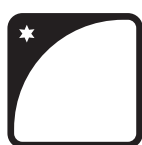


FJ-16

Laboratório Java com Testes,
XML e Design Patterns



Caelum
Ensino e Inovação

www.caelum.com.br



A Caelum atua no mercado com consultoria, desenvolvimento e ensino em computação. Sua equipe participou do desenvolvimento de projetos em vários clientes e, após apresentar os cursos de verão de Java na Universidade de São Paulo, passou a oferecer treinamentos para o mercado. Toda a equipe tem uma forte presença na comunidade através de eventos, artigos em diversas revistas, participação em muitos projetos *open source* como o VRaptor e o Stella e atuação nos fóruns e listas de discussão como o GUJ.

Com uma equipe de mais de 60 profissionais altamente qualificados e de destaque do mercado, oferece treinamentos em Java, Ruby on Rails e Scrum em suas três unidades - São Paulo, Rio de Janeiro e Brasília. Mais de 8 mil alunos já buscaram qualificação nos treinamentos da Caelum tanto em nas unidades como nas próprias empresas com os cursos *incompany*.

O compromisso da Caelum é oferecer um treinamento de qualidade, com material constantemente atualizado, uma metodologia de ensino cuidadosamente desenvolvida e instrutores capacitados tecnicamente e didaticamente. E oferecer ainda serviços de consultoria ágil, mentoring e desenvolvimento de projetos sob medida para empresas.

Comunidade



Nossa equipe escreve constantemente artigos no **Blog da Caelum** que já conta com 150 artigos sobre vários assuntos de Java, Rails e computação em geral. Visite-nos e assine nosso RSS:

➡ blog.caelum.com.br



Acompanhe também a equipe Caelum no **Twitter**:

➡ twitter.com/caelumdev/equipe



O **GUJ** é maior fórum de Java em língua portuguesa, com 700 mil posts e 70 mil usuários. As pessoas da Caelum participam ativamente, participe também:

➡ www.guj.com.br



Assine também nossa **Newsletter** para receber as novidades e destaques dos eventos, artigos e promoções da Caelum:

➡ www.caelum.com.br/newsletter



No site da Caelum há algumas de nossas **Apostilas** disponíveis gratuitamente para download e alguns dos **artigos** de destaque que escrevemos:

➡ www.caelum.com.br/apostilas

➡ www.caelum.com.br/artigos

Conheça alguns de nossos cursos



FJ-11:
Java e Orientação a objetos



FJ-26:
Laboratório de MVC com
Hibernate e JSF para a Web



FJ-16:
Laboratório Java com Testes,
XML e Design Patterns



FJ-31:
Java EE avançado e
Web Services



FJ-19:
Preparatório para Certificação
de Programador Java



FJ-91:
Arquitetura e Design de
Projetos Java



FJ-21:
Java para Desenvolvimento
Web



FJ-27:
Spring Framework



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.



Caelum
Ensino e Inovação

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

Índice

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	2
1.4	Mais bibliografia	2
1.5	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	4
2.3	O projeto Tail	5
2.4	O projeto Argentum: modelando o sistema	7
2.5	Palavra chave final	8
2.6	Imutabilidade de objetos	9
2.7	Trabalhando com datas: Date e Calendar	10
2.8	Exercícios: o modelo do Argentum	13
2.9	Exercícios: fábrica de Candlestick	21
2.10	Exercícios opcionais	23
3	Testes Automatizados	25
3.1	Nosso código está funcionando corretamente?	25
3.2	Exercícios: testando nosso modelo sem frameworks	25
3.3	Definindo melhor o sistema e descobrindo mais bugs	27

3.4	Testes unitários	28
3.5	JUnit	28
3.6	Anotações	30
3.7	JUnit 4 e suas anotações	30
3.8	Test Driven Development - TDD	32
3.9	Exercícios: migrando os testes do main para JUnit	32
3.10	Exercícios: novos testes	37
3.11	Para saber mais: Import Estático	40
3.12	Mais exercícios opcionais	41
3.13	Discussão em aula: testes são importantes?	42
4	Trabalhando com XML	43
4.1	Os dados da bolsa de valores	43
4.2	XML	44
4.3	Lendo XML com Java de maneira difícil, o SAX	45
4.4	XStream	47
4.5	Exercícios: Lendo o XML	49
4.6	Exercícios: Separando os candles	53
4.7	Exercícios opcionais	56
4.8	Discussão em aula: Onde usar XML e o abuso do mesmo	57
5	Interfaces gráficas com Swing	58
5.1	Interfaces gráficas em Java	58
5.2	Portabilidade	58
5.3	Look And Feel	58
5.4	Componentes	59
5.5	Começando com Swing - Diálogos	59
5.6	Exercícios: Escolhendo o XML com JFileChooser	60
5.7	Componentes: JFrame, JPanel e JButton	61
5.8	O design pattern Composite: Component e Container	62
5.9	Tratando eventos	63
5.10	Classes internas e anônimas	63
5.11	Exercícios: nossa primeira tela	64

5.12 JTable	66
5.13 Implementando um TableModel	68
5.14 Exercícios: Tabela	69
5.15 Formatando Datas: DateFormat	73
5.16 Exercícios: formatação	73
5.17 Para saber mais	75
5.18 Discussão em sala de aula: Listeners como classes top level, internas ou anônimas?	75
6 Refatoração: os Indicadores da bolsa	76
6.1 Análise Técnica da bolsa de valores	76
6.2 Indicadores Técnicos	77
6.3 As médias móveis	78
6.4 Exercícios: criando indicadores	80
6.5 Refatoração	83
6.6 Exercícios: Primeiras refatorações	83
6.7 Nossos indicadores e o design pattern Strategy	85
6.8 Exercícios: refatorando para uma interface e usando bem os testes	86
6.9 Exercícios opcionais	87
6.10 Discussão em aula: quando refatorar?	88
7 Gráficos com JFreeChart	89
7.1 JFreeChart	89
7.2 Utilizando o JFreeChart	89
7.3 Isolando a API do JFreeChart: baixo acoplamento	91
7.4 Para saber mais: Design Patterns Factory Method e Builder	93
7.5 Exercícios: JFreeChart	93
7.6 Exercícios opcionais	97
7.7 Indicadores mais Elaborados e o Design Pattern Decorator	97
7.8 Exercícios: Indicadores mais espertos e o Design Pattern Decorator	98
7.9 Desafio: Imprimir Candles	100
7.10 Desafio: Fluent Interface	100

8 Mais Swing: layout managers, mais componentes e detalhes	102
8.1 Gerenciadores de Layout	102
8.2 Layout managers mais famosos	104
8.3 Exercícios: usando layout managers	105
8.4 Integrando JFreeChart	107
8.5 Exercícios: completando a tela da nossa aplicação	108
8.6 Input de dados formatados: Datas	110
8.7 Exercícios: filtrando por data	111
8.8 Para saber mais: barra de menu	113
8.9 Exercícios opcionais: escolher os indicador(es) para o gráfico	114
8.10 Discussão em sala de aula: uso de IDEs para montar a tela	116
9 Reflection e Annotations	117
9.1 Por que Reflection?	117
9.2 Reflection: Class, Method	118
9.3 Usando anotações	119
9.4 Usar JTables é difícil	119
9.5 Criando sua própria anotação	119
9.6 Lendo anotação com Reflection	121
9.7 TableModel com Reflection	121
9.8 Exercícios: ReflectionTableModel	122
9.9 Para saber mais: Formatter, printf e String.format	124
9.10 Exercícios opcionais	125
9.11 Discussão em sala de aula: quando usar reflection, anotações e interfaces	126
10 Apêndice: O processo de Build: Ant e Maven	127
10.1 O processo de build	127
10.2 O Ant	127
10.3 Exercícios com Ant	127
10.4 O Maven	129
10.5 O Project Object Model	129
10.6 Plugins, goals e phases	130
10.7 Exercícios: build com o Maven	131
10.8 Discussão em sala de aula: IDE, ant ou Maven?	134

11 Apêndice - Swing Avançado	135
11.1 Dificuldades com Threads e concorrência	135
11.2 SwingWorker	135
11.3 Exercícios: resolvendo concorrência com SwingWorker	136
11.4 Para saber mais: A parte do JFreeChart	137
12 Apêndice - Logging com Log4j	139
12.1 Usando logs - LOG4J	139
12.2 Exercícios: Adicionando logging com Log4J	140

Versão: 12.4.10

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

Porque 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 unitários (conhecido também como testes de unidade) 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 do atual líder do projeto Eclipse na IBM, Erich Gamma, 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 “Programa 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

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

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

1.4 - Mais bibliografia

Além dos livros já citados, podemos incluir:

- **Test Driven Development: By Example**, Kent Beck.
- **Domain Driven Design**, Eric Evans.
- **Swing Second Edition**, Matthew Robinson e Pavel Vorobiev, editora Manning;
- **Tutorial oficial de Swing**, em <http://java.sun.com/docs/books/tutorial/uiswing/>.

1.5 - Para onde ir depois?

O FJ-21 é indicado para ser feito antes ou depois deste curso, dependendo das suas necessidades e do seu conhecimento.

Depois destes cursos, que constituem a formação consultor Java da Caelum, indicamos dois outros cursos, da formação avançada:

<http://www.caelum.com.br/curso/formacao-consultor-java-ee-avancado/>

O FJ-26 aborda JSF e Hibernate e o FJ-31 envolve EJBs, RMI, serialização, JMS e *web services* utilizando a plataforma Java EE, além de muitos detalhes arquiteturais de um projeto. Ambos passam por diversos *design patterns*.

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, métodos ágeis de programação e a importância dos testes unitários. 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 tão atual 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, dada suas habilidades de transporte e precificação do arroz, uma mercadoria em crescente produção em consumo no país. Sua situação social, de mercador, não permitia que ele fosse tão rico dado o sistema de castas da época, e logo o governo confiscou todo seu dinheiro e suas posses. Depois dele outros vieram e tentaram esconder suas origens como mercadores: muitos tiveram seus filhos executados e seu dinheiro confiscado.

Apesar da triste história, foi em Dojima, no jardim do mesmo Yodoya Keian, que nasceu a bolsa de arroz do Japão. Lá eram negociados vários tipos de arroz, e também eram precificados e categorizados.

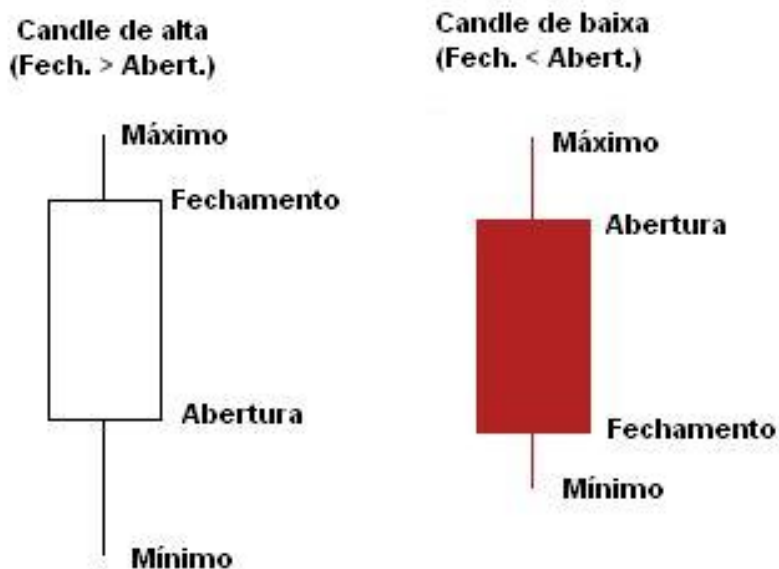
Para anotar os preços do arroz desenhavam-se figuras no papel. Essas figuras parecem muito como velas (daí a analogia **candlestick**, um candelabro, gráfico que carrega várias velas, ou **candles**).

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).

O preço de abertura e fechamento depende 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 branco, e um de baixa costuma ser vermelho ou preto. Caso o preço não tenha se movimentado, o candle tem a mesma cor que o do dia anterior.

Para calcular as informações necessárias para a construção de um candlestick, é necessário os dados de todos os **negócios** (*trades*) de um dia. Um **negócio** possui três informações: o **preço** em que foi fechado, a **quantidade** negociada, e a **data** que foi executado.



Você pode ler mais sobre a história dos candles em: http://www.candlestickforum.com/PPF/Parameters/1_279_/candlestick.asp <http://www.thepitmaster.com/tricks/japanesecandlesticks.htm>

Uma outra forma de representar os candles mais sucintamente é através de barras, que são análogas:



Apesar de falarmos que o candlestick representa o preço 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** (**T**echnical **A**nalysis **I**ndicator **L**ibrary) nasceu quando um grupo de alunos da Universidade de São Paulo procurou o professor doutor Alfredo Goldman para orientá-los no desenvolvimento de um

software para o projeto de conclusão de curso.

Ele então teve a 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. So-mente então o grupo definiu o tema: o desenvolvimento de um software open source de análise técnica (veremos o que é a análise técnica em capítulos posteriores).

O software está disponível no sourceforge:

<http://tail.sourceforge.net/>



Essa idéia, 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 Manssuer, Márcio Vinicius dos Santos, Thiago Garutti Thies, Paulo Silveira (mestre em Geometria Computacional pela USP, e diretor de treinamentos da Caelum), Julian Monteiro (mestre em sistemas distribuídos pela USP e doutorando no INRIA, em Sophia Antipolis, na França) e Danilo Sato (mestre em Metodologias Ágeis pela USP, e desenvolvedor da ThoughtWorks em Londres).

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:

`anonymous@tail.cvs.sourceforge.net:/cvsroot/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, e por isso usamos algo similar a ele como base para este treinamento 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 entidades formarão a base do projeto que criaremos durante o treinamento, o **Argentum** (do latim, dinheiro ou prata). As funcionalidades do sistema serão os seguintes:

- Converter `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 fabricar uma série temporal.

- Funcionalidades na `SerieTemporal`

Buscar candles por data, intervalos e outros

- Utilizar indicadores técnicos

Para isso implementar um pequeno framework de indicadores, e criar alguns deles, facilitando o desenvolvimento de novos.

- Gerar gráficos

Tanto dos candles, quanto dos indicadores.

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 - 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` na declaração da variável (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!  
  
}
```

Qual seria a vantagem de trabalhar assim? Veremos na próxima sessão.

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` você tem uma variável `public static final` chamada `EMPTY_LIST`. É convenção essas variáveis terem letras maiúsculas e separadas por *underscore* em vez de subir e descer (chamado *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.

2.6 - Imutabilidade de objetos

Effective Java

Item 15: Minimize mutabilidade

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

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

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

Qual seria a motivação de criar uma classe de tal maneira? Vejamos algumas.

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 alguém resolver tocar nesses valores. Essa manipulação possibilita um cache de suas partes internas, além de facilitar a manipulação desses objetos.

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

Isto acontece com a `String`, que compartilha a array privada de `char` quando métodos como `substring` são invocados!

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

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 ocorrer duas modificações concorrentes!

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.

Nossa classe `Negocio` sendo imutável simplifica 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 correndo o risco que um método que constrói o candlestick mexa nos nossos atributos (deixando ou não num estado inconsistente), além de garantir que não haverá problemas no caso de acesso concorrente ao objeto.

2.7 - Trabalhando com datas: `Date` e `Calendar`

A classe abstrata `Calendar` encapsula um instante, em milissegundos. Também provê métodos para manipulação desse instante.

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 internas 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 da Sun 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 mudam. 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.

Repare ainda que há um overload desse método que recebe `Locale` ou `Timezone` como argumento, caso você queira que essa *factory* trabalhe sem ser de acordo com os valores defaults que a JVM descobrir em relação ao seu ambiente. Um 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... 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(Calendar.HOUR, 10); // fazemos hora valer 10
c.set(Calendar.MINUTE, 30); // fazemos minuto valer 30
c.set(2005, 11, 25); // mudamos a data para o Natal, mês começa do 0
```

Outro método bastante usado é `add`, que adiciona uma certa quantidade a qualquer campo do `Calendar`. Por exemplo, para 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 `Calendar` em questão estiver num momento no tempo maior que o 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.

Por último, um dos problemas mais comuns quando lidamos com datas é verificar o intervalo entre duas datas. 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));  
}
```

Date

A classe `Date` não 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 a internacionalização e localização do sistema de datas.

Você pode pegar um `Date` de um `Calendar` e vice-versa através dos getters e setters de time:

```
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: Deprecated

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? 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 com uma solução melhor existindo.

Mas como garantir que códigos novos não usem as funcionalidades antigas e desrecomendadas? 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 este fato 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.

JodaTime

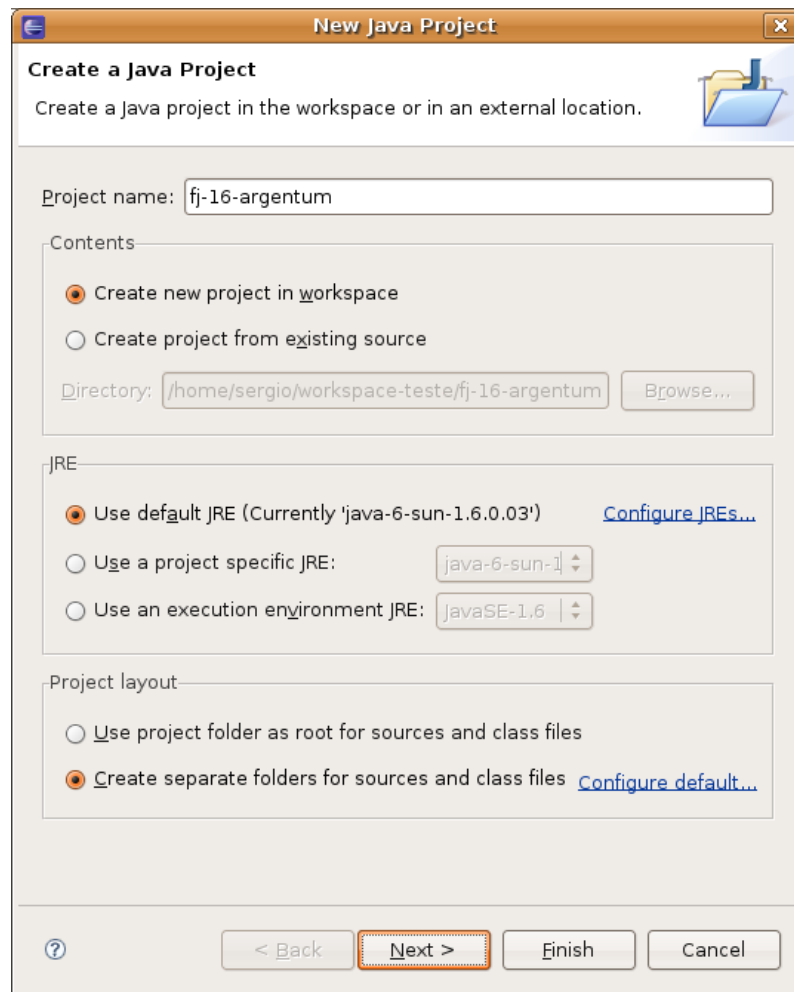
A API de datas do Java, mesmo considerando algumas melhorias da `Calendar` em relação a `Date`, ainda é muito pobre. Existe uma chance de que na versão 7 entre novas classes para facilitar o trabalho com datas e horários, baseado na excelente biblioteca **JodaTime**.

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>

2.8 - Exercícios: o modelo do Argentum

1) Vamos criar o projeto `fj-16-argentum` no Eclipse:

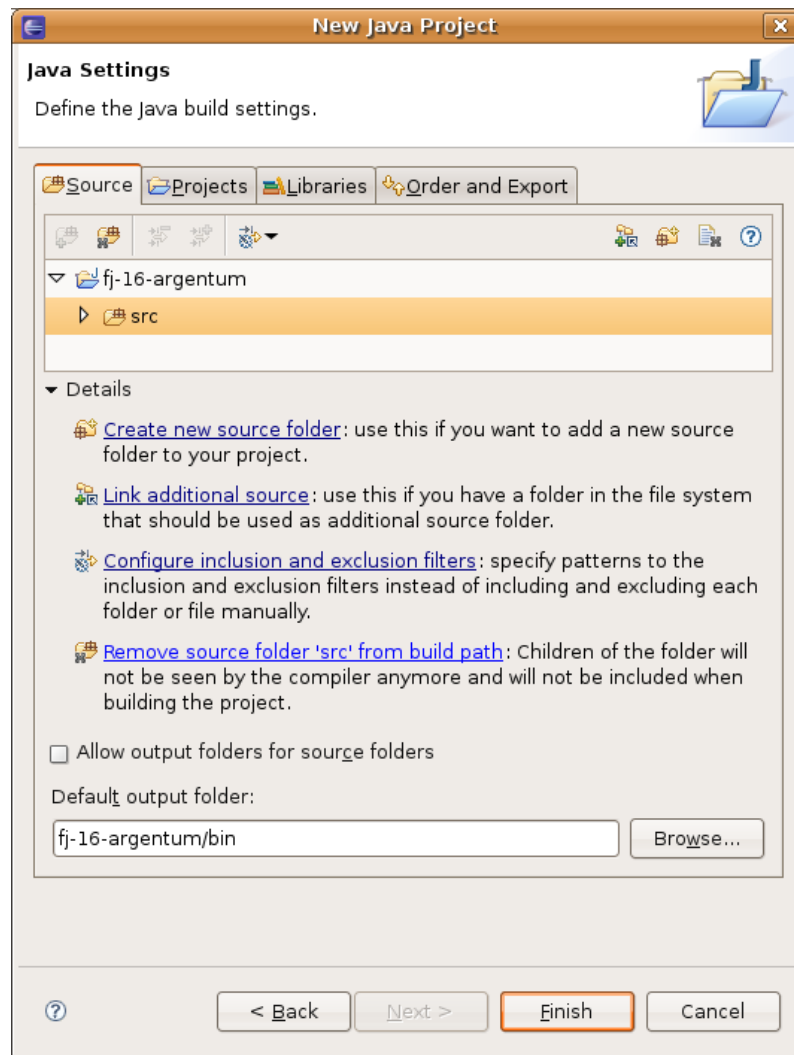
- a) Vá em **File > New > Java Project**;
- b) Preencha o nome do projeto como **fj-16-argentum** e clique em **Next**:




- c) Nesta próxima tela, podemos definir uma série de configurações do projeto (que poderiam ser feitas depois, através do menu *Properties*, clicando com o botão da direita no projeto).

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

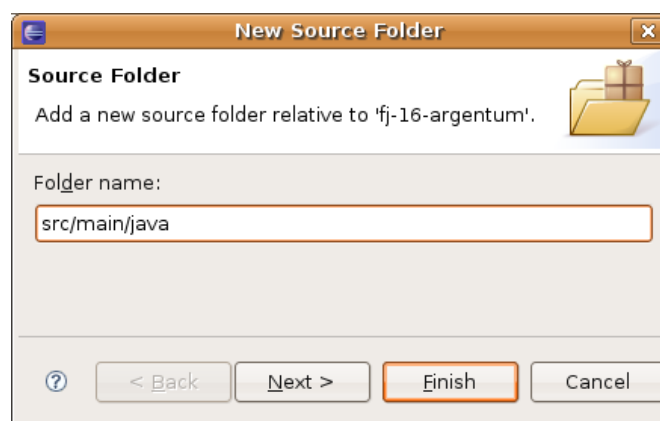
Nessa tela, remova o diretório **src** da lista de diretórios 'fonte':



- d) Agora, na mesma tela, adicione um novo diretório fonte, chamado **src/main/java**. Clique em **Create new source folder**:

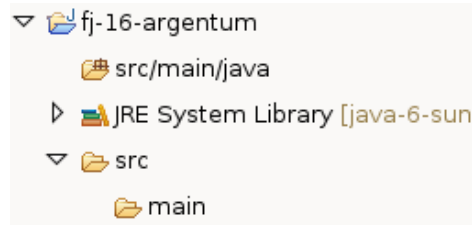
 [Create new source folder](#): use this if you want to add a new source folder to your project.

