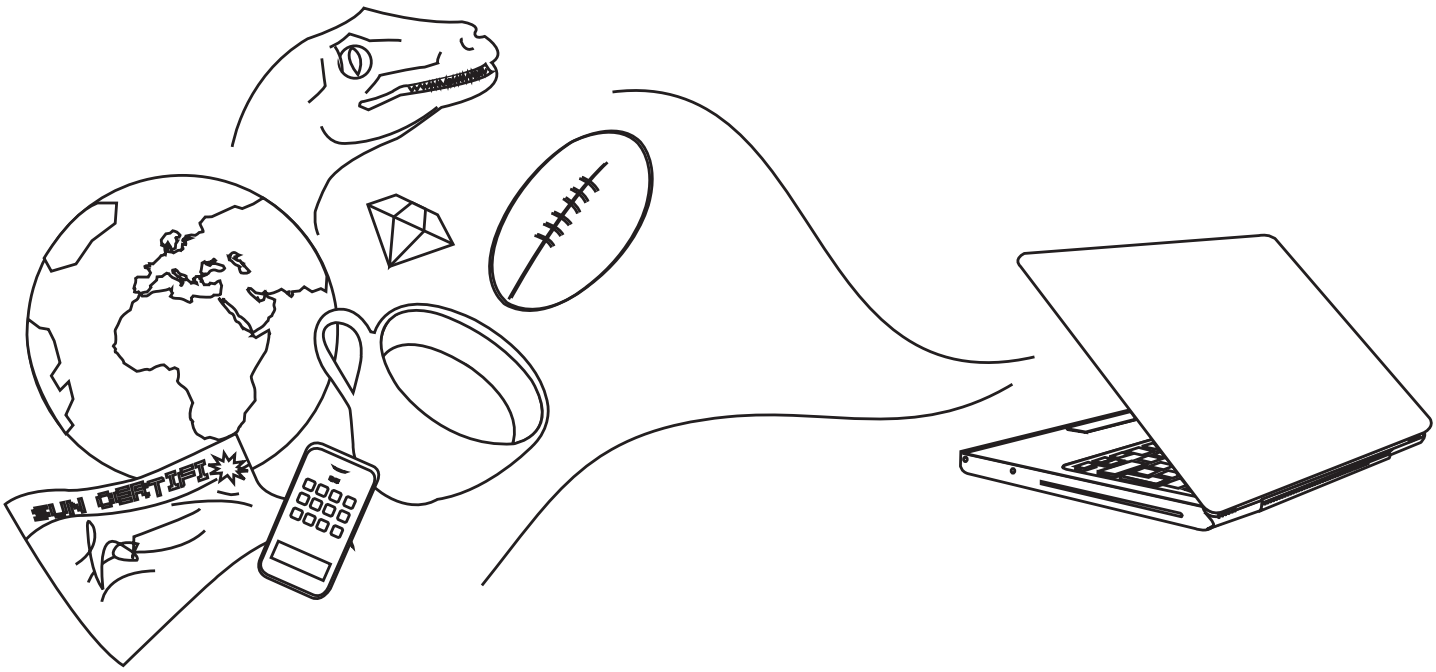
Caelum
Ensino e Inovação




# Cursos Caelum

www.caelum.com.br

design by aef.com.br

Caelum
Ensino e Inovação

# Conheça mais da Caelum.

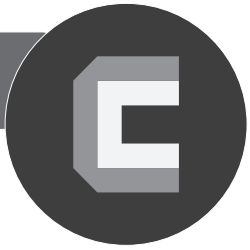
## Cursos Online

www.caelum.com.br/online

## Casa do Código

Livros para o programador

www.casadocodigo.com.br

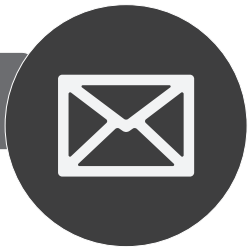


## Blog Caelum

blog.caelum.com.br

## Newsletter

www.caelum.com.br/newsletter

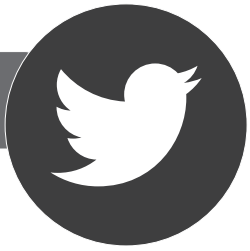


## Facebook

www.facebook.com/caelumbr

## Twitter

twitter.com/caelum



# Sumário

**Versão: 19.1.9**

Capítulo 1

# Um treinamento sobre arquitetura

*"As coisas devem ser feitas o mais simples possível, mas não mais simples que isso."*
– Albert Einstein

## 1.1 Design e Arquitetura?

Joel Spolsky, conhecido arquiteto de software, e um dos fundadores do stackoverflow.com, possui um interessante post em seu blog a respeito dos principais problemas enfrentados na escolha de uma plataforma e de uma arquitetura.

Ele possui duas citações que consideramos fundamental para qualquer arquiteto que vá definir como um projeto será desenvolvido e quais tecnologias serão utilizadas.

*People all over the world are constantly building web applications using .NET, using Java, and using PHP all the time. None of them are failing because of the choice of technology.*

Isto é, nenhuma aplicação falha por causa da escolha da plataforma, seja ela qual for.

*All of these environments are large and complex and you really need at least one architect with serious experience developing for the one you choose, because otherwise you'll do things wrong and wind up with messy code that needs to be restructured.*

Essas plataformas são grandes e complexas, exigindo um conhecimento vasto do arquiteto, caso contrário decisões erradas serão tomadas e resultarão em um código bagunçado que precisa ser reestruturado.

http://www.joelonsoftware.com/items/2006/09/01.html?everything_old_is_new_again

### Nosso objetivo

O objetivo deste treinamento é discutir e abranger o conhecimento do arquiteto, com a finalidade de indagar conceitos pré existentes, quebrar paradigmas, desvendar buzzwords[1] e enxergar vantagens e desvantagens sobre toda e qualquer face da tecnologia em questão.

Isso tudo com o intuito de conhecer muito bem a plataforma Java e todas as suas possibilidades, diminuindo a chance de uma escolha errada pelo arquiteto.

## 1.2 Indo da visão micro para a visão macro e gerenciamento

O treinamento está organizado indo desde a plataforma java e orientação a objetos, passando por diversos Design Patterns durante o curso, discutindo questões de escalabilidade, performance, segurança e também sobre frameworks, sejam eles ORM, de injeção de dependências e diversos outros existentes. Por fim, comparamos as diversas tecnologias existentes para realizar a integração de sistemas, discutindo os pontos positivos e negativos de cada uma e analisando os cenários onde são mais indicados.

Os exercícios disponíveis nessa apostila são distribuídos entre, práticos, teóricos, pesquisas e discussões sobre os resultados alcançados. Todo o código fonte dos exercícios estão disponíveis nos computadores da Caelum e podem ser levados para casa pelo aluno, dessa forma, facilitando a revisão do curso e o processo de aprendizado e fixação do conteúdo.

## 1.3 O livro: Introdução a Arquitetura e Design de Software

Desde que esse treinamento foi criado em 2007, pensávamos em em extrair alguns dos tópicos e discussões ricas que temos com os alunos para um livro.

Esse livro se tornou realidade no final de 2011 e, desde então, é entregue a todos os alunos do curso FJ-91. Ele faz o papel de texto base com o conteúdo principal e esta apostila é um complemento usado no treinamento.

Nesta curta apostila, estão exercícios complementares às discussões do livro e novos tópicos que vamos incluindo com o tempo. O curso é dinâmico, em constante evolução e, naturalmente, sempre mais atual que o livro impresso.

[1] Buzzwords são palavras de efeito, normalmente super valorizadas.

Esperamos que você consulte o livro e leia seus tópicos como complemento às aulas do treinamento. No último apêndice desta apostila, consta uma cópia da errata oficial.

http://www.arquiteturajava.com.br/

Capítulo 2

# A Plataforma Java

*"As coisas devem ser feitas o mais simples possível, mas não mais simples que isso."*
– Albert Einstein

## 2.1 Leituras recomendadas

Para os tópicos deste capítulo, recomendamos a leitura dos seguintes capítulos do livro:

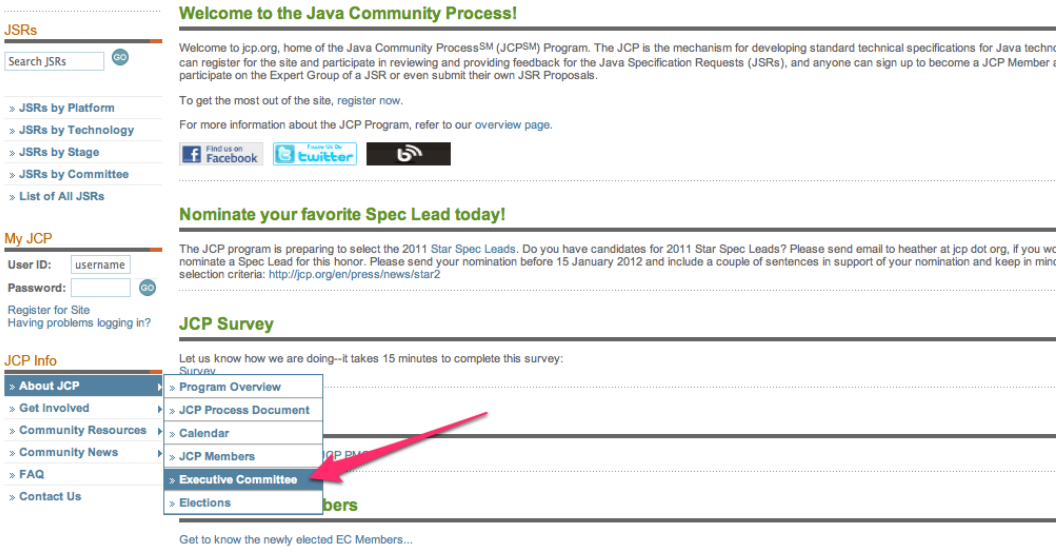
- 1.1 - Java como plataforma, além da linguagem
- 1.2 - Especificações ajudam ou atrapalham?
- 1.3 - Use a linguagem certa para cada problema

## 2.2 Exercícios: A organização do Java, JCP, JSRs e Expert Groups

1) O JCP (Java Community Process) é o processo que rege o desenvolvimento da plataforma Java. Esse processo dita a entrada de novas especificações e evoluções das versões existentes e que envolvam a tecnologia Java.

   Acesse o site do JCP (http://jcp.org) , onde é possível encontrar todas as informações sobre as especificações.

2) No site do JCP, no menu esquerdo chamado *JCP Info*, acesse o menu *About JCP* e clique em Executive Committee para conhecer as empresas e grupos que fazem parte do JCP.

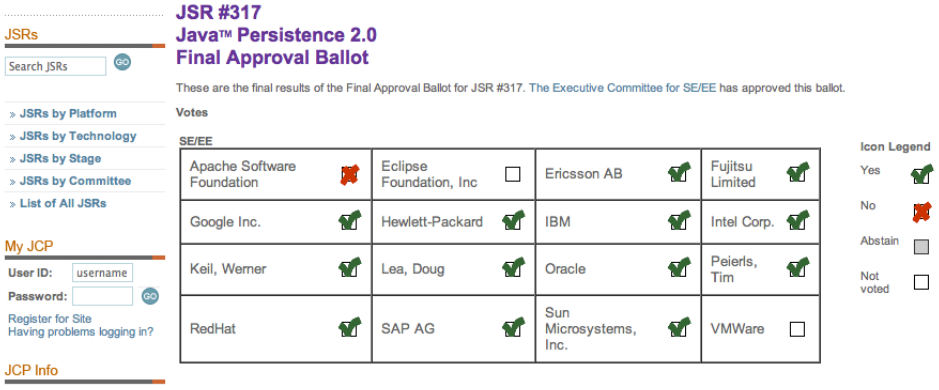



3) No menu esquerdo, existe uma caixa de texto para realizar pesquisas por JSRs. Podemos pesquisar por qualquer uma, por exemplo, a JSR 317, que é a JPA 2.0. Para isso, basta digitar o número 317 e clicar no botão para realizar a pesquisa.

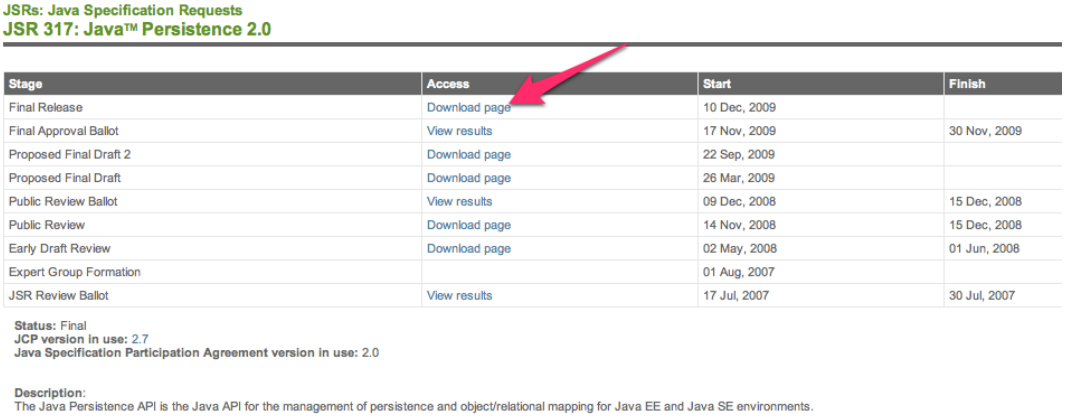

4) Na tela da JSR, é possível descobrir os membros do JCP que estão envolvidos nela. Tanto como líderes, como Expert Group. Também podemos visualizar uma tabela contendo a evolução da especificação, com suas votações, drafts e seu release final.

Para visualizarmos a votação para a aprovação final da especificação, basta clicar no link *View Results*, da linha *Final Approval Ballot*.

Nessa tela, é possível visualizar quem votou a favor e contra a especificação e caso exista, também é possível ver as justificativas para os votos.




5) É possível também realizar o Download da especificação, através da página inicial da JSR. Para isso, basta clicar em *Download Page* na linha *Final Release* na tabela com as datas da especificação.




É possível visualizar dois tipos de especificação: uma para criar uma nova implementação da especificação e outra para avaliação da mesma. Cada uma possui uma abordagem diferente com relação a especificação. Em seguida, basta escolher uma delas, aceitar a licença e clicar no PDF para realizar o Download.

Caso prefira, o arquivo da especificação da JPA 2.0 se encontra disponível nos computadores da Caelum, no diretório `Caelum/91/fj-91-exercicios`.

## 2.3 Exercícios: Outras linguagens na JVM - Scala

1) Importe no Eclipse o projeto `scala`, que está disponível no diretório `Caelum/91/fj-91-exercicio` a partir do seu Desktop. Para isso, basta ir ao menu `File -> Import` e escolha a opção `Existing Project into Workspace`. Na próxima tela, informe o diretório onde o projeto se encontra. Após a importação, abra o novo projeto.

2) Vá ao `src/main/scala` e abra a classe `Produto.scala`. Note a sintaxe diferente da linguagem Scala, se comparado com o Java.

3) a) Abra a classe `Programa.scala`. Note a diferente definição do método `main`, seguindo a sintaxe do Scala. Além disso, outras nuâncias possíveis de se observar na sintaxe, são a ausência de `;` (ponto e vírgula) ao final das linhas.

   b) Repare na linha 7, que não é definido o tipo do lado esquerdo da atribuição. Isso acontece pelo fato do Scala inferir o tipo da variável a partir do objeto que foi instanciado. Isso é chamado de *type inference* (inferência de tipos).

   c) Os métodos `nome`, `quantidade` e `preco` foram definidos na classe `Produto` a partir do momento que usamos as palavras chaves `var` e `val`. Esses métodos são semelhantes aos `getters` do Java e servem para recuperar o valor de um atributo. Eles são conhecidos em outras linguagens como `propriedades`, como é o caso do C#.

   d) Na linha 15, é utilizado o método `quantidade =` no objeto `p`. Ele recebe o valor `120`. Na linguagem Scala, em algumas situações, os parênteses são opcionais para a passagem de parâmetro, portanto nessa situação, a linha poderia ser escrita como:

   ```
   p.quantidade =(120)
   ```

   e) Na linha 19, é criada uma instância da classe `AtualizadoDeProduto`, que é uma classe implementada em Java. É possível vê-la em `src/main/java`.

   f) Para executar essa a classe `Programa.scala`, abra o arquivo `build.xml` disponibilizado no projeto e na view `Outline`, execute a tarefa `roda-programa` clicando com o botão direito sobre ela e em seguida `Run as -> Ant Build`.

4) Umas das características de Scala, ser uma linguagem que mistura a programação orientada a objetos com o paradigma funcional. Para visualizar um pouco desse paradigma, existe a classe `ProgramaFuncional.scala`. Abra-a.

   a) No início do método `main`, são criados 4 produtos e incializado uma lista com eles.

   b) Na linha 14, é realizado o `foreach` na lista, passando como parâmetro a função `println`. Dessa maneira, os 4 produtos serão exibidos no console.

c) Em seguida, na linha 17, é realizado outra invocação do método `foreach`, dessa vez, enviando como parâmetro uma função que receba um produto `p` e realize um `println` nesse produto, exibindo seu nome.

d) Por fim, na linha 20, é introduzido o método `filter` que dada uma condição, retorna uma **nova** lista contendo os elementos que a satisfaça. E em seguida, essa nova lista tem seus elementos impressos no console.

e) Execute o programa através da tarefa `funcional` existente no `build.xml`.

5) (opcional) Melhore a saída do console, implementando o método toString() na classe `Produto` para mostrar o nome e o preço dele. O código do método será similar a:

```
override def toString = {
    nome + " - " + preco
}
```

## 2.4 Para saber mais: Como aprender Scala

Muitos desenvolvedores se interessam por Scala, devido ao fato dela permitir a escrita de códigos mais enxuto e mesmo assim continuar sendo expressiva e realizando as mesmas tarefas que seriam possíveis com uma linguagem como o Java.

No entanto, é muito comum esbarrar em uma possível barreira na curva de aprendizado, pois aprender Scala pode envolver também aprender um novo paradigma, o funcional. Recomendamos os seguintes livros e blogs ao aluno que queira aprender essa linguagem e o paradigma funcional:

- *Programming Scala*, escrito por Martin Odersky, Lex Spoon e Bill Venners;
- *Structure and Interpretation of Computer Programs*, escrito por Gerald Jay Sussman;
- Existem alguns artigos escritos no blog da Caelum, que podem ser encontrados em http://blog.caelum.com.br/tag/scala/ e http://blog.caelum.com.br/tag/funcional/

## 2.5 Exercícios: Outras linguagens na JVM - Javascript

1) Importe no Eclipse o projeto `javascript`, que está disponível no diretório `Caelum/91/fj-91-exercicios` a partir do seu Desktop. Para isso, basta ir ao menu `File -> Import` e escolha a opção `Existing Project into Workspace`. Na próxima tela, informe o diretório onde o projeto se encontra. Após a importação, abra o novo projeto.

2) Abra a classe `Programa` disponível em `src/main/java`. Na linha 14, é utilizado o método `eval` da classe ScriptEngine para converter um Array em JavaScript num Array do Java.

Em seguida, na mesma classe Java, são mostradas as informações existentes no Array criado via JavaScript. Execute a classe `Programa` e veja as saídas no console.

3) Abra a classe `ProgramaSwing`. Note que ele apenas interpreta o conteúdo do arquivo `swing.js`, que por sua vez, utiliza a API de Swing do Java para exibir uma janela com um botão, ou seja, conseguimos utilizar classes e componentes Java em um arquivo JavaScript.

   Execute a classe `ProgramaSwing` e clique no botão da janela e visualize a saída no console do Eclipse.

