8,9,9,4,3,2,2,2,2);
GeradorDeGrafico g = new GeradorDeGrafico(serie, 2, 10);
```

Repare como esse código de teste não possui nada que o ligue ao JFreeChart especificamente. O dia que precisarmos mudar de biblioteca, precisaremos mudar apenas a classe GeradorDeGrafico. **Encapsulamento!**

Vamos encapsular também a criação do gráfico e o salvamento em um fluxo de saída. Para criar o gráfico, precisamos colocar algumas informações como título e eixos. E, para desenhar a linha do gráfico, vamos chamar o `plotaMediaMovelSimples`.

```
1 public class GeradorDeGrafico {
2     private SerieTemporal serie;
3     private int comeco;
4     private int fim;
5
6     private DefaultCategoryDataset dados;
7     private JFreeChart grafico;
8
9     public GeradorDeGrafico(SerieTemporal serie, int comeco, int fim) {
10        this.serie = serie;
11        this.comeco = comeco;
12        this.fim = fim;
13        this.dados = new DefaultCategoryDataset();
14        this.grafico = ChartFactory.createLineChart("Indicadores",
15                                                    "Dias", "Valores",
16                                                    dados,
17                                                    PlotOrientation.VERTICAL,
18                                                    true, true, false);
19    }
20
21    public void plotaMediaMovelSimples() {
22        MediaMovelSimples ind = new MediaMovelSimples();
23        for (int i = comeco; i <= fim; i++) {
24            double valor = ind.calcula(i, serie);
25            dados.addValue(valor, ind.toString(), Integer.valueOf(i));
26        }
27    }
28 }
```

```
26     }
27 }
28
29 public void salvar(OutputStream out) throws IOException {
30     ChartUtilities.writeChartAsPNG(out, grafico, 500, 350);
31 }
32 }
```

Vamos analisar esse código em detalhes. O construtor recebe os dados da série temporal e, além atribuir essas informações aos atributos, ele ainda cria o objeto que receberá os dados e o JFreeChart em si - e, claro, guarda esses objetos em atributos.

O método `plotaMediaMovelSimples` cria um objeto da `MediaMovelSimples` e passa pela `serieTemporal` recebida para calcular o conjunto de dados que vão para o gráfico.

Por fim, o método `salva` coloca o gráfico em um `OutputStream` que recebe como argumento. É boa prática deixar nossa classe a mais genérica possível para funcionar com qualquer fluxo de saída. Quem decide onde realmente salvar (arquivo, rede etc) é quem chama o método dessa classe.

Veja como fica o programa de teste que usa essa classe:

```
SerieTemporal serie = GeradorDeSerie.criaSerie(1,2,3,4,5,6,7,8,8,
                                                9,9,4,3,2,2,2,2);
GeradorDeGrafico g = new GeradorDeGrafico(serie, 2, 15);
g.criaGrafico("Meu grafico");
g.plotaMediaMovelSimples();
g.salva(new FileOutputStream("saida.png"));
```

Note que, para quem usa o `GeradorDeGrafico` nem dá para saber que, internamente, ele usa o JFreeChart! É um código **encapsulado, flexível, pouco acoplado e elegante**: usa as boas práticas de OO.

7.4 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 nessa classe.

Ambos são chamados de **padrões de criação (creational patterns)**, pois nos ajudam a criar objetos complicados.

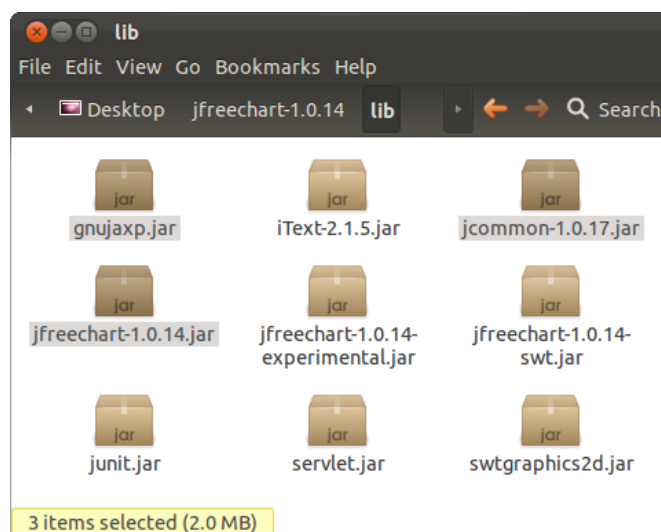
A factory é usada pelo JFreeChart na classe `ChartFactory`. A ideia é que criar um objeto JFreeChart diretamente é complicado. Então criaram um *método de fábrica* que encapsula essas complicações e já devolve o objeto prontinho para uso.

O padrão Builder é o que estamos usando na classe GeradorDeGrafico. Queremos encapsular a criação complicada de um gráfico e que pode mudar depois com o tempo (podemos querer usar outra API de geração de gráficos). 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.

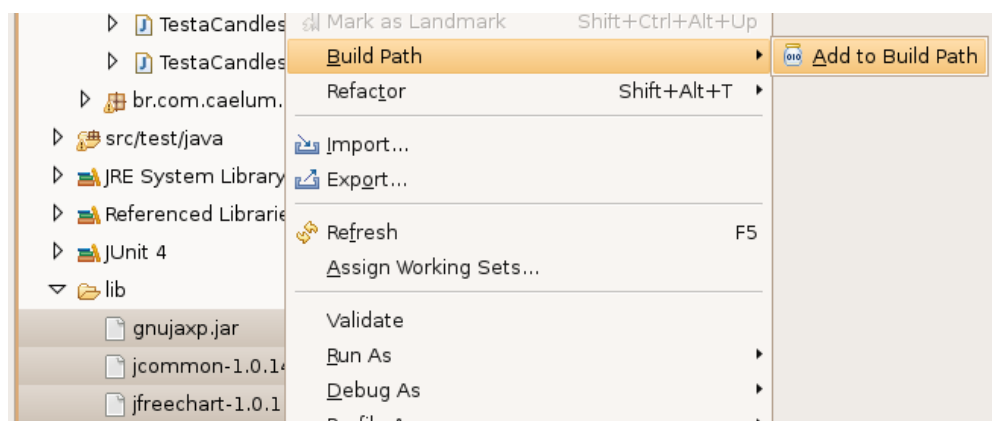
Leia mais sobre esses e outros Design Patterns no livro do GoF.

7.5 EXERCÍCIOS: JFREECHART

- 1) a) Abra a pasta caelum/16 no seu Desktop. Localize o zip do JFreeChart, **copie ele e cole** no Desktop;
- b) Depois, clique com o botão direito e escolha *Extract here*;
- c) Abra a pasta do JFreechart no seu Desktop e entre em lib. Copie **somente** os JARs **gnujaxp**, **jcommons** e **jfreechart** para a pasta lib do seu projeto:



- d) No Eclipse, selecione **os JARs** na pasta lib do projeto, clique da direita e escolha Build Path > Add to build path:



- 2) Crie a classe GeradorDeGrafico no pacote `br.com.caelum.argentum.grafico` para encapsular a criação dos gráficos.

Use os recursos do Eclipse para escrever esse código! Abuse do `ctrl + espaço`, do `ctrl + 1` e do `ctrl + shift + 0`.

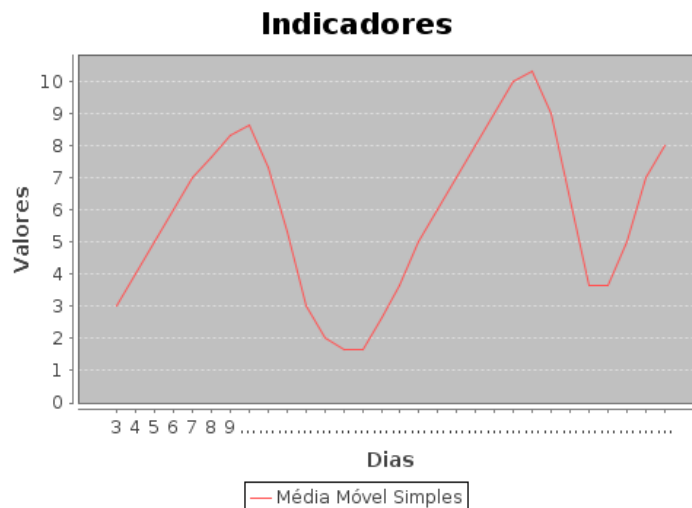
```
1 public class GeradorDeGrafico {
2     // atributos: serie, comeco, fim, grafico e dados
3     // (todos gerados com ctrl + 1, conforme você criar o construtor)
4
5     public GeradorDeGrafico(SerieTemporal serie, int comeco, int fim) {
6         this.serie = serie;
7         this.comeco = comeco;
8         this.fim = fim;
9         this.dados = new DefaultCategoryDataset();
10        this.grafico = ChartFactory.createLineChart("Indicadores",
11                                                    "Dias", "Valores", dados,
12                                                    PlotOrientation.VERTICAL,
13                                                    true, true, false);
14    }
15
16    public void plotaMediaMovelSimples() {
17        MediaMovelSimples ind = new MediaMovelSimples();
18        for (int i = comeco; i <= fim; i++) {
19            double valor = ind.calcula(i, serie);
20            dados.addValue(valor, ind.toString(), Integer.valueOf(i));
21        }
22    }
23
24    public void salva(OutputStream out) throws IOException {
25        ChartUtilities.writeChartAsPNG(out, grafico, 500, 350);
26    }
27 }
```

- 3) Escreva uma classe para gerar um gráfico com alguns dados de teste. Crie uma classe TestaGrafico no pacote `br.com.caelum.argentum.testes` com um main que usa nosso gerador:

```
1 public class TestaGrafico {
2     public static void main (String[] args) throws IOException {
3         SerieTemporal serie = GeradorDeSerie.criaSerie(1,2,3,4,5,6,7,
4                                                         8,8,9,9,4,3,2,1,2,2,4,5,6,7,
5                                                         8,9,10,11,10,6,3,2,6,7,8,9);
6         GeradorDeGrafico g = new GeradorDeGrafico(serie, 3, 32);
7         g.plotaMediaMovelSimples();
8         g.salva(new FileOutputStream("grafico.png"));
9     }
10 }
```


Repare que essa classe não faz import a *nenhuma classe do JFreeChart!* Conseguimos encapsular a biblioteca com sucesso, desacoplando-a do nosso código!

- 4) Rode a classe TestaGrafico no Eclipse. Dê um F5 no nome do projeto e veja que o arquivo **grafico.png** apareceu.



Pode parecer que o JFreeChart é lento, mas isso ocorre apenas na primeira vez que uma aplicação gerar um gráfico. Se você fosse gerar dois na mesma aplicação, o segundo gráfico seria gerado mais rapidamente.

- 5) Podemos em vez de criar um arquivo com a imagem, mostrá-la no Swing, para isso adicionamos um método que gera um JPanel dado um gráfico. Adicione o método na classe GeradorDeGrafico:

```
public JPanel getPanel() {
    return new ChartPanel(grafico);
}
```

E na classe TestaGrafico, adicione no final do main:

```
JFrame frame = new JFrame("Minha janela");
frame.add(g.getPanel());

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.pack();
frame.setVisible(true);
```

7.6 EXERCÍCIOS OPCIONAIS

- 1) Você pode associar um JAR ao seu respectivo código fonte para que o Eclipse possa te dar a documentação juntamente com o *autocomplete*, como o que já ocorre com as bibliotecas padrão.

Para isso, clique com o botão da direita na biblioteca desejada (dentro de *referenced libraries* do *package explorer*) e selecione propriedades. Agora selecione source location e associe-o ao diretório *src* do *jfree-*

chat. Você também poderia associar diretamente ao javadoc, caso ele estivesse presente em vez do código fonte. O Eclipse consegue extrair o JavaDoc de qualquer uma dessas duas formas.

- 2) Como fazer para mudar as cores do gráfico? Tente descobrir um jeito pela documentação do JFreeChart:

<http://www.jfree.org/jfreechart/api/javadoc/index.html>

7.7 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 da série. Nosso cliente certamente precisará de mais indicadores técnicos diferentes, como o de Média Móvel Ponderada ou ainda indicadores mais simples como, por exemplo, um indicador 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.

Então, com o indicador criado, precisaremos que o `GeradorDeGrafico` consiga plotar cada um desses gráficos também. Terminaremos com uma crescente classe `GeradorDeGrafico` com métodos **extremamente** parecidos:

```
public class GeradorDeGrafico {
    //...

    public void plotaMediaMovelSimples() {
        MediaMovelSimples ind = new MediaMovelSimples();
        for (int i = comeco; i <= fim; i++) {
            double valor = ind.calcula(i, serie);
            dados.addValue(valor, "Média Móvel Simples",
                            Integer.valueOf(i));
        }
    }

    public void plotaMediaMovelPonderada() {
        MediaMovelSimples ind = new MediaMovelSimples();
        for (int i = comeco; i <= fim; i++) {
            double valor = ind.calcula(i, serie);
            dados.addValue(valor, "Média Móvel Ponderada",
                            Integer.valueOf(i));
        }
    }

    public void plotaIndicadorFechamento() {
        IndicadorFechamento ind = new IndicadorFechamento();
        for (int i = comeco; i <= fim; i++) {
```

```
        double valor = ind.calcula(i, serie);
        dados.addValue(valor, "Indicador de fechamento",
                        Integer.valueOf(i));
    }
}

//...
}
```

Mas o maior problema é outro: cada vez que criarmos um indicador técnico diferente, teremos que criar um método novo na classe `GeradorDeGrafico`. Isso é uma indicação clássica de acoplamento no sistema.

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

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 nossas classes `MediaMovelSimples` e `MediaMovelPonderada` implementarem a interface `Indicador`. Com isso, podemos criar apenas um método na classe do gráfico que recebe um `Indicador` qualquer:

```
public class GeradorDeGrafico {
    public void plotaIndicador(Indicador ind) {
        for (int i = comeco; i <= fim; i++) {
            double valor = ind.calcula(i, serie);
            dados.addValue(valor, ind.toString(), Integer.valueOf(i));
        }
    }
}
```

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

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

A idéia de usar uma *interface comum* é ganhar *polimorfismo* e poder trocar os indicadores. Se usamos um Indicador, podemos trocar a classe específica sem mexer no nosso código, isto nos dá flexibilidade. Podemos inclusive criar novos indicadores que implementem a interface e passá-los para o gráfico sem que nunca mais mexamos na classe GeradorDeGrafico.

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, elas devolvem o valor do indicador. Esse é o design pattern chamado de **Strategy**.

DESIGN PATTERNS

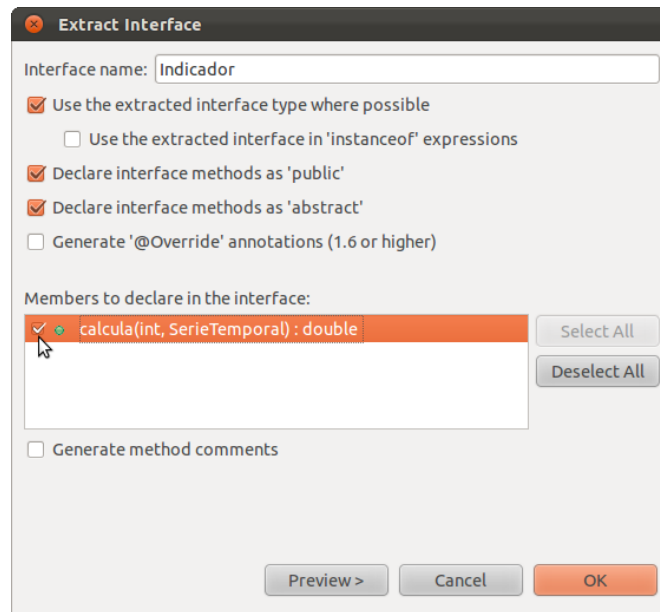
Design Patterns são aquelas soluções catalogadas para problemas clássicos de orientação a objetos, como este que temos no momento: encapsular e ter flexibilidade.

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 GeradorDeGrafico isolada das diferentes formas de cálculo dos indicadores.

7.8 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 vá use o **ctrl + 3** *Extract interface*. Selecione o método calcula e dê o nome da interface de Indicador:



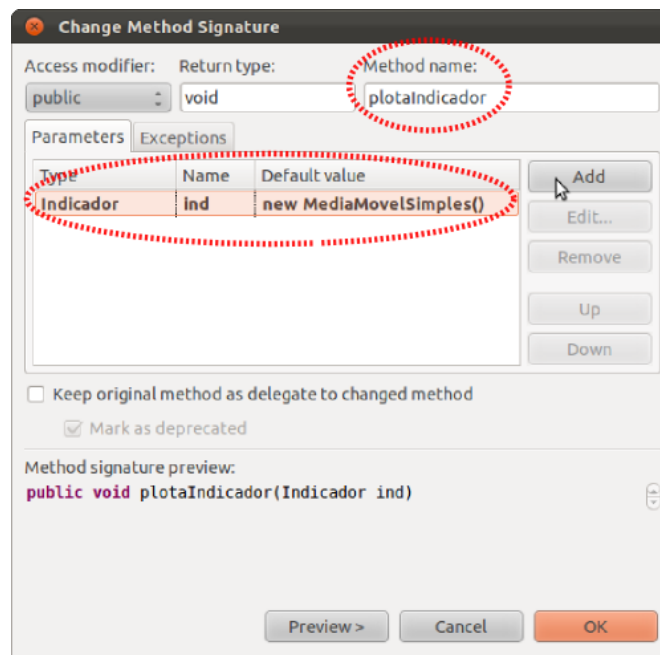
EFFECTIVE JAVA

Item 52: Refira a objetos pelas suas interfaces

- 2) Vamos criar também uma classe para, por exemplo, ser o indicador do preço de fechamento, o `IndicadorFechamento` que implementa a interface `Indicador`:

```
1 public class IndicadorFechamento implements Indicador {  
2  
3     @Override  
4     public double calcula(int posicao, SerieTemporal serie) {  
5         return serie.getCandle(posicao).getFechamento();  
6     }  
7 }
```

- 3) Volte para a classe `GeradorDeGrafico` 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 ind) {
    for (int i = comeco; i <= fim; i++) {
        double valor = ind.calcula(i, serie);
        dados.addValue(valor, "Média Móvel Simples",
                        Integer.valueOf(i));
    }
}
```

- 4) Finalmente, para que a legenda do gráfico apareça corretamente e para que as linhas não sobrescrevam umas às outras, é preciso trocar a String *Média Móvel Simples* para algo que diferencie um indicador do outro.

Já vimos isso antes! O método que usamos quando queremos que a representação em String de um objeto seja personalizada é o `toString()`. Utilizaremos ele, então, em vez da String *hard coded*:

```
public void plotaIndicador(Indicador ind) {
    for (int i = comeco; i <= fim; i++) {
        double valor = ind.calcula(i, serie);
        dados.addValue(valor, ind.toString(), Integer.valueOf(i));
    }
}
```

- 5) (opcional) Se você criou a classe *MediaMovelPonderada* no exercício anterior, coloque agora o `implements Indicador` nela.

Se quiser, crie também outros indicadores para preço de abertura e máximo.

- 6) Para ver o resultado dessa refatoração, abra a classe TestaGrafico e altere o método para adicionar a plotagem do IndicadorFechamento.

```
//...
GeradorDeGrafico g = new GeradorDeGrafico(serie, 3, 32);
g.plotaIndicador(new MediaMovelSimples());
g.plotaIndicador(new IndicadorFechamento());
//...
```

Rode a classe novamente e veja as duas linhas plotadas!

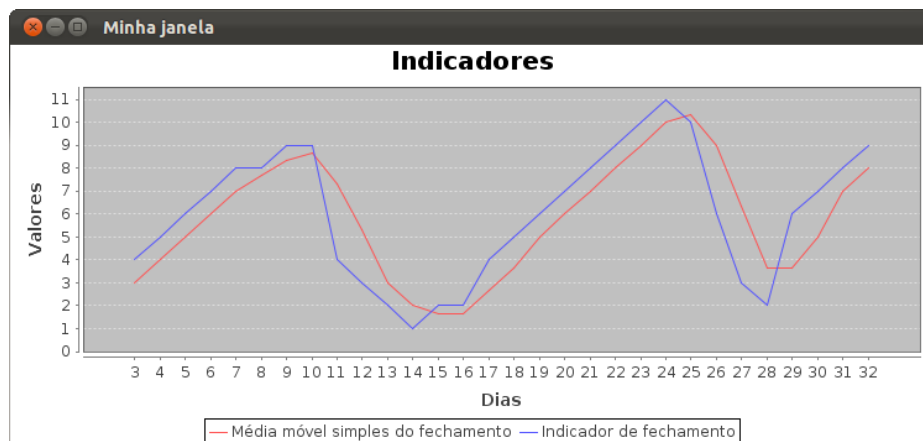
- 7) (opcional - interessante) Sobrescreva o toString nos seus indicadores para que a legenda do gráfico fique correta. Por exemplo, na MediaMovelSimples: **Arquivo: MediaMovelSimples.java**

```
public class MediaMovelSimples implements Indicador {
    //...

    @Override
    public String toString() {
        return "Média móvel simples do fechamento";
    }
}
```

Faça, também, similarmente no **indicador de fechamento**!

O resultado deve ser parecido com esse:



7.9 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) (trabalhoso) Fizemos o gráfico que exibe as médias móveis que nos interessam. Mas e quanto aos candles que geramos? Faça um gráfico usando JFreeChart que, dada uma lista de candles, gera o gráfico com os desenhos corretos dos candles (cores, traços, etc)

Dica: procure por *Candlestick* na API do JFreeChart.

- 3) 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.

Mais Swing: layout managers, mais componentes e detalhes

8.1 GERENCIADORES DE LAYOUT

Até agora, vínhamos adicionando novos componentes sem dar a informação da localização deles no código. Contudo... como o Java soube onde posicioná-los? Por que sempre são adicionados do lado direito? Se redimensionamos a tela (teste isso), os elementos *fluem* para a linha de cima. Por quê?

As partes do Swing responsáveis por dispor os elementos na tela são os **Layout Managers**, os gerenciadores de layout do Swing/AWT. O Java já vem com uma série de **Layouts** diferentes, que determinam como os elementos serão dispostos na tela, seus tamanhos preferenciais, como eles se comportarão quando a janela for redimensionada e muitos outros aspectos.

Ao escrever uma aplicação Swing, você deve indicar qual Layout Manager você deseja utilizar. Por padrão, é utilizado o `FlowLayout` que especifica que os elementos devem ser justapostos, que eles devem “fluir” um para baixo do outro quando não couberem lado a lado na tela redimensionada e etc.

Usando o `FlowLayout` padrão, teste redimensionar a janela de várias formas. Podemos acabar com disposições como essa:

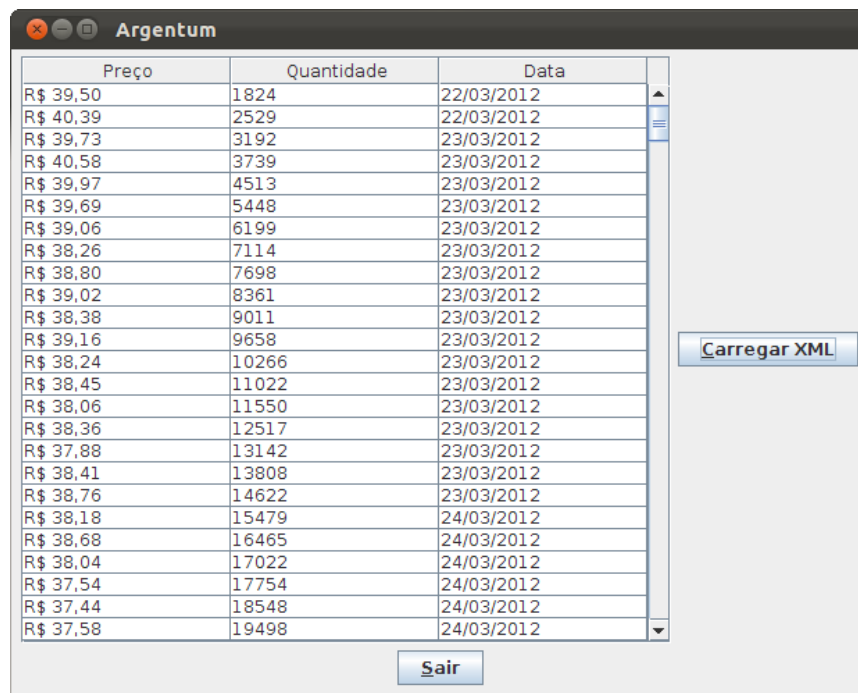


Figura 8.1: ArgentumUi com FlowLayout

Poderíamos usar um outro Layout Manager como o GridLayout, por exemplo. Fazer a mudança é simples. Adicione no método preparaPainelPrincipal:

```
painelPrincipal.setLayout(new GridLayout());
```

Mas repare como nossa aplicação fica totalmente diferente:

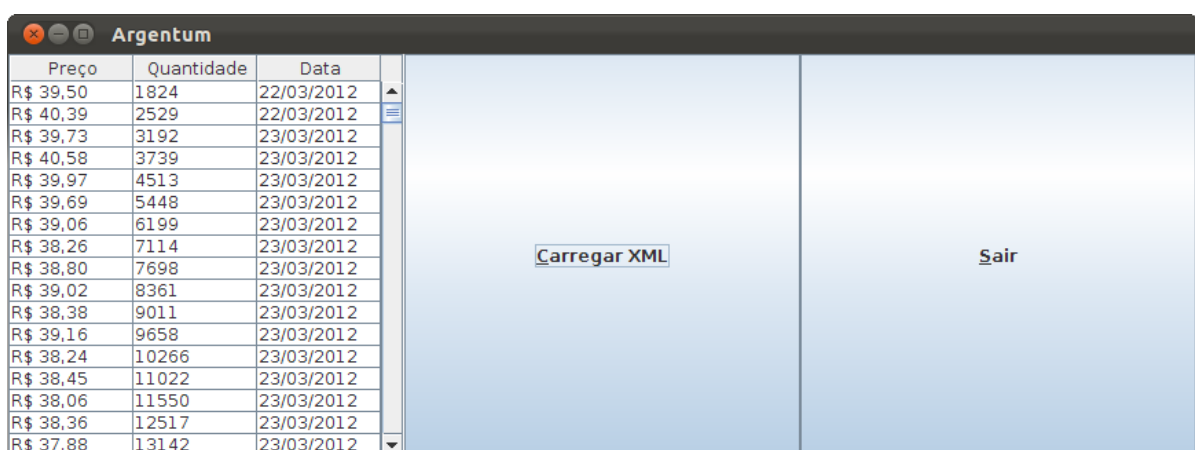


Figura 8.2: ArgentumUi com GridLayout

Agora os componentes tem tamanho igual (repare que o tamanho que colocamos para a tabela não é respeitado). Note como os elementos parecem estar dispostos em uma grade (um grid). Ao redimensionar essa

tela, por exemplo, os elementos não fluem como antes; eles são redimensionados para se adaptarem ao novo tamanho do grid.

Ou ainda, usando o `BoxLayout` pelo eixo y:

```
painelPrincipal.setLayout(  
    new BoxLayout(painelPrincipal, BoxLayout.Y_AXIS));
```



Preço	Quantidade	Data
R\$ 39,50	1824	22/03/2012
R\$ 40,39	2529	22/03/2012
R\$ 39,73	3192	23/03/2012
R\$ 40,58	3739	23/03/2012
R\$ 39,97	4513	23/03/2012
R\$ 39,69	5448	23/03/2012
R\$ 39,06	6199	23/03/2012
R\$ 38,26	7114	23/03/2012
R\$ 38,80	7698	23/03/2012
R\$ 39,02	8361	23/03/2012
R\$ 38,38	9011	23/03/2012
R\$ 39,16	9658	23/03/2012
R\$ 38,24	10266	23/03/2012
R\$ 38,45	11022	23/03/2012
R\$ 38,06	11550	23/03/2012
R\$ 38,36	12517	23/03/2012
R\$ 37,88	13142	23/03/2012
R\$ 38,41	13808	23/03/2012
R\$ 38,76	14622	23/03/2012

Carregar XML
Sair

Figura 8.3: ArgentumUi com `BoxLayout`

Agora os botões não são mais redimensionados e a tabela tem seu tamanho redimensionado junto com a janela. Os componentes são dispostos um abaixo do outro pelo eixo Y.