E preencha com **src/main/java**:



(Em capítulos posteriores criaremos novos diretórios.)

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

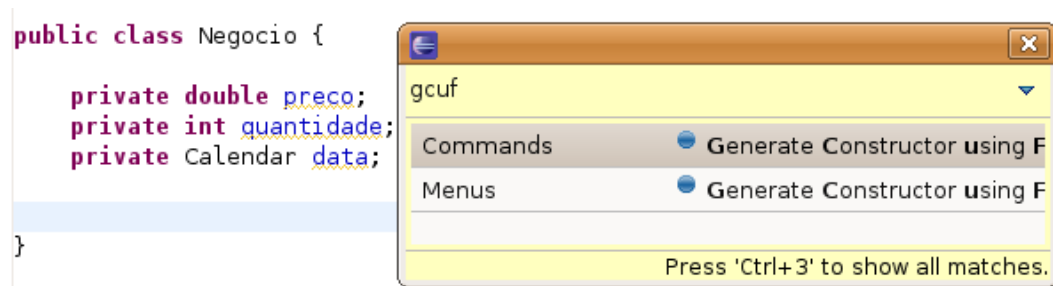


2) Crie a classe `Negocio` dentro do pacote `br.com.caelum.argentum`, declarando os três atributos que fazem parte de um `Negocio`:

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

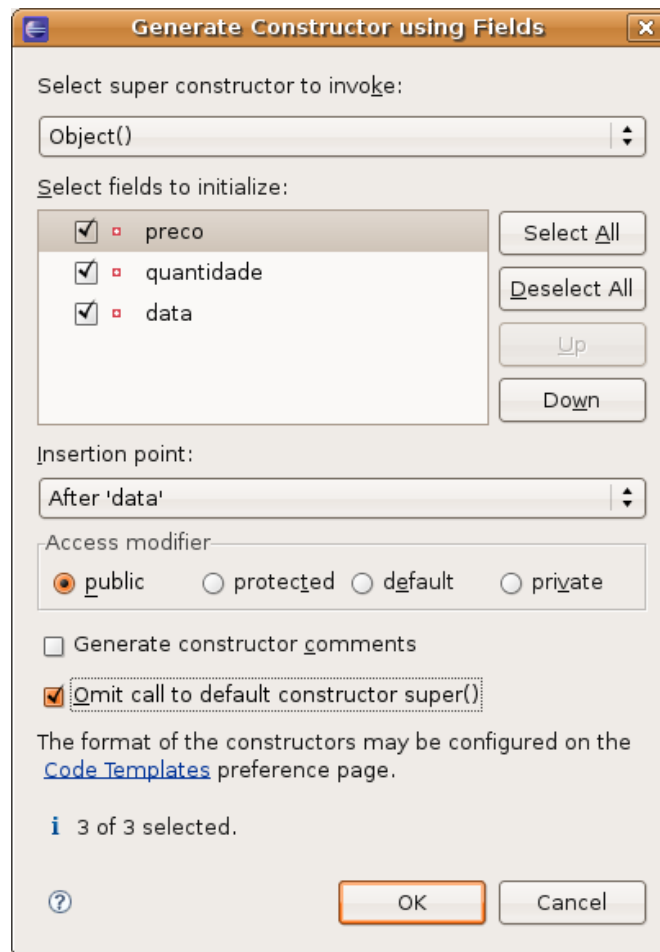
Vamos criar um construtor que recebe esses dados, já que são obrigatórios para nosso domínio. Em vez de fazer isso na mão, você pode ir ao menu **Source/Generate Constructors using Fields**.

Alternativamente, tecle *Control+3* e comece a digitar *constructor*, ele vai mostrar uma lista das alternativas que contém “constructor”. Ou melhor ainda, tecle *Control+3* e digite *gcuf*, que são as iniciais do menu que queremos acessar.



Agora, selecione tantos os campos, e marque, opcionalmente, 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.



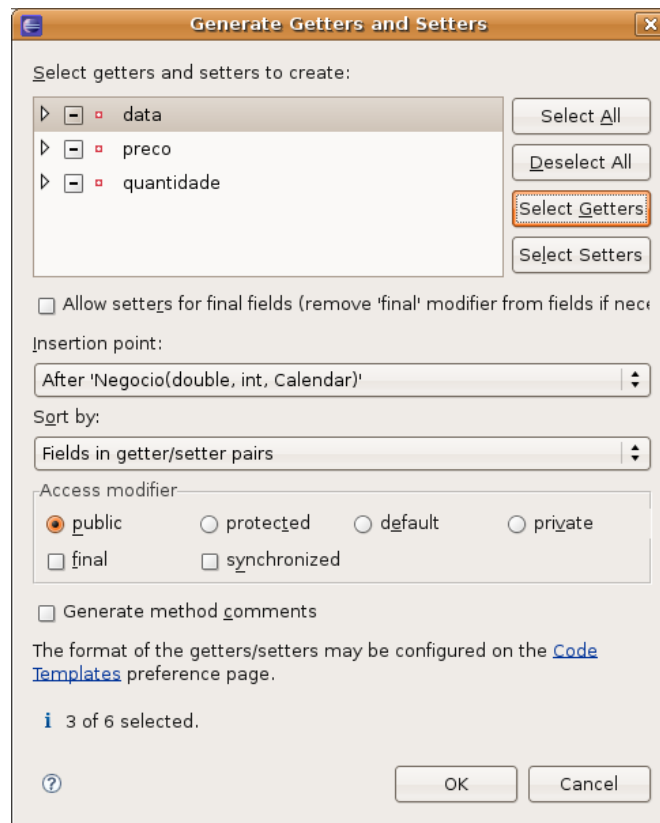
Pronto! Mande gerar. Veja o código que deve ser gerado:

```
// isso será gerado pelo Eclipse

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

- 3) Agora, vamos gerar os getters dessa classe. Você pode ir ao menu *Source/Generate Getters and Setters* ou tecla *Control+3* e digite *ggas*. Novamente, como alternativa, digite *Control+3* e comece a digitar *getter*, as opções aparecerão e basta você escolher gerar *getters*. É sempre bom praticar os atalhos do *Control+3*.

Na tela que abrir, clique em **Select getters** e depois **Finish**:



Declare todos os nossos três atributos como `final`, assim como a própria classe. Ao término desses passos, seu código deve estar parecido com o que segue:

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

import java.util.Calendar;

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;
    }
}
```

- 4) Um dado importante na bolsa de valores é o volume de dinheiro negociado.

Vamos fazer nossa classe `Negocio` devolver o volume de dinheiro daquele `Negocio`. 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`:

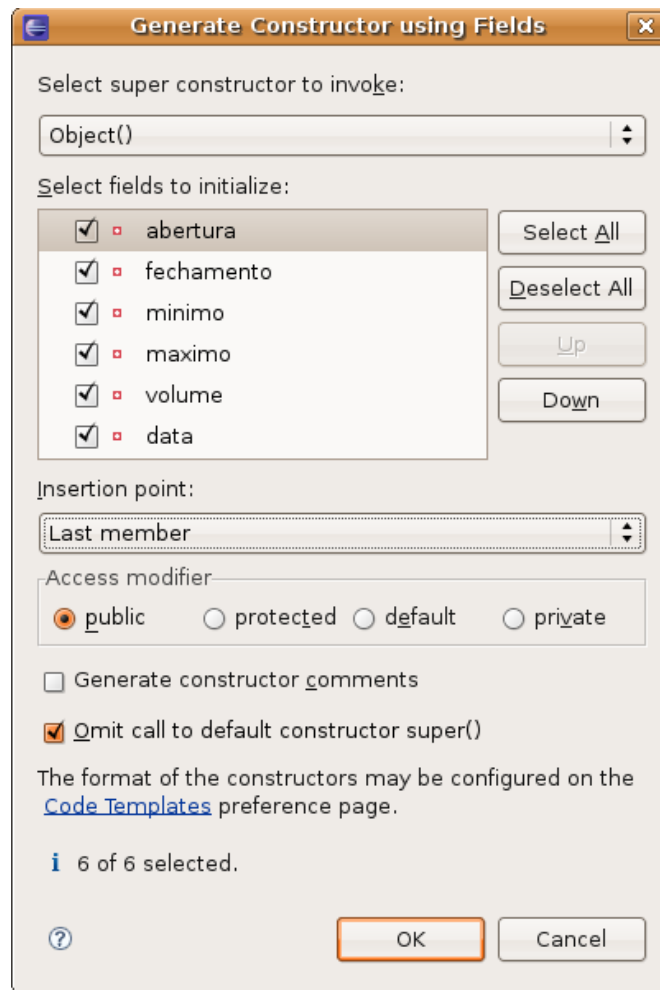
```
public double getVolume() {
    return preco * quantidade;
}
```

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

- 5) Siga o mesmo procedimento para criar a classe `Candlestick`. Ela deve possuir os seguintes atributos, nessa ordem:

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

Gere o construtor com os seis atributos. **Ctrl+3** e **gcuf**. Atenção à ordem dos parâmetros no construtor:



Gere também os seis respectivos getters. **Ctrl+3** e **ggas**. Selecione apenas os getters. Agora, adicione o modificador `final` em cada atributo e também na classe. A classe deve ficar parecida com a que segue:

```
1 package br.com.caelum.argentum;
2
3 import java.util.Calendar;
4
5 public final class Candlestick {
6     private final double abertura;
7     private final double fechamento;
8     private final double minimo;
9     private final double maximo;
10    private final double volume;
11    private final Calendar data;
12
13    public Candlestick(double abertura, double fechamento, double minimo,
14        double maximo, double volume, Calendar data) {
15        this.abertura = abertura;
16        this.fechamento = fechamento;
17        this.minimo = minimo;
18        this.maximo = maximo;
19        this.volume = volume;
20        this.data = data;
```

```
21     }
22
23     public double getAbertura() {
24         return abertura;
25     }
26
27     public double getFechamento() {
28         return fechamento;
29     }
30
31     public double getMinimo() {
32         return minimo;
33     }
34
35     public double getMaximo() {
36         return maximo;
37     }
38
39     public double getVolume() {
40         return volume;
41     }
42
43     public Calendar getData() {
44         return data;
45     }
46 }
```

- 6) 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;
}
```

2.9 - Exercícios: fábrica de Candlestick

- 1) Vamos criar a classe responsável por pegar uma lista de `Negocio` e criar um `Candlestick` baseado nesses dados e em uma data.

Calculamos o preço máximo e mínimo varrendo todas os negócios e achando-os. Para isso usaremos variáveis temporárias, e dentro do `for` verificamos se o preço do negócio atual bate ou máximo, se não bater, vemos se ele é menor que o mínimo.

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

```
}
```

Calculamos o volume somando o volume de cada negócio em uma variável:

```
volume += negocio.getVolume();
```

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

Como a construção de um candle 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` parecida com:

```
1 public class CandlestickFactory {
2     public Candlestick constroiCandleParaData(Calendar data, List<Negocio> negocios) {
3
4         double maximo = negocios.get(0).getPreco();
5         double minimo = negocios.get(0).getPreco();
6         double volume = 0;
7
8         // digite foreach<control_espaco> para ajudar aqui!
9         for (Negocio negocio : negocios) {
10             volume += negocio.getVolume();
11
12             if (negocio.getPreco() > maximo) {
13                 maximo = negocio.getPreco();
14             } else if (negocio.getPreco() < minimo) {
15                 minimo = negocio.getPreco();
16             }
17         }
18
19         double abertura = negocios.get(0).getPreco();
20         double fechamento = negocios.get(negocios.size() - 1).getPreco();
21
22         return new Candlestick(abertura, fechamento, minimo, maximo, volume, data);
23     }
24 }
```

2) Vamos testar nosso código, criando 4 negócios e calculando o `Candlestick`, finalmente.

```
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, negocio3, negocio4);
12     }
```

```
13      CandlestickFactory fabrica = new CandlestickFactory();
14      Candlestick candle = fabrica.constroiCandleParaData(hoje, negocios);
15
16      System.out.println(candle.getAbertura());
17      System.out.println(candle.getFechamento());
18      System.out.println(candle.getMinimo());
19      System.out.println(candle.getMaximo());
20      System.out.println(candle.getVolume());
21  }
22 }
```

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
```

2.10 - Exercícios opcionais

Effective Java

1) Item 10: Sempre reescreva o `toString`

Reescreva o `toString` da classe `Negocio` e da `Candlestick`. Faça com que a data seja corretamente visualizada, usando para isso o método `data.get(Calendar.DATE)` e as outras constantes.

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 `Negocio`, fora de qualquer método, pressionar `ctrl+espaço`. Aparecerá uma lista com todas as opções de métodos que você pode reescrever. Escolha o `toString`, e ao pressionar `enter` o esqueleto da reescrita será montado.

2) 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.

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

A idéia é que a gente possa criar um *candle* da seguinte maneira:

```
CandleBuilder builder = new CandleBuilder();

builder.abertura(40.5);
builder.fechamento(42.3);
builder.minimo(39.8);
builder.maximo(45);
builder.volume(145234.20);
builder.data(new GregorianCalendar(2008, 8, 12, 0, 0, 0));

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

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

```
Candlestick candle = new CandleBuilder().abertura(40.5)

    .fechamento(42.3).minimo(39.8).maximo(45)
    .volume(145234.20).data(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 abertura(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 te ajudar a construir `Strings` através de *fluent interface* e métodos auxiliares.

- 3) 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.

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 retornar um `Candlestick` que possui um significado especial. Retornar `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 enfiar um `if` logo de cara para verificar se `negocios.isEmpty()`, porém podemos tentar algo mais sutil, sem ficar tendo de 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`. Mesma coisa 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 de petrobrás a 30 reais, com uma quantidade negativa de negócios é válido? E com número zero de negócios?

Em outras palavras, posso dar `new` num `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 unitários

Testes unitários são testes que testam apenas uma classe ou método, verificando se seu comportamento está de acordo com o desejado. Em testes unitários, 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 unitários, 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 unitário, 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 acabam testando 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 sistema como um todo).

Portanto, lembre-se sempre: testes unitários testam apenas unidades!

3.5 - JUnit

O **JUnit** (junit.org) é um framework muito simples para facilitar a criação destes testes unitários 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.

A medida que você mexe no seu código, você roda novamente aquela bateria de testes. Com isso você ganha a confiança de que novos bugs não estão sendo introduzidos (ou reintroduzidos) a medida que você cria novos recursos e conserta antigos bugs. Diferentemente do que ocorreu quando fizemos os testes todos dentro do `main`, executando cada vez um.

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, recurso introduzido no Java 5.0. Algumas anotações podem ser retidas (retained) no `.class`, fazendo com que, por *reflection*, nós possamos descobrir essas informações.

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 - JUnit 4 e suas anotações

Para cada classe, teremos uma classe correspondente (normalmente com o sufixo `Test`) que contará todos os testes relativos aos métodos dessa classe.

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

```
public class CandlestickFactoryTest {

    public void testSimplesSequenciaDeNegocios() {
        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 eloquente 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 testSimplesSequenciaDeNegocios() {  
        // ...  
    }  
}
```

Pronto! Quando rodarmos essa classe como sendo um teste do junit, esse método será executado e verificado se tudo ocorreu bem. Tudo ocorre bem quando o método é executado sem lançar nenhuma exceção inesperada e se todas as **asserções** passarem.

Uma asserção é realizada através dos métodos estáticos da classe `Assert`. 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*, o valor que esperamos que o segundo argumento valha (chamado de *actual*). Se não for, uma exceção será lançada, e o teste considerado uma falha.

Isso substitui elegantemente um possível `if` que verificaria isto manualmente, para então lançar uma exceção. É um modo de automatizar tudo: além de rodar todos os nossos testes de uma maneira uniforme através de um simples clique, não precisaremos ficar olhando (literalmente) para a saída do console a fim de verificar se a saída está de acordo com o esperado.

O terceiro argumento 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. Isso porque estamos tratando com valores `double` e o erro de arredondamento é comum. Dessa forma, precisamos informar também o quanto de erro é aceitável na nossa aplicação.

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 após os exercícios:

```
1 public class CandlestickFactoryTest {  
2  
3     @Test  
4     public void testSimplesSequenciaDeNegocios() {  
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, negocio4);  
13  
14        CandlestickFactory fabrica = new CandlestickFactory();  
15        Candlestick candle = fabrica.constroiCandleParaData(hoje, negocios);  
16  
17        Assert.assertEquals(40.5, candle.getAbertura(), 0.00001);  
18    }  
19 }
```



```
18 Assert.assertEquals(42.3, candle.getFechamento(), 0.00001);
19 Assert.assertEquals(39.8, candle.getMinimo(), 0.00001);
20 Assert.assertEquals(45.0, candle.getMaximo(), 0.00001);
21 Assert.assertEquals(1676.0, candle.getVolume(), 0.00001);
22 }
23 }
```

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

3.8 - Test Driven Development - TDD

TDD É uma técnica que consiste de pequenas iterações, onde novos casos de testes de funcionalidades desejadas são criados antes mesmo do início da implementação. Nesse momento, o teste escrito deve falhar, já que não existe a funcionalidade implementada. Então, o código necessário para que os testes passem deve ser escrito e o teste deve passar.

O benefício dessa técnica consiste no fato de que, como os testes são escritos antes da implementação do trecho a ser testado, o programador não é influenciado pelo código já feito, bem como escreve testes melhores, pensando no comportamento ao invés da implementação; até porque, o cliente sabe o comportamento que espera do sistema, e não a implementação correta.

Além disso, TDD traz baixo acoplamento, o que é ótimo pois classes muito acopladas são difíceis de testar e, como criaremos os testes antes, desenvolveremos classes menos acopladas, separando melhor as responsabilidades.

O TDD também é uma maneira de te guiar: 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.

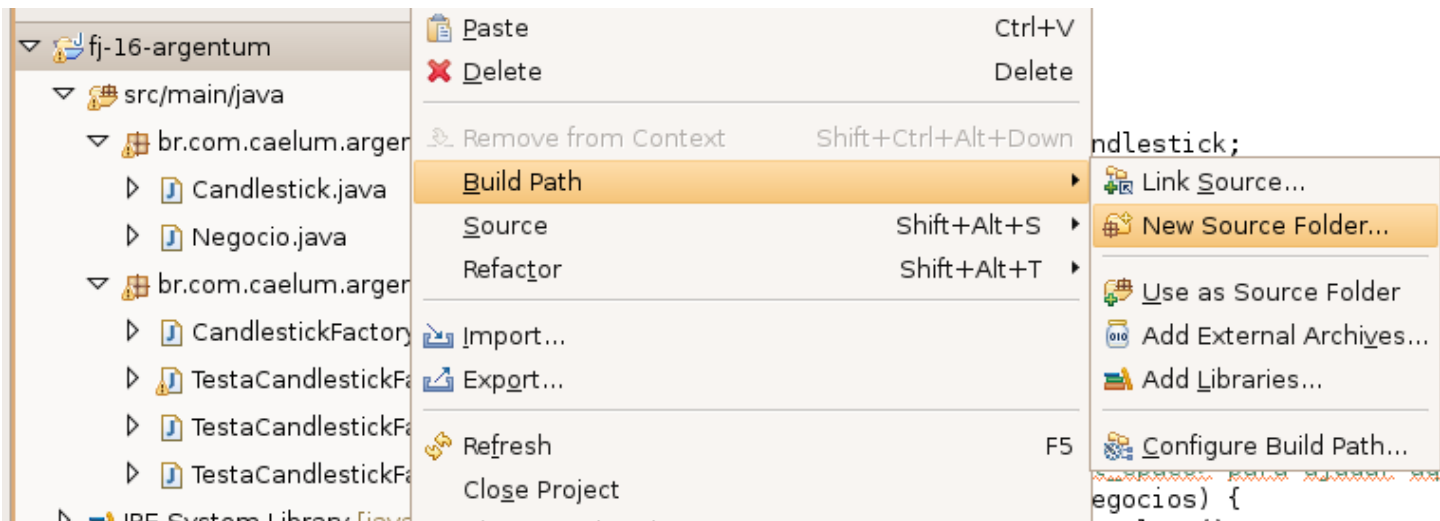
Imagine que você já tenha um sistema com muitas classes e nenhum teste, provavelmente para iniciar a criação de testes muitas refatorações terão de ser feitas, mas como modificar seu sistema garantindo o funcionamento do mesmo após as mudanças se não existem testes que garantam que seu sistema tenha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

TDD é algo difícil de se implementar, mas depois que você constroi um sistema dessa maneira, o hábito é adquirido e você vê claramente as diversas vantagens dessa técnica.

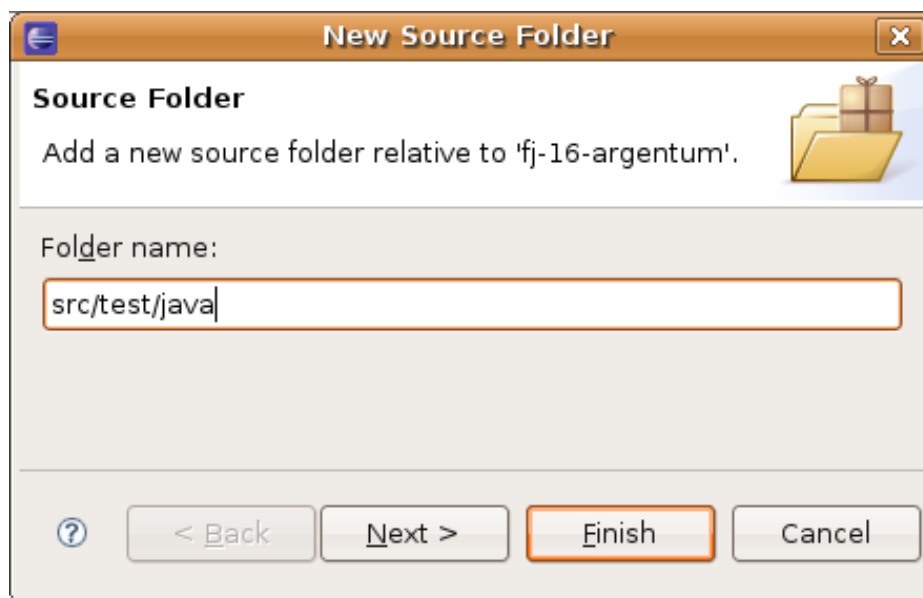
3.9 - Exercícios: migrando os testes do main para JUnit

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

a) Clique da direita no nome do projeto e vá em **Build path, New source folder**:



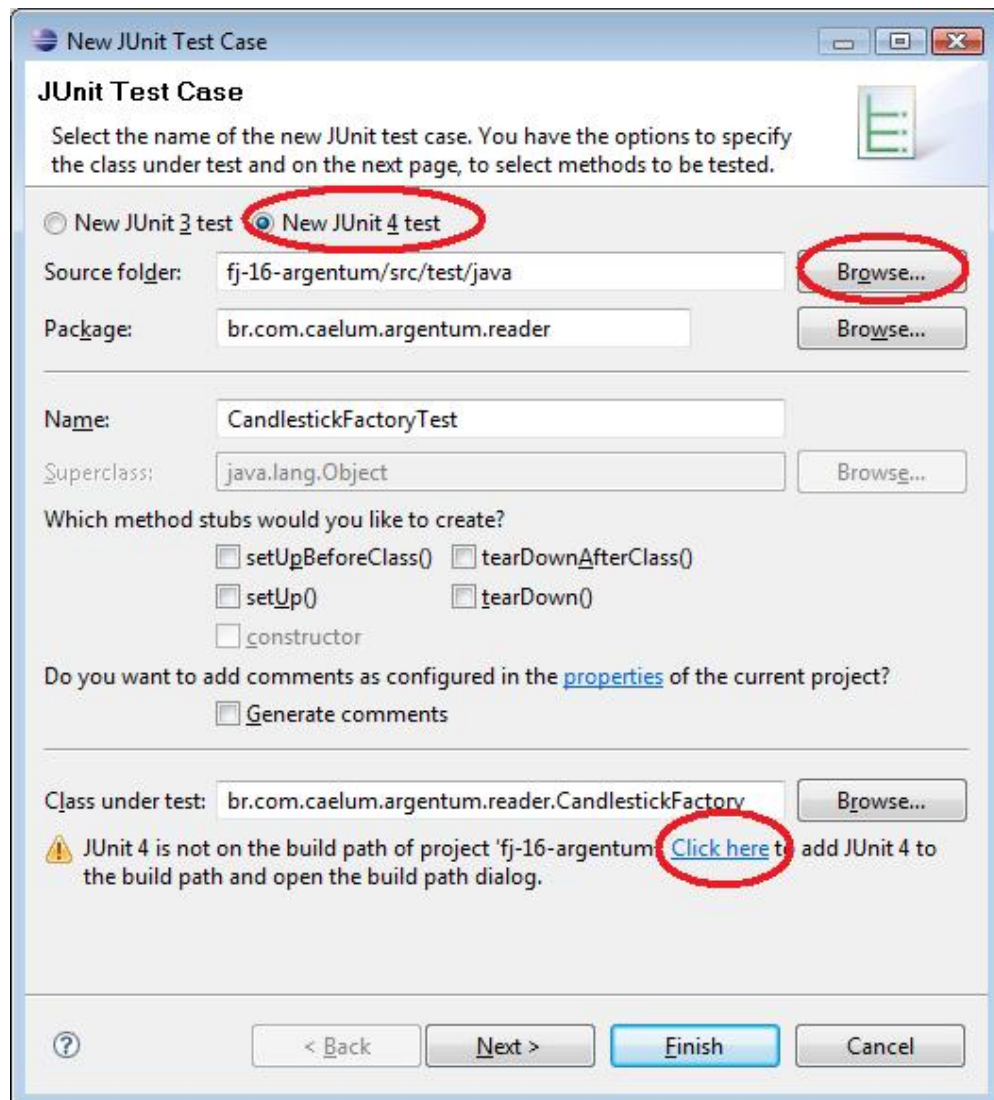
b) Preencha com **src/test/java** e clique **Finish**:



Nesse novo diretório que você colocará todos seus testes unitários.