4) Na classe `ValidacaoJavascript`, o arquivo `validacoes.js` é lido e interpretado pelo Nashorn. Esse arquivo possui duas funções declaradas: `comecaComMaiuscula` e `possuiMinimoDeCaracteres`, autoexplicativas.

   Na linha 23, é invocado é utilizado o método `invokeFunction`, da interface **Invocable**. Perceba que o uso é bem direto, dizemos o nome da função javascript que queremos invocar e depois passamos os argumentos.

   Por fim, é feito o teste com a invocação da função `possuiTamanhoMinimoDeCaracteres`. Da mesma maneira que a invocação anterior, o resultado, um booleano, é impresso no console.

Capítulo 3

# Como aproveitar ao máximo o que a JVM oferece

## 3.1 Leituras recomendadas

Para os tópicos deste capítulo, recomendamos a leitura dos seguintes capítulos do livro:

- 2.1 - Princípios de garbage collection
- 2.2 - Não dependa do gerenciamento de memória
- 2.3 - JIT Compiler: compilação em tempo de execução
- 2.4 - Carregamento de classes e classloader hell

## 3.2 Exercícios: Análise de performance de uma aplicação Java com o JIT

1) Copie para um diretório de sua preferência o projeto `jit`, existente na pasta `Caelum` que está disponível no seu Desktop. Em seguida, acesse esse diretório através do Terminal.

2) a) Execute o programa `fibonacci`, escrito em linguagem C, no terminal para visualizar o seu tempo de execução no cálculo de Fibonnaci de 5. Para isso, o seguinte comando deverá ser realizado:

   ```
   time ./fibonacci 5
   ```

   Execute esse comando 3 vezes e anote a média do tempo de execução exibido na medida `real`.

b) Agora, execute o mesmo programa, porém em Java, para calcular Fibonnaci de 5.

```
time java Fibonacci 5
```

Execute esse comando 3 vezes e anote a média do tempo de execução exibido na medida `real`.

c) Descubra quantas vezes o programa em C foi mais rápido que o programa Java. Basta dividir o real time de um pelo do outro.

3) Repita os passos do item anterior, testando em C e em Java o tempo de execução para Fibonacci de 20, 30, 40 e 45. Calcule quantas vezes um é mais lento ou mais rápido que o outro.

4) Agora, descubra se sua JVM está executando em modo `client` ou modo `server`. Para isso, pegue os resultados de Fibonacci de 40 e compare-o com a execução dos seguintes comandos:

```
time java -server Fibonacci 40

time java -client Fibonacci 40
```

O comando que gerar o resultado dado na primeira execução indicará qual era o modo em que a JVM estava sendo executado. Quem escolhe essas definições para nós, é um recurso da JVM chamado *Ergonomics*, que leva em consideração os recursos que nosso computador tem disponíveis.

5) A provável queda na diferença do tempo de execução entre o programa em C e em Java, tem como um dos fatores as otimizações que a JVM aplica ao programa em execução. Essas otimizações são realizadas pelo JIT - Just in Time Compiler -, que realiza algumas mudanças em nosso programa durante sua execução para que ele seja mais otimizado.

Essas otimizações são feitas nos métodos mais demandados, ou seja, nos mais invocados pelo seu programa. Ao definirmos o modo `server`, estamos dizendo que, quando o método receber 10000 invocações, aí sim ele será otimizado. Enquanto o modo `client` define em 1500 invocações.

Podemos visualizar os métodos otimizados, executando o programa com a flag `-XX:+PrintCompilation`:

```
time java -server -XX:+PrintCompilation Fibonacci 40
```

A saída no console mostrará os métodos que são otimizados pela JVM.

6) Caso queiramos um número de invocações entre 10000 (`server`) e 1500 (`client`), podemos utilizar a flag `-XX:CompileThreshold=<um_numero>`:

```
time java -XX:CompileThreshold=3000 \
-XX:+TieredCompilation -XX:+PrintCompilation Fibonacci 40
```

Teste com 3000 invocações, veja se há diferença nos métodos otimizados.

7) Uma tendência muito comum é pensarmos que quanto mais otimizações acontecerem, mais rápido será o programa. No entanto, podemos executar o programa do Fibonacci de 40, definindo o `-XX:CompileThreshold` para 1, ou seja, a cada invocação de método, a otimização acontecerá.

   Execute e analise os resultados.

8) Também é possível desabilitar totalmente o JIT, bastando colocar um número extremamente alto no `CompileThreshold`, ou utilizando a Flag `-Xint`. Para testar, execute o seguinte comando:

   ```
   time java -Xint -XX:+PrintCompilation Fibonacci 40
   ```

   Ao executar essa linha de comando, repare que nada é impresso no console, ou seja, não existe nenhuma otimização sendo feita.

> **Variações**
>
> **É importante** ter em mente que os resultados desses exercícios podem variar de acordo com o computador em uso e suas características.

## 3.3 Exercícios: Garbage Collector e tuning de memória

1) Copie o projeto `gc`, existente em `Caelum/91/fj-91-exercicios` que é acessível a partir do seu Desktop, para o seu um diretório de sua preferência e acesse-o através do terminal.

2) Podemos executar o programa `EstressaGC` e visualizar as execuções do Garbage Collector introduzindo a flag `-verbose:gc`:

   ```
   java -verbose:gc EstressaGC
   ```

   A saída do console exibirá um resultado similar ao seguinte:

   ```
   [GC 74549K->72108K(92416K), 0.0988019 secs]
   [GC 92132K->90583K(109996K), 0.0972691 secs]
   [Full GC 109490K->3922K(109996K), 0.1050284 secs]
   ```

   A identificação GC, indica que aconteceu um *minor GC*, enquanto que a indicação Full GC, representa um GC completo sendo executado.

   Os números indicados em seguida representam o seguinte:

   ```
   GC memoria_antes -> memoria_depois (total_livre_para_objetos), tempo_de_execucao
   ```

3) Podemos definir o tamanho da memória heap da JVM, através das flags `-Xmx` e `-Xms`, indicando o tamanho da memória que vamos querer. Teste a execução com um número mais alto de memória também com um número baixo, como por exemplo:

```
java -verbose:gc -Xmx512M -Xms512M EstressaGC

java -verbose:gc -Xmx30M -Xms30M EstressaGC
```

Analise a diferença entre os resultados das execuções dos Garbage Collectors. Qual demora mais? Qual é mais rápido?

4) É possível definir o tamanho da geração Young da memória de duas maneiras. Uma é definir um número absoluto da memória para ela, através da flag `-XX:NewSize`:

```
java -verbose:gc -Xmx100M -Xms100M -XX:NewSize=80M EstressaGC
```

A outra possibilidade é definir a proporção do tamanho da memória heap que a geração Young utilizará. Em máquinas que estão executando em modo `server`, essa proporção será de 1/3, ou seja, um terço da memória é destinado à Young. Podemos definir essa proporção através da flag `-XX:NewRatio`, como em:

```
java -verbose:gc -Xmx100M -Xms100M -XX:NewRatio=2 EstressaGC
```

5) Vamos configurar o VisualVM, para termos uma visualização mais interessante do que acontece com a JVM e como a nossa aplicação consome a memória.

   a) Descompacte o VisualVM, que está disponível dentro do diretório `Caelum/91/`, em uma pasta de sua preferência.

   b) Execute o programa através do executável disponível dentro do diretório `bin`.

   c) No VisualVM, vá ao menu `Tools -> Plugins` e na tela que é aberta, abra a opção `Downloaded`. Em seguida, clique em `Add Plugins...` e escolha o VisualGC, que está disponível na pasta `Desktop/Caelum/91/`.

   d) Clique em `Install`.

6) No terminal, execute o programa `EstressaGCComPausa`.

```
java EstressaGCComPausa
```

Nesse momento, será exibida a seguinte mensagem:

```
Executando! Agora abra no VisualVM e volte aqui pra liberar a execucao
```

Agora, você deve ir ao VisualVM onde aparecerá a execução do programa `EstressaGCComPausa`. Dê um duplo clique sobre ele e em seguida vá à opção `VisualGC`. Observe a visualização da memória e todas as suas gerações.

Nesse instante, volte ao terminal, e libere a execução do programa com pressionando *Enter* e volte para o VisualVM, para visualizar as mudanças na memória da sua aplicação.

7) Veja o conteúdo das outras abas, como, por exemplo, a *Profiler* e a aba *Monitor*. Analise as informações que elas lhe oferecem.

8) Realize outras execuções do programa, diminuindo o tamanho da memória, até forçar um `OutOfMemoryError: java heap space`.

## 3.4 Exercícios: Como funcionam os Classloaders e o Classloader Hell

1) a) Importe o projeto `classloader` que está disponível em `Desktop/Caelum/91/fj-91-exercicios` para o Eclipse e abra-o.

   b) Observe a classe `ObjetoTeste` que se encontra no pacote `br.com.caelum.fj91.classloader.teste` e, em seguida, abra a classe `TestaComparaClasses`.

   c) Essa classe, em seu método `main`, carrega a classe `ObjetoTeste` através de um Classloader customizado e, em seguida, carrega novamente a mesma classe pelo Classloader da aplicação.

   d) Ao executá-la, vemos a saída do console indicando o nome dos Classloaders de onde elas foram carregadas e a mensagem indicando que as classes são diferentes.

```
ClassLoader da 1a classe: java.net.URLClassLoader@1d5ee671
ClassLoader da 2a classe: sun.misc.Launcher$AppClassLoader@53372a1a
Classes são iguais? false
```

   e) (opcional) Na linha 24, onde está a instanciação do objeto da classe carregada pelo Classloader customizado, faça um cast para `ObjetoTeste`. Dessa forma, a linha 24 ficará igual a:

```
ObjetoTeste o = (ObjetoTeste) classe.newInstance();
```

   Executando-a, teremos uma exceção no console, no caso, uma `ClassCastException`.

```
Exception in thread "main" java.lang.ClassCastException:
            br.com.caelum.fj91.classloader.teste.ObjetoTeste
             cannot be cast to br.com.caelum.fj91.classloader.teste.ObjetoTest
    at br.com.caelum.fj91.classloader.teste.TesteComparaClasses.main
    (TesteComparaClasses.java:24)
```

2) a) Importe o projeto `classloader-web` no seu Eclipse e abra-o.

   b) Configure o Tomcat 6.0.20, que também está disponível no diretório `Caelum/91/fj-91-exercicios`, no Eclipse e associe o projeto recém importado à ele.

   c) Inicie o servidor e acesse o endereço http://localhost:8080/classloader-web no seu navegador.

d) Um erro acontecerá, indicando que um método não foi encontrado, no caso um `NoSuchMethodError`.

e) O problema é que estamos usando o Tomcat 6.0.20, que utiliza a Servlets na versão 2.5, e, no `WEB-INF/lib` do nosso projeto, colocamos o jar das Servlets na versão 3.0. Como as Servlets são carregadas pelo Container, a versão que é utilizada é a 2.5, ou seja, os métodos novos existentes na nova versão são ignorados.

f) Podemos corrigir parando de utilizar o método que não existe. Para isso, comente a linha 20 da Servlet `MostraDados`. Essa linha possui o seguinte conteúdo:

```
out.printf("Response headers: %s", response.getHeaderNames());
```

O método `getHeaderNames()` em `response` é justamente o método que não existe na versão 2.5 de Servlets.

g) Ao realizar a alteração, não inicie o servidor, deixe que o *Hot Deploy* seja feito pelo Tomcat e pelo Eclipse.

h) Assim que o deploy for feito, acesse novamente a página e veja que agora a saída é mostrada no navegador.

i) Realize mais uma alteração na classe mostra dados, por exemplo, para mostrar o seu nome.

```
out.println("João");
```

j) Deixe o servidor realizar o *Hot Deploy* e acesse a página novamente. Veja que tudo está funcionando.

k) Repita esse ciclo por algumas vezes, até o momento em que seu servidor parará de responder. Nesse instante, ao acessar uma página no servidor ou visualizando o console do mesmo, é possível visualizar o erro: `OutOfMemoryError: PermGen space`.

OBS: Para simular o erro no Java 8 devemos limitar o tamanho do Metaspace, e podemos fazer isso adicionando a flag `-XX:MaxMetaspaceSize=64m` nas configurações do Tomcat: (Aba server -> duplo clique no Tomcat -> open launch config. -> aba arguments -> vm arguments)

l) A geração *Perm* da memória acabou lotando e o Garbage Collector não conseguiu liberar nada de lá. O motivo para que ela acabasse lotada é o carregamento do driver JDBC, na classe `Inicializa`. O `DriverManager`, que está no Bootstrap Classloader mantém uma referência para o driver JDBC, que está no Classloader da sua aplicação e essa referência nunca é liberada. Dessa forma, quando o *Hot Deploy* acontece, todas as classes que foram criadas anteriormente **não** são liberadas.

m) A correção é liberar a referência do `DriverManager` para o driver. Para isso, na classe `Inicializa`, descomente o conteúdo do método `contextDestroyed`, que é executado quando a aplicação está sendo finalizada.

Capítulo 4

# Design e Orientação à Objetos

## 4.1 Leituras recomendadas

Para os tópicos deste capítulo, recomendamos a leitura dos seguintes capítulos do livro:

- 3.1 - Programe voltado à interface, não à implementação
- 3.2 - Componha comportamentos
- 3.3 - Evite herança, favoreça composição
- 3.4 - Favoreça imutabilidade e simplicidade
- 3.5 - Cuidado com o modelo anêmico
- 3.6 - Considere Domain-Driven Design
- 5.1 - Testes de sistema e aceitação
- 5.2 - Teste de unidade, TDD e design de código
- 5.3 - Testando a integração entre sistemas
- 5.4 - Feedback através de integração contínua

Além disso, o livro *Design Patterns* do GoF é bastante recomendado.

## 4.2 Exercícios: Boa prática de orientação à objetos - Encapsulamento

1) Importe em seu Eclipse, o projeto `banco0`, disponível em `Caelum/91/fj-91-exercicios` e abra-o.

2) Abra a classe `TestaNovaConta` que se encontra no pacote `br.com.caelum.fj91.banco.teste` e note que seu código deixa várias responsabilidades expostas para os desenvolvedores que terão que manipular essa classe. É necessário que eles saibam que as chamadas para abrir uma nova conta deve ser feita em determinada sequência, por exemplo.

3) O primeiro passo encapsular esse comportamento, é colocarmos toda essa lógica de criação de conta bancária na classe `Banco`. Para isso, selecione o trecho de código que vai da linha 19 até a linha 32, na classe `TestaNovaConta` e em seguida, vá ao menu `Refactor -> Extract Method`. Isso fará com que um novo método, contendo o comportamento selecionado seja criado. Chame-o de `abreNovaConta`.

4) Dê um duplo clique no nome do método `abreNovaConta`, para que ele fique selecionado e em seguida, vá novamente ao menu `Refactor`, mas agora, escolha a opção `Move` e confirme que o método deve ser movido para a classe `Banco`.

5) Vá a classe `Dao` que se encontra no pacote `br.com.caelum.fj91.banco.persistencia`. Note que todos os métodos lançam `SQLException`, que é uma exceção específica de quando se utiliza JDBC, ou seja, esse detalhe de implementação está vazando do nosso `Dao` para as classes que a utilizam.

   Uma alternativa melhor, é lançar uma exceção mais genérica, como por exemplo, uma `RuntimeException` ou uma outra exceção, como uma `PersistenciaException`, mas que não seja algo específico de uma tecnologia.

   Um motivo para isso, é que no momento em que quisermos trocar de implementação de persistência, por exemplo, substituindo o JDBC pela JPA (que não lança `SQLException`), não precisarmos mudar nada no código que usará o nosso `Dao`.

6) Abra a classe `Conta` no pacote `br.com.caelum.fj91.banco.modelo` e note que ela é uma classe contendo vários getters e setters, sendo que alguns deles, são desnecessários, como o `setDataAbertura` e o `setNumero`. Aperte CTRL+3 e digite gcuf (as iniciais de *Generate Constructor Using Fields*). Na janela que abriu, selecione apenas os atributos `dataAbertura` e `numero`.

7) Agora que usamos o construtor para receber esses valores, podemos remover os dois setters, o de `dataAbertura` e o de `numero`. Porém, ao removê-los, as classes `Programa` e `Banco` param de compilar. Nessas classes, basta trocar as invocações dos setters, por uma invocação ao novo contrutor criado.

8) Abra a classe `OperacoesBancarias` que se encontra no pacote `br.com.caelum.fj91.banco.logica`. Note que ela é uma classe que não contém dados nenhum, apenas métodos, que agem sobre as informações das contas. Podemos movê-los para lá, assim, teremos um modelo mais rico de objetos.

   Para isso, faça o `Move` dos métodos para a classe `Conta` e ao final, a `OperacoesBancarias` vai deixar de ter uma razão para existir, pois não haverá nenhum código dentro dela. Nesse momento, ela pode ser excluída.

9) Por fim, uma última mudança que podemos fazer, para mantermos nosso código mais encapsulado, é remover o `setSaldo` existente na classe `Conta`. Nesse instante, ele é apenas utilizado dentro da própria classe. Para removê-lo e ainda assim, mantermos o código da atribuição do saldo onde for necessário, vamos fazer o *Inline* do método. Para isso, basta clicar com o botão direito na declaração do `setSaldo` e ir em `Refactor -> Inline`.

10) Nesse instante, temos um código mais encapsulado e que não expõe os detalhes internos de implementação para as classes clientes, ou seja, para as classes que vão utilizá-las.

## 4.3 Exercícios: Cuidados com a herança e a composição com alternativa

1) Importe em seu Eclipse, o projeto `banco1`, disponível em `Caelum/91/fj-91-exercicios` e abra-o.

2) a) Abra a classe `Conta` e note ao final dela, os dois métodos `aplica`, um recebendo `DescontaJuros` e o outro recebendo `TributoUsoDoChequeEspecial`.

      Ambos os métodos são utilizados na classe `TestaTributos`, que realiza a tributação de uma `Conta`, através dos dois tipos de tributos existentes.

   b) Ao precisarmos introduzir um novo tipo de tributo, precisamos além de criar a classe especializada com o cálculo do novo tributo, também introduzir o novo método `aplica` à classe `Conta`, ou seja, uma simples alteração, faz com que várias classes sejam modificadas. Isso é um sinal de alto acoplamento. Vamos diminuir esse acoplamento através de algumas técnicas.

   c) Abra a classe `TributoUsoDoChequeEspecial` e extraia uma interface para ela, através do menu `Refactor -> Extract Interface`. Chame a nova interface de `Tributo` e indique que o método `calculaDesconto` deve ir para a interface.

   d) Vá até a classe `DescontaJuros` e faça-a implementar a interface `Tributo`.

   e) Agora, na classe `Conta`, faça com que um dos métodos `aplica` receba como parâmetro a interface `Tributo` ao invés da implementação. Enquanto que o outro método `aplica` pode ser

apagado.

f) Repare que a classe `TestaTributos` não parou de funcionar e agora utilizamos apenas uma implementação do método `aplica`, mas que funciona para qualquer um dos `Tributo`s. Atravé do polimorfismo, ganhamos maior flexibilidade e desacoplamento no código. Nesse caso, introduzimos o Design Pattern chamado Strategy.