Há uma série de Layout Managers disponíveis no Java, cada um com seu comportamento específico. Há inclusive Layout Managers de terceiros (não-oficiais do Java) que você pode baixar. O projeto **JGoodies**, por exemplo, tem um excelente Layout Manager otimizado para trabalhar com formulários, o `FormLayout`:

<http://www.jgoodies.com/>

8.2 LAYOUT MANAGERS MAIS FAMOSOS

Vimos algumas aplicações de Layout Manager diferentes antes. Vamos ver brevemente as principais características dos layout managers mais famosos:

FlowLayout

É o mais simples e o padrão de todos os `JPanels`. Organiza os componentes um ao lado do outro em linha, da esquerda para a direita, usando o tamanho que você definiu ou, se não houver definição, seus tamanhos preferenciais. Quando a linha fica cheia, uma nova linha é criada.

BoxLayout

Organiza os componentes sequencialmente pelo eixo X ou eixo Y (indicamos isso no construtor) usando os tamanhos preferenciais de cada componente.

GridLayout

Organiza os componentes em um grid (tabela) com várias linhas e colunas (podemos definir no construtor). Os componentes são colocados um por célula e com tamanho que ocupe a célula toda.

GridBagLayout

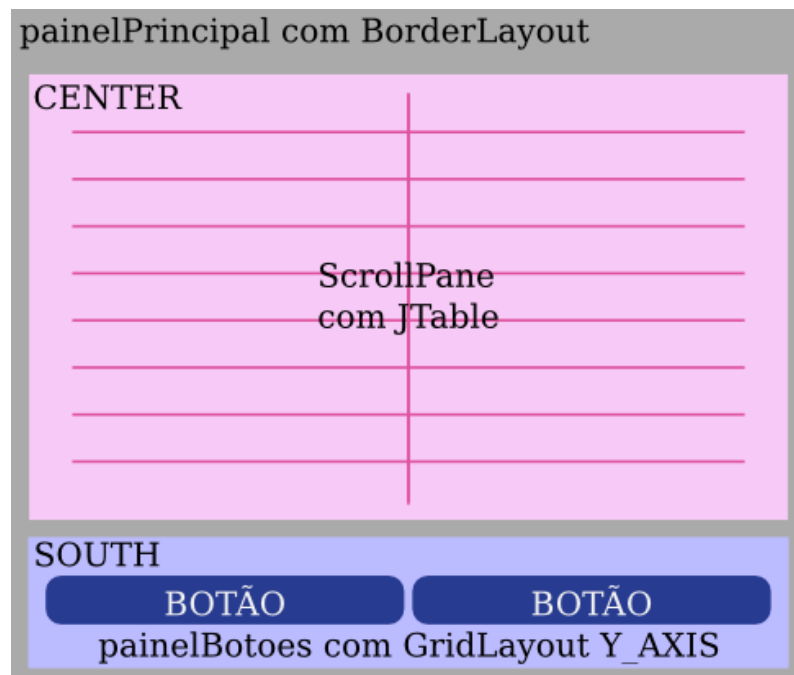
É o mais complexo layout e é baseado no GridLayout. A idéia também é representar a tela como um grid com linhas e colunas. Mas no GridBagLayout podemos posicionar elementos ocupando várias células em qualquer direção, o que permite layouts mais customizados, mas também causa um custo muito alto de manutenção. A definição de onde deve ser colocado cada componente é feita através de restrições (GridBagConstraints) passadas ao método add.

BorderLayout

Divide o container em cinco regiões: Norte, Sul, Leste, Oeste e Centro. Ao adicionar um componente, indicamos a região onde queremos adicioná-lo. Na hora de renderizar, o BorderLayout primeiro coloca os componentes do Norte e Sul em seus tamanhos preferenciais; depois, coloca os do Leste e Oeste também nos tamanhos preferenciais; por último, coloca o componente do Centro ocupando todo o restante do espaço.

8.3 EXERCÍCIOS: USANDO LAYOUT MANAGERS

1) Vamos organizar melhor nossos componentes usando alguns *layout managers* que vimos. Veja o esquema:



- 2) Na classe ArgentumUI, **altere** o método montaTela para chamar nosso novo método (cuidado com a ordem):

```
public void montaTela() {
    preparaJanela();
    preparaPainelPrincipal();
    preparaTabela();
    preparaPainelBotoes();    // adicione essa linha!
    preparaBotaoCarregar();
    preparaBotaoSair();
    mostraJanela();
}
```

A chamada ao preparaPainelBotoes vai ficar sublinhada em vermelho, indicando que o método ainda não existe. Na linha do erro, use o ctrl + 1 e escolha a opção de **criar o método** preparaPainelBotoes. Adicione as seguintes linhas de implementação:

```
private void preparaPainelBotoes() {
    painelBotoes = new JPanel(new GridLayout());
    painelPrincipal.add(painelBotoes);
}
```

Agora, quem vai indicar erro é o atributo painelBotoes. Basta usar o Ctrl + 1 e escolher a opção de colocar a variável num novo atributo (*Create field painelBotoes*). Ele vai inserir **automaticamente** na lista de atributos o código:

```
private JPanel painelBotoes;
```

Altere os métodos preparaBotaoCarregar e preparaBotaoSair que adicionam os botões para adicionar ambos ao novo painelBotoes. Por exemplo, troque:

```
painelPrincipal.add(botaoCarregar);
```

Por:

```
painelBotoes.add(botaoCarregar);
```

Rode a classe `ArgentumUI`. Repare que, ao redimensionar, os botões não mais “escorregam” separadamente - eles agora estão agrupados.

- 3) Próximo passo: usar o `BorderLayout` para posicionar a tabela e os botões corretamente.

Altere o método `preparaPainelPrincipal` adicionando a chamada ao layout:

```
private void preparaPainelPrincipal() {  
    painelPrincipal = new JPanel();  
    painelPrincipal.setLayout(new BorderLayout());  
    janela.add(painelPrincipal);  
}
```

Altere o método `preparaTabela` para indicar que queremos adicioná-la ao centro da tela. Basta **adicionar um parâmetro** na chamada ao método `add`:

```
painelPrincipal.add(scroll, BorderLayout.CENTER);
```

Altere o método `preparaPainelBotoes` para indicar que queremos adicioná-lo na região sul. Como o acima, basta apenas **adicionar um parâmetro** na chamada ao método `add`:

```
painelPrincipal.add(painelBotoes, BorderLayout.SOUTH);
```

Rode a aplicação novamente e veja a diferença em relação a nossa tela anterior.



The screenshot shows a Java Swing window titled "Argentum". Inside the window, there is a table titled "Lista de Negócios". The table has three columns: "Preço", "Quantidade", and "Data". It contains 20 rows of data. Below the table, there are two buttons: "Carregar XML" and "Sair".

Preço	Quantidade	Data
R\$ 39,50	1824	22/03/2012
R\$ 40,39	2529	22/03/2012
R\$ 39,73	3192	23/03/2012
R\$ 40,58	3739	23/03/2012
R\$ 39,97	4513	23/03/2012
R\$ 39,69	5448	23/03/2012
R\$ 39,06	6199	23/03/2012
R\$ 38,26	7114	23/03/2012
R\$ 38,80	7698	23/03/2012
R\$ 39,02	8361	23/03/2012
R\$ 38,38	9011	23/03/2012
R\$ 39,16	9658	23/03/2012
R\$ 38,24	10266	23/03/2012
R\$ 38,45	11022	23/03/2012
R\$ 38,06	11550	23/03/2012
R\$ 38,36	12517	23/03/2012
R\$ 37,88	13142	23/03/2012
R\$ 38,41	13808	23/03/2012
R\$ 38,76	14622	23/03/2012
R\$ 38,18	15479	24/03/2012

- 4) (opcional) Se você fez o título no exercício anterior, você deve ter notado que o título sumiu! Para trazê-lo de volta, acrescente a opção `BorderLayout.NORTH` similarmente ao que foi feito nos itens anteriores.

8.4 INTEGRANDO JFREECHART

No capítulo anterior, desvendamos o JFreeChart e criamos toda a infraestrutura necessária para criar gráficos complexos para nossa análise técnica. Agora, vamos integrar esses gráficos à nossa aplicação.

Lembre que, na classe `GeradorDeGrafico`, já criamos um método `getPanel` que devolve um componente pronto para ser exibido no Swing. O que vamos fazer é gerar o gráfico logo após a leitura do XML.

E, para exibir o gráfico em nossa interface, vamos organizar tudo com abas (*tabs*). Usando um `JTabbedPane` vamos organizar a tabela e o gráfico, cada um em uma aba diferente. Usar um `JTabbedPane` é muito fácil:

```
JTabbedPane abas = new JTabbedPane();
abas.add("Label 1", componenteDa1aAba);
abas.add("Label 2", componenteDa2aAba);
```

Para gerar o gráfico baseado na lista de negócios, usaremos toda a infraestrutura que fizemos até agora: obteremos a lista de negócios do XML e mandaremos a `CandlestickFactory` construir as *candles* para nós. Então, criaremos uma `SerieTemporal` com elas e a passaremos para o `GeradorDeGrafico`. Depois de plotar os indicadores, conseguiremos o painel com gráfico para adicionar à aba!

O código disso tudo ficará parecido com esse:

```
1 List<Negocio> negocios = ...
2
3 CandlestickFactory fabrica = new CandlestickFactory();
4 List<Candle> candles = fabrica.constroiCandles(negocios);
5
6 SerieTemporal serie = new SerieTemporal(candles);
7
8 GeradorDeGrafico gg =
9     new GeradorDeGrafico(serie, 2, serie.getTotal() - 1);
10 gg.plotaIndicador(new MediaMovelSimples());
11 gg.plotaIndicador(new IndicadorFechamento());
12 JPanel grafico = gg.getPanel();
```

8.5 EXERCÍCIOS: COMPLETANDO A TELA DA NOSSA APLICAÇÃO

- 1) Na classe `ArgentumUI`, **adicione** a chamada ao `preparaAbas` no método `montaTela` (cuidado com a ordem):

```
public void montaTela() {  
    preparaJanela();  
    preparaPainelPrincipal();  
    preparaAbas();  
    preparaTabela();  
    preparaPainelBotoes();  
    preparaBotaoCarregar();  
    preparaBotaoSair();  
    mostraJanela();  
}
```

Note que **a linha ficará vermelha**. Novamente, use o `ctrl + 1` para criar o método automaticamente.

2) **Adicione** a implementação ao método `preparaAbas()` recém-criado.

```
private void preparaAbas() {  
    abas = new JTabbedPane();  
    abas.addTab("Tabela", null);  
    abas.addTab("Gráfico", null);  
    painelPrincipal.add(abas);  
}
```

Altere o método `preparaTabela` para colocar a tabela com scroll na primeira aba do painel abas. Comente (ou remova) a linha que adiciona o scroll ao `painelPrincipal` e adicione a chamada ao `abas.setComponentAt` como mostrado abaixo:

```
private void preparaTabela() {  
    tabela = new JTable();  
    JScrollPane scroll = new JScrollPane();  
    scroll.setViewportView(tabela);  
    // painelPrincipal.add(scroll, BorderLayout.CENTER);  
    abas.setComponentAt(0, scroll);  
}
```

Rode novamente e observe a montagem das abas.



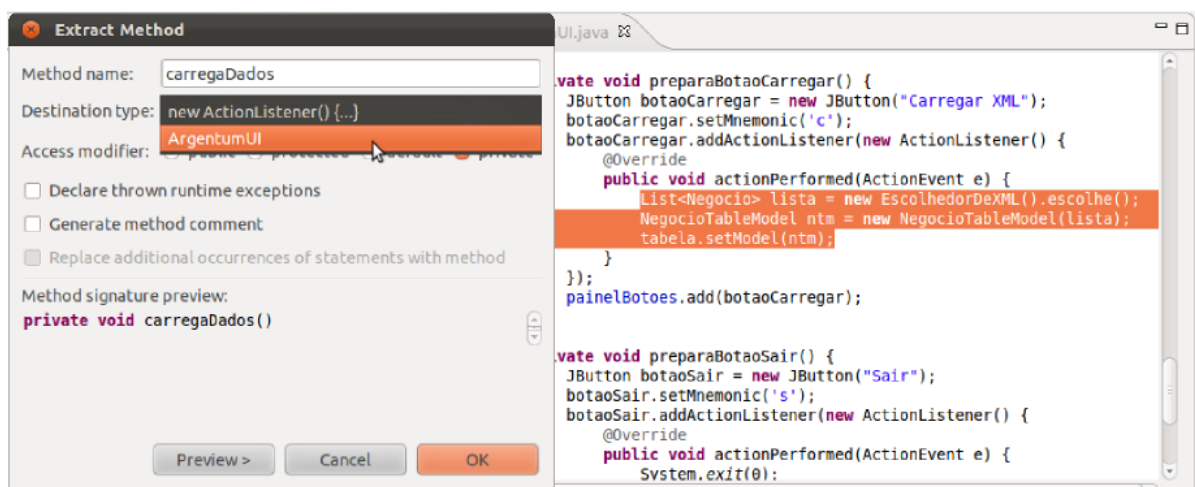
The screenshot shows a Java Swing window titled "Argentum" with a subtitle "Lista de Negócios". It has two tabs: "Tabela" (selected) and "Gráfico". The table has three columns: "Preço", "Quantidade", and "Data". It contains 15 rows of data. At the bottom, there are two buttons: "Carregar XML" and "Sair".