- 2) Vamos criar um novo *unit test* em cima da classe *CandlestickFactory*. O Eclipse já te ajuda bastante: **dê um clique da direita nessa classe**, e vá em **New > JUnit Test Case**, selecione o *source folder* como **src/test/java**.

Nesta mesma tela, **não esqueça de selecionar JUnit4**. Abaixo do wizard poderá haver um warning indicando que o JUnit não está no seu path, aceite a sugestão do eclipse para **adicionar o JUnit 4 ao seu build path**. Caso não apareça o warning, como acontece em versões mais novas no Eclipse, isto será sugerido após o clique em *finish*.



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 primeiramente o teste inicial:

```
1 public class CandlestickFactoryTest {  
2  
3     @Test  
4     public void testSimplesSequenciaDeNegocios() {  
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, negocio4);  
13  
14        CandlestickFactory fabrica = new CandlestickFactory();
```

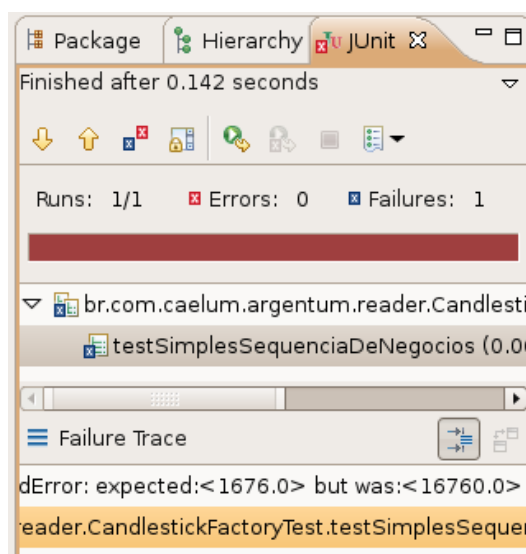
```

15      Candlestick candle = fabrica.constroiCandleParaData(hoje, negocios);
16
17      Assert.assertEquals(40.5, candle.getAbertura(), 0.00001);
18      Assert.assertEquals(42.3, candle.getFechamento(), 0.00001);
19      Assert.assertEquals(39.8, candle.getMinimo(), 0.00001);
20      Assert.assertEquals(45.0, candle.getMaximo(), 0.00001);
21      Assert.assertEquals(1676.0, candle.getVolume(), 0.00001);
22  }
23  }
  
```

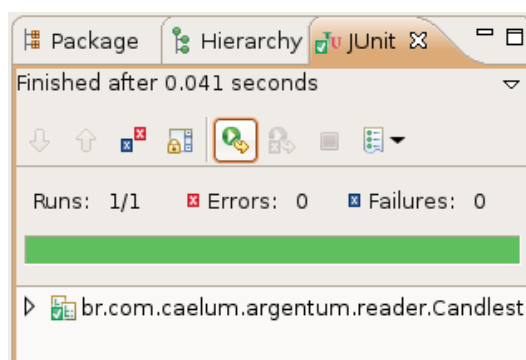
Para rodar, clique da direita no .java do seu teste, **Run As..., JUnit Test::**.



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 ele falhou:



O número correto é mesmo **16760.0**. Adicione esse zero na classe de teste e rode-o novamente (você pode clicar no botão play na view do JUnit para rodar novamente):



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

- 3) Vamos **adicionar** outro método de teste na mesma classe `CandlestickFactoryTest`, dessa vez para testar o método no caso de não haver nenhum negócio:

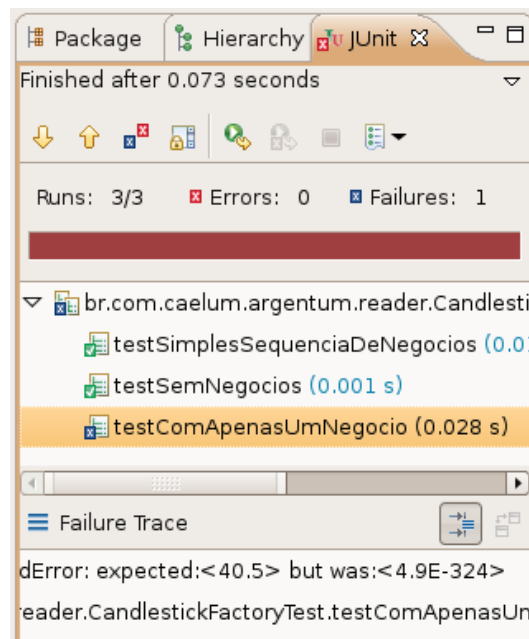
```
1 @Test
2 public void testSemNegocios() {
3     Calendar hoje = Calendar.getInstance();
4
5     List<Negocio> negocios = Arrays.asList();
6
7     CandlestickFactory fabrica = new CandlestickFactory();
8     Candlestick candle = fabrica.constroiCandleParaData(hoje, negocios);
9
10    Assert.assertEquals(0.0,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

Rode o teste (use o botão de 'play' da barra de ferramentas para rodar novamente o último teste).

- 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 testComApenasUmNegocio() {
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?

3.10 - Exercícios: novos testes

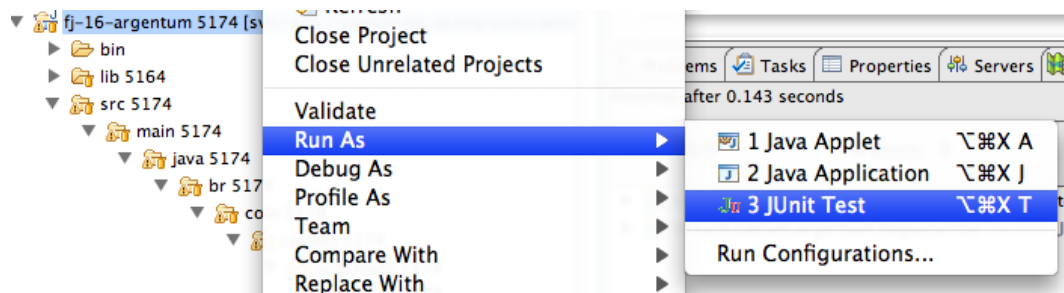
1) A classe `Negocio` é realmente imutável?

Vamos criar um novo *Unit Test* em cima da classe `Negocio`. Clique com o botão direito no nome da classe `Negocio` e vá em **New > JUnit Test Case**. **Lembre-se de colocá-lo no diretório `src/test/java` e selecionar o JUnit 4.**

```
1 public class NegocioTest {
2
3     @Test
4     public void testDataDoNegocioEhImutavel() {
5         Calendar c = Calendar.getInstance();
6         c.set(Calendar.DAY_OF_MONTH, 15);
7
8         Negocio n = new Negocio(10, 5, c);
9
10        // essa mudança não deveria ficar visível, vamos testar:
11        n.getData().set(Calendar.DAY_OF_MONTH, 20);
12
13        Assert.assertEquals(15, n.getData().get(Calendar.DAY_OF_MONTH));
14    }
15 }
```

Você pode rodar esse teste em particular clicando com o botão da direita nele, ou pode fazer melhor e o que é mais comum, rodar **todos** os unit testes de um projeto.

Basta clicar da direita no projeto, Run as, JUnit Test:



Esse teste falha porque devolvemos um objeto mutável através de um *getter*. Deveríamos ter retornado uma cópia desse objeto.

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();  
}
```

Outra maneira

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 era melhor que não. Podemos lançar uma exceção já, para isso não ocorrer depois. Vamos criar uma própria exceção? Já existe a `IllegalArgumentException`, vamos usá-la para isso.

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

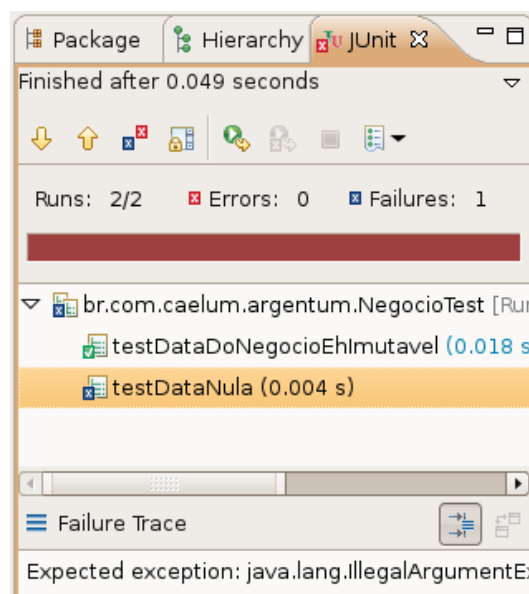
Antes de fazer a modificação na classe, vamos preparar o teste. Mas o que queremos testar? Queremos saber se nossa classe `Negocio` lança uma `IllegalArgumentException` quando passamos `null` no construtor. Ou seja, *esperamos* que uma exceção aconteça! Para o teste *passar*, ele precisa dar a exceção (parece meio contraditório). É fácil fazer isso com JUnit.

Adicione um novo método `testNegocioComDataNula` 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 testNegocioComDataNula() {
    Negocio n = new Negocio(10, 5, null);
}
```

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

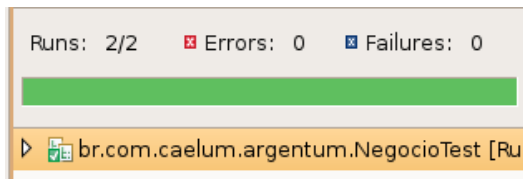


Vamos **alterar** a classe `Negocio`, para que ela lance a exceção no caso de uma data nula. No construtor, adicione o seguinte `if`:

```
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) 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 `testPrecoMaximoNaoPodeSerMenorQueMinimo` e faça um `new` passando argumentos que quebrem isso. O teste deve esperar pela `IllegalArgumentException`.

A idéia 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 de algo assim gera exception (e deveria gerar):

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

- 4) (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.

- 5) (opcional) Crie mais dois testes na `CandlestickFactoryTest`: o `testNegociosEmOrdemCrescenteDeValor` e `testNegociosEmOrdemDecrescenteDeValor`, 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(2 + 2, 4);  
    }  
}
```

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

3.12 - Mais exercícios opcionais

- 1) Um `Candlestick` com preço de abertura igual ao de fechamento deve ser considerado de alta.
Crie o teste `testSeEhAltaQuandoAberturaIgualFechamento` 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!
- 2) 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 `testGeracaoDeCandleSemTodosOsDadosNecessarios` 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.
- 3) Teste o `toString` do seu `Candlestick`. Ele está correto? Você lembrou de subtrair 1 na hora de imprimir o número do mês? Você imprime sempre com dois dígitos? Imprime quatro ou dois números do ano? Os números de ponto flutuante vão ser impressos de acordo com o `Locale` correto (e no Brasil então sairia com vírgula) ou sempre com '.'?
Repare que talvez algumas dessas perguntas nem mesmo você saiba responder, pois não havia decidido o comportamento exato daquele método. Os testes unitários te forçam a pensar mais sobre o que cada método vai fazer, e parar para refletir nos casos especiais (*corner cases*).
Tome as decisões e faça testes que reflitam elas. Corrija seu código de acordo com testes que falharem.
- 4) 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?

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>555454646</time>
    </data>
  </negocio>
  <negocio>
    <preco>44.1</preco>
    <quantidade>700</quantidade>
    <data>
      <time>555454646</time>
    </data>
  </negocio>
  <negocio>
    <preco>42.3</preco>
    <quantidade>1200</quantidade>
    <data>
      <time>559329406</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:

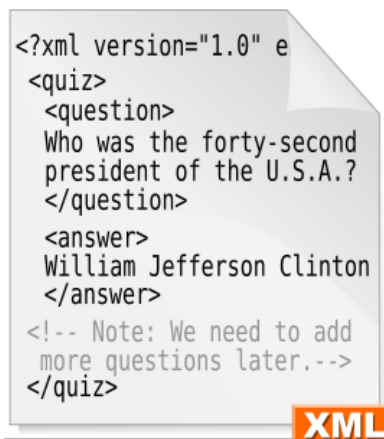
<http://www.w3.org/XML/>

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 idéia 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:



```
<?xml version="1.0" e
<quiz>
  <question>
    Who was the forty-second
    president of the U.S.A.?
  </question>
  <answer>
    William Jefferson Clinton
  </answer>
  <!-- Note: We need to add
    more questions later.-->
</quiz>
```

XML

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

```
<?xml version="1.0" encoding="ISO-859-1" ?>
```

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 JavaSE 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 seis 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 ao bean. Podemos simplesmente criar um e populá-lo, ou este pode ser, por exemplo, uma entidade do Hibernate.

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` vai ser 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 um overload 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 da Sun, e a do XStream é ser mais flexível e deixar trabalhar com classes imutáveis.

@XstreamAlias

Ao invés de chamar `stream.alias('negocio', Negocio.class);`, podemos fazer essa configuração direto na classe `Negocio` com uma anotação:

```
@XstreamAlias("negocio")
```

```
public class Negocio {  
}
```

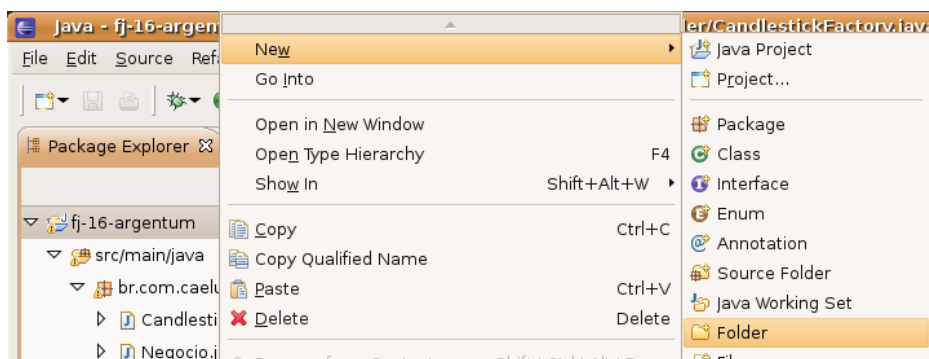
Para habilitar o suporte a anotações, precisamos chamar no `xstream`:

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

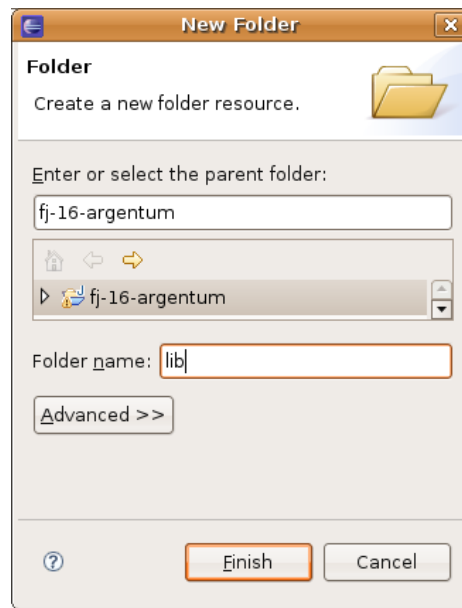
4.5 - Exercícios: Lendo o XML

- 1) Para usarmos o XStream, precisamos copiar seus JARs para o nosso projeto e adicioná-los no Build Path. Para facilitar, vamos criar uma pasta **lib** para colocar todos os arquivos JAR que necessitarmos.

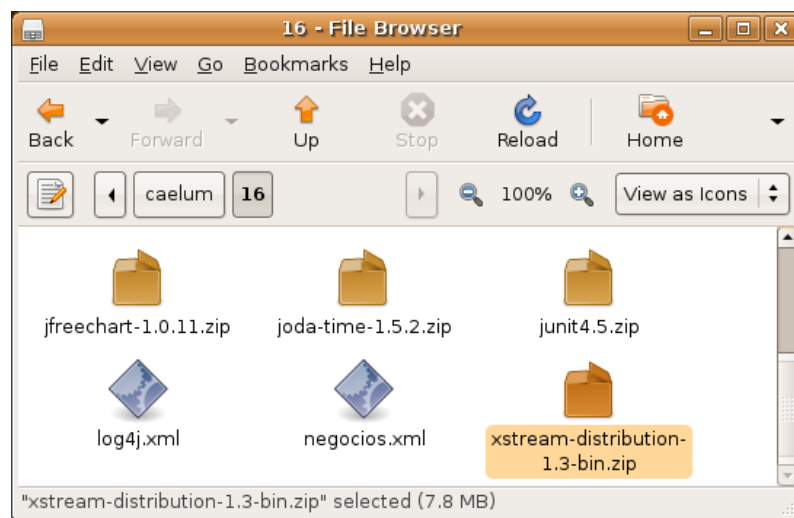
Clique com o botão direito no nome do projeto e vá em **New > Folder**:



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

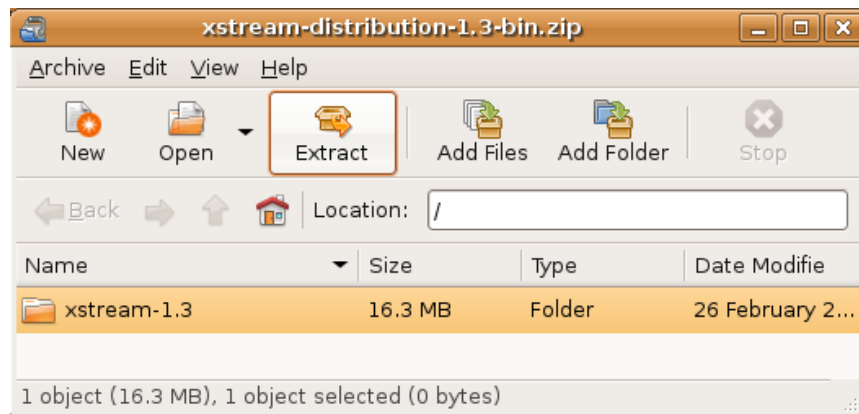


- 2) Vamos instalar o XStream no nosso projeto. Vá na pasta **Caelum** no seu Desktop e entre em **16**. Localize o arquivo do xstream:

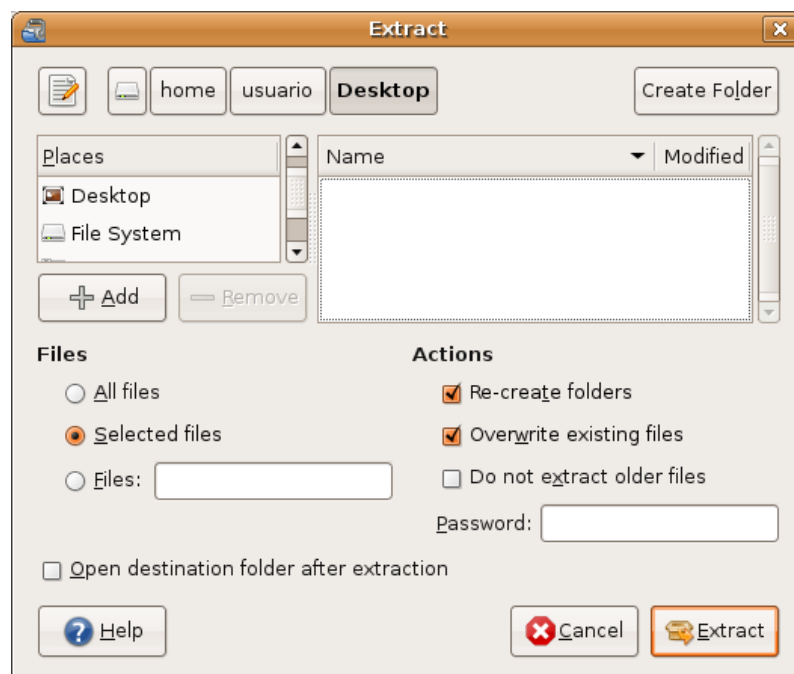


Esse é o mesmo arquivo que é baixado do site do XStream.

Copie esse arquivo para o seu Desktop. Dê dois cliques para abrir o descompactador do Linux. Na próxima tela, clique em Extract:



Com o próprio Desktop selecionado, clique Extract:



- 3) Entre na pasta **xstream-1.3/lib** no seu Desktop. Há vários JARs do XStream lá, vamos precisar apenas do **xstream-1.3.jar**. O restante são dependências que não precisamos. Copie esse JAR para dentro da sua pasta **lib** do projeto.

Depois, pelo Eclipse, entre na pasta **lib**, de refresh nela, 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.

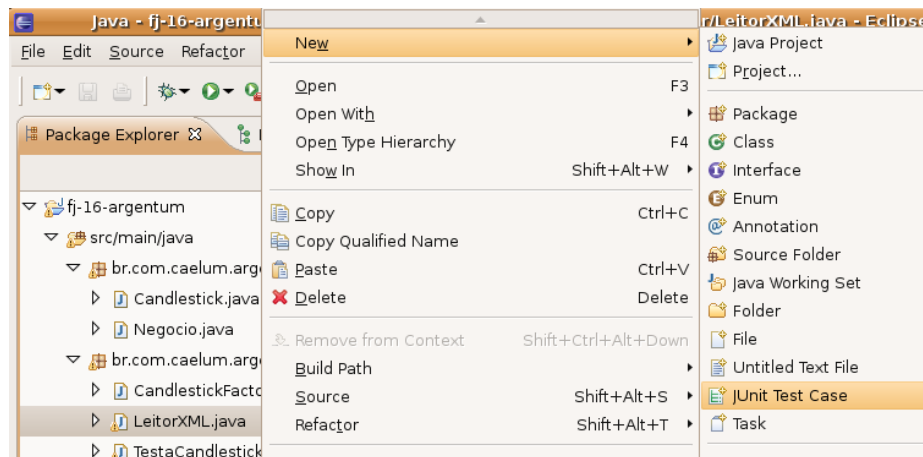
- 4) Vamos agora, finalmente, implementar a leitura do XML, delegando o trabalho para o XStream. Criamos a classe `LeitorXML` dentro do pacote `br.com.caelum.argentum.reader`:

```
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
13 }
```

- 5) Crie um teste unitário `LeitorXMLTest` pelo Eclipse para testarmos a leitura do XML. Basta clicar com botão direito na classe `LeitorXML` e selecionar **New > JUnit Test Case**:



Lembre-se de colocá-lo no diretório **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 testLeitorDeXmlCarregaListaDeNegocio() {
    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 as reticências no código acima pelo texto abaixo*:

```

<list>
  <negocio>
    <preco>43.5</preco>
    <quantidade>1000</quantidade>
    <data>
      <time>555454646</time>
    </data>
  </negocio>
</list>

```

Faça algumas asserções:

- a lista devolvida deve ter tamanho 1

- o negócio deve ter preço 43.5
- a quantidade deve ser 1000

Opcionalmente crie mais de um teste se possível, testando casos excepcionais, como XML inválido, zero negócios, e dados faltando.

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

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 quase um teste de integração, e não de unidade. Se criássemos esse teste, e ele falhasse, ficaria muito difícil de detectar o ponto de falha! Mais ainda: pode parecer difícil de perceber, mas já que testamos todas as peças (units) do nosso motor (aplicação), é muito provável que ele funcione quando rodado! Só falta testar as “ligas”, que normalmente são bem leves e poucas (o tal do baixo acoplamento).

Pensar em sempre testar as menores unidades possíveis força que a gente sempre crie programas com baixo acoplamento: poderíamos ter criado um `LeitorXML` que devolvesse diretamente uma `List<Candlestick>`, já chamando a fábrica. Mas isso faria com que o nosso teste do leitor de XML testasse muito mais que apenas a leitura de XML (pois estaria passando pela nossa `CandlestickFactory`).

4.6 - 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) Vamos então, em nossa classe de factory, pegar uma série de negócios e transformar em uma lista de candles. Para isso vamos precisar separar os negócios que são da mesma data.

Para saber se ainda estamos percorrendo negócios da mesma data, vamos usar um método auxiliar que verifica se o dia, mês e ano são os mesmos.

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

```
@Test
public void testComparaMesmoDiaCalendar() {
    CandlestickFactory factory = new CandlestickFactory();
    Calendar data1 = Calendar.getInstance();
    Calendar data2 = Calendar.getInstance();

    Assert.assertTrue(factory.isMesmoDia(data1, data2));
}
```

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**.