3) a) A classe `Contas` representa uma classe que pode guardar dentro dela várias contas. Dessa maneira, é interessante que ela tenha comportamentos para adicionar elementos e descobrir a quantidade de elementos já cadastrados.

b) Todos esses comportamentos, já existem na classe `HashSet`, portanto, fizemos com que ela herdasse dessa classe.

c) O método `add` e `addAll` reescritos de `HashSet`, introduzem um contador, para saber se as contas adicionadas são do primeiro ou do segundo semestre e por fim, **invocam o método respectivo da classe pai (`HashSet`)**.

d) Abrindo a classe `TestaContas`, são criadas 7 contas, sendo 3 com datas do primeiro semestre e 4 no segundo semestre e adicionadas, através do método `addAll` à instância de `Contas`. Em seguida o método `size()`, `getTotalContasPrimeiroSemestre` e `getTotalContasSegundoSemestre`, respectivamente, são chamados. Porém, ao executar essa classe, a saída no console será:

```
Total de contas: 7
Contas do 1o semestre: 6
Contas do 2o semestre: 8
```

Repare que o resultado é totalmente inconsistente. Como temos 7 contas sendo que 6 são do primeiro semestre e 8 no segundo semestre? Esse cálculo não bate. Está completamente errado.

e) O problema é que o método `addAll`, que é invocado na classe `TestaContas` faz a execução do contador. Porém, ao delegar para o `addAll` de `HashSet`, esse método chama o `add`. Como ele foi reescrito para fazer o contador, passamos pela segunda vez nas condições e incrementamos os valores novamente.

Nesse caso **uma possível solução** é comentar a implementação do método `addAll`.

f) Note que tivemos que saber como a implementação de uma classe funciona, para utiliza-la corretamente. Esse é um problema muito sutil, porém frequente, que acontece ao trabalhar com herança. **Aconteceu a quebra do encapsulamento**.

4) Outro problema existente na classe `Contas` é que devido ao fato dela herdar de `HashSet`, podemos utilizar todos os métodos existentes nela, como por exemplo, o `clear`. Ao utilizarmos a herança, **aumentamos a interface de uso** da nossa aplicação. Vamos resolver esse problema, deixando de utilizar a herança.

a) Remova a herança e faça com que a classe `Contas` tenha um atributo chamado `contas` do tipo `HashSet`.

```
HashSet<Conta> contas = new HashSet<Conta>();
```

b) No método `add` e `addAll` (caso ele esteja comentado, descomente-o), ao invés de chamar o método da classe pai, por exemplo, `super.addAll`, chame o método do atributo `contas`.

c) No Eclipse, vá ao menu `Source -> Generate Delegate Methods` e escolha o método `size()`.

d) Note que agora, não conseguimos invocar mais o método `clear` em um objeto do tipo `Contas`. Portanto, **conseguimos reaproveitar o código, sem aumentar desnecessariamente a interface de uso e de quebra, não precisamos saber nada sobre a implementação interna do HashSet**. Ou seja, ganhamos **desacoplamento**.

## 4.4 Exercícios: Teste de unidade e o acoplamento semântico

1) Importe em seu Eclipse, o projeto `banco2`, disponível em `Caelum/91/fj-91-exercicios`, abra-o e associe o mesmo ao Tomcat.

2) Inicie o servidor, e manipule a aplicação, disponível em http://localhost:8080/banco2.

3) Abra a classe `ContaTest` disponível em `src/test/java` e execute os testes de unidade. Note que todos passam.

4) Simulando o trabalho com vários desenvolvedores, vá até a classe `Conta` e introduza um bug no método `saca`, por exemplo, troque o método `subtract` para `add` e execute novamente todos os testes, que agora terão falhas.

   Repare que a maneira que tivemos de descobrir que o código não funciona mais corretamente, foi executar novamente todos os testes.

   **Volte o método `saca` para utilizar o método `subtract`.**

5) No pacote `br.com.caelum.fj91.banco.tributacao` em `src/main/test`, abra o `DescontaJurosTest`.

6) Devido ao fato dos comportamentos do `aplica` da classe `DescontaJuros` depender do comportamento de outras classes, como a `Conta`, nós simulamos as interações com as dependências, através de um objeto falso, um *Mock*.

   Para facilitar o trabalho de criar esse objeto falso, que responde às interações da forma que quisermos, utilizamos a biblioteca Mockito.

7) Execute os testes da classe `DescontaJurosTest`.

8) No pacote `br.com.caelum.fj91.banco.integracao`, abra a classe `CadastroDeContaTest`. Essa classe, utiliza a biblioteca Selenium, para testar a aplicação web, simulando as interações com a página, entradas de inputs de formulário, cliques de botão e assim por diante. Dessa maneira, conseguimos saber se a aplicação responde corretamente às manipulações sofridas na tela.

9) Execute o teste `CadastroDeContaTest` e note a rapidez com que a execução é feita. **Lembre-se de que o servidor deverá estar executando**. Caso prefira visualizar a execução passo a passo, execute o teste em modo de debug (lembre-se de adicionar um breakpoint no começo do teste).

## 4.5 Exercícios: Código mais expressivo e o padrão Builder

1) Importe em seu Eclipse, o projeto `banco3`, disponível em `Caelum/91/fj-91-exercicios`.

2) Abra a classe `TransferenciaSimplesAgendada` e repare que todos os valores são recebidos no construtor. Para usá-la, temos a classe `AgendaNovaTransferenciaAction`. Note nessa classe, o uso do construtor de `TransferenciaSimplesAgendada`, onde existem parâmetros sendo passados na ordem errada.

   No entanto, tudo parece normal, não há erro de compilação. Apenas uma má escolha de nomes de variáveis que nos confunde na hora de usar o construtor. A conta de destino está invertida com a conta de origem. Corrija.

3) Vá até a classe `AgendaNovaTransferenciaRecorrenteAction` e abra-a. Note a definição dos valores para o objeto da classe `TransferenciaRecorrenteAgendada` através das diversas chamadas a métodos setters. Vamos melhorar esse código, para deixá-lo como:

```
TransaferenciaRecorrenteAgendada t = new TransferenciaRecorrenteAgendada()
    .durante(periodo)
    .comValor(400)
    .daConta(c1)
    .paraConta(c2)
    .noDia(10);
```

4) O primeiro passo, é fazer com que todos os setters devolvam uma referência para `this`, ou seja, um objeto do tipo `TransferenciaRecorrenteAgendada`. Por exemplo, o método `setDia` ficará como:

```
public TransferenciaRecorrenteAgendada setDia(Integer dia) {
    this.dia = dia;
    return this;
}
```

Faça o mesmo processo para os outros métodos.

5) Agora, renomeie os métodos para um nome que não utilize o prefixo `set`, como por exemplo, `setPeriodo` para `durante`, `setDestino` para `paraConta` e assim por diante.

6) Encadeie as chamadas como no código a seguir (fazendo ajustes nos nomes de métodos, caso necessário):

```
TransaferenciaRecorrenteAgendada t = new TransferenciaRecorrenteAgendada()
    .durante(periodo)
    .comValor(400)
    .daConta(c1)
    .paraConta(c2)
    .noDia(10);
```

7) (opcional) Abra a classe `TestaBuilder` e veja o uso da classe `ContaBuilder`. Analise seu código e sua implementação. Preste atenção no uso das classes internas e como elas forçam uma sequência de chamadas.

## 4.6 Exercícios opcionais: DSLs em Java e em outras linguagens

1) Um exemplo muito famoso de DSL em Java, é a biblioteca Joda Time, que é uma alternativa a classe `Calendar` e a API de datas disponível no Java. Acesse o site http://joda-time.sourceforge.net/ e analise a expressividade dos exemplos disponíveis.

2) Importe em seu Eclipse, o projeto `dsls`, disponível em `Caelum/91/fj-91-exercicios` e abra-o.

3) a) Abra o arquivo `dsl.rb` e observe o conteúdo do mesmo a partir da linha 40. Note que os métodos `reais` e `dolares` estão sendo invocados em um objeto que representa um número, que no caso do Ruby, é um objeto do tipo `Fixnum`.

   b) Na linha 1, estamos *abrindo* a classe `Fixnum` e adicionando novos métodos. Ambos os métodos devolvem um objeto do tipo `Dinheiro`, que recebe em seu construtor o tipo da moeda que vai se trabalhar e o próprio número.

   c) Na classe `Dinheiro` é definido o método `+` que recebe como parâmetro outro `Dinheiro` e o cálculo da cotação entre as moedas é feito, para se chegar ao resultado final.

4) a) Abra o pacote `br.com.caelum.fj91.calculodata.dsl` e entre no arquivo `CalculoData.scala`.

   b) Observe dentro do método `main`, o código que é utilizado para realizar cálculos com as datas. Parece que é um simples texto e não um código que está sendo compilado.

c) Abra a classe `Data` no mesmo pacote. É ela quem define os métodos que podem ser invocados e realiza todo o cálculo internamente através do uso da classe `Calendar`.

d) Para executar o programa, abra o arquivo `build.xml` e execute a tarefa `calculodata`.

5) a) Abra o pacote `br.com.caelum.fj91.criacaodata.dsl` e entre no arquivo `CriacaoData.scala`.

b) Observe dentro do método `main`, o código que é utilizado para criar novos objetos `Calendar`. Novamente, parece que é um simples texto e não um código que está sendo compilado.

c) Abra a classe `Data` no mesmo pacote. É ela quem define os métodos que podem ser invocados e realiza todo o cálculo internamente através do uso da classe `Calendar`.

d) Ao final do arquivo `Data.scala`, na linha 38, é definido o método **implícito** `intToDia`, que converte o objeto `Int` para algum outro tipo, no caso, um objeto do tipo `Dia`, que é retornado pelo método. Isso permite que o método `de` possa ser invocado em qualquer número.

e) A classe `Dia` tem o papel de *aumentar* o conjunto de métodos disponíveis no número. Chamamos isso de *adaptação*, e é a base do Design Pattern Adapter.

f) Para executar o programa, abra o arquivo `build.xml` e execute a tarefa `criacaodata`.

6) **Desafio**: Faça com que o seguinte código em Java compile e devolva um `Calendar`:

```
Calendar data = new Dia(11).de(Fevereiro).de(2012);
```

Dica: Para os meses, pode-se utilizar uma `enum` e se aproveitar do `import static` do Java.

Capítulo 5

# Separação de Responsabilidades

## 5.1 Leituras recomendadas

Para os tópicos deste capítulo, recomendamos a leitura dos seguintes capítulos do livro:

- 4.1 - Obtenha baixo acoplamento e alta coesão
- 4.2 - Gerencie suas dependências através de injeção
- 4.3 - Considere usar um framework de Injeção de Dependências
- 4.4 - Fábricas e o mito do baixo acoplamento
- 4.5 - Proxies dinâmicas e geração de bytecodes

## 5.2 Exercícios: Separação de responsabilidades, injeçao de dependências e inversão de controle

1) a) Importe em seu Eclipse, o projeto `spring`, disponível em `Caelum/91/fj-91-exercicios` e abra-o.

   b) No pacote `br.com.caelum.spring`, abra a classe `Main` e note que em seu método, ela utiliza o Spring, lendo o arquivo `beans.xml` para criar uma instância da classe `Menu`.

   c) Abra o arquivo `beans.xml`, e observe a declaração das dependências.

d) No arquivo `Menu.java`, repare os *setters* que permitem que as dependências sejam injetadas pelo Spring.

e) Execute a classe `Main` e manipule o programa no seu Console.

f) (opcional) Para visualizar o momento em que os *setters* são invocados pelo Spring, podemos adicionar alguns `System.out.println` neles. Na classe `Menu`, adicione aos *setters* o código para exibir algo no console e execute novamente a classe `Main`.

Atualmente, o Spring suporta as configurações via anotações e a injeção de dependências via construtor.

2) a) Importe em seu Eclipse, o projeto `guice`, disponível em `Caelum/91/fj-91-exercicios` e abra-o.

b) Observe a classe `Main` e note que não existe configuração em XML.

c) Abra a classe `Menu` e observe o construtor, onde há a anotação `@Inject` e recebe os dois parâmetros que devem ser injetados. O Guice invoca o construtor enviado os objetos criados como parâmetros.

d) Execute a classe `Main` e manipule o programa no seu Console.

3) a) Importe em seu Eclipse, o projeto `cdi`, disponível em `Caelum/91/fj-91-exercicios`, abra-o e associe ao seu Tomcat.

b) Acesse em seu navagador o endereço http://localhost:8080/cdi e manipule a aplicação.

c) Abra a classe `ListaContas` no pacote `br.com.caelum.fj91.banco.servlet` e observe a anotação `@Inject` no atributo `ContaDao`. Isso indica que uma instância de `ContaDao` será injetada quando uma instância de `ListaContas` for criada.

d) Vá até a classe `Sessao` que se encontra no pacote `br.com.caelum.fj91.banco.persistencia` e veja que é utilizado a anotação `@RequestScoped`, para identificar qual é o escopo do objeto do tipo `Sessao`. É possível definir outros escopos, como `SessionScoped` e `ApplicationScoped`, por exemplo.

e) Abra a Servlet `AplicaTributos` e repare que o atributo `tributos` está anotado com `@Any`, além dele ser um objeto do tipo `Instance<Tributo>`. A anotação `@Any` faz com que sejam injetados um objeto de cada implementação da interface `Tributo`, que está tipada em `Instance<Tributo>`. Com isso, é possível aplicar todos os tributos, simplesmente realizando um `for` por todos os elementos de `Instance`.

f) (opcional) Crie uma nova implementação da interface `Tributo`, re-inicie a aplicação e veja que sua nova implementação será injetada também. Não foi preciso realizar nenhuma configuração.

g) Abra a classe `DescontaJuros` que se está no pacote `br.com.caelum.fj91.banco.tributacao`. Repare que ele recebe injetado uma instância de `BigDecimal`, porém, qual o número que deve ser utilizado nesse `BigDecimal`?

h) Note a anotação `@TaxaJuros` no atributo `taxaDeJuros` junto da anotação `@Inject`. Isso significa que existe uma definição chamada `TaxaJuros` para a produção desse valor. Essa definição, está disponível na classe `Configuracoes`. Abra-a.

i) Repare no método `produzTaxaDeJuros` que ele está anotado com `@Produces` e `@TaxaJuros`, ou seja, aí está a definição de como gerar `TaxaJuros`. Chamamos esse recurso de `Qualifier`, que especifica (qualifica) a criação da instância.

j) Abra a Servlet `Sacar` e note o atributo `Event<Saque> eventoSaque`. No CDI, um evento, ao ser disparado, desencadeia ações em uma série de objetos, de forma similar ao pattern Observer. No caso, o evento é disparado na linha 34 da Servlet.

   No pacote `br.com.caelum.fj91.banco.eventos`, está disponível o evento `NotificaBancoCentral`. Note no método dessa classe, o parâmetro `Saque` anotado com `Observes`. Isso indica que quando o evento para o `Saque` for disparado, esse método vai ser invocado.

## 5.3 Exercício Opcional: Programação Orientada à Aspectos

1) * Importe em seu Eclipse, o projeto `aspectj`, disponível em `Caelum/91/fj-91-exercicios` e abra-o.

   * Nesse projetos, temos uma classe `Conta` no pacote `br.com.caelum.fj91.modelo` contendo alguns atributos e *getters* e métodos como *saca* e *deposita*.

   * Na classe `LogAspect` estão definidos os comportamentos que queremos que sejam executados e em para quais métodos, ou seja, temos as definições dos pointcuts utilizando a sintaxe do AspectJ.

   * Na classe `ExceptionHandler` está definido um método que é invocado quando houve uma *RuntimeException*.

   * A classe `Main` instancia uma nova `Conta`, executa um getter e saca um valor inválido para causar uma *RuntimeException*.

   * Para executarmos, clique com o botão direito na classe e em seguida vá até *Run as* e *Run Configurations...*. Na aba *Arguments*, adicione em *VM Arguments* o conteúdo:

   ```
   -javaagent:aspectjweaver.jar
   ```

* Mande executar a classe `Main` e veja a saída no console.

## 5.4 Exercício Opcional: Manipulação de Bytecode

1) * Importe em seu Eclipse, o projeto `javassist`, disponível em `Caelum/91/fj-91-exercicios` e abra-o.

   * No pacote `br.com.caelum.fj91.loader`, existe a classe `ObjetoTeste`, que é uma classe simples para manipularmos seu bytecode.

   * Temos também a classe `LoaderModificado`, que define um novo Classloader, filho de Bootstrap, que em seu método `findClass`, descobre quais são os métodos existentes na classe que está sendo carregada e adiciona um `System.out.println` em seu bytecode através da biblioteca Javassist.

   * Na classe `TesteJavassistViaClassLoader`, existe um método `main` que carrega a classe `ObjetoTeste` através do `LoaderModificado`, ou seja, nesse momento em que a classe é carregada, ela terá seu bytecode manipulado. Execute esse método e veja a saída no console.

Capítulo 6

# Decisões arquiteturais e trade-offs

## 6.1 Leituras recomendadas

Para os tópicos deste capítulo, recomendamos a leitura dos seguintes capítulos do livro:

- 6.1 - Dividindo em camadas: tiers e layers
- 6.2 - Desenvolvimento Web MVC: Actions ou Componentes?
- 6.3 - Domine sua ferramenta de mapeamento objeto relacional
- 6.4 - Distribuição de objetos
- 6.5 - Comunicação assíncrona
- 6.6 - Arquitetura contemporânea e o Cloud

Além disso, o livro *Patterns of Enterprise Application Architecture* de Martin Fowler é bastante recomendado.

## 6.2 Exercícios opcionais - Parte 1: Balanceamento de carga e o tradeoff entre escalabilidade, disponibilidade e confiabilidade

Nesse projeto vamos escalar uma aplicação web. Todas as configurações estão preparadas e serão executadas na linha de comando. Não utilizaremos Eclipse. Tarefa é escalar a aplicação sem mexer no código.

1) Copie o arquivo `Caelum/91/fj-91-exercicios/loadbalancer.tar` para sua pasta pessoal e extraia o arquivo nesta pasta. Nele se encontram todos os arquivos e configurações necessários.

2) Na linha de comando entre na pasta `loadbalancer/haproxy`. Vamos iniciar o load balancer no modo *sticky* usando *sticky session*. Para isso execute no terminal:

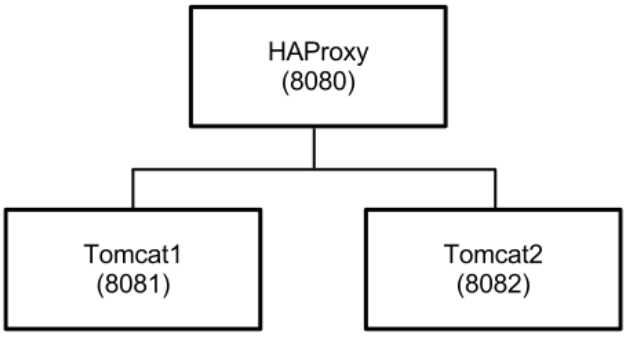
```
cd ~/loadbalancer/haproxy
sh run-haproxy-sticky.sh
```

Isso inicializa o haproxy usando o arquivo de configuração `haproxy-sticky.conf` que está na mesma pasta.

Com o proxy rodando podemos acessar pelo Firefox: http://localhost:8080 .

No navegador deve aparecer o código HTTP 503.

3) Vamos inicializar o *backend*, os servidores atrás do haproxy. Subiremos duas instâncias do Tomcat nas portas `8081` e `8082` respectivamente.




4) **Abra um novo terminal** para a primeira instância do Tomcat e entre na pasta `serverstate-session`:

```
cd ~/loadbalancer/configs/serverstate-session
sh run-tomcat1.sh
```

5) **Abra outro terminal**, entre na mesma pasta e suba o tomcat 2:

```
cd ~/loadbalancer/configs/serverstate-session
sh run-tomcat2.sh
```

6) Com o proxy e backend rodando podemos acessar a aplicação:

http://localhost:8080/livraria .

