



Edição 46

Projetando classes em Java

O papel do design no processo
de desenvolvimento de software

Aprenda a criar plug-ins para o Eclipse

Saiba como estender ou criar funcionalidades
para a principal IDE Java



PROGRAMAÇÃO ORIENTADA A OBJETOS

Conheça as vantagens sobre o
paradigma estruturado na prática

ISSN 2179625-4



DESENVOLVA PARA CORPORAÇÕES OU PARA O DIA A DIA COM JAVA



FORMAÇÃO DESENVOLVEDOR **JAVA**

DOMINE O DESENVOLVIMENTO DE APLICAÇÕES ORIENTADAS A OBJETO COM **UML** E APRENDA TÓPICOS AVANÇADOS DE PROGRAMAÇÃO COMO O USO DE **SERVLETS, JSP, FRAMEWORKS WEB DENTRO DO PADRÃO MVC**.

CERTIFICAÇÕES:

➤ Oracle Certified Professional, Java SE 6 Programmer

➤ Oracle Certified Professional, Java EE 5 Web Component Developer



Sumário

Conteúdo sobre Boas Práticas

06 – Programação Orientada a objetos x Programação Estruturada: Quem é melhor?

[John Soldera]

Conteúdo sobre Boas Práticas

17 – Projetando classes em Java

[José Fernandes A. Júnior]

Artigo no estilo Solução Completa

26 – Eclipse Plugins: Como criar plug-ins no Eclipse

[Eclipse Plugins: Como criar plug-ins no Eclipse]

CURSOS ONLINE

A Revista Java Magazine oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.



CONHEÇA ALGUNS DOS CURSOS:

- Curso de noSQL (Redis) com Java
- Curso Básico de JDBC
- Java Básico: Aplicações Desktop
- JSF com Primefaces
- Conhecendo o Apache Struts

Para mais informações :

www.devmedia.com.br/curso/javamagazine

(21) 3382-5038



Edição 46 • 2014 • ISSN 2173625-4



Assine agora e tenha acesso a todo o conteúdo da DevMedia:
www.devmedia.com.br/mvp

EXPEDIENTE

Editor

Eduardo Spínola (eduspinola@gmail.com)

Consultor Técnico Davi Costa (davigc_08@hotmail.com)

Produção

Jornalista Responsável Kaline Dolabella - JP24185

Capa e Diagramação Romulo Araujo

Atendimento ao leitor

A DevMedia possui uma Central de Atendimento on-line, onde você pode tirar suas dúvidas sobre serviços, enviar críticas e sugestões e falar com um de nossos atendentes. Através da nossa central também é possível alterar dados cadastrais, consultar o status de assinaturas e conferir a data de envio de suas revistas. Acesse www.devmedia.com.br/central, ou se preferir entre em contato conosco através do telefone 21 3382-5038.

Publicidade

publicidade@devmedia.com.br – 21 3382-5038

Anúncios – Anunciando nas publicações e nos sites do Grupo DevMedia, você divulga sua marca ou produto para mais de 100 mil desenvolvedores de todo o Brasil, em mais de 200 cidades. Solicite nossos Media Kits, com detalhes sobre preços e formatos de anúncios.

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique à vontade para entrar em contato com os editores e dar a sua sugestão!

Se você estiver interessado em publicar um artigo na revista ou no site Easy Java Magazine, entre em contato com o editor, informando o título e mini-resumo do tema que você gostaria de publicar:



EDUARDO OLIVEIRA SPÍNOLA

eduspinola.wordpress.com

@eduspinola / @Java_Magazine

CURSOS ONLINE

A Revista SQL Magazine oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.

Feita para Desenvolvedores de Software e DBAs



CONHEÇA OS CURSOS MAIS RECENTES:

- **Cursos: Curso de noSQL (Redis) com Java**
- **Desenvolvimento para SQL Server com .NET**
- **Curso PostgreSQL - Treinamento de banco de dados (Curso Básico)**

Para mais informações :

<http://www.devmedia.com.br/cursos/banco-de-dados>
(21) 3382-5038



DEVMEDIA



CRIANDO APLICAÇÕES INOVADORAS QUE VÃO MUDAR O MUNDO?

DESENVOLVA COM AS FERRAMENTAS QUE OS MELHORES DESENVOLVEDORES
USAM: GIT, REDMINE, JENKINS, SONAR, MAVEN E MUITO MAIS.

A suite ToolsCloud é um conjunto integrado das melhores ferramentas para desenvolvimento de software do mercado.

Um ambiente pronto para usar e com suporte, que os desenvolvedores curtem e os gerentes se surpreendem.

- Aumenta a produtividade;
- Facilita a colaboração e comunicação;
- Maior visibilidade e acompanhamento;
- Melhor planejamento e mais agilidade.

Na suite ToolsCloud você tem a sua disposição:

- As ferramentas mais usadas, integradas e de uso imediato;
- Repositório de código (git e svn), integração contínua (Jenkins/Hudson), Gerência de pendências (Redmine), e mais Maven, Nexus, Sonar;
- Infraestrutura robusta, em ambiente de nuvem, suporte e segurança;
- Facilita práticas de agilidade e lean startups;
- Ambiente isolado, customizável e flexível;
- Colaboração garantida: wiki, fóruns de discussão, chat (IRC).

Acompanhe a Toolscloud:

 toolscloud@toolscloud.com



twitter.com/toolscloud



ToolsCloud
toolscloud.com

Programação Orientada a objetos x Programação Estruturada: Quem é melhor?

Certamente, a programação estruturada é o primeiro paradigma que muitos desenvolvedores se deparam ao iniciar seus estudos. Tal paradigma leva a programas que são conhecidos por estruturarem as suas funcionalidades em sub-rotinas, geralmente levando à criação de sistemas caracterizados por conter grandes blocos de código-fonte ou sub-rotinas sem organização dos dados.

Por outro lado, a programação orientada a objetos sinaliza para uma maneira mais apropriada de realizar a correta divisão de um programa em conceitos (classes), permitindo que cada classe centralize em si todos os dados relativos ao conceito que ela representa, além de reunir todas as funcionalidades que pertencem a ela. Por exemplo, o conceito de pessoa origina a classe **Pessoa**, e esta contém atributos como nome, idade e CPF, além de todas as operações (métodos) que são realizadas acessando ou modificando o mesmo conceito.

Em muitas instituições de ensino ainda se começa a aprender programação com o estudo de linguagens que não são orientadas a objetos, ou pior, em muitas vezes se utilizam linguagens orientadas a objetos, mas os programas são criados da mesma forma que no paradigma estruturado.

A programação orientada a objetos é conhecida hoje por ser a sucessora do paradigma estruturado, porém não se deve descartar totalmente o conhecimento sobre esse antigo paradigma, pois ainda podemos chegar a situações onde deve-se dar manutenção a programas estruturados, ou ainda alguns determinados problemas de processamento em lote que podem ter uma melhor performance ao adotar o paradigma estruturado, uma vez que a quantidade de chamadas a métodos é reduzida.

De forma geral, os ganhos das linguagens de programação orientadas a objetos são concentrados na legibilidade do código, na facilidade de manutenção e expansão e na programação simplificada, como ocorre com a linguagem Java, que é interpretada, adicionando assim vantagens como ser independente de plataforma e ter uma sintaxe simples e legível, sem implicar negativamente na performance.

Fique por dentro

Esse artigo apresenta uma comparação entre o paradigma de programação estruturado e o orientado a objetos. Para isso, implementaremos um exemplo seguindo estes dois paradigmas e assim destacaremos as vantagens e facilidades que a Orientação a Objetos fornece ao desenvolvedor em termos de leitura, compreensão, manutenção e evolução do código.

Além disso, a programação orientada a objetos é a escolha natural em qualquer projeto de desenvolvimento de software, onde deseja-se modelar as entidades do mundo real manipulando-se informações das mesmas e realizando operações sobre os dados, o que facilita a análise e o projeto de software, em comparação com o paradigma de programação estruturado.

Dito isso, neste artigo serão mostrados vários exemplos de programação estruturada e sua conversão para a programação orientada a objetos, destacando as vantagens da programação orientada a objetos.

Primeiro exemplo (Programação Estruturada)

Para começar, apresentaremos um exemplo onde se deseja calcular o total de um pedido de venda feito em uma loja, que é uma situação corriqueira a ser solucionada por programação. Nesse exemplo, manipularemos dados de clientes, vendedores, produtos e do próprio pedido, incluindo os seus itens.

O objetivo é, a partir de uma série de dados, processar todos os itens de um pedido de venda e escrever um pequeno relatório incluindo o total do pedido. Vamos escrever esse programa na linguagem Java de forma orientada a objetos, mas primeiro apresentaremos uma versão “estruturada”, onde todo o processamento se concentra em um grande bloco de código.

Na **Listagem 1** foram apresentadas todas as variáveis do programa, contendo informações importantes sobre os produtos, clientes, o pedido e seus itens, os vendedores e demais dados do pedido, muitas vezes na forma de vetores.

Nas linhas 5, 6, 7 e 8 são apresentados os dados dos produtos existentes na loja, caracterizados pelas variáveis **codigoProdutos**, **nomeProdutos**, **valorProdutos** e **estoqueProdutos**. Todas essas variáveis são vetores de mesmo tamanho, sendo que os

elementos de mesmo índice em cada um desses vetores se referem ao mesmo produto.

O mesmo comportamento ocorre para as variáveis **codigoCientes**, **nomeCientes** e **enderecoCientes** (linhas 9, 10 e 11). Em seguida, os códigos dos produtos vendidos no pedido são mantidos na variável **codigoProdutoVendidos** (linha 12) e as quantidades de cada item são mantidas na variável **qtdVendaProdutos** (linha 13). Por sua vez, os códigos, nomes e percentuais de comissão de cada vendedor são atribuídos, respectivamente, às variáveis **codigoVendedores**, **nomeVendedores** e **percentualComissaoVendedores** (linhas 14, 15 e 16).

Listagem 1. Aplicação exemplo – versão estruturada (primeira parte).

```
01. package Exemplo1;
02.
03. public class Programa1 {
04.
05.     private static int[] codigoProdutos;
06.     private static String[] nomeProdutos;
07.     private static float[] valorProdutos;
08.     private static int[] estoqueProdutos;
09.     private static int[] codigoCientes;
10.    private static String[] nomeCientes;
11.    private static String[] enderecoCientes;
12.    private static int[] codigoProdutoVendidos;
13.    private static int[] qtdVendaProdutos;
14.    private static int[] codigoVendedores;
15.    private static String[] nomeVendedores;
16.    private static float[] percentualComissaoVendedores;
17.    private static int codigoClienteVenda;
18.    private static int codigoVendedorVenda;
```

Por fim, o código do cliente que fez a compra é mantido na variável **codigoClienteVenda** (linha 17) e o código do vendedor que realizou a venda é mantido na variável **nomeVendedorVenda** (linha 18).

Na **Listagem 2** é apresentado o método para popular essas variáveis para a aplicação exemplo, gerando nosso conjunto de dados.

Em aplicações reais, teríamos muito mais dados a serem carregados no sistema e os mesmos seriam obtidos a partir de um banco de dados, porém carregar dados fixos é conveniente para a nossa aplicação exemplo, pois manipular um pequeno número de clientes, produtos e vendedores torna a aplicação exemplo mais didática. Continuando a declaração da classe **Programa1**, na **Listagem 3**, é apresentado o método **main()**, responsável por realizar a venda, deduzindo os itens vendidos do estoque, e exibir um relatório na tela.

Observe que na linha 37 são carregados os dados chamando o método **criarDadosTeste()** e na linha 38 é realizada a venda, deduzindo as quantidades vendidas do estoque e calculando o valor total do pedido pelo método **realizarVenda()**, que é declarado na **Listagem 4**, onde a sua primeira parte é mostrada.

Nota-se que a lista de produtos vendidos é percorrida (linha 46) e também a lista de produtos disponíveis no sistema (linha 47). Quando um produto vendido é encontrado na lista de produtos

disponíveis (linha 50), seus dados são acessados (linhas 51 a 54), obtendo-se o nome do produto (linha 51), o valor do produto (linha 52), o estoque atual do produto (linha 53) e a quantidade vendida (linha 54).

Listagem 2. Aplicação exemplo – versão estruturada (segunda parte).

```
19. private static void criarDadosTeste() {
20.     codigoProdutos = new int[] { 1, 2, 3, 4 };
21.     nomeProdutos = new String[] { "Mesa", "Cadeira", "Fogão", "Sofá" };
22.     valorProdutos = new float[] { 500, 150, 1000, 2000 };
23.     estoqueProdutos = new int[] { 10, 20, 5, 4 };
24.     codigoCientes = new int[] { 1, 2, 3 };
25.     nomeCientes = new String[] { "João", "Carlos", "Carina" };
26.     enderecoCientes = new String[] { "Rua X, 100", "Rua Y, 200", "Rua Z, 400" };
27.     codigoProdutoVendidos = new int[] { 1, 2, 3 };
28.     qtdVendaProdutos = new int[] { 1, 6, 1 };
29.     codigoVendedores = new int[] { 1, 2 };
30.     nomeVendedores = new String[] { "Ademar", "Rosália" };
31.     percentualComissaoVendedores = new float[] { 0.1f, 0.2f };
32.     codigoClienteVenda = 2;
33.     codigoVendedorVenda = 1;
34. }
```

Listagem 3. Aplicação exemplo – versão estruturada (terceira parte).

```
35. public static void main(String[] args) {
36.     System.out.println("Iniciando Venda (1).");
37.     criarDadosTeste();
38.     realizarVenda();
39.     System.out.println("Venda Concluída.");
40. }
```

Listagem 4. Aplicação exemplo – versão estruturada (primeira parte do método **realizarVenda()**).

```
41. private static void realizarVenda() {
42.
43.     float totalPedido = 0;
44.
45.     // Remove os produtos vendidos do estoque e calcula o total:
46.     for (int contProdutoVendido = 0; contProdutoVendido < codigoProdutoVendidos.length; contProdutoVendido++) {
47.         int codigoProdutoVendido = codigoProdutoVendidos[contProdutoVendido];
48.         for (int contProduto = 0; contProduto < codigoProdutos.length; contProduto++) {
49.             int codigoProduto = codigoProdutos[contProduto];
50.             if (codigoProdutoVendido == codigoProduto) {
51.                 String nomeProduto = nomeProdutos[contProduto];
52.                 float valorProduto = valorProdutos[contProduto];
53.                 int estoqueProduto = estoqueProdutos[contProduto];
54.                 int qtdItem = qtdVendaProdutos[contProduto];
55.                 float totalItem = valorProduto * qtdItem;
56.                 estoqueProdutos[contProduto] = estoqueProduto - qtdItem;
57.                 totalPedido = totalPedido + totalItem;
58.                 System.out.println("Item Pedido: Produto:");
59.                 System.out.println(" Nome do produto = " + nomeProduto);
60.                 System.out.println(" Estoque Anterior = " + estoqueProduto);
61.                 System.out.println(" Quantidade Item = " + qtdItem);
62.                 System.out.println(" Valor = " + valorProduto);
63.                 System.out.println(" Total Item = " + totalItem);
64.             }
65.         }
66.     }
```

O total do item corrente é calculado na linha 55, multiplicando o valor do produto pela quantidade vendida. Em seguida, a quantidade vendida é removida do estoque (linha 56) e a variável

Programação Orientada a objetos x Programação Estruturada: Quem é melhor?

totalPedido é somada ao total do item (linha 57), fazendo com que no final do processamento desse bloco de código se tenha o valor total do pedido, incluindo a criação de um relatório que descreve os detalhes de cada item (linhas 58 a 63).

Dando continuidade à declaração do método **realizarVenda()**, na **Listagem 5** é apresentado o cálculo da comissão do vendedor.

Listagem 5. Aplicação exemplo – versão estruturada (segunda parte do método **realizarVenda()**).

```
67. // Busca nome do vendedor e calcula a comissão:  
68. float percentualComissao = 0;  
69. for (int contVendedor = 0; contVendedor < codigoVendedores.length;  
       contVendedor++) {  
70.   int codigoVendedorLista = codigoVendedores[contVendedor];  
71.   if (codigoVendedorLista == codigoVendedorVenda) {  
72.     percentualComissao = percentualComissaoVendedores[contVendedor];  
73.     String nomeVendedor = nomeVendedores[contVendedor];  
74.     System.out.println("Nome do Vendedor:" + nomeVendedor);  
75.     System.out.println("Percentual de Comissão:" + percentualComissao);  
76.     break;  
77.   }  
78. }  
79. float totalComissao = totalPedido * percentualComissao;
```

Observe que a lista de vendedores é percorrida (linha 69). Ao encontrar o vendedor que realizou o pedido (linha 71), o percentual de comissão e o nome do vendedor são obtidos (linhas 72 e 73). Como resultado, um pequeno relatório é escrito nas linhas 74 e 75. Por fim, o total da comissão é determinado na linha 79, multiplicando o percentual de comissão pelo total do pedido.

Na **Listagem 6** é apresentado o código responsável por obter os dados do cliente e o valor total do pedido.

Listagem 6. Aplicação exemplo – versão estruturada (terceira parte do método **realizarVenda()**).

```
80. // Busca nome do cliente:  
81. for (int contCliente = 0; contCliente < codigoClientes.length; contCliente++) {  
82.   int codigoClienteLista = codigoClientes[contCliente];  
83.   if (codigoClienteLista == codigoClienteVenda) {  
84.     String nomeCliente = nomeClientes[contCliente];  
85.     String enderecoCliente = enderecoClientes[contCliente];  
86.     System.out.println("Nome do Cliente:" + nomeCliente);  
87.     System.out.println("Endereço do Cliente:" + enderecoCliente);  
88.     break;  
89.   }  
90. }  
91.  
92. // Acessa os dados totais:  
93. System.out.println("Valor Total do pedido:" + totalPedido);  
94. System.out.println("Valor Total da Comissão:" + totalComissao);  
95. }  
96.}
```

Repare que a lista de clientes é percorrida na linha 81 usando um laço de repetição, e ao encontrar o cliente que realizou o pedido (linha 83), seus dados são acessados (linhas 84 e 85) e impressos na tela (linhas 86 e 87). Por último, nas linhas 93 e 94, o valor total do pedido e da comissão são impressos.

O resultado da execução desse programa impresso no console é apresentado na **Listagem 7**.