```
CandlestickFactory fabrica = new CandlestickFactory();
List<Candlestick> candles = fabrica.construirCandles(negocios);
...
// ... e1.get(Calendar.YEAR) == e2.get(Calendar.YEAR) ...
Create method 'construirCandles(List<Negocio>)' ir
```

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

```
public boolean isMesmoDia(Calendar data1, Calendar data2) {
```

```
        return data1.equals(data2);  
    }  
}
```

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 `CandlestickFactoryTest`:

```
@Test  
  
public void testComparaMesmoDiaHorasDiferentes() {  
    CandlestickFactory factory = new CandlestickFactory();  
  
    // usando GregorianCalendar(year, month, day, hour, minute)  
    Calendar data1 = new GregorianCalendar(2008, 12, 25, 8, 30);  
    Calendar data2 = new GregorianCalendar(2008, 12, 25, 10, 30);  
  
    Assert.assertTrue(factory.isMesmoDia(data1, data2));  
}
```

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 correta que compara dia, mês e ano seria:

```
public boolean isMesmoDia(Calendar data1, Calendar data2) {  
  
    return data1.get(Calendar.DAY_OF_MONTH) == data2.get(Calendar.DAY_OF_MONTH)  
        && data1.get(Calendar.MONTH) == data2.get(Calendar.MONTH)  
        && data1.get(Calendar.YEAR) == data2.get(Calendar.YEAR);  
}
```

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

- 3) Próximo passo: dada uma lista de Negocios de várias datas diferentes mas ordenada por data, quebrar em uma lista de `Candlesticks`, um para cada data.

TDD: começamos pelo teste! **Adicione o método** `testConstruirCandlesDeMuitosNegocios` na classe `CandlestickFactoryTest`:

```
1 @Test  
  
2 public void testConstruirCandlesDeMuitosNegocios() {  
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
3     List<Candlestick> candles = new ArrayList<Candlestick>();
4
5     // lista com negocios que sejam do mesmo dia que dataPrimeiro
6     List<Negocio> negociosMesmoDia = new ArrayList<Negocio>();
7     Calendar dataPrimeiro = todosNegocios.get(0).getData();
8
9     for (Negocio negocio : todosNegocios) {
10         // se não for mesmo dia, fecha candle e reinicia variáveis
11         if (!isMesmoDia(dataPrimeiro, negocio.getData())) {
12             candles.add(constroiCandleParaData(dataPrimeiro, negociosMesmoDia));
13
14             negociosMesmoDia = new ArrayList<Negocio>();
15             dataPrimeiro = negocio.getData();
16         }
17         negociosMesmoDia.add(negocio);
18     }
19
20     // adiciona último candle
21     candles.add(constroiCandleParaData(dataPrimeiro, negociosMesmoDia));
22 }
```



```
22
23     return candles;
24 }
```

Rode o teste!

4.7 - 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`.

Pra isso modificamos o código da seguinte forma:

```
for (Negocio negocio : todosNegocios) {

    if (dataPrimeiro.after(negocio.getData())) {
        throw new IllegalStateException("negocios em ordem errada");
    }
    ...
}
```

E criamos um teste que espera essa exceção ser lançada. Crie o `testConstruirCandlesComNegociosForaDeOrdem` e faça os devidos asserts. Basta usar o mesmo teste que tínhamos antes, porém dessa vez adicionar os negócios com datas diferentes e não crescentes.

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

```
1 public class GeradorXMLAleatorio {
2
3     public static void main(String[] args) {
4
5         Calendar data = Calendar.getInstance();
6         Random random = new Random(123);
7         List<Negocio> negocios = new ArrayList<Negocio>();
8
9         double valor = 40;
10        int quantidade = 1000;
11
12        // 30 dias:
13        for (int i = 0; i < 30; i++) {
14
15            // cada dia com 0 a 19 negocios dentro
16            for (int x = 0; x < random.nextInt(20); x++) {
17                // sobe 1,00 ou cai 1,00
18                valor += (random.nextInt(200) - 100) / 100.0;
19
20                // sobe 100 ou desce 100
21                quantidade += (random.nextInt(200) - 100);
```

```
22
23         Negocio n = new Negocio(valor, quantidade, data);
24         negocios.add(n);
25     }
26
27     data = (Calendar) data.clone();
28     data.add(Calendar.DAY_OF_YEAR, 1);
29 }
30
31 XStream stream = new XStream(new DomDriver());
32 stream.alias("negocio", Negocio.class);
33 System.out.println(stream.toXML(negocios));
34
35 }
36 }
```

O `NO_REFERENCES` serve para indicar ao `XStream` que não queremos que ele crie 'referências' a tags que já foram serializadas iguaizinhas. Faça o teste sem essa opção para entender a diferença.

- 3) Faça com que `Negocio` seja um objeto `Comparable`, implementando essa interface. O critério de comparação deve ser delegado para a data do negócio (`Calendar` é `Comparable`).

Agora podemos, 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.

4.8 - 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.

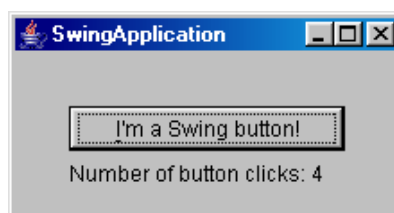
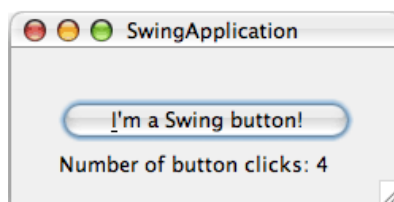
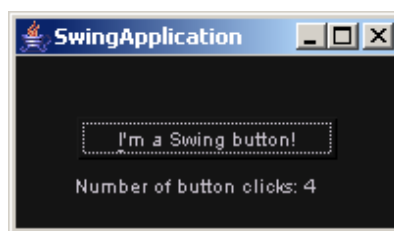
5.3 - Look And Feel

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: frames, botões, inputs, tabelas, janelas, abas, scroll, trees 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 - Diálogos

A classe mais simples do Swing, que costumamos usar muito no começo, é a `JOptionPane` que mostra diálogos de mensagem, confirmação, erros e outros.

Podemos mostrar uma mensagem para o usuário:

```
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 o diálogo:

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

O argumento do `showOpenDialog` indica qual o componente pai da janela de diálogo (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 ret = fileChooser.showOpenDialog(null);  
  
if (ret == 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 e horário da última negociação.

Usaremos um `JFileChooser` e um `JOptionPane` para mostrar o resultado.

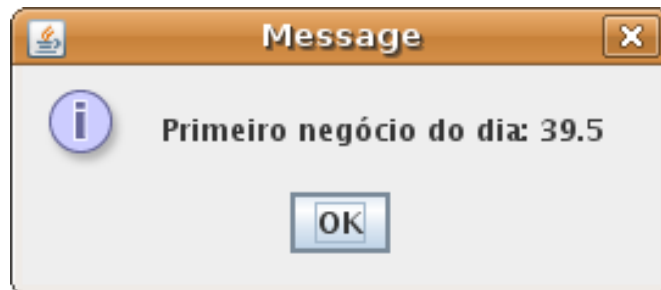
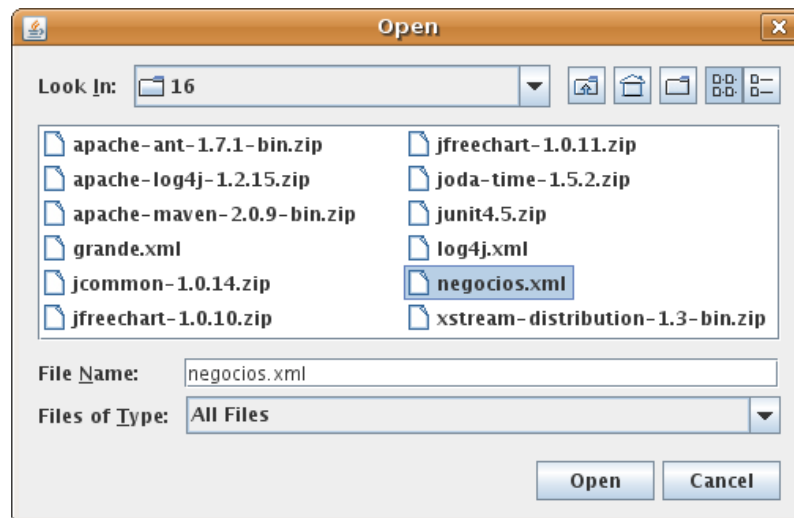
Crie a classe `EscolheXML` no pacote `br.com.caelum.argentum.ui`. Escreva o método `escolher`:

```
1 public void escolher() {  
2     try {  
3         JFileChooser fileChooser = new JFileChooser();  
4         int retorno = fileChooser.showOpenDialog(null);  
5  
6         if (retorno == JFileChooser.APPROVE_OPTION) {  
7             FileReader reader = new FileReader(fileChooser.getSelectedFile());  
8             List<Negocio> negocios = new LeitorXML().carrega(reader);  
9  
10            Negocio primeiroNegocio = negocios.get(0);  
11            String msg = "Primeiro negócio do dia: " + primeiroNegocio.getPreco();  
12            JOptionPane.showMessageDialog(null, msg);  
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 EscolheXML().escolher();  
}
```

Rode a classe e escolha o arquivo **negocios.xml** na pasta **Caelum/16** do 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 nosso projeto, por exemplo, faríamos:

```
new JFileChooser(".");
```

- 3) (opcional) Do jeito que fizemos, o usuário pode selecionar qualquer arquivo e não apenas os XMLs. Podemos filtrar por extensão de arquivo:

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

5.7 - Componentes: JFrame, JPanel e JButton

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 botão é simples:

```
 JButton botao = new JButton("Clique aqui");
```

Mas, para exibirmos vários componentes organizadamente, usamos um painel que vai conter esses componentes, o JPanel:

```
 JButton botaoCarregar = new JButton("Carregar XML");
 JButton botaoSair = new JButton("Sair");

 JPanel panel = new JPanel();
 panel.add(botaoCarregar);
 panel.add(botaoSair);
```

Repare que a ordem em que os componentes são adicionados importa. E repare também que não definimos posicionamento algum para os nossos componentes: estamos usando o comportamento default. (falaremos mais sobre os layout managers logo mais)

Por fim, uma janela em nosso sistema é representada pela classe JFrame:

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

O método `pack()` de `JFrame`, chamado acima, serve para redimensionar nosso frame para um tamanho adequado baseado nos componentes que ele tem. E o `setVisible` recebe um `boolean` indicando se queremos que a janela seja 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).

Uma prática comum no Swing é estender as classes containers, para trabalhar de uma maneira mais ‘fácil’. Muitas pessoas criticam essa abordagem, apesar de facilitar um pouco o desenvolvimento, dado o menor número de linhas digitadas. O problema é que as classes `Component` e `Container` possuem um ciclo de vida complicado, e reescrever seus métodos corretamente não é tarefa fácil.

Como já vimos no FJ-11, devemos priorizar o uso da composição, em vez de herança.

Effective Java

Item 16: Favoreça composição em vez de herança

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.

Um dos patterns base usados pelo Swing é o **Composite** aplicado aos componentes e containeres. 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) é disparada no 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 com métodos para serem disparados por eventos. 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.

Mas o mais comum de se encontrar é a implementação 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 muito pouco produtivo criar uma classe nova (em um `.java` separado) para cada evento. O mais comum é criarmos **classes internas** junto à classe principal que manipula os componentes que desejamos tratar.

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. Basicamente cria-se uma classe sem precisar declarar `class BlaBla` em momento algum.

Vamos usar uma classe anônima para tratar o evento de clique no botão que desliga a aplicação:

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

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:

```
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 isso é impossível de acontecer já que estamos falando 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` que chamada o `exit`. Mas qual é o nome dessa classe? Não sei, por isso é classe anônima.

5.11 - Exercícios: nossa primeira tela

- 1) Vamos começar nossa interface gráfica pela classe chamada `ArgentumUI` que possui um método `montaTela` que desenha a tela em Swing e um método `main` que apenas dispara a classe:

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

Sugestão: alternativamente à ordem dos exercícios, você pode começar criando o “esqueleto” do método `montaTela` (exercício 4), antes de prosseguir para os exercícios 2 e 3.

Criando primeiro esse guia, com o cursor sobre os erros de compilação que devem aparecer, aperte <Ctrl+1> e escolha a opção de gerar automaticamente os métodos. O Eclipse já criará os métodos com a assinatura correta.

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

Antes de tudo, vamos declarar alguns atributos que vamos precisar. **Adicione** na classe `ArgentumUI` os seguintes atributos:

```
private JFrame janela;  
  
private JPanel painelPrincipal;
```

- 3) Vamos escrever três métodos principais: um para montar a janela, um para montar o `painelPrincipal` e um para exibir a tela ao final. Mas para montar o `painelPrincipal` vamos precisar de mais dois métodos auxiliares para montar cada um dos botões. No final, teremos:

```
1 private void montaJanela() {  
2     janela = new JFrame("Argentum");  
3     janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
4 }  
5  
6 private void montaPainelPrincipal() {  
7     painelPrincipal = new JPanel();  
8     janela.add(painelPrincipal);  
9 }  
10  
11 private void montaBotaoCarregar() {  
12     JButton botaoCarregar = new JButton("Carregar XML");  
13     botaoCarregar.addActionListener(new ActionListener() {  
14         public void actionPerformed(ActionEvent e) {  
15             new EscolheXML().escolher();  
16         }  
17     });  
18 }
```

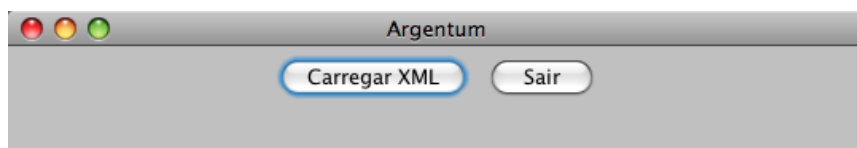
```
18     painelPrincipal.add(botaoCarregar);
19 }
20
21 private void montaBotaoSair() {
22     JButton botaoSair = new JButton("Sair");
23     botaoSair.addActionListener(new ActionListener() {
24         public void actionPerformed(ActionEvent e) {
25             System.exit(0);
26         }
27     });
28     painelPrincipal.add(botaoSair);
29 }
30
31 private void mostraJanela() {
32     janela.pack();
33     janela.setSize(540, 540);
34     janela.setVisible(true);
35 }
```

É bastante código, mas temos tudo bem separado em pequenos métodos.

- 4) **Adicione as chamadas** aos nossos métodos auxiliares *dentro* do método `montaTela` que criamos antes:

```
public void montaTela() {
    montaJanela();
    montaPainelPrincipal();
    montaBotaoCarregar();
    montaBotaoSair();
    mostraJanela();
}
```

- 5) Rode a classe acima e você deve ter o seguinte resultado:



Teste os botões e o disparo dos eventos.

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

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 maleável, podendo servir para exibição e edição de dados, podendo ter outros componentes dentro, com reorganização das colunas, drag and drop, ordenação e muitas outras coisas.

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 o comportamento das colunas com `TableColumn` e muitas outras coisas.

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 scroll pane. Depois esse scroll pane será adicionado ao panel com `panel.add(ps);`.

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 `DataModel` responsável por representar os dados da tabela.

Primeiro, vamos fazer o método `escolher` da classe `EscolheXML` devolver a `List<Negocio>` ao invés de manipulá-la. E no método que trata o evento de carregamento fazemos:

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

        // como passar um List<Negocio> para um TableModel?
        table.setModel(?????);
    }
});
```

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 o código acima funcionar.

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 um `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
32        return null;
33    }
34 }
```

Agora, basta usar nosso table model 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);  
    }  
});
```

5.14 - Exercícios: Tabela

- 1) Antes de colocarmos a tabela, vamos **alterar** a classe `EscolheXML` para devolver a lista de negócios ao invés de mostrar o diálogo de mensagem. **Altere** o método `escolher` para:

```
1 public List<Negocio> escolher() {  
2     try {  
3         JFileChooser fileChooser = new JFileChooser(".");  
4         int retorno = fileChooser.showOpenDialog(null);  
5  
6         if (retorno == JFileChooser.APPROVE_OPTION) {  
7             FileReader reader = new FileReader(fileChooser.getSelectedFile());  
8             return new LeitorXML().carrega(reader);  
9         }  
10    } catch (FileNotFoundException e) {  
11        e.printStackTrace();  
12    }  
13    return Collections.emptyList();  
14 }
```

`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, melhor que devolver `null`.

- 2) Na classe `ArgentumUI`, **adicione** um atributo para a tabela:

```
private JTable tabela;
```

E **adicione** o método `montaTabelaComScroll`:

```
1 private void montaTabelaComScroll() {  
2     tabela = new JTable();  
3     tabela.setBorder(new LineBorder(Color.black));  
4     tabela.setGridColor(Color.black);  
5     tabela.setShowGrid(true);  
6  
7     JScrollPane scroll = new JScrollPane();  
8     scroll.getViewPort().setBorder(null);  
9     scroll.getViewPort().add(tabela);  
10    scroll.setSize(450, 450);
```

```
11
12     painelPrincipal.add(scroll);
13 }
```

E, no final, **adicione a chamada** a esse método *dentro* do método `montaTela` logo acima das chamadas aos botões:

```
public void montaTela() {

    montaJanela();
    montaPainelPrincipal();

    montaTabelaComScroll();

    montaBotaoCarregar();
    montaBotaoSair();
    mostraJanela();
}
```

3) Crie a classe `NegociosTableModel` no pacote `br.com.caelum.argentum.ui` como vimos:

```
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     public int getColumnCount() {
10        return 3;
11    }
12
13    public int getRowCount() {
14        return negocios.size();
15    }
16
17    public Object getValueAt(int rowIndex, int columnIndex) {
18        Negocio n = negocios.get(rowIndex);
19        switch (columnIndex) {
20            case 0:
21                return n.getPreco();
22            case 1:
23                return n.getQuantidade();
24            case 2:
25                return n.getData();
26        }
27
28        return null;
29    }
30 }
```

4) Agora, precisamos **alterar** a classe anônima que trata o evento do botão de carregar XML para pegar a lista de negócios e transformá-la em um `NegociosTableModel` que depois vai ser passado para a tabela:

```

JButton botaoCarregar = new JButton("Carregar XML");

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

```

5) Rode a aplicacao e veja o resultado:

[illegible]

6) (opcional) Há alguns detalhes que podemos ainda melhorar. Por exemplo, fechar a tabela para edição (servirá apenas para visualizacao, analise dos dados). Ou então mudar os nomes das colunas.

Vá na classe `NegociosTableModel` e **adicione** os seguintes métodos, sobrescrevendo os herdados:

```
1 @Override
2 public boolean isCellEditable(int rowIndex, int columnIndex) {
3     return false;
4 }
5
6 @Override
7 public String getColumnName(int column) {
8     switch (column) {
9         case 0:
```



```

10         return "Preço";
11     case 1:
12         return "Quantidade";
13     case 2:
14         return "Data";
15     }
16
17     return null;
18 }
  
```

Rode a aplicacao e veja o resultado:

Preço	Quantidade	Data
5.0	6	java.util.GregorianCal...
5.0	6	java.util.GregorianCal...
5.0	6	java.util.GregorianCal...
5.0	6	java.util.GregorianCal...
5.0	6	java.util.GregorianCal...
5.0	6	java.util.GregorianCal...
5.0	6	java.util.GregorianCal...
5.0	6	java.util.GregorianCal...
5.0	6	java.util.GregorianCal...
5.0	6	java.util.GregorianCal...
5.0	6	java.util.GregorianCal...
5.0	6	java.util.GregorianCal...
5.0	6	java.util.GregorianCal...
5.0	6	java.util.GregorianCal...
5.0	6	java.util.GregorianCal...
5.0	6	java.util.GregorianCal...
5.0	6	java.util.GregorianCal...
5.0	6	java.util.GregorianCal...
5.0	6	java.util.GregorianCal...
5.0	6	java.util.GregorianCal...
5.0	6	java.util.GregorianCal...
5.0	6	java.util.GregorianCal...
5.0	6	java.util.GregorianCal...
5.0	6	java.util.GregorianCal...
5.0	6	java.util.GregorianCal...

Carregar XML Sair

7) (opcional) Vamos adicionar um titulo na nossa janela, usando um JLabel. Crie o componente assim:

```

private void montaTitulo() {

    JLabel titulo = new JLabel("Lista de Negócios");
    titulo.setFont(new Font("Verdana", Font.BOLD, 25));
    titulo.setForeground(new Color(50, 50, 100));
    titulo.setHorizontalAlignment(SwingConstants.CENTER);
    painelPrincipal.add(titulo);
}
  
```

Depois, basta adicioná-lo ao panel. **Altere** o do método montaTela:

```

public void montaTela() {

    montaJanela();
    montaPainelPrincipal();
}
  
```

```
    montaTitulo();  
    montaTabelaComScroll();  
    montaBotaoCarregar();  
    montaBotaoSair();  
    mostraJanela();  
}
```

Há ainda um problema bastante irritante que é a exibição da data e do valor. Vamos ver isso nas próximas seções.

5.15 - Formatando Datas: DateFormat

A classe `java.text.DateFormat` permite converter Strings de inputs do usuário para Dates, seguindo um determinado formato. Serve também para o caminho contrário: pegar uma data e gerar uma string de saída em um certo formato.