Preço	Quantidade	Data
R\$ 39,50	1824	22/03/2012
R\$ 40,39	2529	22/03/2012
R\$ 39,73	3192	23/03/2012
R\$ 40,58	3739	23/03/2012
R\$ 39,97	4513	23/03/2012
R\$ 39,69	5448	23/03/2012
R\$ 39,06	6199	23/03/2012
R\$ 38,26	7114	23/03/2012
R\$ 38,80	7698	23/03/2012
R\$ 39,02	8361	23/03/2012
R\$ 38,38	9011	23/03/2012
R\$ 39,16	9658	23/03/2012
R\$ 38,24	10266	23/03/2012
R\$ 38,45	11022	23/03/2012
R\$ 38,06	11550	23/03/2012
R\$ 38,36	12517	23/03/2012
R\$ 37,88	13142	23/03/2012
R\$ 38,41	13808	23/03/2012

- 3) Precisamos alterar a classe anônima que trata o evento do botão de carregar o XML na classe ArgentumUI para gerar o gráfico através das classes que criamos. Mas se colocarmos todo esse código dentro da classe anônima, com certeza perderemos legibilidade.

Vamos então criar um **método auxiliar** chamado `carregaDados` na classe `ArgentumUI`, que será responsável por tratar o evento de carregar tanto a **tabela** quanto o **gráfico**.

Começamos isolando as linhas que já existem no método `actionPerformed` do `botaoCarregar` em um método da classe `ArgentumUI`. Para isso, selecione as linhas de dentro desse método e faça **alt + shift + M**. Na janelinha que abrir, dê o nome para esse método (`carregaDados`) e **não esqueça** de mudar o *Destination type* para que ele seja declarado na própria `ArgentumUI`.



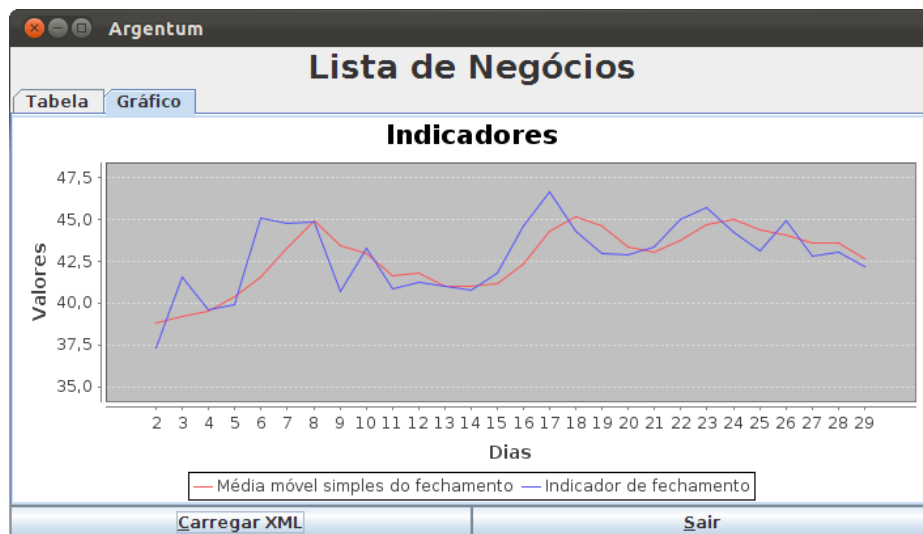
- 4) Agora, só falta preencher o método `carregaDados` criado com a construção das candles, do gerador de

gráfico e adicionar o painel à segunda aba! **Complete** o método como indicado abaixo (note que você já criou o método e já tem a implementação até a linha 4:

```

1 private void carregaDados() {
2     List<Negocio> lista = new EscolhedorDeXML().escolhe();
3     NegocioTableModel ntm = new NegocioTableModel(lista);
4     tabela.setModel(ntm);
5
6     CandlestickFactory fabrica = new CandlestickFactory();
7     List<Candle> candles = fabrica.constroiCandles(lista);
8     SerieTemporal serie = new SerieTemporal(candles);
9
10    GeradorDeGrafico gerador = new GeradorDeGrafico(serie, 2, serie.getTotal() - 1);
11    gerador.plotaIndicador(new MediaMovelSimples());
12    gerador.plotaIndicador(new IndicadorFechamento());
13
14    abas.setComponentAt(1, gerador.getPanel());
15 }
  
```

5) Rode novamente e teste o gráfico e as abas!



8.6 INDICADORES MAIS ELABORADOS E O DESIGN PATTERN DECORATOR

Vimos no capítulo anterior 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 `getVolume`.

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 - 2; i <= posicao; i++) {  
        soma += outroIndicador.calcula(i, serie);  
    }  
}
```

```
}  
    return soma / 3;  
}
```

Nossa classe `MediaMoveISimples` 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 `MediaMoveISimples` **é um** Indicador e também **tem um** outro Indicador. Já vimos antes outro tipo que se comporta da mesma forma, você se lembra?

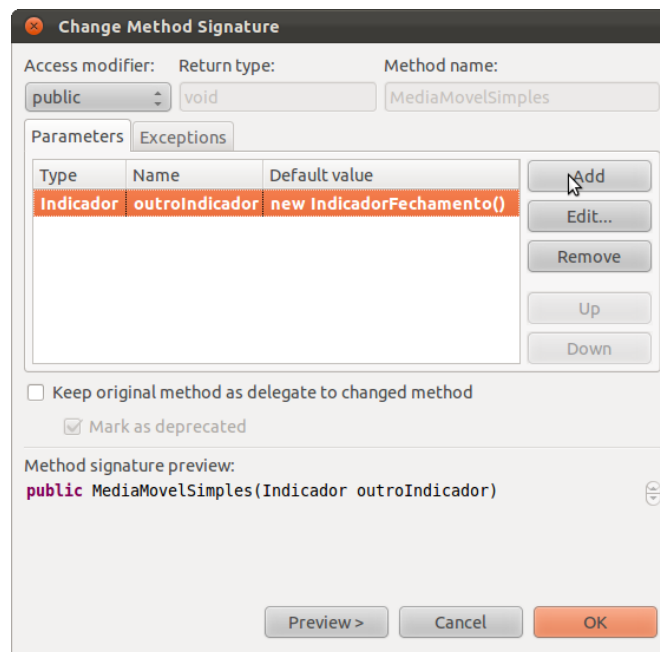
Assim como os componentes do Swing, nossa `MediaMoveISimples` se tornou **também** um exemplo de *Composite*.

8.7 EXERCÍCIOS: INDICADORES MAIS ESPERTOS E O DESIGN PATTERN DECORATOR

- 1) Faremos uma grande mudança agora: nossas médias móveis devem receber como argumento um outro indicador, formando o *design pattern* Decorator, como visto no texto.

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

- a) Comece a escrever o nome da classe e mande autocompletar: `Med<ctrl + espaço>;`
 - b) Use o criador automatico de construtores: `ctrl + 3 Constructor`.
- 2) Modifique o método 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 - 2; i <= posicao; 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 `MediaMoveISimples`, 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) Agora, para que o `JFreeChart` consiga plotar uma linha do gráfico para cada uma das médias, ainda é preciso que o `toString` da `MediaMoveISimples` seja, também, decorado, isto é, é preciso modificá-lo para:

```
public String toString() {
    return "Média móvel simples do " + outroIndicador;
}
```

- 6) (opcional) Se você fez a `MediaMoveIPonderada`, modifique também essa outra classe 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 `MediaMovelSimple`.

Não esqueça de alterar também o `toString` da `MediaMovelPonderada`.

- 7) (opcional) Crie uma outra classe `IndicadorAbertura` e faça um teste unitário na classe `MediaMovelSimpleTest` que use esse novo indicador em vez do de fechamento.

Faça também para máximo, mínimo ou outros que desejar.

8.8 DISCUSSÃO EM SALA DE AULA: USO DE IDEs PARA MONTAR A TELA

CAPÍTULO 9

Reflection e Annotations

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 métodos ela tem, qual chamada de método está sendo feita e assim por diante.

Mas existem algumas raras situações onde essa garantia toda 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!

Nesse capítulo você vai conhecer um pouco desse recurso avançado e muito poderoso, mas que deve ser usado de forma sensata!

9.1 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:

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

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

O **javax.reflection** é um pacote do Java que permite criar chamadas em tempo de execução, sem precisar conhecer as classes e objetos envolvidos quando escrevemos nosso código (tempo de compilação). Esse dinamismo é necessário para resolvermos determinadas tarefas que nosso programa só descobre serem necessárias ao receber dados, em tempo de execução.

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

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 métodos dinamicamente baseado no nome do método como String;
- Descobrir se determinados pedaços do código têm annotations.

9.2 CLASS, FIELD E METHOD

O ponto de partida de 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:

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

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

A saída será:

```
preco
quantidade
data
```

Fazendo similarmente para métodos é possível conseguir a lista:

```
getPreco
getQuantidade
getData
getVolume
isMesmoDia
```

É possível fazer muito mais. Investigue a API de reflection usando **ctrl + espaço** no Eclipse e pelo JavaDoc.

9.3 USANDO ANOTAÇÕES

Anotações (annotations) são uma novidade do Java 5 cujo objetivo é possibilitar a declaração de **metadados** nos nossos objetos, isto é, configurações de uma classe podem ficar dentro dela, em vez de em um XML à parte!

Até agora, usamos anotações para indicar que um método não deve mais ser usado (`@Deprecated`), que ele foi sobrescrito (`@Override`) e para configurar um método como teste do JUnit (`@Test`) e talvez já a tenhamos visto em outros lugares.

Em todas essas ocasiões, percebemos que a presença da anotação não influi no comportamento daquela classe, daqueles objetos. Não são códigos executáveis, que mudam o que é executado ou não. São *metadados*: informações (dados) que falam sobre nossa classe mas não fazem parte da classe em si.

Metadados são muito usados para configurações de funcionalidades anexas àquela classe. Por exemplo, usamos a anotação do `Test` para configurar que o JUnit entenda aquele método como um teste e rode ele quando usarmos o `alt + shift + X T`. Mas, se retirarmos a anotação, a nossa classe continua compilando normalmente - apenas, quando rodarmos o JUnit, esse método será ignorado.

Em geral, portanto, usamos anotações para criar configurações nos nossos artefatos com objetivo de que depois essas anotações sejam lidas e processadas por alguém interessado naquelas informações.

9.4 USAR JTABLES É DIFÍCIL

No capítulo de Swing, vimos que usar `JTable` é uma tarefa trabalhosa. Precisamos definir as características de renderização do componente e o modelo de dados que vai ser exibido (nosso `TableModel`).

Criamos anteriormente uma classe chamada `NegocioTableModel` para devolver os dados dos negócios que queremos exibir na tabela. Mas imagine se nosso cliente decidisse que precisa ver também as tabelas de listagem de candles, um para cada dia. Mais ainda, ele um dia pode querer mostrar uma tabela com as Séries Temporais do sistema todo! Em ambos os casos, provavelmente criaríamos um `TableModel` muito parecido com o de negócios, apenas substituindo as chamadas aos getters de uma classe pelos getters de outra.

Usando um pouco de *reflection*, será que conseguiríamos criar um único `TableModel` capaz de lidar com qualquer dos nossos modelos? Um `TableModel` que, dependendo da classe (`Negocio` / `Candle`) passada, consegue chamar os getters apropriados? Poderíamos colocar um monte de `if's` que avaliam o tipo do objeto e escolhem o getter a chamar, mas com um pouco mais de *reflection* teremos bem menos trabalho.

Vejamos o que será preciso para implementar um `ArgentumTableModel` que consiga lidar com todas as classes de modelo do `Argentum`. Lembre-se que para termos um `TableModel` é preciso que implementemos, no mínimo, os três métodos: `getRowCount`, `getColumnCount` e `getValueAt`.

O método `getRowCount` meramente devolve a quantidade de itens na lista que será mostrada na tabela, portanto, a sua implementação é igual. Agora, para sabermos o número de colunas da tabela é preciso que olhemos a classe em questão e contemos a quantidade de, por exemplo, atributos dela!

A partir do tipo dos elementos da lista que estamos manipulando, conseguimos pegar a `Class` desses elementos e iterar nos seus atributos (`Field`) para pegar seus valores ou, simplesmente o que precisamos: o número de atributos. Dessa forma, o método ficaria assim:

```
@Override
public int getColumnCount() {
    Class<?> classe = lista.get(0).getClass();
    Field[] atributos = classe.getDeclaredFields();
    return atributos.length;
}
```

9.5 USANDO BEM ANOTAÇÕES

Note, contudo, haverá muitas vezes em que não gostaríamos de mostrar todos os atributos na tabela ou ainda, haverá vezes em que precisaremos mostrar informações compostas em uma mesma célula da tabela.

Nesses casos, podemos criar um método que traz tal informação e, em vez de contar atributos, contaremos esses métodos. Apenas, tanto no caso de contarmos atributos quanto no caso de contarmos métodos, não queremos contar **todos** eles, mas apenas os que representarão colunas na nossa tabela.

Para **configurar** quais métodos trarão as colunas da tabela, podemos usar um recurso do Java 5 que você já viu diversas vezes durante seu aprendizado: **anotações**. Por exemplo, podemos marcar os métodos de `Negocio` que representam colunas assim:

```
public final class Negocio {

    // atributos e construtor

    @Coluna
    public double getPreco() {
        return preco;
    }

    @Coluna
    public int getQuantidade() {
        return quantidade;
    }

    @Coluna
    public Calendar getData() {
        return (Calendar) data.clone();
    }

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

Dessa forma, a contagem deverá levar em consideração **apenas** os métodos que estão anotados com `@Coluna`, isto é, temos que passar por cada método declarado nessa classe contando os que tiverem a anotação `@Coluna`. Modificando nosso `getColumnCount` temos:

```
@Override
public int getColumnCount() {
    Object objeto = lista.get(0);
    Class<?> classe = objeto.getClass();
```

```
int colunas = 0;
for (Method metodo : classe.getDeclaredMethods()) {
    if (metodo.isAnnotationPresent(Coluna.class)) {
        colunas++;
    }
}
return colunas;
}
```

Um outro problema ao montar a tabela dinamicamente aparece quando tentamos implementar o método `getValueAt`: precisamos saber a ordem em que queremos exibir as colunas. Agora, quando usamos *reflection* não sabemos exatamente a ordem em que os métodos são percorridos.