Listagem 7. Aplicação exemplo – versão estruturada (resultado da execução).

```
Iniciando Venda (1).  
Item Pedido: Produto:  
Nome do produto = Mesa  
Estoque Anterior = 10  
Quantidade Item = 1  
    Valor = 500.0  
Total Item = 500.0  
Item Pedido: Produto:  
Nome do produto = Cadeira  
Estoque Anterior = 20  
Quantidade Item = 6  
    Valor = 150.0  
Total Item = 900.0  
Item Pedido: Produto:  
Nome do produto = Fogão  
Estoque Anterior = 5  
Quantidade Item = 1  
    Valor = 1000.0  
Total Item = 1000.0  
Nome do Vendedor: Ademar  
Percentual de Comissão: 0.1  
Nome do Cliente: Carlos  
Endereço do Cliente: Rua Y, 200  
Valor Total do pedido: 2400.0  
Valor Total da Comissão: 240.0  
Venda Concluída.
```

Uma constatação que pode ser feita sobre o exemplo apresentado é que para que determinado cliente seja acessado, é necessário percorrer a lista de clientes até que se encontre o cliente com o código informado. Felizmente existem outras formas de efetuar essa busca, como usando SQL sobre um banco de dados. Porém, nesse exemplo, utilizamos buscas sequenciais para fins didáticos.

Outro fato importante é que mesmo com o código-fonte escrito numa linguagem orientada a objetos, não quer dizer que esse código seja orientado a objetos, pois ele concentra em um único método todo o processamento da lógica de negócios, apresentando uma baixa otimização no acesso aos dados e falta de organização. Por outro lado, poderíamos criar estruturas de dados para cada conceito, como por exemplo, para os conceitos de cliente, vendedor e produto. No entanto, com isso, estaríamos apenas organizando essas variáveis em objetos separados, não acrescentando nenhuma melhoria relevante no sistema, uma vez que as otimizações necessárias são muito mais profundas.

Além disso, se dividíssemos o método **realizarVenda()** em três partes, sendo cada parte um método distinto, ainda estariamos tendo os mesmos problemas apresentados. Para realizar a conversão desse exemplo em um código totalmente orientado a objetos, precisamos reescrevê-lo totalmente, designando cada funcionalidade a uma classe distinta e determinando os atributos das classes e suas operações, inclusive criando referências entre os objetos, melhorando a performance.

Sendo assim, a seguir será apresentada a conversão para a forma puramente orientada a objetos do exemplo de programação

estruturada analisado anteriormente. A primeira classe, chamada de **Cliente** é declarada na **Listagem 8**.

Observe que na classe **Cliente** são declarados todos os atributos pertinentes ao conceito de cliente, como o código, o nome e o endereço (linhas 5 a 7). Adicionalmente, são declarados todos os métodos assessores desses mesmos atributos.

Dando continuidade ao exemplo orientado a objetos, na **Listagem 9** é declarada a classe **Vendedor**, que segue o mesmo padrão, concentrando os atributos e métodos relativos ao conceito de vendedor.

Na forma puramente orientada a objetos, as funcionalidades são designadas aos conceitos aos quais elas pertencem. Como existe informação suficiente relativa ao conceito de produto, é originada a classe **Produto**, declarada na **Listagem 10**, sendo mais uma das classes que representam os conceitos pertencentes à aplicação.

Além dos *getters* e *setters* clássicos, a classe **Produto** contém outros métodos especializados para manipular o atributo **estoque**, que são os métodos **entradaEstoque()** e **saidaEstoque()** (linhas 49 a 55). Estes possibilitam adicionar ou remover uma certa quantidade de produtos ao estoque, oferecendo uma forma mais intuitiva para manipular a variável relacionada (**estoque**).

Outra classe muito importante no exemplo puramente orientado a objetos é apresentada na **Listagem 11**, chamada de **ItemVenda**. Esta tem a função de representar os produtos que compõem uma venda, indicando a quantidade vendida e contendo uma referência ao produto vendido.

A classe **ItemVenda** contém dois atributos: o primeiro armazena a quantidade vendida e o segundo é uma referência para o produto vendido (linhas 5 e 6). Além dos *getters* e *setters*, **ItemVenda**

Listagem 8. Aplicação exemplo – versão orientada a objetos (classe Cliente).

```
01. package Exemplo2;
02.
03. public class Cliente {
04.
05.     private int codigo;
06.     private String nome;
07.     private String endereco;
08.
09.     public Cliente(int codigo, String nome, String endereco) {
10.         this.codigo = codigo;
11.         this.nome = nome;
12.         this.endereco = endereco;
13.     }
14.
15.     public int getCodigo() {
16.         return codigo;
17.     }
18.
19.     public void setCodigo(int codigo) {
20.         this.codigo = codigo;
21.     }
22.
23.     public String getNome() {
24.         return nome;
25.     }
26.
27.     public void setNome(String nome) {
28.         this.nome = nome;
29.     }
30.
31.     public String getEndereco() {
32.         return endereco;
33.     }
34.
35.     public void setEndereco(String endereco) {
36.         this.endereco = endereco;
37.     }
38. }
```

Conhecimento faz diferença!

The advertisement displays several issues of the 'engenharia de software magazine'. The visible issues include:

- Edição 29 :: Ano 3: Agilidade: Negociação de contratos em projeto; Projeto: Diagrama de sequência na prática; Projeto: Como inserir padrões de projeto através de refeirações – Parte 2.
- Edição 28 :: Ano 2: SOA: Processo e levantamento de requisitos de negócios – Parte 2; Qualidade de Software: Definição, características e importância.
- Edição 29 :: Ano 2: Gerenciamento de Configuração: Definição + Ferramentas; Processo: Executar testes funcionais com Hudson e Selenium RC.
- Edição 28 :: Ano 2: Evolução do software: Definições, preocupações e custo; Cuidados a serem tomados na implantação.

A large red starburst graphic in the bottom right corner contains the text '+ de 290 vídeos para assinantes'.

Faça já sua assinatura digital! | www.devmedia.com.br/es

Faça um upgrade em sua carreira

Em um mercado cada vez mais focado em qualidade, ter conhecimentos aprofundados sobre requisitos, metodologia, análises, testes, entre outros, pode ser a diferença entre conquistar ou não uma boa posição profissional. Sabendo disso a DevMedia lançou uma publicação totalmente especializada em Engenharia de Software. Todos os meses você pode encontrar artigos sobre Metodologias Ágeis; Metodologias tradicionais (document driven); ALM (application lifecycle); SOA (aplicações orientadas a serviços); Análise de sistemas; Modelagem; Métricas; Orientação à Objetos; UML; testes e muito mais. **Assine Já!**



DEV MEDIA

Programação Orientada a objetos x Programação Estruturada: Quem é melhor?

contém o método `calcularValorTotalItem()`, que multiplica a quantidade pelo valor de venda do produto, retornando o valor total do item.

Ademais, existe um conceito que representa a maior parte das regras de negócio existentes na aplicação, que é o conceito de pedido, implementado na classe **Pedido**. A primeira parte dessa classe é implementada na **Listagem 12**.

A classe **Pedido** contém todos os dados relativos ao conceito de pedido, que são **cliente**, **vendedor** e seus **itens** (linhas 5 a 7). Além disso, são declarados todos os *getters* e *setters* desses atributos. Continuando a declaração da mesma classe, na **Listagem 13** são declarados mais dois métodos.

Como explicitado no código apresentado, o método `calcularValorTotalPedido()` percorre a lista de itens (linha 40) com uma sintaxe mais limpa que no exemplo estruturado (sem usar variáveis de índices), e para cada item, obtém o valor total do mesmo chamando o método `calcularValorTotalItem()` (linha 41), disponibilizado na classe **ItemPedido**.

Outro método importante da classe **Pedido** é o `calcularComissaoPedido()` (vide linha 46). Este é responsável por calcular a comissão do pedido, invocando o método `calcularValorTotalPedido()` e multiplicando o resultado pelo percentual de comissão do vendedor.

Ao analisar as classes apresentadas, nota-se que cada variável está contida no conceito a qual ela pertence, tornando o código muito mais intuitivo, aumentando a sua capacidade de entendimento e manutenção.

Falta agora declarar o método que concentra a maior parte das funcionalidades da aplicação, chamado de `realizarVenda()`. Este método tem a função de processar todos os itens da venda, realizar a dedução das quantidades do estoque, realizar o cálculo da comissão do vendedor e imprimir os resultados na tela. Veja o código na **Listagem 14**.

Agora, com um código que realmente adota a orientação a objetos, o processamento de um pedido se tornou muito mais intuitivo, pois não precisamos mais da lógica de programação complicada

Listagem 9. Aplicação exemplo – versão orientada a objetos (classe Vendedor).

```
01. package Exemplo2;
02.
03. public class Vendedor {
04.
05.     private int codigo;
06.     private String nome;
07.     private float percentualComissao;
08.
09.     public Vendedor (int codigo, String nome, float percentualComissao) {
10.         this.codigo = codigo;
11.         this.nome = nome;
12.         this.percentualComissao = percentualComissao;
13.     }
14.
15.     public int getCodigo() {
16.         return codigo;
17.     }
18.
19.     public void setCodigo(int codigo) {
20.         this.codigo = codigo;
21.     }
22.
23.     public String getNome() {
24.         return nome;
25.     }
26.
27.     public void setNome(String nome) {
28.         this.nome = nome;
29.     }
30.
31.     public float getPercentualComissao() {
32.         return percentualComissao;
33.     }
34.
35.     public void setPercentualComissao(float percentualComissao) {
36.         this.percentualComissao = percentualComissao;
37.     }
38. }
```

Listagem 10. Aplicação exemplo – versão orientada a objetos (classe Produto).

```
01. package Exemplo2;
02.
03. public class Produto {
04.
05.     private int codigo;
06.     private String nome;
07.     private int estoque;
08.     private float valor;
09.
10.     public Produto(int codigo, String nome, int estoque, float valor) {
11.         this.codigo = codigo;
12.         this.nome = nome;
13.         this.estoque = estoque;
14.         this.valor = valor;
15.     }
16.
17.     public int getCodigo() {
18.         return codigo;
19.     }
20.
21.     public void setCodigo(int codigo) {
22.         this.codigo = codigo;
23.     }
24.
25.     public String getNome() {
26.         return nome;
27.     }
28.
29.     public void setNome(String nome) {
30.         this.nome = nome;
31.     }
32.     // código omitido...
33.
34.     public void entradaEstoque(int quantidade) {
35.         estoque += quantidade;
36.     }
37.
38.     public void saidaEstoque(int quantidade) {
39.         estoque -= quantidade;
40.     }
41. }
```

Listagem 11. Aplicação exemplo – versão orientada a objetos (classe ItemVenda).

```
01. package Exemplo2;
02.
03. public class ItemVenda {
04.
05.     private int quantidade;
06.     private Produto produto;
07.
08.     public ItemVenda(int quantidade, Produto produto) {
09.         this.quantidade = quantidade;
10.         this.produto = produto;
11.     }
12.
13.     public int getQuantidade() {
14.         return quantidade;
15.     }
16.
17.     public void setQuantidade(int quantidade) {
18.         this.quantidade = quantidade;
19.     }
20.
21.     public Produto getProduto() {
22.         return produto;
23.     }
24.
25.     public void setProduto(Produto produto) {
26.         this.produto = produto;
27.     }
28.
29.     public float calcularValorTotalItem(){
30.         return quantidade * produto.getValor();
31.     }
32. }
```

Listagem 12. Aplicação exemplo – versão orientada a objetos (classe Pedido - primeira parte).

```
01. package Exemplo2;
02.
03. public class Pedido {
04.
05.     private Cliente cliente;
06.     private Vendedor vendedor;
07.     private ItemVenda[] itens;
08.
09.     public Pedido(Cliente cliente, Vendedor vendedor, ItemVenda[] itens) {
10.         this.cliente = cliente;
11.         this.vendedor = vendedor;
12.         this.itens = itens;
13.     }
14.
15.     public Cliente getCliente() {
16.         return cliente;
17.     }
18.
19.     public void setCliente(Cliente cliente) {
20.         this.cliente = cliente;
21.     }
22.
23.     public Vendedor getVendedor() {
24.         return vendedor;
25.     }
26.
27.     public void setVendedor(Vendedor vendedor) {
28.         this.vendedor = vendedor;
29.     }
30.
31.     public ItemVenda[] getItens() {
32.         return itens;
33.     }
34.
35.     public void setItens(ItemVenda[] itens) {
36.         this.itens = itens;
37.     }
```

que era utilizada anteriormente para manipular os vetores de dados. Na versão orientada a objetos, as regras de negócio foram organizadas de acordo com o conceito a que elas pertencem.

Listagem 13. Aplicação exemplo – versão orientada a objetos (classe Pedido - segunda parte).

```
38.     public float calcularValorTotalPedido() {
39.         float valorTotal = 0;
40.         for (ItemVenda item : itens) {
41.             valorTotal = valorTotal + item.calcularValorTotalItem();
42.         }
43.         return valorTotal;
44.     }
45.
46.     public float calcularComissaoPedido() {
47.         return calcularValorTotalPedido() * vendedor.getPercentualComissao();
48.     }
```

Inicialmente, nas linhas 52 a 66, ocorre a retirada dos produtos vendidos do estoque. Neste trecho de código, é percorrida a lista de itens do pedido (linha 52) e o produto é acessado usando o *getter* `getProduto()` (linha 53). Adicionalmente, os dados do produto são acessados nas linhas seguintes e o método `calcularValorTotalItem()`, da classe `ItemVenda`, é chamado para calcular o total do item do pedido, sendo a quantidade vendida do item retirada do estoque na linha 49, usando o método `saidaEstoque()`.

Nas linhas 68 a 72 são obtidos e escritos os dados do vendedor e nas linhas 75 a 78 são acessados os dados do cliente do pedido. Por fim, o valor total do pedido é calculado na linha 81, usando o método `calcularValorTotalPedido()`, e o total da comissão é calculado pelo método `calcularValorComissao()` na linha 82, finalmente escrevendo a seguir os mesmos na tela, repetindo tudo o que foi feito na **Listagem 7**.

Comparando a versão estruturada com a versão puramente orientada a objetos, fica claro que a versão orientada a objetos soluciona o mesmo problema de forma muito mais intuitiva e simples, organizando os dados em conceitos representativos que contêm também métodos, os quais são referentes às operações do sistema.

A seguir, considerando o exemplo apresentado como uma parte de um projeto de software, será apresentada uma demanda de manutenção no código-fonte, em que é necessário adicionar novas funcionalidades.

Segundo exemplo (Programação Orientada a Objetos)

O próximo exemplo a ser analisado é dado pela adição de novas funcionalidades ao exemplo anterior, como se estivéssemos fazendo a manutenção de um sistema. A primeira alteração a ser implementada é a possibilidade de suportar clientes especiais, que têm um percentual de desconto a ser aplicado a qualquer compra que realizarem.

A segunda alteração a ser implementada é fazer com que os itens de pedido possam ser vendidos com desconto, ou seja, cada item de pedido deve conter também o percentual de desconto com que o produto foi vendido.

Programação Orientada a objetos x Programação Estruturada: Quem é melhor?

Listagem 14. Aplicação exemplo – versão orientada a objetos (classe Pedido - terceira parte).

```
49. public void realizarVenda() {  
50.  
51. // Remove os produtos vendidos do estoque:  
52. for (ItemVenda itemVenda : getItens()) {  
53. Produto produto = itemVenda.getProduto();  
54. String nomeProduto = produto.getNome();  
55. float valorProduto = produto.getValor();  
56. int estoqueProduto = produto.getEstoque();  
57. int qtdItem = itemVenda.getQuantidade();  
58. float totalItem = itemVenda.calcularValorTotalItem();  
59. produto.saidaEstoque(qtdItem);  
60. System.out.println("Item Pedido: Produto:");  
61. System.out.println(" Nome do produto = " + nomeProduto);  
62. System.out.println(" Estoque Anterior = " + estoqueProduto);  
63. System.out.println(" Quantidade Item = " + qtdItem);  
64. System.out.println(" Valor = " + valorProduto);  
65. System.out.println(" Total Item = " + totalItem);  
66. }  
67.
```



```
68. // Busca nome do vendedor e calcula comissão do vendedor:  
69. float percentualComissao = getVendedor().getPercentualComissao();  
70. String nomeVendedor = getVendedor().getNome();  
71. System.out.println("Nome do Vendedor: " + nomeVendedor);  
72. System.out.println("Percentual de Comissão: " + percentualComissao);  
73.  
74. // Busca nome do cliente:  
75. String nomeCliente = getCliente().getNome();  
76. String enderecoCliente = getCliente().getEndereco();  
77. System.out.println("Nome do Cliente: " + nomeCliente);  
78. System.out.println("Endereço do Cliente: " + enderecoCliente);  
79.  
80. // Acessa os dados totais:  
81. float totalPedido = calcularValorTotalPedido();  
82. float totalComissao = calcularComissaoPedido();  
83. System.out.println("Valor Total do pedido: " + totalPedido);  
84. System.out.println("Valor Total da Comissão: " + totalComissao);  
85. }  
86. }
```

Como verificaremos na solução puramente orientada a objetos, as duas alterações sugeridas são simples, porém, antes disso, vamos ver como elas são implementadas na versão estruturada, como mostra a **Listagem 15**, onde é apresentada a primeira parte da classe **Programa3**.

Na primeira parte da classe **Programa3**, é declarado o método **criarDadosTeste()**, que cria dados de teste para serem utilizados no exemplo, além do método **main()**, que realiza a venda, carregando antes os dados de teste. Continuando a declaração, a **Listagem 16** apresenta o método **realizarVenda()**.

A fim de implementar o desconto por produto vendido e o desconto dado por clientes especiais, são utilizados os vetores nas linhas 14, 20 e 21. Como retomamos ao exemplo do código estruturado, fica mais evidente a sua desvantagem em relação ao código puramente orientado a objetos, pois o que se tem é uma grande poluição visual causada pela falta de organização hierárquica de dados e operações, dificultando a manutenção.

Analisando o código apresentado, verifica-se que nas linhas 66 a 69 é calculado o desconto sobre os itens de venda (produtos), determinando o valor do desconto multiplicando o percentual de desconto pelo valor total do item. Por fim, nas linhas 87 a 101 foi calculado o desconto para clientes especiais, ou seja, se o cliente que efetuou a compra estiver na lista de clientes especiais **codigoClientesEspeciais**, é dado o devido desconto.

