



 DEV MEDIA

DESAFIO: Auditoria com Hibernate Envers
Aprenda a implementar este importante requisito

Dominando estruturas de dados

Primeiros passos com
os métodos de busca

PADRÓES DE PROJETO

Conheça as opções
Singleton e Flyweight

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.**

➤ A linguagem Java está presente em complexos sistemas como em aplicativos mobile e desktop, servindo a milhares de pessoas. Inicie sua carreira e certifique-se no instituto que valoriza seu currículo. Faça Infnet!

CERTIFICAÇÕES:

➤ Oracle Certified Professional, Java SE 6 Programmer

➤ Oracle Certified Professional, Java EE 5 Web Component Developer



Sumário

06 - Estrutura de dados: Primeiros passos com métodos de busca

[Paulo Afonso Parreira Júnior]

Conteúdo sobre Boas Práticas

16 - Vantagens e desvantagens dos padrões de projeto Singleton e Flyweight

[Ednardo Luiz Martins e Ricardo da S. Longa]

Conteúdo sobre Boas Práticas

23 - Hibernate Envers: Auditoria de dados em Java

[Carlos Alberto Silva]

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 52 • 2015 • 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 Diogo Souza (diagosouzac@gmail.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

Estrutura de dados: Primeiros passos com métodos de busca

Aprenda a recuperar informações em Java a partir da implementação dos principais métodos de busca da estrutura de dados

Este artigo trata da recuperação de dados a partir de um conjunto de informações previamente armazenado. Em geral, no meio computacional, a informação é dividida em registros e cada registro possui uma chave para ser usada na pesquisa e uma ou mais informações de interesse do usuário. Por exemplo, o registro de um aluno em uma universidade pode conter uma chave que identifica unicamente a matrícula e um conjunto de informações sobre este aluno, como nome, endereço, telefone, entre outros.

O objetivo da pesquisa é encontrar uma ou mais ocorrências de registros com chaves iguais à chave de pesquisa e para essa finalidade existem vários métodos. A escolha do mais adequado depende, principalmente: (i) da quantidade de dados envolvidos; e (ii) da possibilidade de o arquivo sofrer inserções e/ou retiradas.

Por exemplo, é diferente encontrar o registro do nome de um estado brasileiro no conjunto de todos os estados brasileiros e encontrar o registro de um aluno que fez o Exame Nacional do Ensino Médio (ENEM). No segundo caso, a massa de dados é muito maior. Também é diferente procurar um registro em um conjunto de dados que sofre poucas alterações (inserções/remoções) como, por exemplo, o conjunto de estados brasileiros; e procurar por uma venda, a partir do seu código, na base de dados de uma grande empresa de *e-commerce*, cujos dados mudam constantemente. No primeiro caso, o importante é minimizar o tempo de pesquisa sem preocupação com o tempo necessário para realizar inserções e remoções no conjunto de dados, uma vez que o mesmo sofre poucas alterações ao longo do tempo.

Fique por dentro

Este artigo é útil para todo desenvolvedor que pretende expandir seus conhecimentos sobre métodos de pesquisa (ou métodos de busca). Para isso, serão apresentados os conceitos básicos sobre três conhecidos métodos de pesquisa: pesquisa sequencial, pesquisa binária e pesquisa por tabela Hash. Juntamente com os métodos de pesquisa, trechos de código serão analisados, ilustrando a implementação de tais métodos na linguagem Java. Além disso, comentários sobre a eficiência de cada método, bem como sobre os cenários nos quais eles podem ser aplicados com sucesso são discorridos ao longo do texto.

A partir disso, neste artigo analisaremos conceitos, na teoria e na prática, da pesquisa interna (ou busca interna), na qual assume-se que o conjunto de dados a ser pesquisado é pequeno o suficiente para ser carregado de uma vez na memória principal (ou memória interna) do computador. Quando a quantidade de informações é grande o suficiente a ponto de não ser possível tratá-la de uma vez na memória principal, métodos de pesquisa externa são necessários. Esse tipo de método é capaz de lidar com conjuntos de dados que estão armazenados na memória auxiliar (externa) do computador, como o HD, fitas magnéticas, entre outros. A prioridade de cada categoria de algoritmos é diferente. Enquanto em uma pesquisa interna procura-se reduzir a quantidade de comparações realizadas pelo método escolhido, na pesquisa externa, além desse requisito, deve-se levar em consideração a quantidade de consultas ao disco necessárias para se encontrar a informação pesquisada.

Abordaremos três métodos de busca interna: sequencial, binária e utilizando a tabela *Hash*. No tópico “Conceitos Preliminares”

apresentaremos o modelo de estrutura de dados que será utilizado para a implementação dos métodos de pesquisa. Em seguida, nos tópicos “Pesquisa Sequencial”, “Pesquisa Binária” e “Pesquisa por Tabela Hash”, serão analisados os três principais métodos de pesquisa existentes na literatura, destacando suas principais características e estratégias de implementação e eficiência.

Conceitos preliminares

Este tópico apresenta alguns conceitos que são fundamentais para o acompanhamento deste artigo, tal como o conceito de análise da complexidade de algoritmos, que será amplamente discutido, e o conceito de “Dicionário”, como um tipo abstrato de dados para implementação de métodos de pesquisa.

A análise da complexidade de algoritmos

Um aspecto predominante na escolha de um método de pesquisa é o tempo gasto para realizá-las, bem como para manipular o conjunto de dados, inserindo ou removendo elementos. Para a pesquisa, a medida de complexidade relevante consiste no número de comparações entre chaves realizadas até que uma resposta seja dada pelo algoritmo. Quanto à inserção/remoção, leva-se em consideração também o número de movimentações (ou trocas) necessárias para acomodar um novo item ou remover um item existente do conjunto de dados.

Assim, as medidas de complexidade analisadas para cada método de pesquisa apresentado neste texto são $C(n)$ e $M(n)$, que correspondem, respectivamente, às funções de complexidade que descrevem o número de comparações e o número de movimentações realizadas por cada método, onde n é a quantidade de itens do conjunto de dados a ser pesquisado.

É importante ressaltar ainda que a maioria dos métodos de pesquisa é sensível à ordem inicial dos itens a serem pesquisados, isto é, o número de comparações e/ou movimentações realizadas por um método pode variar caso o conjunto esteja ordenado ou não ou se o elemento a ser pesquisado estiver no início ou no final do conjunto, entre outros. Assim, $C(n)$ e $M(n)$ devem ser considerados, sempre quando possível, para três casos:

- **O melhor caso:** corresponde ao menor número de comparações/movimentações sobre todas as possíveis entradas de tamanho n ;
- **O pior caso:** corresponde ao maior número de comparações/movimentações sobre todas as possíveis entradas de tamanho n ;
- **O caso médio (ou caso esperado):** corresponde à média do número de comparações/movimentações de todas as possíveis entradas de tamanho n .

O tipo abstrato de dados Dicionário

É muito importante considerar métodos de pesquisa como um Tipo Abstrato de Dado – TAD, isto é, uma estrutura de dados com um conjunto de operações associado a ela. O motivo é que um TAD promove a independência dos possíveis tipos de implementação para esses métodos de pesquisa. Por exemplo, um programador pode implementar o método de pesquisa sequencial com vetores, outro pode utilizar listas encadeadas, mas de qualquer forma,

todos os casos tratam do mesmo propósito (a pesquisa) e devem oferecer as mesmas funções aos seus usuários.

Um TAD comumente utilizado para pesquisa é o *dicionário*. Um dicionário é um TAD cujas operações são responsáveis por inicializar a estrutura de dados utilizada no dicionário, pesquisar um ou mais registros com determinada chave, inserir um novo registro e remover um registro específico.

A implementação dos métodos de pesquisa apresentados neste texto baseia-se no TAD dicionário e na linguagem de programação Java. Para simplificar a apresentação dos métodos de pesquisa, apenas as funções *inserir* e *pesquisar* serão discutidas. A linguagem Java foi escolhida porque tem sido utilizada em diversos livros-textos sobre estruturas de dados, como em Goodrich (2013), Sedgewick (2013), Ziviani (2007), além de ser uma linguagem amplamente conhecida, bem documentada e que apresenta características interessantes para o desenvolvimento de estruturas de dados genéricas.

K1Para a implementação do TAD dicionário, as interfaces **Item** e **Dicionario** foram criadas e são apresentadas na **Listagem 1**.

Listagem 1. Interfaces utilizadas nas implementações do TAD Dicionario.

```
public interface Item {  
    public static final int MENOR = -1;  
    public static final int IGUAL = 0;  
    public static final int MAIOR = 1;  
    public abstract int compara(Item it) throws Exception;  
}  
  
public interface Dicionario {  
    public Item pesquisar(Item it) throws Exception;  
    public void inserir(Item it) throws Exception;  
}
```

A interface **Item** representa a abstração de um item que será armazenado no dicionário. A única operação que todo item deve prover é a que compara dois itens distintos e diz se o primeiro é menor, maior ou igual ao segundo item – método **compara()**. Para melhorar a legibilidade do código, constantes públicas que representam as possíveis saídas do método de comparação entre itens foram criadas, a saber: **MENOR**, **IGUAL** e **MAIOR**. Também é importante salientar que o método **compara()** pode lançar uma exceção quando o programador tentar comparar itens de tipos diferentes.

Dito isso, uma possível implementação da classe **Item**, considerando um item cuja chave é um número inteiro, pode ser observada no trecho de código da **Listagem 2**.

A classe **MeuItem** implementa a interface **Item** e, consequentemente, o método **compara()**, considerando as características deste tipo de item. É importante ressaltar que, na prática, muitas vezes estamos mais interessados no conteúdo do item do que em sua chave. Assim, a classe **MeuItem** deveria conter os atributos

Estrutura de dados: Primeiros passos com métodos de busca

que representam a informação desejada. Por exemplo, no caso de um aluno, deveriam haver atributos que representam os dados de um aluno, como nome, endereço, entre outros; se fosse um produto, teríamos atributos para a descrição, o preço, o prazo de validade do produto, entre outros. Contudo, para simplificar a apresentação dos métodos de pesquisa neste artigo, apenas a chave do item será considerada.

Listagem 2. Exemplo de um tipo Item.

```
public class MeulItem implements Item {  
  
    private final int chave;  
  
    // informações extras sobre o item  
  
    public MeulItem(int chave) {  
        this.chave = chave;  
    }  
  
    @Override  
    public int compara(Item it) throws Exception {  
        if (it instanceof MeulItem) {  
            if (chave < ((MeulItem) it).chave) return MENOR;  
            else if (chave > ((MeulItem) it).chave) return MAIOR;  
            else return IGUAL;  
        } else throw new Exception("Não é possível comparar itens de  
        tipos diferentes!");  
    }  
}
```

A interface **Dicionario** especifica os métodos que qualquer implementação de um dicionário deve conter, ou seja, as operações de inserção e pesquisa de elementos no mesmo. A partir desta interface, diversas alternativas de implementação de um dicionário podem existir utilizando listas encadeadas, vetores, árvores, entre outros. Os tópicos “Pesquisa Sequencial”, “Pesquisa Binária” e “Pesquisa por Tabela Hash” apresentam algumas das possíveis formas de se implementar um dicionário a partir da interface criada, discutindo seus pontos fortes e desvantagens.

Pesquisa sequencial

A pesquisa sequencial é um dos métodos de pesquisa mais simples que existe. Ele funciona da seguinte forma: a partir do primeiro registro (ou do último), pesquisa sequencialmente até encontrar a chave procurada ou até chegar ao final (ou ao início) do conjunto de registros.

A **Figura 1** ilustra os passos da execução da busca sequencial para encontrar o elemento **g** no vetor **[a, b, c, d, e, f, g, h]**.

A implementação de um dicionário para a pesquisa sequencial, utilizando um vetor (*array*) de itens não ordenados, é apresentada na **Listagem 3**. Os métodos mais importantes desta listagem são **pesquisar()** e **inserir()**. O primeiro retorna o registro do item passado por parâmetro, caso ele exista no vetor denominado **itens**. Observa-se que a pesquisa ocorre elemento por elemento, até o que o mesmo seja encontrado ou até que o final do vetor seja

atingido. Caso o elemento não exista no vetor, o valor **null** é retornado. Já o método **inserir()** simplesmente insere o item passado por parâmetro na primeira posição vazia do vetor.

Índices do vetor:	1	2	3	4	5	6	7	8
Vetor	a	b	c	d	e	f	g	h
Passo 1	g							
Passo 2		g						
Passo 3			g					
Passo 4				g				
Passo 5					g			
Passo 6						g		
Passo 7							g	

Figura 1. Ilustração da execução do método de pesquisa sequencial

Listagem 3. Implementação do método de pesquisa sequencial com um vetor não ordenado.

```
1. public class ArrayDic implements Dicionario {  
2.  
3.     private final int TAM_MAX = 100;  
4.     private final Item itens[];  
5.     private int pos;  
6.  
7.     public ArrayDic() {  
8.         itens = new Item[TAM_MAX];  
9.         pos = 0;  
10.    }  
11.  
12.    public boolean estahCheia() {  
13.        return (pos == TAM_MAX);  
14.    }  
15.  
16.    @Override  
17.    public Item pesquisar(Item it) throws Exception {  
18.        for (int i = 0; i < pos; i++) {  
19.            if (itens[i].compara(it) == Item.IGUAL) {  
20.                return itens[i];  
21.            }  
22.        }  
23.        return null;  
24.    }  
25.  
26.    @Override  
27.    public void inserir(Item it) throws Exception {  
28.        if (!estahCheia()) {  
29.            itens[pos] = it;  
30.            pos++;  
31.        } else {  
32.            throw new Exception("Capacidade máxima atingida!");  
33.        }  
34.    }  
35.}
```

Algumas características da estratégia de pesquisa sequencial são:

1. Por se tratar de um vetor não ordenado, a inserção de qualquer elemento é feita de forma eficiente, inserindo-o sempre no final do vetor, com um custo computacional constante, isto é, $M(n) = 1$;
2. O método **pesquisar()** executa em tempo linear, realizando, no pior caso, $C(n) = n$ comparações para encontrar um elemento. Para o caso médio, $C(n) = \frac{n+1}{2}$.

Uma vantagem da pesquisa sequencial é que ela é bastante flexível, permitindo a utilização de conjuntos de itens que possuem diferentes tipos de chaves e não apenas chaves numéricas, bem como conjuntos que não estejam necessariamente ordenados.

Uma maneira de melhorar a performance do método **pesquisar()** é utilizar um registro sentinel. Trata-se de um registro contendo a chave de pesquisa que é colocado no início do vetor de itens. Este evita que uma comparação a mais por item seja feita, para verificar se a busca chegou ao final do vetor (linha 18). O custo de se utilizar a sentinel é a alocação de um espaço de memória extra no vetor de itens. A **Listagem 4** apresenta como seria o método pesquisa da **Listagem 3** caso uma sentinel fosse utilizada.

Listagem 4. Implementação do método pesquisar() com uma sentinel.

```
public Item pesquisar(Item it) throws Exception {  
    int i = pos - 1;  
    while (itens[i].compara(it) != Item.IGUAL) i--;  
    if (i == 0) return itens[i];  
    else return null;  
}
```

Outra possível implementação do método de pesquisa sequencial é manter o vetor de itens ordenado. Uma busca sem sucesso (ou seja, quando o elemento pesquisado não está presente no vetor) em um vetor não ordenado fará o algoritmo executar n vezes, sempre. Já com um vetor ordenado, uma busca sem sucesso poderá parar antes de visitar todos os n elementos. Por exemplo, considerando o vetor [2, 3, 5, 7] e o elemento 4 a ser pesquisado, a busca pode parar após ter comparado 4 com 5, pois como 5 é maior do que 4, não há possibilidade de o quatro estar no restante do vetor. Assim, o número de comparações $C(n)$ para o pior caso cai para $C(n) = \frac{n+1}{2}$, que é o mesmo custo para o caso médio.

Porém, manter o vetor ordenado torna a inserção dependente da quantidade de elementos existentes no vetor. Isto porque, agora, o elemento não poderá mais ser inserido no final do vetor, mas sim na posição adequada para se manter a ordenação. Deste modo, vale a pena manter o vetor ordenado caso o número de consultas seja bem maior do que a quantidade de inserções. Neste caso, a quantidade de movimentações $M(n)$ exigidas será: $M(n) = 1$ para o melhor caso; $M(n) = n$ no pior caso; e $M(n) = \frac{n+1}{2}$ para o caso médio.

Para implementarmos a busca sequencial utilizando vetores ordenados os métodos **inserir()** e **pesquisar()** devem ser alterados, conforme pode ser visto na **Listagem 5**.