Como a posição das colunas em uma tabela é importante, precisamos adicionar essa configuração às colunas:

```
public final class Negocio {

    // atributos e construtores

    @Coluna(posicao=0)
    public double getPreco() {
        return preco;
    }

    @Coluna(posicao=1)
    public int getQuantidade() {
        return quantidade;
    }

    @Coluna(posicao=2)
    public Calendar getData() {
        return (Calendar) data.clone();
    }
}
```

Então, basta percorrer os atributos dessa classe, olhar para o valor dessas anotações e montar a tabela dinamicamente com essas posições - isto é, substituiremos aquele `switch` que escrevemos no `NegocioTableModel` por algumas linhas de *reflection*:

```
@Override
public Object getValueAt(int linha, int coluna) {
    try {
        Object objeto = lista.get(linha);
        Class<?> classe = objeto.getClass();
    }
}
```

```
    for (Method metodo : classe.getDeclaredMethods()) {
        if (metodo.isAnnotationPresent(Coluna.class)) {
            Coluna anotacao = metodo.getAnnotation(Coluna.class);
            if (anotacao.posicao() == coluna) {
                return metodo.invoke(objeto);
            }
        }
    }
    return "";
} catch (Exception e) {
    return "Erro";
}
```

9.6 CRIANDO SUA PRÓPRIA ANOTAÇÃO

Para que nosso código compile, no entanto, precisamos descobrir como escrever uma anotação em Java! As anotações são tipos especiais declarados com o termo `@interface`, já que elas surgiram apenas no Java 5 e não quiseram criar uma nova palavra chave para elas.

```
public @interface Coluna {

}
```

Os parâmetros que desejamos passar à anotação, como `posicao` são declarados como métodos que servem tanto para setar o valor quanto para devolver. A sintaxe é meio estranha no começo:

```
public @interface Coluna {
    int posicao();
}
```

Nesse momento, já temos a anotação e um parâmetro obrigatório `posicao`, mas para que ela de fato faça seu trabalho, ainda é preciso informar duas configurações importantes para toda anotação: que tipo de estrutura ela configura (método, atributo, classe, ...) e também se ela serve apenas para ajudar o desenvolvedor enquanto ele está programando, tempo de compilação, ou se ela precisará ser lida inclusive durante a execução - por exemplo, a anotação `@Override` só precisa estar presente durante a compilação, enquanto a `@Test` precisa ser lida na hora de executar o teste em si.

Nossa *annotation* ficará como abaixo:

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Coluna {
```

```
    int posicao();  
}
```

Com a anotação compilando, a classe `Negocio` devidamente configurada e o `ArgentumTableModel` estamos preparados para mostrar tabelas de qualquer modelo que tenha seus *getters* anotados com `@Coluna(posicao=...)`!

9.7 EXERCÍCIOS: ARGENTUMTABLEMODEL

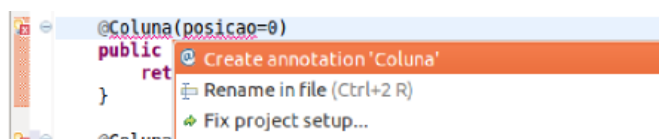
Atenção: os imports necessários para esse exercício devem ser `java.lang.annotation`

- 1) Anote os getters da classe `Negocio` com `@Coluna`, já passando o atributo `posicao` e sem se preocupar com os erros de compilação ainda.

Adicione em cada método **apenas** a anotação, passando posições diferentes:

```
public final class Negocio {  
  
    // atributos e construtores  
  
    @Coluna(posicao=0)  
    public double getPreco() {  
        return preco;  
    }  
  
    @Coluna(posicao=1)  
    public int getQuantidade() {  
        return quantidade;  
    }  
  
    @Coluna(posicao=2)  
    public Calendar getData() {  
        return (Calendar) data.clone();  
    }  
}
```

- 2) Usando o **ctrl + 1** mande o Eclipse *create new annotation* para você e lembre-se de **alterar** o pacote dela para `br.com.caelum.argentum.ui`, já que essa anotação só existe para que a `JTable` funcione com qualquer modelo nosso.



Voltando na classe `Negocio` ela ainda não compila porque o atributo `posicao` ainda não existe na anotação nova! Use o **ctrl + i** novamente para criar o atributo. **Confira!** Sua anotação deve estar assim:

```
public @interface Coluna {  
  
    int posicao();  
  
}
```

- 3) Para que essa anotação seja guardada para ser lida em tempo de execução é preciso configurá-la para tal. Além disso, como queremos que o desenvolvedor anote métodos (e não classes ou atributos, por exemplo) também é preciso fazer essa configuração.

Adicione as anotações à sua annotation:

```
@Target(ElementType.METHOD)  
@Retention(RetentionPolicy.RUNTIME)  
public @interface Coluna {  
  
    int posicao();  
  
}
```

- 4) Crie a classe `ArgentumTableModel` no pacote `br.com.caelum.argentum.ui`, e utilize o código aprendido durante o capítulo:

```
1 public class ArgentumTableModel extends AbstractTableModel {  
2  
3     private final List<?> lista;  
4     private Class<?> classe;  
5  
6     public ArgentumTableModel(List<?> lista) {  
7         this.lista = lista;  
8         this.classe = lista.get(0).getClass();  
9     }  
10  
11     @Override  
12     public int getRowCount() {  
13         return lista.size();  
14     }  
15  
16     @Override  
17     public int getColumnCount() {  
18         int colunas = 0;  
19         for (Method metodo : classe.getDeclaredMethods()) {  
20             if (metodo.isAnnotationPresent(Coluna.class))  
21                 colunas++;  
22     }  
23 }
```

```
22     }
23     return colunas;
24 }
25
26 @Override
27 public Object getValueAt(int linha, int coluna) {
28     try {
29         Object objeto = lista.get(linha);
30         for (Method metodo : classe.getDeclaredMethods()) {
31             if (metodo.isAnnotationPresent(Coluna.class)) {
32                 Coluna anotacao = metodo.getAnnotation(Coluna.class);
33                 if (anotacao.posicao() == coluna)
34                     return metodo.invoke(objeto);
35             }
36         }
37     } catch (Exception e) {
38         return "Erro";
39     }
40     return "";
41 }
42 }
```

- 5) **Altere** o método `carregaDados` da classe `ArgentumUI` para usar o nosso novo `TableModel`. Onde tínhamos:

```
NegociosTableModel model = new NegociosTableModel(negocios);
tabela.setModel(model);
```

Substitua por:

```
ArgentumTableModel model = new ArgentumTableModel(negocios);
tabela.setModel(model);
```

Rode novamente e deveremos ter a tabela montada dinamicamente.

(Note que perdemos os formatos. Vamos adicioná-los em seguida.)

- 6) Alguns frameworks usam bastante reflection para facilitar a criação de suas telas. Pesquise a respeito.

9.8 EXERCÍCIOS OPCIONAIS: NOMES DAS COLUNAS

- 1) Queremos possibilitar também a customização dos títulos das colunas. Para isso, vamos **alterar** as chamadas na classe `Negocio` para usar o novo parâmetro:

```
@Coluna(nome="Preço", posicao=0)
// ...
```



```
@Coluna(nome="Quantidade", posicao=1)
// ...
```

```
@Coluna(nome="Data", posicao=2)
// ...
```

- 2) Para fazer esse código compilar, basta usar novamente o **ctrl + 1** e pedir que Eclipse crie esse atributo para você! Ele mesmo vai adicionar tal parâmetro à anotação:

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Coluna {

    String nome();
    int posicao();
}
```

- 3) Na classe `ArgentumTableModel`, **adicione** o método `getColumnName`. Ele é muito parecido com os códigos que escrevemos antes:

```
1 @Override
2 public String getColumnName(int coluna) {
3     for (Method metodo : classe.getDeclaredMethods()) {
4         if (metodo.isAnnotationPresent(Coluna.class)) {
5             Coluna anotacao = metodo.getAnnotation(Coluna.class);
6             if (anotacao.posicao() == coluna)
7                 return anotacao.nome();
8         }
9     }
10    return "";
11 }
```

Rode novamente e observe os nomes das colunas na tabela.

9.9 PARA SABER MAIS: FORMATTER, PRINTF E STRING.FORMAT

Nossa tabela, apesar de bem mais flexível, perdeu todas as formatações que tínhamos feito antes. Note que a data está de volta ao `toString` padrão do `Calendar` e, se você havia feito o opcional de formatação dos valores, ela também se perdeu.

A partir do Java 5 a API de `Formatter` nos ajuda em casos assim. Ela provê uma forma bastante robusta de se trabalhar formatação de dados para impressão e é fortemente baseada nas ideias do `printf` do C.

Basicamente, fazemos uma `String` de formato que descreve em uma sintaxe especial o formato de saída dos dados que passamos como argumento depois.

É possível formatar números, definindo por exemplo a quantidade de casas decimais, ou formatar a saída de um Calendar usando seus campos, etc. Para imprimirmos diretamente no console, podemos usar essa sintaxe com o método `System.out.printf`, que também surgiu no Java 5. Alguns usos:

```
// usar printf é mais facil que concatenar Strings
String nome = "Manoel";
String sobrenome = "Silva";
System.out.printf("Meu nome é %s e meu sobrenome é %s\n", nome, sobrenome);
// saída: Meu nome é Manoel e meu sobrenome é Silva

// formatando casas decimais
System.out.printf("PI com 4 casas decimais: %.4f\n", Math.PI);
//saída: PI com 4 casas decimais: 3.1416

// a data de hoje em dia/mes/ano
System.out.printf("Hoje é %1$tD/%1$tM/%1$tY", Calendar.getInstance());
// saída: Hoje é 04/05/2012
```

Caso precisemos apenas da String formatada em vez de imprimí-la, podemos usar:

```
String msg = String.format("PI com 4 casas decimais: %.4f\n", Math.PI);
```

9.10 PARA SABER MAIS: PARÂMETROS OPCIONAIS

Da forma como criamos atributos na anotação, eles são obrigatórios: isto é, dá erro de compilação quando anotamos um método com `@Coluna` mas não preenchemos sua posição (e seu nome, se você tiver feito o exercício opcional).

Contudo, há diversas situações em que passar uma informação para a anotação é opcional. Um exemplo disso é o atributo `expected` da anotação `@Test`: só é preciso passá-lo se esperamos receber uma exceção.

No caso da nossa formatação, não precisaremos fazer nada se recebermos uma String como parâmetro, mas precisaremos modificar a formatação se recebermos datas ou valores, por exemplo.

É justo fazer um paralelo entre atributos de uma anotação e getters, na hora de recuperar o valor. Assim, é fácil notar que, para funcionar sempre, a anotação **precisa** que você tenha um valor para devolver. Assim, se queremos criar um atributo opcional é preciso ter um valor padrão para ser devolvido se não passarmos nada nesse parâmetro.

Para indicar esse valor padrão a ser adotado se não passarmos nada como, no nosso exemplo, máscara de formatação usamos a palavra chave `default`:

```
//...
public @interface Coluna {
```

```
//...  
String formato() default "%s";  
}
```

9.11 EXERCÍCIOS OPCIONAIS: FORMATAÇÕES NA TABELA

As datas e valores monetários não estão sendo exibidos corretamente. Para solucionar isso, vamos usar formataadores baseados na classe `Formatter` e o método `String.format`.

- 1) **Adicione** na anotação `Coluna` mais um parâmetro. Será a `String` com o formato desejado.

```
String formato() default "%s";
```

Repare no uso do **default** para indicar que, se alguém não passar o formato, temos um formato padrão. E esse formato é o mais simples (%s) possível: ele simplesmente imprime a `String`.

- 2) **Altere** o método `getValueAt` na classe `ArgentumTableModel` para usar o formato de cada coluna na hora de devolver o objeto. Felizmente, é muito simples fazer essa modificação! **Altere apenas a linha do retorno:**

```
@Override  
public Object getValueAt(int linha, int coluna) {  
    // outras linhas...  
    if (anotacao.posicao() == coluna)  
        return String.format(anotacao.formato(),  
                               metodo.invoke(objeto));  
    // mais linhas...  
}
```

- 3) **Altere** a classe `Negocio` passando agora nas anotações o formato desejado. Para a coluna de preço, usaremos:

```
@Coluna(nome="Preço", posicao=0, formato="R$ %,#.2f")
```

E, para a coluna data, usamos:

```
@Coluna(nome="Data", posicao=2, formato="%1$td/%1$tm/%1$tY")
```

Rode novamente e observe a formatação nas células.

- 4) Faça testes de unidade para o nosso `ArgentumTableModel`.
- 5) Faça todos os passos necessários para adicionar uma quarta coluna na tabela de negócios mostrando o **volume** do `Negocio`.

9.12 DISCUSSÃO EM SALA DE AULA: QUANDO USAR REFLECTION, ANOTAÇÕES E INTERFACES

Apêndice: O processo de Build: Ant e Maven

10.1 O PROCESSO DE BUILD

Até agora, utilizamos o Eclipse para compilar nossas classes, rodar nossa UI, verificar nossos testes. Também usáramos ele para fazer um JAR e outras tarefas.

Algumas dessas tarefas podem requerer uma sequência grande de passos, que manualmente podem dar muito trabalho. Mais ainda, elas podem variar de IDE para IDE.

Existem milhares de ferramentas de build, sendo a mais conhecida o make, muito usado para construir e executar tarefas de projetos escritos em C e C++, mas que para Java possui muitas desvantagens e não é utilizado.

As ferramentas mais conhecidas do Java possuem suporte das IDEs, e são facilmente executadas em qualquer sistema operacional.

10.2 O ANT

O ant é uma das ferramentas de build mais famosas do Java, e uma das mais antigas. <http://ant.apache.org/>

A idéia dele é definir uma série de tarefas, que são invocadas através de tags XML. Um target é composto por uma sequência de tarefas que devem ser executadas. Por exemplo, para compilar precisamos deletar os .class antigos, e depois compilar todos os arquivos do diretório X, Y e Z e jogar dentro de tais diretórios.