A manutenção do código fonte da nossa aplicação de pedidos pode ser facilitada se considerarmos dividir as variáveis e as funcionalidades em conceitos distintos. Se realizarmos a conversão desse exemplo estruturado para a forma puramente orientada a objetos e depois implementarmos as novas funcionalidades, teremos uma manutenção simplificada com alterações pontuais. Considerando que o código-fonte orientado a objetos será utilizado por uma organização que precisa crescer com o passar do tempo, é importante que sejam empregados outros conceitos da orientação a objetos, como herança, encapsulamento e polimorfismo, os quais maximizam a reusabilidade, eficiência e flexibilidade das implementações.

Listagem 15. Aplicação exemplo (segundo exemplo) – versão estruturada (Primeira Parte).

```
001. package Exemplo3;  
002.  
003. public class Programa3 {  
004.  
005. // código omitido...  
...  
014. private static float[] descontoProdutoVendido;  
015. // código omitido...  
...  
020. private static int[] codigoClientesEspeciais;  
021. private static float[] percentualDescontoClientesEspeciais;  
022.  
023. private static void criarDadosTeste() {  
024. codigoProdutos = new int[] { 1, 2, 3, 4 };  
025. nomeProdutos = new String[] { "Mesa", "Cadeira", "Fogão", "Sofá" };  
026. valorProdutos = new float[] { 500, 150, 1000, 2000 };  
027. estoqueProdutos = new int[] { 10, 20, 5, 4 };  
028. codigoClientes = new int[] { 1, 2, 3 };  
029. nomeClientes = new String[] { "João", "Carlos", "Carina" };  
030. enderecoClientes = new String[] { "Rua X, 100", "Rua Y, 200",  
"Rua Z, 400" };  
031. codigoProdutosVendidos = new int[] { 1, 2, 3 };  
032. qtdVendaProdutos = new int[] { 1, 6, 1 };  
034. descontoProdutoVendido = new float[] { 0.01f, 0.02f, 0.04f };  
035. codigoVendedores = new int[] { 1, 2 };  
036. nomeVendedores = new String[] { "Ademar", "Rosália" };  
037. percentualComissaoVendedores = new float[] { 0.1f, 0.2f };  
038. codigoClienteVenda = 2;  
039. codigoVendedorVenda = 1;  
040.  
041. codigoClientesEspeciais = new int[] { 2 };  
042. percentualDescontoClientesEspeciais = new float[] { 0.01f };  
043. }  
044.  
045. public static void main(String[] args) {  
046. System.out.println("Iniciando Venda (3)");  
047. criarDadosTeste();  
048. realizarVenda();  
049. System.out.println("Venda Concluída.");  
050. }  
051.
```

Listagem 16. Aplicação exemplo (segundo exemplo) – versão estruturada (Segunda Parte).

```
052. private static void realizarVenda() {  
053.  
054.     float totalPedido = 0;  
055.  
056.     // Remove os produtos vendidos do estoque  
057.     for (int contProdutoVendido = 0; contProdutoVendido <  
            codigoProdutosVendidos.length; contProdutoVendido++) {  
058.         int codigoProdutoVendido = codigoProdutosVendidos[contProdutoVendido];  
059.         for (int contProduto = 0; contProduto < codigoProdutos.length;  
                contProduto++) {  
060.             int codigoProduto = codigoProdutos[contProduto];  
061.             if (codigoProdutoVendido == codigoProduto) {  
062.                 String nomeProduto = nomeProdutos[contProduto];  
063.                 float valorProduto = valorProdutos[contProduto];  
064.                 int estoqueProduto = estoqueProdutos[contProduto];  
065.                 int qtdItem = qtdeVendaProdutos[contProduto];  
066.                 float descontoItem = descontoProdutoVendido[contProduto];  
067.                 float totalItem = valorProduto * qtdItem;  
068.                 float desconto = totalItem * descontoItem;  
069.                 totalItem = totalItem - desconto;  
070.                 estoqueProdutos[contProduto] = estoqueProduto - qtdItem;  
071.                 totalPedido = totalPedido + totalItem;  
072.                 System.out.println("Item Pedido: Produto:");  
073.                 System.out.println(" Nome do produto = " + nomeProduto);  
074.                 System.out.println(" Estoque Anterior = " + estoqueProduto);  
075.                 System.out.println(" Quantidade Item = " + qtdItem);  
076.                 System.out.println("Percentual Desconto = " + descontoItem);  
077.                 System.out.println(" Valor = " + valorProduto);  
078.                 System.out.println(" Valor Desconto = " + desconto);  
079.                 System.out.println(" Total Item = " + totalItem);  
  
080.     }  
081. }  
082. }  
083.  
084. // verifica se é um cliente especial e aplica desconto  
085. for (int contCliente = 0; contCliente <  
            codigoClientesEspeciais.length; contCliente++) {  
086.     {  
087.         int codigoClienteEspecial = codigoClientesEspeciais[contCliente];  
088.         if (codigoClienteVenda == codigoClienteEspecial) {  
089.             {  
090.                 float percentualDesconto = percentualDescontoClientesEspeciais  
                        [contCliente];  
091.                 float totalPedidoAntes = totalPedido;  
092.                 float descontoPedido = totalPedidoAntes * percentualDesconto;  
093.                 totalPedido = totalPedidoAntes - descontoPedido;  
094.             }  
095.             System.out.println(" Percentual Desconto Cliente Especial = " +  
                        percentualDesconto);  
096.             System.out.println(" Total Pedido Antes do Desconto = " +  
                        totalPedidoAntes);  
097.             System.out.println(" Valor do Desconto = " +  
                        descontoPedido);  
098.             System.out.println(" Total Pedido Antes do Desconto = " +  
                        totalPedido);  
099.             break;  
100.         }  
101.     }  
102. }
```

Portanto, convertendo para orientação a objetos a nossa aplicação estruturada (**Listagens 15, 16 e 17**), ao realizar a manutenção para adicionar as funcionalidades de clientes especiais e desconto por item de pedido, podemos ainda empregar herança, encapsulamento e polimorfismo para extrair mais proveito dos recursos da orientação a objetos.

Dentre esses, o conceito de encapsulamento já é utilizado na nossa aplicação, como visto nos diversos atributos **protected**, porém ainda podemos aplicar o conceito de herança sobre as classes **Cliente** e **Vendedor**, de forma a criar uma classe **Pessoa**, mãe de ambas, viabilizando a reutilização de código e levando a uma organização mais intuitiva das funcionalidades dessas classes.

Deste modo, a seguir é apresentada uma nova versão da aplicação exemplo, incorporando as duas novas funcionalidades de clientes especiais e descontos por item vendido, além de aproveitar melhor as relações de herança, encapsulamento e polimorfismo. Assim, facilitam-se muitas tarefas que ocorrerão em decorrência da expansão desse código na organização em que ele seria utilizado.

Como visto na **Listagem 18**, a classe **Pessoa** contém os atributos e métodos que são comuns às classes **Cliente** e **Vendedor**. De forma a especializar **Pessoa**, na **Listagem 19** é declarada a classe **Cliente**, herdando assim todos os atributos e métodos desta (com exceção dos privados). Outra classe importante é a classe **Vendedor**, declarada na **Listagem 20**, que também especializa **Pessoa**. Uma vantagem dessa nova organização é que a classe **Vendedor** se tornou menor e mais simples, pois herda os métodos da classe **Pessoa**, aumentando a legibilidade da aplicação.

Listagem 17. Aplicação exemplo (segundo exemplo) – versão estruturada (Terceira Parte).

```
103. // Busca nome do vendedor e calcula comissão do vendedor:  
104. float percentualComissao = 0;  
105. for (int contVendedor = 0; contVendedor < codigoVendedores.length;  
        contVendedor++) {  
106.     int codigoVendedorLista = codigoVendedores[contVendedor];  
107.     if (codigoVendedorLista == codigoVendedorVenda) {  
108.         percentualComissao = percentualComissaoVendedores[contVendedor];  
109.         String nomeVendedor = nomeVendedores[contVendedor];  
110.         System.out.println("Nome do Vendedor: " + nomeVendedor);  
111.         System.out.println("Percentual de Comissão: " + percentualComissao);  
112.         break;  
113.     }  
114. }  
115.  
116. float totalComissao = totalPedido * percentualComissao;  
117.  
118. // Busca nome do cliente:  
119. for (int contCliente = 0; contCliente < codigoClientes.length; contCliente++) {  
120.     int codigoClienteLista = codigoClientes[contCliente];  
121.     if (codigoClienteLista == codigoClienteVenda) {  
122.         String nomeCliente = nomeClientes[contCliente];  
123.         String enderecoCliente = enderecoClientes[contCliente];  
124.         System.out.println("Nome do Cliente: " + nomeCliente);  
125.         System.out.println("Endereço do Cliente: " + enderecoCliente);  
126.         break;  
127.     }  
128. }  
129.  
130. // Acessa a dados totais:  
131. System.out.println("Valor Total do pedido: " + totalPedido);  
132. System.out.println("Valor Total da Comissão: " + totalComissao);  
133. }  
134. }
```

Programação Orientada a objetos x Programação Estruturada: Quem é melhor?

Listagem 18. Aplicação exemplo (segundo exemplo) – versão orientada a objetos (classe Pessoa).

```
01. package Exemplo5;
02.
03. public class Pessoa {
04.
05.     private int codigo;
06.     private String nome;
07.
08.     public Pessoa(int codigo, String nome) {
09.         this.codigo = codigo;
10.         this.nome = nome;
11.     }
12.
13.     public int getCodigo() {
14.         return codigo;
15.     }
16.
17.     public void setCodigo(int codigo) {
18.         this.codigo = codigo;
19.     }
20.
21.     public String getNome() {
22.         return nome;
23.     }
24.
25.     public void setNome(String nome) {
26.         this.nome = nome;
27.     }
28. }
```

Listagem 19. Aplicação exemplo (segundo exemplo) – versão orientada a objetos (classe Cliente).

```
01. package Exemplo5;
02.
03. public class Cliente extends Pessoa {
04.
05.     private String endereco;
06.
07.     public Cliente(int codigo, String nome, String endereco) {
08.         super(codigo, nome);
09.         this.endereco = endereco;
10.     }
11.
12.     public String getEndereco() {
13.         return endereco;
14.     }
15.
16.     public void setEndereco(String endereco) {
17.         this.endereco = endereco;
18.     }
19. }
```

Listagem 20. Aplicação exemplo (segundo exemplo) – versão orientada a objetos (classe Vendedor).

```
01. package Exemplo5;
02.
03. public class Vendedor extends Pessoa {
04.
05.     private float percentualComissao;
06.
07.     public Vendedor(int codigo, String nome, float percentualComissao) {
08.         super(codigo, nome);
09.         this.percentualComissao = percentualComissao;
10.     }
11.
12.     public float getPercentualComissao() {
13.         return percentualComissao;
14.     }
15.
16.     public void setPercentualComissao(float percentualComissao) {
17.         this.percentualComissao = percentualComissao;
18.     }
19. }
```

Neste momento vale lembrar que a implementação de cliente especial na versão estruturada envolvia a criação de um vetor que continha os códigos dos clientes que eram especiais e outro vetor para manter os descontos dos mesmos, o que é uma péssima estratégia de codificação. Na programação orientada a objetos, pode ser aplicada uma solução muito mais elegante para a mesma funcionalidade: criar uma classe filha de **Cliente** chamada **ClienteEspecial** que, além de tudo o que um cliente faz, providencia a implementação do desconto de cliente especial.

Dessa forma, aplicamos diretamente o conceito de polimorfismo, pois podemos ter objetos da classe **Cliente** que são clientes normais e outros que são clientes especiais. Antes, por exemplo, para verificar se um determinado cliente era especial, era necessário percorrer a lista de clientes especiais procurando pelo código do cliente. Agora, para verificar se um cliente é especial, é necessário apenas utilizar o operador **instanceof**. Na **Listagem 21** é apresentada a classe **ClienteEspecial**, filha de **Cliente**, sendo também um exemplo de aplicação de herança.

Apresentada a classe **ClienteEspecial**, que introduz uma especialização de **Cliente**, declaramos, na **Listagem 22**, a classe **Produto**, que não sofreu nenhuma alteração em comparação com a versão anterior, puramente orientada a objetos (**Listagem 10**).

Listagem 21. Aplicação exemplo (segundo exemplo) – Versão orientada a objetos (classe ClienteEspecial).

```
01. package Exemplo5;
02.
03. public class ClienteEspecial extends Cliente {
04.
05.     private float percentualDescontoClienteEspecial;
06.
07.     public ClienteEspecial(int codigo, String nome, String endereco,
08.                           float percentualDescontoClienteEspecial) {
09.         super(codigo, nome, endereco);
10.         this.percentualDescontoClienteEspecial =
11.             percentualDescontoClienteEspecial;
12.     }
13.
14.     public float getPercentualDescontoClienteEspecial() {
15.         return percentualDescontoClienteEspecial;
16.     }
17.
18.     public void setPercentualDescontoClienteEspecial(
19.         float percentualDescontoClienteEspecial) {
20.         this.percentualDescontoClienteEspecial =
21.             percentualDescontoClienteEspecial;
22.     }
23.
24.     public float calculaDescontoPedido(float totalPedido) {
25.         return totalPedido * percentualDescontoClienteEspecial;
26.     }
27. }
```

Listagem 22. Aplicação exemplo (segundo exemplo) – Versão orientada a objetos (classe Produto).

```
01. package Exemplo5;
02.
03. public class Produto {
04.     //restante do código omitido...
```

A próxima classe a ser apresentada é a classe **ItemVenda**, introduzida na **Listagem 23**. Esta classe recebeu a adição do atributo **percentualDesconto** e novos métodos para calcular o total do item, além de considerar o desconto aplicável sobre os produtos vendidos em sua implementação.

Listagem 23. Aplicação exemplo (segundo exemplo) – Versão orientada a objetos (classe ItemVenda).

```

01. package Exemplo5;
02.
03. public class ItemVenda {
04.
05.     private int quantidade;
06.     private Produto produto;
07.     private float percentualDesconto;
08.
09.     public ItemVenda(int quantidade, Produto produto, float percentualDesconto) {
10.         this.quantidade = quantidade;
11.         this.produto = produto;
12.         this.percentualDesconto = percentualDesconto;
13.     }
14.
15.     //código omitido...
...
38.
49.     public float calcularValorTotalItemSemDesconto() {
50.         return quantidade * produto.getValor();
51.     }
52.
53.     public float calcularDescontoItem() {
54.         float totalItem = calcularValorTotalItemSemDesconto();
55.         float desconto = totalItem * percentualDesconto / 100;
56.         return desconto;
57.     }
58.
59.     public float calcularValorTotalItem() {
60.         return calcularValorTotalItemSemDesconto() - calcularDescontoItem();
61.     }
62.

```

Outra classe fundamental para a nossa solução orientada a objetos é declarada na **Listagem 24** e corresponde à classe **Pedido**. **Pedido** contém alterações devido à implementação das novas funcionalidades, de forma a suportar agora clientes especiais e descontos por itens de pedido.

Note que até a linha 45 não há nenhuma alteração em relação ao exemplo anterior, porém, depois disso, é exibida a nova implementação do método **calcularComissaoPedido()**, que verifica se o cliente é um cliente especial usando o operador **instanceof** (linha 49). Se for um cliente especial, temos um cálculo diferenciado (linhas 50 a 52), caso contrário, segue o mesmo cálculo usado anteriormente (linha 55).

No método **realizarVenda()**, nas linhas 62 a 70, são calculados os valores dos itens e deduzidas do estoque as quantidades vendidas. Em seguida, nas linhas 71 a 78, é impresso na tela um relatório que inclui os descontos individuais dos produtos. Na linha 81, por sua vez, chama-se o método para calcular o valor total do pedido. Veja a **Listagem 25**.

Listagem 24. Aplicação exemplo – Versão orientada a objetos (classe Pedido).

```

001. package Exemplo5;
002.
003. public class Pedido {
004.
005.     private Cliente cliente;
006.     private Vendedor vendedor;
007.     private ItemVenda[] itens;
008.
009.     public Pedido(Cliente cliente, Vendedor vendedor, ItemVenda[] itens) {
010.         this.cliente = cliente;
011.         this.vendedor = vendedor;
012.         this.itens = itens;
013.     }
014.
015.     // código omitido...
...
039.     public float calcularValorTotalPedido() {
040.         float valorTotal = 0;
041.         for (ItemVenda item : itens) {
042.             valorTotal = valorTotal + item.calcularValorTotalItem();
043.         }
044.         return valorTotal;
045.     }
046.
047.     public float calcularComissaoPedido() {
048.
049.         if (cliente instanceof ClienteEspecial) {
050.             ClienteEspecial clienteEspecial = (ClienteEspecial) cliente;
051.             float totalPedido = calcularValorTotalPedido();
052.             return (totalPedido - clienteEspecial.calculaDescontoPedido(totalPedido)) *
053.                 vendedor.getPercentualComissao();
053.         }
054.         else
055.             return calcularValorTotalPedido() * vendedor.getPercentualComissao();
056.     }
057.

```

Logo após, nas linhas 83 a 94, se o cliente for um cliente especial, são impressos seus dados na tela; teste que é feito através do uso do operador **instanceof** (linha 83). O objetivo desse bloco de código é calcular o desconto do cliente especial (linha 88), considerando esse valor no cálculo do valor total do pedido (linha 89). O restante do processamento do método **realizarVenda()** não foi alterado.

O resultado da execução do segundo exemplo estruturado e seu equivalente na forma orientada a objetos é apresentado na **Listagem 26**.

Quanto mais antigo o projeto de software, mais ele é parecido com o paradigma de programação estruturada, onde os programas tendem a apresentar aglomeração de comandos, criando grandes blocos de código. A evolução natural de tal paradigma é a programação orientada a objetos, que permite organizar dados e operações na forma de conceitos, ou seja, classes, simplificando os projetos de software em que a orientação a objetos é aplicada.

Entretanto, não é suficiente usar uma linguagem de programação orientada a objetos. Paralelamente, é de vital importância programar orientado a objetos, adotando sempre que possível herança, encapsulamento e polimorfismo, levando assim a uma maior facilidade de compreensão, manutenção e legibilidade do código.

Programação Orientada a objetos x Programação Estruturada: Quem é melhor?