Alguns autores investigam a utilização de listas encadeadas, em vez de vetores, para a implementação do método de pesquisa sequencial. Segundo os mesmos, isso exigirá mais memória, uma vez que um ponteiro para cada elemento do conjunto de itens deverá ser armazenado; contudo, não haverá a necessidade de se saber antecipadamente a quantidade de elementos a serem inseridos no conjunto.

Uma implementação do dicionário com uma lista encadeada simples é apresentada na **Listagem 6** (neste exemplo, o conjunto

de elementos não está ordenado). Note que há uma classe interna, denominada **Noh**, para representar cada elemento da lista. O atributo **lista**, por sua vez, é uma referência para o nó inicial da lista e cada nó possui, além da informação de interesse do usuário, uma referência para o próximo nó.

Listagem 5. Implementação do método de pesquisa sequencial com um vetor ordenado.

```
@Override  
public Item pesquisar(Item it) throws Exception {  
    int i = 0;  
    for (; i < pos; i++) {  
        if (itens[i].compara(it) != Item.MENOR) break;  
    }  
    if (i == pos) return null;  
    if (itens[i].compara(it) == Item.IGUAL) return itens[i];  
    return null;  
}  
  
@Override  
public void inserir(Item it) throws Exception {  
    if (!estahCheia()) {  
        int i = pos;  
        while (i > 0 && itens[i - 1].compara(it) == Item.MAIOR) {  
            itens[i] = itens[i - 1];  
            i--;  
        }  
        itens[i] = it;  
        pos++;  
    } else {  
        throw new Exception("Capacidade máxima atingida!");  
    }  
}
```

Listagem 6. Implementação do método de pesquisa sequencial com lista encadeada.

```
public class ListDic implements Dionario {  
  
    private class Noh {  
        private Item item;  
        private Noh prox;  
  
        public Noh(Item it, Noh prox) {  
            this.item = it;  
            this.prox = prox;  
        }  
  
        private Noh lista;  
  
        public ListDic() {  
            lista = null;  
        }  
  
        @Override  
        public Item pesquisar(Item it) throws Exception {  
            Noh aux = lista;  
            while (aux != null) {  
                if (aux.item.compara(it) == Item.IGUAL) {  
                    return aux.item;  
                }  
                aux = aux.prox;  
            }  
            return null;  
        }  
  
        @Override  
        public void inserir(Item it) throws Exception {  
            Noh noh = new Noh(it, lista);  
            lista = noh;  
        }  
    }  
}
```

Pesquisa binária

Como foi visto anteriormente, manter o vetor ordenado para realização de uma pesquisa sequencial pode ser útil, pois o tempo de execução médio para buscas sem sucesso cai pela metade. Porém, o custo para a inserção de um elemento em um vetor ordenado é significativamente maior do que para a inserção em um vetor não ordenado.

Ainda sobre buscas, há um método de pesquisa que pode utilizar melhor a vantagem de termos um vetor ordenado para melhoria do tempo consumido para localizar um elemento. Esse método é conhecido como pesquisa binária e será descrito nesta seção.

A ideia da pesquisa binária é dividir o conjunto de itens em duas partes, determinar em qual dessas duas partes o elemento pesquisado pode pertencer e então concentrar a busca apenas nessa parte. A estratégia utilizada para resolução deste problema é conhecida como “dividir para conquistar” e está presente em vários tipos de algoritmos, como os de ordenação *QuickSort*, *MergeSort*, entre outros. A ideia é reduzir um problema maior, que não se sabe como resolver a priori, em problemas menores, para os quais se conhece a solução. Depois, utilizar tal solução para resolver o problema maior.

Pesquisa binária – Implementação com Vetor

Para saber se um elemento pertence a um vetor utilizando a pesquisa binária, compara-se a chave deste elemento com a chave do elemento que está no meio do conjunto (lembrando que o vetor deve estar ordenado). Se a chave for igual à procurada, então a busca termina com sucesso; caso a chave seja menor, então o elemento procurado deve estar na primeira parte do vetor (do início até o elemento do meio - 1); se a chave for maior, então o elemento deve estar na segunda parte do vetor, que vai do elemento que está após o elemento do meio até o último elemento. Esse mesmo procedimento descrito deve ser repetido na metade escolhida até que se encontre o elemento ou até que todos os elementos do vetor tenham sido investigados sem sucesso.

A **Figura 2** ilustra os passos da execução de uma pesquisa binária para encontrar o elemento **g** no vetor **[a, b, c, d, e, f, g, h]**.

Indices do vetor:	1	2	3	4	5	6	7	8
Vetor	a	b	c	d	e	f	g	h
Passo 1					e	f	g	h
Passo 2							g	h

Figura 2. Ilustração da execução do método de pesquisa binária

O trecho de código da **Listagem 7** apresenta uma versão recursiva do método de pesquisa binária. Como podemos verificar, o método **pesquisar()** delega o trabalho de encontrar o elemento para a função **pesquisaRecursiva()**. Esta recebe, além do elemento a ser pesquisado, dois parâmetros inteiros **e** e **d** que correspondem, respectivamente, aos índices do primeiro e do último elemento do vetor que está sendo analisado no momento.

É por meio desses parâmetros que o sistema controla qual parte do vetor deve ser analisada a cada passo.

Listagem 7. Implementação do método de pesquisa binária em um vetor.

```
1. private Item pesquisaRecursiva(Item it, int e, int d) throws Exception {  
2.     if (e > d) return null;  
3.     int m = (e + d)/2;  
4.     if (itens[m].compara(it) == Item.IGUAL) return itens[m];  
5.     else if (itens[m].compara(it) == Item.MENOR)  
6.         return pesquisaRecursiva(it, m + 1, d);  
7.     else return pesquisaRecursiva(it, e, m - 1);  
8. }  
9.  
10. @Override  
11. public Item pesquisar(Item it) throws Exception {  
12.     return pesquisaRecursiva(it, 0, pos - 1);  
13. }
```

A função **pesquisaRecursiva()** possui dois casos base: quando o elemento é encontrado (linha 4); e quando o valor do parâmetro **e** é maior do que o valor do parâmetro **d** (linha 2), o que significa que todo o vetor foi pesquisado e que o elemento procurado não se encontra nele. Como passos recursivos, há duas possibilidades. A primeira ocorre quando a chave do elemento procurado é maior do que a chave do elemento do meio do vetor (linha 5). Neste caso, a função **pesquisaRecursiva()** é invocada recursivamente, mas o vetor a ser considerado é aquele que vai do índice do elemento do meio + 1 até o índice do último elemento do vetor original. E a segunda possibilidade ocorre quando a chave do elemento procurado é menor do que a chave do elemento do meio do vetor (linha 7). Neste caso, a função **pesquisaRecursiva()** também é invocada recursivamente, mas o vetor a ser considerado é aquele que vai do índice do primeiro elemento do vetor original até o índice do elemento do meio - 1.

A quantidade de comparações realizadas pelo algoritmo de pesquisa binária é dada por: $C(n) = 1 + C(n/2)$. Isto é, o número de comparações realizadas pela pesquisa binária para um vetor de tamanho n é igual a uma comparação mais o custo de encontrar o elemento na metade do vetor, que é dado por $C(n/2)$, o que leva à seguinte função de complexidade: $C(n) = \lg n + 1$.

Isto quer dizer que, para casos de sucesso ou insucesso, a pesquisa binária nunca executará em mais do que $\lg n + 1$ passos, o que é um resultado significativamente melhor do que os resultados do método de pesquisa sequencial. Por exemplo, para $n = 10^6$, o método de pesquisa sequencial gastaria 106 passos no caso de uma busca sem sucesso, uma vez que sua complexidade do pior caso é proporcional a n . Já o método de pesquisa binária executaria em $\lg 10^6$, que é aproximadamente igual a 20.

Entretanto, é importante levar em consideração o custo para manter o vetor ordenado, isto é, para cada inserção de um novo elemento. No pior caso, n movimentações serão realizadas. Sendo assim, a busca binária, da maneira como foi implementada nesta seção, deve ser utilizada em conjuntos pouco dinâmicos, em que a quantidade de consultas realizadas seja bem maior do que a quantidade de inserções feitas no conjunto de elementos.

O uso de listas encadeadas para implementação da pesquisa binária pode ser uma estratégia não muito promissora, uma vez que a eficiência da busca binária depende da capacidade de se acessar rapidamente o elemento do meio de um conjunto e para atingir o elemento do meio utilizando uma lista encadeada, seria necessário percorrer, sequencialmente, todos os elementos anteriores a este, utilizando o encadeamento da lista.

Com o intuito de otimizar ainda mais o número de comparações exigido pela pesquisa binária, podemos adotar uma heurística conhecida como **busca interpolada**. A busca interpolada baseia-se na ideia de que se o valor de uma chave é muito pequeno, é melhor procurar mais no início do vetor do que exatamente no meio. Para isso, utiliza-se uma proporção entre a “distância” da chave procurada para a chave inicial do vetor, com relação à “distância” entre a chave final do vetor e a chave inicial do mesmo. Essa heurística pode aumentar bastante a eficiência da pesquisa binária, reduzindo o número médio de comparações de lgn para $lg\lg n$. Por exemplo, enquanto $\lg 10^6 \approx 20$, $\lg \lg 10^6 \approx 4$. Porém, essa heurística funciona apenas para registros cujas chaves são numéricas e seus resultados são promissores apenas quando as chaves estão uniformemente distribuídas.

Pesquisa binária – Implementação com Árvores Binárias de Busca (ABBs)

Grandes benefícios em termos da redução do número de comparações podem ser obtidos com o uso do método de pesquisa binária em um vetor ordenado. Contudo, o custo para inserção de elementos de forma ordenada neste vetor é alto (proporcional a n). Para se obter boas taxas de execução, tanto para busca quanto para inserção de elementos, estruturas mais sofisticadas, como as Árvores de Binária de Busca (ABBs) – ou Árvores Binárias de Pesquisa – devem ser utilizadas. Para demonstrar isso, este tópico apresenta a implementação de um dicionário utilizando uma ABB.

Uma árvore binária é definida como uma árvore vazia (sem nós) ou uma árvore com nó raiz conectado a um par de árvores binárias, as quais são denominadas sub-árvore esquerda e sub-árvore direita. Uma ABB é uma árvore binária em que, para cada nó, a seguinte propriedade é verdadeira: todos os registros com chaves menores do que a chave deste nó estão em sua sub-árvore esquerda e todos os registros com chaves maiores estão em sua sub-árvore direita.

A **Figura 3** apresenta alguns exemplos de árvores binárias, destacando quais são e quais não são ABBs.

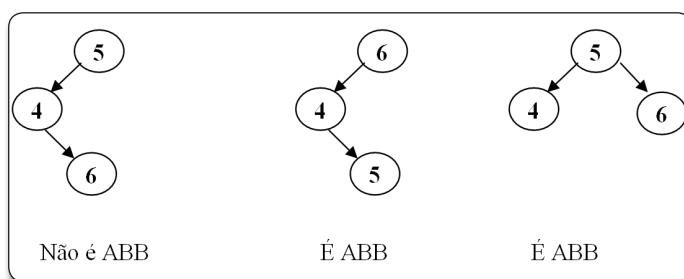


Figura 3. Exemplos de árvores binárias

O trecho de código da **Listagem 8** apresenta uma parte da implementação de um dicionário com a utilização de uma ABB.

Listagem 8. Trecho da implementação de um dicionário com ABB.

```
public class ABBDic implements Dicionario {
    private class Noh {
        private Item item;
        private Noh esq, dir;

        public Noh(Item it, Noh esq, Noh dir) {
            this.item = it;
            this.esq = esq;
            this.dir = dir;
        }
    }

    private Noh raiz;

    public ABBDic() {
        raiz = null;
    }
}
```

A classe **ABBDic** possui um atributo denominado **raiz** que consiste em uma referência para a raiz da árvore. Além disso, cada nó, além do item que representa o registro com as informações de interesse do usuário, contém referências para as sub-árvore esquerda e direita (atributos **esq** e **dir**, respectivamente).

O método de pesquisa binária com ABBs segue a mesma lógica da pesquisa binária com vetores, só que dessa vez, em vez de dividir o vetor em duas partes, reduzimos nosso espaço de busca ao procurar o elemento em uma das duas sub-árvore do nó raiz. Para isso, devemos comparar o elemento pesquisado à raiz da árvore e decidir em qual sub-árvore devemos continuar a busca. O trecho de código da **Listagem 9** apresenta a implementação do método **pesquisaRecursiva()** para ABBs.

Listagem 9. Implementação do método de pesquisa binária em uma ABB.

```
private Item pesquisaRecursiva(Item it, Noh raiz) throws Exception {
    if (raiz == null) return null;
    else if (raiz.item.compara(it) == Item.IGUAL) return raiz.item;
    else if (raiz.item.compara(it) == Item.MENOR)
        return pesquisaRecursiva(it, raiz.dir);
    else return pesquisaRecursiva(it, raiz.esq);
}
```

Com relação à inserção de elementos, para que ela seja realizada na árvore, inicialmente deve-se encontrar o lugar certo a inserir o nó. A **Listagem 10** apresenta o método de inserção de um elemento em uma ABB.

Agora, para definir a complexidade dos métodos **inserirRecursivo()** e **pesquisaRecursiva()**, apresentados nas **Listagens 9 e 10**, é importante entender o conceito de altura de uma árvore. A altura de uma árvore é dada pelo comprimento do caminho mais longo da sua raiz até um nó folha. Os nós encontrados durante a recursão dos métodos **inserirRecursivo()** e **pesquisaRecursiva()**

formam um caminho partindo da raiz que, no pior caso, chega ao nó folha mais distante da raiz da árvore. Sendo assim, o número de comparações realizadas por estes métodos é proporcional a h , onde h representa a altura da árvore.

Listagem 10. Implementação do método de inserção em uma ABB.

```
private Noh inserirRecursivo(Item it, Noh raiz) throws Exception {  
    if (raiz == null) return new Noh(it, null, null);  
    else if (raiz.item.compara(it) == Item.MENOR)  
        raiz.dir = inserirRecursivo(it, raiz.dir);  
    else raiz.esq = inserirRecursivo(it, raiz.esq);  
    return raiz;  
}  
  
@Override  
public void inserir(Item it) throws Exception {  
    this.raiz = inserirRecursivo(it, this.raiz);  
}
```

Para uma árvore perfeitamente balanceada, isto é, uma árvore cujos nós internos têm dois filhos e todas as folhas estão no mesmo nível, é possível provar que $h = \lg n$, portanto, pesquisa e inserção podem ser realizadas com $C(n) = \lg n$ comparações.

Um ponto importante a ser destacado é que, ao contrário do que acontece na implementação do dicionário com vetores, tanto o tempo de busca quanto o tempo de inserção de um elemento no conjunto, utilizando ABB, é proporcional a $\lg n$. No caso do dicionário com vetores, o tempo de inserção de um elemento é proporcional a n , ou seja, é bem maior do que $\lg n$.

Contudo, é importante ressaltar que as ABBs sofrem de um problema que pode prejudicar seu tempo de execução. Quando os elementos de uma ABB são inseridos em ordem, crescente ou decrescente, devido à sua propriedade, os elementos formarão uma grande lista encadeada, o que fará com que a altura da árvore seja igual a $n - 1$, onde n refere-se à quantidade de elementos inseridos na árvore. Além disso, é fácil prever que, após várias operações de inserção/remoção, uma ABB tende a ficar desbalanceada, já que essas operações não garantem o balanceamento. Para tratar desse problema, há estruturas mais sofisticadas, conhecidas como árvores平衡adas. Alguns exemplos de árvores平衡adas são AVL, Árvore Rubro-Negra, entre outras.

Pesquisa por Tabela Hash

Os métodos de pesquisa apresentados anteriormente são baseados na comparação da chave de pesquisa com as chaves do conjunto de dados. O método de pesquisa por tabela hash (ou *hashing*) é completamente diferente: os registros armazenados em uma tabela são diretamente endereçados a partir de uma transformação aritmética realizada sobre a chave de pesquisa.

O *hashing* é uma extensão do método de pesquisa indexado por chave (*key-indexed search method*) que usa valores das chaves como índices de um vetor, em vez de compará-las. A grande vantagem deste método é que inserções, remoções e consultas a elementos do conjunto podem ser realizadas com um número de comparações

e movimentações constante. Porém, esse método só é aplicável a chaves inteiras positivas, não duplicadas e menores do que M , onde M é o tamanho da tabela de endereçamento.