A `SimpleDateFormat` é a forma mais fácil de fazer tudo isso:

```
// usuário digita uma String  
String s = "21/02/1999";  
  
// date format  
DateFormat df = new SimpleDateFormat("dd/MM/yyyy");  
  
// converte para Date  
Date data = df.parse(s);  
  
// formata para String de novo  
String formatada = df.format(data);
```

5.16 - Exercícios: formatação

- 1) Faça a classe `NegocioTableModel` usar um `SimpleDateFormat` para o mostrar a coluna de data com uma formatação interessante.

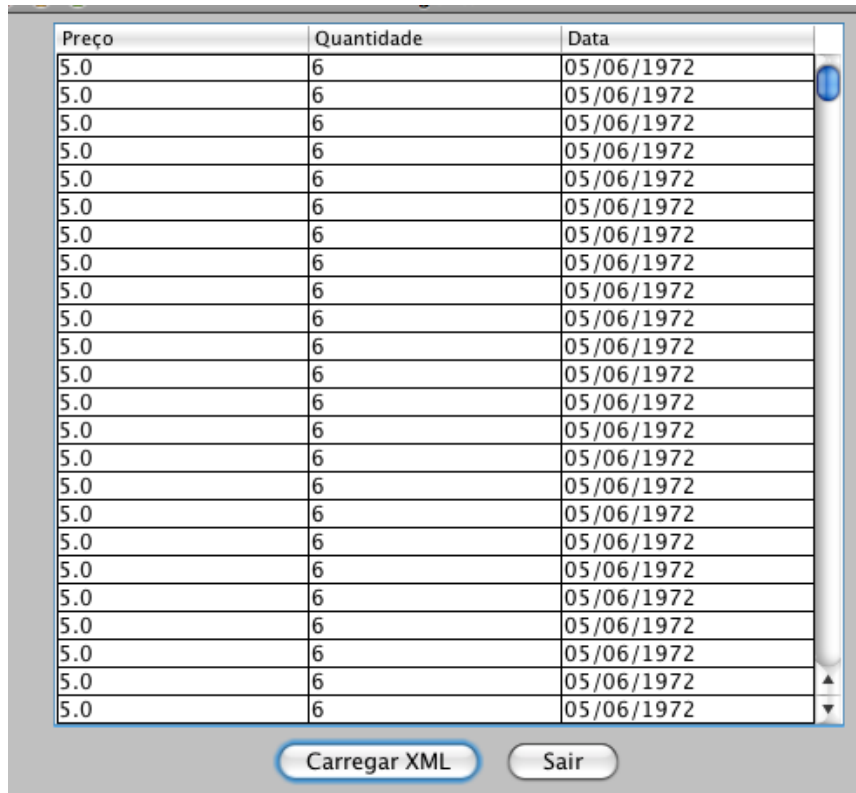
Crie um atributo novo na `NegocioTableModel`:

```
private final DateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy");
```

Agora, dentro do método `getValueAt`, ao invés de retornar `getData` direto, faremos:

```
return dateFormat.format(n.getData().getTime());
```

Rode novamente a aplicação e deveremos ter o seguinte:



Preço	Quantidade	Data
5.0	6	05/06/1972
5.0	6	05/06/1972
5.0	6	05/06/1972
5.0	6	05/06/1972
5.0	6	05/06/1972
5.0	6	05/06/1972
5.0	6	05/06/1972
5.0	6	05/06/1972
5.0	6	05/06/1972
5.0	6	05/06/1972
5.0	6	05/06/1972
5.0	6	05/06/1972
5.0	6	05/06/1972
5.0	6	05/06/1972
5.0	6	05/06/1972
5.0	6	05/06/1972
5.0	6	05/06/1972
5.0	6	05/06/1972
5.0	6	05/06/1972
5.0	6	05/06/1972
5.0	6	05/06/1972
5.0	6	05/06/1972
5.0	6	05/06/1972
5.0	6	05/06/1972
5.0	6	05/06/1972

Carregar XML Sair

2) (opcional) Podemos fazer a tabela formatar também os valores monetário.

Crie um atributo novo na `NegocioTableModel`:

```
private final NumberFormat numberFormat = NumberFormat.getCurrencyInstance();
```

Agora, dentro do método `getValueAt`, ao invés de retornar `getPreco` direto, faremos:

```
return numberFormat.format(n.getPreco());
```

Rode novamente a aplicação e veja o resultado.

Locale

Podemos alterar o formato de exibição mudando o `Locale` utilizado. Por padrão, é usado o locale da máquina virtual, mas podemos passar outro tanto para `DateFormat` quanto para `NumberFormat`. Mas o modo mais comum é alterar o `Locale` do sistema todo usando o método estático `setDefault` logo no início da aplicação. Você pode mudar isso colocando no método `main` da classe `ArgumentumUI`:

```
Locale.setDefault(new Locale("pt", "BR"));
```

5.17 - Para saber mais

- 1) Consultar o javadoc do Swing pode não ser muito simples. Por isso, a Sun disponibiliza um ótimo tutorial online sobre Swing em seu Site:
<http://java.sun.com/docs/books/tutorial/uiswing/>
- 2) Existem alguns bons editores visuais (Drag-and-Drop) para se trabalhar com Swing, entre produtos comerciais e livres. Destaque para:
 - Matisse, que vem embutido no Netbeans e é considerado, hoje, o melhor editor
 - VEP (Visual Editor Plugin), um plugin que pode ser instalado no Eclipse
- 3) 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 você pode utilizar Swing escrevendo suas interfaces gráficas em XML.
- 4) Existem diversos frameworks, como o JGoodies, que auxiliam na criação de telas através de APIs teoricamente mais simples que o Swing. Não são editores, e sim bibliotecas mais amigáveis. Apesar da existência de tais bibliotecas, é importante conhecer as principais classes e interfaces e seu funcionamento no Swing.

5.18 - Discussão em sala de aula: Listeners como classes top level, internas ou anônimas?

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

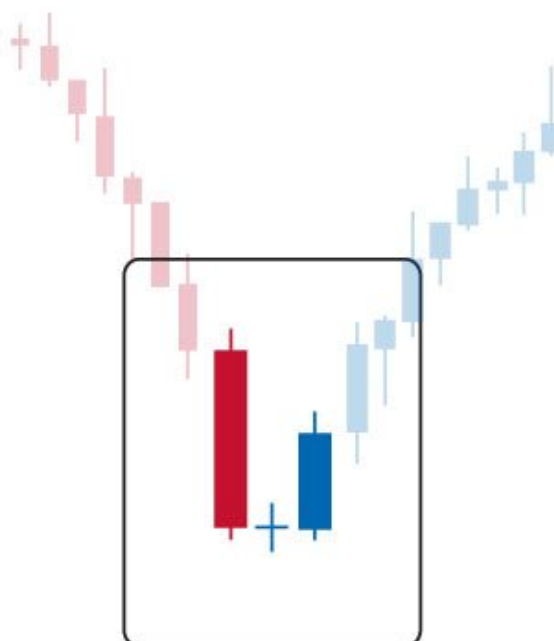
A **Análise Técnica Grafista** é uma escola econômica que tem como objetivo avaliar o melhor momento para compra e venda de ações através da análise histórica e comportamental do ativo na bolsa.

Suas principais características são:

- Análise dos dados gerados nas transações (preço, volume, etc);
- Uso de gráficos na busca de padrões;
- Análise de tendências.

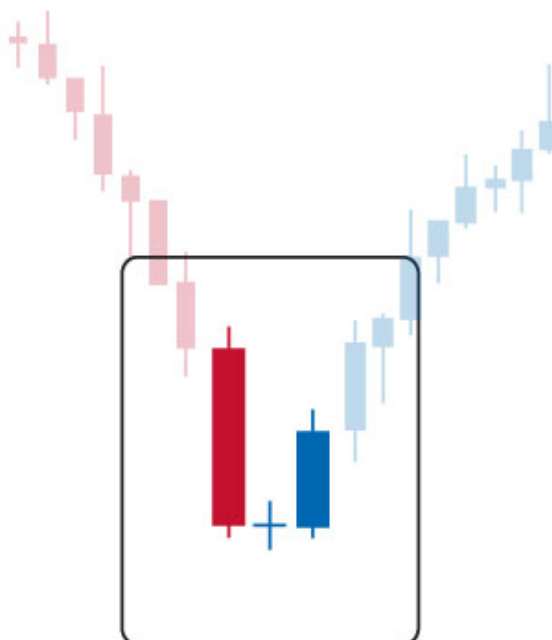
A análise técnica surgiu no início do século 20, com o trabalho de Charles Dow e Edward Jones. Eles criaram a empresa Dow Jones & Company e foram os primeiros a inventarem índices para tentar prever o comportamento do mercado de ações. O primeiro índice era simplesmente uma média ponderada de 11 ativos 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, dizem que se pode 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 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), entre uma grande queda e uma grande alta, indica que o mercado se inverteu, e começou a subir.

6.2 - Indicadores Técnicos

Como se trata de algo muito subjetivo, fica difícil buscar esses padrões através do computador (apesar de não ser impossível). Usaremos algo mais fácil para um computador traçar e calcular, totalmente baseado em critérios objetivos e numéricos: os indicadores técnicos.

Uma das várias formas de se aplicar as premissas da análise técnica grafista é através do uso de indicadores técnicos. São fórmulas que analisam as transações através de seus dados, como preço de abertura, volume, número de negócios, etc. Esse novo número, obtido através de uma fórmula, é determinístico e de fácil cálculo por um computador. Tanto que os analistas financeiros costumam programar diversas dessas fórmulas em macros VBScript, para poder vê-las dentro do Excel.

É comum, em uma mesma tela, termos uma combinação de gráficos, indicadores e até dos candlesticks:



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 sempre 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. As 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 Sun Microsystems (que não andam nada bem atualmente).



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.

Se você for pesquisar mais sobre as ações da Sun, hoje em dia sua sigla foi mudada para **JAVA** em vez de **SUNW**.

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:	1
dia 2:	2
dia 3:	3
dia 4:	4
dia 5:	3
dia 6:	5
dia 7:	4
dia 8:	3

Vamos fazer umas contas em que a média móvel calcula 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 **4**: $(5 + 3 + 4) / 3$. 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, ao invés de uma média aritmética simples, faz-se uma média ponderada onde damos mais *peso* para o valor mais atual e vamos diminuindo o peso dos valores conforme vão sendo mais antigos.

Por exemplo, para os dias a seguir:

DIA	FECHAMENTO
dia 1:	1
dia 2:	2
dia 3:	3
dia 4:	4
dia 5:	5
dia 6:	6

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 5.33: $(6*3 + 5*2 + 4*1)/6$.

6.4 - Exercícios: criando indicadores

- 1) Calcular uma média móvel é feito a partir de uma lista de resultados 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 que faz parte da média móvel que iremos calcular, 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 }
```

(opcional) Faça um teste unitário. O que vamos testar? O `getCandle(int)` já possui um bom comportamento: se for invocado com índices errados, a própria lista interna lançará uma `exception` adequada. Mesmo assim é bom testar, e deixar declarado que esperamos uma `IndexOutOfBoundsException` (filha de `ArrayIndexOutOfBoundsException`).

Teste também que essa classe não pode receber uma lista nula.

- 2) Vamos criar a média móvel, dentro do pacote `br.com.caelum.argentum.indicadores`. Por enquanto vamos criar apenas o que ela deve receber no construtor, e o método que calcula o valor da média dos últimos 3 dias dado o dia que se quer a média:

```
public class MediaMoveISimples {
```

```
    public double calcula(int posicao, SerieTemporal serie) {  
        return 0;  
    }  
  
}
```

A ideia é passarmos para o método calcula a SerieTemporal e o dia para o qual queremos calcular a média móvel. 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).

- 3) Como vamos fazer o teste? Precisamos criar uma SerieTemporal para isso. Isso dá muito trabalho! Criar candle por candle, para então criar a série. Seria interessante um método auxiliar que crie uma série 'de mentira' de acordo com valores dados, como por exemplo:

```
SerieTemporal serie = criaSerie(1, 2, 3, 4, 3, 4, 5, 4, 3);
```

Crie então a classe MediaMovelSimplesTest na pasta de testes e, lá dentro, crie um método auxiliar:

```
1 private SerieTemporal criaSerie(double... valores) {  
2     List<Candlestick> candles = new ArrayList<Candlestick>();  
3     for (double d : valores) {  
4         // cria candle com todos valores iguais, mil de quantidade e uma data qualquer  
5         candles.add(new Candlestick(d, d, d, d, 1000, Calendar.getInstance()));  
6     }  
7     return new SerieTemporal(candles);  
8 }
```

Repare que queremos criar uma série com diversos valores diferentes. Por isso criamos nosso método usando o recurso de **varargs** do Java 5.

E, agora sim, vamos ao teste da média móvel:

```
1 @Test  
  
2 public void testExemploSimplesParaMedia() {  
3     SerieTemporal serie = criaSerie(1, 2, 3, 4, 3, 4, 5, 4, 3);  
4     MediaMovelSimples mms = new MediaMovelSimples();  
5  
6     Assert.assertEquals(2.0, mms.calcula(2, serie), 0.00001);  
7     Assert.assertEquals(3.0, mms.calcula(3, serie), 0.00001);  
8     Assert.assertEquals(10.0 / 3, mms.calcula(4, serie), 0.00001);  
9     Assert.assertEquals(11.0 / 3, mms.calcula(5, serie), 0.00001);  
10    Assert.assertEquals(4.0, mms.calcula(6, serie), 0.00001);  
11    Assert.assertEquals(13.0 / 3, mms.calcula(7, serie), 0.00001);  
12    Assert.assertEquals(4.0, mms.calcula(8, serie), 0.00001);  
13 }
```

Rode o teste, **ele falha**, pois só retornamos zero! **Implemente** agora a lógica de negócio *dentro do método* da classe MediaMovelSimples (não na classe de teste). 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
10        return soma / 3;
11    }
12
13 }
```

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

- 4) (opcional) Crie a classe `MediaMovelPonderada` análoga a `MediaMovelSimples`. Ela deve passar pelo seguinte teste (crie também o método `criaSerie`):

```
public class MediaMovelPonderadaTest {

    @Test
    public void testMediaPonderada() {
        SerieTemporal serie = criaSerie(1, 2, 3, 4, 5, 6);
        MediaMovelPonderada mmp = new MediaMovelPonderada();

        // exemplo: calcula(2) é 1*1 + 2*2 + 3*3 = 14. Divide por 6, da 14/6
        Assert.assertEquals(14d / 6, mmp.calcula(2, serie), 0.00001);
        Assert.assertEquals(20d / 6, mmp.calcula(3, serie), 0.00001);
        Assert.assertEquals(26d / 6, mmp.calcula(4, serie), 0.00001);
        Assert.assertEquals(32d / 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:

```
1 public class MediaMovelPonderada {
2
3     public double calcula(int posicao, SerieTemporal serie) {
4         double soma = 0.0;
5         int peso = 1;
6
7         for (int i = posicao - 2; i <= posicao; i++) {
8             Candlestick c = serie.getCandle(i);
9             soma += c.getFechamento() * peso;
10            peso++;
11        }
12        return soma / 6;
13    }
14 }
```

Repare que o peso começa valendo 1 e vai sendo incrementado. O último dia deve ter peso 3, que é o tamanho do nosso intervalo.

A divisão por 6 no final é a soma dos pesos para o intervalo de 3 dias ($3 + 2 + 1 = 6$).

Rode o teste novamente e veja se passamos!

6.5 - Refatoração

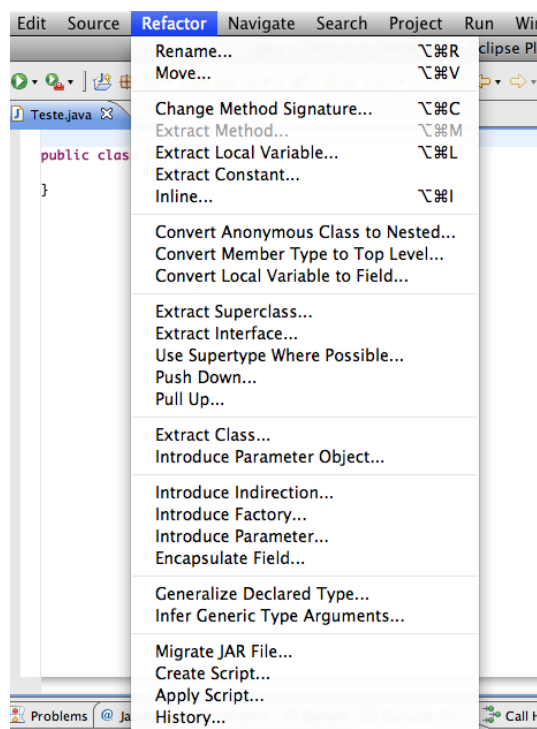
*Refatoração é uma técnica controlada para reestruturar um trecho de código existente, alterando sua estrutura interna sem modificar seu comportamento externo. Consiste em uma série de pequenas transformações que preservam o comportamento inicial. Cada transformação (chamada de refatoração) reflete em 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 que o sistema continue se comportando da mesma maneira que antes, é fundamental o uso de testes. Com eles, qualquer erro introduzido será imediatamente apontado, facilitando a correção a cada passo da refatoração de maneira imediata.

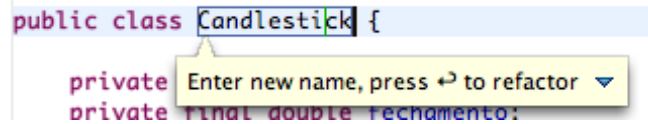
Algumas das refatorações mais recorrentes ganharam nomes que identificassem sua utilidade (veremos algumas nas próximas seções). Além disso, **Martin Fowler** escreveu o livro **Refactoring: Improving the Design of Existing Code**, onde descreve em detalhes as principais.

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**. Esse substantivo 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 no nome dela e ir no menu **Refactor > Rename**.



- 2) Abra a classe `CandlestickFactory` e observe o método `construirCandles`. Escrevemos esse método no capítulo de XML com um algoritmo para separar todos os negócios em vários candles.

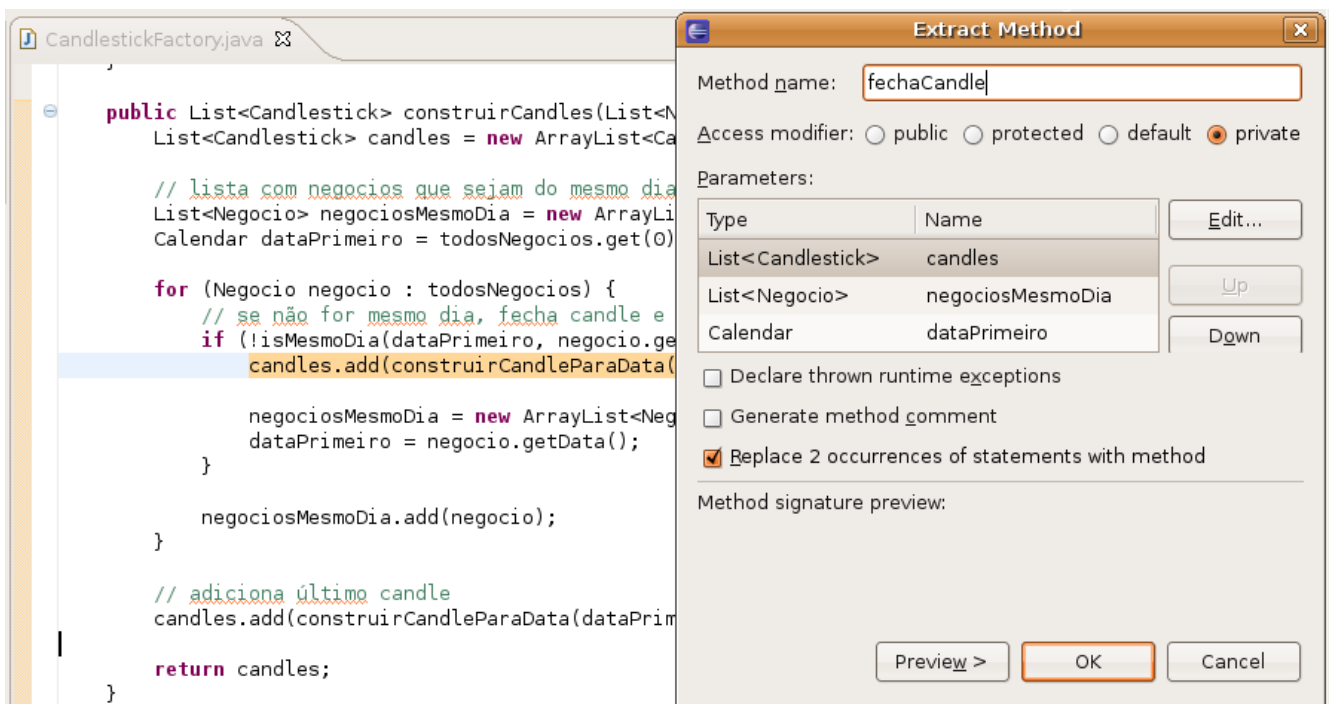
Mas, no meio desse código, há uma linha mais complicada que fecha cada candle, construindo o mesmo e adicionando na lista:

```
candles.add(construirCandleParaData(dataPrimeiro, negociosMesmoDia));
```

Pior, observe que essa linha aparece duas vezes no código (dentro e fora do for)!

Vamos isolar então essa parte repetida e complicada em um novo método que pode ser chamado várias vezes e 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 essa linha de código dentro do método `construirCandles`. Agora basta ir no menu *Refactor* e em *Extract Method*. Dê o nome do método de `fechaCandle` e clique em OK.



Repare como a IDE resolve os parâmetros e ainda substitui as chamadas ao código repetido pela chamada ao novo método.

Refactoring scripts

No Eclipse, é possível criar scripts de refatoração que incluem vários passos. É possível salvar esses scripts, aplicá-los em outros projetos e muitas outras coisas.

6.7 - Nossos indicadores e o design pattern Strategy

Agora que criamos nossas classes `MediaMovelSimples` e `MediaMovelPonderada` vamos criar gráficos para visualizar esses indicadores técnicos. Imagine que temos uma classe `GeradorDeGrafico` que plota gráficos de linha baseado em uma `SerieTemporal`. Algo assim:

```
SerieTemporal serie = ....
GeradorDeGrafico grafico = new GeradorDeGrafico(serie);
```

Como plotar as linhas de `MediaMovelSimples` e `MediaMovelPonderada`? Vamos encapsular em métodos para cada tipo de gráfico:

```
SerieTemporal serie = ....
GeradorDeGrafico grafico = new GeradorDeGrafico(serie);
grafico.plotaMediaMovelSimples();
grafico.plotaMediaMovelPonderada();
```

Temos um problema: cada vez que precisarmos desenhar um indicador técnico diferente, teremos que criar um método novo na classe `GeradorDeGrafico`. E como seria a implementação desses métodos que plotam? Praticamente a mesma, provavelmente apenas mudando que classe estamos instanciando. Será que conseguimos criar um único método para plotar e passar como argumento qual *indicador técnico* queremos plotar naquele momento?

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 {
    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 i) {
        // ....
    }
}
```