Listagem 25. Aplicação exemplo – Versão orientada a objetos (classe Pedido).

```
058. public void realizarVenda() {  
059.  
060. // Remove os produtos vendidos do estoque  
061. for (ItemVenda itemVenda : getItens()) {  
062. Produto produto = itemVenda.getProduto();  
063. String nomeProduto = produto.getNome();  
064. float valorProduto = produto.getValor();  
065. int estoqueProduto = produto.getEstoque();  
066. int qtdItem = itemVenda.getQuantidade();  
067. float descontoItem = itemVenda.getPercentualDesconto();  
068. float desconto = itemVenda.calcularDescontoItem();  
069. float totalItem = itemVenda.calcularValorTotalItem();  
070. produto.saidaEstoque(qtdItem);  
071. System.out.println("Item Pedido: Produto:";  
072. System.out.println(" Nome do produto = " + nomeProduto);  
073. System.out.println(" Estoque Anterior = " + estoqueProduto);  
074. System.out.println(" Quantidade Item = " + qtdItem);  
075. System.out.println("Percentual Desconto = " + descontoItem);  
076. System.out.println(" Valor = " + valorProduto);  
077. System.out.println(" Valor Desconto = " + desconto);  
078. System.out.println(" Total Item = " + totalItem);  
079. }  
080.  
081. float totalPedido = calcularValorTotalPedido();  
082. float descontoPedido = 0;  
083. if (cliente instanceof ClienteEspecial) {  
084.  
085. ClienteEspecial clienteEspecial = (ClienteEspecial) cliente;  
086. float percentualDesconto =  
clienteEspecial.getPercentualDescontoClienteEspecial();  
087. float totalPedidoAntes = totalPedido;  
088. descontoPedido = clienteEspecial.calculaDescontoPedido  
(totalPedidoAntes);  
089. totalPedido = totalPedidoAntes - descontoPedido;  
090. System.out.println(" Percentual Desconto Cliente Especial = " +  
percentualDesconto);  
091. System.out.println(" Total Pedido Antes do Desconto = " +  
totalPedidoAntes);  
092. System.out.println(" Valor do Desconto = " + descontoPedido);  
093. System.out.println(" Total Pedido Antes do Desconto = " + totalPedido);  
094. }  
095.  
096. // Busca nome do vendedor e calcula comissão do vendedor:  
097. float percentualComissao = getVendedor().getPercentualComissao();  
098. String nomeVendedor = getVendedor().getNome();  
099. System.out.println("Nome do Vendedor: " + nomeVendedor);  
100. System.out.println("Percentual de Comissão: " + percentualComissao);  
101.  
102. // Busca nome do cliente:  
103. String nomeCliente = getClient().getNome();  
104. String enderecoCliente = getClient().getEndereco();  
105. System.out.println("Nome do Cliente: " + nomeCliente);  
106. System.out.println("Endereço do Cliente: " + enderecoCliente);  
107.  
108. // Acessa os dados totais:  
109. float totalComissao = calcularComissaoPedido();  
110. System.out.println("Valor Total do pedido: " + totalPedido);  
111. System.out.println("Valor Total da Comissão: " + totalComissao);  
112. }  
113. }
```

Listagem 26. Aplicação exemplo (segundo exemplo) – Resultado da execução.

```
Iniciando Venda (3).  
Item Pedido: Produto:  
Nome do produto = Mesa  
Estoque Anterior = 10  
Quantidade Item = 1  
Percentual Desconto = 0.01  
Valor = 500.0  
Valor Desconto = 5.0  
Total Item = 495.0  
Item Pedido: Produto:  
Nome do produto = Cadeira  
Estoque Anterior = 20  
Quantidade Item = 6  
Percentual Desconto = 0.02  
Valor = 150.0  
Valor Desconto = 18.0  
Total Item = 882.0  
Item Pedido: Produto:  
Nome do produto = Fogão  
Estoque Anterior = 5  
Quantidade Item = 1  
Percentual Desconto = 0.04  
Valor = 1000.0  
Valor Desconto = 40.0  
Total Item = 960.0  
Percentual Desconto Cliente Especial = 0.01  
Total Pedido Antes do Desconto = 2337.0  
Valor do Desconto = 23.369999  
Total Pedido Antes do Desconto = 2313.63  
Nome do Vendedor: Ademar  
Percentual de Comissão: 0.1  
Nome do Cliente: Carlos  
Endereço do Cliente: Rua Y, 200  
Valor Total do pedido: 2313.63  
Valor Total da Comissão: 231.36299  
Venda Concluída.
```

Autor



John Soldera

johnsoldera@gmail.com

É bacharel em Ciências da Computação pela UCS (Universidade de Caxias do Sul), mestre em Computação Aplicada pela Unisinos e cursa atualmente doutorado em Ciências da Computação pela UFRGS (Universidade Federal do Rio Grande do Sul). Trabalha com Java há 12 anos e possui a certificação SCJP.



Links:

Javadoc da plataforma Java SE 7.
<http://docs.oracle.com/javase/7/docs/api/>

Projetando classes em Java

Confira neste artigo um olhar sobre o papel do design no contexto do processo de desenvolvimento de software

Para conseguirmos criar um bom programa em uma linguagem orientada a objetos, temos que ter uma base sólida de fundamentos sobre como criar os nossos objetos. No caso da linguagem Java, como sabemos, o trabalho de instanciar estes objetos cabe às classes. Ao entendermos os bons princípios de desenhos de classes em Java, estaremos aptos a escrever unidades de código independentes e suficiente para serem reutilizadas de várias formas e mantendo um código robusto.

Para compreendermos a importância de um bom desenho de classes, vamos considerar um fabricante de automóveis. Uma grande empresa construtora de automóveis não fabrica todas as partes para cada um dos seus modelos. Várias peças, como rodas, bancos, pneus, sistema de freios, velas, etc., já vêm prontas, de diferentes fabricantes, e depois são montadas nos seus respectivos carros. Este modelo de fabricação especializado por peças ajuda a manter um custo baixo, ao mesmo tempo em que traz uma alta qualidade, criando uma resposta rápida e eficiente às necessidades do mercado. Assim, a principal responsabilidade do fabricante de automóveis é a de desenhar seus carros e montá-los a partir de peças já existentes.

A Programação Orientada a Objetos (POO) nasceu desta mesma ideia de se utilizar módulos pré-existentes na construção de aplicações. Este modelo de programação fornece aos desenvolvedores uma maneira natural e elegante de dividir aplicações em peças pequenas e reutilizáveis, chamadas de objetos, o que viabiliza a construção de aplicações de forma mais eficiente, através da reutilização das coleções de componentes disponíveis.

Assim como as construtoras de automóveis, temos disponível na linguagem Java várias peças, ou objetos, prontos para serem utilizados, quer sejam objetos do próprio framework, quer sejam objetos de terceiros (como da Apache, da SpringSource, etc.), que nos disponibilizam tais recursos através de arquivos JAR (Java ARchive).

Fique por dentro

Os criadores da linguagem Java estão constantemente aperfeiçoando a máquina virtual Java para eliminar os gargalos de performance, como a lentidão de métodos sincronizados e os custos do Garbage Collector (GC). O objetivo com essas melhorias é que os programadores não precisem mais se preocupar com tais gargalos ao projetarem os seus programas, devendo manter atenção apenas na implementação de soluções bem estruturadas. Mas no que constitui um bom desenho de um programa? Mostrar o que é e como alcançar esta meta é o objetivo deste artigo. Para isso, será explicado como desenvolver um programa em Java ajudando a pensar de forma planejada e estruturada antes de começar a escrever o código.

e uma classe já existente. Com o uso de herança, a informação se torna gerenciável por uma ordem hierárquica, como veremos nos exemplos mais à frente. Mas, como a linguagem Java suporta a herança entre objetos?

Sabemos que diferentes tipos de objetos têm, frequentemente, uma certa quantidade de características ou comportamentos em comum. Em Java, estas características e comportamentos são herdados de objetos mais genéricos para outros mais específicos, através da utilização de subclasses, o que é definido com a palavra **extends**. No entanto, a linguagem possui uma restrição que determina que uma classe possa estender apenas uma classe. Esta restrição é chamada de herança única, que evita vários problemas da herança múltipla (como acontece na linguagem C++) onde, por exemplo, um objeto pode herdar dois métodos com o mesmo nome de classes distintas, podendo assim entrar em conflito. Mas ao mesmo tempo que a linguagem Java coloca esta restrição por motivos de segurança, ela também fornece a simulação de herança múltipla através da implementação de interfaces (este assunto será tratado mais à frente), permitindo-nos usufruir dos seus maiores benefícios, sem sofrer dos seus inconvenientes.

Os aspectos mais importantes da herança que devemos ter em conta são: a reutilização de código, o polimorfismo e a invocação de métodos virtuais. Vejamos um exemplo prático para entendermos estes aspectos.

Projetando classes em Java

Herança na prática

Para entender melhor como funciona a herança em Java, vamos criar um programa que guarde as informações de empregados de uma empresa. Sabemos que existem diversos tipos de empregados, como gestores, engenheiros, programadores, secretárias, estagiários, etc. Cada um tem um tipo específico de trabalho e suas particularidades, mas todos eles são, para a empresa, um empregado. E para a empresa, todos os empregados devem ter, ao menos, um identificador único, um nome e um salário (para simplificar o exemplo). A **Listagem 1** mostra o código que representa o empregado.

Listagem 1. Classe que representa um empregado - Employee.

```
public class Employee {  
    private String _id;  
    private String _name;  
    private double _salary;  
  
    public Employee(String id, String name, double salary) {  
        _id = id;  
        _name = name;  
        _salary = salary;  
    }  
  
    public String getId() {  
        return _id;  
    }  
  
    public String getName() {  
        return _name;  
    }  
  
    public double getSalary() {  
        System.out.println("Salário do empregado.");  
        return _salary;  
    }  
}
```

Para deixarmos o exemplo um pouco mais simples, vamos representar apenas quatro tipos de empregados da empresa, a saber: gestores, engenheiros, estagiários e diretores. Por definição da empresa, o estagiário é um empregado sem qualquer regra específica sobre ele, ou seja, o que ele recebe de salário é o valor que fica guardado na variável **salary**, sem qualquer alteração, não tendo bônus ou qualquer outro benefício (nada contra os estagiários, estamos apenas simplificando o exemplo). Um gestor é um empregado que recebe um bônus fixo, de acordo com seu desempenho. Um engenheiro, por sua vez, é um empregado com bônus que varia segundo a sua categoria. E, finalmente, um diretor tem que ser obrigatoriamente um engenheiro, e possui, além do bônus que recebe como engenheiro, um subsídio para carro.

O estagiário

Um estagiário é um empregado sem qualquer regra específica sobre seu cargo. Dito isso, se um estagiário é um empregado, por regra da empresa, ele deve ter apenas um identificador único, um nome e um salário. Uma forma de representar um estagiário em código é escrever uma classe idêntica à **Employee** e mudar o nome para **Trainee**, por exemplo, como mostra a **Listagem 2**.

Listagem 2. Classe que representa um estagiário da empresa - Trainee.

```
1 public class Trainee {  
2     private String _id;  
3     private String _name;  
4     private double _salary;  
5  
6     public Trainee(String id, String name, double salary) {  
7         _id = id;  
8         _name = name;  
9         _salary = salary;  
10    }  
11  
12    public String getId() {  
13        return _id;  
14    }  
15  
16    public String getName() {  
17        return _name;  
18    }  
19  
20    public double getSalary() {  
21        return _salary;  
22    }  
23 }
```

Quanto código replicado, não é mesmo? Isso vai dificultar, entre outras coisas, a manutenção do código, já que, se houver alguma alteração na definição dos empregados, quer seja pela adição ou alteração de um campo, quer seja pela adição ou alteração de uma funcionalidade, teremos que alterar as mesmas características e métodos na classe **Trainee** e todas as outras classes que representam empregados da empresa. Contudo, como sabemos que um **Trainee** é um empregado e que a classe que representa o empregado já está definida, podemos lembrar do primeiro dos três aspectos mais importantes da herança, repensar a nossa classe e reutilizar o código já existente. Para isso, vamos estender a classe **Employee**, definindo uma herança entre esta e a classe **Trainee**, o que em Java é feito através da palavra-chave **extends**.

A **Listagem 3** mostra o novo código da classe **Trainee**, agora reutilizando código através da herança.

Listagem 3. Novo código da classe **Trainee**, reutilizando a classe **Employee** através de herança.

```
1 class Trainee extends Employee {  
2  
3     public Trainee(String id, String name, double salary) {  
4         super(id, name, salary);  
5     }  
6 }
```

Agora sim temos um código melhor. E quanta diferença! Passamos de 23 para seis linhas de código e termos exatamente a mesma funcionalidade. Como podemos notar pela **Listagem 3**, a classe **Trainee** estende **Employee**, o que pode ser compreendido como: "Um estagiário é um empregado". Deste modo, o objeto estagiário possui todas as características do objeto empregado (representado pela classe **Employee**) e qualquer alteração na definição de um

empregado (na classe **Employee**) será propagada para a classe **Trainee** ou qualquer classe que as estendam.

Para visualizar melhor esta hierarquia, observe a view *Type Hierarchy* do Eclipse (tecla de atalho *F4*), apresentada na **Figura 1**.

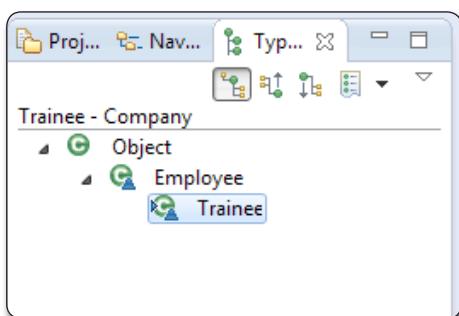


Figura 1. Hierarquia do objeto estagiário

Como podemos notar, a classe **Trainee** herda da classe **Employee** que, por sua vez, herda da classe **Object**. Vale lembrar aqui que todas as classes do Java herdam da classe **Object**, ou seja, tudo no Java são objetos.

Voltando ao código, a classe **Trainee** chama em seu construtor o método **super()**, que nada mais é que o “super construtor” desta classe, ou seja, o construtor que vem de cima, neste caso, da classe **Employee**. Se este método não for chamado no construtor da classe **Trainee**, teremos o erro: “O construtor implícito **super()** da classe **Employee** não está definido. Deve chamar explicitamente outro construtor”. Isso porque quando não chamamos o construtor **super()** nos construtores de nossas classes, o Java faz isso automaticamente, chamando o construtor padrão sem argumentos. No nosso caso, como a classe **Employee** define um construtor com argumentos, o construtor padrão deixa de existir, daí a mensagem de erro. Como podemos ver, uma subclasse herda todos os métodos e variáveis da superclasse, mas não herda os seus construtores!

Aos mais curiosos, agora pode ter surgido uma pergunta: E a classe **Employee**, que herda da **Object**, não deveria chamar o **super()** da classe **Object**? A resposta é sim, deve, e está fazendo isso. Apesar de não ter sido declarado explicitamente, a classe **Employee** (e todas as classes que não estendem outra classe) chama implicitamente o construtor **super()** (sem argumentos) da classe **Object**. É por esta razão que tudo em Java são objetos, e por isso que todos os objetos criados em Java possuem funcionalidades básicas, como: sincronização, garbage collection, etc., que são funcionalidades definidas na classe **Object** e herdadas por todas as outras classes.

Como temos herança simples em Java, se uma classe estende outra, que não seja a classe **Object** (como no caso da **Trainee**, que estende **Employee**), ela acabará estendendo **Object** por herança (como **Employee** estende **Object**, logo, **Trainee** estende **Object** por herança). Não importa quantas classes estejam na hierarquia, chegará uma altura que uma classe estenderá **Object** e passará, por herança, as características de **Object** para todas as suas subclasses.

Porque tudo no Java são objetos?

Antes de prosseguirmos, vamos abrir um parêntese para explicar com mais um exemplo o que o Java faz automaticamente quando criamos uma classe simples, pois isso é muito importante para compreendermos melhor o desenho de classes.

Seja **SomeClass** uma classe qualquer que criamos, sem imports ou construtores definidos, apenas para representar um objeto, como mostra a **Listagem 4**.

Esse código nos permite criar um objeto do tipo **SomeClass** com todas as funcionalidades padrões de um objeto Java. Isso acontece porque o Java, implicitamente, importa o pacote **java.lang**, estende a classe **Object**, cria um construtor padrão sem argumentos e chama o super construtor da classe **Object**, tudo isso automaticamente; o que significa que o código da **Listagem 4** e o código da **Listagem 5**, demonstrado a seguir, são equivalentes.

Listagem 4. Classe sem construtores ou imports definidos.

```
public class SomeClass {  
}
```

Listagem 5. Classe equivalente à classe **SomeClass** da **Listagem 4**.

```
import java.lang.*;  
  
public class SomeClass extends Object {  
  
    public SomeClass() {  
        super();  
    }  
}
```

Tudo o que está nessa listagem e não está na **Listagem 4** é inserido automaticamente pelo Java no momento da compilação. Por isso elas são iguais em termos funcionais. Entendeu agora porque tudo no Java são objetos?

O gerente

Continuando com o nosso exemplo, vamos definir agora a representação dos gerentes da nossa empresa. Um gerente, assim como todos os trabalhadores, é um empregado. Logo, deve ter todas as características e funcionalidades que todos os empregados têm, mais a característica específica que lhe diz respeito, o bônus. Como mencionado, todos os gerentes da empresa têm um bônus fixo, que é acertado no momento em que um funcionário passa a ter este cargo. Dito isso, usando novamente o conceito de herança em Java, vamos criar a classe **Gerente**, como demonstra o código da **Listagem 6**.

Como podemos notar, um gerente tem um identificador único, um nome, um salário e um bônus (declarados em seu construtor). Todos os campos, exceto o bônus, são herdados da classe **Employee**. E se repararmos bem, vamos ver também que o gerente não tem apenas um método a mais para retornar o campo extra que foi introduzido, mas também reescreveu o método de retorno de salário (**getSalary()**), de forma que o seu salário conte com o bônus. A reescrita de um método herdado, no inglês, é conhecida

como Overriding, daí a anotação `@Override` sobre o método de retorno de salário. Outro ponto importante é que podemos chamar o método da superclasse da mesma forma que chamamos o seu construtor, ou seja, utilizando a palavra **super** (BOX 1).

Listagem 6. Classe que representa um gerente da empresa – Manager.

```
public class Manager extends Employee {
    private double _bonus;

    public Manager(String id, String name, double salary, double bonus) {
        super(id, name, salary);
        _bonus = bonus;
    }

    @Override
    public double getSalary() {
        System.out.println("Salário do gerente.");
        return super.getSalary() + _bonus;
    }

    public double getBonus() {
        return _bonus;
    }
}
```

BOX 1. Super

A palavra **super** é uma referência à superclasse da classe que o utiliza. Vimos nestes exemplos que **super()** é uma chamada ao super construtor da classe, mas apenas a palavra **super** é uma referência à superclasse que nos dá acesso aos seus atributos e métodos, quer sejam da classe herdada, quer sejam de classes acima da hierarquia da superclasse.