Uma tabela hash pode ser descrita como uma estrutura de dados eficaz para implementação de um dicionário. Embora a busca por um elemento em uma tabela de espalhamento possa demorar tanto quanto em uma lista encadeada – o número de comparações no pior caso é proporcional a n – na prática, *hashing* funciona bem. Em geral, o número de comparações médio para pesquisar um elemento em uma tabela de espalhamento é constante. Esse é um tipo de estrutura interessante para aplicações (por exemplo, um compilador precisa realizar inúmeras consultas em sua tabela de símbolos durante a compilação de um programa) em que a busca em um conjunto de dados é realizada tantas vezes que um número de comparações proporcional a $\lg n$ ainda é alto.

Um método de pesquisa que utiliza *hashing* é constituído de duas etapas principais:

- 1. Computar o valor da função de transformação ou função hashing:** responsável por transformar a chave de pesquisa em um endereço da tabela hash;
- 2. Tratar as colisões:** considerando que duas ou mais chaves podem ser transformadas em um mesmo endereço, é necessário realizar o tratamento de colisões.

Função de transformação

Uma função de transformação é responsável por mapear chaves em inteiros dentro do intervalo $[0..M-1]$, onde M é o tamanho da tabela na qual os dados estarão armazenados. A função de transformação ideal é aquela que: (i) é simples de ser computada; e (ii) para cada chave de entrada, qualquer valor entre $[0..M-1]$ é igualmente provável de ocorrer.

Várias funções de transformação têm sido estudadas por pesquisadores ao longo dos anos. Uma destas funções, que é bem simples e que funciona muito bem, é a que utiliza o resto da divisão por M : $h(K) = K \bmod M$, onde $h(K)$ é a chave transformada e K é um inteiro correspondente à chave de pesquisa. Nesta opção, o valor de M deve ser escolhido com bastante cautela. Vejamos um exemplo para entender por quê: se M for par, então $h(K)$ será par quando K for par e será ímpar quando K for ímpar, uma vez que o resto da divisão de um número par pelo outro é par e o resto da divisão de um número ímpar pelo outro é ímpar. Isso leva a função de transformação a não espalhar as chaves de forma uniforme. Por exemplo, se 90% das chaves forem pares, esses 90% irão ocupar 50% da tabela de endereçamento e os outros 10% ocuparão os 50% restantes.

O ideal é que M seja um número primo, mas não qualquer primo. Segundo Cormen *et al.* (2012), o ideal é escolher um primo não muito próximo de uma potência de 2. Por exemplo, se desejamos criar uma tabela de espalhamento para conter aproximadamente 2000 chaves e não nos importamos de examinar, por exemplo, uma média de três elementos em uma busca malsucedida, podemos escolher uma tabela de espalhamento de tamanho $M = 701$, uma vez que 701 é um primo próximo de $2000/3$ e não muito próximo

de uma potência de 2 (a menor potência antes de 701 é 512 e a maior potência depois de 701 é 1024).

Como as transformações sobre as chaves são aritméticas (conforme o método apresentado anteriormente), devemos transformar as chaves não numéricas em números. Considerando uma **String** (conjunto de caracteres) como chave, uma alternativa seria somar o decimal de cada caractere desta **String**, de acordo com sua representação na tabela ASCII. Contudo, essa não é uma estratégia muito boa para *strings*, para as quais a ordem dos elementos é relevante. Por exemplo, utilizando essa estratégia, as *strings* "temp01" e "temp10" terão a mesma representação numérica, da mesma forma que as palavras "stop", "pots", "spot" e "tops".

A transformação de uma chave não numérica para uma chave numérica deve levar em consideração a posição dos elementos nesta chave. Uma alternativa é escolher uma constante $a > 1$ e transformar a chave não numérica da seguinte forma:

$$(x_1 * a^{k-1}) + (x_2 * a^{k-2}) + \dots + (x_{n-1} * a) + x_n$$

Por exemplo, dados $a = 5$, a **String** "aab" e considerando que o valor de cada letra corresponde à sua posição no alfabeto da língua portuguesa, temos:

$$(1 * 5^2) + (1 * 5^1) + 2 = 32$$

Caso a **String** fosse "baa", o resultado seria diferente:

$$(2 * 5^2) + (1 * 5^1) + 1 = 56$$

O método **hashCode()**, da linguagem Java, tem exatamente a função de transformar uma chave não numérica em um inteiro. Tal função utiliza a estratégia comentada anteriormente, com $a = 31$. Estudos experimentais sugerem que 31, 33, 37, 39 e 41 são boas opções para o valor de a , quando as **Strings** se referem a palavras da língua inglesa.

Tratamento de colisões

Mesmo que se obtenha uma função de transformação que distribua os registros de forma uniforme entre as entradas da tabela de espalhamento, existe alta probabilidade de haver colisões. O paradoxo do aniversário diz que, em um grupo de 23 pessoas juntas ao acaso, existe uma chance maior do que 50% de que duas pessoas comemorem aniversário no mesmo dia.

Fazendo uma analogia com a questão da tabela de espalhamento e das colisões, isso quer dizer que se for utilizada uma função uniforme que enderece 23 chaves randômicas em uma tabela de tamanho $M = 365$, a probabilidade de que haja colisões é maior do que 50%.

Tratamento de colisões com listas encadeadas

Uma das formas de resolver colisões é simplesmente construir uma lista linear encadeada para cada endereço da tabela. Assim, todas as chaves com mesmo endereço são encadeadas em uma lista

linear. Este método é conhecido também como endereçamento externo (ou encadeamento separado).

Exemplo: se a i -ésima letra do alfabeto é representada pelo número i e a função de transformação $h(\text{Chave}) = \text{Chave mod } M$ é utilizada para $M = 7$, o resultado da inserção das chaves P E S Q U I S A na tabela *hash* ocorre como ilustrado na **Figura 4**.

Assumindo que qualquer item do conjunto tem igual probabilidade de ser endereçado para qualquer entrada, então o comprimento esperado de cada lista encadeada é de N/M , onde N representa o número de registros na tabela e M representa o tamanho da tabela. Esse valor é chamado de fator de carga da tabela de espalhamento, que pode ser menor, igual ou maior do que 1.

Chave textual	Chave numérica	$h(\text{Chave numérica})$
P	16	2
E	5	5
S	19	5
Q	17	3
U	21	0
I	9	2
A	1	1

T

```

graph TD
    T[0] --> U1["U"]
    T[1] --> A1["A"]
    T[2] --> P1["P"]
    T[3] --> Q1["Q"]
    T[4] --> nil1["nil"]
    T[5] --> E1["E"]
    T[6] --> nil2["nil"]
    U1 --> nil3["nil"]
    A1 --> nil4["nil"]
    P1 --> Q1
    Q1 --> nil5["nil"]
    E1 --> S1["S"]
    S1 --> S2["S"]
    S2 --> nil6["nil"]
  
```

Figura 4. Tabela hash com endereçamento externo

Logo, as operações de pesquisa, inserção e retirada terão um custo proporcional a $1 + N/M$, em média. Deste cálculo, 1 representa o tempo para encontrar a entrada na tabela e N/M é o tempo para percorrer a lista encadeada. Isso significa que se o número de posições da tabela de espalhamento é, no mínimo, proporcional ao número de elementos na tabela, temos que as operações de pesquisa, inserção e retirada terão um custo esperado constante.

O pior caso para esta estratégia ocorre quando a função de transformação leva todas as chaves a uma mesma posição da tabela de espalhamento, criando uma única lista de comprimento igual a N . Neste caso, o pior para a busca será proporcional a N .

O trecho de código da **Listagem 11** apresenta a implementação de um dicionário por meio de uma tabela de espalhamento. Neste caso, para tratamento de colisões, o endereçamento externo com listas encadeadas é utilizado.

Nesta implementação, a função **hash()** chama a função **hashCode()** do Java para gerar a chave numérica utilizada na transformação. O valor de M é instanciado com **maxN / 5**, onde **maxN** é a quantidade de chaves que serão adicionadas pelo usuário. Isto é feito porque queremos que cada lista contenha, em média, 5 elementos, visando reduzir o tempo de uma busca sem sucesso.

Estrutura de dados: Primeiros passos com métodos de busca

Listagem 11. Implementação do método de pesquisa por tabela hash (endereçamento externo).

```
public class HashingDic implements Dicionario {  
  
    private class Noh {  
        private Item item;  
        private Noh prox;  
  
        public Noh(Item it, Noh prox) {  
            this.item = it;  
            this.prox = prox;  
        }  
    }  
  
    private Noh table[];  
    private int M;  
  
    public HashingDic(int maxN) {  
        M = maxN / 5;  
        table = new Noh[M];  
    }  
  
    private Item pesquisarLista(Item it, Noh lista) throws Exception {  
        Noh aux = lista;  
        while (aux != null) {  
            if (aux.item.compara(it) == Item.IGUAL) {  
                return aux.item;  
            }  
            aux = aux.prox;  
        }  
        return null;  
    }  
  
    private int hash(Item it, int M) {  
        return it.toString().hashCode() % M;  
    }  
  
    @Override  
    public Item pesquisar(Item it) throws Exception {  
        return pesquisarLista(it, table[hash(it, M)]);  
    }  
  
    @Override  
    public void inserir(Item it) throws Exception {  
        int i = hash(it, M);  
        table[i] = new Noh(it, table[i]);  
    }  
}
```

A função `pesquisar()` delega a busca pelo elemento à função `pesquisaLista()`. Para isso, a função `pesquisar()` recupera a lista onde o elemento deve estar aloocado a partir do índice retornado pela função `hash()`. A função inserir, da mesma forma, utiliza o índice retornado pela função `hash()` para inserir o elemento no início da lista correspondente a este índice.

Tratamento de colisões com endereçamento aberto

A estratégia do endereçamento externo é interessante, contudo requer o uso de uma estrutura de dados auxiliar – lista encadeada – para armazenar os elementos que sofreram colisões.

Quando o número de registros a serem armazenados na tabela puder ser previamente estimado, não haverá necessidade de usar listas encadeadas para armazenar os registros. Neste sentido, há

vários métodos para armazenar N registros em uma tabela de tamanho $M > N$, isto é, com o número de registros menor do que a tabela. Estes métodos utilizam os lugares vazios na própria tabela para resolver as colisões e são chamados de **endereçamento aberto**. Em outras palavras, todas as chaves são armazenadas na própria tabela, sem a necessidade de listas encadeadas.

No entanto, diferentemente do endereçamento externo, no endereçamento aberto a tabela de espalhamento pode ficar “cheia”, de tal forma que nenhuma inserção adicional possa ser realizada. Ou seja, o fator de carga da tabela nunca será maior do que 1.

Quando uma chave x é endereçada para uma entrada da tabela que já esteja ocupada, uma sequência de localizações alternativas $h_1(x), h_2(x), \dots$ é escolhida. Se nenhuma posição estiver vazia, então a tabela está cheia e não podemos inserir a chave x .

As propostas para a escolha de localizações alternativas são diversas. A mais simples é chamada de *hashing linear*, onde a posição h_j na tabela é dada por: $h_j = (h(x) + j) \bmod M$, para $1 \leq j \leq 1$. Em outras palavras, para inserir um elemento na tabela *hash* utilizando endereçamento aberto, examinamos sucessivamente a tabela de espalhamento até encontrar uma posição vazia na qual podemos inserir a chave. Por exemplo: se a i-ésima letra do alfabeto é representada pelo número i e a função de transformação $h(\text{Chave}) = \text{Chave} \bmod M$ é utilizada para $M = 7$, então o resultado da inserção das chaves L U N E S na tabela, usando *hashing linear* para resolver colisões, é o exposto na **Figura 5**.

Chave textual	Chave numérica	$h(\text{Chave numérica})$
L	12	5
U	21	0
N	14	0
E	5	5
S	19	5

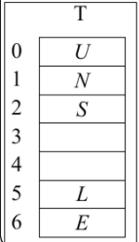


Figura 5. Tabela hash com endereçamento aberto

O *hashing linear* é fácil de implementar, mas sofre de um problema conhecido como agrupamento. Esse fenômeno ocorre quando a tabela começa a ficar “cheia”, pois a inserção de uma nova chave tende a ocupar uma posição contígua a outras já ocupadas, o que deteriora o tempo necessário para novas pesquisas.

No exemplo da figura anterior, para encontrar a chave S , teremos que realizar cinco comparações, uma vez que S está há cinco posições de distância de sua posição original (aquele retornada pela função de transformação). O custo médio necessário para

uma busca com sucesso em uma tabela de espalhamento com *hashing linear* é: $\frac{1}{2}\left(1 + \left(\frac{1}{1-a}\right)\right)$, onde $a = N/M$.

Segundo dados experimentais, para uma tabela *hash* com 50% de fator de carga ($a = 0.5$), em média, uma consulta com sucesso realiza menos do que duas comparações, o que é um resultado interessante.

A **Listagem 12** apresenta a versão do dicionário que utiliza tabelas de espalhamento com endereçamento aberto. Note que a função de hash foi omitida, pois é a mesma do código da **Listagem 11**.

Listagem 12. Implementação do método de pesquisa por tabela hash (endereçamento aberto).

```
public class HashingLinearDic implements Dicionario {

    private final Item table[];
    private final int M;

    public HashingLinearDic(int maxN) {
        M = maxN * 5;
        table = new Item[M];
    }

    @Override
    public Item pesquisar(Item it) throws Exception {
        int i = hash(it, M);
        while (table[i] != null) {
            if (table[i].compara(it) == Item.IGUAL) {
                return table[i];
            } else {
                i = (i + 1) % M;
            }
        }
        return null;
    }

    @Override
    public void inserir(Item it) throws Exception {
        int i = hash(it, M);
        while (table[i] != null) {
            i = (i + 1) % M;
        }
        table[i] = it;
    }
}
```

Como vantagens da utilização de tabelas de espalhamento, cita-se: (i) alta eficiência de custo de pesquisa no caso médio; e (ii) simplicidade de implementação. Como desvantagens, tem-se: (i) o custo para recuperar os registros inseridos na tabela em ordem crescente é alto, sendo necessário ordená-los; e (ii) seu pior caso é proporcional a N .

A **Figura 6** apresenta a comparação entre as diversas implementações de um dicionário (vetor ordenado, vetor não ordenado, lista encadeada ordenada, lista encadeada não ordenada, busca binária com vetores, busca binária com ABB e *hashing*) quanto à quantidade de comparações realizadas pelos métodos *inserir* e *pesquisar*, em seus piores casos e casos médios.

Tipo de Implementação	Pior caso		Caso médio		
	Inserir	Pesquisar	Inserir	Pesquisa sucesso	Pesquisa insucesso
Vetor ordenado	N	N	N/2	N/2	N/2
Lista encadeada ordenada	N	N	N/2	N/2	N/2
Vetor não ordenado	1	N	1	N/2	N
Lista encadeada não ordenada	1	N	1	N/2	N
Busca binária com vetores	N	lgN	N/2	lgN	lgN
Busca binária com ABB	N	N	lgN	lgN	lgN
Hashing	1	N	1	1	1

Figura 6. Comparação entre as diferentes implementações de um dicionário

A pesquisa binária é o primeiro método de busca que traz o número de comparações no caso médio abaixo de N – no caso, lgN . Com pesquisas sequenciais, o máximo que se consegue alcançar é algo proporcional a N . A vantagem de se utilizar ABB para a implementação da pesquisa binária está no fato de que a inserção também pode ser realizada com lgN comparações no pior caso, mas, para isso, a ABB deve estar devidamente balanceada.

Por fim, pode-se notar que tabelas de espalhamento conseguem atingir eficiência muito boa para buscas com e sem sucesso, no caso médio, chegando a apresentar um número de comparações constante. Contudo, essa não é uma boa estrutura a ser utilizada quando o objetivo é recuperar os registros da tabela em ordem crescente ou decrescente.

Autor



Paulo Afonso Parreira Júnior

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

Atualmente é 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:

Cormen, T. H. et al. (2012). "Algoritmos: teoria e prática". Tradução da 3ª edição americana.

Goodrich, M. T. e Tamassia, R. (2013) "Estruturas de Dados & Algoritmos em Java". 5ª edição.

Sedgewick, R. (2013). "Algorithms in Java. Parts 1-4". 3ª edição.

Ziviani, N. (2007) "Projeto de Algoritmos com implementações em Java e C++".

Vantagens e desvantagens dos padrões de projeto Singleton e Flyweight

Conheça neste artigo as vantagens, desvantagens, quando e como utilizar os padrões de projeto Singleton e Flyweight

O que seria de nós se não existissem os padrões? A reflexão acerca deste questionamento pode parecer sem nexo, contudo, refletir sobre os benefícios da padronização vem ao encontro com o assunto abordado neste artigo, quase que em sua totalidade.

A fim de estimular o estudo sobre padrões, podemos utilizar um exemplo: imagine se as posições das letras dos teclados de computadores variassem de acordo com cada fabricante? Para o usuário seria muito ruim, pois ele teria que se adaptar aos vários tipos de teclados, tornando tarefas simples extremamente ineficientes, como digitar este artigo, por exemplo.