Como é uma das ferramentas mais populares, as grandes IDEs, como Eclipse e Netbeans, já vem com suporte ao Ant.

No ant, basicamente usamos o XML como uma linguagem de programação: em vez de possuir apenas dados estruturados, o XML possui também comandos. Muitas pessoas criticam o uso do XML nesse sentido.

10.3 EXERCÍCIOS COM ANT

- 1) Crie um arquivo, chamado `build.xml` no diretório raiz do nosso projeto (clique da direita no projeto, new/file).

Abra o arquivo e dê um `ctrl+espaço`, e escolha *buildfile template*.



Um `build.xml` de rascunho será gerado para você.

- 2) A partir daí vamos gerar um novo **target** chamado *compilar*

```
<project name="Argentum" default="compilar">

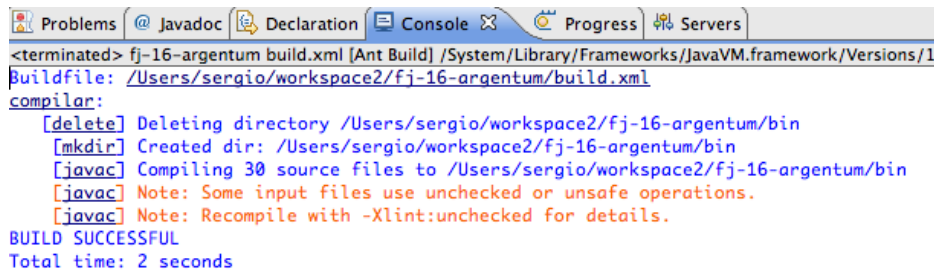
    <target name="compilar">
        <delete dir="bin" />
        <mkdir dir="bin" />

        <javac srcdir="src/main/java" destdir="bin" >
            <classpath>
                <fileset dir="lib">
                    <include name="*.jar" />
                </fileset>
            </classpath>
        </javac>
    </target>

</project>
```

Pode parecer um trecho de XML grande, mas lembre-se que você só vai escrevê-lo uma vez. Aproveite o `ctrl+espaço` para que esse trabalho seja facilitado.

- 3) Agora, clique com a direita no seu `build.xml` e escolha *Run as.../Ant Build*. Ele vai rodar o seu target default.



```
<terminated> fj-16-argentum build.xml [Ant Build] /System/Library/Frameworks/JavaVM.framework/Versions/1
Buildfile: /Users/sergio/workspace2/fj-16-argentum/build.xml
compilar:
[delete] Deleting directory /Users/sergio/workspace2/fj-16-argentum/bin
[mkdir] Created dir: /Users/sergio/workspace2/fj-16-argentum/bin
[javac] Compiling 30 source files to /Users/sergio/workspace2/fj-16-argentum/bin
[javac] Note: Some input files use unchecked or unsafe operations.
[javac] Note: Recompile with -Xlint:unchecked for details.
BUILD SUCCESSFUL
Total time: 2 seconds
```

- 4) Vamos criar um target para gerar um **jar**, que se chama empacotar. A sintaxe é bem simples:

```
<target name="empacotar" depends="compilar">
    <mkdir dir="jar" />
    <jar destfile="jar/argentum.jar" basedir="bin"/>
</target>
```

Repare que ele depende do target de compilar, logo ele vai compilar todo o projeto antes de empacotar. Aproveite e troque o target default do seu projeto para ser empacotar em vez de compilar.

Experimente executar novamente o seu arquivo de build. De um refresh no seu projeto, e veja o jar gerado!

- 5) Por último, vamos criar o target que executa. Ele depende de empacotar antes:

```
<target name="executar" depends="empacotar">
    <java classname="br.com.caelum.argentum.ui.ArgentumUI" fork="true">
        <classpath>
            <filelist files="jar/argentum.jar"/>
            <fileset dir="lib">
                <include name="*.jar" />
            </fileset>
        </classpath>
    </java>
</target>
```

Rode o seu build, lembrando de ter seus jars dentro da pasta lib, e escolhendo o target executar!

- 6) (opcional) Você pode melhorar seu arquivo de build, adicionando um target separado para limpar o diretório de classes compiladas, e ainda criar uma variável para definir o classpath e não ter de definir aquele *fileset* duas vezes.
- 7) (opcional) Existe uma view no Eclipse, chamada Ant View, que facilita o uso do Ant para executar diversos targets. Use ela, arrastando o seu build.xml até lá.

10.4 O MAVEN

O **maven** é outra ferramenta de build do grupo apache, nascido depois do Ant. Você pode baixa-lo e instalá-lo através do site: <http://maven.apache.org>

Diferentemente do ant, a idéia do maven é que você não precise usar tarefas e criar targets. Já existem um monte de tarefas pré-definidas, chamadas **goals**, que são agrupadas por plugins. Por exemplo, o plugin `compiler` possui goals como `compile`, que compila o código fonte, e o goal `testCompile`, que compila nossos unit tests.

Você pode invocar goal por goal, ou então usar os **phases**. Uma fase é composta por uma série de goals pré definidos, muito comumente usados. Por exemplo, a fase de empacotar é composta por compilar, testar e depois fazer o jar.

O interessante é que você não precisa escrever nada disso. Basta **declararmos** o que o nosso projeto necessita, que o maven se encarrega do resto.

10.5 O PROJECT OBJECT MODEL

A unidade básica do maven é um arquivo XML, onde você declara todas as informações do seu projeto. Baseado nisso o Maven vai poder compilar, testar, empacotar, etc.

Diferentemente do ant, não usamos o XML aqui como linguagem de programação, e sim como um modelo de dados hierárquico.

Esse arquivo é conhecido como o **POM**: Project Object Model. A menor versão possível dele é como a que segue:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>br.com.caelum</groupId>
  <artifactId>argentum</artifactId>
  <version>1.0</version>
</project>
```

Aqui estamos indicando que queremos usar a versão 4 do formato dos POMs do maven, que o grupo que representa a nossa “empresa” é `br.com.caelum` e que o nosso artefato chama-se `argentum`, de versão 1.0. Artefato é qualquer tipo de entregável, como um JAR.

No maven, podemos declarar que, para aquele nosso projeto, temos a necessidade de um outro artefato. Fazemos isso através da tag de dependências:

```
<dependencies>
  <dependency>
    <groupId>com.thoughtworks.xstream</groupId>
    <artifactId>xstream</artifactId>
    <version>1.3</version>
  </dependency>
</dependencies>
```


Agora, quando formos pedir para o maven fazer algo com o nosso projeto, antes de tudo ele vai baixar, a partir do repositório central de artefatos, o POM do XStream, e aí baixar também o jar do XStream e de suas dependências, recursivamente. Esse é um truque muito útil do maven: ele gerencia para você todos os jars e dependências.

Você pode descobrir centenas de artefatos que já estão cadastrados no repositório central do maven através do site: <http://www.mvnrepository.com/>

10.6 PLUGINS, GOALS E PHASES

E agora, como fazemos para compilar nosso código? Basta digitarmos:

```
mvn compile
```

Você poderia, em vez de chamar essa **phase**, ir diretamente chamar um **goal** do **plugin** compiler:

```
mvn compiler:compile
```

Porém, chamar uma phase tem muito mais vantagens. A phase `compile` chama dois goals:

```
mvn resources:resources compiler:compile
```

então gerar o jar:

```
mvn resources:resources compiler:compile resources:testResources  
    compiler:testCompile surefire:test jar:jar
```

O mais comum é invocarmos phases, e não goals, porém é interessante conhecer alguns goals. O maven possui muitos plugins e goals diferentes já prontos: <http://maven.apache.org/plugins/>

Você pode ler mais sobre o ciclo de vida, fases e goals do maven: <http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html>

10.7 EXERCÍCIOS: BUILD COM O MAVEN

- 1) Em vez de criar um **POM** do zero, vamos pedir auxílio ao próprio maven para que ele crie um projeto de exemplo. Assim, aproveitamos o pom gerado.

Entre no console, e de dentro da sua home, depois de instalar o maven, invoque o gerador:

```
mvn archetype:generate
```

O maven vai fazer vários downloads. Isso é normal, ele sempre tenta buscar versões mais atualizadas das bibliotecas e plugins utilizados.

Vamos escolher o maven-archetype-quickstart (opção 98), que é default. A partir daí, vamos preencher alguns dados, falando sobre o nosso projeto:

```
Define value for groupId: : br.com.caelum
Define value for artifactId: : argentum
Define value for version: 1.0-SNAPSHOT: : (enter)
Define value for package: : br.com.caelum.argentum
```

Finalize o processo confirmando os dados. O maven criou no diretório argentum toda uma estrutura de diretório, que não vamos usar, mas ele gerou o pom.xml algo como segue:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>br.com.caelum</groupId>
  <artifactId>argentum</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>argentum</name>
  <url>http://maven.apache.org</url>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

</project>
```

Vamos usar este como base do nosso pom. Copie este pom para a raiz do seu projeto no workspace do Eclipse, e vamos editá-lo com o Eclipse.

- 2) Agora aproveite do *control-espaco* para te ajudar no preenchimento das tags do maven. Apesar de não ter um plugin para o maven instalado, o Eclipse consegue fazer o autocomplete pois esse xml possui um *schema* definido!

Precisamos escrever em nosso POM quais são as dependências que ele possui. Como vamos saber quais são os artefatos e de que grupo eles pertencem? Pra isso podemos usar o site de busca por projetos que já estão “mavenizados”:

<http://www.mvnrepository.com>

Por exemplo, procure por jfreechart, e escolha a versão 1.09. Verifique que ele mesmo te mostra o trecho de XML que deve ser copiado e colado para o seu pom.

Vamos então declarar nossas quatro dependências (e retire a do junit 3.8):

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.8.1</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>jfree</groupId>
    <artifactId>jfreechart</artifactId>
    <version>1.0.12</version>
  </dependency>
  <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.16</version>
  </dependency>
  <dependency>
    <groupId>com.thoughtworks.xstream</groupId>
    <artifactId>xstream</artifactId>
    <version>1.3.1</version>
  </dependency>
</dependencies>
```

- 3) Além disso, precisamos definir que usamos Java 5 para compilar (forçar que seja usado -source 1.5 pelo compilador).

Adicione, logo abaixo das suas dependências:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <source>1.6</source>
        <target>1.6</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

4) Vamos agora criar um mecanismo para rodar os testes...

Não é necessário! Nosso projeto está configurado. Os diretórios que usamos ao longo desse treinamento são os padrões do Maven, então não precisamos configurar esses dados.

De dentro do diretório do seu projeto (workspace/argentum), rode:

```
mvn test
```

test é considerado um **plugin** do Maven. Cada plugin possui uma série de goals.

5) Para gerar o Jar da sua aplicação:

```
mvn package
```

Pelo Eclipse, dê um refresh no seu projeto e verifique o diretório target.

6) Você pode gerar um site descrevendo diversos dados do seu projeto, para isso:

```
mvn site
```

Agora de refresh no seu projeto, e abra no browser o arquivo `index.html` que está dentro de `target/site`

Você ainda pode adicionar documentação, usando uma estrutura de diretórios e arquivos num formato específico, conhecido como APT:

<http://maven.apache.org/guides/mini/guide-site.html>

7) Esse site possui uma série de relatórios. Podemos incluir alguns interessantes. Um deles é chamado o relatório de cobertura, que indica quanto do seu código está sendo testado pelos unit tests.

Para poder gerar esse relatório, adicione após o build:

```
<reporting>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>cobertura-maven-plugin</artifactId>
    </plugin>
  </plugins>
</reporting>
```

Regere o site e verifique o relatório de cobertura de testes, dentro da aba *project reports*.

Repare que ele até mostra quais linhas do seu código fonte estão sendo passadas por testes, e quais não estão.

8) (opcional) Outro relatório interessante para você testar, adicionando dentro de `reporting/plugins` do nosso `pom.xml`:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
```

```
<artifactId>maven-surefire-report-plugin</artifactId>  
<version>2.7.1</version>  
</plugin>
```

Se quiser explorar mais, PMD e Javadoc são outros dois relatórios para você adicionar ao seu pom.

- 9) (opcional) Invoque o `mvn eclipse:eclipse` Agora de refresh no seu projeto do Eclipse. O que ocorreu? Verifique seu arquivo `.classpath`.

Aproveite e consulte sobre outros plugins padrões do Maven:

<http://maven.apache.org/plugins/>

10.8 USO DENTRO DO ECLIPSE

O Eclipse já vem com suporte integrado ao Ant. Ao editar um arquivo **build.xml**, você já terá autocompletar e recursos de validação das tags e configurações. Para executar o arquivo no Eclipse, basta clicar com o botão direito e ir em **Run As > Ant Build**.

Para o Maven, é possível instalar um plugin, o **Eclipse m2e** que automatiza certas tarefas:

<http://www.eclipse.org/m2e/>

Com ele, a criação e manutenção dos XMLs do Maven são facilitadas. Além disso, você não precisará mais usar a linha de comando, podendo executar os comandos diretamente dentro do Eclipse.

10.9 DISCUSSÃO EM SALA DE AULA: IDE, ANT OU MAVEN?

Apêndice - Mais swing e recursos avançados

Com o que foi visto de Swing até agora, você já deve conseguir fazer diversas outras brincadeiras apenas procurando mais recursos na API e lembrando que, trabalhando com Swing, sempre usaremos composição de componentes!

Esse capítulo mostra algumas outras modificações que você pode fazer no Argentum para torná-lo mais interessante. Além disso, lendo mais adiante você também conhecerá um recurso poderoso ao lidar com processos demorados dentro do Swing.

11.1 INPUT DE DADOS FORMATADOS: DATAS

Usar entrada de datas em um sistema Swing é extremamente fácil: podemos usar a classe `JTextField` que permite obter e alterar os dados digitados. Essa classe é muito flexível também, possibilitando alterar vários aspectos da exibição dos dados, dos formatos, dos eventos associados, etc.

No nosso sistema Argentum, vamos adicionar um campo para entrada de data que permita filtrar a lista de negócios usada para exibir a tabela e o gráfico. Queremos um campo de texto com barras para separar campos da data e, que aceite apenas datas com formatação válida. Uma classe filha de `JTextField` já faz boa parte desse trabalho: a `JFormattedTextField`.