Overloading vs Overriding

E porque a anotação `@Override` é importante? Porque esta anotação diz ao compilador que estamos explicitamente reescrevendo um método em nossa classe. Se alguém, por alguma razão qualquer, inadvertidamente fizer uma alteração na assinatura do método `getSalary()` da classe **Manager**, ou da classe **Employee**, e tivermos esta anotação, teremos um erro de compilação nos avisando que o método deve reescrever o método já existente, ou que o tipo de retorno do método que estamos reescrevendo não é o mesmo. Isso porque estamos dizendo: “– Eu quero reescrever este método.”, e quando alteramos a assinatura do método, não o estamos fazendo. Por isso, inserir a anotação no método torna o desenho de nossa classe mais robusta. A **Listagem 7** mostra um exemplo de como o erro é gerado em tempo de compilação se alterarmos a assinatura do método `getSalary()` da classe **Manager**.

O erro do método `getSalary()` diz que temos que reescrever o método da classe de cima da hierarquia (no caso, da classe **Employee**), e isso não está sendo feito, pois alteramos a assinatura do método introduzindo um novo argumento que não existe no método “original”. Daí termos o erro que nos alerta para corrigirmos o problema. Caso não tivéssemos a anotação `@Override`, não teríamos este erro, o que podemos verificar pelo método `getSalary()` da linha 6 (experimente copiar o código e inserir a anotação para ver o erro aparecer também para este método). Neste caso,

o compilador não assume que estamos fazendo Overriding do método, mas sim Overloading, ou seja, estamos adicionando um novo método, com o mesmo nome, mas com outra assinatura (ele não existe na classe de cima), logo, não temos erro.

Deste modo, fazendo overriding de métodos, podemos modificar o comportamento herdado da classe pai (como vemos no nosso exemplo com os métodos de retorno de salário). Assim, uma subclasse pode criar um método com uma funcionalidade diferente da herdada, desde que mantenha: o mesmo nome do método, o mesmo tipo de retorno (a partir da versão 5 do Java o tipo de retorno pode ser uma subclasse do tipo de retorno do método herdado) e a mesma lista de argumentos. O método reescrito também não pode ter uma visibilidade menor que a visibilidade do método herdado, ou seja, se o método herdado tiver uma visibilidade pública, e se tentarmos fazer override do método com uma visibilidade privada, teremos um erro de compilação dizendo que não podemos reduzir a visibilidade do método herdado; logo, neste caso, só poderemos dar a visibilidade pública ao método reescrito. Por outro lado, se o método herdado tiver uma visibilidade privada, o método reescrito pode ter uma visibilidade privada, protegida, padrão (sem modificadores) ou pública. Para verificar este conceito na prática, experimente alterar a visibilidade do método `getSalary()` das classes **Engineer** e **Manager**.

Outro fator muito importante que também diferencia Overloading de Overriding é que Overloading é decidido em tempo de compilação, enquanto Overriding é decidido em tempo de runtime. Isso quer dizer que se chamarmos o método `getSalary()` da linha 6, sabemos, em tempo de compilação, qual método será chamado para o objeto. Por outro lado, se fizermos a chamada do método `getSalary()` da linha 2 (assumindo que o método esteja corrigido), só saberemos qual método será chamado quando o programa estiver em execução. Isso porque o método chamado vai depender de como o objeto foi instanciado, e assim, em tempo de execução, a máquina virtual Java vai procurar o método desde a classe da instância, subindo na sua hierarquia até encontrar o método que deve executar. Vamos ver um exemplo explicativo sobre este assunto mais à frente, quando tratarmos de polimorfismo.

Listagem 7. Exemplo de um erro de overriding em um método da classe Manager.

```
1 @Override
The method getSalary(int) of type Manager must override or implement a
supertype method
2 public double getSalary(int x) {
3     return super.getSalary() + _bonus;
4 }
5
6 public double getSalary(double bonus) {
7     return super.getSalary() + bonus;
8 }
```

O Engenheiro

Antes, contudo, vamos primeiro implementar as classes restantes. O próximo empregado a implementar é o engenheiro. Como mencionado, o engenheiro é um empregado que possui

uma categoria e esta, por sua vez, define qual a porcentagem de bônus que o mesmo irá receber sobre o seu salário. A **Listagem 8** mostra uma possível representação desta classe.

Como podemos verificar, esta classe também faz override do método de retorno de salário, já que o seu salário possui regras específicas, diferentes dos demais empregados da empresa. Representamos a categoria de um engenheiro através de um **Enum** (para mais informações sobre o tipo **Enum**, veja o artigo “Tipos Enum no Java”, publicado no site da DevMedia e escrito pelo autor Thiago Palmeira), que mostramos na **Listagem 9**.

Listagem 8. Classe representa um engenheiro da empresa - **Engineer**.

```
1 public class Engineer extends Employee {  
2     private EngineerCategory _engineerCategory;  
3  
4     public Engineer(String id, String name, double salary, EngineerCategory  
        engineerCategory) {  
5         super(id, name, salary);  
6         _engineerCategory = engineerCategory;  
7     }  
8  
9     @Override  
10    public double getSalary() {  
11        System.out.println("Salário do engenheiro.");  
12        return super.getSalary() * _engineerCategory.salaryBonusPercentage();  
13    }  
14  
15    public EngineerCategory getEngineerCategory() {  
16        return _engineerCategory;  
17    }
```

Listagem 9. **EngineerCategory:** **Enum** que representa cada categoria que um engenheiro pode ter na empresa.

```
1 public enum EngineerCategory {  
2     FIRST_CLASS(1.6), SECOND_CLASS(1.3), THIRD_CLASS(1.2), FOURTH_CLASS(1.1);  
3  
4     private final double _salaryBonusPercentage;  
5  
6     EngineerCategory(double salaryBonusPercentage) {  
7         _salaryBonusPercentage = salaryBonusPercentage;  
8     }  
9  
10    public double salaryBonusPercentage() {  
11        return _salaryBonusPercentage;  
12    }  
13}
```

O código nos mostra que um engenheiro da nossa empresa pode ter uma de quatro categorias diferentes: de engenheiros de primeira classe até engenheiros de classe quatro. Um engenheiro de primeira classe tem um bônus de 60% sobre o seu salário, o de segunda classe recebe um bônus de 30%, o de terceira classe, 20% e, por fim, o de quarta classe recebe um bônus de 10%. Resta-nos agora definir o cargo de diretor da empresa.

0 Diretor

Por regra da empresa, vimos que um diretor deve ser um engenheiro. A **Listagem 10** nos mostra exatamente isso. Como a classe **Director** estende **Engineer**, o método **super()**, no construtor

mostrado na **Listagem 10**, está chamando o construtor da classe **Engineer**, que por sua vez chama o **super()** da classe **Employee**, que então chama o **super()** da classe **Object**. Outro fator importante é que nesta classe não temos a reescrita do método de retorno de salário, o que quer dizer que **Director** recebe o mesmo salário que o engenheiro. A diferença do diretor para o engenheiro é que existe um subsídio a mais para o cargo de diretor.

Listagem 10. Classe que representa o diretor da empresa - **Director**.

```
1 public class Director extends Engineer {  
2     private double _carAllowance;  
3  
4     public Director(String id, String name, double salary, EngineerCategory  
        engineerCategory,  
        double carAllowance) {  
5         super(id, name, salary, engineerCategory);  
6         _carAllowance = carAllowance;  
7     }  
8  
9     public double getCarAllowance() {  
10        return _carAllowance;  
11    }  
12}
```

Para melhor visualizar a hierarquia de classes desenhada, a **Figura 2** nos mostra o que temos neste momento.

Como podemos notar, um diretor é um engenheiro e todos são empregados. Assim, todos os membros da classe empregado são herdados por todas as suas subclasses. Também podemos identificar que o diretor herda todos os membros da classe engenheiro e tanto o engenheiro (e o diretor, por herança) quanto o gerente possuem cálculos de salários específicos. Além disso, constatamos que apenas o gerente tem um bônus, o engenheiro e o diretor têm uma categoria que os classifica como um tipo de engenheiro (que define o bônus para seus respectivos salários) e o diretor possui um subsídio para o carro.

Polimorfismo

Vamos falar agora sobre o assunto que nos referimos como sendo o segundo aspecto mais importante sobre herança. Olhando para a **Figura 2**, podemos ver que um empregado pode assumir várias formas, ou seja, um empregado pode assumir a forma de estagiário, gerente, engenheiro ou diretor. A essa habilidade damos o nome de polimorfismo!

No Java, dois pontos muito importantes que valem destaque sobre polimorfismo são:

- Um objeto pode ter apenas uma forma;
- Uma variável de referência pode se referir a objetos de diferentes formas.

Explicando de outra forma, podemos dizer que um objeto não é polimórfico, mas sim a variável que o referencia.

Deste modo, a linguagem de programação Java nos permite referenciar um objeto com uma variável que seja do tipo de uma classe parente sua, ou seja, a variável que referencia um objeto pode ser

Projetando classes em Java

do mesmo tipo do objeto ou um tipo superior na sua hierarquia de classes, que será sempre de um tipo menos específico que o do próprio objeto. Vejamos o código da **Listagem 11**.

Listagem 11. Exemplo de uma variável do tipo empregado referenciando um gerente.

```
Employee e = new Manager("M0001", "Eduardo", 8000, 3600);
```

Listagem 12. Exemplo de uma variável do tipo empregado referenciando um gerente.

```
1 NumberFormat formatter = NumberFormat.getCurrencyInstance  
  (Locale.forLanguageTag("pt-BR"));  
2  
3 Employee e = new Manager("M0001", "Eduardo", 8000, 3600);  
4 System.out.println(formatter.format(e.getBonus()));
```

Invocação de métodos virtuais

Vamos voltar neste momento a uma afirmação anterior, onde dissemos que: “Overloading é decidido em tempo de compilação, enquanto Overriding é decidido em tempo de runtime”. Se uma classe faz overloading de um método, quer dizer que este método tem outra assinatura e assim não tem como nos confundirmos, porque só existe um método a ser chamado. Agora, no caso do overriding, se chamarmos o método `getSalary()` do gerente, que é referenciado por uma variável do tipo empregado, que método será chamado? O método do empregado (da variável), ou o do gerente (do tipo do objeto)? A resposta é que será sempre o método que vem do tipo do objeto e não da variável! Por isso que é decidido em runtime. A **Listagem 13** mostra o código que serve de exemplo.

Listagem 13. Exemplo de uma variável do tipo empregado referenciando um gerente.

```
1 public class TestClass {  
2  
3     public static void main(String[] args) {  
4         NumberFormat formatter = NumberFormat.getCurrencyInstance  
          (Locale.forLanguageTag("pt-BR"));  
5  
6         Employee e = new Manager("M0001", "Eduardo", 8000, 3600);  
7         System.out.println("Salary: " + formatter.format(e.getSalary()));  
8     }  
9 }
```

Output:
Salário do gerente.
Salário do empregado.
Salary: R\$ 11.600,00

Como podemos verificar no output gerado, o método chamado para o cálculo do salário foi o do gerente, ou seja, o da instância do objeto. Este, por sua vez, chama o método `super()`, que corresponde ao método do cálculo de salário do empregado. Experimente pressionar a tecla F3 no Eclipse, em cima da chamada do método `getSalary()` da linha 7, para ver qual método é referenciado pelo IDE. Como ele está sendo chamado com a variável empregado “e”, o IDE irá abrir o método da classe empregado, mas em tempo de runtime a JVM irá resolver o tipo de objeto e chamar o método da classe gerente. Essa resolução do tipo do objeto e chamada do método correto em runtime é conhecida como invocação de métodos virtuais.

Coleções heterogêneas

Chegou a hora de colocarmos toda essa maravilha para funcionar! Tendo criado a estrutura de nosso programa usando um bom desenho de classes, com a hierarquia de empregados da empresa corretamente representada, como podemos aproveitar nosso desenho da melhor maneira?

Devido ao polimorfismo, a linguagem Java nos permite criar coleções de objetos de todos os tipos disponíveis (lembrando que todos os objetos Java descendem da classe `Object`). Tais coleções são chamadas de coleções heterogêneas.

Figura 2. Hierarquia de classes de empregados

Este código funciona porque um gerente é um empregado. Usando a variável do tipo empregado, podemos apenas acessar as características do objeto que fazem parte da classe empregado. As informações específicas do gerente não são acessíveis desta maneira, pois para o compilador, o empregado representado pela variável “e” é apenas um empregado (tipo da variável), e não um gerente (tipo da instância do objeto). É por esta razão que o código da **Listagem 12** não irá funcionar.

Este código irá provocar um erro que informa que o método `getBonus()` não está definido para o tipo empregado, mesmo sabendo que o objeto (que é um gerente) possui este método.

Parece estranho criarmos um objeto gerente com detalhes específicos a este e deliberadamente referenciá-lo com uma variável do tipo empregado, que não nos permite acessar a todas as suas características. No entanto, como iremos verificar, existem razões para que queiramos fazer isso.

Suponhamos que desejamos criar uma lista com o identificador, o nome e o salário de todos os trabalhadores da empresa. Graças à hierarquia de classes que definimos em nosso desenho e também às coleções heterogêneas, podemos implementar uma lista heterogênea de empregados. Como os dados solicitados estão em variáveis comuns a empregado, podemos criar esta lista muito facilmente, como demonstra o código da **Listagem 14**.

Listagem 14. Exemplo de criação de lista heterogênea.

```

1 public class TestClass {
2
3     public static void main(String[] args) {
4         NumberFormat formatter = NumberFormat.getCurrencyInstance(
5             Locale.forLanguageTag("pt-BR"));
6
7         // Demonstração de polimorfismo.
8         ArrayList<Employee> employeeList = new ArrayList<Employee>();
9         employeeList.add(new Manager("M0054", "Carlos", 3000, 500));
10        employeeList.add(new Engineer("E0080", "Roberto", 2000,
11            EngineerCategory.FIRST_CLASS));
12        employeeList.add(new Trainee("T0182", "Rodrigo", 1000));
13
14        // Como sabemos, todos os empregados têm um identificador,
15        // nome e salário.
16        for (Employee employee : employeeList) {
17            System.out.println("Colaborador:" + employee.getId() + ", Nome:" +
18                employee.getName() + ", Salário:" + formatter.format(employee.getSalary()));
19        }
20    }
21
22 }
```

Output:

```

Colaborador: M0054, Nome: Carlos, Salário: R$ 3.500,00
Colaborador: E0080, Nome: Roberto, Salário: R$ 3.200,00
Colaborador: T0182, Nome: Rodrigo, Salário: R$ 1.000,00
```

Desta forma, podemos processar todos os empregados da empresa da mesma maneira, sem distinções, o que torna o processo muito mais fácil. É de notar que, apesar de os dados estarem contidos na classe **Employee**, os métodos chamados nem sempre estão nesta classe, como acontece para o cálculo do salário do gerente, do engenheiro e do diretor. Note, no entanto, que não temos que nos preocupar com isso, pois o Java trata de executar o método correto para cada objeto da lista.

Argumentos polimórficos

Outro exemplo importante na utilização do polimorfismo está na construção de métodos com argumentos polimórficos. Podemos escrever métodos que aceitem argumentos genéricos e que funcionem corretamente para qualquer objeto que seja de uma subclasse do tipo do argumento.

Por exemplo, suponhamos que queremos calcular o imposto sobre o salário de cada um dos nossos empregados, e que este

cálculo é feito sobre o vencimento total de cada trabalhador (incluindo o bônus), com uma taxa de 20% sobre esse valor. A **Listagem 15** mostra uma forma de fazermos isso em um método com argumento polimórfico.

Listagem 15. Exemplo de método com argumento polimórfico.

```

1 public class TestClass {
2
3     public static double getTaxRate(Employee employee) {
4         return employee.getSalary() * 0.2;
5     }
6
7     public static void main(String[] args) {
8         NumberFormat formatter = NumberFormat.getCurrencyInstance(
9             Locale.forLanguageTag("pt-BR"));
10
11        // Demonstração de polimorfismo.
12        ArrayList<Employee> employeeList = new ArrayList<Employee>();
13        employeeList.add(new Manager("M0054", "Carlos", 3000, 500));
14        employeeList.add(new Engineer("E0080", "Roberto", 2000,
15            EngineerCategory.FIRST_CLASS));
16        employeeList.add(new Trainee("T0182", "Rodrigo", 1000));
17
18        // Exemplo de método com argumento polimórfico.
19        for (Employee employee : employeeList) {
20            System.out.println("Imposto sobre salário do empregado "
21                + employee.getName() + ":" +
22                formatter.format(getTaxRate(employee)));
23        }
24    }
25 }
```

Output:

```

Imposto sobre salário do empregado Carlos: R$ 700,00
Imposto sobre salário do empregado Roberto: R$ 640,00
Imposto sobre salário do empregado Rodrigo: R$ 200,00
```

No código podemos ver que o método **getTaxRate()** recebe um empregado como argumento. No ciclo **for** da linha 17, vários tipos de empregados são enviados para este método (gerente, engenheiro e estagiário) e todos eles calculam o seu imposto da mesma maneira, ou seja, o seu salário multiplicado por 0,2. Sem este benefício trazido pelos argumentos genéricos, teríamos que construir vários métodos de cálculo de imposto, um para cada tipo de empregado (no caso do exemplo são apenas três, mas poderia ser muitos mais).

O operador instanceof

No caso de precisarmos saber que tipo de objeto nos é passado em runtime, podemos usar o operador **instanceof**. “Instance of”, traduzido do inglês, significa “instância de”, e nos diz precisamente que o objeto que nos é passado é uma instância de um determinado tipo.

Com isso, podemos tratar os objetos recebidos de formas distintas. Por exemplo, suponhamos que precisamos de uma listagem apenas dos engenheiros e diretores da empresa. Para isso, basta perguntarmos qual o tipo de instância de um empregado com o operador **instanceof** e tratar apenas os que forem dos tipos pretendidos.

Cast de objetos

Mas, e se nesta listagem precisarmos também saber um dado específico da classe dos objetos de um tipo específico? Para usarmos dados específicos de um objeto que seja de uma subclasse de um objeto recebido, temos que fazer um cast. O cast de um objeto restaura todas as funcionalidades que este objeto possui.