7) Adicione alguns livros no carrinho e verifique nos terminais qual Tomcat respondeu. Cada ação pelo navegador causa um *System.out.println* no lado do servidor.

8) No Firefox verifique o cookie `JSESSIONID`. Para tal, abra as propriedades da página (Botão direito -> Propriedades da página -> Aba Segurança -> Exibir cookies).

   Repare que aparece o nome do Tomcat no cookie, por exemplo:

   ```
   JSESSIONID=tomcat2~A7D83AEC756577D29B27BDA329E60C80
   ```

9) Vamos simular uma queda de um dos servidores no *backend*. Para isso vamos matar o processo do Tomcat. Escolha o terminal do Tomcat que respondeu ao seu cliente e execute:

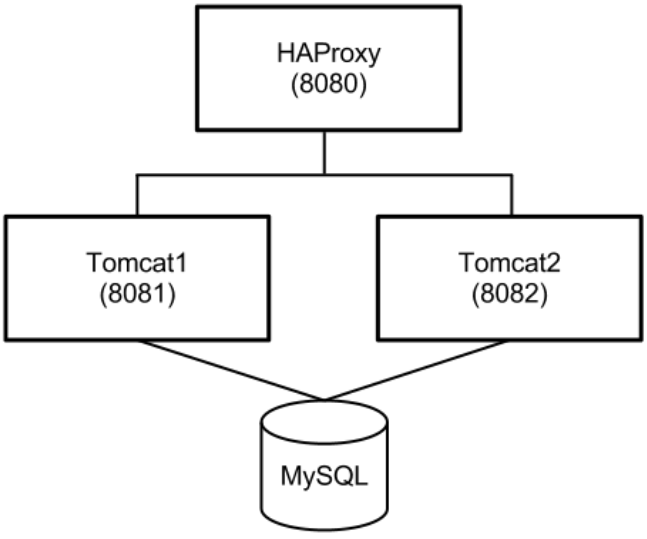
   ```
   sh kill-tomcat[aqui o numero do tomcat].sh
   ```

10) Tente usar a aplicação pelo navegador. Repare que a aplicação continua em pé. O Haproxy delega as requisições para o outro Tomcat, mas perdemos os livros no carrinho.

    (Opcional) Discuta com o instrutor o tradeoff entre escalabilidade, disponibilidade e confiabilidade.

## 6.3 Exercícios opcionais - Parte 2: Balanceamento de carga e o tradeoff entre escalabilidade, disponibilidade e confiabilidade

Conseguimos escalar a aplicação e deixá-la disponível. Também queremos aumentar a confiabilidade - ou seja, não queremos perder os dados do carrinho de compras. Para isso usaremos o banco de dados como backup dos dados de sessão.



1) Antes de testar outras configurações verifique se todas as instâncias do Tomcat foram desligadas.

   Para tal execute no terminal do Tomcat:

   ```
   sh kill-tomcat1.sh
   ```

   e

   ```
   sh kill-tomcat2.sh
   ```

   Para verificar, acesse pelo navegador a aplicação: http://localhost:8080/livraria .

   No navegador deve aparecer o código HTTP 503.

2) Novamente subiremos duas instâncias do Tomcat, mas dessa vez os Tomcats foram configurados para gravar os dados da sessão periodicamente no banco mysql. Por isso temos que preparar o MySQL antes.

   **Abra um novo terminal** e execute:

   ```
   cd ~/loadbalancer/config/serverstate-database
   sh create-db.sh
   ```

   Verifique o banco no MySQL:

   ```
   mysql -u root

   use tomcat;

   select session_id from sessions;
   ```

   Deixe o terminal aberto.

3) **Abra um novo terminal** para rodar a primeira instância do Tomcat. Entre na pasta `serverstate-database`:

   ```
   cd ~/loadbalancer/config/serverstate-database
   sh run-tomcat1.sh
   ```

4) **Abra outro terminal**, entre na mesma pasta e suba o Tomcat 2:

   ```
   cd ~/loadbalancer/config/serverstate-database
   sh run-tomcat2.sh
   ```

5) Com o proxy e backend rodando, podemos acessar a aplicação novamente:

   http://localhost:8080/livraria .

6) Adicione alguns livros no carrinho e verifique nos terminais qual Tomcat respondeu.

7) Verifique se a sessão foi gravada no MySQL. No terminal do MySQL execute novamente o select:

```
select session_id from sessions;
```

Como o backup é feito assincronamente, pode demorar um pouco até aparecerem os dados de sessão no banco.

8) Novamente simularemos a queda do servidor matando o processo do Tomcat. Escolha o terminal do Tomcat que respondeu ao seu cliente e execute:

```
sh kill-tomcat[aqui o numero do tomcat].sh
```

9) Tente usar a aplicação pelo navegador. O outro Tomcat deve assumir o trabalho e recuperar o estado da sessão do banco de dados.

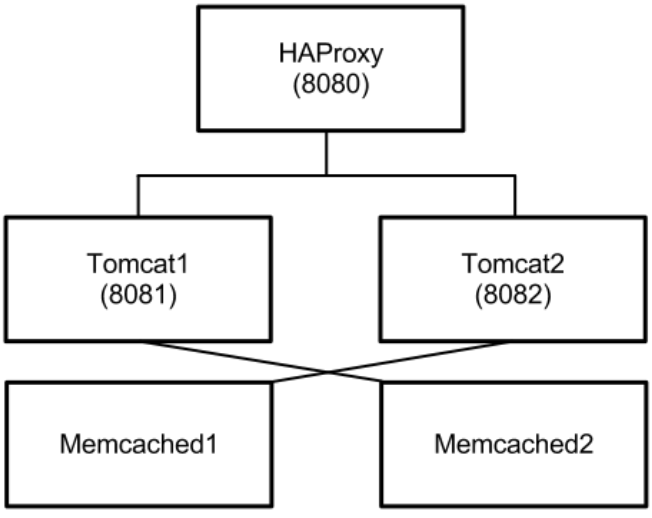
10) (Opcional) Verifique também a configuração do Tomcat para gravar os dados da sessão como backup.

Abra o arquivo `~/loadbalancer/configs/serverstate-database/tomcat1/conf/context.xml`.

Repare o `PersistentManager` e o `JDBCStore`.

## 6.4 Exercícios opcionais - Parte 3: Balanceamento de carga e o tradeoff entre escalabilidade, disponibilidade e confiabilidade

Conseguimos escalar a aplicação, deixá-la disponível e aumentamos a confiabilidade. Porém, o banco de dados *relacional* não parece ser o lugar ideal para gravar os dados do tipo chave-valor. Também não queremos torná-lo responsável por mais uma tarefa já que ele administra todos os dados da aplicação. Por isso usaremos um *Key-Value Store* para backup da sessão, no nosso caso **Memcached**.



1) Novamente, antes de testar outras configurações, verifique se todas as instâncias do Tomcat foram desligadas.

   Para tal execute no terminal do Tomcat:

   ```
   sh kill-tomcat1.sh
   ```

   e

   ```
   sh kill-tomcat2.sh
   ```

   Para verificar, acesse pelo navegador a aplicação: http://localhost:8080/livraria .

   No navegador deve aparecer o código HTTP 503.

2) **Abra dois novos terminais** (um para cada nó do Memcached) e execute:

   No terminal 1:

   ```
   cd ~/loadbalancer/config/serverstate-Memcached
   sh run-memcached1.sh
   ```

   No terminal 2:

   ```
   cd ~/loadbalancer/config/serverstate-Memcached
   sh run-memcached2.sh
   ```

3) Novamente subiremos duas instâncias do Tomcat. Elas foram configuradas para gravar os dados da sessão periodicamente no Memcached. Cada Tomcat usará um nó como backup, aumentando ainda mais a disponibilidade e confiabilidade.

   **Abra um novo terminal** para rodar a primeira instância do Tomcat. Entre na pasta `~/loadbalancer/config/serverstate-database`:

   ```
   cd ~/loadbalancer/config/serverstate-Memcached
   sh run-tomcat1.sh
   ```

4) **Abra outro terminal**, entre na mesma pasta e suba o Tomcat 2:

   ```
   cd ~/loadbalancer/config/serverstate-Memcached
   sh run-tomcat2.sh
   ```

5) Com o proxy, backend e nós do memchaced rodando podemos acessar a aplicação novamente:

   http://localhost:8080/livraria .

6) Adicione alguns livros no carrinho e verifique nos terminais qual Tomcat respondeu.

7) Novamente simularemos a queda do servidor matando o processo do Tomcat. Escolha o terminal do Tomcat que respondeu ao seu cliente e execute:

```
sh kill-tomcat[aqui o numero do tomcat].sh
```

8) Tente usar a aplicação pelo navegador. O outro Tomcat deve assumir o trabalho e recuperar o estado da sessão do nó do Memcached.

9) (Opcional) Verifique também a configuração do Tomcat para gravar os dados da sessão como backup no Memcached.

   Abra o arquivo `~/loadbalancer/configs/serverstate-Memcached/tomcat1/conf/context.xml`.

   Repare o `MemcachedBackupSessionManager` e a configuração dos nós do Memcached.

## 6.5 Exercícios: Gerenciabilidade

1) Descompacte em seu `Desktop` o arquivo `Caelum/91/probe.zip`, e copie o arquivo `probe.war` para o diretório `webapps` do seu Tomcat.

2) Abra o arquivo `tomcat-users.xml` existente no diretório `conf` que está na raiz do seu Tomcat e adicione as seguintes linhas dentro da tag `tomcat-users`:

```
<role rolename="probeuser" />
<role rolename="poweruser" />
<role rolename="poweruserplus" />
<role rolename="manager" />

<user username="admin" password="senha" roles="manager" />
```

3) Inicialize o Tomcat na linha de comando. Entre na pasta `bin` da instalação do seu Tomcat e execute:

```
sh startup.sh
```

4) Acesse o endereço http://localhost:8080/probe e visualize os resultados exibidos pelo probe em sua página.

## 6.6 Exercícios: Evite injeção de Scripts em suas páginas

1) Inicie o Tomcat com o projeto `banco2` associado à ele e acesse http://localhost:8080/banco2 .

2) Cadastre uma nova conta cujo conteúdo do nome do cliente seja `<script>alert('uma mensagem');</script>`.

3) Abra a tela para visualizar as informações da nova conta e observe que a mensagem de alerta sempre abre. Poderíamos ter colocado um Javascript mais ofensivo, como por exemplo, enviar requisições AJAX em um looping infinito para a aplicação.

4) Corrija a vulnerabilidade alterando a página `visualiza.jsp` no diretório `WEB-INF/jsp/conta` para realizar o escape das tags XML, dessa forma, as tags não serão interpretadas pelo navegador e sim, simplesmente exibidas.

   a) Adicione o import da Taglib core ao começo do arquivo:

   ```
   <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
   ```

   b) Utilize a tag `c:out` com o atributo `escapeXml` para exibir as informações da conta:

   ```
   <h1>Conta <c:out value="${conta.numero}" escapeXml="true" /> do
       <c:out value="${conta.titular.nome}" escapeXml="true"/></h1>
   ```

5) Acesse novamente a página e veja o resultado.

6) Veja também os artigos no blog da caelum sobre script e parameter injection:

   - http://blog.caelum.com.br/seguranca-em-aplicacoes-web-xss/
   - http://blog.caelum.com.br/seguranca-em-aplicacoes-web-injecao-de-novos-parametros/

## 6.7 Exercícios: Boas e más práticas com o Hibernate

1) No seu Terminal, acesse o `mysql` e crie o banco de dados `testehibernate`:

   ```
   mysql -u root

   create database testehibernate;
   ```

2) Importe em seu Eclipse o projeto `hibernate`, associe-o com o Tomcat. Configure o servidor para iniciar com 1GB de memória e inicialize-o(Run Configurations -> Aba Arguments -> VM arguments: -Xms1G -Xmx1G).

3) Acesse http://localhost:8080/hibernate e clique no link "Gerar 1 milhao de registros", que inserirá uma grande quantidade de informações no banco de dados. Essa operação pode demorar alguns minutos.

4) Novamente na página inicial do projeto, acesse o link "200.000 registros sem Cursor (Hibernate)" e anote o tempo de demora que é exibido no console.

   Compare esse tempo com a execução dos outros dois links: "200.000 registros com Cursor (Hibernate)" e "200.000 registros com Cursor (JDBC)".

5) Clique no link para aumentar os valores em 10% através de um update em batch e veja o tempo levado na execução. Tente realizar a mesma tarefa com o Hibernate objeto por objeto e note que em um determinado momento, o servidor para de responder, pois vai acontecer um `OutOfMemoryError`.

6) Reinicie o servidor e acesse a tela inicial do projeto `hibernate`.

7) Nesse momento, o projeto não utiliza cache de segundo nível. Execute a consulta para mostrar os registros de 2011 e anote o tempo de execução. Execute aproximadamente 3 vezes, para ter um tempo médio das execuções.

8) Habilite o cache de segundo nível, através das instruções dadas na página inicial do projeto, reinicie o servidor e acesse novamente o relatório com os registros de 2011. Note que o tempo de execução agora é menor e a consulta só é feita no banco de dados na primeira vez.

## 6.8 Exercícios: OpenSessionInView ou Queries planejadas

1) Acesse http://localhost:8080/hibernate/contas/index . Inicie Tomcat, caso for necessário.

   Na página inicial clique no link: "Gerar contas e algumas transacoes". Isso gera 25 contas.

   Após ter gerado as contas clique no link: "Listar todas as contas". Verifique a quantidade de selects no console do Eclipse (queries n+1).

2) Abra a classe `Conta` e verifique o relacionamento com a classe `Transcacao`, trata se de um relacionamento @OneToMany.

3) Vamos planejar a query para não executar n+1 queries. Na classe `ContasController` procure o método `listar()`. Nele abriremos a sessão do Hibernate manualmente para depois executar uma query planejada.

   Coloque o código seguinte na método `listar()`. Repare que estamos carregando as contas junto com as transações:

```
Session sessao = factory.openSession();
List<Conta> contas = sessao.createQuery("from Conta c
    left join fetch c.transacoes").list();
result.include("contas", contas);
sessao.close();
```

   Reinicie o servidor.

4) Após execução verifique novamente a console e a quantidade de selects executadas. Quais são as vantagens e desvantagens do OpenSessionInView ou Queries planejadas?

## 6.9 Exercícios: Action ou Component-based

Temos dois projetos preparados: um component-based e outro action-based. Utilizaremos JSF e Spring MVC respectivamente.

Em cada projeto temos uma página na qual o usuário poderá digitar seu nome e idade e uma lista com os dados cadastrados.

Faremos alguns exercícios que envolvem requisições ajax e suporte a HTML 5 em cada um deles.

## 6.10 Exercício - Ajax

Vamos implementar requisição Ajax para cada um dos projetos, mas primeiro vamos importar cada um deles.

1) Importe no eclipse os projetos **mvc-component** e **mvc-action** respectivamente.

2) Agora com os projetos importados, vamos implementar o recurso ajax primeiro no projeto `mvc-action`. Abra o arquivo `WebContent/WEB-NF/view/cadastro.jsp`.

3) Faremos nossas requisições ajax com auxílio do jQuery. jQuery já está importado no fim da página HTML, **antes** do fechamento da tag do `body`. Verifique:

```
<script src="resources/js/jquery.js"></script>
```

4) Logo abaixo da importação do jQuery, temos o código que fará a requisição Ajax e que também atualizará dinamicamente nossa tabela:

```
<script>
    $("input[type=submit]").on("click", function(event) {
        event.preventDefault();
        $.ajax({
            url : "adiciona",
            data : $("form").serialize(),
            success : function(retorno, codigo, xhr) {
                $(retorno).appendTo("table");
            }
        });
    });
</script>
```

5) Precisamos implementar ainda a resposta à requisição ajax em nosso controlador. Para isso, abra o arquivo `PessoaController.java` e adicione o seguinte método:

```
@RequestMapping("/adiciona")
public String adiciona(Pessoa pessoa, Model model) {
    // aplicação real gravaria no banco de dados
    model.addAttribute("pessoa", pessoa);
    return "tr";
}
```

6) Crie o arquivo WebContent/WEB-INF/views/tr.jsp. O arquivo deve ter apenas o código seguinte:

```
<tr>
        <td>${pessoa.nome}</td>
        <td>${pessoa.idade}</td>
</tr>
```

7) Agora que já temos tudo preparado, rode seu projeto e acesse: http://localhost:8080/mvc-action/

   Cadastre algumas informações e verifique se tudo esta funcionando.

8) Agora que já adicionamos suporte Ajax em nossa aplicação action-based, faremos a mesma coisa com nossa aplicação component-based.

9) No projeto `mvc-component`, abra a página `WebContent/index.xhtml`. Para o componente `h:commandButton`, adicione a tag **f:ajax** conforme o exemplo abaixo:

```
<h:commandButton action="#{pessoasController.incluir}" value="gravar">
    <f:ajax render="@form" execute="@form"/>
</h:commandButton>
```

10) Não há necessidade de adicionarmos código no lado do servidor.

11) Rode a aplicação e acesse a página: http://localhost:8080/mvc-component/

    Cadastre algumas informações e verifique se tudo esta funcionando.

## 6.11 Exercício - Utilizando HTML5

Nos últimos anos muito tem se falado sobre a próxima versão do HTML, o HTML5. Esse projeto é um grande esforço do W3C para atender a uma série de necessidades do desenvolvimento da Web.

Apesar da especificação ainda não estar completa e existirem diferenças entre as implementações adotadas pelos diferentes navegadores ainda hoje, o mercado está tomando uma posição bem agressiva quanto à adoção dos novos padrões e hoje muitos projetos já são iniciados com eles.

Neste exercício focaremos apenas na utilização dos novos `types` do elemento `input` no HTML5.

1) Vamos utilizar o tipo "number" para o `input` que guarda o valor para o campo idade. No HTML5, ele é representado por um "spinner".

   Não esqueça de verificar primeiro se o navegador escolhido suporta esta tipo de entrada. Você pode verificar facilmente acessando o site **http://html5test.com** que apresenta uma lista todos os recursos suportados e também os não suportados do HTML5 de seu navegador.

2) No projeto `mvc-action`, na página `cadastro.jsp`, vamos alterar o input da idade:

```
<input type="number" name="idade"/>
```

3) Teste o resultado. O campo que recebe a idade deve ser apresentado como um `spinner`.

4) Faça agora a mesma coisa para o projeto `mvc-component`, abrindo a página `index.xhtml` e incluindo o `type number` para o componente `h:inputText`:

```
<h:inputText type="number" id="idade" value="#{pessoasController.pessoa.idade}" />
```

5) Verifique o resultado. O campo foi apresentado como um `spinner`?

## 6.12 Exercícios opcionais - MVC client-side

Neste exercício, teremos uma lista e um campo no qual o usuário poderá digitar o nome de uma linguagem de programação que será adicionado na lista quando o usuário clicar no botão 'Adiciona'.

Existem duas implementações, uma usando jQuery e outra com o modelo MVC utilizando **AngularJS**.

1) Copie a pasta `mvc-client` que está disponível em `Caelum/91` para sua pasta pessoal. Visualize a página **angular.html**. Visualize a página em seu navegador e experimente cadastrar algumas linguagens.

2) Abra **angular.html** e perceba que dois scripts do AngularJS foram importados antes do fechamento da tag `body`. O primeiro representa a biblioteca AngularJS, o segundo contém os códigos do controlador e modelo do MVC.

3) Abra o arquivo **js/com-angular.js**. O AngularJS trabalha com um sistema de módulos e para a criação foi disponibilizado um objeto globalmente chamado **angular**.