Podemos criar um `JFormattedTextField` e associar a ele um `javax.swing.text.Formatter`, como um `DateFormatter`, um `NumberFormatter` ou um `MaskFormatter`. Vamos usar o `MaskFormatter` para aplicar uma máscara no campo e permitir um *input* de datas fácil pelo usuário:

```
MaskFormatter mascara = new MaskFormatter("##/##/####");  
mascara.setPlaceholderCharacter('_');
```

```
JFormattedTextField campoDataInicio = new JFormattedTextField(mascara);
```

Repare na chamada ao `setPlaceholderCharacter` que diz ao componente para colocar um underscore nos espaços de digitação enquanto o usuário não preenche com a data desejada. Após o usuário digitar, podemos obter o valor chamando o `getValue`. A partir daí, podemos transformar o objeto em um `Date` para uso:

```
String valor = (String) campoDataInicio.getValue();
```

```
DateFormat formato = new SimpleDateFormat("dd/MM/yyyy");  
Date date = formato.parse(valor);
```

11.2 EXERCÍCIOS OPCIONAIS: FILTRANDO POR DATA

- 1) Abra a classe `ArgentumUI` e adicione a chamada ao `montaCampoData` ao método `montaTela` (cuidado com a ordem):

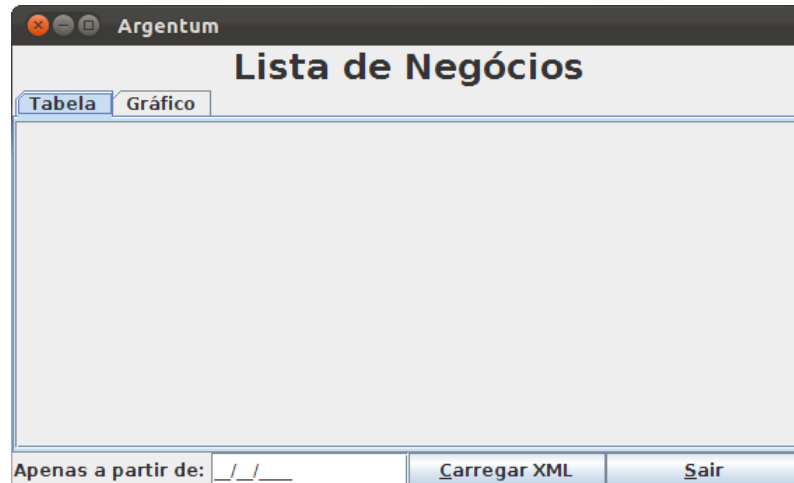
```
public void montaTela() {  
    preparaJanela();  
    preparaPainelPrincipal();  
    preparaAbas();  
    preparaTabela();  
    preparaPainelBotoes();  
    preparaCampoData();  
    preparaBotaoCarregar();  
    preparaBotaoSair();  
    mostraJanela();  
}
```

- 2) Usando o **ctrl + 1** na linha recém adicionada, indique para o Eclipse que ele crie esse novo método (`preparaCampoData`). Implemente o método como abaixo, prestando atenção no `campoData`, que deve ser um **atributo** (use o *Assign statement to new field* ou equivalente):

```
private void preparaCampoData() {  
    try {  
        JLabel labelData = new JLabel("Apenas a partir de:");  
  
        MaskFormatter mascara = new MaskFormatter("##/##/####");  
        mascara.setPlaceholderCharacter('_');  
        campoData = new JFormattedTextField(mascara);  
  
        painelBotoes.add(labelData);  
        painelBotoes.add(campoData);  
    }  
}
```

```
    } catch (ParseException e) {  
        e.printStackTrace();  
    }  
}
```

3) Rode a aplicação e veja o resultado:



4) Agora, precisamos implementar o filtro da data no momento de carregar o XML.

Se o campoData estiver preenchido, **adicione** dentro do carregaDados uma chamada ao método filtra de uma nova classe FiltradorPorData, que implementaremos na sequência. Faça isso na linha **logo após** a criação da lista.

```
private void carregaDados() {  
    List<Negocio> lista = new EscolhedorDeXML().escolhe();  
    if (campoData.getValue() != null) {  
        new FiltradorPorData(campoData.getText()).filtra(lista);  
    }  
  
    ArgentumTableModel ntm = new ArgentumTableModel(lista);  
    //...
```

5) Vamos pedir que o Eclipse **crie essa classe** que recebe uma dataDigitada no construtor e a converte para um Calendar. Abuse do **ctrl + 1** para criar a classe e o construtor. **Complete** a implementação do construtor:

```
public class FiltradorPorData {  
  
    private Calendar dataInicial;  
  
    public FiltradorPorData(String dataDigitada) {  
        try {  
            SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
```



```

    dataInicial = Calendar.getInstance();
    dataInicial.setTime(sdf.parse(dataDigitada));
  } catch (ParseException e) {
    throw new RuntimeException(e);
  }
}
}

```

- 6) Estamos quase lá! Agora, só falta criar o método `filtra` que passa pela lista de negócios lida do XML removendo dela todos os negócios que sejam de datas anteriores à data selecionada pelo usuário.

Dentro da `ArgentumUI` ainda deve haver um erro de compilação informando de o método `filtra` ainda não existe na classe `FiltradorPorData`. Mande o Eclipse criar esse método para você e o **complete** com o código para remover os Negócios de antes da data indicada.

Existe uma limitação da lista quando tentamos fazer uma remoção nela dentro de um laço que a percorre. Para conseguir fazer isso, usaremos o `Iterator`:

```

1 public void filtra(List<Negocio> lista) {
2     if (dataInicial == null)
3         return;
4     Iterator<Negocio> it = lista.iterator();
5     while (it.hasNext()) {
6         if (it.next().getData().before(dataInicial))
7             it.remove();
8     }
9 }

```

- 7) Rode novamente sua aplicação e faça alguns testes. Teste carregar sem a data setada, setando uma data válida, uma inválida, etc.



The screenshot shows a window titled "Argentum" with a tabbed interface. The "Tabela" tab is selected, displaying a table of stock trades. The table has four columns: "Preço", "Quantidade", "Data", and "Volume". The data is filtered to show trades from 25/03/2012 onwards. At the bottom of the window, there is a text field labeled "Apenas a partir de:" with the date "25/03/2012" entered, and two buttons: "Carregar XML" and "Sair".

Preço	Quantidade	Data	Volume
R\$ 36,84	21876	25/03/2012	R\$ 805.911,84
R\$ 37,23	22533	25/03/2012	R\$ 838.903,59
R\$ 37,52	23233	25/03/2012	R\$ 871.702,16
R\$ 38,49	23935	25/03/2012	R\$ 921.258,15
R\$ 38,67	24608	25/03/2012	R\$ 951.591,36
R\$ 38,87	25518	25/03/2012	R\$ 991.884,66
R\$ 38,75	26469	25/03/2012	R\$ 1.025.673,75
R\$ 38,89	27194	25/03/2012	R\$ 1.057.574,66
R\$ 38,15	27701	25/03/2012	R\$ 1.056.793,15
R\$ 37,80	28625	25/03/2012	R\$ 1.082.025,00
R\$ 38,67	29575	25/03/2012	R\$ 1.143.665,25
R\$ 38,12	30270	25/03/2012	R\$ 1.153.892,40

Apenas a partir de: 25/03/2012 Carregar XML Sair

ITERATOR E REMOÇÕES

Sempre use a classe `Iterator` quando precisar remover elementos de uma `Collection` enquanto percorre seus elementos.

- 8) (opcional) Repare que, ao colocar valores que extrapolem o limite do campo (por exemplo um mês 13), é assumido que é o primeiro mês do ano seguinte. Para mudar esse comportamento, chame `setLenient(false)` no `SimpleDateFormat` que estamos usando.

```
public FiltradorPorData(String dataDigitada) {  
    try {  
        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");  
        sdf.setLenient(false);  
  
        //...
```

O que acontece quando você tenta filtrar uma data inválida agora?

Trate a exceção no método `carregaDados`, limpando o `campoData` e dando uma mensagem de *pop-up* para avisar o que aconteceu.

11.3 PARA SABER MAIS: BARRA DE MENU

Para criar uma barra de menu no topo de nossas janelas, basta usarmos a classe `JMenuBar` do Swing. Dentro de um `JMenuBar`, colocamos vários `JMenu` que representam cada menu (File, Edit, View Help etc). E, dentro de cada `JMenu`, colocamos vários `JMenuItem`s que representam cada ação (Abrir, Salvar, Sair etc).

Existem ainda duas subclasses de `JMenuItem`, mais específicas, que transformam aquela entrada do menu em um checkbox ou radio button, `JCheckBoxMenuItem` e `JRadioButtonMenuItem`.

```
JMenuBar menuBar = new JMenuBar();  
janela.setJMenuBar(menuBar);  
  
JMenu menuInicio = new JMenu("Início");  
menuBar.add(menuInicio);  
  
JMenu carregar = new JMenu("Carregar");  
menuInicio.add(carregar);  
  
JMenu sair = new JMenu("Sair");  
menuInicio.add(sair);  
  
JCheckBoxMenuItem check = new JCheckBoxMenuItem("Checkbox");  
menuInicio.add(check);
```

E podemos tratar os eventos desses menus da mesma forma que com botões, com `ActionListeners`.

11.4 EXERCÍCIO: ESCOLHENDO INDICADORES PARA O GRÁFICO

Nosso gráfico não nos dá a chance de escolher quais indicadores queremos ver. Vamos, através de um `JMenuBar`, possibilitar ao usuário escolhê-los. Para isso, criaremos uma barra de menu em uma classe auxiliar chamada `MenuIndicadores`.

- 1) Na classe `ArgentumUI`, **altere** novamente o método `montaTela` para chamar o método `preparaMenu` logo abaixo da janela:

```
public void montaTela() {  
    preparaJanela();  
    preparaMenu();  
    preparaPainelPrincipal();  
    preparaAbas();  
    preparaTabela();  
    preparaPainelBotoes();  
    preparaCampoData();  
    preparaBotaoCarregar();  
    preparaBotaoSair();  
    mostraJanela();  
}
```

- 2) Faça o Eclipse criar esse método para você e o complete como abaixo:

```
private void preparaMenu() {  
    menu = new MenuIndicadores();  
    janela.setJMenuBar(menu.getMenuBar());  
}
```

- 3) Crie o construtor da nova classe `MenuIndicadores` que o Eclipse gerará para você através do **ctrl + 1** e, nele, instancie cada um dos indicadores possíveis e crie um `JCheckBoxMenuItem` para cada um deles. Para facilitar a recuperação dos objetos, guardaremos em um `HashMap` os itens do menu e seus respectivos indicadores.

Seu código ficará parecido com esse:

```
public MenuIndicadores() {  
    List<Indicador> indicadores = new ArrayList<Indicador>();  
    indicadores.add(new IndicadorFechamento());  
    indicadores.add(new MediaMovelSimples(new IndicadorFechamento()));  
    indicadores.add(new MediaMovelPonderada(new IndicadorFechamento()));  
    // mais indicadores, se houver  
  
    menuBar = new JMenuBar();
```

```

JMenu menuIndicadores = new JMenu("Indicadores");
menuBar.add(menuIndicadores);

indicadoresNoMenu = new HashMap<JCheckBoxMenuItem, Indicador>();

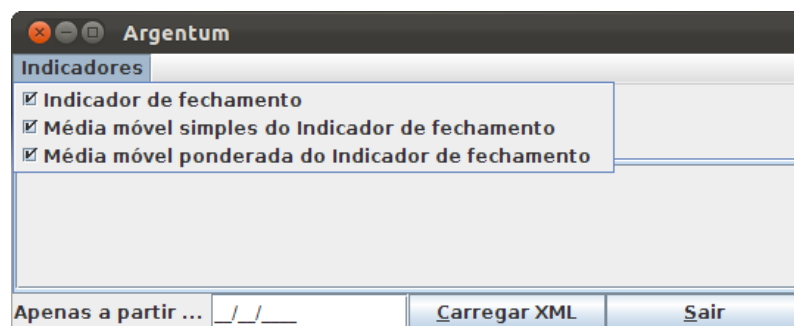
for (Indicador indicador : indicadores) {
    JCheckBoxMenuItem opcaoIndicador =
        new JCheckBoxMenuItem(indicador.toString(), true);
    menuIndicadores.add(opcaoIndicador);
    indicadoresNoMenu.put(opcaoIndicador, indicador);
}
}

```

- 4) No código acima, você deve ter notado que a barra de menu menuBar foi transformada em um atributo. Precisamos, agora, de um getter para esse atributo. Na classe MenuIndicadores, faça:

```
getM <ctrl + espaço>
```

- 5) Rode sua aplicação e teste o menu, que, por enquanto não faz nada:



- 6) Precisamos agora alterar o método carregaDados para usar os indicadores selecionados no menu. **Altere** as linhas que tratam de chamar o plotaIndicador para iterar pela lista de indicadores selecionados:

```

private void carregaDados() {
    // algumas linhas...

    GeradorDeGrafico gerador = new GeradorDeGrafico(serie, 2, serie.getTotal() - 1);
    List<Indicador> indicadores = menu.getIndicadoresSelecionados();
    for (Indicador indicador : indicadores) {
        gerador.plotaIndicador(indicador);
    }
    abas.setComponentAt(1, gerador.getPanel());
}

```

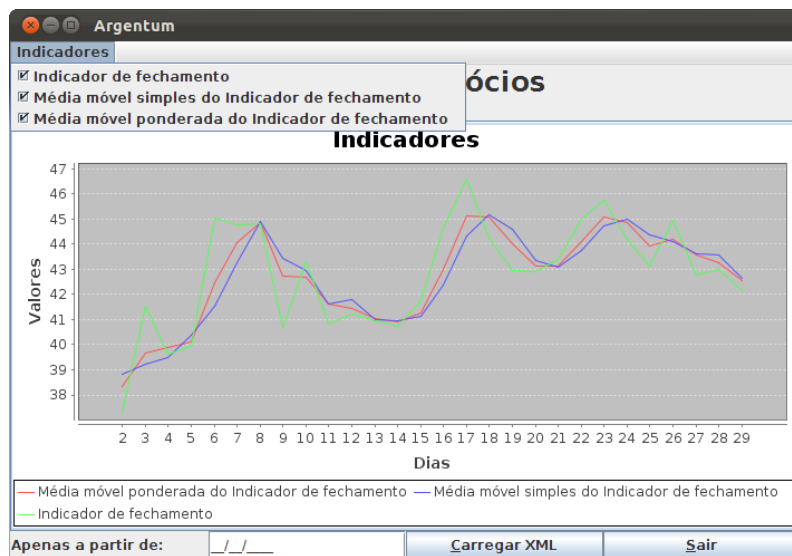
- 7) Agora, só falta implementar esse método getIndicadoresSelecionados na classe MenuIndicadores. Deixe que o Eclipse crie o cabeçalho do método para você e apenas o preencha verificando quais itens do menu foram marcados (item.isSelected()) e devolvendo os Indicadores de acordo.

```

public List<Indicador> getIndicadoresSelecionados() {
    ArrayList<Indicador> indicadores = new ArrayList<Indicador>();
    for (JCheckBoxMenuItem menuItem : indicadoresNoMenu.keySet()) {
        if (menuItem.isSelected())
            indicadores.add(indicadoresNoMenu.get(menuItem));
    }
    return indicadores;
}

```

8) Rode a aplicação novamente e teste escolher uma data e os indicadores.