Assim como os padrões são fundamentais em diversas áreas, não poderia ser diferente na Engenharia de Software. Nesta área, podemos, por exemplo, documentar um software de várias maneiras, seja através de um documento de texto, um desenho em uma folha de papel ou até mesmo em uma planilha. No entanto, quando utilizamos um padrão, como a *Unified Modeling Language* (UML), propiciamos uma compreensão mais clara e objetiva da documentação gerada.

Assim como a UML para a documentação, no desenvolvimento de software a adoção de padrões favorece a criação de um código-fonte melhor no que diz respeito à compreensão, além de colaborar para a manutenção e a evolução do projeto. Com o propósito de alcançarmos esses benefícios e apoiar o desenvolvedor na resolução de problemas comuns, como o acesso a diversas fontes de dados, a localização de um serviço, a mediação entre o formulário e a classe controladora, a auditoria das ações do usuário, o mecanismo de autenticação e autorização, entre outros, existem técnicas que visam

Fique por dentro

Este artigo é útil para leitores que desejam aperfeiçoar seus conhecimentos sobre Design Patterns. Nele, mostraremos por meio de exemplos reais como a utilização dos padrões Singleton e Flyweight melhora a qualidade e favorece a manutenção e evolução do código. Além disso, analisaremos como estes padrões são empregados em recursos da plataforma Java EE, o que reforça ainda mais nossos conhecimentos para a importante decisão de quando adotá-los em nossos projetos.

facilitar o desenvolvimento de forma padronizada, conhecidas como *Design Patterns*.

Baseando-se em problemas comuns, quatro autores criaram vinte e três soluções padronizadas, catalogando-as no livro *Design Patterns: Elements of reusable object-oriented software*. Este livro tornou-se referência mundial quando o assunto é a resolução de problemas específicos em projetos de software orientados a objetos, independente da linguagem utilizada. Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, mais conhecidos como a Gangue dos Quatro (*Gang of Four - GoF*), dividiram seus respectivos padrões em três categorias: Padrões de Criação, Estruturais e Comportamentais.

Com base nas experiências destes autores, será apresentada neste artigo uma visão pragmática acerca da utilização dos *Design Patterns Singleton* e *Flyweight*, com exemplos reais aplicados nas plataformas *Java Standard Edition* e *Enterprise Edition*.

Introdução ao Singleton

Você já implementou alguma classe responsável por armazenar e gerenciar um *cache* de quaisquer dados e se deparou com a

necessidade de compartilhar uma única instância desta por toda a aplicação?

Sabemos que, ao utilizar a palavra reservada **new**, um novo objeto é criado e mantido em memória até que não existam mais referências ao mesmo, tornando-o passível de descarte pelo *Garbage Collector*.

Na maioria dos casos é comum a criação de diversos objetos de uma mesma classe, mas existem cenários em que podemos gerar, manter e compartilhar um único objeto. É neste momento que podemos nos beneficiar de um dos padrões mais conhecidos e de fácil compreensão pelos desenvolvedores, o Singleton. Criado pela Gangue dos Quatro, este design pattern faz parte da categoria dos padrões de criação e seu nome é oriundo do jogo de baralho Singleton, mais especificamente quando resta apenas uma carta na mão.

O Singleton tem como objetivos:

1. Garantir a existência de uma única instância da classe relacionada no sistema;
2. Disponibilizar um ponto de acesso central a essa instância.

Ao garantir uma única instância de uma classe, podemos evitar situações indesejáveis, como o alto consumo de memória quando precisamos de um mesmo objeto repetidas vezes. Por exemplo, ao criarmos um objeto responsável por recuperar valores de um arquivo de propriedades, podemos mantê-lo e reutilizá-lo sempre que uma propriedade for solicitada.

Implementando o Singleton

A implementação do Singleton é considerada simples, podendo ser realizada em apenas uma classe e com poucas linhas de código. Podemos visualizar, na **Listagem 1**, a estrutura mais famosa de um Singleton. O primeiro detalhe a ser observado é que a classe é detentora de sua única instância, criada e mantida no atributo estático **INSTANCE** na primeira chamada ao método **getInstance()**. A partir deste momento, qualquer chamada a este método implica que a mesma instância será retornada. Vale ressaltar ainda que, para evitar a criação de uma instância externa, o construtor deve ser definido com o modificador de acesso **private**.

No entanto, sua classe ainda corre o risco de ter várias instâncias devido aos recursos do Java de reflexão e desserialização, analisados a seguir:

- Via reflexão podemos invocar qualquer construtor privado. Assim, se for necessário se proteger deste recurso da linguagem, lance uma exceção no método construtor caso a instância já esteja criada, conforme a **Listagem 2**;
- Ao realizar a desserialização do objeto que deveria ser único, uma nova instância será criada. A fim de evitarmos a existência de duas instâncias, é necessário definir os atributos do Singleton como **transient** (isto fará com que esses atributos não sejam serializados) e implementar o método **readResolve()** (invocado pelo processo de desserialização), retornando a instância já existente em memória no atributo **INSTANCE**.

Listagem 1. Estrutura básica mais conhecida de um Singleton.

```
01. public class SuaClasseSingleton {  
02.     private static final SuaClasseSingleton INSTANCE = new SuaClasseSingleton();  
03.  
04.     private SuaClasseSingleton() {}  
05.  
06.     public static SuaClasseSingleton getInstance() {  
07.         return INSTANCE;  
08.     }  
09.  
10.    public void qualquerMetodoDeNegocio() {}  
11.}
```

Listagem 2. Estrutura básica mais conhecida de um Singleton.

```
01. public class SuaClasseSingleton {  
02.     private static final SuaClasseSingleton INSTANCE = new SuaClasseSingleton();  
03.  
04.     private SuaClasseSingleton() {  
05.         if(INSTANCE != null){  
06.             }  
07.     }  
08.  
09.     public static SuaClasseSingleton getInstance() {  
10.         return INSTANCE;  
11.     }  
12.  
13.     public void qualquerMetodoDeNegocio() {}  
14.}
```

É importante lembrar que o recurso de serialização tem acesso, inclusive, aos construtores privados, para a instanciação de novos objetos, e é justamente neste ponto que precisamos ficar atentos.

Felizmente, o mecanismo de desserialização invoca um método (não estático) da classe do objeto a ser desserializado, chamado **readResolve()**, após a desserialização. Ao implementarmos esse método, retornamos a única instância do Singleton que deve existir na aplicação e que está referenciada no atributo privado e estático **INSTANCE**. Veja um exemplo disso na **Listagem 3**.

Ao implementarmos um Singleton em seu formato tradicional, nos deparamos com alguns riscos que podem em jogo seus próprios objetivos. No entanto, esses riscos foram minimizados a partir do Java 5, com a inclusão dos *Enums*. Com este recurso tornou-se mais simples e seguro implementar este padrão (veja como fazê-lo na **Listagem 4**).

Ao criar um Singleton a partir de um **enum** não haverá mais a possibilidade de se criar outras instâncias via reflexão ou serialização de objetos, visto que *enums* não são instanciáveis, ou seja, não possuem métodos construtores.

Onde o Singleton é utilizado no universo Java EE?

Agora que temos uma boa base do padrão Singleton e suas peculiaridades, fica fácil entender sua implementação e como funciona a especificação JSR-318: Enterprise JavaBeans 3.1, que rege a implementação e o gerenciamento do ciclo de vida dos EJBs na plataforma Java Enterprise Edition 6.

Uma das novidades da especificação mencionada é o componente Singleton SessionBean, reconhecido através do uso da anotação

Vantagens e desvantagens dos padrões de projeto Singleton e Flyweight

@javax.ejb.Singleton, demonstrada na **Listagem 5**. Sua principal diferença em relação ao Stateful SessionBean é que existirá uma única instância do objeto por JVM.

Listagem 3. Singleton protegido do retorno de uma nova instância na desserialização.

```
import java.io.ObjectStreamException;
import java.io.Serializable;

public final class SuaClasseSingleton implements Serializable {
    private static final long serialVersionUID = 1L;

    private static final SuaClasseSingleton INSTANCE = new SuaClasseSingleton();

    private SuaClasseSingleton() {}

    public static SuaClasseSingleton getInstance() {
        return INSTANCE;
    }

    public void qualquerMetodoDeNegocio() {}

    /*
     * Método que será invocado pelo Java no processo de desserialização.
     * Garantimos a unicidade retornando o objeto referenciado pelo
     * atributo estático INSTANCE.
     */
    private Object readResolve() throws ObjectStreamException {
        return INSTANCE;
    }
}
```

Listagem 4. Estrutura simples e pouco conhecida de um Singleton.

```
public enum SuaClasseSingletonEnum {

    INSTANCE;

    public void qualquerMetodoDeNegocio() {}
}
```

Listagem 5. Exemplo de Singleton SessionBean.

```
import javax.ejb.Singleton;
import javax.ejb.SessionBean;
import javax.ejb.LocalBean;

@Singleton
public class SeuEJBSingleton extends SessionBean {
    public void qualquerMetodoDeNegocio() {}
}
```

Ao traçar um paralelo com a implementação do Singleton, por sua vez, é comum pensarmos que existirá apenas uma instância do Singleton SessionBean por aplicação, mesmo em um ambiente distribuído em vários nós. Porém, a especificação estabelece que caso sua aplicação esteja em execução em um ambiente clusterizado, por exemplo, com dez nós, existirão dez instâncias diferentes do Singleton SessionBean, uma em cada JVM. Dessa forma, se seu Singleton foi projetado como um serviço que mantém o estado de objetos em memória, serão dez instâncias do mesmo, uma para cada nó (JVM). Nesse momento então, temos um problema, pois cada nó mantém as informações de forma independente, gerando divergências que podem comprometer o funcionamento e a consistência da aplicação.

Vejamos um exemplo desta situação. Suponha que sua aplicação não é executada em um ambiente distribuído e seu Singleton SessionBean tenha sido modelado com o intuito de compartilhar o estado. No entanto, algum tempo depois, você decide tornar a sua aplicação distribuída, acarretando em novas instâncias do Singleton de acordo com os nós do cluster (já que existirá uma instância por JVM). Diante disso, como podemos compartilhar esse estado desses objetos entre as diversas instâncias geradas? Para solucionar esse problema, uma técnica que pode ser adotada é compartilhar o estado do Singleton através de um sistema de cache de segundo nível – com suporte a replicação de dados entre os nós do ambiente clusterizado – como, por exemplo, o Infinispan.

Além da facilidade de criação de um Singleton SessionBean, a especificação Enterprise JavaBeans 3.1 oferece outros benefícios, a saber:

- Uso da anotação **@Startup** para inicializar o Singleton junto ao carregamento da aplicação;
- Uso da anotação **@DependsOn** para definir a ordem de inicialização entre diversos Singletons;
- Uso da anotação **@ConcurrencyManagement** para especificar quem controlará o acesso concorrente ao objeto, se o próprio bean ou o container.

Essas anotações são parte da Java Enterprise Edition 6. Nessa plataforma, além do EJB 3.1, temos também o CDI (*Context and Dependency Injection*, especificado pela JSR-299), que define um comportamento diferente para a anotação **@Singleton**. Ao utilizarmos EJBs, somos beneficiados pelo controle transacional, concorrência no acesso ao objeto e distribuição do Singleton Session Bean. Porém, quando precisamos criar um objeto único mais simplificado, utilizamos o CDI, pois dessa forma o servidor de aplicação fica responsável apenas por gerenciar o ciclo de vida do Singleton, não havendo preocupação com transações, concorrência de acesso ou distribuição do objeto criado.

Introdução ao Flyweight

Assim como o Singleton, o padrão Flyweight – que em Português significa “peso-mosca”, a categoria mais leve do boxe – também faz parte do catálogo de Design Patterns do GoF, pertencendo à categoria de padrões Estruturais. O Flyweight é utilizado para compartilhar objetos otimizando o uso da memória e o desempenho em relação ao tempo de execução do seu código. Seu objetivo é estabelecer um padrão para cachear objetos muito utilizados na aplicação e facilitar a manipulação de grandes quantidades de dados.

Conceitualmente, o uso eficiente do Flyweight ocorre quando possuímos muitas instâncias de uma classe e quando parte destas instâncias podem ser substituídas por poucos objetos compartilhados.

Para entendermos melhor o funcionamento do Flyweight, vamos trazer esse conceito para um problema real: considere que precisamos modelar a construção de uma loja que vende exclusivamente

computadores. Como características mais relevantes da nossa loja, destacam-se os seguintes aspectos:

- Vendemos um único modelo de computador de uso pessoal (*desktops*), com uma especificação e preço padrão;
- A especificação e preço obedecem a um padrão único;
- Vendemos um tipo de computador que tem alto desempenho, utilizado como servidor de aplicação;
- Vendemos um computador exclusivo para ser utilizado como servidor de banco de dados;
- A venda mensal de computadores de uso pessoal é cerca de 100 vezes maior que o servidor de banco de dados.

De acordo com estes requisitos, podemos concluir que apesar de todas as entidades envolvidas serem computadores, elas podem assumir estados e configurações diferentes. Além disso, um pequeno grupo (*desktops*) pode ser representado por um único objeto imutável compartilhado. Esse, portanto, é um ótimo cenário para aplicarmos o *Flyweight*.

Com o objetivo de ilustrar o benefício do compartilhamento de objetos imutáveis, observe a **Figura 1**, que possui a representação de dois carrinhos de compras em forma de listas, contendo os objetos criados para cada tipo de computador com e sem a utilização do pattern *Flyweight*.

Considerando que nosso cliente possui em seu carrinho quatro computadores desktop, um servidor de aplicação e um servidor de banco de dados, na versão sem o Flyweight podemos observar um desperdício de recursos, pois todas as vezes em que um desktop for adicionado ao carrinho de compras será criada uma nova instância. Por sua vez, analisando a opção com o Flyweight, baseado na premissa da imutabilidade, teremos apenas uma instância criada e compartilhada todas as vezes que um computador desktop for adicionado à lista de compras.

Implementando o Flyweight

Agora que entendemos melhor os conceitos do Flyweight, nosso desafio é aplicar esse Design Pattern na implementa-

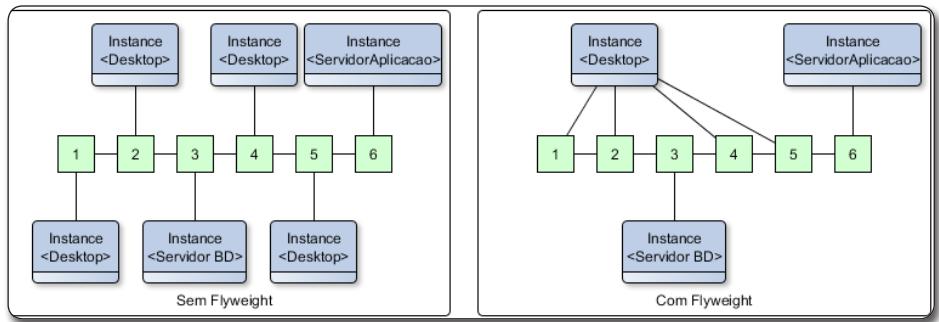


Figura 1. Carrinho de compras sem e com o Flyweight

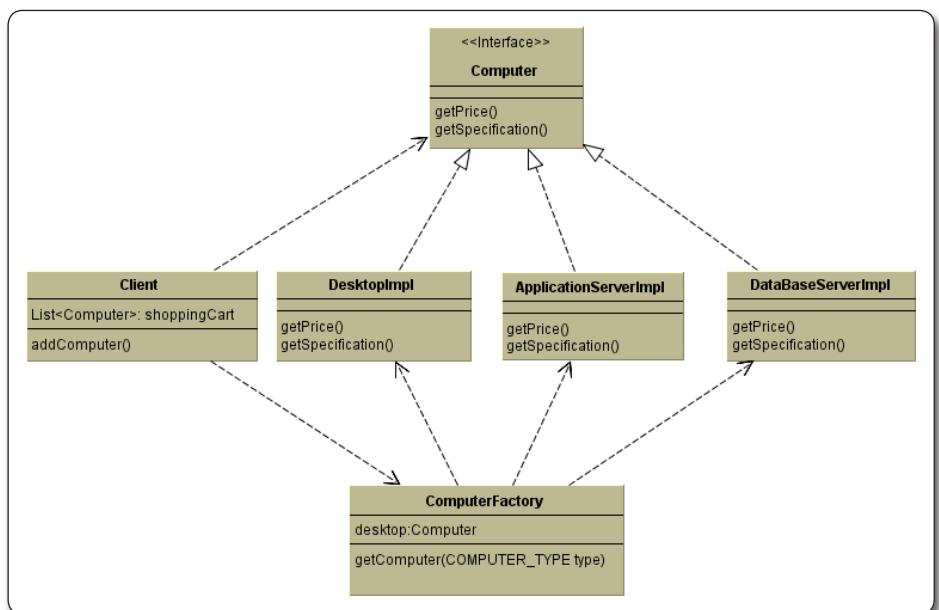


Figura 2. Diagrama de classes do exemplo

tação das principais classes da solução do nosso carrinho de compras. Para isso, utilizaremos os requisitos da nossa loja de computadores.