Vale ressaltar que os casts são validados em runtime! Isso quer dizer que se fizermos um cast errado na hierarquia de objetos a que este pertence, não há como o compilador nos avisar do problema, e apenas saberemos dele em runtime. Para validarmos que estamos fazendo um cast corretamente, devemos utilizar o operador `instanceof`.

Os casts para cima da hierarquia são sempre permitidos e feitos implicitamente, ou seja, não é necessário utilizar a operação `cast` manualmente. Veja como exemplo as linhas 12, 13 e 14 da [Listagem 15](#). Na linha 11, dessa mesma listagem, é criada a `employeeList`, uma lista de empregados do tipo `Employee`, mas note que adicionamos a ela um gerente, um engenheiro e um estagiário, isto é, estamos falando de uma lista polimórfica. Deste modo, quando adicionamos um empregado à lista, qualquer que seja o seu subtipo, é feito um `cast` para empregado automaticamente, sem necessidades de intervenção do programador.

Os casts para baixo da hierarquia devem ser testados para saber se são possíveis. Por exemplo, não é possível fazer um cast de um engenheiro para um estagiário, porque em tempo de compilação é feita a validação que o estagiário não pertence à hierarquia do engenheiro, como mostra a [Listagem 16](#).

Neste caso teremos um erro dizendo que não é possível fazer o cast de engenheiro para estagiário.

No entanto, se o engenheiro é referenciado por uma variável do tipo empregado, então, em tempo de compilação não teremos erro se fizermos esse mesmo cast feito anteriormente, mas teremos uma exceção do tipo `ClassCastException` em runtime, como nos mostra o código da [Listagem 17](#).

Listagem 16. Exemplo de erro de cast em tempo de compilação.

```
1 // Cast incorreto, com erro em tempo de compilação.  
2 Trainee t = (Trainee)new Engineer("E0080", "Roberto", 2000, EngineerCategory.  
FIRST_CLASS);
```

Listagem 17. Exemplo de erro de cast em tempo de runtime.

```
1 // Upcast implícito: ok.  
2 Employee e = new Engineer("E0080", "Roberto", 2000, EngineerCategory.  
FIRST_CLASS);  
3 // Downcast correto em tempo de compilação, mas com erro em runtime!  
4 Trainee t = (Trainee)e;
```

Na linha 2 temos um objeto de instância do tipo engenheiro sendo referenciado por uma variável do tipo empregado, onde acontece um cast automático para cima da hierarquia. Na linha 4 da mesma listagem, fazemos um cast para baixo, de uma variável do tipo empregado, para outra do tipo estagiário, que é válido em tempo de compilação, mas como a instância é do tipo engenheiro, este cast irá gerar uma exceção em tempo de execução.

Vamos então ver um exemplo de tratamento diferenciado de objetos pelos seus tipos utilizando o cast de maneira segura com o operador `instanceof`. Neste exemplo, queremos implementar uma lista com todos os empregados da empresa, e em seguida, imprimir os seus nomes e respectivos cargos. Além disso, queremos saber também qual a classe de todos os engenheiros. O código da [Listagem 18](#) nos permite atingir este objetivo.

Listagem 18. Usando cast de maneira segura.

```
1 public class TestClass {  
2  
3 // Demonstração do polimorfismo.  
4 ArrayList<Employee> employeeList = new ArrayList<Employee>();  
5 employeeList.add(new Manager("M0054", "Carlos", 3000, 500));  
6 employeeList.add(new Engineer("E0080", "Roberto", 2000, EngineerCategory.  
FIRST_CLASS));  
7 employeeList.add(new Director("D0021", "José", 2000, EngineerCategory.  
.FIRST_CLASS, 50000));  
8 employeeList.add(new Trainee("T0182", "Rodrigo", 1000));  
9  
10 // Cast seguro com a utilização do operador instanceof.  
11 for (Employee employee : employeeList) {  
12     if (employee instanceof Engineer) {  
13         System.out.println("O empregado " + employee.getName() +  
" é engenheiro de categoria "  
+ ((Engineer) employee).getEngineerCategory());  
14     }  
15 }  
16 else if (employee instanceof Director) {  
17     System.out.println("O empregado " + employee.getName() +  
" possui o cargo de diretor com categoria de engenheiro " +  
((Engineer) employee).getEngineerCategory());  
18 }  
19 else if (employee instanceof Manager) {  
20     System.out.println("O empregado " + employee.getName() + " é gerente.");  
21 }  
22 else if (employee instanceof Trainee) {  
23     System.out.println("O empregado " + employee.getName() + " é estagiário");  
24 }  
25 }
```

Output:

```
O empregado Carlos é gerente.  
O empregado Roberto é engenheiro de categoria FIRST_CLASS.  
O empregado José é engenheiro de categoria FIRST_CLASS.  
O empregado Rodrigo é estagiário
```

Ok, conseguimos atingir o nosso objetivo e criamos a nossa lista de saída. Mas..., alguém notou algo estranho no output? Se não, veja novamente.

Na linha 7 estamos adicionando um diretor à nossa lista polimórfica, mas José foi tratado como um engenheiro, como pode ser visto no output da mesma listagem; o que está correto, porque um diretor é um engenheiro. Desta maneira, José foi tratado no `if` da linha 11 do código. Isso mostra que, para a lógica do código funcionar corretamente neste caso, a ordem dos `ifs` onde perguntamos qual é o tipo de instância do objeto é importante. Se invertermos a ordem de validação do tipo de instância do engenheiro (linha 11) com a do diretor (linha 15), teremos o output mostrado na [Listagem 19](#).

Listagem 19. Output gerado pela correção do código da **Listagem 18.**

O empregado Carlos é gerente.
O empregado Roberto é engenheiro de categoria FIRST_CLASS.
O empregado José possui o cargo de diretor com categoria de engenheiro FIRST_CLASS.
O empregado Rodrigo é estagiário

Para conseguirmos desenvolver um bom desenho em nossos programas, temos que procurar entender como tudo funciona nas peças de construção da linguagem de programação. No caso da linguagem Java, estamos falando das classes, os templates criadores dos nossos objetos. Entendendo como criar classes bem estruturadas, como utilizar a hierarquia de classes, como utilizar o polimorfismo e a invocação de métodos virtuais, amplia os nossos horizontes de criatividade para desenhar programas com menor esforço, com mais reutilização de código, menos manutenção, menor susceptibilidade a cometer erros e maior facilidade para corrigi-los, caso ocorram.

Assim, esperamos com este artigo que o leitor possa ter uma visão mais clara de como estruturar seus programas de modo a aproveitar ao máximo os benefícios oferecidos pela linguagem Java. Até a próxima!

Autor



José Fernandes A. Júnior

jfajunior@gmail.com

Mestre em Engenharia Informática pela Faculdade de Ciências e Tecnologias da Universidade Nova de Lisboa (FCT-UNL). Trabalha com Java há 10 anos, mas vem trabalhando com diversas linguagens, em diversos ramos e áreas, desde core engines de empresas de telecomunicação, até aplicações móveis para Android e iOS. Trabalhou como formador autorizado da Sun Microsystems nos cursos de Java, Unix, Shell Programming e JCAPS. Possui as certificações de Java Swing, Enterprise Java Beans, Java Composite Application Platform Suite (JCAPS), Certificado de Aptidão Profissional (CAP) e Titanium Certified App Developer (TCAD).



Links:

Trail: Learning the Java Language

<https://docs.oracle.com/javase/tutorial/java/>

Lesson: Object-Oriented Programming Concepts.

<http://docs.oracle.com/javase/tutorial/java/concepts/>

Object-oriented language basics – Part 1: Classes and objects.

<http://www.javaworld.com/article/2075202/core-java/object-oriented-language-basics-part-1.html>

DÊ UM SALTO EM CONHECIMENTO!



Acesse o maior
portal para
desenvolvedores
da América
Latina!

20
mil
posts

430
mil
cadastrados

10
milhões de
page-views
por mês

Eclipse Plugins: Como criar plug-ins no Eclipse

Veja neste artigo como estender ou criar funcionalidades para a principal IDE de Java

Convenções de nomenclatura são diretrizes para a confecção de nomes (identificadores) de classes, interfaces, atributos, métodos, entre outros, para uma determinada plataforma. Um exemplo de convenção de nomenclatura bem conhecido para a linguagem Java é: “o identificador das classes deve iniciar com letra maiúscula”. Parece uma regra ingênua ou até mesmo óbvia, mas um aspecto importante para a garantia da qualidade de um software é seguir as boas práticas estabelecidas e preconizadas pela comunidade científica e pelos profissionais, durante o desenvolvimento do mesmo.

Alguns ambientes de desenvolvimento integrados (em inglês, *Integrated Development Environment – IDE*), como o Eclipse, informam o usuário sobre a violação de algumas convenções de nomenclatura no momento da criação de um elemento, como uma classe, uma interface, um projeto, entre outros. Por exemplo, a Figura 1 apresenta um aviso enviado pelo Eclipse, desencorajando o usuário a criar uma classe denominada **minhaClasse**, cujo identificador não inicia com letra maiúscula.

Essa funcionalidade do Eclipse é importante, porém ela apresenta a seguinte limitação: “se considerarmos um projeto de software já existente (legado), não há como descobrir quais são os problemas presentes neste projeto, com respeito às convenções de nomenclatura”. Neste sentido, este artigo tem como finalidade apresentar os passos para construção de um plug-in para o IDE Eclipse cujo intuito é identificar violações de convenções de nomenclatura em softwares implementados em Java.

Inicialmente, iremos abordar apenas uma convenção de nomenclatura simples e que envolve os identificadores das classes. Contudo, com as informações presentes neste artigo, o leitor será capaz de estender o plug-in desenvolvido, para que o mesmo possa identificar outros tipos de violações, envolvendo métodos, atributos, interfaces, entre outros.

Fique por dentro

Este artigo é útil para todo aquele que pretende expandir as funcionalidades do IDE Eclipse, com o intuito de adequá-lo às suas necessidades. Inicialmente, será apresentada a motivação para o desenvolvimento de plug-ins para um IDE. Posteriormente, um plug-in “Hello, World” será desenvolvido e serão descritas as maneiras disponíveis para distribuição desse plug-in aos usuários. Por fim, será apresentada a construção de um plug-in capaz de verificar se o identificador das classes de um projeto atende ou não às convenções de nomenclatura para código Java, disponibilizadas pela Oracle.

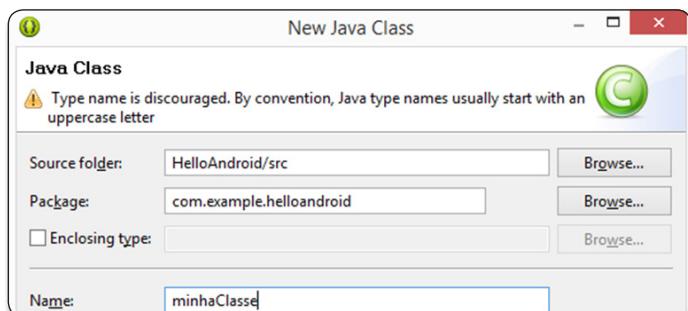


Figura 1. Aviso sobre a violação de uma convenção de nomenclatura, enviado pelo Eclipse

Nota

O plug-in para a ferramenta CASE Astah® tem como intuito verificar se a nomenclatura das classes de um diagrama de classes da UML (Unified Modeling Language) estava de acordo com algumas convenções de nomenclatura preconizadas pela Engenharia de Software; neste caso, o guia de convenção de nomenclatura para a linguagem Java, disponibilizado pela Oracle, foi adotado.

Nota

É importante salientar que já existem bons plug-ins para Eclipse cujo intuito é verificar se há violação de convenções de nomenclatura em um código fonte Java, como o Eclipse-CS. Sendo assim, o intuito deste artigo não é ser inovador, mas sim, apresentar os principais conceitos para construção de plug-ins para o IDE Eclipse.

Neste trabalho, o IDE Eclipse foi escolhido pelos seguintes motivos:

- (i) é open-source e gratuito;
- (ii) trata-se de um IDE robusto e bem conhecido no mercado; e
- (iii) é flexível a ponto de permitir a construção de plug-ins que possam complementar sua funcionalidade.

Cabe ressaltar que o enfoque principal deste artigo não está sobre o uso do Eclipse para construção de projetos Java, mas sim, para a extensão deste IDE, por meio da construção de plug-ins. A razão para isso se baseia em dois aspectos:

1. O uso do Eclipse para construção de projetos de software é bastante difundido no meio acadêmico e industrial. Sendo assim, bons tutoriais e conteúdos sobre o uso dessa ferramenta para esta finalidade podem ser facilmente obtidos e entendidos, ao contrário do que ocorre com o assunto relacionado à confecção de plug-ins. A quantidade de material é bem escassa e esse assunto é pouco discutido em salas de aulas;
2. Há um grande potencial científico relacionado com a construção de plug-ins para ferramentas desse tipo. Muitas pesquisas têm sido conduzidas em torno da avaliação da qualidade de código fonte e a construção de plug-ins para IDEs pode contribuir para que as teorias propostas nestas pesquisas sejam devidamente automatizadas. Isso pode ampliar a probabilidade de estas pesquisas serem realmente aplicadas em cenários reais de produção de software.

Neste artigo abordaremos inicialmente a preparação do ambiente de desenvolvimento, a construção de um plug-in no estilo “Hello, World” e sua instalação no Eclipse e, por fim, a construção do plug-in de exemplo. A **Tabela 1** apresenta a lista de softwares utilizados neste artigo, bem como suas respectivas versões. Os endereços para download podem ser verificados na seção **Links**.

Software	Versão
Java SDK	Oracle JDK 8
Eclipse for RCP and RAP Developers	4.4 (Luna)

Tabela 1. Softwares necessários para acompanhamento deste artigo

É importante salientar que todos os testes foram realizados sobre o Sistema Operacional Windows, versão 8.1. O processo de instalação do Java e do Eclipse não será contemplado neste artigo, por se tratar de um processo bem simples e para o qual existem bons tutoriais disponíveis na internet.

Desenvolvendo um plug-in “Hello, World”

Como é de praxe na escrita da maioria dos artigos e tutoriais técnicos, vamos começar por um projeto “Hello, World”. Este exemplo servirá para entendermos: (i) como criar um projeto para desenvolvimento de plug-ins no Eclipse; e (ii) quais são as duas principais formas de disponibilização de plug-ins aos usuários do Eclipse.

O plug-in “Hello, World”

Para criar seu primeiro plug-in, abra o Eclipse e acesse o menu *File > Other > Plug-in Project*. Na primeira tela que aparecer, a informação mais importante é o nome do projeto do seu plug-in; vamos denominá-lo **com.plugin.helloworld**. Após pressionar o botão *Next*, a segunda tela solicita o nome do seu plug-in, seu ID e o número de versão do mesmo. Vamos configurá-los da seguinte forma:

- **ID:** com.plugin.helloworld;
- **Versão:** 1.0.0.qualifier;
- **Nome:** Hello World.

Após pressionar *Next* mais uma vez, a tela que aparece irá questioná-lo sobre a utilização ou não de um template para construção do plug-in. Neste caso, vamos escolher a opção *Hello, World* e clicar em *Finish*.

Pronto! Um projeto de plug-in funcional já encontra-se disponível em seu workbench. Para testá-lo, basta clicar com o botão direito do mouse sobre o projeto criado e escolher a opção *Run As > Eclipse Application*. O resultado esperado é um novo botão, denominado *Sample Menu*, na barra de menus do Eclipse. Este menu possui um item denominado *Sample Action* (**Figura 2**), que ao ser acionado, exibirá a mensagem da **Figura 3**.

Nota

É muito comum, tanto no desenvolvimento de plug-ins quanto no desenvolvimento de aplicativos móveis, utilizarmos a descrição de uma URL invertida como ID do nosso plug-in ou aplicativo. Isso ocorre, pois o ID tanto das aplicações móveis quanto dos plug-ins deve ser único, dentro de um amplo conjunto de plug-ins e aplicações disponíveis publicamente. Utilizando esta prática, garante-se que seu ID será único universalmente.

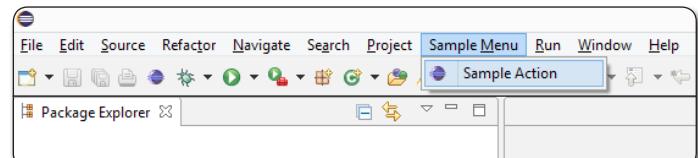


Figura 2. Menu extra, adicionado na interface do Eclipse pelo plug-in Hello World

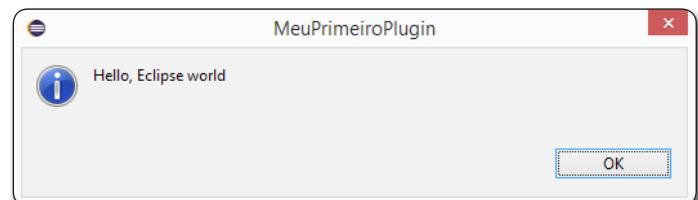


Figura 3. Mensagem exibida ao acionar o botão Sample Action, do plug-in Hello World

Disponibilizando plug-ins aos usuários

As duas principais formas de se disponibilizar um plug-in aos usuários do Eclipse são: (i) oferendo a eles um arquivo **.jar**, que poderá ser copiado para a pasta **dropins** do Eclipse e, após a reinicialização do ambiente, o plug-in já estará instalado; ou (ii) permitindo a instalação automática do plug-in via opção *Install New Software* do Eclipse.

Eclipse Plugins: Como criar plug-ins no Eclipse

A primeira forma é muito simples de se implementar, porém oferece duas desvantagens: (i) possui o inconveniente de “obrigar” o usuário a ter que baixar um arquivo **.jar** e copiá-lo para a pasta adequada do Eclipse; e (ii) não oferece uma maneira automática de se realizar a atualização do plug-in, isto é, a cada nova versão, o usuário deverá repetir o procedimento descrito no passo (i). Além disso, a verificação de novas versões do plug-in será de responsabilidade apenas do usuário.

Mesmo assim, esta estratégia pode ser útil em alguns casos, portanto, vamos descrevê-la. O procedimento é bem simples, basta clicar com o botão direito do

mouse sobre o nome do projeto do plug-in e escolher a opção *Export > Plug-in Development > Deployment plug-ins and fragments*. Depois, basta informar o diretório para exportação do plug-in e clicar em *Finish*. Para testar, copie o arquivo **.jar** gerado para a pasta **dropins** do Eclipse e reinicie o IDE. O resultado deve ser o mesmo obtido com o processo de execução do plug-in, descrito na seção anterior. Caso não queira mais trabalhar com este plug-in, você pode desinstalá-lo por meio da opção *Help > Installation Details*. Feito isso, selecione o plug-in desejado e clique em *Uninstall*.