   É através dele que executamos uma série de tarefas, inclusive a criação de nosso módulo que se chama **linguagemModule**:

```
var app = angular.module('linguagemModule', []);
```

   O segundo parâmetro é um array vazio. Ele serve para indicar todos os módulos que o nosso módulo depende. Nesse exemplo, não temos dependência alguma.

4) No arquivo **angular.html** já foi adicionado o atributo **ng-app** na tag `<html>` apontando para o nome do nosso módulo.

   A tag HTML já está assim:

```
<html ng-app="linguagemModule">
....
```

Repare que apesar de termos chamado `ng-app` de atributo, ele não existe no mundo HTML. Na verdade, `ng-app` é uma **diretiva do Angular**. Diretivas servem para ensinar novos truques para o navegador. Utilizaremos outras ao longo do exercício.

5) Volte para **com-angular.js**. Depois da definição do módulo se encontra a função **controller** para a criação do controlador MVC.

   Ela recebe como primeiro parâmetro o nome do controlador e como segundo uma função que nos dá acesso ao objeto **$scope** que representa o escopo do controlador:

```
app.controller('LinguagensController', function($scope) {

});
```

6) Em AngularJS, qualquer tipo primitivo ou objetos podem ser `model` para view.

   O objeto `$scope` é a ponte de *ligação entre o controlador e a view*. Sendo um objeto JavaScript ele permite que sejam adicionadas quantas propriedades quisermos. Estas propriedades estarão disponíveis para a view através de **angular expression** {{}}:

```
$scope.linguagens = [new Linguagem("Java"), new Linguagem("JavaScript")];
```

   No exemplo anterior, definimos dinamicamente o atributo **linguagens** que possui uma lista com dois objetos que possuem a propriedade **nome**.

7) Volte para **angular.html**. Procure o atributo `ng-controller` que faz parte da tag `<body>` para associar nosso controller ao elemento do DOM:

```
<body class="container text-center" ng-controller="LinguagensController">
```

   Quando a página for carregada, o AngularJS se encarregará de instanciar nosso controller para nós.

8) A diretiva **ng-repeat** foi utilizada para montar elementos `<li>` dinamicamente:

```
<li class="lista" ng-repeat="linguagem in linguagens">
    {{linguagem.getNome()}}
</li>
```

   Repare que para cada item da lista 'linguagens', disponibilizada no `$scope` do controller, será criado uma `<li>`, inclusive usamos **angular expression** para chamar a função que devolve o valor da propriedade **nome** da linguagem.

9) No **js/com-angular.js** existe mais uma propriedade em **$scope** para guardar uma linguagem que associaremos ao input do HTML:

```
$scope.nome = "";
```

   No `angular.html` já fizemos a associação da view com o model:

```
<input type="text" ng-model="nome"/>
```

Repare que agora a ligação entre view e model é bidirecional, isto é, quando digitarmos no `input` queremos que nosso model seja atualizado e vice-versa. Isso se chama **two-way databinding**. Isso é feito através da diretiva **ng-model**.

Desta vez não usamos `{{}}`, porque queremos uma ligação bidirecional e `{{}}` realiza apenas ligações unidirecionais.

10) Precisamos incluir na lista a liguagem digitada quando clicarmos no botão. Para isso, existe no escopo do controller uma propriedade que guarda a função que será chamada quando o botão for clicado:

```
$scope.adiciona = function() {
    var novaLinguagem = new Linguagem($scope.nome);
    $scope.linguagens.push(novaLinguagem);
    $scope.nome = "";
};
```

Veja que a função recupera o nome da nova linguagem digitada adicionando-a logo em seguida na lista.

No HTML existe a diretiva **ng-click** no botão da página. Esta diretiva chamará a função **adiciona()** no escopo de nosso controlador:

```
<button class="btn btn-primary" ng-click="adiciona()">
    Adiciona
</button>
```

11) Por fim, verifique a implementação dessa funcionalidade através do jQuery. Abra o arquivo `js/com-jquery.js`. Repare as manipulações dos elementos no DOM pelo código JavaScript.

## 6.13 Exercícios: Java EE vs Spring Framework

1) No Eclipse, se não tiver disponível ainda, baixe o server adapter JBoss AS Tools.

2) No seu Terminal, acesse o `mysql` e crie o banco de dados `fj91`:

```
mysql -u root

drop database fj91;

create database fj91;
```

3) Entre na pasta `caelum/91` e descompacte o arquivo `jbosss-as-7.x.x-FINAL.zip` para sua pasta pessoal.

4) No Eclipse, com o server adapter instalado, configure JBoss AS como servidor.

   Obs: O JBoss AS 7 não tem suporte para o Java 8, portanto verifique se no Eclipse está instalado o Java 7, e caso não esteja adicione-o: Window -> Preferences -> Java -> Installed JREs -> Botão Add -> Standard VM -> Botão Directory -> escolha o diretório do Java 7 em: File System/usr/lib/jvm

5) Importe os projetos `spring-framework` e `javaee` em seu Eclipse abrindo-os em seguida.

6) O projeto `javaee` depende de uma configuração externa do servidor JBoss AS, nesse caso foi preciso configurar um *datasource* e um *security-domain*.

   Um *datasource* encapsula os detalhes da conexão (credencias, pool etc) e um *security-domain* autentica e autoriza um usuário (ou *principal*).

   As configurações são tipicamente feitas pelo administrador do servidor, por isso já foram implementadas, no entanto vamos conferi-las.

   Entre na pasta do *JBossAS*/standalone/configuration e abra o arquivo *standalone.xml*. Com o XML aberto procure por **mysqlDS** e verifique a configuração do pool de conexão.

   Depois procure por **fj91sec** para ver a configuração do *security-domain*. Repare que usamos o banco de dados (*datasource*) para recuperar o usuário e e os papeis dele.

7) Os dois são projetos web com IoC/DI que acessam o banco de dados MySQL, publicam um web-service SOAP e Rest e utilizam um mecanismo de segurança.

   Suba o JBoss e use http://localhost:8080/javaee .

   Após ter testado a aplicação desligue o JBoss AS.

8) Associe a aplicação `spring-framework` como o Tomcat.

   Inicie o Tomcat e acesse uma vez http://localhost:8080/spring-framework .

9) Analise os projetos, observe como eles foram configurados e quais são os arquivos de configuração. Anote os frameworks, componentes e camadas. Faça desenhos se for preciso.

   Quais são os componentes Java EE que o próprio projeto `spring-framework` usa?

10) Repare a quantidade de JARs na pasta `WEB-INF/lib` do projeto `spring-framework`. Quem é responsável por carregar os JARs? Seria fácil rodar o projeto em outro servlet container como Jetty? Como atualizar, por exemplo, a versão do Hibernate no projeto? E no projeto `javaee`?

11) Quais são os containers usados no projeto `javaee`? Seria possível não usar o EJB Container? Seria facil testar a camada de persistência do projeto `javaee`? E do `spring-framework`?

12) O projeto `spring-framework` usa Spring MVC (action-based), o `javaee` usa JSF (component-based). Seria possível usar JSF com Spring? E Spring MVC com Java EE?

13) Compare a segurança nos projetos. Ambos seguem da forma declarativa - o Spring com Spring Security e Java EE com JAAS. Verifique os arquivos `application-context.xml` do `spring-framework` e `web.xml` do `javaee`.

    Como saber qual provider de segurança foi utilizado no projeto `javaee`? O que é mais transparente, o `spring-framework` ou `javaee`?

14) Onde se encontra a documentação do Spring? Quem dá continuidade e suporte ao Spring? E no Java EE?

Capítulo 7

# Integração na Web e REST

## 7.1 Leituras recomendadas

Para os tópicos deste capítulo, recomendamos a leitura dos seguintes capítulos do livro:

- 7.1 - Princípios de integração de sistemas na Web
- 7.2 - Padronizações, contratos rígidos e SOAP
- 7.3 - Evite quebrar compatibilidade em seus serviços
- 7.4 - Princípios do SOA
- 7.5 - REST: arquitetura distribuída baseada em hipermídia

## 7.2 Exercícios: Consumo de serviços - POX (Plain Old XML)

1) Importe no seu Eclipse o projeto `webservices-flickr` e abra-o.
2) Execute a classe `PegaXmlDoFlickr` e veja o XML resultante do consumo do serviço das fotos do flickr.
3) Para trabalhar com o XML de maneira mais fácil, existem as classes `Photo` e `Result`, que estão mapeadas com o XStream para fazer o parse do XML para objetos. Visualize-as.
4) Execute a classe `FotosInteressantesDoFlickr` que consome uma URL, que devolve um XML como resposta. No XML, existe a URL da foto, que é utilizada para exibi-la em um JFrame.

## 7.3 Exercícios: Consumo de serviços - Web Services SOAP

1) Importe o projeto `webservices-soap` em seu Eclipse e abra-o.

2) Abra a classe `Dicionario` e repare na anotação `@WebService`, indicando que os métodos dentro dela devem ser expostos como um Web Service, para serem consumidos por outros desenvolvedores.

   Esse serviço simula um dicionário, onde dada uma palavra, a sua tradução é carregada de um banco de dados, inclusive novas traduções podem ser adicionadas.

3) Podemos colocar o serviço no ar, executando a classe `PublicaEndpoint`. Caso estivéssemos em um ambiente JavaEE, o próprio servidor se encarregaria de colocar o serviço no ar.

4) Com o serviço executando, acesse http://localhost:8080/Dicionario?wsdl para visualizar seu WSDL.

5) Para consumí-lo, execute no terminal, dentro do diretório do projeto, o comando:

```
wsimport -d src/ -s src/ -p
br.com.caelum.fj91.cliente http://localhost:8080/Dicionario?wsdl
```

   Esse comando, lerá o WSDL do serviço e vai gerar as classes necessárias (Stubs) para consumir o serviço.

6) No Eclipse, dê um F5 no projeto, e veja o novo pacote `cliente` que foi criado.

7) Nesse novo pacote, crie a classe chamada `ConsumidorServico` com o seguinte conteúdo:

```
package br.com.caelum.fj91.cliente;

public class ConsumidorServico {
    public static void main(String[] args) {

        Dicionario dicionario = new DicionarioService().getDicionarioPort();
        String palavra = "bola";
        System.out.println("Traduzindo a palavra '" + palavra + "'");
        String traducao = dicionario.traduz(palavra);
        System.out.println("Tradução:" + traducao);

        System.out.println("###############");
        String novaPalavra = "ambiente";
        String novaTraducao = "environment";
        System.out.println("Adicionando nova tradução:"
            + novaPalavra + "/" + novaTraducao);

        dicionario.adicionaTraducao(novaPalavra, novaTraducao);
    }
}
```

8) Ao executar a classe `ConsumidorServico`, uma requisição é enviada para o serviço que retorna um XML que é parseado pelo próprio JAX-WS. Como resultado final, temos a tradução da palavra que pedimos. Observe que toda a infra-estrutura para se trabalhar com WebServices, fica abstraída do desenvolvedor.

## 7.4 Exercício Opcional: Consumo de serviços - Web Services REST

1) Importe o projeto `webservices-rest` em seu Eclipse e abra-o.

2) Abra o pacote `br.com.caelum.fj91.rest.model` e veja o modelo existente para aplicação, onde existem os pedidos, clientes, produtos e endereço dos clientes.

3) No pacote `br.com.caelum.fj91.rest.db` existe uma classe chamada `Repositorio`, fazendo o papel de um banco de dados em memória, contendo pedidos já cadastrados.

4) No pacote `br.com.caelum.fj91.rest.resources`, existe uma classes que possuem definições de recursos para serem trabalhados com o Jersey.

5) Abra a classe `PedidoResource` onde encontra-se todas as operações que podem ser realizadas com os pedidos, como criar um novo, buscar, atualizar, comentar, cancelar e assim por diante.

6) Associe o projeto com o Apache Tomcat e inicie o servidor. No Firefox acesse:

   http://localhost:8080/webservices-rest/pedido/1.

   Veja o conteúdo exibido no navegador.

7) Na raíz do projeto *webservices-rest* tem um arquivo README que possui comandos `CURL` para interagir com o recurso pela linha de comando.

8) (opcional) É possível utilizar bibliotecas que enviam requisição HTTP, para automatizar o processo de mudança de estado do recurso, como por exemplo, fazer com que pedidos RECEBIDOS cuja quantidade seja maior que 20 sejam automaticamente cancelados. Tudo isso, através de um programa, sem nenhuma interação humana.

## 7.5 Exercícios opcionais: Mensageria assíncrona e o JMS

1) Copie a pasta `apache-activemq-5.9.0` que está disponível em `Caelum/91` para sua pasta pessoal.

2) Pelo terminal, em uma aba, acesse o diretório `apache-activemq-5.9.0/bin` da pasta onde o JMS foi descompactado, em seguida, dentro dela, execute o comando:

```
sh activemq console
```

3) Importe o projeto `Caelum/91/jms.zip` em seu Eclipse abrindo-o logo em seguida.

4) Execute a classe `RegistraMessageListener`. Ela aguardará a chegada de novas mensagens enviadas para a fila `fj91`. Quando uma mensagem for enviada, ela será pega e exibida no console pelo `JmsMessageListener`.

5) Rode a classe `MandaMensagem` e veja no console do `RegistraMessageListener` a mensagem que foi recebida pelo outro processo, através do ActiveMQ.

6) (opcional) Altere o arquivo jndi.properties para apontar para o IP de algum colega de sala. Certifique-se que o ActiveMQ está executando no computador dele, e envie mensagens para a fila dele.

## 7.6 Exercícios opcionais: Padrões de integração com Apache Camel

1) Importe o projeto `Caelum/91/camel.zip` em seu Eclipse abrindo-o logo em seguida.

   O Apache Camel é nada mais do que um **roteador** (*routing engine*) e a tarefa do desenvolvedor é configurar, através de um *Builder*, as regras de roteamento. O desenvolvedor decide de onde vem as mensagens (`from()`), para onde enviar (`to()`) e o que fazer com a mensagem no meio desse processo (*mediation engine*).

2) Abra a classe `RotaFileParaSoap` que define as rotas do Camel.

   Lá podemos ver o tratamento de exceções e mais 3 rotas (*from()*) configuradas. As rotas funcionam dentro de uma cadeia, uma chamando a outra.

   Veja a primeira rota que lê arquivos XML de uma pasta:

```
from("file:itens)
    ....
```

   A segunda rota gera uma mensagem SOAP usando a template engine Velocity.

```
...
to("velocity:soap_request.vm")
...
```

   A última rota envia o XML SOAP para o Web Service:

```
...
to("http4://localhost:8080/EstoqueWS");
```

3) Rode a classe `RodaCamel` no Eclipse e veja no console as mensagens de saída.

4) Acesse a página http://camel.apache.org/components.html e verifique a grande variedade des componentes disponíveis.

Capítulo 8

# Apêndice - Design Patterns

*"Que se cale aquele que fez um benefício. Que o divulgue aquele que o recebeu."*
– Sêneca

Vamos ver e discutir alguns dos principais design patterns encontrados e utilizados. Selecionamos alguns bem comuns e outros nem tantos, mas todos com exemplos práticos extraídos da própria Java SE!

## 8.1 Para estudar Design Patterns

Para um estudo prático de patterns e vê-los aplicados de acordo com sua necessidade, a Caelum possui um curso online:

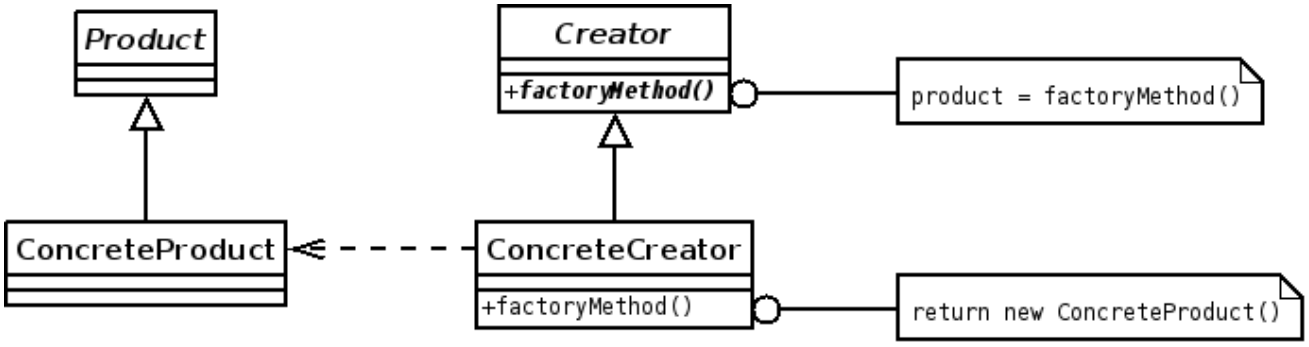
**Design Patterns para Bons Programadores**

http://www.caelum.com.br/curso/online/design-patterns/

## 8.2 Factory Method

As vezes gostaríamos de abstrair o processo de instanciação de uma classe, por diversos motivos: um seria para fazer cache de objetos, outro seria para devolver uma instância de uma classe filha.

Além de todas essas vantagens, usar um factory method possibilita nomes mais descritivos do que os construtores, que não possuem nomes.




## 8.3 Exercícios

1) Um factory method não precisa devolver necessariamente uma instância daquela própria classe. Ela pode devolver uma instância de alguma subclasse compatível com aquele retorno. Como fazer isso? Decidir em base de quê?

2) Verifique a classe `DriverManager` do `java.sql`, e o método `getConnection`. Um design pattern não precisa seguir a mesma fórmula de sempre.

3) As classes wrappers, como `Boolean` e `Integer`, possuem factory methods, porém seus construtores não são privados. Há algum problema nisso? Vantagens?

## 8.4 Singleton

Sem dúvida um dos mais amados e odiados patterns.

O objetivo é querer ter apenas um objeto de determinada classe, por uma série de razões. Para isso impossibilitamos a criação de novas instâncias, assim como possibilitar o acesso a uma única instância. Fazemos isso tornando seus construtores privados, e a própria classe tem um atributo estático para essa única instância.

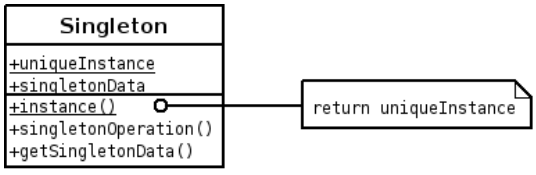
Imagine que queremos ter um `ConnectionPool`, como todo mundo vai usa-lo, será melhor garantir que só existe um dele:

```
public class ConnectionPool {

    private List<Connection> conexoes = new ArrayList<Connection>();

    private static ConnectionPool pool = new ConnectionPool();
```

```
    private ConnectionPool() {
    }

    public static ConnectionPool getPool() {
        return pool;
    }

    // outros métodos
}
```

E sempre que invocarmos o `getPool`, a mesma instância será devolvida:

```
public static void main(String[] args) {
    ConnectionPool pool1 = ConnectionPool.getPool();
    ConnectionPool pool2 = ConnectionPool.getPool();

    System.out.println(pool1 == pool2);
}
```




## 8.5 Exercícios

1) Como o atributo de instância pra própria classe de um singleton é estático, ele é lido em tempo de classloading. Como fazer isto de maneira lazy? Qual é a desvantagem?

2) Qual parece ser a principal desvantagem de um Singleton?

3) (opcional) Só existe um objeto singleton em cada JVM? Lembre-se do classloading.