Como primeiro passo, para facilitar o entendimento, observe a **Figura 2**, que mostra o diagrama de classes da solução com as abstrações e as relações entre elas.

Dado esse diagrama, podemos traçar um paralelo de como nossas classes estão relacionadas ao Flyweight:

- A interface **Computer**, apresentada na **Listagem 6**, representa o contrato que todos os computares implementam, ou seja, para ser um computador no sistema uma classe deve implementar essa interface;
- A classe concreta **DesktopImpl**, apresentada na **Listagem 7**, é a abstração

que define nosso objeto do mundo real: o computador **desktop**. É esta classe que terá apenas uma instância criada e compartilhada no sistema;

- Na **Listagem 8** temos as classes **ApplicationServer** e **DataBaseServer**. Elas representam os tipos de computadores menos vendidos, ou seja, com poucas instâncias criadas em nosso carrinho de compras, não atingindo os requisitos necessários para serem um *Concret Flyweight*;
- Na **Listagem 9** temos a classe **ComputerFactory**, nossa fábrica de instâncias dos computadores e também responsável por garantir a criação e compartilhamento da única instância de **DesktopImpl**;
- Por fim, observamos na **Listagem 10** a classe **Client**, que solicita a **ComputerFactory** as instâncias dos computadores adicionados ao carrinho de compras.

Vantagens e desvantagens dos padrões de projeto Singleton e Flyweight

Listagem 6. Código da interface Computer.

```
public interface Computer {  
    float getPrice();  
    String getSpecification();  
}
```

Listagem 7. Código da classe DesktopImpl.

```
public class DesktopImpl implements Computer {  
  
    public DesktopImpl() {  
        System.out.println("Create new Desktop instance.\n");  
    }  
  
    @Override  
    public String getPrice() {  
        return 1200.00;  
    }  
  
    @Override  
    public String getSpecification() {  
        return "I'm Desktop";  
    }  
}
```

Observando a **Listagem 6**, note que criamos a interface **Computer** partindo do princípio que em nossa loja vendemos computadores e que toda abstração que representa um tipo de computador deve implementar esse contrato.

Na **Listagem 7**, a classe concreta **DesktopImpl** implementa a interface **Computer** e define a abstração que representa nosso computador desktop. Dessa classe, de acordo com a estrutura do Flyweight, teremos apenas uma instância imutável armazenada em uma variável estática que será gerenciada pela nossa fábrica de instâncias, **ComputerFactory** (vide **Listagem 10**). Essa instância, posteriormente, será compartilhada por todos os carrinhos de compras dos clientes.

Na **Listagem 8** temos duas classes que implementam a interface **Computer**, mais especificamente as classes concretas **ApplicationServer** e **DataBaseServer**. Analisando os requisitos informados, não existe a necessidade do compartilhamento de instâncias, pois a quantidade de computadores desses tipos que são vendidos é mínima e consequentemente o número de instâncias criadas não é relevante a ponto de interferir na memória.

Já na **Listagem 9** apresentamos o código de **ComputerFactory**, responsável por retornar objetos de abstrações que implementam a interface **Computer**. Essa fábrica cria as instâncias conforme o parâmetro **COMPUTER_TYPE**, podendo ser dos tipos **DESKTOP**, **APP_SERVER** ou **DB_SERVER**. Nesta classe também implementamos a criação do cache do objeto Flyweight. Esse cache nada mais é do que uma instância atribuída a uma variável estática que será retornada toda vez que um cliente solicitar um computador do tipo **DESKTOP** (**DesktopImpl**).

A implementação do Flyweight requer uma fábrica de objetos que tem como uma das suas principais metas evitar que o **Client** fique responsável pela criação dessas instâncias.

Listagem 8. Código das classes ApplicationServer e DataBaseServer.

```
public class ApplicationServer implements Computer {  
  
    public ApplicationServer () {  
        System.out.println("Create new ApplicationServer instance.\n");  
    }  
  
    @Override  
    public String getPrice() {  
        return 5200.00;  
    }  
  
    @Override  
    public String getSpecification() {  
        return "I'm ApplicationServer";  
    }  
}  
  
public class DataBaseServer implements Computer {  
  
    public DataBaseServer () {  
        System.out.println("Create new DataBaseServer instance.\n");  
    }  
  
    @Override  
    public String getPrice() {  
        return 8000.00;  
    }  
  
    @Override  
    public String getSpecification() {  
        return "I'm DataBaseServer";  
    }  
}
```

Listagem 9. Código da classe ComputerFactory.

```
public class ComputerFactory {  
  
    private static Computer desktop;  
  
    public enum COMPUTER_TYPE {  
        DESKTOP, APP_SERVER, DB_SERVER;  
    }  
  
    public static Computer getComputer(COMPUTER_TYPE type) {  
        if (type == null) {  
            throw new IllegalArgumentException();  
        }  
  
        if (COMPUTER_TYPE.DESKTOP.equals(type)) {  
            return getDesktop();  
        }  
  
        if (COMPUTER_TYPE.APP_SERVER.equals(type)) {  
            return new ApplicationServerImpl();  
        }  
  
        if (COMPUTER_TYPE.DB_SERVER.equals(type)) {  
            return new DataBaseServerImpl();  
        }  
        Return null;  
    }  
  
    private static Computer getDesktop() {  
        if (desktop == null) {  
            desktop = new DesktopImpl();  
        }  
  
        return desktop;  
    }  
}
```

Apresentada na **Listagem 10**, a classe **Client** representa nosso carrinho de compras. Nela, note que por meio da execução do método **addShoppingCart()** podemos adicionar computadores ao carrinho de compras. Sempre que esse método for executado será solicitada uma instância de computador à fábrica, que retornará um objeto de acordo com **COMPUTER_TYPE** passado como parâmetro.

Listagem 10. Código da classe Client (Client).

```
public class Client {  
  
    private Map<Quantity, Computer> shoppingCart = new HashMap<Quantity,  
    Computer>();  
  
    public static void main(String[] args) {  
        Client manager = new Client();  
        manager.addShoppingCart(4, COMPUTER_TYPE.DESKTOP);  
        manager.addShoppingCart(1, COMPUTER_TYPE.APP_SERVER);  
        manager.addShoppingCart(1, COMPUTER_TYPE.APP_SERVER);  
        manager.showShoppingCart();  
    }  
  
    private void addShoppingCart(int quantity, COMPUTER_TYPE type) {  
        Quantity quantity = new Quantity(quantity, type);  
        shoppingCart.put(quantity, ComputerFactory.getComputer(type));  
    }  
  
    private void showShoppingCart() {  
        for (Entry<Quantity, Computer> entry : shoppingCart.entrySet()) {  
            System.out.println("Quantity: " + entry.getKey().getQuantity());  
            System.out.println("Specification: " + entry.getValue().getSpecification());  
        }  
    }  
  
    class Quantity{  
  
        Integer quantity;  
        COMPUTER_TYPE computerType;  
  
        public Quantity(Integer quantity, COMPUTER_TYPE computerType) {  
            this.quantity = quantity;  
            this.computerType = computerType;  
        }  
  
        public Integer getQuantity() {  
            return quantity;  
        }  
  
        public COMPUTER_TYPE getComputerType() {  
            return computerType;  
        }  
    }  
}
```

Ao executar a classe **Client** constatamos que quando chamamos o método **getComputer()** de **ComputerFactory** para obter um novo objeto de **DesktopImpl**, nossa factory sempre retornará a mesma instância, independentemente do número de instâncias de **DesktopImpl** solicitadas por um ou vários clientes.

Onde o Flyweight é utilizado no universo Java e Java EE?

Após demonstrarmos a implementação de um exemplo com o padrão de projeto Flyweight, vamos citar alguns casos concretos onde esse pattern é aplicado no Java.

Ao observar algumas classes do JDK podemos perceber o uso desse padrão como, por exemplo, na implementação da classe **Integer**. Para reforçar o conceito desse pattern, na **Listagem 11** lançamos um pequeno desafio, onde o leitor deve analisar um trecho de código e escolher a opção correta sobre a utilização do padrão de projeto Flyweight pelo *wrapper* **Integer** do JDK.

Listagem 11. Desafio sobre a utilização do Flyweight pela classe Integer.

```
public class Desafio {  
  
    public static void main(String[] args) {  
        Integer a = Integer.valueOf(127);  
        Integer b = Integer.valueOf(127);  
  
        System.out.println(a == b);  
  
        a = Integer.valueOf(128);  
        b = Integer.valueOf(128);  
  
        System.out.println(a == b);  
    }  
}
```

Ao executar esse código, quais valores booleanos serão impressos:
a) true, true
b) false, false
c) true, false
d) false, true

Ao utilizar quaisquer métodos **valueOf()** da classe **Integer**, se o valor passado como parâmetro estiver entre -128 e 127, será utilizado o cache de instâncias previamente criadas de acordo com a recomendação do padrão Flyweight. Dessa forma, toda vez que o método **valueOf()** for executado recebendo como parâmetro um valor dentro desse intervalo, será retornada uma referência de um objeto **Integer** já criado na memória *heap* e mantido em um array de **Integer** estático dentro da classe **IntegerCache**. Portanto, a resposta correta para esse desafio é a alternativa c.

Na **Listagem 12** demonstramos trechos reduzidos da classe **IntegerCache**, com a implementação do cache dos objetos mais utilizados e imutáveis.

O padrão Flyweight tem como principal critério de utilização que os objetos a serem compartilhados sejam imutáveis. Como a classe **Integer** é imutável, ao observar o código desta classe, veremos que o atributo **value** é **final**, sendo atribuído apenas pelo construtor. No JDK, além da classe **Integer**, **String**, **Boolean**, **Byte**, **Character**, **Short** e **Long** também implementam o Flyweight.

Ainda sobre a aplicabilidade desse padrão, na plataforma Java Enterprise Edition também o encontramos. Por exemplo: quando trabalhamos com EJB 3.1 e possuímos um cliente externo que precisa utilizar um serviço EJB interno ao servidor de aplicação, como alternativa ao mecanismo de Injeção de Dependências, podemos criar um serviço de localização de EJB utilizando Flyweight.

Esse serviço de localização permite buscar e consequentemente executar serviços remotos internos ao servidor de aplicação.

Vantagens e desvantagens dos padrões de projeto Singleton e Flyweight

Listagem 12. Inner class IntegerCache da classe java.lang.Integer.

```
/*
 * Classe interna privada e estática da classe java.lang.Integer. Extraídos trechos de
 * código que representam o uso de Flyweight para reutilização de valores numéricos
 * entre -128 e 127.
 */
private static class IntegerCache {
    static final int low = -128;
    static final int high = 127;

    // +1 é para considerar o valor zero entre -128 e 127.
    static final Integer[] cache = new Integer[(high - low) + 1];

    static {
        int j = low;
        for (int k = 0; k < cache.length; k++) {
            vcache[k] = new Integer(j++);
        }
    }
}
```

Quando nos deparamos com essa situação, uma solução é implementar o padrão *Service Locator* do *Core J2EE Patterns*. Dessa forma, sempre que precisarmos de um EJB fora do servidor de aplicação, podemos solicitar ao nosso *Service Locator*. No entanto, mesmo com a criação desse serviço, a operação de localização manual pode ter um alto custo, porque envolve o acesso ao servidor de nomes do servidor de aplicação e a obtenção do objeto remoto *Proxy*. Dependendo do número de objetos e do número de vezes que esta operação for executada, podemos onerar a performance da aplicação.

Com o objetivo de melhorar o serviço de localização de um proxy, implementamos o Flyweight para armazenar e compartilhar esse objeto. Dessa forma, a localização é realizada apenas uma vez e reaproveitada todas as vezes que precisarmos utilizá-lo.

O entendimento dos padrões de projeto em sua plenitude é essencial para a eficiência da implementação quando for necessário utilizá-los. Sabendo disso, nosso principal objetivo é despertar no leitor uma visão crítica sobre dois importantes padrões: o Flyweight e o Singleton.

Neste contexto, diversas peculiaridades sobre o Singleton ainda são obscuras a muitos desenvolvedores da plataforma Java EE, principalmente quando se trata de projetos que são executados sobre um servidor em cluster. Peculiaridades estas que podem causar uma série de transtornos, tais como a criação de novas instâncias via desserialização, a não replicação automática do estado de cada instância de EJB Singleton rodando em cada nó do cluster (quando espera-se um único Singleton com estado para a aplicação) e as diferenças em relação ao EJB Singleton e o Singleton do CDI.

Quanto ao Flyweight, vale ressaltar sua eficiência em situações simples, que sugerem evitar repetições na criação de objetos e assim prover a reutilização de instâncias. Sempre que estiver em dúvida sobre quando aplicar esse pattern, lembre-se que se sua solução possui um grande número de objetos imutáveis que podem ser reutilizados, o uso do Flyweight deve ser considerado, melhorando o desempenho e proporcionando uma economia de memória.

Autor



Ednardo Luiz Martins

ednardomartins@gmail.com

Graduado em Ciências da Computação pela Universidade Federal de Ouro Preto. Em contato com Java desde 2002, trabalhou como Arquiteto Java em alguns projetos na área de Segurança Pública e Justiça. Atualmente é Analista de Sistemas Sênior em Florianópolis/SC.



Autor



Ricardo da S. Longa

ricardo.longa@gmail.com

Graduado em Sistemas de Informação e pós-graduado em Engenharia de Software pela Universidade do Sul de Santa Catarina, é um artesão de software há 11 anos, blogueiro nas horas vagas e professor de cursos técnicos, graduação e pós-graduação. Atualmente trabalha como Analista de Sistemas na Neoway Business Solutions, realiza palestras relacionadas ao Java/Android desde 2013 e é integrante/evangelista do Grupo de Usuários Java de SC.



Links:

What is CDI?

<http://cdi-spec.org/>

Página da especificação do EJB 3.1.

<https://jcp.org/en/jsr/detail?id=318>

GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. DESIGN PATTERNS: Elements of reusable object-oriented software. Addison-Wesley Longman, Inc. 1995.

KERIEVSKY, Joshua. REFACTORING TO PATTERNS. Person Education, Inc. 2005.

ALUR, Deepak; CRUPI, John; MALKS, Dan. CORE J2EE PATTERNS: Best Practices and Design Strategies. Prentice Hall / Sun Microsystems Press. 2003.

Hibernate Envers: Auditoria de dados em Java

Aprenda como implementar a auditoria de dados com o Hibernate Envers, um requisito tão solicitado pelas corporações

Auditoria pode ser considerada como um exame sistemático das atividades desenvolvidas em determinada empresa ou setor, tendo como objetivo averiguar se essas atividades estão de acordo com as diretrizes planejadas e/ou estabelecidas previamente, se foram implementadas com eficácia e se estão adequadas.

Com o surgimento dos sistemas computacionais, as empresas passaram a confiar seus dados à área de Tecnologia da Informação (TI), que se tornou responsável pela proteção e garantia da consistência desses dados.

Como essas informações, na maioria das vezes, ficam armazenadas em bancos de dados, ele é o ponto de partida para iniciar uma auditoria ou conferência, tendo como insumos os dados recebidos e enviados pelas aplicações e sistemas corporativos. Dessa forma, auditar a persistência requer uma análise direta de operações que envolvem o acesso a bancos de dados.

Uma das funções da auditoria é monitorar quando e como o dado foi inserido, com o intuito de prevenir e detectar problemas no cumprimento das regras de negócio. Outra função tem relação com questões de segurança, objetivando garantir que usuários não autorizados não estejam acessando o banco de dados.

Uma boa auditoria de persistência deve fornecer informações sobre as operações realizadas nas tabelas envolvidas, como: quem inseriu, editou, excluiu ou até mesmo consultou informações; que informações foram essas; como estavam antes e como ficaram depois; quando foi feito, etc. Com estas informações e caso algum dado seja inserido/alterado incorretamente, o analista terá meios para que possa recuperar a versão antiga da informação que deseja ou mesmo identificar o autor daquela alteração.

Fique por dentro

Este artigo é útil por mostrar como rastrear em um banco de dados o histórico de mudanças de entidades para saber quem, quando e o que foi alterado. Aprenderemos de forma básica e simples os principais conceitos relativos ao Envers, fazendo uma análise geral sobre seu funcionamento, mostrando como configurar um ambiente de desenvolvimento com Maven, Eclipse e MySQL para uso desta tecnologia e, em seguida, como implementar um sistema de auditoria.