Na hora de desenhar os gráficos:

```
SerieTemporal serie = ....
GeradorDeGrafico grafico = new GeradorDeGrafico(serie);
```

```
grafico.plotaIndicador(new MediaMovelSimples());  
grafico.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 *Grafico*.

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*, *IndicadorVolume* etc.

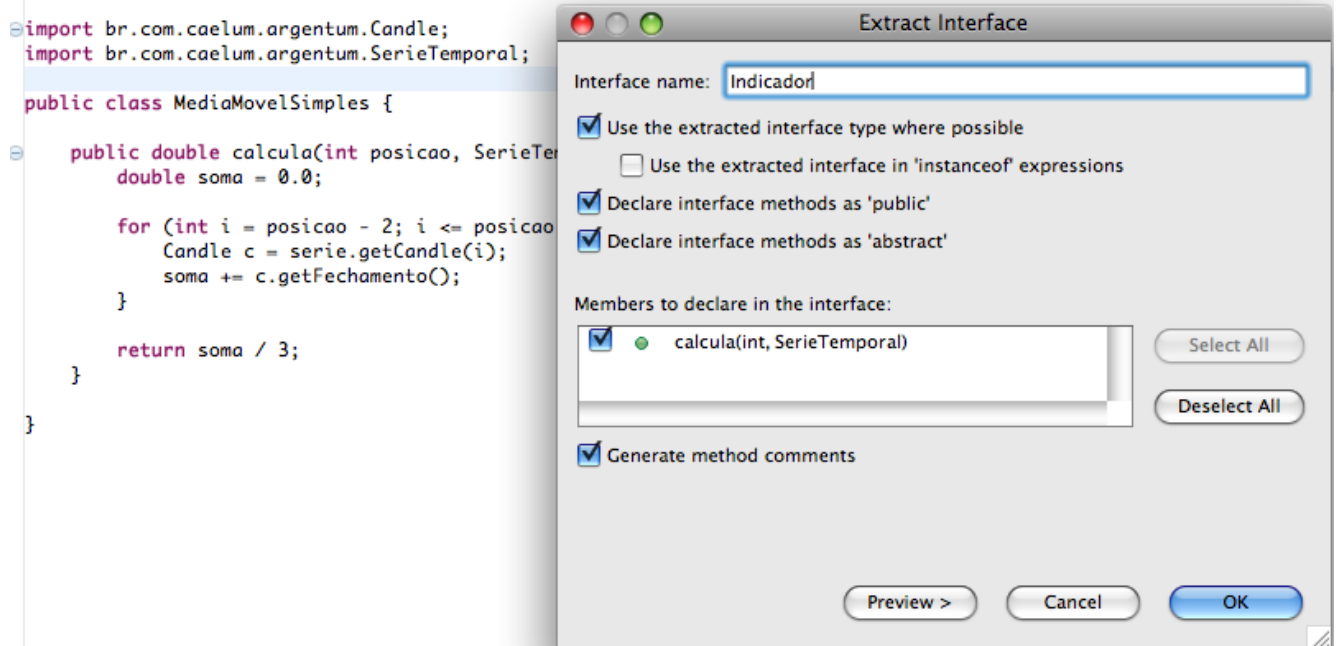
Temos agora vários **indicadores** diferentes, cada um com sua própria **estratégia** de recuperacao de valor, mas todos obedecendo a mesma interface: dada uma série temporal elas devolvem o valor do indicador. Esse é o design pattern chamado de **Strategy**.

Design Patterns são aquelas soluções padronizadas para problemas clássicos de orientação a objetos, como este que temos no momento (encapsular e ter flexibilidade). Nesse caso, está nos ajudando a deixar a classe *GeradorDeGrafico* isolada das diferentes estratégias de indicadores.

6.8 - Exercícios: refatorando para uma interface e usando bem os testes

- 1) Já que nossas classes de médias móveis são indicadores calculados, começamos extraindo a interface de um *Indicador* a partir dessas classes.

Abra a classe *MediaMovelSimples* e vá em **Refactor > 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) Precisamos criar uma classe para, por exemplo, ser o indicador do preço de fechamento, o `IndicadorFechamento`:

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

- 3) (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 volume.

- 4) 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.

No próximo capítulo, vamos usar nossos indicadores para desenhar os gráficos.

6.9 - 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) Extraia uma interface chamada `Serie` da classe `SerieTemporal`. Crie uma outra classe que implementa `Serie`, chamada de `SerieMock`, dentro do seu diretório fonte de test.

Essa classe deve receber no construtor um monte de doubles, e usar esse números para criar candles “de mentira”:

```
1 public class SerieMock implements Serie {
2     private final double[] valores;
3
4     public SerieMock(double... valores) {
5         this.valores = valores;
6     }
7
8     public Candle getCandle(int i) {
9         return new Candle(valores[i], valores[i], valores[i], valores[i],
10             1000, Calendar.getInstance());
11     }
12
13     public int getTotal() {
14         return valores.length;
15     }
16 }
```

Ela não é muito útil no dia a dia. Isso é uma classe que finge ser outra, um **mock**, útil para testar classes que possuem dependências.

6.10 - Discussão em aula: quando refatorar?

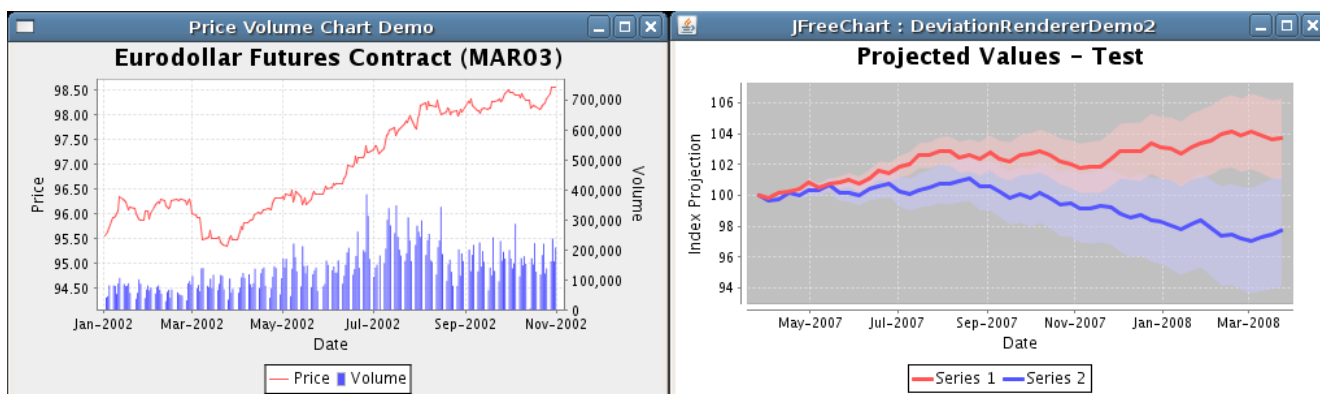
Gráficos com JFreeChart

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.

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 mais um monte de coisas. Consegue dar saída em JPG, PNG, SVG, EPS e 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; existe a `org.jfree.chart.ChartFactory` que possui uma série de métodos estáticos para facilitar nosso trabalho.

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 conseguimos 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 (podemos ter várias) e o label desse valor no eixo X.

Depois de criado o gráfico, queremos salvá-lo em um arquivo, enviar via rede, exibir na tela, mandar pra impressora ou qualquer coisa do gênero. A classe `org.jfree.chart.ChartUtilities` possui uma série de métodos para fazer coisas do tipo.

Talvez o método mais interessante e poderoso de `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:

Parameters:

- *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 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 salvar o mesmo na saída que quisermos. Mas se ela vai receber os dados para gerar o gráfico, como virão esses dados? Meu programa vai passar um `CategoryDataset`? E no dia que precisar trocar o `JFreeChart` e todo o meu programa usar o `CategoryDataset`?

Vamos encapsular a idéia de se passar os dados para o gráfico: vamos passar uma `SerieTemporal` e o intervalo que desejamos plotar. A própria classe `GeradorDeGrafico` se encarrega 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);
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 saber algumas informações como título e labels. E, para desenhar a linha do gráfico, vamos passar qual `Indicador` nos interessa.

```
1 public class GeradorDeGrafico {
2
3     private SerieTemporal serie;
4     private int comeco;
5     private int fim;
6
7     private DefaultCategoryDataset dataset;
8     private JFreeChart grafico;
9
10    public GeradorDeGrafico(SerieTemporal serie, int comeco, int fim) {
11        this.serie = serie;
12        this.comeco = comeco;
13        this.fim = fim;
14    }
15
16    public void criaGrafico(String titulo) {
17        this.dataset = new DefaultCategoryDataset();
18        this.grafico = ChartFactory.createLineChart(titulo, "Dias", "Valores",
19            dataset, PlotOrientation.VERTICAL, true, true, false);
20    }
21
22    public void plotaIndicador(Indicador indicador) {
23        for (int i = comeco; i <= fim; i++) {
24            double valor = indicador.calcula(i, serie);
```

```
25         dataset.addValue(valor, indicador.toString(), "" + i);
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 método `criaGrafico` recebe apenas uma `String` com o título do gráfico e cria o objeto `JFreeChart`, salvando-o no atributo `grafico`.

O método `plotaIndicador` recebe um `Indicador` qualquer e transforma a `serieTemporal` do gráfico no dataset gerado por esse indicador.

Por fim, o método `salvar` salva 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 a classe.

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

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

GeradorDeGrafico g = new GeradorDeGrafico(serie, 2, 15);
g.criaGrafico("Meu grafico");
g.plotaIndicador(new MediaMovelSimples());
g.salvar(new FileOutputStream("saida.png"));
```

Não usa nada específico do `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 no nosso sistema.

Ambos são ditos **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 me devolve o objeto prontinho para uso.

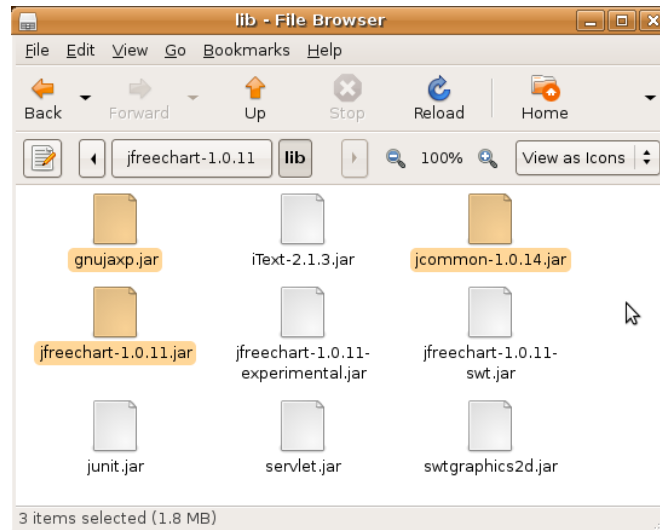
O padrão `Builder` é o que estamos usando na classe `GeradorDeGrafico`. Queremos encapsular a criação complicada de um gráfico e que pode mudar depois com o tempo (podemos querer usar outra API de geração de gráficos). Entra aí o *objeto construtor* da nossa classe `Builder`: seu único objetivo é descrever 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.

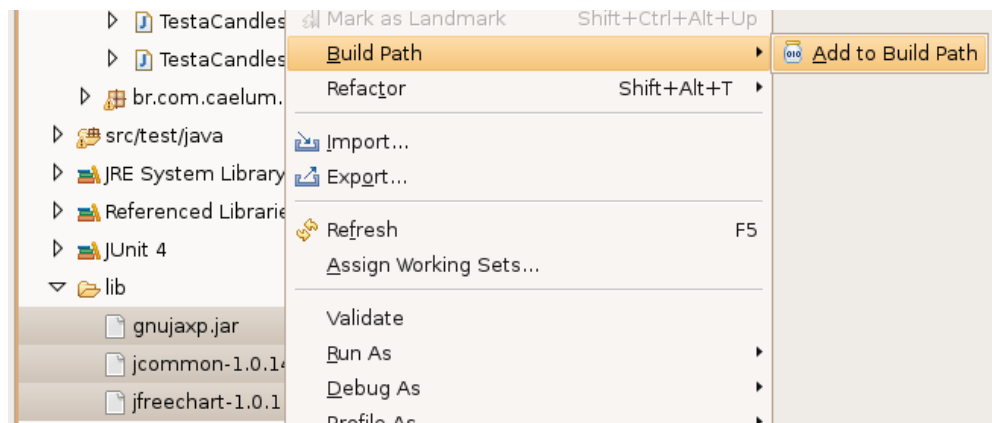
7.5 - Exercícios: JFreeChart

- 1) Abra a pasta **Caelum/16** no seu Desktop. Localize o ZIP do JFreeChart e dê dois cliques. Clique em **Extract** e indique o **Desktop** como destino.

Abra a pasta do jfreechart no seu Desktop e entre em **lib**. Copie os JARs do **gnujaxp**, do **jcommons** e do **jfreechart** para a pasta lib do seu projeto:



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



- 2) Crie a classe GeradorDeGrafico no pacote `br.com.caelum.argentum.grafico` como vimos antes. Ela encapsula o acesso ao JFreeChart.

Use os recursos do Eclipse para escrever esse código! Abuse do `Ctrl+Espaço`, do `Ctrl+I` e do `Ctrl+Shift+O`.

```

1 public class GeradorDeGrafico {
2
3     private final SerieTemporal serie;
4     private final int comeco;
5     private final int fim;
6
7     private DefaultCategoryDataset dataset;
8     private JFreeChart grafico;
9

```

```

10     public GeradorDeGrafico(SerieTemporal serie, int comeco, int fim) {
11         this.serie = serie;
12         this.comeco = comeco;
13         this.fim = fim;
14     }
15
16     public void criaGrafico(String titulo) {
17         this.dataset = new DefaultCategoryDataset();
18         this.grafico = ChartFactory.createLineChart(titulo, "Dias", "Valores",
19             dataset, PlotOrientation.VERTICAL, true, true, false);
20     }
21
22     public void plotaIndicador(Indicador indicador) {
23         for (int i = comeco; i <= fim; i++) {
24             double valor = indicador.calcula(i, serie);
25             dataset.addValue(valor, indicador.toString(), "" + i);
26         }
27     }
28
29     public void salvar(OutputStream out) throws IOException {
30         ChartUtilities.writeChartAsPNG(out, grafico, 500, 350);
31     }
32 }

```

- 3) Escreva uma classe para gerar um gráfico com alguns dados de teste. Crie uma classe TestaGrafico com um método main que usa nosso gerador:

```

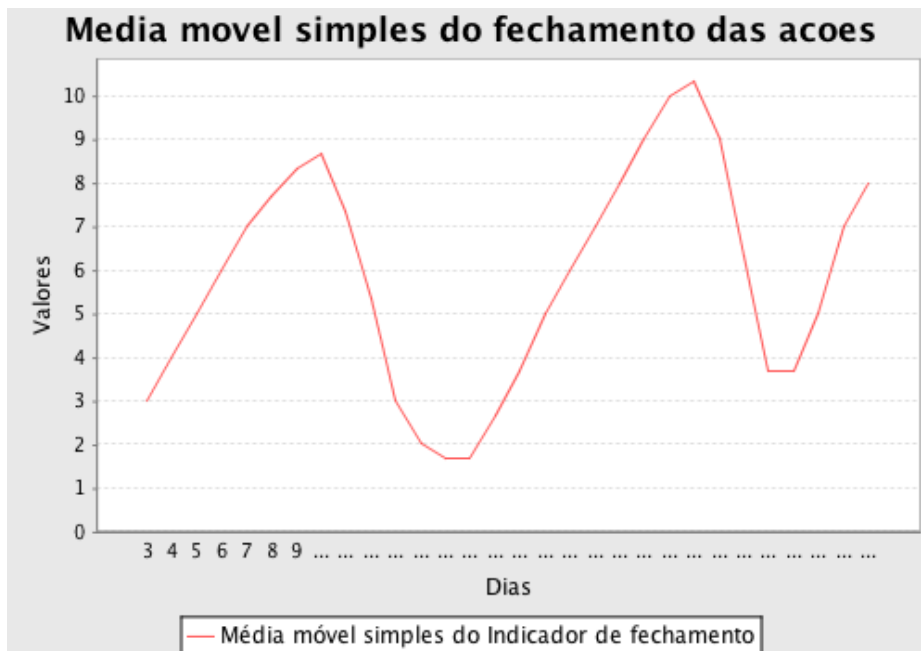
1 public class TestaGrafico {
2     public static void main (String[] args) throws IOException {
3         SerieTemporal serie = criaSerie(1,2,3,4,5,6,7,8,8,9,9,4,3,2,1,2,2,
4             4,5,6,7,8,9,10,11,10,6,3,2,6,7,8,9);
5
6         GeradorDeGrafico g = new GeradorDeGrafico(serie, 3, 32);
7         g.criaGrafico("Media movel simples do fechamento das acoes");
8         g.plotaIndicador(new MediaMovelSimples());
9         g.salvar(new FileOutputStream("grafico.png"));
10    }
11 }

```

(copie o criaSerie feito antes - ele está na classe MediaMovelSimplesTest, não esqueça de deixar o método como static)

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 é gerado rapidamente.

Para que a legenda apareça corretamente, sobrescreva os métodos `toString` nas classes `MediaMovelSimples` e `MediaMovelPonderada` (se existir).

- 5) Podemos em vez de criar um arquivo com a imagem, mostrá-la no Swing. Pra isso adicionamos um método que gera um `JPanel` dado um gráfico:

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

E na classe que tem o `main`:

```
JFrame frame = new JFrame("Minha janela");

frame.add(g.getPanel());

frame.pack();
frame.setVisible(true);
```

No próximo capítulo, vamos incorporar o gráfico à nossa aplicação em Swing.

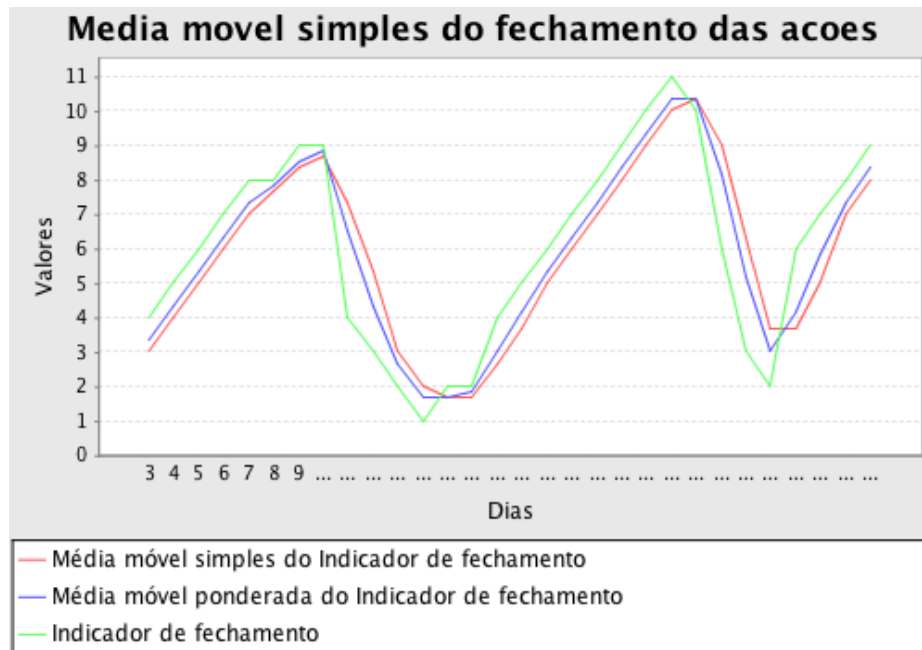
- 6) (opcional) Para que as legendas saiam corretas, implemente o método `toString` nas classes dos indicadores utilizados.
- 7) (opcional) Da forma que fizemos nosso Gerador, é possível plotar vários indicadores diferentes no mesmo gráfico. Altere seu teste para mostrar também um gráfico só com o preço de fechamento. E, se você fez o opcional anterior das médias ponderadas, inclua-as nesse gráfico.

Algo assim:

```
g.plotaIndicador(new MediaMovelSimples());
```

```
g.plotaIndicador(new IndicadorFechamento());
```

E o resultado será parecido com:



7.6 - Exercícios opcionais

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

Para isso, clique da direita na biblioteca (dentro de *referenced libraries* do *package explorer*), e selecione propriedades. Agora selecione source location e associe-o ao diretório *src* do *jfreechat*. 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 - Indicadores mais Elaborados e o Design Pattern Decorator

Construímos nosso Gerador de gráficos e, com isso, promovemos o desacoplamento entre nossa aplicação e a API de gráficos do JFreeChart. Agora sempre que quisermos um novo gráfico, precisamos apenas fornecer ao nosso Gerador uma *SerieTemporal*, o intervalo que desejamos plotar e logo em seguida o *Indicador* que deve ser utilizado na geração do nosso gráfico.

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 em cima do preço de fechamento. Mas, e se quisermos calculá-las a partir de outros indicadores? Por exemplo, a *média móvel simples do preço máximo*, ou do volume total negociado no dia, ou de outro indicador qualquer?

Criaríamos classes como `MediaMovelSimplesAbertura` e `MediaMovelSimplesVolume`? 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 classes `IndicadorFechamento`, `IndicadorAbertura` 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 MediaMovelSimples(new IndicadorAbertura());
```

Repare na flexibilidade desse código. O cálculo de média fica totalmente independente do dado usado. 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`, ao invés de chamarmos o `getFechamento`, delegamos a chamada para o `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;  
}
```

Ganhamos **muita flexibilidade**! Nossa classe `MediaMovelSimples`, junto com a interface `Indicador` que ela, ao mesmo tempo, implementa (é um indicador) e referencia (tem um outro indicador), formam o design pattern **Decorator**.

7.8 - Exercícios: Indicadores mais espertos e o Design Pattern Decorator

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

Em vez de chamarmos direto o `getFechamento`, delegamos isso para o indicador que sabe pegar o valor do fechamento.

- a) Crie o atributo **outroIndicador** na classe `MediaMovelSimple`:

```
private Indicador outroIndicador;
```

- b) Gere um novo construtor usando o atributo anterior (**Generate constructor using fields**).

- c) 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;  
}
```

- 2) Lembre que toda refatoração **deve** ser acompanhada dos testes correspondentes. Mas ao mexer na nossa classe `MediaMovelSimple` quebramos o teste `MediaMovelSimpleTest` que nem compila mais!

Precisamos mudar no teste a chamada ao construtor da nossa classe:

```
Indicador mms = new MediaMovelSimple(new IndicadorFechamento());
```

Agora, rode o teste 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.

- 3) Altere também a classe `TestaGrafico` para passar o `IndicadorFechamento` no construtor. Se você tiver outros indicadores, faça outras combinações e repare na flexibilidade.

Por exemplo:

```
g.plotaIndicador(new MediaMovelSimple(new IndicadorFechamento()));  
  
g.plotaIndicador(new MediaMovelSimple(new IndicadorAbertura()));  
g.plotaIndicador(new MediaMovelPonderada(new IndicadorFechamento()));  
  
// estranho, mas possível:  
g.plotaIndicador(  
    new MediaMovelSimple(new MediaMovelPonderada(new IndicadorFechamento()))  
);
```

- 4) (opcional) Modifique sua outra classe `MediaMovelPonderada` para também ser um Decorator.

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

- 5) (opcional) Crie uma outra classe `IndicadorVolume` e faça um teste unitário na classe `MediaMovelSimplesTest` que use esse novo indicador ao invés do de fechamento.

Faça também para abertura, máximo, mínimo ou outros que desejar.

7.9 - Desafio: Imprimir Candles

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.

7.10 - Desafio: Fluent Interface

Dois grandes nomes da orientação a objetos, Eric Evans e Martin Fowler, criaram o termo **Fluent Interfaces** para descrever interfaces de uso de classes mais limpas e intuitivas.

Eles argumentam que frequentemente precisamos usar muitas variáveis locais quando precisamos fazer operações uma após a outra que estejam interligadas. E sugerem um modo mais elegante de se fazer isso. Veja com o exemplo.

Hoje usamos a `GeradorDeGrafico` dessa forma:

```
GeradorDeGrafico g = new GeradorDeGrafico(serie, 3, 10);
g.criaGrafico("Meu Grafico");
g.plotaIndicador(new MediaMovelSimples(new IndicadorFechamento()));
g.salvar(new FileOutputStream("grafico.png"));
```

Em Fluent Interface ficaria algo como:

```
new Grafico()
    .chamado("Meu Grafico")
    .comSerieTemporal(serie)
    .de(3).ate(10)
    .comIndicador(new MediaMovelSimples(new IndicadorFechamento()))
    .salvar(new FileOutputStream("grafico.png"));
```

Repare que chamamos os métodos de forma encadeada, sem variáveis locais envolvidas.

O desafio é implementar a classe `GeradorDeGrafico` de forma a usar Fluent Interface e permitir chamadas como as feitas no código acima. O objetivo é **legibilidade**, além de encadear métodos.