A segunda forma de disponibilização de um plug-in minimiza as desvantagens da primeira forma, porém é mais trabalhosa

de se colocar em prática. Para implementá-la, inicialmente, é necessário criar um projeto de features. Um projeto de features pode ser entendido como um container de plug-ins, isto é, um projeto que contém uma lista de plug-ins a serem disponibilizados ao usuário. Este tipo de projeto é necessário caso você queria utilizar o gerenciador de atualizações do Eclipse. Assim, quando uma nova versão do seu plug-in estiver disponível, o Eclipse irá reconhecer e comunicar aos seus usuários. Para isso, acesse *File > New > Other > Plug-in Development > Feature Project* e nomeie o projeto como **com.plugin.instalador**. Na próxima tela, selecione o plug-in desenvolvido anteriormente, isto é, o plug-in **Hello World** e clique em *Finish*.

Antes de exportarmos o plug-in utilizando esta estratégia, é interessante criar uma categoria, que será exibida pelo Eclipse durante o processo de instalação do plug-in e serve para descrever melhor o tipo de plug-in que está sendo instalado. Para criar uma categoria, clique com o botão direto do mouse sobre o projeto **com.plugin.instalador** e acesse *New > Other > Plug-in development > Category Definition*, clique em *Next* e depois em *Finish*.

Na tela de definição de categorias, clique em *New category* e informe os dados, conforme apresentado na **Figura 4** (lado esquerdo).

Posteriormente, é necessário adicionar os plug-ins que pertencerão a esta categoria e que você deseja disponibilizar para instalação na máquina do usuário. Para isso, selecione a categoria previamente criada, conforme mostra a **Figura 4** (lado esquerdo), clique em *Add Plug-in* e selecione o plug-in **com.plugin.helloworld**.

Feito isso, agora é possível gerar os arquivos necessários para instalação do plug-in. Deste modo, clique com o botão direito do mouse sobre o projeto **com.plugin.instalador** e acesse *Export > Plug-in Development > Deployment features*. Em seguida, acesse a aba *Options* e na opção *Categorize repository*, selecione o arquivo com a descrição da categoria criada por você (se você não alterou nada durante a criação da categoria, o nome deve ser *category.xml*), conforme a **Figura 5**.

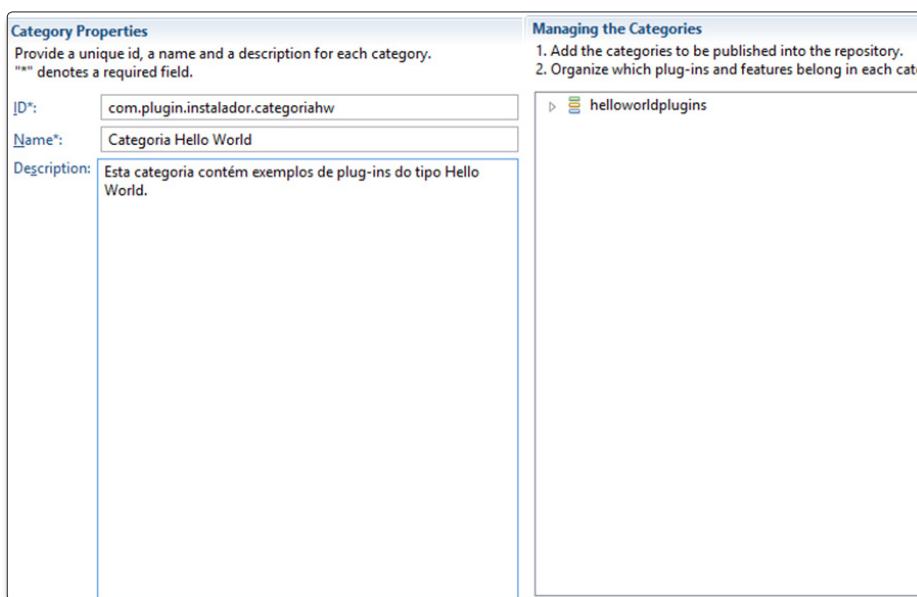


Figura 4. Tela para definição de uma categoria

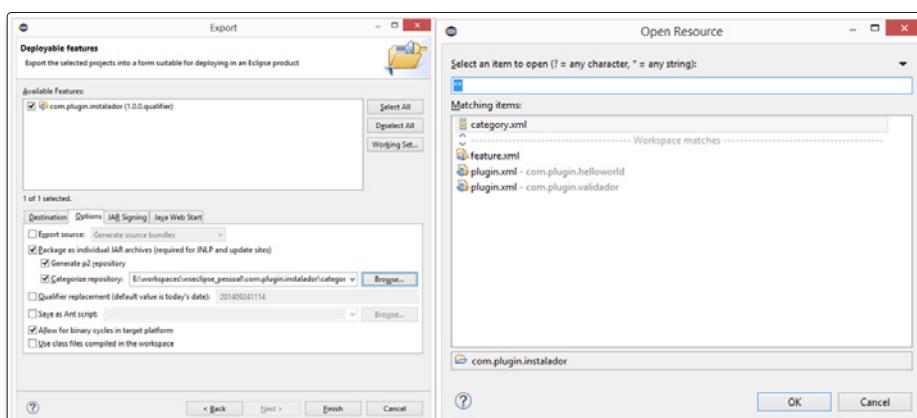


Figura 5. Exportando o plug-in para instalação via opção *Install New Software*

Neste momento, basta voltar para a aba *Destination*, escolher o diretório onde serão exportados os arquivos e clicar em *Finish*. Pronto! Os arquivos gerados podem ser disponibilizados em algum repositório na internet para que os usuários possam instalar seu plug-in automaticamente, por meio da opção *Help > Install new software* do Eclipse.

No entanto, também é possível instalar o plug-in localmente, desde que o usuário tenha acesso aos arquivos gerados anteriormente. Para testarmos a instalação automática do plug-in, acesse *Help > Install new software*. Posteriormente, clique no botão *Add*, dê um nome para o repositório, por exemplo, **Hello World Plugins**, e localize o diretório onde se encontram os arquivos gerados anteriormente pelo Eclipse.

Assim, a categoria criada para seu plug-in, bem como o próprio plug-in aparecerão como disponíveis para instalação (**Figura 6**). Agora basta clicar em *Next* e depois em *Finish*. Assim o Eclipse deverá emitir uma mensagem avisando que você está instalando um plug-in não assinado; você pode confirmar a instalação. Um plug-in não assinado é aquele cuja autenticidade não foi validada por alguma entidade certificadora. Isso significa que você pode estar baixando um plug-in de fonte desconhecida e não confiável, o que não é o caso aqui, pois nós mesmos desenvolvemos o plug-in. Para evitar esse tipo de inconveniente é possível assinar digitalmente um plug-in, porém foge ao escopo deste artigo apresentar tal recurso.

Ao realizar todo esse processo, seu plug-in será instalado no diretório *plugins* do Eclipse. Caso não deseje mais trabalhar com esse plug-in, você pode desinstalá-lo a partir da opção *Help > Installation Details*. Ao abrir esta janela, selecione o plug-in desejado e clique em *Uninstall*.

Para testarmos a instalação remota do nosso plug-in, compactamos os arquivos gerados pelo Eclipse e disponibilizei-o no Dropbox, por meio do link: <https://db.tt/lpG0b1Gk>. Dito isso, acesse *Help > Install new software*, clique no botão *Add*, dê um nome para o repositório, por exemplo, **Hello World Plugins Remoto**, e informe a URL: *jar:https://db.tt/lpG0b1Gk!/*. Pronto!

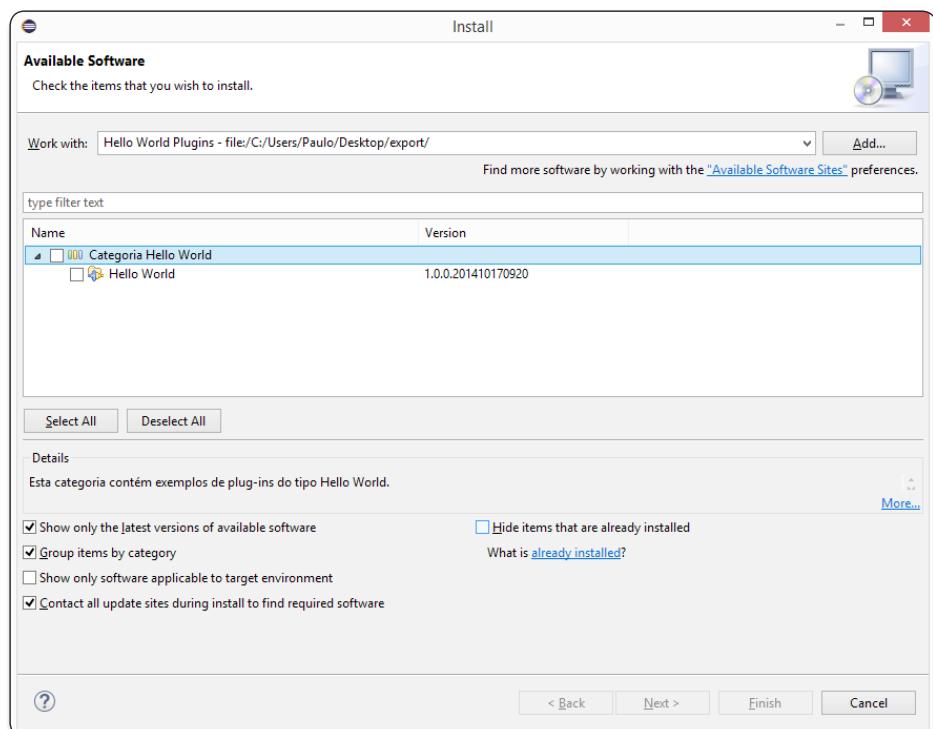


Figura 6. Instalação do plug-in via opção *Install New Software*

O restante do processo é idêntico ao que foi explicado anteriormente

Desenvolvendo o plug-in “Validador”

Agora, vamos iniciar o desenvolvimento do nosso plug-in, que receberá o nome **com.plugin.validador**. Antes de iniciarmos com o desenvolvimento, vamos discutir os principais requisitos para uma primeira versão do plug-in:

1. O plug-in deverá ser acionado a partir de um item de menu, denominado **Validar Classes**, que aparecerá toda vez que o usuário clicar com o botão direito do mouse sobre algum elemento da área **Package Explorer** do Eclipse (área onde ficam os projetos criados ou importados pelo usuário);
2. Quando acionado, o plug-in deverá verificar se o projeto é um projeto de código Java e se o mesmo encontra-se aberto. Caso não, a mensagem “Este não é um projeto Java ou o mesmo encontra-se fechado!” deve ser exibida ao usuário. Caso contrário, o plug-in deverá varrer este projeto a fim de identificar classes Java;
3. Para cada classe Java encontrada, o plug-in deverá verificar se o identificador da mesma inicia-se com letra maiúscula;

4. Ao final da execução, o plug-in poderá exibir: (i) a mensagem “Não há classes neste projeto que violem as convenções de nomenclatura!”, caso não haja classes que se iniciem com letra minúscula; ou (ii) uma lista com o nome das classes cujos identificadores inician-se com letra minúscula, o que viola uma convenção de nomenclatura para a linguagem Java.

Para criar o plug-in, acesse o menu *File > Other > Plug-in Project*. Logo após, nomeie o projeto como **com.plugin.validador** e preencha os seguintes dados para seu plug-in:

- **ID:** com.plugin.validador;
- **Versão:** 1.0.0.qualifier;
- **Nome:** Validador.

Desta vez, ao ser questionado sobre a utilização ou não de templates para construção do plug-in, não escolha qualquer tipo de template, ou seja, apenas clique em *Finish*.

Criando um item de menu

Para se criar um novo item de menu, uma nova view ou qualquer outro tipo

Eclipse Plugins: Como criar plug-ins no Eclipse

de contribuição ao ambiente Eclipse, devemos trabalhar com o conceito de **extension** (extensão). Neste artigo utilizaremos três tipos de extensão, a saber:

- **Comando**: encapsula a descrição de um comando que pode ser executado no plug-in do Eclipse;
- **Tratador**: define o comportamento a ser realizado por um comando. Isto é, a classe Java que deve ser invocada quando um comando for acionado;
- **Menu**: define como e onde um comando deve ser incluído na interface gráfica do Eclipse, por exemplo, na barra de tarefas, no menu principal, entre outros.

A vantagem desse tipo de separação é que podemos ter um mesmo comando com um tratador de evento e diversas extensões do tipo menu, oferecendo a opção de o usuário acessar o mesmo comando por mais de uma forma. Ou ainda, termos um mesmo tratador para diferentes tipos de comandos.

Para criar um comando, abra o arquivo *MANIFEST.MF* do plug-in e escolha a aba *Extensions*. Posteriormente, clique sobre o botão *Add*, procure por **org.eclipse.ui.commands** e clique em *Finish*. Feito isso, clique com o botão direito sobre o ícone com o nome **org.eclipse.ui.commands** e escolha a opção *New > Command*, como demonstra a Figura 7.

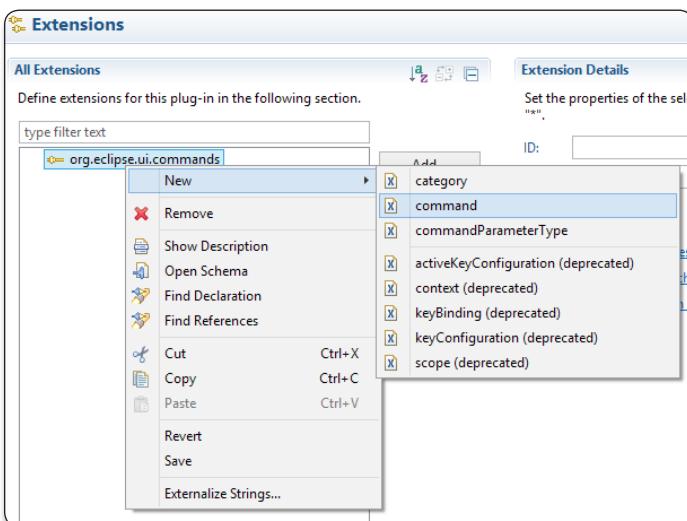


Figura 7. Criação de um comando para o plug-in Validador

Em seguida, preencha os campos, conforme apresentado na Figura 8. Como pode ser observado, apenas os atributos **id** e **name** são obrigatórios. O atributo **id** identificaunicamente um comando em seu plug-in. Já o atributo **name** representa o nome do comando, a ser exibido na interface gráfica do seu plug-in, caso o mesmo seja vinculado a um item de menu.

Feito isso, podemos criar um tratador para este comando, isto é, uma classe responsável pelo comportamento do comando **Validar Classes**. Para isso, clique sobre o botão *Add*, na tela de extensões, procure por **org.eclipse.ui.handler** e clique em *Finish*.

Figura 8. Declaração de um comando para o plug-in Validador

Você poderá notar que será criado automaticamente um handler, para o qual configuraremos o atributo **commandId** como **com.plugin.validador.validadarClasse**. Este atributo indica qual é o comando para o qual o tratador será vinculado. Neste caso, relacionamos o comando que acabamos de criar, ou seja, o **Validar Classe**.

Agora, devemos criar uma classe para tratar o comportamento do comando **Validar Classe**. Isto pode ser feito clicando sobre o atributo **class** do nosso tratador. Ao fazer isso, o Eclipse irá abrir uma janela para criação da nossa classe handler. Neste caso, preencha os dados conforme ilustrado na Figura 9.

Figura 9. Criação de um tratador para o comando Validar Classes

A única novidade deste passo, com relação ao processo de criação de uma classe Java comum, é que devemos declarar a herança entre nossa classe tratadora e a classe **AbstractHandler**, uma classe padrão para tratadores de comandos disponibilizada pelo Eclipse. Como resultado, tem-se uma classe cujo código é apresentado na **Listagem 1**.

O método **execute()** será invocado automaticamente, assim que o comando **Validar Classes** for requisitado. Para testarmos se nossa classe tratadora está realmente funcionando, coloquemos uma instrução para escrever uma mensagem no console, como indicado na **Listagem 2**.

Listagem 1. Código gerado automaticamente para o tratador do comando Validar Classes.

```
package com.plugin.validador;

import org.eclipse.core.commands.AbstractHandler;
import org.eclipse.core.commands.ExecutionEvent;
import org.eclipse.core.commands.ExecutionException;

public class ValidadorClasse extends AbstractHandler {

    @Override
    public Object execute(ExecutionEvent event) throws ExecutionException {
        // TODO Auto-generated method stub
        return null;
    }
}
```

Listagem 2. Código modificado do tratador do comando Validar Classes.

```
package com.plugin.validador;

import org.eclipse.core.commands.AbstractHandler;
import org.eclipse.core.commands.ExecutionEvent;
import org.eclipse.core.commands.ExecutionException;

public class ValidadorClasse extends AbstractHandler {

    @Override
    public Object execute(ExecutionEvent event) throws ExecutionException {
        System.out.println("Tratador invocado!");
        return null;
    }
}
```

Contudo, para que possamos testar nosso plug-in, antes, precisamos definir um item de menu, que será apresentado na interface do Eclipse e permitirá que o usuário possa acionar o comando criado anteriormente.

Portanto, clique sobre o botão *Add*, na tela de extensões, procure por **org.eclipse.ui.menus** e depois clique em *Finish*. Feito isso, clique com o botão direito sobre o ícone com o nome **org.eclipse.ui.menus** e escolha a opção *New > Menu contribution*. O único atributo obrigatório deste elemento é o *locationURI*. Este atributo é responsável por definir onde e como o menu será exibido na interface do Eclipse. Por exemplo, se você quiser adicionar um item de menu a um menu pré-existente, cujo **id** seja **fileMenu**,

depois de um item de menu já existente, cujo **id** seja **itemMenu**, o *locationURI* deve ser: **menu:fileMenu?after=itemMenu**.

Ao apontar o cursor do mouse para o rótulo *locationURI*, na interface do Eclipse, uma descrição de como este atributo deve ser especificado é apresentada (**Figura 10**).