Um tipo de auditoria muito usado nos sistemas atuais é o baseado em *Stored Procedures* e *Triggers*, realizado diretamente no SGBD. O inconveniente desse método está na necessidade de se desenvolver uma estrutura específica para as auditorias diretamente no banco de dados, gerando assim uma alta dependência desse componente.

Independentemente do porte da aplicação, é cada vez mais comum a necessidade real de monitorar as ações do usuário frente ao sistema, de modo a prover controle sobre seu uso, assim como resguardar a integridade das informações administradas pela mesma. Em torno desta necessidade, hoje, grande parte das aplicações desenvolvidas sobre a plataforma Java faz uso de frameworks ORM, sendo o Hibernate o mais adotado. E foi através dessa necessidade que foi criado o Envers.

Portanto, ao longo desse artigo abordaremos os principais conceitos e características do Hibernate Envers, e veremos como desenvolver um sistema de cadastro de projetos considerando como um dos seus principais requisitos não funcionais a realização de auditoria sobre seus dados. Para tanto, serão empregados, além do Envers, a linguagem de programação Java, o ambiente de desenvolvimento Eclipse integrado ao Maven e o sistema de gerenciamento de banco de dados MySQL. O Hibernate será usado como solução integrante da camada de persistência, viabilizando a interface entre a aplicação e o MySQL.

Hibernate Envers

O Hibernate é um framework de mapeamento objeto relacional muito popular entre os desenvolvedores Java. Distribuído com a licença LGPL, foi criado por Gavin King em 2001, sendo, sem dúvida, o framework de persistência de dados mais utilizado, sobretudo por dar suporte a classes codificadas com agregações (aqueles que usam outras classes em suas operações), herança, polimorfismo, composições e coleções, por implementar a especificação Java Persistense API, por não restringir a arquitetura da aplicação e por ser amplamente documentado. Segundo a documentação oficial: “o Hibernate pretende retirar do desenvolvedor cerca de 95% das tarefas mais comuns de persistência de dados”.

Sua principal característica é o mapeamento de classes Java em tabelas da base de dados (e dos tipos de dados Java para os da SQL). Além disso, o Hibernate gera os comandos SQL e libera o desenvolvedor do trabalho manual de transformação, mantendo o programa portável para quaisquer bancos de dados SQL.

O Envers é um projeto open source criado pelo engenheiro de software Adam Warski e hoje é conduzido pela empresa JBoss. Essa biblioteca permite a auditoria de classes persistentes (ORM) de forma simples e fácil, utilizando-se do conceito de revisões, de modo similar ao Subversion, onde cada transação é uma revisão. A motivação para a utilização desse recurso é poder auditar completamente os dados, ou seja, rastrear o histórico de mudanças e ter as informações de quem, quando e o que foi modificado.

A grande inovação dessa ferramenta é a geração e alimentação automática de tabelas idênticas às tabelas originais, permitindo o controle de versões do conteúdo das tabelas mapeadas. Essas tabelas geradas podem ser chamadas de versionadas ou de histórico.

Cada vez que uma tabela auditável sofre uma alteração em seus registros é criada uma revisão, que por sua vez representa um registro na tabela versionada contendo um atributo de revisão como chave primária, o tipo de operação realizada e todos os campos auditáveis da tabela que sofre auditoria. Em suma, para cada UPDATE realizado em uma tabela, um INSERT é feito na tabela versionada.

Como vantagens da utilização do Hibernate Envers podemos citar: poucas alterações no código fonte da aplicação, sendo necessário apenas a inclusão de algumas anotações; independência de fornecedor para o banco de dados; agregação de valor ao produto entregue ao cliente; ganho de produtividade; e pode ser adotado em qualquer ambiente computacional em que o Hibernate Core funcione.

Como veremos, o uso do Envers é simples e basicamente resume-se a incluir a anotação **@Audited** no código da classe mapeada. Feito isso, o framework criará uma tabela com o histórico das mudanças dos dados da entidade. Caso o desenvolvedor necessite que algum dos campos não seja incluído no histórico, basta anotá-lo com **@NotAudited**. Feito isso, se um campo não auditado sofrer uma mudança, nenhuma entrada será inserida na tabela de auditoria.

Nota

Devido a sua importância, a partir da versão 3.5 do Hibernate o Envers passou a fazer parte do projeto core do Hibernate.

Configuração

Após conhecer as principais informações relacionadas ao Envers, vamos partir para a parte prática e desenvolver uma aplicação que possibilite ao leitor visualizar como o framework funciona e sua utilidade.

No entanto, antes de começarmos a codificar, é importante instalar os softwares que nos auxiliarão nesse trabalho e também preparar o ambiente de desenvolvimento para utilização do Hibernate Envers.

Preparando o ambiente de desenvolvimento

Neste artigo optamos por utilizar o Eclipse, pois se trata de um ambiente de desenvolvimento extremamente poderoso e flexível, sendo o mais utilizado no mercado. Na seção **Links** você encontra o endereço para download desta IDE. Neste artigo foi adotada a versão Luna SR2 (4.4.2).

O processo de instalação do Eclipse é bastante simples, sendo necessário apenas ter o JDK 8 instalado no sistema. Ainda na seção **Links**, está disponível o endereço onde pode ser obtido este JDK.

Após baixar o arquivo adequado ao seu sistema, descompacte-o no local de sua preferência. Optamos pela pasta *C:\Eclipse_Luna*. Logo após, basta executar o arquivo *eclipse.exe* (em ambientes Windows).

Integrando o Maven ao Eclipse

O Maven é uma ferramenta pertencente à Fundação Apache que tem como finalidade o gerenciamento e automação de construção (build) de projetos Java. De acordo com o próprio site do Maven, a ferramenta tem os seguintes objetivos: prover um padrão para o desenvolvimento de aplicações; criar mecanismos para uma clara definição da estrutura do projeto; controlar versão e artefatos; e possibilitar o compartilhamento de bibliotecas entre vários projetos. Como grande destaque, a principal facilidade oferecida pelo Maven é o gerenciamento de dependências e este será o nosso motivador para utilizá-lo em parceria com a IDE Eclipse.

Na seção **Links** você encontrará o endereço para download desta ferramenta. Aqui, adotamos a versão 3.3.1. Após o download, crie uma pasta de nome Maven e copie o arquivo baixado para dentro dela. Por fim, descompacte esse arquivo.

Para integrar o Maven ao Eclipse, recomenda-se utilizar o plugin M2E. No entanto, a versão do IDE que optamos por utilizar já vem com este plugin e com uma instalação “embocada” do Maven.

Para visualizarmos esta instalação, com o Eclipse aberto, acesse o menu *Window > Preferences* e escolha a opção *Maven > Installations*, conforme a **Figura 1**. Perceba a presença da versão “embocada”. Porém, vamos utilizar a versão do Maven que baixamos, que é uma versão mais atual do software.

Para adicionar o Maven ao Eclipse, ainda na **Figura 1**, clique em *Add* e selecione a pasta com a nossa instalação, no caso: *C:\ Maven\apache-maven*. Feito isso, automaticamente o arquivo de configuração do Maven é localizado, como pode ser visto na **Figura 2**, e esta versão é adicionada ao IDE. Por fim, clique em *Ok*.

O sistema gerenciador de banco de dados

O banco de dados que utilizaremos será o MySQL, uma opção gratuita, de código aberto, simples e muito adotado por desenvolvedores Java.

Entre suas características, podemos citar: excelente desempenho e estabilidade; suporte a praticamente qualquer sistema operacional (portabilidade); e interface gráfica com boa usabilidade. Na seção **Links** você encontrará o endereço para download e mais informações. A versão adotada neste artigo foi a Community Server 5.6.22.

Após o download do instalador do MySQL, os seguintes passos devem ser executados para instalar o banco de dados no sistema:

1. Clique duas vezes sobre o instalador. Feito isso, uma nova janela será exibida solicitando que o usuário aguarde enquanto o sistema operacional configura o instalador do SGBD;
2. Em seguida, uma tela exibe os termos do contrato de licença do produto. Neste ponto, aceite os termos e clique em *Next*;
3. O próximo passo é escolher o tipo de instalação. Abaixo de cada tipo existe uma breve explicação a respeito. O usuário deve escolher a opção que melhor atenda aos seus propósitos. Aqui, optamos pela *Custom*. Logo após, clique em *Next*;
4. A próxima tela oferece ao usuário a possibilidade de escolher os produtos que deseja instalar. Escolha *MySQL Server 5.6.21* e clique em *Next*;
5. Em seguida é possível verificar os produtos selecionados na tela anterior. Clique em *Execute* para dar sequência à instalação;
6. Após a instalação ser concluída, a mensagem presente na coluna *Status* da tela de instalação mudará de *Ready to Download* para *Complete*. Selecione *Next*;
7. Finalizada a instalação, o próximo passo é configurar o MySQL Server. Sendo assim, clique mais uma vez em *Next*;
8. Nesse momento o usuário deve escolher o tipo de configuração para o MySQL Server. Em nosso estudo, apenas mantenha os valores padrão e clique em *Next*;
9. A tela seguinte (veja a **Figura 3**) pede para definir a senha do administrador. Portanto, informe uma senha e clique em *Next*;

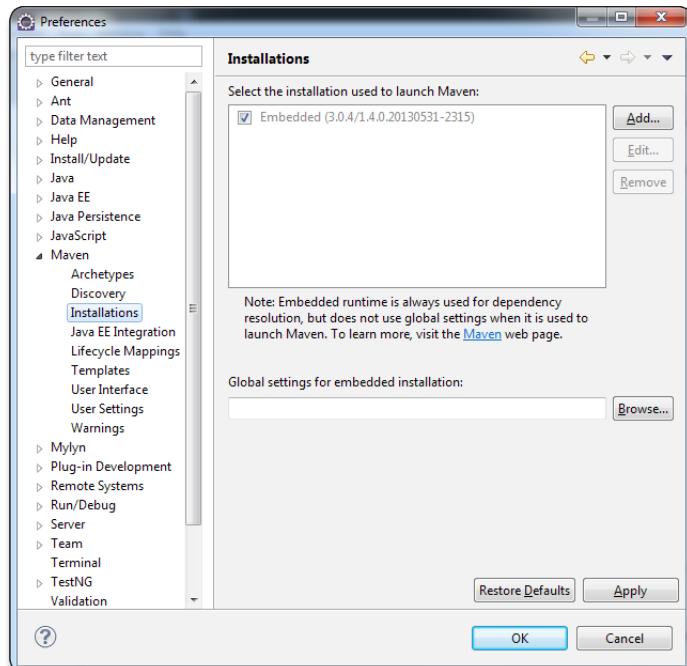


Figura 1. Instalação do Maven fornecida pelo Eclipse

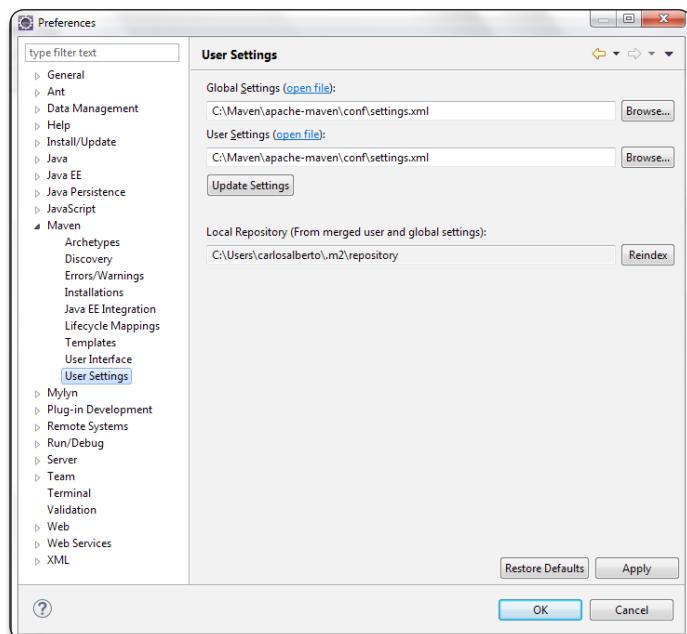


Figura 2. Adicionando o Maven ao Eclipse

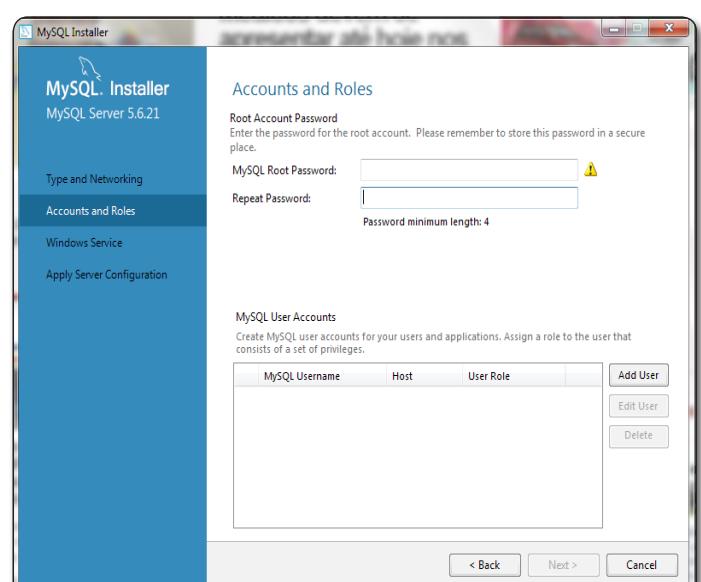


Figura 3. Definindo a senha do administrador

10. Feito isso, a próxima tela permite ao usuário configurar o MySQL como um serviço do Sistema Operacional Windows. Para tanto, basta manter os valores *default* e clicar em *Next*;
11. A janela que aparecerá detalha todos os passos de configuração que serão aplicados. Confirmadas estas escolhas, clique em *Execute*, como pode ser visto na **Figura 4**;
12. Neste momento, uma tela mostrando o fim do processo será exibida. Clique então em *Finish*;
13. Então, uma nova janela informa que a configuração do MySQL Server foi realizada (**Figura 5**). Clique pela última vez em *Next*;
14. Para encerrar, uma janela é exibida indicando que a instalação foi concluída. Neste momento, apenas clique em *Finish*.

Desenvolvendo o cadastro de funcionários

Agora que temos o banco de dados instalado, assim como o Eclipse e o Maven integrados, vamos partir para a construção de

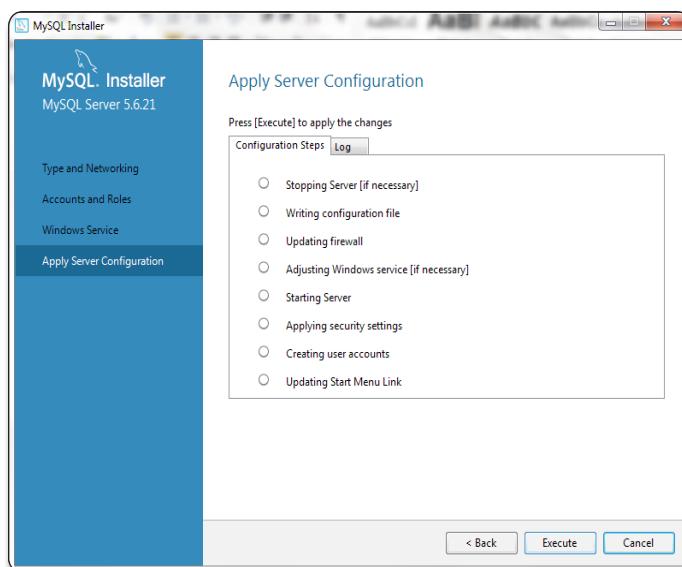


Figura 4. Aplicando as configurações definidas no Servidor

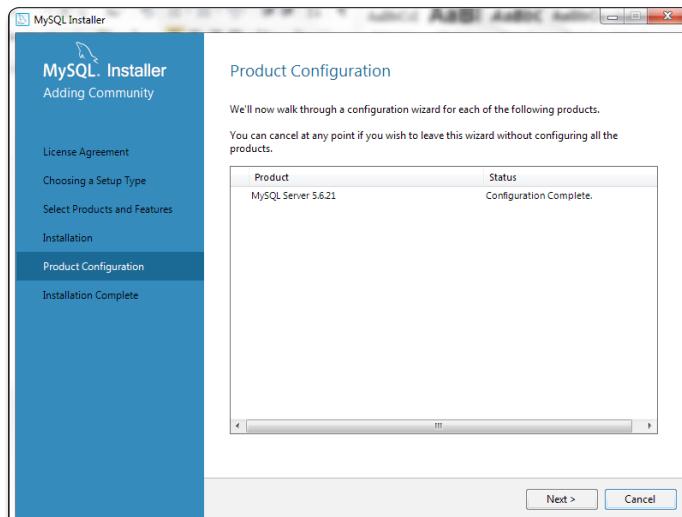


Figura 5. Configuração do MySQL finalizada

uma aplicação desktop. O objetivo desta será gerenciar o cadastro de projetos de uma empresa.