Ótimo artigo sobre Fluent Interface

Martin Fowler fala bastante sobre fluent interfaces nesse ótimo artigo: <http://martinfowler.com/bliki/FluentInterface.html>

Usos famosos 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. A API Joda Time e o JMock são dois excelentes exemplos.

São muito usadas (e recomendadas) na construção de DSLs (Domain Specific Languages). Leia o artigo citado antes e pesquise sobre isso.

Mais Swing: layout managers, mais componentes e detalhes

8.1 - Gerenciadores de Layout

Quando adicionamos novos componentes, como o Java sabe onde posicioná-los? Por que sempre são adicionados do lado direito? Se redimensionamos a tela (teste isso), os elementos *fluem* para a linha de baixo. Por quê?

Essas e outras perguntas são respondidas pelo **Layout Manager**, o gerenciador de layout do Swing/AWT. O Java 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 a tela for redimensionada e etc.

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

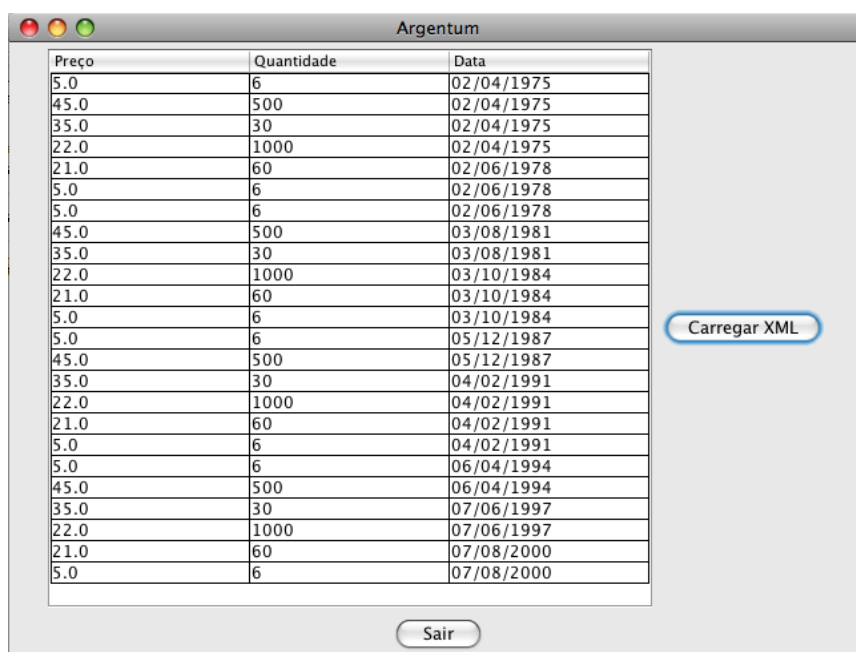


Figura 8.1: ArgentumUi com FlowLayout

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

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

Mas repare como nossa aplicação fica totalmente diferente:

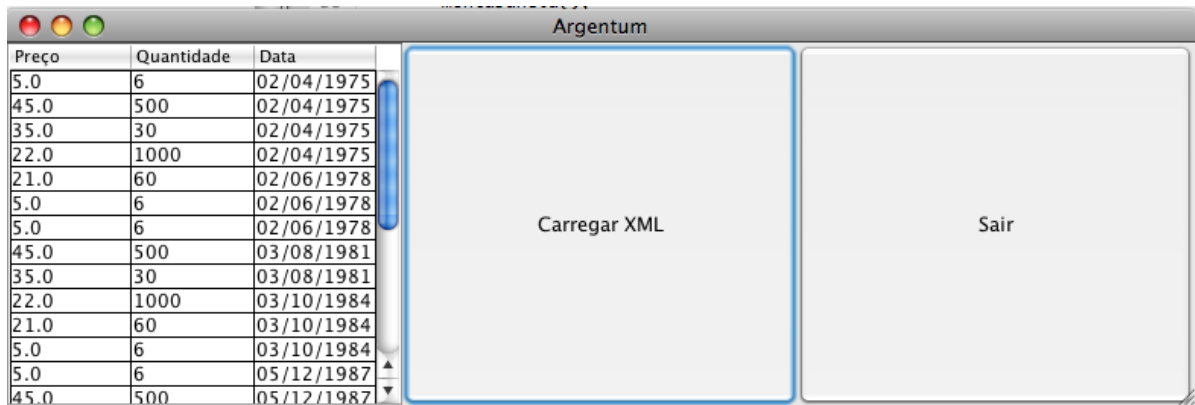
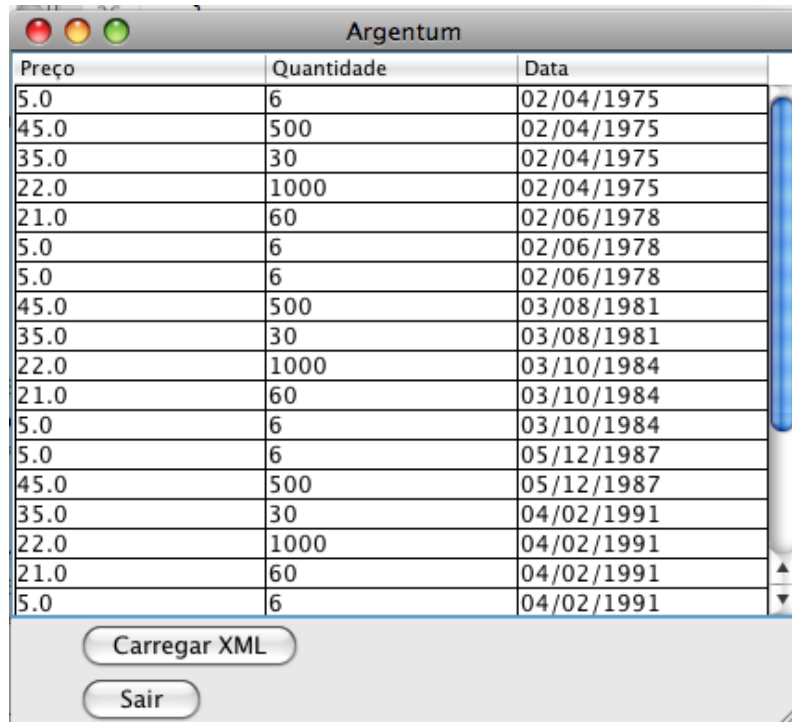


Figura 8.2: ArgentumUi com GridLayout

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
5.0	6	02/04/1975
45.0	500	02/04/1975
35.0	30	02/04/1975
22.0	1000	02/04/1975
21.0	60	02/06/1978
5.0	6	02/06/1978
5.0	6	02/06/1978
45.0	500	03/08/1981
35.0	30	03/08/1981
22.0	1000	03/10/1984
21.0	60	03/10/1984
5.0	6	03/10/1984
5.0	6	05/12/1987
45.0	500	05/12/1987
35.0	30	04/02/1991
22.0	1000	04/02/1991
21.0	60	04/02/1991
5.0	6	04/02/1991

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 `LayoutManager` 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 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.

BorderLayout

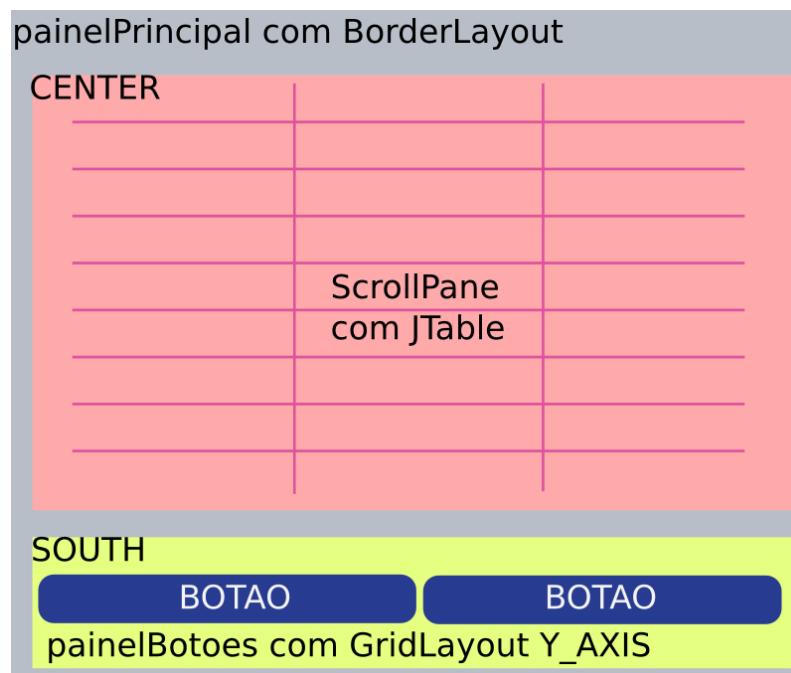
Divide o container em cinco regiões: Norte, Sul, Leste, Oeste e Centro. Ao adicionar um componente, indicamos a região onde queremos adicioná-lo. Na hora de renderizar, o BorderLayout primeiro coloca os componentes do Norte e Sul em seus tamanhos preferenciais; depois, coloca os do Leste e Oeste também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

GridBagLayout

É o mais complexo e poderoso layout, baseado no GridLayout. A ideia 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. A definição de onde deve ser colocado cada componente é feita através de restrições (GridBagConstraints) passadas ao método add.

8.3 - Exercícios: usando layout managers

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



2) **Adicione** o atributo `painelBotoes` na classe `ArgentumUI`:

```
private JPanel painelBotoes;
```

Adicione um método `montaPainelBotoes`:

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

Altere o método `montaTela` para chamar nosso novo método (cuidado com a ordem):

```
public void montaTela() {  
  
    montaJanela();  
    montaPainelPrincipal();  
    montaTabelaComScroll();  
    montaPainelBotoes();  
    montaBotaoCarregar();  
    montaBotaoSair();  
    mostraJanela();  
}
```

Altere os métodos `montaBotaoCarregar` e `montaBotaoSair` 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.

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

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

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

Altere o método `montaTabelaComScroll` 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 `montaPainelBotoes` para indicar que queremos adicioná-lo na região sul. Basta 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.

Argentum		
A	B	C
R\$ 39,50	1076	13/09/2008
R\$ 40,45	1033	13/09/2008
R\$ 39,82	1118	13/09/2008
R\$ 39,21	1144	13/09/2008
R\$ 39,86	1081	13/09/2008
R\$ 39,06	1166	13/09/2008
R\$ 38,22	1188	13/09/2008
R\$ 37,58	1138	13/09/2008
R\$ 37,11	1046	13/09/2008
R\$ 37,32	990	13/09/2008
R\$ 37,62	1023	14/09/2008
R\$ 38,37	1076	14/09/2008
R\$ 38,72	1162	14/09/2008
R\$ 38,14	1105	14/09/2008
R\$ 37,28	1041	14/09/2008
R\$ 37,96	1031	15/09/2008
R\$ 38,10	1081	15/09/2008

Carregar XML Sair

4) (opcional) Se você fez o título no exercício anterior, agora adicione-o ao painelPrincipal na posição NORTH.

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 (tab). 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", meuComponente1);
abas.add("Label 2", meuComponente2);
```

Para gerar o gráfico baseado na lista de negócios, usamos toda a infraestrutura que fizemos até agora:

```
1 List<Negocio> negocios = ....
2
3 CandlestickFactory candlestickFactory = new CandlestickFactory();
4 List<Candle> candles = candlestickFactory.constroiCandles(negocios);
5
6 SerieTemporal serie = new SerieTemporal(candles);
7
8 GeradorDeGrafico geradorDeGrafico = new GeradorDeGrafico(serie, 2, serie.getTotal() - 1);
9 geradorDeGrafico.criaGrafico("Média Móvel Simples");
10 geradorDeGrafico.plotaIndicador(new MediaMovelSimples(new IndicadorFechamento()));
11 JPanel grafico = geradorDeGrafico.getPanel();
```

8.5 - Exercícios: completando a tela da nossa aplicação

1) Na classe `ArgentumUI`, **adicione** o atributo `abas`:

```
private JTabbedPane abas;
```

Crie o método `montaAbas`:

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

Altere o método `montaTabelaComScroll` para colocar a tabela com scroll na primeira aba do tabbed pane:

```
// retire a linha abaixo:  
  
//painelPrincipal.add(scroll, "Center");  
  
// adicione:  
abas.setComponentAt(0, scroll);
```

Adicione a chamada ao `montaAbas` no método `montaTela` (cuidado com a ordem):

```
public void montaTela() {  
    montaJanela();  
    montaPainelPrincipal();  
    montaAbas();  
    montaTabelaComScroll();  
    montaPainelBotoes();  
    montaBotaoCarregar();  
    montaBotaoSair();  
    mostraJanela();  
}
```

Rode novamente e observe a montagem das abas.



Tabela de Negócios		
A	B	C
R\$ 39,50	1076	13/09/2008
R\$ 40,45	1033	13/09/2008
R\$ 39,82	1118	13/09/2008
R\$ 39,21	1144	13/09/2008
R\$ 39,86	1081	13/09/2008
R\$ 39,06	1166	13/09/2008
R\$ 38,22	1188	13/09/2008
R\$ 37,58	1138	13/09/2008
R\$ 37,11	1046	13/09/2008
R\$ 37,32	990	13/09/2008
R\$ 37,62	1023	14/09/2008
R\$ 38,37	1076	14/09/2008
R\$ 38,72	1162	14/09/2008
R\$ 38,14	1105	14/09/2008

Carregar XML Sair

- 2) 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.

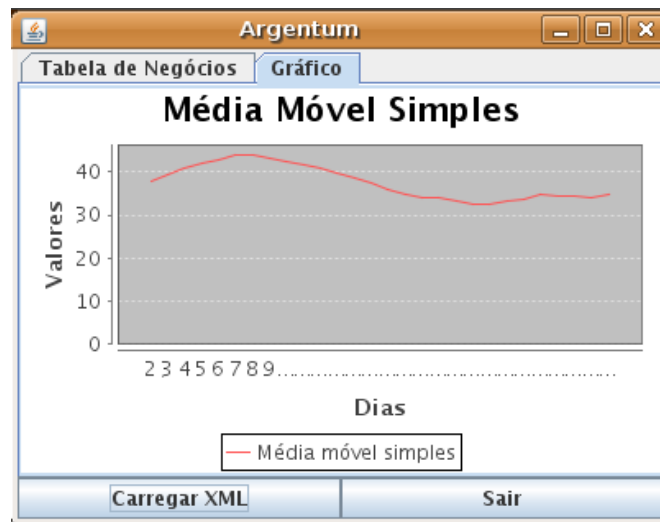
Vamos então criar um **método auxiliar** chamado `carregarDados` que será responsável por tratar o evento de carregar tanto a **tabela** quanto o **gráfico**. O código fica grande mas é tudo o que já vimos até agora:

```
1 private void carregarDados() {
2     List<Negocio> negocios = new EscolheXML().escolher();
3
4     // atualiza tabela
5     NegociosTableModel ntm = new NegociosTableModel(negocios);
6     this.tabela.setModel(ntm);
7
8     // gera SerieTemporal
9     CandlestickFactory candlestickFactory = new CandlestickFactory();
10    List<Candle> candles = candlestickFactory.constroiCandles(negocios);
11    SerieTemporal serie = new SerieTemporal(candles);
12
13    // mostra grafico
14    GeradorDeGrafico geradorDeGrafico = new GeradorDeGrafico(serie, 2, serie.getTotal() - 1);
15    geradorDeGrafico.criaGrafico("Média Móvel Simples");
16    geradorDeGrafico.plotaIndicador(new MediaMovelSimples(new IndicadorFechamento()));
17    JPanel grafico = geradorDeGrafico.getPanel();
18    this.abas.setComponentAt(1, grafico);
19 }
```

Agora, vamos **alterar** a classe anônima que trata o evento do `botaoCarregar` para, ao invés de executar toda a lógica lá dentro, chamar **apenas** o `carregarDados`:

```
botaoCarregar.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        carregarDados();
    }
});
```

Rode novamente e teste o gráfico e as abas!



8.6 - 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 separadores de data, formatador, que aceite apenas datas válidas. Uma classe filha de `JTextField` já traz boa parte desse trabalho pronto: 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. Depois, basta adicioná-lo a um painel e obtemos o seguinte resultado:



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);
```

8.7 - Exercícios: filtrando por data

1) Na classe `ArgentumUI`, **adicione** um novo atributo `campoDataInicio`:

```
private JFormattedTextField campoDataInicio;
```

Adicione um novo método chamado `montaCampoData` para inicializar o `campoDataInicio`:

```
private void montaCampoData() {  
    try {  
        MaskFormatter mascara = new MaskFormatter("##/##/####");  
        mascara.setPlaceholderCharacter('_');  
  
        campoDataInicio = new JFormattedTextField(mascara);  
        painelBotoes.add(campoDataInicio);  
    } catch (ParseException e) {  
        e.printStackTrace();  
    }  
}
```

No método `montaTela`, adicione a chamada ao `montaCampoData` (cuidado com a ordem):

```
public void montaTela() {  
    montaJanela();  
    montaPainelPrincipal();  
    montaAbas();  
    montaTabelaComScroll();  
    montaPainelBotoes();  
    montaCampoData();  
    montaBotaoCarregar();  
    montaBotaoSair();  
    mostraJanela();  
}
```

Rode a aplicação e veja o resultado:



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

Vamos **criar um método** que recebe a lista de negócios e remove dessa lista todos os negócios que sejam de data menores que a data selecionada pelo usuário. O método deverá se chamar `filtraPorData` e receberá a `List<Negocio>` lida do XML.

Internamente, vamos obter o objeto `Date` direto do `campoDataInicio` e depois percorrer a lista removendo os elementos antigos. Para isso, usaremos o `Iterator`:

```

1 private void filtraPorData(List<Negocio> negocios) {
2     try {
3         String valor = (String) campoDataInicio.getValue();
4         SimpleDateFormat formato = new SimpleDateFormat("dd/MM/yyyy");
5         Date dataInicio = formato.parse(valor);
6
7         Iterator<Negocio> it = negocios.iterator();
8         while(it.hasNext()) {
9             if (it.next().getData().getTime().before(dataInicio)) {
10                 it.remove();
11             }
12         }
13     } catch (Exception e) {
14         campoDataInicio.setValue(null);
15     }
16 }
  
```

Repare que, primeiro, precisamos fazer a transformação do campo para um `Date`. Caso ocorra uma exception aí (data inválida), não mexemos na lista e ainda mudamos o valor do campo de data para `null`, assim o usuário poderá digitar outro valor válido se quiser.

Agora só falta **adicionar** a chamada a esse novo método dentro do `carregarDados`. Faça isso na linha *logo* após a criação da lista:

```

private void carregarDados() {

    List<Negocio> negocios = new EscolheXML().escolher();
    filtraPorData(negocios);

    // ...
  
```

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 Java Swing window titled "Argentum". It has two tabs: "Tabela de Negócios" (selected) and "Gráfico". The table has three columns: A, B, and C. Column A contains monetary values in Brazilian Reals (R\$), column B contains integers, and column C contains dates. At the bottom of the window, there is a date field set to "12/09/2008" and two buttons: "Carregar XML" and "Sair".

A	B	C
R\$ 39,50	1076	13/09/2008
R\$ 40,45	1033	13/09/2008
R\$ 39,82	1118	13/09/2008
R\$ 39,21	1144	13/09/2008
R\$ 39,86	1081	13/09/2008
R\$ 39,06	1166	13/09/2008
R\$ 38,22	1188	13/09/2008
R\$ 37,58	1138	13/09/2008
R\$ 37,11	1046	13/09/2008
R\$ 37,32	990	13/09/2008
R\$ 37,62	1023	14/09/2008
R\$ 38,37	1076	14/09/2008
R\$ 38,72	1162	14/09/2008

Iterator e remoções

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

- 3) (opcional) Repare que, ao colocar valores que extrapolem o limite do campo (por exemplo um mês 13), é assumido que é o mês de 01 do ano seguinte. Para mudar esse comportamento, chame `setLenient(false)` no `SimpleDateFormat` que estamos usando.
- 4) (opcional) Use um `JLabel` para colocar um label ao lado do campo de data

8.8 - 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` 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`.

8.9 - Exercícios opcionais: escolher os indicador(es) para o gráfico

- 1) Nosso gráfico atualmente apenas desenha a média móvel simples de do indicador de fechamento. Vamos, através de um `JMenuBar`, possibilitar ao usuário escolher quais indicadores ele deseja plotar.

Na classe `ArgentumUI`, **adicione** dois novos atributos:

```
private JCheckBoxMenuItem mediaFechamento;  
  
private JCheckBoxMenuItem mediaAbertura;
```

Adicione agora um método chamado `montaMenu`:

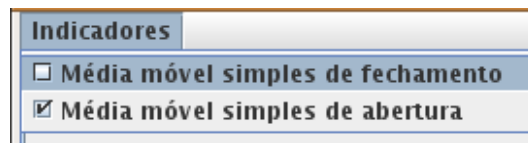
```
1 private void montaMenu() {  
2     JMenuBar menuBar = new JMenuBar();  
3     janela.setJMenuBar(menuBar);  
4  
5     JMenu menuIndicadores = new JMenu("Indicadores");  
6     menuBar.add(menuIndicadores);  
7  
8     mediaFechamento = new JCheckBoxMenuItem("Média móvel simples de fechamento");  
9     menuIndicadores.add(mediaFechamento);  
10  
11    mediaAbertura = new JCheckBoxMenuItem("Média móvel simples de abertura");  
12    menuIndicadores.add(mediaAbertura);  
13  
14 }
```

Você pode adicionar outros indicadores nesse menu se desejar.

Agora, **altere** o método `montaTela` para chamar esse nosso `montaMenu` (cuidado com a ordem):

```
public void montaTela() {  
    montaJanela();  
    montaMenu();  
    montaPainelPrincipal();  
    montaAbas();  
    montaTabelaComScroll();  
    montaPainelBotoes();  
    montaCampoData();  
    montaBotaoCarregar();  
    montaBotaoSair();  
    mostraJanela();  
}
```

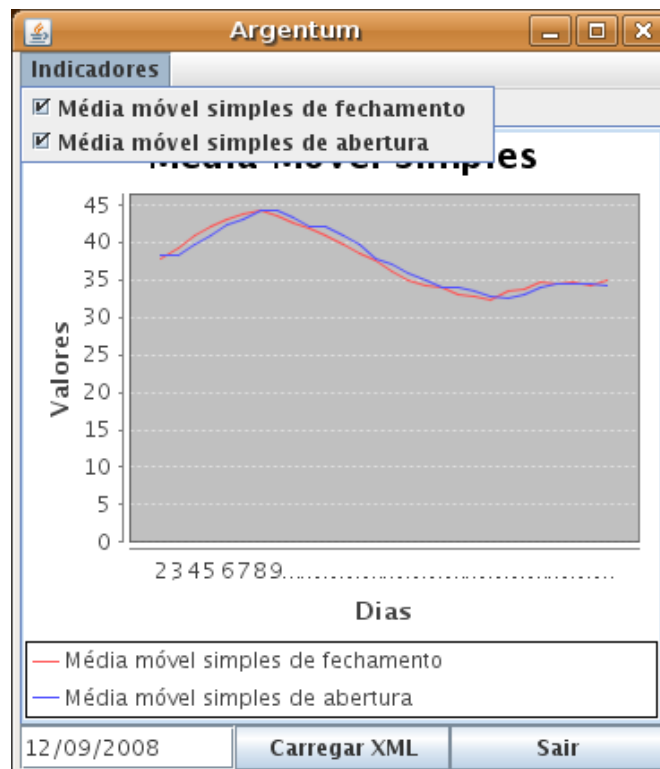
Rode sua aplicação e teste o menu (por enquanto, sem efeito).