9) (opcional) Faça mais indicadores e adicione-os ao menu. Se for necessário, crie mais menus para melhorar a usabilidade.

10) (opcional) Faça com que as ações e “Carregar XML” e “Sair” também possam ser executadas via alguma opção no menu bar.

11.5 DIFICULDADES COM THREADS E CONCORRÊNCIA

Quando rodamos uma aplicação com Swing/AWT, uma das maiores dificuldades é trabalhar com **threads**. Isso porque temos que saber gerenciar o desenho e atualização da tela ao mesmo tempo que queremos executar as lógicas de negócio pesadas e complexas.

Toda vez que rodamos uma aplicação Java, como sabemos, executamos o método `main` na *thread* principal do programa. Quando rodamos um programa com Swing/AWT, uma nova *thread* é criada, além da principal, para tratar do desenho da interface e da interação do usuário.

Isso quer dizer que quando criamos *listeners* para eventos do Swing, eles são executados pela *thread* do AWT e não pela *thread* do `main`. É considerado má prática de programação executar tarefas pesadas (com muita

contenção) dentro desses *listener* para não travar a *thread* do AWT e deixar a interface com aparência de travada.

Faça o teste: pegue um XML de negócios **bem** grande (50 mil linhas ou mais) e mande seu programa carregá-lo. Ao mesmo tempo, clique no botão de sair. Repare que o evento demora muito para ser processado.

Quando formos executar operações potencialmente pesadas, a boa prática é que o listener do evento não faça essa execução, mas dispare uma nova *thread* para essa execução. O problema é que quando executamos alguma lógica de negócios é muito comum ao final fazer alguma alteração nos componentes visuais -- no nosso programa, depois de processar o XML, precisamos atualizar o `TableModel` do `JTable`, por exemplo.

Mas toda atualização nos componentes deve ser feita no AWT e não na nova *thread*, já que essa é a única forma de evitar completamente problemas de concorrência: o AWT mantém uma fila de tarefas a serem processadas e as executa sem perigo de problemas de concorrência.

Gerenciar essas *threads* todas e conseguir sincronizar tudo corretamente, executando tudo no seu devido lugar, é um dos pontos mais complexos do Swing. A partir do Java 6 existe uma classe que procura simplificar esse trabalho todo: `SwingWorker`.

11.6 SWINGWORKER

O princípio por trás da classe `SwingWorker` é permitir executar códigos em uma *thread* nova concorrentemente a outras tarefas da *thread* do AWT. Essa classe traz toda a infra-estrutura necessária para fazer a comunicação entre essas *threads* de forma segura.

Para fazê-la funcionar, implementamos dois métodos:

- `doInBackground()` - contém o código pesado que deve ser executado na *thread* concorrente. Pode devolver algum objeto que será enviado para a *thread* do AWT;
- `done()` - é executado na *thread* do AWT logo que o `doInBackground` termina. Aqui podemos recuperar o objeto devolvido na outra *thread* e atualizar itens da interface.

Temos que escrever uma classe que herda de `SwingWorker` e implementar esses métodos. Na hora de estender o `SwingWorker`, precisamos passar dois argumentos para seu *generics*: o tipo do objeto que o `doInBackground` devolve e um outro tipo usado (opcionalmente) para indicar progresso (que não precisamos agora).

Vamos ver uma implementação inicial:

```
public class CarregadorXMLWorker extends SwingWorker<List<Negocio>, Void> {  
  
    private final JTable tabela;  
  
    public CarregadorXMLWorker(JTable tabela)  
    {  
        this.tabela = tabela;  
    }  
}
```

```
@Override
protected List<Negocio> doInBackground() throws Exception {
    return new EscolheXML().escolher();
}

@Override
protected void done() {
    NegociosTableModel model = new NegociosTableModel(get());
    tabela.setModel(model);
}
}
```

Repare que o `doInBackground` **não** salva nada em atributos. O método `done` obtém a lista de negócios a partir da chamada ao método `get`. Isso é feito para garantir *thread-safety*. Repare também que passamos a tabela como argumento no construtor, já que vamos precisar modificá-la.

BARRA DE PROGRESSO E OUTRAS POSSIBILIDADES

A classe `SwingWorker` traz ainda outros recursos mais avançados, como a capacidade de se implementar uma barra de progresso fazendo com que a thread em background notifique a thread do AWT sobre pequenos progressos feitos no seu processamento. Tudo de forma *thread-safe* e robusta.

11.7 EXERCÍCIOS: PESQUISANDO MAIS NA API

- 1) Procure saber mais sobre o uso do `SwingWorker` nos tutoriais da Oracle.

Esses tutoriais vêm desde o tempo da Sun e trazem uma boa idéia do que o Swing oferece aos desenvolvedores Java.

- 2) Outro ponto interessante são aqueles em que a API em si não nos ajuda tanto quanto gostaríamos.

Felizmente, a comunidade do Java é bastante forte e podemos tirar proveito disso! Procure, por exemplo, componentes *open source* que fazem abrir um widget de calendário para escolhermos uma data.

Apêndice - Logging com Log4j

12.1 USANDO LOGS - LOG4J

O que fazer quando rodamos um teste e ele não passa? Como saber o que deu errado para corrigir? Ou pior: como saber o que deu errado quando acontece um erro em produção?

Normalmente, as respostas para essas perguntas são sempre duas: usamos o Debug para saber o que acontece ou colocamos alguns `System.out.println` para ver algum resultado e analisá-lo.

Mas debug não está disponível em produção e, mesmo em desenvolvimento, pode ser difícil de usar. `System.outs` são bastante limitados e normalmente soluções temporárias.

Existe uma maneira mais robusta de obter informações sobre o fluxo de execução de nosso programa sem recorrer a debugs ou sysouts: utilizar uma API de Logging. Boas APIs nos permitem ativar/desativar sem alterar código, pode ser persistido em algum lugar (arquivos, BD etc) e ainda podemos controlar o nível de informações que queremos ver (desde bem detalhadas a ver apenas os erros que acontecem).

Há várias bibliotecas de logging no mercado. O log4j da Apache é o mais conhecido e o mais usado no mercado hoje. Vamos usá-lo também. O uso desta biblioteca é tão disseminado que a chance de utilizarmos uma outra biblioteca (Spring, Hibernate, Lucene, etc) que já depende do log4j é muito grande.

Para usarmos o log4j precisamos do jar do log4j e de um arquivo de configuração, que pode ser `.properties` ou `.xml`. Basicamente, escrevemos em nossas classes uma linha de código para obtermos um `Logger` e, de posse deste, chamamos os métodos que escrevem as mensagens.

Para inicializar seu log, utilizamos a fábrica `Logger.getLogger`, que, dada uma `String` (a categoria do log), devolve um `Logger`. É comum usarmos a sobrecarga desse método que recebe um `class` como argumento:

```
Logger logger = Logger.getLogger(MinhaClasse.class);
```

Dessa forma ele vai nomear esse logger com o full qualified name dessa classe. Agora podemos usar o logger em seus vários níveis: trace, debug, info, warn e error. Por exemplo:

```
logger.warn("Ocorreu um problema ao gerar relatório pdf: " + problema);
```

12.2 NÍVEIS DE LOGS

Há diversos níveis de log disponíveis, inclusive a possibilidade de se criar os próprios níveis (o que não é muito recomendado). Os que vem por padrão no Log4J são:

- TRACE
- DEBUG
- INFO
- WARN
- ERROR
- FATAL

A ideia é representar mensagens de log da menor gravidade para a maior gravidade. Há métodos de logging para cada um desses níveis.

12.3 APPENDERS E LAYOUT

Mas onde essas mensagens vão sair? Num arquivo texto? Na saída padrão? Todos no mesmo arquivo? Você tem o controle total sobre qual mensagem vai sair em que lugar através do XML de configuração. Para isso, existem dois conceitos importantes nesta api:

- 1) **Appenders:** appenders são as classes que efetivamente escrevem a mensagem em algum lugar. Existem appenders prontos para escrever no console, enviar um email, gravar no banco e muitos outros. Ainda existe a possibilidade de escrevermos nossos próprios appenders, como por exemplo um que enviasse uma mensagem SMS. Os appenders que são distribuídos com o log4j são extremamente configuráveis. É possível customizar o formato das mensagens, tamanho de arquivos, backup etc.

```
<appender name="stdout" class="org.apache.log4j.ConsoleAppender">  
  <layout class="org.apache.log4j.PatternLayout">  
    <param name="ConversionPattern"
```

```
value="%d{HH:mm:ss,SSS} %5p [%-20c{1}] %m%n" />
</layout>
</appender>
```

- 2) **Categories:** categories configuram appenders específicos para partes específicas de nossa aplicação. Por exemplo gostaríamos que o módulo que acessa o banco de dados utilizasse um appender que escreve em arquivo, mas o módulo da interface com o usuário apenas escrevesse no console. Também é importante notar que nas categories definimos qual o nível de severidade mínimo das mensagens para que ela seja escrita. Por exemplo, gostaríamos que o módulo que envia emails só mostre mensagens do tipo INFO para cima. Desta forma, mensagens do tipo INFO, WARN e ERROR seria escritas.

```
<category name="br.com.caelum">
  <priority value="WARN" />

  <!--
    "stdout" referencia o nome do
    appender declarado anteriormente
  -->
  <appender-ref ref="stdout" />
</category>
```

12.4 EXERCÍCIOS: ADICIONANDO LOGGING COM LOG4J

- 1) Adicione os jars do log4J ao buildpath do Argentum, como já feito anteriormente para adicionar outras bibliotecas (importe para o diretório lib, e depois adicione ao build path através da view package explorer, ou nas propriedades do projeto).
- 2) Na classe MediaMovelSimples, vamos adicionar um logger como atributo:

```
private static final Logger logger =
    Logger.getLogger(MediaMovelSimples.class);
```

Logo no começo do método calcula adicione a informação de log:

```
logger.info("Calculando média móvel simples para posição " + posicao);
```

Rode o MediaMovelSimplesTest. O que saiu no logger?

- 3) Configure o log4J propriamente. Para isso, crie um diretório dentro de src/test que se chama resources, e jogue lá dentro o arquivo log4j.xml como se segue:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">

  <appender name="stdout" class="org.apache.log4j.ConsoleAppender">
    <layout class="org.apache.log4j.PatternLayout">
```

```
<param name="ConversionPattern"
        value="%d{HH:mm:ss,SSS} %5p [%-20c{1}] %m%n" />
</layout>
</appender>

<root>
    <level value="TRACE" />
    <appender-ref ref="stdout" />
</root>

</log4j:configuration>
```

Adicione esse diretório como source folder do seu projeto, fazendo assim com que esse arquivo consiga ser encontrado pelo Log4J. Rode novamente o teste.

- 4) Adicione o log analogamente na classe `MediaMoveiPonderada`.
- 5) Adicione o log analogamente na classe `MediaMoveiPonderada`.

12.5 O SL4J

O **Simple Logging Facade for Java** (SLF4J) é uma abstração de logging que expõe uma API única para diversos projetos de log. É possível usar Log4J, `java.util.logging` ou `logback`, por exemplo, com a mesma API, diminuindo o acoplamento do código com algum framework específico.

Sua API é bem familiar:

```
Logger logger = LoggerFactory.getLogger(MediaMoveiSimples.class);
logger.info("Calculando média móvel");
```

Para usá-lo, você precisa do **slf4j-api.jar** e da implementação específica para algum framework específico, como **slf4j-log4j.jar**. Nesse caso, toda configuração deverá ser feita no `log4j.xml` como antes, apenas o código fica desacoplado.

Índice Remissivo

Anotações, 137
anotações, 35
AWT, 70

BorderLayout, 124
BoxLayout, 123

Calendar, 12
Candlestick, 5, 8

datas, 12
DateFormat, 87
Design patterns, 116

Factory pattern, 13
final, 9
FlowLayout, 121
Formatter, 145

GridBagLayout, 124
GridLayout, 122

JButton, 74
JCheckboxMenuItem, 162
JFileChooser, 72
JFormattedTextField, 158
JFrame, 74
JFreeChart, 105
JMenu, 162
JMenuBar, 162
JMenuItem, 162
JOptionPane, 72
JPanel, 74
JRadioButtonMenuItem, 162
JTabbedPane, 127
JTable, 82
JTextField, 158
JUnit, 33

LaF - Look-and-Feel, 71
Layout Manager, 121

Maven, 151

Negocio, 8

OutputStream, 108

Reflection, 136

static import, 48
Swing, 70
SwingWorker, 166
SWT, 70

Tabelas, 82
Tail, 6
TDD, 62
Test driven design, 62
testes de unidade, 32
testes unitários, 32