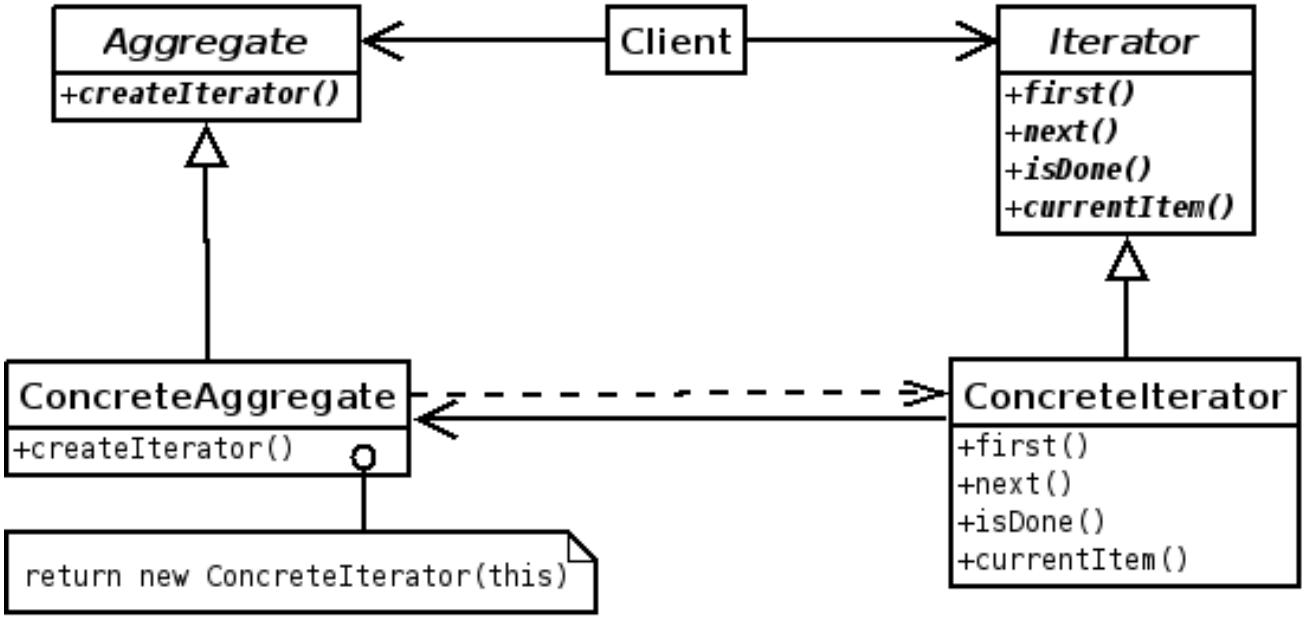
   Force com que existam duas instâncias de um singleton, comparando com == e verificando que dá false.

4) Podemos enxergar um Singleton como um Factory Method?

## 8.6 Iterator

Um iterador é um objeto que sabe percorrer os elementos a quem pertece. É uma maneira orientada a objetos de não atrelar estado de posição ao objeto o qual estamos percorrendo, possibilitando que diver-

sas pessoas diferentes percorram o mesmo objeto simultaneamente, e acessem posições bem distintas.




## 8.7 Exercícios

1) A interface do java.util, a `Iterator`, é um exemplo mais que óbvio. Consute também a interface `Iterable`. Como adaptar nossa tabela para utilizá-la em vez da `Collection`?

2) Um Iterator pode ser mais rebuscado. Verifique os métodos definidos na interface `ListIterator`.

## 8.8 Observer

É um pattern usado quando precisamos ser notificados de um determinado evento. Imagine que precisamos ser notificados toda vez que alguém clicar em um objeto visual. Como saberemos que isto aconteceu?

Para isso, avisamos ao objeto observável que ele possui um novo observador, adicionando-o em sua lista de observadores.

Quando o determinado evento ocorrer, o observável percorre todos os seus observadores notificando-os sobre o evento ocorrido.

O exemplo a seguir mostra um observador comum:

```
public interface Observador {
```

```
    void notifica(Observavel observavel);
}
```

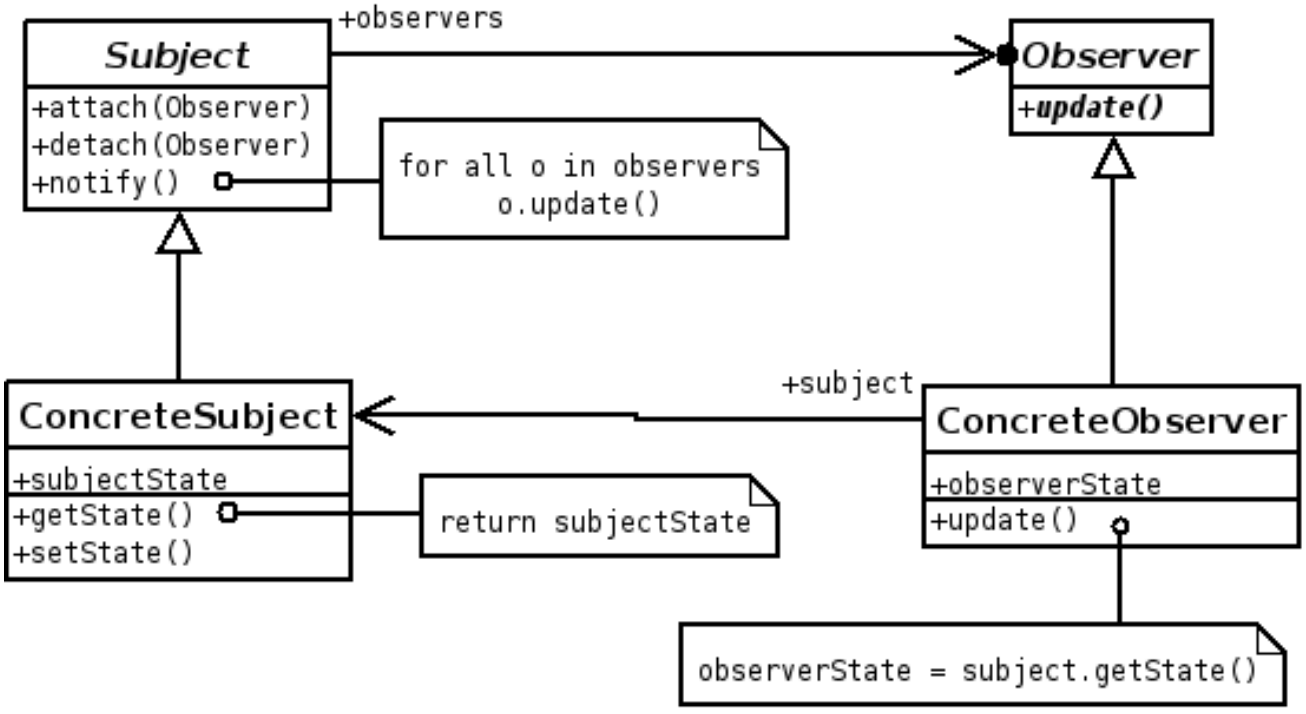
E a interface `Observavel`, que permite a adição de observadores:

```
public interface Observavel {
    void add(Observador o);
}
```

Uma implementação de `Observavel` é capaz de adicionar observadores e notificá-los quando necessário:

```
public class CampoDeTexto implements Observavel {

    private String texto = "";

    private final List<Observador> observadores = new ArrayList<Observador>();

    public void add(Observador o) {
        this.observadores.add(o);
    }

    public void algumMetodo() {
        // notifica os observadores
        for(Observador o : this.observadores) {
            o.notifica(this);
        }
    }

}
```

É comum o uso de `Generics` na interface `Observavel` para que o método de notificação receba uma referência exata ao objeto desejado a fim de evitar o *casting* necessário no exemplo anterior.




## 8.9 Exercícios

1) As interfaces de Listeners do Swing tem uma mãe em comum. Quem ela é? Consulte o javadoc

2) Como será que o Hibernate avisa sobre eventos de mudança de estado em determinados objetos?

## 8.10 Visitor

Algumas vezes é preciso coletar dados de elementos da estrutura de um objeto ou adicionar uma nova operação baseado nessa estrutura.

O padrão **Visitor** é uma maneira elegante de atravessar uma estrutura. Isso pode ser util na hora de gerar um relatório ou analisar um objeto do seu modelo de dominio. O **Visitor** deve ter um método **visit** para qualquer objeto do dominio que será visitado.

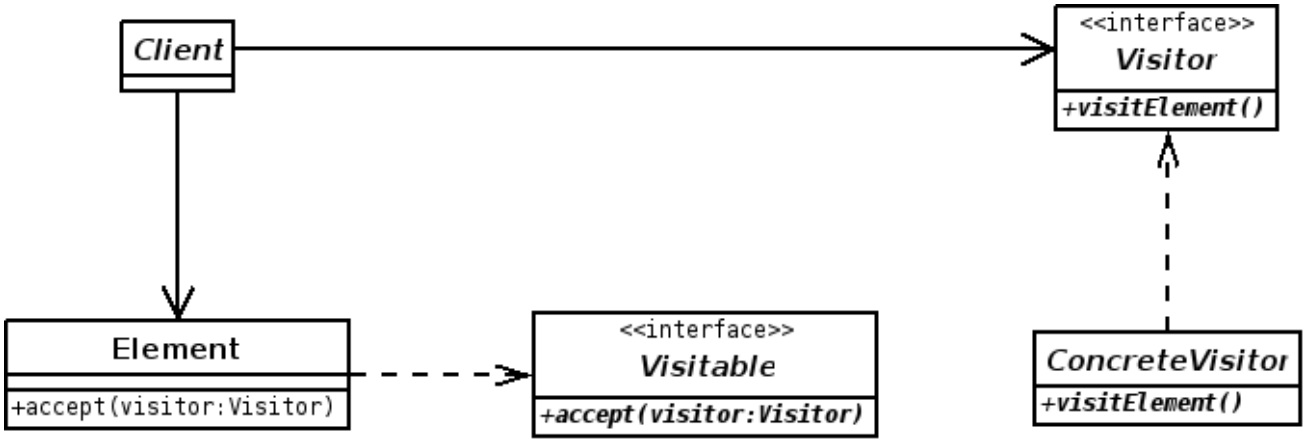
Por exemplo:

```
public class Visitor {

    public void visitProduto(Produto produto) {
        //analise do produto
    }
```

```
    public void visitCliente(Cliente cliente) {
        //analise do Cliente
    }
}
```

Além disso é preciso uma interface **Visitable** com um método **accept(Visitor visitor)**. Qualquer objecto no domínio que gostario de ser visitado deve implementar essa interface.

```
public interface Visitable {
    void accept(Visitor visitor);
}
public class Cliente implements Visitable{

    //atributos e métodos

    public void accept(Visitor visitor) {
        visitor.visitCliente(this);
    }
}
```

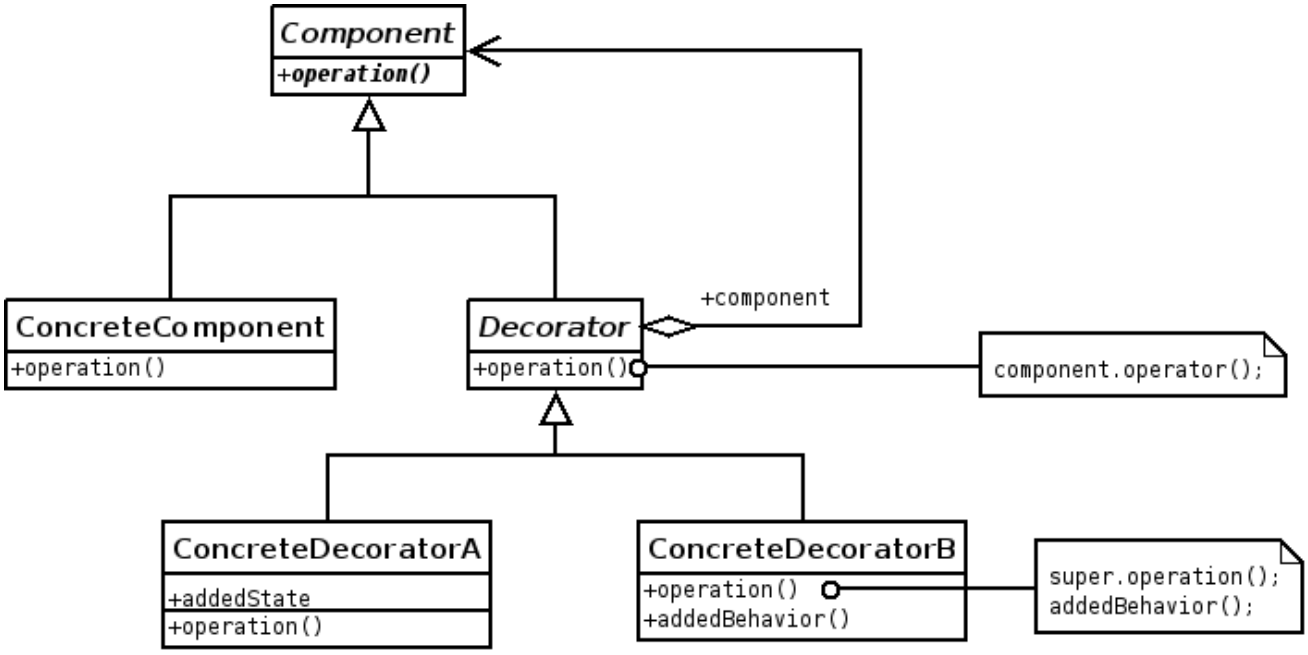



Algumas vezes visitors estão relacionados com Composites, como por exemplo o Graphics que visita todos os Components enquando é invocado o método `paint`.

## 8.11 Decorator

Algumas vezes criamos classes para simplesmente fazer algo a mais que alguma outra classe já faz. Esse algo a mais pode ser reaproveitado e reaplicado a outras classes.

As classes `BufferedReader` e `BufferedInputStream` do java.io são dois excelentes exemplos.




## 8.12 Exercício

1) Encontre outros decorators de `Reader`/`InputStream` dentro de `java.io`.

## 8.13 Composite

Queremos criar um sistema que renderiza componentes como HTML.

```
public interface HTMLizable {
    String toHTML();
}
```

Toda classe que implementar essa itnerface deve saber se renderizar como HTML. Exemplos seriam a classe de Italico e a de Negrito:

```
public class Italico implements HTMLizable {

    private String texto;

    public Italico(String texto) {
```

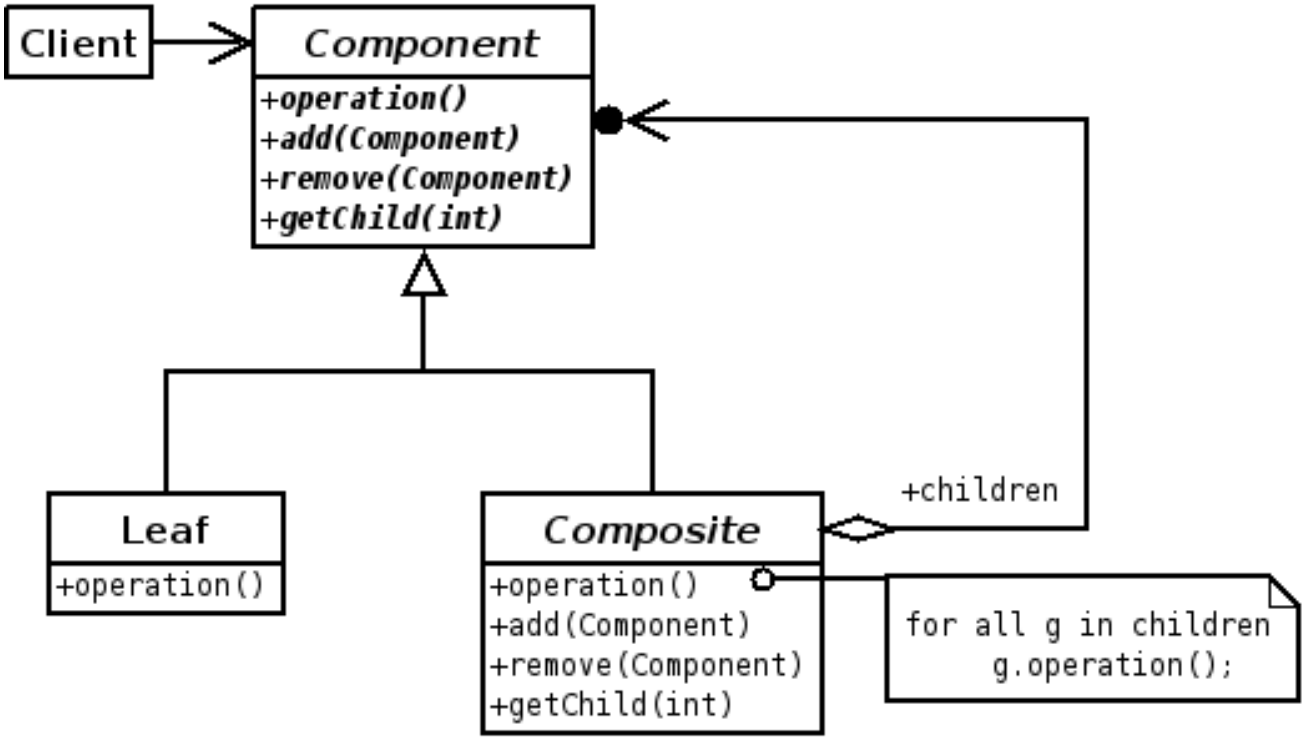
```
        this.texto = texto;
    }

    public String toHTML() {
        return "<i>" + texto + "</i>";
    }

}

public class Negrito implements HTMLizable {

    private String texto;

    public Negrito(String texto) {
        this.texto = texto;
    }

    public String toHTML() {
        return "<b>" + texto + "</b>";
    }

}
```

Algumas classes podem precisar de um conjunto de `HTMLizable` para trabalhar. Chamamos elas de composite, que serão galhos da nossa árvore de componentes:

```
public class Paragraph implements HTMLizable {

    private List<HTMLizable> components = new ArrayList<HTMLizable>();

    public void add(HTMLizable html) {
        this.components.add(html);
    }

    public String toHTML() {
        String result = "<p>";
        for (HTMLizable h : components) {
            result += h.toHTML();
        }
        result += "</p>";
        return result;
    }
```

```
}
```

Quando renderizarmos um `Paragrafo`, boa parte do trabalho será delegado aos componentes que formam esse objeto composto. O dia que tiver um novo objeto componente, ele servirá para paragrafo sem ter de alterar nada o código pré existente.




## 8.14 Exercícios

1) Toda AWT aplica o Composite Pattern para trabalhar com janelas e componentes visuais. Um JFrame possui um método para adicionar botões, text fields, etc. Quem são as classes bases dessa hierarquia? Qual a desvantagem?

2) (opcional) Um perigo do composite pattern é um método acabar entrando em recursão infinita. Como isso ocorre? Como se precaver?

3) Outro exemplo real de Composite, até mais sofisticado, é como funcionam as classes JSF que renderizaram o html dos componentes visuais. Estude a respeito.

## 8.15 Template Method

As vezes temos grande parte de um problema resolvido, faltando apenas alguns detalhes para que aquela parte do programa esteja pronta. Essa pequena parte pode ser feita de diversas maneiras, que deixamos a cargo de classes diferentes.

A classe abstrata `java.io.InputStream` é um excelente exemplo. Ela possui um conjunto de métodos para leitura de bytes, porém apenas um deles é abstrato: o método `read` que lê apenas um único byte. Segue seu fonte:

```
public abstract int read() throws IOException;
```

Ele é abstrato pois essa classe não sabe exatamente de onde será realizada a leitura: da entrada padrão? de um arquivo? de uma socket? Esse comportamento vai ser definido através da reescrita desse método em uma de suas subclasses concretas: `FileInputStream`, `SocketInputStream`, `ByteArrayInputStream`, entre outras. Essas sim sabem realizar a operação de leitura de um byte.