- 2) Precisamos agora **alterar** o método `carregarDados()` para usar os indicadores escolhidos no menu. Altere as linhas que tratam de chamar o `plotaIndicador`:

```
private void carregarDados() {  
  
    // ... comeco ...  
  
    if (mediaFechamento.isSelected())  
        geradorDeGrafico.plotaIndicador(new MediaMovelSimples(new IndicadorFechamento()));  
  
    if (mediaAbertura.isSelected())  
        geradorDeGrafico.plotaIndicador(new MediaMovelSimples(new IndicadorAbertura()));  
  
    // ... fim ...  
}
```

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



- 3) (opcional) Faça mais indicadores, com média ponderada e outros.
- 4) (opcional) Faça com que as ações e “Carregar XML” e “Sair” também possam ser executadas via alguma opção no menu bar.

8.10 - Discussão em sala de aula: uso de IDEs para montar a tela

Reflection e Annotations

9.1 - Por que Reflection?

Por ser uma linguagem compilada, Java permite que, enquanto escrevemos nosso código, tenhamos total controle sobre exatamente o que vai ser executado, o 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, onde precisamos de características dinâmicas. Por exemplo, imagine permitir que o usuário passe um nome de método e nós invocamos esse método. Como fazer?

```
public void chamaMetodo(String metodo) {  
    if (metodo.equals("metodo1")) {  
        this.metodo1();  
    } else if (metodo.equals("metodo2")) {  
        this.metodo2();  
    } else if (metodo.equals("metodo3")) {  
        this.metodo3();  
    }  
    // etc...  
}
```

O código acima não parece repetitivo? O grande problema é que, em tempo de desenvolvimento, não sabemos exatamente qual método vai ser chamado. Isso é **dinâmico**, o usuário escolhe em **tempo de execução** e nós chamamos. Não seria muito mais interessante se houvesse uma forma de falar ao Java: “invoque o método cujo nome está na variável `metodo`”.

Pense no XStream que vimos anteriormente. Dado um XML, ele consegue criar objetos para nós e colocar os dados todos nos nossos atributos. 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. Será que no código deles há um `new` para cada objeto possível e imaginável do mundo? Obviamente não. Mas então como ele consegue instanciar minha classe, chamar meus atributos, tudo dinamicamente sem precisar ter `new Negocio` escrito dentro dele?

Reflection é um pacote do Java que permite chamadas dinâmicas em tempo de execução sem precisar conhecer as classes e objetos envolvidos quando escrevemos nosso código (tempo de compilação). É ideal para resolvermos determinadas coisas que nosso programa só descobre quando realmente estiver o rodando. O XStream só descobre o nome da nossa classe `Negocio` quando rodamos o programa; enquanto escreviam o framework, não tinham a menor idéia de que um dia usaríamos com a classe `Negocio`.

É possível fazer muitas coisas usando reflection. Apenas para citar algumas possibilidades:

- Listar todos os atributos de uma classe e pegar seus valores em um objeto específico;
- 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 foram anotados com as anotações do Java 5.

9.2 - Reflection: Class, Method

O ponto de partida de reflection é a classe `Class`. É uma classe do Java que representa cada classe que temos no sistema (classes nossas, de jars, do próprio Java). Através dessa classe `Class` conseguimos obter informações sobre qualquer classe do sistema, como os atributos dela, os métodos, os construtores etc.

Dado um objeto, é fácil pegar o `Class` dele:

```
Negocio n = new Negocio();

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

Mas não precisamos de um objeto para pegar esse objeto especial. Direto no nome da classe também conseguimos:

```
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 `T`, parametrizado pela classe.

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

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

A saída será:

```
getPreco
getQuantidade
getData
```

getVolume

É possível fazer muito mais coisas. Dê uma investigada na API de reflection pelo *Ctrl+Espaço* no Eclipse e pelo javadoc.

9.3 - Usando anotações

Anotações (annotations) são uma novidade do Java 5 cujo objetivo é trazer a capacidade de declaração de **metadados** nos nossos objetos. Até agora, usamos anotações para indicar que um método não deve mais ser usado (`@Deprecated`), para configurar um alias do XStream (`@XStreamAlias`), para indicar que estamos sobrescrevendo um método (`@Override`) e talvez 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 com anotações são muito usados para configurações de outras funcionalidades. Por exemplo, usamos a anotação do XStream para configurar o comportamento do XStream em relação à nossa classe. Mas se retirarmos a anotação, do ponto de vista da nossa classe (não do XStream), nada de diferente acontecerá.

Em geral, portanto, usamos anotações para criar metadados sobre nossos artefatos de programação 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` não é uma tarefa fácil. Precisamos definir desde as características de renderização do componente até o modelo de dados que vai ser exibido (`TableModel`).

Criamos anteriormente uma classe chamada `NegocioTableModel` para devolver os dados dos negócios que queremos exibir na tabela. Mas imagine se depois temos que criar uma tabela com uma listagem de candles, um para cada dia. Provavelmente criaríamos um `TableModel` muito parecido com o de negócios, e apenas substituiríamos as chamadas aos getters de uma classe pelos getters de outra.

São códigos muito parecidos. A diferença é que cada um chama os getters do objeto que vai ser mostrado naquele table model específico. Mas será que conseguiríamos unir tudo em um único table model? Um table model que, dependendo da classe (`Negocio / Candle`) passada, consegue chamar os getters apropriados? Poderíamos colocar um monte de IFs, mas com reflection teremos bem menos trabalho.

A partir do `Class` que estamos manipulando, conseguimos iterar nos seus atributos (`Field`) por exemplo. E podemos pegar os valores desses atributos e o nome do atributo para montar a tabela dinamicamente.

9.5 - Criando sua própria anotação

Um problema ao montar a tabela dinamicamente é saber a ordem em que queremos exibir as colunas. Quando usamos reflection não sabemos exatamente a ordem em que os fields são percorridos.

Para *configurar* esse tipo de informação (a posição na tabela), poderíamos usar uma anotação. Por exemplo, imagine anotar a classe `Negocio` dessa forma:


```
public class Negocio {  
  
    @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();  
    }  
  
    // ... outras coisas  
}
```

Depois, vamos querer percorrer os atributos dessa classe, olhar para o valor dessas anotações e montar a tabela dinamicamente com essas posições.

Mas, como escrever uma anotação em Java? As anotações são tipos especiais declarados com `@interface`, pois eles não quiserem criar uma nova palavra chave para isso. Basta criar um arquivo `Coluna.java` e declarar:

```
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();  
}
```

Default

É possível deixar um valor padrão declarado em algum campo da anotação:

```
int posicao() default 0;
```

E podemos ainda usar outras anotações do Java para anotar essa nossa anotação! Metadados em metadados! Em particular, podemos indicar onde a anotação pode ser usada (*Target*): atributos, classes, métodos etc. E podemos indicar também se a anotação serve apenas em tempo de compilação ou se pode ser usada em tempo de execução também, isso é, de que maneira ela vai ficar retida no arquivo `.class` (*Retention*).

```
@Target(ElementType.METHOD)  
@Retention(RetentionPolicy.RUNTIME)
```

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

9.6 - Lendo anotação com Reflection

Usando então a anotação definida antes, queremos saber quais são as posições de cada atributo definido. Com um for e alguns métodos, conseguimos facilmente:

```
Class<Negocio> classe = Negocio.class;  
for (Method method : classe.getDeclaredMethods()) {  
  
    String atributo = method.getName();  
    int posicao = method.getAnnotation(Coluna.class).posicao();  
  
    System.out.printf("posicao do metodo %s é %d \n", atributo, posicao);  
}
```

A classe `Field` que vimos antes possui métodos que permitem acessar as anotações. O `getAnnotation` recebe a classe da anotação e devolve a instância da anotação correspondente (ou null caso não tenha). Há também o `isAnnotationPresent()` que recebe também a classe da anotação mas devolve true ou false.

9.7 - TableModel com Reflection

O `TableModel` que fizemos anteriormente tinha um construtor que recebia `List<Negocio>`. Para criarmos um `TableModel` genérico, vamos receber uma `List` de qualquer coisa. Essa lista pode ser um `List<Negocio>` ou um `List<Candle>` ou qualquer outra lista. O que precisamos é: descobrir de que classe é a lista para depois percorrer seus métodos procurando por `@Coluna`.

```
public class ReflectionTableModel extends AbstractTableModel {  
  
    private final List lista;  
    private final Class classe;  
  
    public ReflectionTableModel(List lista) {  
        this.lista = lista;  
        this.classe = lista.get(0).getClass();  
    }  
}
```

Generics

A classe acima não usa generics para garantir que qualquer lista pode ser passada. Mas com generics, a forma correta de referenciar a uma lista de qualquer coisa é `List<?>`, por isso o warning. Da mesma forma, a classe deveria ser `Class<?>`.

E, para implementar os métodos de `AbstractTableModel`, não teremos muita dificuldade:

```
public int getRowCount() {
    // devolve a quantidade de elementos na lista
    return lista.size();
}

public int getColumnCount() {
    // percorre os atributos procurando por @Coluna
    int colunas = 0;
    for (Method m : classe.getDeclaredMethods()) {
        if (m.isAnnotationPresent(Coluna.class))
            colunas++;
    }
    return colunas;
}

public Object getValueAt(int row, int column) {
    Object objeto = lista.get(row);
    for (Method m : classe.getDeclaredMethods()) {
        Coluna c = m.getAnnotation(Coluna.class);
        if (c != null && c.posicao() == column) {
            return m.invoke(objeto);
        }
    }
    return null;
}
```

Repare que o código não é muito complexo. Observe também que as chamadas com reflection acima podem lançar exceptions, omitidas no texto mas tratadas no exercício.

9.8 - Exercícios: ReflectionTableModel

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

- 1) Crie a anotação `Coluna` no pacote `br.com.caelum.argentum.ui`. Dica: vá em **New > Annotation** no Eclipse.

```
@Target(ElementType.METHOD)

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

- 2) Anote os getters da classe `Negocio` com `@Coluna`. Repare que ele vai obrigar a você a passar o atributo `posicao`.

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

```
public class Negocio {

    @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();
}

}
```

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

```
1 public class ReflectionTableModel extends AbstractTableModel {
2
3     private List<?> lista;
4     private Class<?> classe;
5
6     public ReflectionTableModel(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 m : classe.getDeclaredMethods()) {
20             if (m.isAnnotationPresent(Coluna.class))
21                 colunas++;
22         }
23         return colunas;
24     }
25
26     @Override
27     public Object getValueAt(int row, int column) {
28         try {
29             Object objeto = lista.get(row);
30             for (Method m : classe.getDeclaredMethods()) {
31                 Coluna c = m.getAnnotation(Coluna.class);
32                 if (c != null && c.posicao() == column) {
33                     return m.invoke(objeto);
34                 }
35             }
36         } catch (Exception e) {
37             e.printStackTrace();
38         }
39         return null;
40     }
41 }
```

```
40     }  
41 }
```

- 4) **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:

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

Rode novamente e deveremos ter a tabela montada dinamicamente.

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

- 5) Alguns frameworks usam bastante reflection para facilitar a criação de suas telas, como o JGoodies, Genesis, entre outros. Pesquise a respeito.

9.9 - Para saber mais: Formatter, printf e String.format

A partir do Java 5 foi inserida a API de `Formatter`. É uma forma bastante robusta de se trabalhar formatação de dados para impressão. Ela é fortemente baseada nas ideias do `scanf` do C.

Basicamente, tem-se uma `String` chamada de *formato* que descreve em uma sintaxe especial o formato de saída dos dados que depois passamos como argumento.

É possível formatar números, definindo por exemplo a quantidade de casas decimais, ou formatar a saída de um `Calendar` usando seus campos, ou uma `String`.

Para imprimirmos diretamente no console, podemos usar o novo método `System.out.printf`. Alguns usos:

```
// usar printf é mais facil que concatenar Strings  
String manoel = "Manoel"; String silva = "Silva";  
System.out.printf("Meu nome é %s e meu sobrenome é %s\n", manoel, silva);  
// 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 03/05/2008
```

Caso precisemos da `String` formatada ao invés de imprimir, podemos usar:

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

9.10 - Exercícios opcionais

- 1) Queremos possibilitar também a customização dos títulos das colunas. Para isso, vamos **adicionar** mais um parâmetro na anotação:

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

E **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)  
// ...
```

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

```
1 @Override  
  
2 public String getColumnName(int column) {  
3     for (Method m : classe.getDeclaredMethods()) {  
4         Coluna c = m.getAnnotation(Coluna.class);  
5         if (c != null && c.posicao() == column) {  
6             return c.nome();  
7         }  
8     }  
9     return null;  
10 }
```

Rode novamente e observe os nomes das colunas na tabela.

- 2) As datas e valores monetários não estão sendo exibidos corretamente. Para solucionar isso, vamos usar formataadores baseados na classe `Formatter` e no método `String.format`, como vimos antes.

- **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), que simplesmente imprime a `String`.

- **Altere** o método `getValueAt` na classe `ReflectionTableModel` para usar o formato de cada coluna na hora de devolver o objeto. É muito simples, **altere apenas** a linha do retorno:

```
public Object getValueAt(int row, int column) {
```

```
try {
    Object objeto = lista.get(row);
    for (Method m : classe.getDeclaredMethods()) {
        Coluna c = m.getAnnotation(Coluna.class);
        if (c != null && c.posicao() == column) {
            return String.format(c.formato(), m.invoke(objeto));
        }
    }
} catch (Exception e) {
    e.printStackTrace();
}
return null;
}
```

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

```
@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.

- 3) Faça uma quarta coluna com o método `getVolume`:

```
@Coluna(posicao=3, formato="R$ %,#.2f", nome="Volume")

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

- 4) Faça testes unitários para o nosso `ReflectionTableModel`.

9.11 - 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 usávamos 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/>

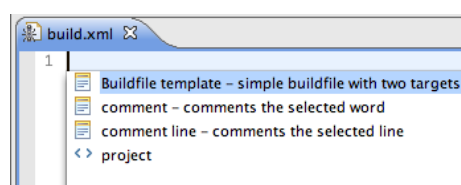
A idéia dele é definir uma série de tarefas, que são invocadas através de tags XML. Um target é composto por uma sequência de tarefas que devem ser executadas. Por exemplo, para compilar precisamos deletar os `.class` antigos, e depois compilar todos os arquivos do diretório X, Y e Z e jogar dentro de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

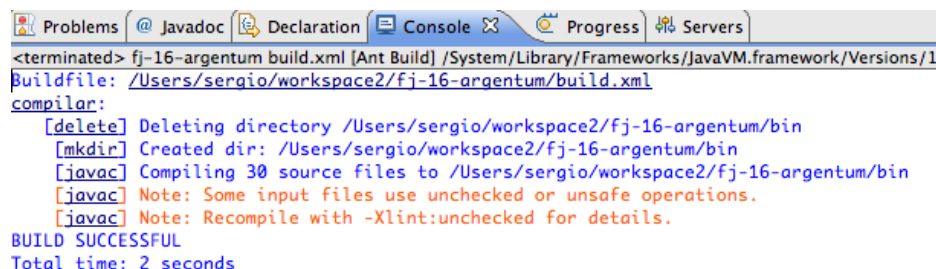
    <target name="compilar">
        <delete dir="bin" />
        <mkdir dir="bin" />

        <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"/>
        </classpath>
    </java>
</target>
```

```
<fileset dir="lib">
  <include name="*.jar" />
</fileset>
</classpath>
</java>
</target>
```

Rode o seu build, escolhendo o target de 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: <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, invoque o gerador do maven:

```
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 15), 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: : argenteum
```

```
Define value for version: 1.0-SNAPSHOT: : (enter)
```

```
Define value for package: : br.com.caelum.argenteum
```

Finalize o processo confirmando os dados. O maven criou no diretório `argenteum` toda uma estrutura de diretório, que não iremos 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>argenteum</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>argenteum</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>4.3.1</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>jfree</groupId>
    <artifactId>jfreechart</artifactId>
    <version>1.0.9</version>
  </dependency>
  <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.11</version>
  </dependency>
  <dependency>
    <groupId>com.thoughtworks.xstream</groupId>
    <artifactId>xstream</artifactId>
    <version>1.3</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>1.5</source>
        <target>1.5</target>
      </configuration>
    </plugin>
  </plugins>
```

```
</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. por e

- 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) Outros relatório interessantes para você testar, adicionando dentro de `reporting/plugins` do nosso pom:

```
<plugin>
```

```
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-jxr-plugin</artifactId>
</plugin>
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-report-plugin</artifactId>
  <version>2.4.2</version>
</plugin>
```

PMD e Javadoc são outros dois relatórios interessantes 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 - Discussão em sala de aula: IDE, ant ou Maven?

Apêndice - Swing Avançado

11.1 - 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 paralelamente a nossa Thread principal para tratar do desenho da interface e da interação do usuário com a mesma (os eventos).

Isso quer dizer que quando criamos listeners para eventos do Swing, esses são executados pela thread do AWT e não pela Thread do `main`. É considerado má prática de programação executar coisas pesadas 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`).

Mas toda atualização nos componentes deve ser feita na Thread do AWT e não na nova Thread em separado, senão poderíamos ter problemas de concorrência! O AWT mantém uma fila de coisas a serem processadas e as executa sem perigo de problemas de concorrência.

Gerenciar essas threads todas e conseguir sincronizar as coisas 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`.

11.2 - SwingWorker

O princípio por trás da classe `SwingWorker` é permitir executar códigos em uma thread nova em paralelo e coisas na thread do AWT. E essa classe traz toda a infra-estrutura necessária para fazer a comunicação entre essas threads de forma segura.

Basicamente, implementamos dois métodos:

- `doInBackground()` - contém o código pesado que deve ser executado na thread em paralelo. Pode devolver algum objeto que será “enviado” para a thread do awt (útil para sincronização);
- `done()` - é executado na thread do AWT logo depois que a outra thread do `doInBackground` termina. Aqui podemos recuperar o objeto devolvido na outra Thread e atualizar coisas da interface.

Temos que escrever uma classe que herda de `SwingWorker` e implementar esses métodos. Na hora da herança, precisamos passar dois argumentos: 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 MeuWorker extends SwingWorker<List<Negocio>, Void> {

    @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 coisas em atributos. O método `done` obtém a lista de negócios a partir da chamada ao método `get`. Isso é feito para garantir thread-safe. Repare também que passamos a tabela como argumento no construtor, já que vamos precisar modificá-la.

Há muitas formas de implementar essa classe nova: ou fazemos uma classe em um novo arquivo, ou criamos uma classe interna ou uma classe anônima. Por legibilidade, vamos criar uma classe em um arquivo novo.

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.

11.3 - Exercícios: resolvendo concorrência com `SwingWorker`

- 1) Na classe `ArgentumUI`, **altere** o método `carregaDados` para receber a lista de negócios como argumento. E **remova** as duas linhas do carregamento do XML de dentro desse método:

```
private void carregarDados(List<Negocio> negocios) {

    // remova as duas linhas abaixo:
    //List<Negocio> negocios = new EscolheXML().escolher();
    //filtraPorData(negocios);

    ReflectionTableModel rtm = new ReflectionTableModel(negocios);
    this.tabela.setModel(rtm);

    // ...
}
```

- 2) Vamos criar uma *classe interna* chamada `CarregaXMLWorker` que tratará do problema da concorrência para nós. **Dentro** da classe `ArgentumUI` mas fora dos métodos, cria uma classe `private`:

```
public class ArgentumUI {  
  
    // metodos, atributos...  
  
    private class CarregaXMLWorker {  
    }  
}
```

Agora, vamos implementar essa classe. Faça ela herdar de `SwingWorker` e implemente os dois métodos:

```
1 private class CarregaXMLWorker extends SwingWorker<List<Negocio>, Void>{  
2  
3     @Override  
4     protected List<Negocio> doInBackground() throws Exception {  
5         List<Negocio> negocios = new EscolheXML().escolher();  
6         filtraPorData(negocios);  
7         return negocios;  
8     }  
9  
10    @Override  
11    protected void done() {  
12        try {  
13            carregarDados(get());  
14        } catch (Exception e) {  
15            e.printStackTrace();  
16        }  
17    }  
18 }
```

- 3) **Altere** a classe anônima que trata o evento do botão de carregar o XML na classe `ArgentumUI` dentro do método `montaBotaoCarregar` para usar nosso worker:

```
private void montaBotaoCarregar() {  
  
    JButton botaoCarregar = new JButton("Carregar XML");  
    botaoCarregar.addActionListener(new ActionListener() {  
        public void actionPerformed(ActionEvent e) {  
            new CarregaXMLWorker().execute();  
        }  
    });  
    painelBotoes.add(botaoCarregar);  
}
```

11.4 - Para saber mais: A parte do JFreeChart

Gerar gráficos pode ser um processo demorado demais para ser executado na thread do AWT. Podemos então colocar esse código para ser executado no nosso `SwingWorker`, no método `doInBackground`. O problema só é que precisamos depois exibir o `JPanel` grafico na tela, mas isso deve ser feito na thread do AWT, ou seja,

no método done.

Portanto, precisamos fazer com que o `doInBackground` devolva, além da lista de negócios como antes, também esse `JPanel` grafico. Para facilitar, vamos mudar o retorno para usar `Object[]` onde a primeira posição terá a lista e a segunda posição terá o grafico. Algo assim:

```
1 public class ProcessaXMLWorker extends SwingWorker<Object[], Void> {
2
3     private final JTable tabela;
4     private final JTabbedPane abas;
5
6     public ProcessaXMLWorker(JTable tabela, JTabbedPane abas) {
7         this.tabela = tabela;
8         this.abas = abas;
9     }
10
11     @Override
12     protected Object[] doInBackground() throws Exception {
13         List<Negocio> negocios = new EscolheXML().escolher();
14
15         CandlestickFactory candlestickFactory = new CandlestickFactory();
16         List<Candle> candles = candlestickFactory.construirCandles(negocios);
17
18         SerieTemporal serie = new SerieTemporal(candles);
19
20         GeradorDeGrafico geradorDeGrafico = new GeradorDeGrafico(serie, 2, serie.getTotal() - 1);
21         geradorDeGrafico.criaGrafico("Média Móvel Simples");
22         geradorDeGrafico.plotaIndicador(new MediaMovelSimples(new IndicadorFechamento()));
23         JPanel grafico = geradorDeGrafico.getPanel();
24
25         return new Object[] {negocios, grafico};
26     }
27
28     @Override
29     protected void done() {
30         try {
31             Object[] retornos = get();
32             List<Negocio> negocios = (List<Negocio>) retornos[0];
33             JPanel grafico = (JPanel) retornos[1];
34
35             NegociosTableModel model = new NegociosTableModel(negocios);
36             tabela.setModel(model);
37
38             abas.setComponentAt(1, grafico);
39         } catch (Exception e) {
40             throw new IllegalStateException(e);
41         }
42     }
43 }
```

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 gambiarras temporárias para ajudar-nos a ver algo.

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 famoso e de longe 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 de duas coisas: o jar do log4j e 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.

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>
```

12.2 - 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`.

Índice Remissivo

Anotações, 119
anotações, 30
AWT, 58

BorderLayout, 105
BoxLayout, 103

Calendar, 10
Candlestick, 4, 7

datas, 10
DateFormat, 73
Design patterns, 86

factory pattern, 10
final, 8
FlowLayout, 102
Formatter, 124

GridBagLayout, 105
GridLayout, 102

JButton, 61
JCheckboxMenuItem, 113
JFileChooser, 60
JFormattedTextField, 110
JFrame, 61
JFreeChart, 89
JMenu, 113
JMenuBar, 113
JMenuItem, 113
JOptionPane, 59
JPanel, 61
JRadioButtonMenuItem, 113
JTabbedPane, 107
JTable, 66
JTextField, 110
JUnit, 28

LaF - Look-and-Feel, 58

Layout Manager, 102

Maven, 129

Negocio, 7

OutputStream, 91

Reflection, 117

static import, 40
Swing, 58
SwingWorker, 135
SWT, 58

Tabelas, 66
Tail, 5
TDD, 32
test driven development, 32
testes de unidade, 28
testes unitários, 28