O sistema que construiremos conterá apenas três entidades (**Funcionário**, **Projeto** e **Trabalha**) e a partir delas serão construídas as principais operações realizadas sobre um cadastro, como salvar, atualizar, listar e excluir.

As informações que cada funcionário carregará são: código, nome, CPF, cargo, endereço e salário. Já um projeto conterá: um código, nome, data de início e data de término.

A entidade **Trabalha** é resultado da relação muitos para muitos existente entre **Funcionário** e **Projeto** e armazenará o código de ambos. Saiba que não é necessário criar uma classe mapeada para essa entidade, uma vez que o próprio Hibernate gera uma tabela de ligação para viabilizar esse relacionamento.

A **Figura 6** mostra o relacionamento existente (diagrama entidade-de-relacionamento) entre as tabelas do banco de dados.

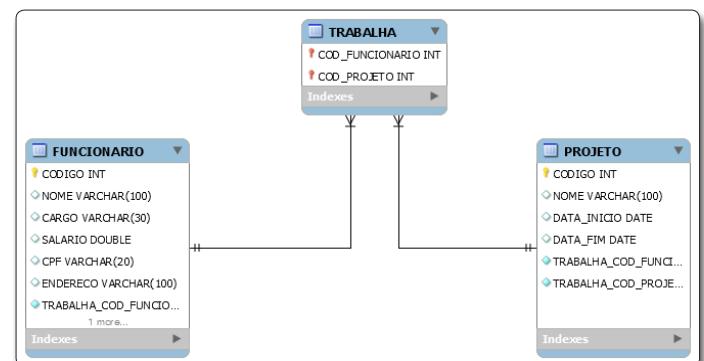


Figura 6. Diagrama Entidade Relacionamento

Criando o banco de dados

Com o MySQL instalado, abra-o e crie o banco *empresabd* executando o script SQL apresentado na **Listagem 1**. Os comandos das linhas 1 e 2 possibilitam, respectivamente, criar o banco de dados e entrar no contexto do banco criado. As tabelas e seus respectivos atributos são criados nas linhas seguintes, sendo que a tabela *trabalha* é resultado da relação muitos para muitos, existente entre as tabelas funcionário e projeto.

Criando o projeto Maven no Eclipse

Com o banco de dados pronto, vamos partir para o projeto Java, criando um novo a partir do Maven no Eclipse. Como utilizaremos anotações JPA, é importante que o projeto adote o Java 1.5 ou superior. Dito isso, com o Eclipse aberto, clique em *File > New Project* e, na tela que aparecer, selecione a opção *Create a simple project (skip archetype selection)* e clique em *Next*.

Na tela seguinte, vamos identificar o projeto preenchendo as seguintes opções:

- Group ID (identificador da empresa/grupo ao qual o projeto pertence): **br.com.devmedia**;
- Artifact ID (nome do projeto): **hibernate-envers**;
- Packaging (forma como o projeto deverá ser empacotado): **jar**;
- Version (versão do projeto): **0.0.1-SNAPSHOT**.

Listagem 1. Script para criação do banco de dados.

```
01. CREATE database empresabd;
02. USE empresabd;
03.
04. CREATE TABLE funcionario (
05.     codigo INT NOT NULL AUTO_INCREMENT,
06.     nome VARCHAR(100),
07.     cargo VARCHAR(30),
08.     salario DOUBLE,
09.     cpf VARCHAR(20),
10.    endereco VARCHAR (100),
11.    PRIMARY KEY (codigo)
12. );
13.
14. CREATE TABLE projeto (
15.     codigo INT NOT NULL AUTO_INCREMENT,
16.     nome VARCHAR(100),
17.     data_inicio DATE,
18.     data_fim DATE,
19.     PRIMARY KEY (codigo)
20. );
21.
22. CREATE TABLE trabalha (
23.     cod_funcionario INT,
24.     cod_projeto INT,
25.     PRIMARY KEY (cod_funcionario, cod_projeto),
26.     CONSTRAINT FK_PROJETO FOREIGN KEY (cod_projeto)
27.         REFERENCES PROJETO (codigo),
28.     CONSTRAINT FK_FUNCIONARIO FOREIGN KEY (cod_funcionario)
29.         REFERENCES FUNCIONARIO (codigo)
30. );
```

Após preencher todos os dados mencionados, clique em *Finish*. Note que a estrutura que criamos ainda não está definida com as configurações de um projeto Java. Sendo assim, clique com o botão direito sobre o projeto e acesse o menu *Maven > Update Project*. Na tela que aparecer, selecione o projeto que acabamos de criar e clique em *Ok* para atualizá-lo de acordo com as configurações do Maven. Feito isso, o projeto passará a ter a estrutura característica do Maven, como demonstra **Figura 7**.

Essa estrutura pode ser descrita da seguinte maneira:

- *src/main/java*: pasta onde ficam os pacotes e classes Java;
- *src/main/resources*: pasta onde ficam os arquivos de propriedades (*persistence.xml*, configuração de logs, etc.);
- *src/test/java*: pasta onde ficam os arquivos de teste unitários;
- *src/test/resources*: pasta onde ficam os arquivos de propriedades que são usados para os testes;
- *pom.xml*: esse é o arquivo onde ficam todas as configurações pertinentes à geração de build, testes, bibliotecas de terceiros, etc.

Adicionando as dependências

Com o projeto pronto, o próximo passo é inserir as dependências, isto é, adicionar as bibliotecas que serão utilizadas pela aplicação no arquivo *pom.xml*. Para isso, clique com o botão direto neste arquivo e escolha a opção *Maven > Add Dependency*. Na tela que aparecer, digite o nome das bibliotecas que deseja adicionar, sendo elas: Hibernate Core, Hibernate Envers e o driver do MySQL. Em seguida, clique em *Ok*.

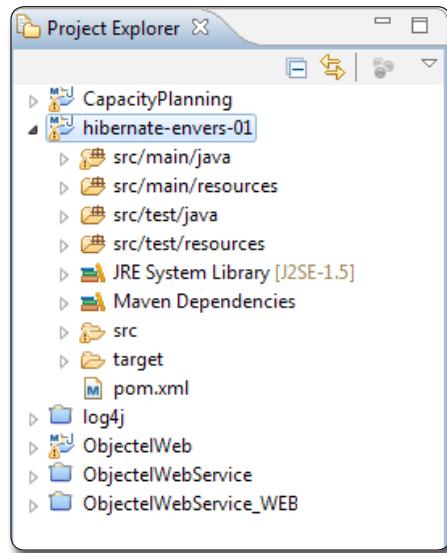


Figura 7. Estrutura do projeto Maven

O arquivo *pom.xml* ficará conforme o código apresentado na **Listagem 2**.

Inseridas as dependências, clique com o botão direito do mouse sobre o projeto e selecione o menu *Maven > Update Project*. Com isso, todas as bibliotecas definidas serão adicionadas ao projeto.

Construindo as entidades

Nosso próximo passo é criar, mapear as entidades e definir aquelas que queremos que sejam auditadas. Tudo isso lançando mão de *annotations*.

Para tanto, primeiramente vamos criar a classe **Funcionario**. Esta é uma das classes de negócio da aplicação e será mapeada pelo Hibernate como uma tabela no banco de dados. Classes desse tipo são classes Java simples, definidas como POJOs. Elas contêm todos os seus atributos encapsulados através dos métodos de acesso get e set e também disponibilizam o construtor padrão.

Em nosso exemplo, as classes de negócio ficarão dentro do pacote **com.jm.entidade**. Para criá-lo, clique com o botão direito do mouse sobre o projeto *hibernate-envers-01*, selecione *New > Package* e informe seu nome. Com o pacote criado, clique com o botão direito do mouse sobre ele, escolha a opção *New > Class* e, na tela que aparecer, dê o nome **Funcionario** à classe, cujo código é apresentado na **Listagem 3**.

Na linha 5 podemos verificar a anotação **@Entity**, principal anotação do JPA. Ela aparece antes do nome da classe e sinaliza que haverá uma tabela relacionada a essa classe no banco de dados e que os objetos desta serão persistidos.

Já a anotação **@id**, vista na linha 12, é utilizada para indicar qual atributo de uma classe anotada com **@Entity** será mapeado como a chave primária da tabela correspondente à classe.

Uma anotação que geralmente acompanha **@id** é a **@GeneratedValue**, presente na linha 14. Esta serve para indicar que o valor do atributo que compõe a chave primária deve ser gerado pelo banco no momento em que um novo registro for inserido

Hibernate Envers: Auditoria de dados em Java

Listagem 2. Configuração do arquivo pom.xml.

```
01. <project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
02. <modelVersion>4.0.0</modelVersion>
03. <groupId>br.com.devmedia</groupId>
04. <artifactId>hibernate-envers-01</artifactId>
05. <version>0.0.1-SNAPSHOT</version>
06. <dependencies>
07.   <dependency>
08.     <groupId>mysql</groupId>
09.     <artifactId>mysql-connector-java</artifactId>
10.    <version>5.1.34</version>
11.  </dependency>
12.  <dependency>
13.    <groupId>org.hibernate</groupId>
14.    <artifactId>hibernate-core</artifactId>
15.    <version>4.3.8.Final</version>
16.  </dependency>
17.  <dependency>
18.    <groupId>org.hibernate</groupId>
19.    <artifactId>hibernate-envers</artifactId>
20.    <version>4.3.8.Final</version>
21.  </dependency>
22. </dependencies>
23. </project>
```

Listagem 3. Código da classe Funcionario.

```
01. package com.jm.entidade;
02.
03. //imports omitidos...
04.
05. @Entity
06. @Audited
07. @Table
08. public class Funcionario implements Serializable {
09.
10.   private static final long serialVersionUID = 1L;
11.
12.   @Id
13.   @Column
14.   @GeneratedValue(strategy = GenerationType.AUTO)
15.   private int codigo;
16.
17.   @Column
18.   private String nome;
19.   @Column
20.   private String cargo;
21.   @Column
22.   private double salario;
23.   @Column
24.   private String cpf;
25.   @Column
26.   private String endereco;
27.
28.   @ManyToMany(mappedBy = "funcionarios", fetch=FetchType.LAZY)
29.   private List<Projeto> projetos;
30.
31. //construtores e métodos gets e sets omitidos
32. }
```

Ainda podemos utilizar as anotações `@Column` e `@Table`, que são usadas para personalizar o nome das tabelas e das colunas, bastando informar o nome entre parênteses. No nosso caso, como não adicionamos essa informação, fica considerado que o nome das tabelas e colunas será exatamente igual ao nome das classes e suas propriedades.

A anotação que indica que a entidade criada passará por auditoria pode ser vista na linha 6. Trata-se da anotação `@Audited`, pertencente ao pacote `org.hibernate.envers`. A presença dela faz com que outra tabela seja criada no banco, sendo utilizada para armazenar todas as alterações sofridas pela entidade auditada.

O nome da tabela criada automaticamente pode ser customizado através da anotação `@AuditTable`. Quando não se utiliza essa anotação, o Envers entende que a nomenclatura da tabela de histórico deve ter o nome da classe com o sufixo `_AUD`.

Agora vamos criar a outra classe, que mapeará a entidade **Projeto**. Os passos são idênticos aos executados para criar a classe **Funcionario**. Na **Listagem 4** é possível visualizar seu código.

Assim como na classe **Funcionario**, perceba na linha 6 a presença da anotação `@Audited`, indicando que a entidade em questão sofrerá auditoria. Portanto, uma tabela de histórico de nome `projeto_AUD` será criada no banco de dados para armazenar todas as alterações ocorridas nessa entidade.

Listagem 4. Código da classe Projeto.

```
01. package com.jm.entidade;
02.
03. //imports omitidos...
04.
05. @Entity
06. @Audited
07. @Table
08. public class Projeto implements Serializable {
09.
10.   private static final long serialVersionUID = 1L;
11.
12.   @Id
13.   @Column
14.   @GeneratedValue(strategy = GenerationType.AUTO)
15.   private int codigo;
16.
17.   @NotAudited
18.   @Column
19.   private String nome;
20.   @Column
21.   private Date data_Inicio;
22.   @Column
23.   private Date data_Fim;
24.
25.   @ManyToMany(cascade = {CascadeType.ALL}, fetch=FetchType.EAGER)
26.   @JoinTable(name="trabalha",joinColumns=
27.   {@JoinColumn(name="cod_projeto")}, inverseJoinColumns=
28.   {@JoinColumn(name="cod_funcionario")})
29.   private List<Funcionario> funcionarios;
30.
31. //construtores e métodos gets e sets omitidos
32. }
```

Configurando a aplicação

Nesse momento, além de informar ao Hibernate onde está nosso banco de dados e como se conectar a ele, precisamos registrar os *event listeners* que permitirão ao Envers checar se alguma entidade auditada foi modificada.

Os *event listeners*, assim como as configurações do Hibernate, devem ser adicionados em um arquivo XML. Sendo assim, crie o arquivo *hibernate.cfg.xml*. Para isso, clique com o botão direito sobre o projeto, selecione *Novo > Documento XML* e o chame de *hibernate.cfg.xml*. Seu conteúdo deve ser semelhante ao da **Listagem 5**.

Listagem 5. Conteúdo do arquivo hibernate.cfg.xml.

```
01. <?xml version="1.0" encoding="UTF-8"?>
02. <!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate
   Configuration DTD 3.0//EN" "http://hibernate.sourceforge.net/
   hibernate-configuration-3.0.dtd">
03. <hibernate-configuration>
04.   <session-factory>
05.     <property name="hibernate.dialect">org.hibernate.dialect.MySQL
        Dialect</property>
06.     <property name="hibernate.connection.driver_class">
        com.mysql.jdbc.Driver</property>
07.     <property name="hibernate.connection.url">
        jdbc:mysql://localhost:3306/empresabd?zeroDateTimeBehavior=
        convertToNull</property>
08.     <property name="hibernate.connection.username">root</property>
09.     <property name="hibernate.connection.password">1234</property>
10.     <property name="hibernate.show_sql">true</property>
11.     <mapping class="com.jm.entidade.Funcioario"/>
12.     <mapping class="com.jm.entidade.Projeto"/>
13.     <listener class="org.hibernate.envers.event.AuditEventListener"
        type="post-insert"/>
14.     <listener class="org.hibernate.envers.event.AuditEventListener"
        type="post-update"/>
15.     <listener class="org.hibernate.envers.event.AuditEventListener"
        type="post-delete"/>
16.     <listener class="org.hibernate.envers.event.AuditEventListener"
        type="pre-collection-update"/>
17.     <listener class="org.hibernate.envers.event.AuditEventListener"
        type="pre-collection-remove"/>
18.     <listener class="org.hibernate.envers.event.AuditEventListener"
        type="post-collection-recreate"/>
19.   </session-factory>
20. </hibernate-configuration>
```

Este arquivo começa com a definição do DTD (*Document Type Definition*) e na linha 3 temos o elemento raiz, **<hibernate-configuration>**. Logo após, encontramos o elemento **<session-factory>**, na linha 4, local onde se inicia a configuração da conexão com o banco de dados e também são adicionadas as informações de mapeamento.

As propriedades configuradas são apresentadas a seguir:

- **dialect (linha 5):** especifica o dialeto com o qual o Hibernate se comunicará com a base de dados, isto é, informa ao Hibernate quais comandos SQL gerar diante de um banco de dados relacional específico;
- **connection.driver_class (linha 6):** determina o nome da classe do driver JDBC utilizado. No nosso caso, trata-se do driver para o MySQL;

- **connection.url (linha 7):** especifica a URL de conexão com o banco de dados;
- **connection.username (linha 8):** refere-se ao nome de usuário com o qual o Hibernate deve se conectar ao banco de dados;
- **connection.password (linha 9):** especifica a senha do usuário com o qual o Hibernate deve se conectar ao banco;
- **show_sql (linha 10):** flag que permite tornar visível no console o SQL que o Hibernate está gerando. Assim, o desenvolvedor pode copiar e colar os comandos no banco de dados para verificar se está tudo conforme o esperado;
- **mapping (linha 11):** informa as classes que estão mapeadas para realizar operações no banco.