Se a classe `InputStream` não sabe ler um byte, como então é possível existir um método read que recebe um array de bytes a ser preenchido pela leitura, que não seja abstrato? Vamos ver o fonte deste método:

```
public int read(byte b[]) throws IOException {
    return read(b, 0, b.length);
}
```

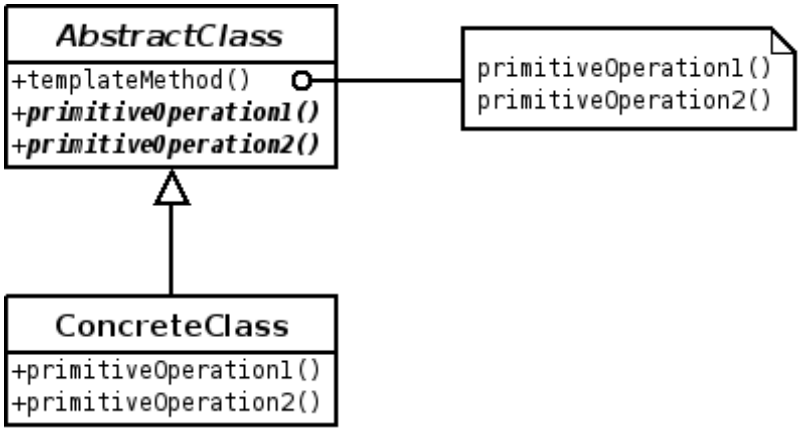
Este por sua vez esta invocando o método sobrecarregado do read que recebe, além da array a ser preenchida, a posição inicial e quantos bytes devem ser lidos. O fonte deste método está abaixo:

```
1  public int read(byte b[], int off, int len) throws IOException {
2     if (b == null) {
3        throw new NullPointerException();
4     } else if (off < 0 || len < 0 || len > b.length - off) {
5        throw new IndexOutOfBoundsException();
6     } else if (len == 0) {
7        return 0;
8     }
9
10    int c = read();
11    if (c == -1) {
12       return -1;
13    }
14
```

```
15      b[off] = (byte) c;
16
17      int i = 1;
18      try {
19          for (; i < len ; i++) {
20              c = read();
21              if (c == -1) {
22                  break;
23              }
24              b[off + i] = (byte)c;
25          }
26      } catch (IOException ee) {
27      }
28      return i;
29  }
```

Nas linhas 10 e 20 temos invocações ao método read que é abstrato! Isso é possível pois sabemos que não existe como instanciar a classe InputStream: ela é abstrata. Essa invocação recairá sobre um objeto que foi instanciado, logo ele possuirá uma implementação deste método read.

Os métodos read que lêem mais de um byte são templates: eles possuem o algoritmo em si, mas ainda falta um pouco para que toda a funcionalidade deles esteja pronta. Essa parte que falta é suprida com a implementação concreta do método read nas classes filhas de InputStream. Quando a classe filha implementa esse método, os demais métodos de InputStream que dependem deste (os template methods) estarão prontos para uso!




## 8.16 Exercícios

1) Veja a documentação da classe `InputStream` e veja quantas filhas ela possui

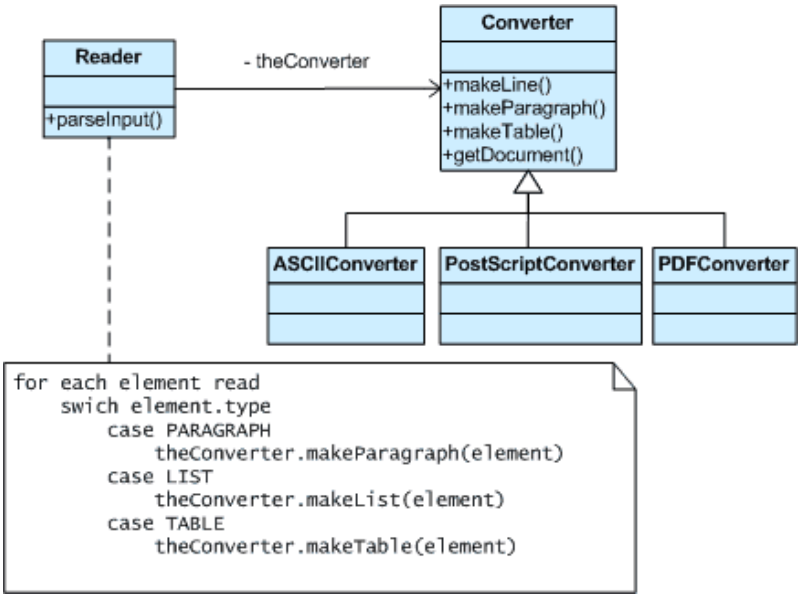
2) Outro bom exemplo de template method é a `AbstractCollection`. Para que ela serve? Quais são os métodos template dela?

3) Seria possível eliminar a herança de um template method? Como ficaria?

## 8.17 Builder

Muitas vezes escrevemos classes com construtores que possuem muitos parâmetros. Para facilitar a criação dos objetos dessas classes, ao invés de usarmos estes construtores, podemos criar uma outra classe cujo único objetivo é ter métodos que devem ser chamados para a construção do mesmo objeto. Um exemplo prático disso é o momento da criação da `SessionFactory` do Hibernate. Usamos, por exemplo, a classe `AnnotationConfiguration` para ir informando tudo que é necessário para a construção da `SessionFactory`. Por exemplo:

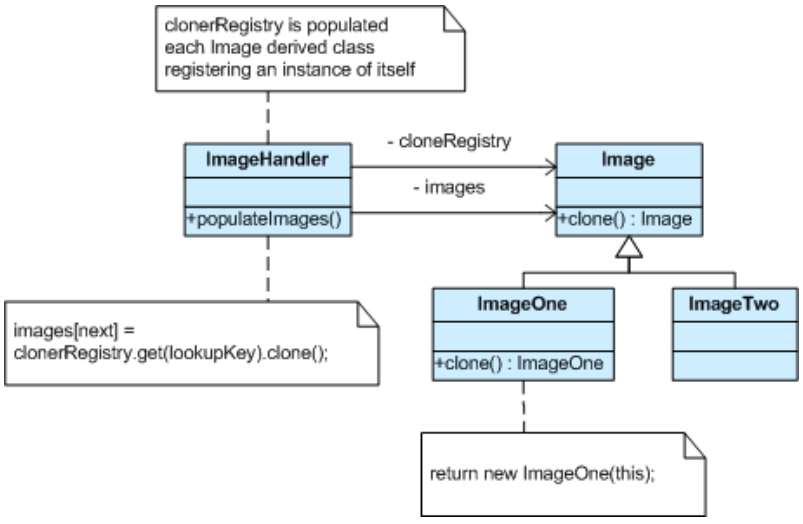
```
AnnotationConfiguration configuration = new AnnotationConfiguration();
configuration.addAnnotatedClass(Produto.class);
configuration.addAnnotatedClass(Categoria.class);
configuration.addAnnotatedClass(Usuario.class);
sessionFactory = configuration.buildSessionFactory();
```




## 8.18 Prototype

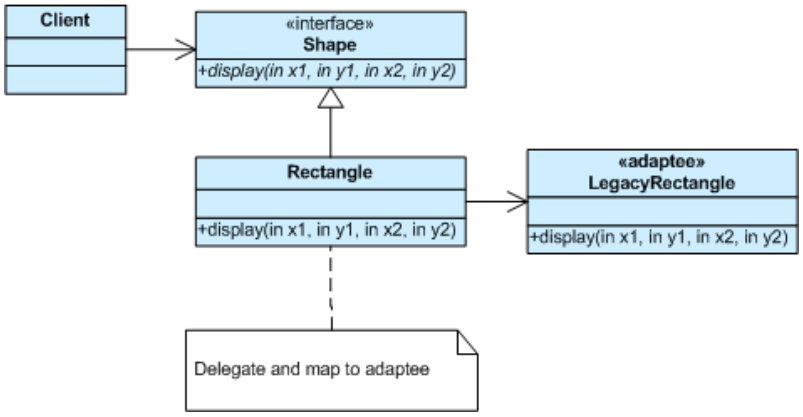
Muitas vezes precisamos criar objetos baseados em outros criados anteriormente. Normalmente o que é feito é copiar tudo de um para outro e entregar a nova instância idêntica a que serviu de base para a

cópia. Um exemplo disso em Java é o uso do método `clone` da classe `Object`.




## 8.19 Adapter

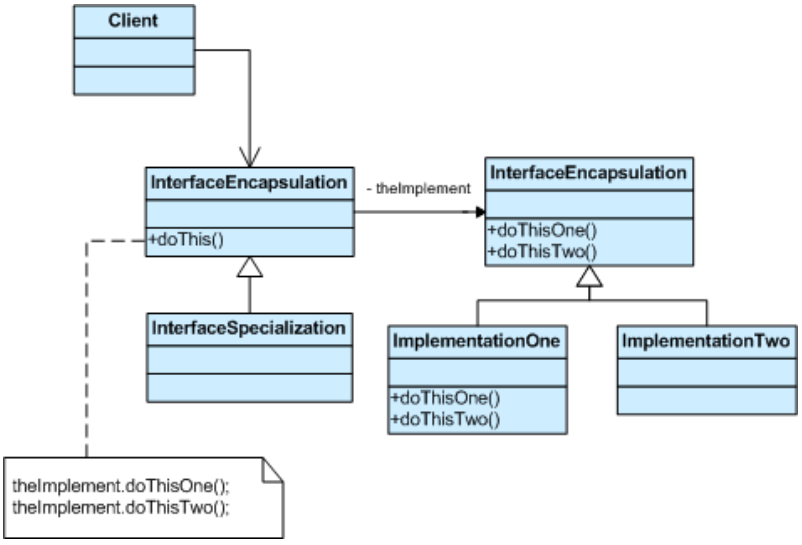
Algumas vezes precisamos realizar algumas operações sobre algum tipo de váriavel e nos deparamos com a impossibilidade da mesma. Imagine que estamos trabalhando com os tipos primitivos em Java, por exemplo, `int`. Quando queremos realizar operações como tranformá-lo para uma `String`, ou para base octal por exemplo, fazemos uso dos *Wrappers*, para o `int` usamos a classe `Integer`. O único objetivo é dar a possibilidade de realizar operações sobre um tipo o qual não podiamos antes.




## 8.20 Bridge

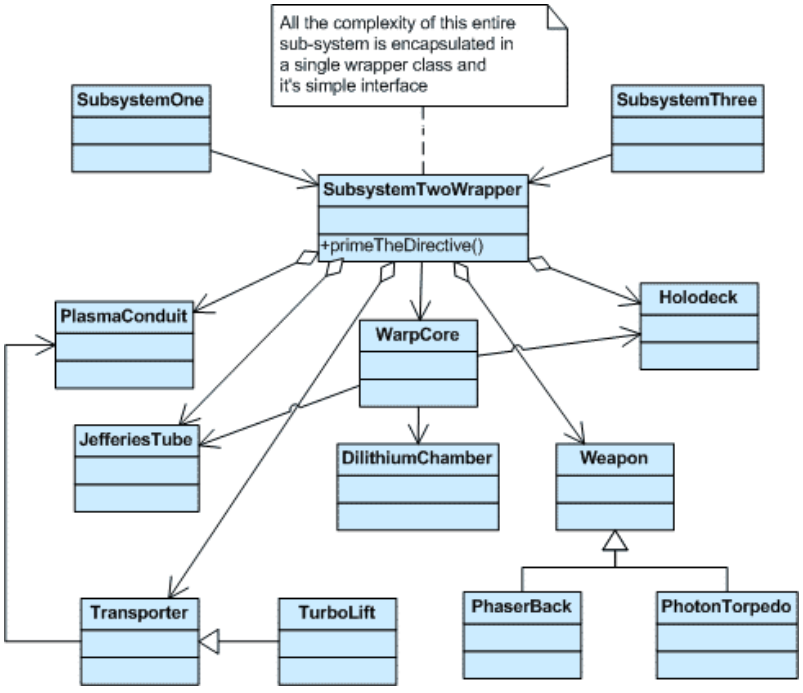
Imagine que precisamos aceitar diversas formas de pagamento num site de e-commerce. Imagine que na classe que representa nossa loja no sistema, precisamos lidar com todos as possiveis formas de pa-

gamento. Primeiramente podemos criar, por exemplo, uma *Interface* para representar a forma de pagamento e, ao invés de ficarmos instanciando as implementações dentro da nossa classe, podemos recebê-la como parâmetro fazendo uso da *Interface* previamente criada. Com isso, podemos sempre trocar a forma de pagamento sem que nossa loja precise ficar sofrendo alterações internas. A idéia é sempre deixar a nossa loja desacoplada da implementação das formas de pagamento.



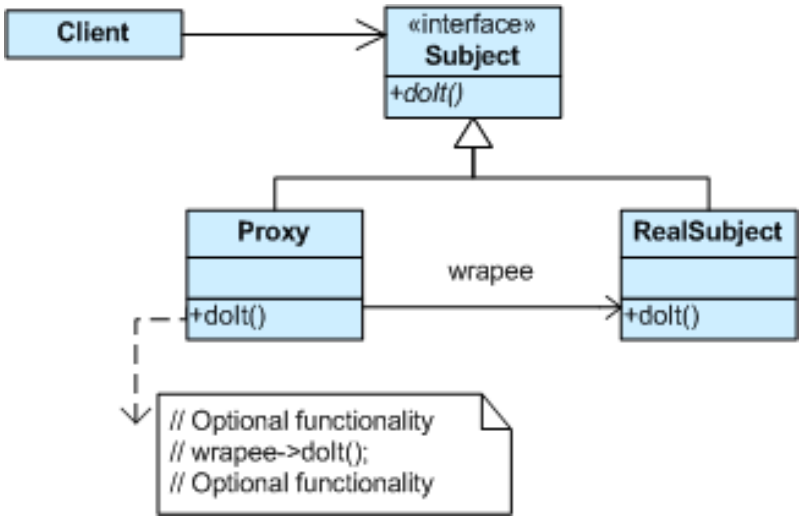

## 8.21 Façade

Muitas vezes acabamos com uma funcionalidade no sistema que para ser realizada precisamos que várias classes trabalhem em conjunto. O problema disso é que precisariamos conhecer todos os passos para executar determinada funcionalidade. Para diminuir essa complexidade criamos uma classe que encapsula toda essa lógica apenas exibindo um método simples para realizar essa tarefa complexa.



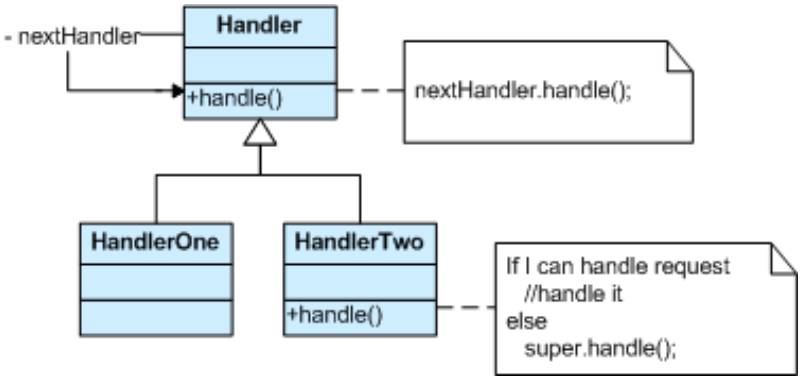

## 8.22 Proxy

Imagine que queremos realizar um log antes e depois de cada chamada de método para determinada classe na sua aplicação. Ao invés de espalharmos esse código em todos nossos métodos, podemos tentar isolar isso, garantindo que, sempre antes e depois de executar o método, um código escrito por nós vai ser executado. Uma maneira tradicional de fazer isso é utilizando Aspectos.



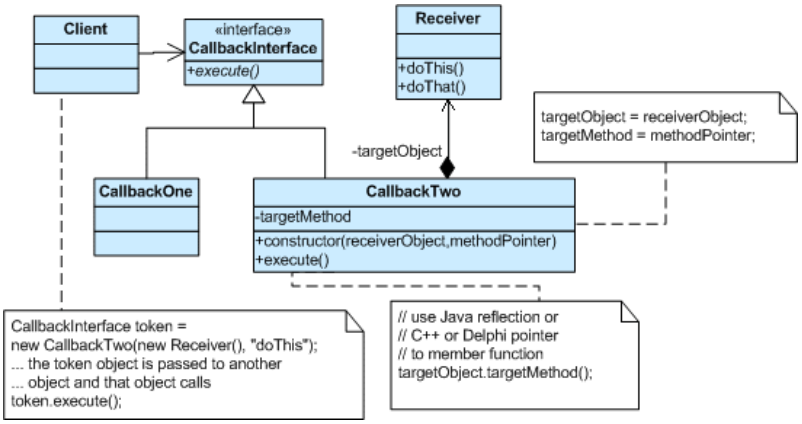
## 8.23 Chain of Responsibility

Imagine um framework como o VRaptor 3, sempre que acessamos algum recurso da aplicação, o VRaptor deve fazer uma série de verificações para saber se tudo está correto. Uma sequência simples seria: verificar se o método solicitado existe, checar o número de parâmetros, converter os parâmetros para os tipos específicos e, por fim, invocar o método por exemplo. A idéia é que o próximo passo só seja executado se o anterior foi executado corretamente. Essa é a idéia do *Chain of Responsibility*. Um detalhe interessante também é que todos esses objetos podem compartilhar das informações que são pertinentes a todos dado o contexto de execução.



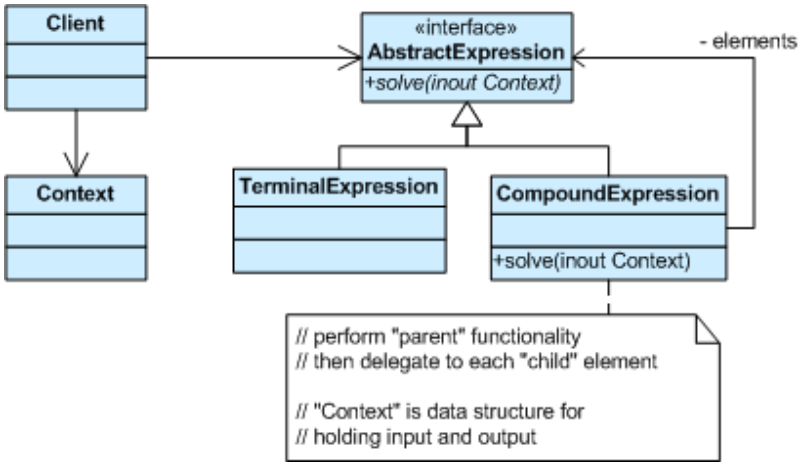

## 8.24 Command

Imagine que temos ações que devem ser realizadas por um sistema web, com por exemplo: criar ou listar produtos. Ao invés de colocarmos esse código todo em um mesmo lugar, criamos algumas classes cujo único objetivo é isolar essas ações. Quando usamos um framework como Struts 1, acabamos usando esse *Design Pattern* o tempo todo, pois estamos criando `Actions` para cada operação que queremos realizar no sistema.