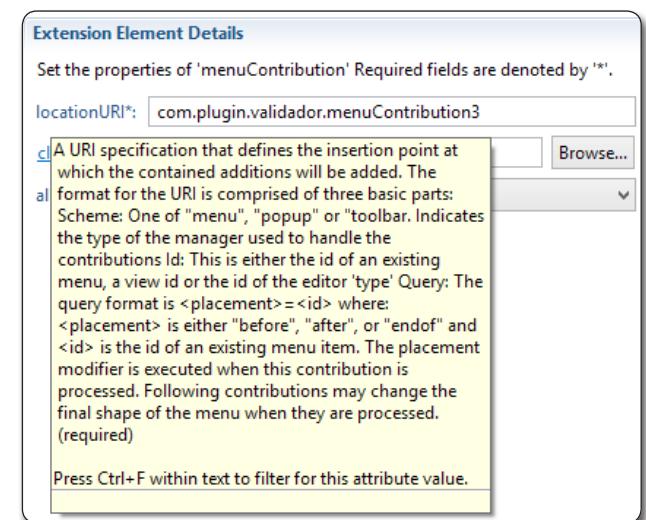


Figura 10. Descrição do atributo locationURI

Como pode ser visto, o template para especificar o valor deste atributo é: **<tipo>:<id1>?<posição>=<id2>**, onde:

- **tipo:** refere-se ao tipo de elemento de interface a ser criado, que pode ser **menu**, para adicionar um menu ou item de menu à interface do Eclipse, **popup**, para adicionar um item a um menu contextual ou **toolbar**, para adicionar um item na barra de ferramentas do Eclipse;
- **id1:** refere-se ao **id** de um menu, uma visão (view) ou de um editor pré-existente na interface do Eclipse;
- **posição:** define a posição onde o menu será apresentado, que pode ser **before** (antes), **after** (depois) ou **endof** (no final de);
- **id2:** refere-se ao **id** de um elemento de referência para posicionamento do menu a ser criado.

Nota

Há um plug-in para o Eclipse, denominado **Spy**, que permite identificar facilmente o ID dos elementos da interface deste IDE. Mais detalhes sobre a instalação e uso deste plug-in podem ser encontrados na seção **Links**.

No nosso exemplo, vamos adicionar o seguinte *locationURI* para nosso menu: **popup:org.eclipse.jdt.ui.PackageExplorer**. Isso fará com que nosso item de menu seja exibido em um menu contextual, isto é, um popup que será exibido quando o usuário clicar com o botão direito do mouse sobre qualquer elemento (classe, projeto, pacote, entre outros) da área *Package Explorer*.

Agora, basta vincular nosso menu ao comando com o qual ele estará relacionado e pronto. Neste caso, será o comando **Validar Classes**. Para fazer isso, clique com o botão direito sobre o menu

Eclipse Plugins: Como criar plug-ins no Eclipse

previamente criado e escolha a opção *New > Command*. Logo após, adicione o **id** do nosso comando, isto é, **com.plugin.validador.validadarClasse**, ao atributo **commandId**, conforme ilustrado na Figura 11.

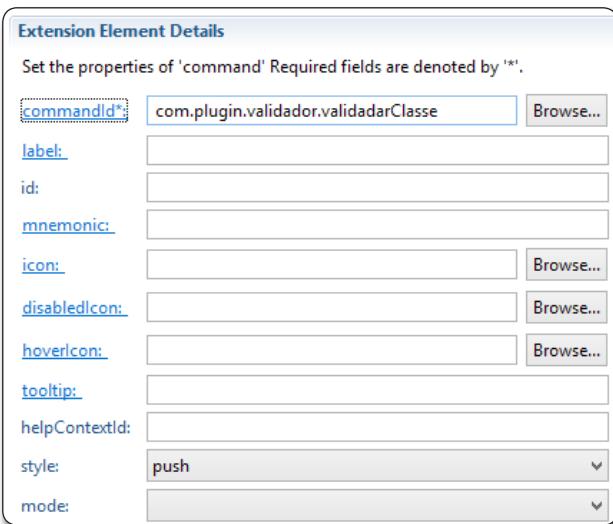


Figura 11. Vinculação do item de menu ao comando Validar Classes

Pronto! Já podemos testar a primeira versão do nosso plug-in. Para tanto, execute o projeto como uma aplicação Eclipse (*Run As > Eclipse Application*). Deste modo, ao clicar com o botão direito sobre algum elemento da área *Package Explorer*, é possível visualizar nosso item de menu *Validar Classes*. Ao clicar neste menu, a mensagem “Tratador invocado!” será apresentada no console da instância do Eclipse em que o plug-in está sendo desenvolvido.

Voltando à questão de como nosso item de menu é exibido na interface do Eclipse, podemos melhorar o visual do nosso plug-in, adicionando um ícone ao item de menu criado anteriormente. Para isso, façamos o seguinte:

- crie a pasta *icon* no projeto do seu plug-in e adicione um ícone de sua preferência a esta pasta;
- selecione o menu criado anteriormente e atribua o ícone adicionado ao atributo *icon* deste menu.

O resultado do menu com o ícone recém-adicionado é apresentado na Figura 12.

Criando o código responsável pela verificação das convenções de nomenclatura

Agora que já aprendemos a criar um item de menu e vinculá-lo a um tratador de eventos, podemos nos ater ao objetivo principal deste plug-in, que é verificar se há classes em nosso projeto que violam uma das convenções de nomenclatura definidas pela Oracle, a saber: “Todo identificador de classe deve iniciar com letra maiúscula”. A primeira parte do código responsável por verificar a nomenclatura das classes Java é apresentado na Listagem 3.

Para que o código da apareça sem erros em seu ambiente de trabalho, é necessário importar o pacote **org.eclipse.jdt.core** como uma dependência para seu plug-in. Isto porque este pacote contém as classes e interfaces necessárias para trabalhamos com os elementos do código fonte de um projeto Eclipse. Para fazer isso, abra o arquivo *MANIFEST.MF* de seu plug-in e acesse a aba *Dependencies*. Posteriormente, clique em *Add*, procure por **org.eclipse.jdt.core** e, por fim, clique em *Ok*.

Listagem 3. Verificando se um projeto Java aberto foi selecionado.

```
01 public class ValidadorClasse extends AbstractHandler {  
02     private void validarProjeto(IJavaProject meuProjeto) {  
03         // To do  
04         // To do  
05     }  
06  
07     @Override  
08     public Object execute(ExecutionEvent event) throws ExecutionException {  
09         IStructuredSelection itemSelecionado = (IStructuredSelection)  
10            HandlerUtil.getActiveMenuSelection(event);  
11         Object primeiroElemento = itemSelecionado.getFirstElement();  
12         if (primeiroElemento instanceof IJavaProject) {  
13             IJavaProject meuProjeto = (IJavaProject) primeiroElemento;  
14             validarProjeto(meuProjeto);  
15         } else {  
16             MessageDialog.openError(null, "Projeto inválido",  
17             "Este não é um projeto Java ou o  
18             mesmo encontra-se fechado!");  
19         }  
20     }  
21 }
```

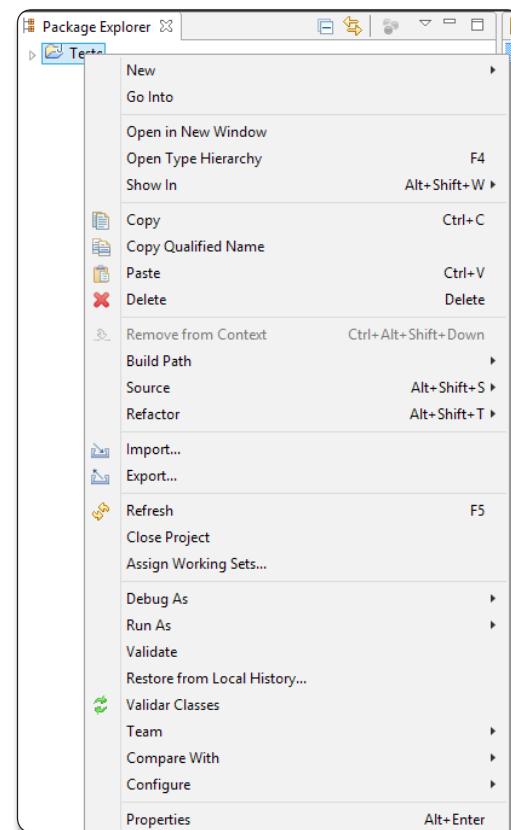


Figura 12. Item de menu Validar Classes com ícone

As linhas 9 a 11 são responsáveis por capturar o item selecionado pelo usuário. A linha 12 testa se o elemento capturado trata-se de um projeto Java. Caso sim, este objeto é convertido para uma instância do tipo **IJavaProject**, isto é, um projeto Java, e o método **validarProjeto()** será invocado. Caso contrário, a seguinte mensagem de erro será exibida: “Este não é um projeto Java ou o mesmo encontra-se fechado!”. Isto porque quando o projeto Eclipse encontra-se fechado, não será possível descobrir o tipo do mesmo, sendo assim, a condicional da linha 12 irá retornar **false** quando o projeto não se tratar de um projeto Java ou quando o mesmo encontrar-se fechado.

Agora, devemos implementar o método **validarProjeto()**, que será responsável por varrer o projeto selecionado a procura de classes Java. Assim que uma classe for encontrada, a nomenclatura da mesma será verificada, para descobrir se ela começa ou não com letra maiúscula. O código responsável por este comportamento é apresentado na **Listagem 4**.

Listagem 4. Verificando a nomenclatura das classes Java.

```

01 private StringBuffer classesComProblemas = new StringBuffer();
02
03 private boolean ehIdentificadorValido(String id) {
04     return (id.charAt(0) == id.toUpperCase().charAt(0));
05 }
06
07 private void validarProjeto(IJavaProject meuProjeto) throws JavaModelException {
08     IPackageFragment[] pacotes = meuProjeto.getPackageFragments();
09     for (IPackageFragment meuPacote : pacotes) {
10         if (meuPacote.getKind() == IPackageFragmentRoot.K_SOURCE) {
11             for (ICompilationUnit elementoCodigoFonte :
12                 meuPacote.getCompilationUnits()) {
13                 String identificadorDaClasse = elementoCodigoFonte.getElementName();
14                 if (!ehIdentificadorValido(identificadorDaClasse)) {
15                     classesComProblemas.append(identificadorDaClasse + "\n");
16                 }
17             }
18         }
19     }

```

A linha 1 declara uma variável do tipo **StringBuffer**, denominada **classesComProblemas**, que irá armazenar o identificador das classes que violam a convenção de nomenclatura. As linhas 3 a 5 descrevem o método responsável por verificar se o identificador de uma classe inicia-se com letra maiúscula. As linhas 7 a 19 descrevem o conteúdo do método **validarProjeto()**, responsável de fato por verificar a nomenclatura das classes do projeto selecionado pelo usuário.

Na linha 8, os pacotes existentes no projeto passado como parâmetro são capturados, por meio do método **getPackageFragments()**. Este método retorna não apenas pacotes de código fonte, mas qualquer tipo de pasta existente dentro do projeto. Assim, ao iterar na lista de pacotes do projeto, a linha 10 da **Listagem 4** verifica se o pacote correto trata-se de um pacote que apresenta código fonte. Para isso, compara-se o tipo do pacote correto com o valor da constante **IPackageFragmentRoot.K_SOURCE**, que representa o tipo de pacote que contém elementos de código.

Uma vez tendo sido feita esta verificação, as linhas 11 à 16 são responsáveis por capturar os elementos de código fonte existentes dentro destes pacotes, para assim, verificar se eles violam a convenção de nomenclatura. Para capturar esses elementos de código fonte, o método **getCompilationUnits()**, da interface **IPackageFragment**, é utilizado.

Por fim, o código do método **execute()** foi alterado, conforme apresentado na **Listagem 5**.

Listagem 5. Alteração no método execute() do tratador do comando Validar Classes.

```

01 @Override
02 public Object execute(ExecutionEvent event) throws ExecutionException {
03     ...
04     if (primeiroElemento instanceof IJavaProject) {
05         IJavaProject meuProjeto = (IJavaProject) primeiroElemento;
06         try {
07             validarProjeto(meuProjeto);
08             if (classesComProblemas.length() == 0) {
09                 MessageDialog.openInformation(null, "Verificação concluída",
10                     "Não há classes neste
11                     projeto que violem as convenções de nomenclatura!");
12             } else {
13                 MessageDialog.openError(null, "Verificação concluída", "Classes que violam
14                     convenções de nomenclatura:\n" + classesComProblemas);
15             }
16         } catch (JavaModelException e) {
17             MessageDialog.openError(null, "Verificação falhou",
18                 "Ocorreu algum erro ao verificar
19                     o projeto atual!");
20         }
21     ...
22 }

```

As modificações mais importantes estão contidas nas linhas 6 a 19. Nesta nova versão, o método **validarProjeto()** é invocado dentro de um bloco **try-catch**, uma vez que a execução do mesmo pode lançar uma exceção do tipo **JavaModelException**. Além disso, após a execução do método **validarProjeto()**, verifica-se se a string **classesComProblema** encontra-se vazia. Caso sim, uma mensagem informativa é enviada ao usuário, dizendo que não há classes com problema de violação de convenções de nomenclatura no projeto. Caso contrário, uma lista com o nome das classes com problemas de nomenclatura é apresentada em uma mensagem de erro. As duas situações mencionadas anteriormente podem ser observadas, respectivamente, nas **Figuras 13 e 14**.

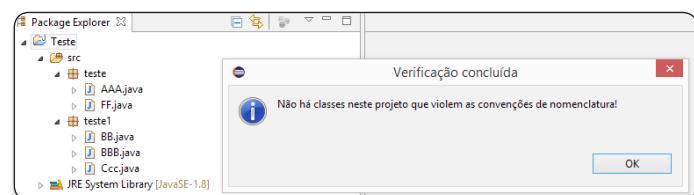


Figura 13. Projeto com classes que não violam as convenções de nomenclatura

Eclipse Plugins: Como criar plug-ins no Eclipse

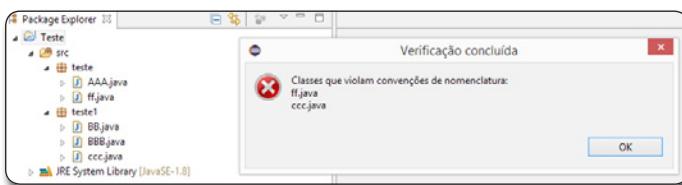


Figura 14. Projeto com duas classes que violam as convenções de nomenclatura

Como pode ser notado, o plug-in desenvolvido neste artigo apresenta diversas limitações, tais como:

- (i) só realiza um tipo de verificação de violação de convenções de nomenclatura;
- (ii) a estratégia utilizada para exibição dos erros de verificação é bastante limitada, principalmente para projetos que apresentam muitas classes; e
- (iii) para efetuar a verificação de um projeto, o usuário precisa sempre acionar o menu Validador Classes.

Autor



Paulo Afonso Parreira Júnior

paulojunior@jatai.ufg.br – <http://paulojunior.jatai.ufg.br>

Atualmente professor do curso de Bacharelado em Ciência da Computação da Universidade Federal de Goiás (Campus Jataí).

É aluno de doutorado do Programa de Pós-Graduação em Ciência da Computação (PPG-CC) da Universidade Federal de São Carlos (UFSCar), na área de Engenharia de Software. É mestre em Engenharia de Software pelo Departamento de Computação da UFSCar (2011). É integrante do Advanced Research Group on Software Engineering (AdvanSE) do Departamento de Computação da Universidade Federal de São Carlos e do Grupo de Pesquisa e Desenvolvimento de Jogos Educacionais Digitais (GrupJED) do Curso de Ciência da Computação da Universidade Federal de Goiás (Regional Jataí). Tem experiência na área de Ciência da Computação, com ênfase em Engenharia de Software, atuando principalmente nos seguintes temas: Manutenção de Software, Desenvolvimento de Software Orientado a Objetos, Desenvolvimento de Software Orientado a Aspectos e Informática na Educação.



Links:

[1] Ferramenta CASE Astah.

<http://astah.net/>

[2] Oracle Code Convention.

<http://www.oracle.com/technetwork/java/codeconvtoc-136057.html>

[3] Oracle JDK.

<http://www.oracle.com/technetwork/pt/java/javase/downloads/index.html>

[4] IDE Eclipse.

<https://www.eclipse.org/downloads/>

[5] Plug-in Eclipse-CS.

<http://eclipse-cs.sourceforge.net/#/>

RENOVE JÁ!

Sua assinatura pode estar acabando

Renovando a assinatura de sua revista favorita você ganha brindes e descontos exclusivos.

Somos tão apaixonados por tecnologia que o nome da empresa diz tudo.

Porta 80 é o melhor que a Internet pode oferecer para sua empresa.

Já completamos 8 anos e estamos a caminho dos 80, junto com nossos clientes.

Adoramos tecnologia. Somos uma equipe composta de gente que entende e gosta do que faz,
assim como você.



Estrutura

100% NACIONAL.
Servidores de primeira linha, links de alta capacidade.



Suporte diferenciado

Treinamos nossa equipe para fazer mais e melhor. Muito além do esperado.



Serviços

Oferecemos a tecnologia mais moderna, serviços diferenciados e antenados com as suas necessidades.



1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única.
Conheça!

Porta 80
WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce |
Aplicações | Streaming | Email corporativo

porta80.com.br | comercial@porta80.com.br | twitter.com/porta80

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486

GRADUAÇÃO METODISTA

*Estudando Sistemas
de Informação
na Metodista,
eu me sinto desafiado
a desenvolver a
minha capacidade
dentro da sala de aula
e no mercado
de trabalho.*

Sérgio

Aluno de Sistemas de Informação

Com os cursos da área de Exatas e Tecnologia da Metodista, você tem contato com laboratórios de última geração, por meio de parcerias com empresas como Microsoft, Oracle e IBM. Além disso, desenvolve o seu lado profissional, ético e cidadão com projetos nas áreas de sustentabilidade e inovação social.

Conheça os cursos da área de Exatas e Tecnologia da Metodista:

GRADUAÇÃO

- Análise e Desenvolvimento de Sistemas (Presencial e EAD)
- Automação Industrial (Presencial)
- Engenharia Ambiental e Sanitária (Presencial)
- Engenharia de Produção (Presencial) – NOVO
- Gestão da Tecnologia da Informação (Presencial e EAD)
- Matemática (Presencial e EAD)
- Sistemas de Informação (Presencial)

processoseletivo.metodista.br



Transforme sua vida. Transforme a realidade.

Para mais informações, acesse: processoseletivo.metodista.br
Grande São Paulo: 11 4366 5000 / Outras localidades: 0800 889 2222
facebook.com/universidade.metodista / twitter.com/metodista