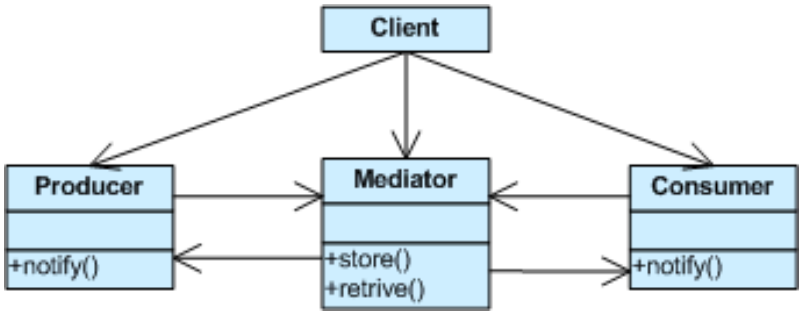
## 8.25 Interpreter

Imagine as linguagens de programação que usamos, Java por exemplo. Quando escrevemos o nosso código, o compilador tem que analisar se usamos as palavras corretas suportadas pela linguagem, se usamos numa sequência válida, etc... Para realizar essas verificações podemos criar classes que representem essas regras para serem executadas. É justamente para isso que serve o *Interpreter*.



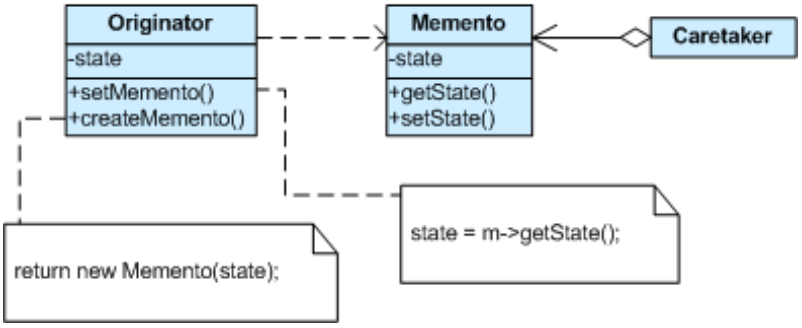

## 8.26 Mediator

Uma coisa que fazemos quando desenvolvemos aplicações usando linguagens orientadas a objeto, como Java, é que, na hora de fazermos a parte de acesso a persistência, sempre temos que ficar convertendo nossos objetos para *SQL* para conseguirmos executar os comandos necessários no banco de dados. Imagine que podemos passar esse trabalho de ficar interagindo com vários objetos para realizar essa conversão para uma classe. O trabalho dela justamente é coordenar todo o trabalho entre esses objetos.



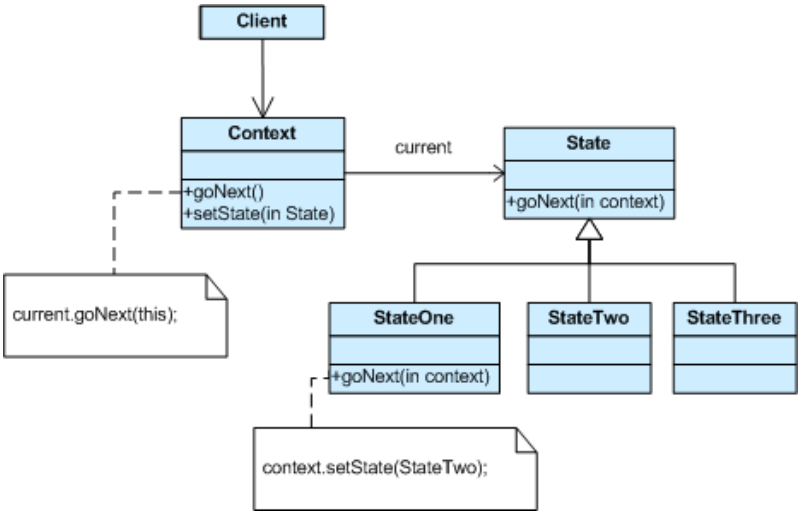
## 8.27 Memento

Sempre que queremos manter estado do objeto que estamos trabalhando. Imagine que o usuário começou a preencher uma ficha de alteração de dados, mas caso ele queira desistir e voltar para o estado anterior temos que dar essa possibilidade. Para isso podemos criar uma classe que guarde tudo que precise ser mantido para este objeto. Dessa maneira, a qualquer momento, podemos restaurar o estado prévio do objeto.




## 8.28 State

Imagine o processo de fechamento da compra num site de e-commerce. Quando falamos que queremos comprar os itens do nosso carrinho a loja tem que fazer uma série de verificações para saber se tudo está correto. Verificações possiveis são: checar estoque dos produtos do carrinho, validar dados do cartão de crédito, endereço de entrega, etc. Para representar esses estados no sistema, criamos classes que representam justamente cada uma das possiveis situações e também usamos uma classe para ir mantendo o estado atual do sistema. Com isso sabemos, por exemplo, que agora estamos fazendo a verificação de estoque, depois validando os dados do cartão, etc.



## 8.29 Patterns no DDD: Repository, Entity, VO, Service

Em muitos momentos, representar a necessidade do usuário na modelagem não é uma tarefa trivial, e mesmo com a Orientação a Objetos, pode não existir uma forma natural de traduzir as necessidades do cliente para o modelo. Devido a isso, o Domain Driven Design possui diversos Design Patterns para auxiliar na modelagem e no desenvolvimento da aplicação.

### Entity

Os objetos da nossa aplicação podem possuir diversos atributos, os quais definem suas características. No entanto, pode existir na mesma aplicação, objetos diferentes que representam a mesma informação. Mas como saber se eles se referem à mesma informação?

Para que se possa distinguir se ambos os diferentes objetos tratam da mesma informação é preciso verificar a igualdade de ambos, e para isso, deve-se existir um critério de igualdade definido para que seja possível determinar se os objetos se referem ou não a informações iguais.

Por exemplo, em um sistema de banco, controla-se a conta corrente dos clientes. A conta corrente é identificada pelo número da conta e pela código da agência. Vários objetos podem existir ao mesmo tempo, em diversos lugares da aplicação representado essa conta, ou seja, pode-se criar um objeto na hora de pagar a fatura do cartão de crédito da `Conta` e pode-se criar outro objeto `Conta` para efetuar o saque. Ambos os objetos `Conta` se referem à mesma conta, através das informações que identificam a conta, ou seja, o número da mesma e o código da agência bancária.

Uma classe que contém os identificadores da informação são as Entities (Entidades) do Domain Driven Design.

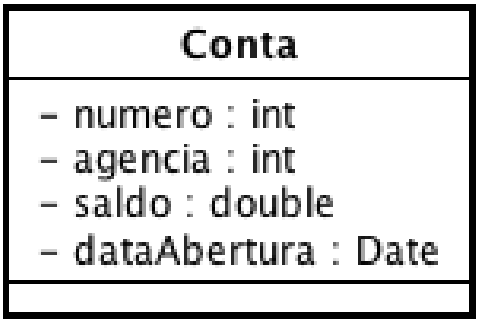



Figura 8.1: Entidade Conta, com seus respectivos identificadores

### Value Object

Da mesma forma que possuímos objetos que devem ter sua identidade definida, que são as entidades, que vimos anteriormente, temos os objetos que não possuem uma identidade explícita. No entanto,

esses objetos servem para complementar outros objetos, apenas adicionando valor aos mesmos. Um exemplo disso é o `Endereco`. Uma entidade `Cliente`, pode ter dentre várias informações possíveis o seu endereço, informação que várias outras entidades podem possuir. Para que não repliquemos as informações necessárias para o endereço do cliente, como rua, bairro, cidade e estado, criamos uma nova classe chamada `Endereco`. Essa classe não possui uma identidade, mas ela será utilizada para adicionar valor à uma outra classe.

Esses objetos que não possuem identidade, mas adicionam valor a outras classes são os *Value Objects*, que são representados na UML através de associações

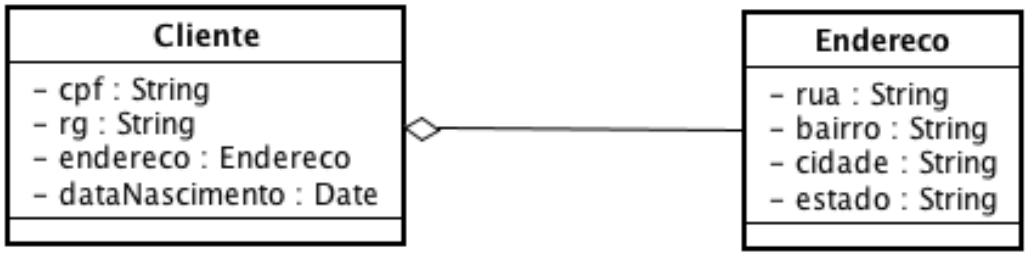



Figura 8.2: Um Value Object associado com uma entidade

### Service

Um dos conceitos importantes ao desenvolvermos *Entities* e *Value Objects* é que os métodos contendo suas regras de negócios devem estar sempre junto à classe que conterá essa regra. No entanto, é bastante comum encontrarmos regras de negócio que não são possíveis de encaixar em uma única entidade. O que fazemos? Replicamos a regra de negócio nas classes envolvidas?

Para resolver esse problema, quando uma determinada regra não se encaixa em uma única entidade ou *Value Object*, criamos uma classe que conterá esse método com a regra. Essa classe é o que é conhecido no Domain Driven Design como um *Service*.

### Repository

Uma das funcionalidades mais presentes nas aplicações atualmente é o fato de que elas devem poder guardar informações, seja em um banco de dados, seja em um arquivo ou uma outra estrutura qualquer. No entanto, se fossemos pensar no código que fará essa gravação, provavelmente caíremos em diversos detalhes de infra-estrutura, como pegar uma conexão com o banco de dados, ou abrir um arquivo e assim por diante.

Isso poderia nos causar um grande problema no momento em que estivéssemos desenvolvendo nossa aplicação no caso de espalharmos esse código de infra-estrutura por todo o nosso código, o que afetaria a manutenção e a legibilidade do nosso código.

O Domain Driven Design, sugere como alternativa para isolar esse código de baixo nível e deixar com o que o desenvolver se preocupe mais com o domínio os *Repositories*.

Um *Repository* nada mais é do que uma classe com uma interface voltado para o negócio que possui métodos para buscar as informações, guardar informações, alterá-las e removê-las. Dessa forma, é possível isolar todo o código de infra-estrutura para se trabalhar com os dados armazenados em uma classe que proverá uma interface de alto nível para se trabalhr com esses dados.

## 8.30 Mau uso de patterns: Singleton versus Injeção de Dependências

São diversos os artigos conhecidos que discutem sobre o mau uso do Singleton:

http://www.artima.com/weblogs/viewpost.jsp?thread=213214

http://blogs.msdn.com/scottdensmore/archive/2004/05/25/140827.aspx

http://www.prestonlee.com/archives/22

**O problema do singleton é ele ser usado por causa do acesso global**

Muitas vezes utilizamos o singleton apenas para não ter de ficar passando variáveis como argumento, e de dentro de algum método utilizamos o idiomismo `Singleton.getSingleton()` (`ConnectionPool.getPool().pegaConexao()`):

```
void metodo() {
    ConnectionPool.getPool().pegaConexao();
    // ...
}
```

Esse código é muito inflexível e fortemente acoplado. O código estaria quebrado se o `ConnectionPool` deixasse de ser singleton, ou se quisessemos que uma classe filha de `ConnectionPool` fosse usada.

Para consertar isso, podemos fazer com que o método receba o `ConnectionPool`:

```
void metodo(ConnectionPool pool) {
    pool.pegaConexao();
    // ...
}
```

Ou, mais ainda, podemos usar injeção de dependências por construtores:

```
class ClasseQuePrecisaDePool {

    private ConnectionPool pool;
    ClasseQuePrecisaDePool(ConnectionPool pool) {
        this.pool = pool;
    }

    void metodo() {
        pool.pegaConexao();
        // ...
    }
}
```

É claro que em algum lugar precisaresmo invocar `ConnectionPool.getPool().pegaConexao()`, mas o importante é diminuir a quantidade de invocação a esse método, para minimizar o impacto da mudança na manipulação dessa classe.

> **Mau sinal?**
>
> O uso de Design patterns vem sendo criticado por desenvolvedores de diversas linguagens. Se um problema sempre aparece recorrentemente e precisamos da mesma solução, não seria uma fraqueza e falta de expressividade da linguagem?
>
> http://blog.caelum.com.br/2006/12/17/design-patterns-um-mau-sinal/

## 8.31 Exercicios

1) Design patterns podem aparecer misturados. Como funcionaria um factory method que é um template method?

2) Imagine que queremos criar aquela nossa antiga classe `ControladorDeConta`, porém em vez de somar um número toda vez que for adicionada uma conta com saldo negativo, queremos que a classe `RelatorioDeDividas` seja notificada sobre a existência dessa conta. Que pattern usar?

3) Uma classe pesada é composta por diversos objetos pequenos. Esses objetos pequenos são inumeros e custam muita memória. Que pattern usar?

4) Outros patterns interessantes e que podem ser implementados através de um dynamic proxy são Proxy e Lazy Initialization. Leia a respeito.

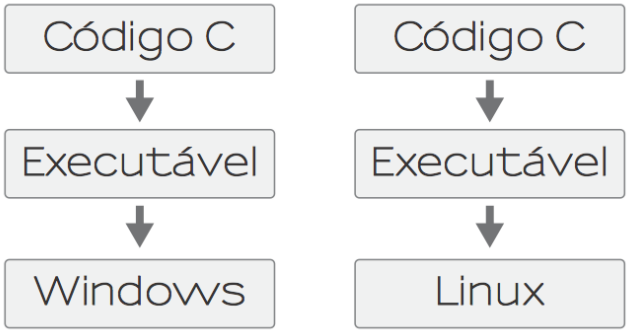
Capítulo 9

# Apêndice - Errata do Livro

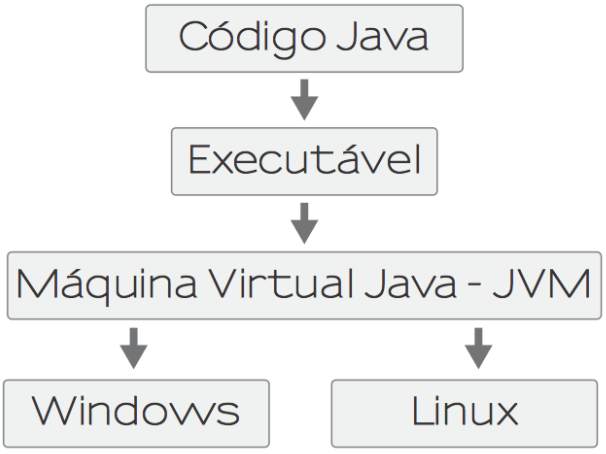
*"Experiência é apenas o nome que damos aos nossos erros."*
– Oscar Wilde

## 9.1 Errata com relação à primeira edição

### Página 9, Figura 1.1



### Página 10, Figura 1.2




### Página 127, primeiro código

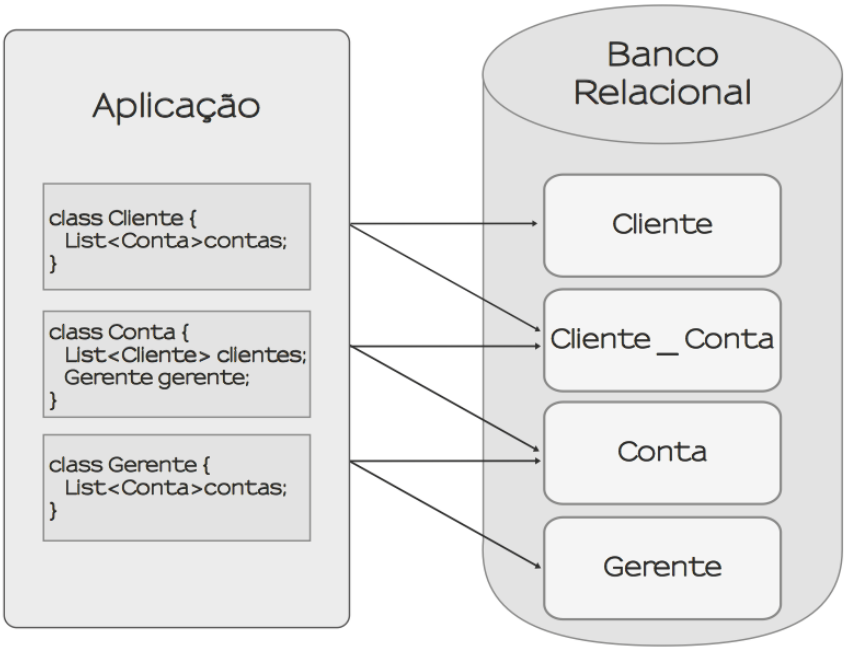
A variável `venda` deveria ter sido declarada antes do uso.

```
@Test
public void devePassarAVendaGeradaPorTodoOProcessoDeVenda() {
    Passo passo1 = mock(Passo.class); Passo passo2 = mock(Passo.class);
    Processo processo = new Processo(passo1, passo2);
    Vendas vendas = new Vendas(processo);
    Venda venda = vendas.vende(umPedido());

    verify(passo1.processa(venda));
    verify(passo2.processa(venda));
}
```

### Página 128, código

A implementação do método processa deveria ser `this.dao.salva(venda)`

### Página 160, figura 6.9




### Páginas 207 e 208, códigos

A variável passada para o método getOrcamento deveria ser pedido e não compra.

### Página 218, código XML

O fecha tags do campo <numeroDoCartao> deveria ser </numeroDoCartao>

### Página 225, código XML

O fecha tags do campo <valor> deveria ser </valor>

## 9.2 Erros de ortografia da primeira impressão

Página 7 - Duplicidade da expressão “fortemente adotada”

Página 11 - Java Virtual Machine Specification, faltou o i

Página 15 - Terceiro parágrafo Design by Committee.

Página 36 - Duplicidade da palavra “cuidar” no segundo parágrafo.

Pagina 41 - Última linha - concordância de plural - “resolução das classes”

Página 42- Primeira frase “... depois as classes do a aplicação ....”, deveria ser “da aplicação”.

Página 44- Último parágrafo, Java Database Connectivity, com dois Ns.

Página 52- Terceiro parágrafo, Java Database Connectivity, com dois Ns.

Página 54 - Último parágrafo, duplicidade da expressão “seriam as”

Página 70 - Nome do “Alan Kay” está “Key”

Pagina 70 - Nas duas ultimas linhas - a palavra “em” e o trecho “Actions do Struts” estão repetidas

Página 74 - Penúltimo parágrafo, duplicidade da expressão “as informações.” .

Página 80 - Faltou a preposição ‘que’, no terceiro parágrafo “... e isso faz com XXX sejam ...”

Página 81- Segundo parágrafo, linha 9, duplicidade da palavra “vezes”.

Página 81- Segundo parágrafo, linha 9, “É o que eu algumas vezes chamei da ‘separação de responsabilidades’ "seria de.

Página 82 e 83 - Duplicidade da expressão “uma referência a java.sql.Connection”

Página 88 - Último parágrafo, duplicidade da expressão “os métodos”

Página 129 - Última frase do segundo parágrafo, duplicidade da expressão “antes mesmo de”

Página 134 - Segundo parágrafo “... corrigir falhas e bugs que possam ter se introduzidos.” faltou o ‘S’.

Página 145- Primeiro parágrafo, penúltima frase possui dois pontos finais.

Página 150- Penúltima frase do terceiro parágrafo duplicidade da palavra VRaptor e Stripes nos concorrentes do Struts.

Página 151- Primeiro parágrafo e última frase, a figura referenciada é a 6.6 mas deveria ser 6.7.

Página 151- Segundo parágrafo e primeira frase, “...layer visual da aplicação do de domínio e negócio.” , retirar o de. E mudar para “do negócio”.

Página 175 - Faltou virgula depois de “DTO”

Página 182 - Segundo parágrafo - Na frase onde começa a falar de EJBs a palavra “como” esta repetida "

Página 207 - Na segunda linha está escrito “orquestration”, deveria ser orchestration.