A partir da linha 13 é possível visualizar os *event listeners*. Estes permitem ao Envers trabalhar em conjunto com o Hibernate e são responsáveis pelas ações de auditoria, ou seja, têm a função de incluir registros nas tabelas de histórico de acordo com a ação realizada pelo usuário na aplicação.

Criando a conexão com o banco de dados

Configurada a aplicação, vamos criar agora a classe auxiliar **HibernateUtil**, que será responsável por entregar uma instância de **SessionFactory** à aplicação.

Com o intuito de manter a organização do projeto, esta classe será criada no pacote **com.jm.util** e seu código pode ser visto na **Listagem 6**.

Listagem 6. Código da classe HibernateUtil.

```
01. package util;
02.
03. //imports omitidos
04.
05. public class HibernateUtil {
06.
07.   private static SessionFactory sessionFactory;
08.
09.   public static SessionFactory getSessionFactory() {
10.     if (sessionFactory == null) {
11.       Configuration configuration = new Configuration().configure();
12.       ServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder()
13.           .applySettings(configuration.getProperties()).build();
14.
15.       sessionFactory = configuration.buildSessionFactory(serviceRegistry);
16.       SchemaUpdate se = new SchemaUpdate(configuration);
17.       se.execute(true, true);
18.     }
19.
20.     return sessionFactory;
21.   }
22.
23. }
```

A **SessionFactory** é uma classe de infraestrutura do Hibernate que implementa o *design pattern Abstract Factory* para construir instâncias de objetos do tipo **org.hibernate.Session**. Essas instâncias são usadas para realizar as tarefas de persistência do framework, sendo que cada instância de **SessionFactory** possui um estado interno imutável, ou seja, uma vez criada, os seus

atributos não serão modificados. Este estado interno inclui o metadata referente ao mapeamento objeto relacional das entidades do sistema e também a referência de quem irá fornecer as conexões com o banco de dados. Por esse motivo, a classe **HibernateUtil** utiliza-se do método estático **getSessionFactory()**, visto na linha 9, para oferecer sempre a mesma instância de um **SessionFactory** no contexto da aplicação.

Por fim, na linha 15, temos o método privado estático **buildSessionFactory()**, que é responsável por construir a instância de **SessionFactory**. Essa construção é feita com base no arquivo de configuração do Hibernate.

Persistindo dados no banco

Agora, criaremos as classes que serão responsáveis por viabilizar todas as operações de persistência disponibilizadas pela aplicação. Para isso, criemos primeiramente o pacote com.

jm.acessobd clicando com o botão direito do mouse sobre o projeto *hibernate-envers-01*, escolhendo *New > Package* e informando “com.jm.acessobd” como nome.

Criado o pacote, clique sobre ele com o botão direito, escolha *New > Class* e dê o nome à classe de **FuncionarioDAO**.

O código fonte dessa classe deve ficar semelhante ao apresentado na **Listagem 7**.

Na linha 9, dentro do construtor da classe **FuncionarioDAO**, o método **getSessionFactory()** é invocado para a criação de um **SessionFactory**. Esse objeto deve ser criado uma única vez, pois, por armazenar os mapeamentos e configurações do Hibernate, é muito pesado e lento de se criar.

Já na linha 12, o método **adicionarFuncionario()** recebe como parâmetro os dados do funcionário e realiza a inserção deste no banco de dados. Para isso, na linha 13 é obtida uma sessão a partir de um **SessionFactory**.

Listagem 7. Código da classe FuncionarioDAO. Disponibiliza todas as operações que serão realizadas no banco de dados.

```
01. package com.jm.acessobd;
02.
03. //imports omitidos...
04.
05. public class FuncionarioDAO {
06.     private static SessionFactory factory;
07.
08.     public FuncionarioDAO() {
09.         factory = HibernateUtil.getSessionFactory();
10.     }
11.
12.     public Integer adicionarFuncionario(Funcionario funcionario) {
13.         Session session = factory.openSession();
14.         Transaction tx = null;
15.         Integer cod_func = null;
16.         try {
17.             tx = session.beginTransaction();
18.             cod_func = (Integer) session.save(funcionario);
19.             tx.commit(); //executa o commit
20.         } catch (HibernateException e) {
21.             if (tx != null) {
22.                 tx.rollback();
23.             }
24.             e.printStackTrace();
25.         } finally {
26.             session.close();
27.         }
28.         return cod_func;
29.     }
30.
31.     public List<Funcionario> listarFuncionarios() {
32.         Session session = factory.openSession();
33.         Transaction tx = null;
34.         List<Funcionario> funcionarios = null;
35.         try {
36.             tx = session.beginTransaction();
37.             funcionarios = session.createCriteria(Funcionario.class).list();
38.             tx.commit();
39.         } catch (HibernateException e) {
40.             if (tx != null) {
41.                 tx.rollback();
42.             }
43.             e.printStackTrace();
44.         } finally {
```



```
45.             session.close();
46.         }
47.         return funcionarios;
48.     }
49.
50.     public void atualizarFuncionario(Integer codigoFuncionario, int salario) {
51.         Session session = factory.openSession();
52.         Transaction tx = null;
53.         try {
54.             tx = session.beginTransaction();
55.             Funcionario funcionario = (Funcionario) session.get(
56.                 Funcionario.class, codigoFuncionario);
57.             funcionario.setSalario(salario);
58.             session.update(funcionario);
59.             tx.commit();
60.         } catch (HibernateException e) {
61.             if (tx != null) {
62.                 tx.rollback();
63.             }
64.             e.printStackTrace();
65.         } finally {
66.             session.close();
67.         }
68.
69.     public void apagarFuncionario(Integer codigoFuncionario) {
70.         Session session = factory.openSession();
71.         Transaction tx = null;
72.         try {
73.             tx = session.beginTransaction();
74.             Funcionario funcionario = (Funcionario) session.get(Funcionario.class,
75.                 codigoFuncionario);
76.             session.delete(funcionario);
77.             tx.commit();
78.         } catch (HibernateException e) {
79.             if (tx != null) {
80.                 tx.rollback();
81.             }
82.             e.printStackTrace();
83.         } finally {
84.             session.close();
85.         }
86.     }
```

Um objeto **Session** pode ser considerado como uma sessão de comunicação com o banco de dados através de uma conexão JDBC. A operação **beginTransaction()**, presente na linha 17, inicia uma transação. Em seguida, o método **save()**, invocado na linha 18, realiza a persistência do objeto. Com isso, um registro é adicionado na tabela *funcionario* de acordo com os valores definidos no objeto. Por fim, na linha 19, o **commit** finaliza a transação e a sessão é fechada na linha 26.

Além deste, outros três métodos foram criados, a saber: **apagarFuncionario()**, **atualizarFuncionario()** e **ListarFuncionarios()**. Basicamente, esses métodos possuem a mesma estrutura com relação ao método já explicado. As diferenças encontram-se nas linhas 37, 61 e 79.

Na linha 37, o método **createCriteria()**, da classe **Session**, cria uma query passando como parâmetro a classe que vai ser pesquisada; no nosso caso, **Funcionario**. Por sua vez, o método **update()**, na linha 61, realiza a atualização do objeto passado como parâme-

tro, assim como o método **delete()**, chamado na linha 79, remove o funcionário passado como parâmetro da base de dados.

Depois de criar a classe **FuncionarioDAO**, deve-se criar a classe **ProjetoDAO**, no mesmo pacote e da mesma forma que a classe anterior. O código fonte ficará conforme a **Listagem 8**. Como podemos verificar, a lógica empregada foi a mesma utilizada para o DAO da classe **Funcionario**. A única alteração é que o objeto que passará a ser persistido é do tipo **Projeto**.

Testando a aplicação

Nesse momento, podemos partir para os testes. Para isso, vamos criar duas classes de teste: **TesteEnvers1** e **TesteEnvers2**. Visando manter a organização do projeto, crie estas classes no pacote **com.jm.teste**.

O código da classe **TesteEnvers1** é apresentado na **Listagem 9**. O seu objetivo é cadastrar alguns funcionários na base de dados. A partir da linha 8 (até a linha 11) são instanciados quatro objetos

Listagem 8. Código da classe ProjetoDAO. Disponibiliza todas as operações que serão realizadas no banco de dados.

```

01. package com.jm.acessobd;
02.
03. //imports omitidos...
04.
05. public class ProjetoDAO {
06.     private static SessionFactory factory;
07.
08.     public ProjetoDAO() {
09.         factory = HibernateUtil.getSessionFactory();
10.     }
11.
12.     public Integer adicionarProjeto(Projeto projeto) {
13.         Session session = factory.openSession();
14.         Transaction tx = null;
15.         Integer cod_func = null;
16.         try {
17.             tx = session.beginTransaction();
18.             cod_func = (Integer) session.save(projeto);
19.             tx.commit(); //executa o commit
20.         } catch (HibernateException e) {
21.             if (tx != null) {
22.                 tx.rollback();
23.             }
24.             e.printStackTrace();
25.         } finally {
26.             session.close();
27.         }
28.         return cod_func;
29.     }
30.
31.     public List<Projeto> listarProjetos() {
32.         Session session = factory.openSession();
33.         Transaction tx = null;
34.         List<Projeto> projetos = null;
35.         try {
36.             tx = session.beginTransaction();
37.             projetos = session.createCriteria(Projeto.class).list();
38.             tx.commit();
39.         } catch (HibernateException e) {
40.             if (tx != null) {
41.                 tx.rollback();
42.             }
43.             e.printStackTrace();
44.         } finally {
45.             session.close();
46.         }
47.         return projetos;
48.     }
49.
50.     public void atualizarProjeto(Integer cod, String nome) {
51.         Session session = factory.openSession();
52.         Transaction tx = null;
53.         try {
54.             tx = session.beginTransaction();
55.             Projeto projeto = (Projeto) session.get(Projeto.class, cod);
56.             projeto.setNome(nome);
57.             session.update(projeto);
58.             tx.commit();
59.         } catch (HibernateException e) {
60.             if (tx != null) {
61.                 tx.rollback();
62.             }
63.             e.printStackTrace();
64.         } finally {
65.             session.close();
66.         }
67.     }
68.
69.     public void apagarProjeto(Integer cod) {
70.         Session session = factory.openSession();
71.         Transaction tx = null;
72.         try {
73.             tx = session.beginTransaction();
74.             Projeto projeto = (Projeto) session.get(Projeto.class, cod);
75.             session.delete(projeto);
76.             tx.commit();
77.         } catch (HibernateException e) {
78.             if (tx != null) {
79.                 tx.rollback();
80.             }
81.             e.printStackTrace();
82.         } finally {
83.             session.close();
84.         }
85.     }
86. }
```

da classe **Funcionario**. Esses objetos são internalizados no banco de dados através do método `adicionarFuncionario()`, implementado na classe **FuncionarioDAO**.

A **Listagem 10** mostra o resultado da execução dessa classe, obtido através do console da IDE Eclipse.

Listagem 9. Código da classe de teste 1.

```
01. package com.jm.teste;
02. //imports omitidos...
03.
04. public class TesteEnvers1 {
05.
06.     public static void main(String[] args) throws ParseException {
07.
08.         Funcionario funcionario1 = new Funcionario("Clenio Rocha",
09.             "Analista de TI", 5600, "06493084765", "Rua Rio Paranaíba");
10.         Funcionario funcionario2 = new Funcionario("Carlos Martins",
11.             "Analista de TI", 4300, "06433055765", "Rua Abadia dos Dourados");
12.         Funcionario funcionario3 = new Funcionario("Daniel Cioqueta",
13.             "Analista de TI", 4200, "06493084444", "Rua Rio das Ostras");
14.         Funcionario funcionario4 = new Funcionario("Yujo Rodrigues",
15.             "Gerente de Projetos", 4110, "07777775765", "Rua dos Patos");
16.
17.         FuncionarioDAO funcionarioDao = new FuncionarioDAO();
18.         funcionarioDao.adicionarFuncionario(funcionario1);
19.         funcionarioDao.adicionarFuncionario(funcionario2);
20.         funcionarioDao.adicionarFuncionario(funcionario3);
21.         funcionarioDao.adicionarFuncionario(funcionario4);
22.     }
23. }
```

Listagem 10. Saída gerada pela execução da classe TesteEnvers1.

```
01. Hibernate: insert into Funcionario (cargo, cpf, endereco, nome, salario)
   values (?, ?, ?, ?, ?)
02. Hibernate: insert into REVINFO (REVTSTMP) values (?)
03. Hibernate: insert into Funcionario_AUD (REVTYPE, cargo, cpf, endereco,
   nome, salario, codigo, REV) values (?, ?, ?, ?, ?, ?, ?, ?)
04. Hibernate: insert into Funcionario (cargo, cpf, endereco, nome, salario)
   values (?, ?, ?, ?, ?)
05. Hibernate: insert into REVINFO (REVTSTMP) values (?)
06. Hibernate: insert into Funcionario_AUD (REVTYPE, cargo, cpf, endereco,
   nome, salario, codigo, REV) values (?, ?, ?, ?, ?, ?, ?, ?)
07. Hibernate: insert into Funcionario (cargo, cpf, endereco, nome, salario)
   values (?, ?, ?, ?, ?)
08. Hibernate: insert into REVINFO (REVTSTMP) values (?)
09. Hibernate: insert into Funcionario_AUD (REVTYPE, cargo, cpf, endereco,
   nome, salario, codigo, REV) values (?, ?, ?, ?, ?, ?, ?, ?)
10. Hibernate: insert into Funcionario (cargo, cpf, endereco, nome, salario)
   values (?, ?, ?, ?, ?)
11. Hibernate: insert into REVINFO (REVTSTMP) values (?)
12. Hibernate: insert into Funcionario_AUD (REVTYPE, cargo, cpf, endereco,
   nome, salario, codigo, REV) values (?, ?, ?, ?, ?, ?, ?, ?)
```

Perceba que, além dos *inserts* na tabela `funcionario` (linhas 1, 4, 7 e 10), foram feitas inserções em outras duas tabelas: `REVINFO` e `Funcionario_AUD`.

A tabela `Funcionario_AUD` armazenará o histórico de mudanças sofridas pela entidade `funcionário`, sendo composta por todos os campos auditados da tabela original, além de outros dois atributos: o código da revisão (`REV`), que representa a chave primária da tabela; e o tipo de revisão (`REVTYPE`), que representa a ação executada, como inclusão, alteração ou exclusão de dados.

Já a tabela `REVINFO` inclui os seguintes campos: `REV`, que faz a ligação com as tabelas de auditoria individuais (as de final `_aud`); e `REVTSTMP`, que indica o momento exato (*timestamp*) em que a operação ocorreu.

A **Figura 8** mostra as alterações provocadas na estrutura do banco de dados após a execução da classe de teste. Note que além das tabelas mencionadas, foram criadas as tabelas versionadas `Projeto_AUD` e `Trabalha_AUD`, que armazenarão o histórico das mudanças sofridas por suas respectivas entidades. Nesse momento, essas tabelas encontram-se vazias pois ainda não foi realizada qualquer operação sobre as tabelas originais.

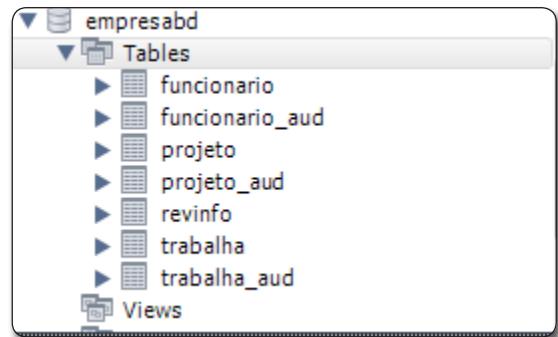


Figura 8. Visão do banco de dados após a execução da classe de teste

A **Figura 9** mostra os registros adicionados à tabela `Funcionario_AUD`. Além de todos os campos auditados, podemos verificar as colunas `REV` e `REVTYPE`. A primeira coluna, como já mencionado, armazena o identificador da revisão, e é chave estrangeira da tabela `REVINFO`. Já a segunda coluna indica o tipo de ação executada. O valor 0 indica que foi adicionado um novo registro. A **Figura 9** mostra, ainda, a partir de uma consulta, como ficou a tabela `REVINFO`. Essa tabela armazena os identificadores das revisões, assim como o momento em que cada operação ocorreu.

Dando sequência aos testes, vamos realizar mais algumas ações contra o banco de dados para avaliar outros detalhes sobre como o Hibernate Envers se comporta. Para isso, execute o código da classe `TesteEnvers2`, apresentado na **Listagem 11**.

O objetivo desta classe é cadastrar um projeto e associar alguns funcionários a ele. Os funcionários associados ao projeto serão aqueles criados na execução anterior e que nesse momento encontram-se na base de dados. A **Listagem 12** mostra o resultado da execução, obtido através do console do Eclipse.

Analisando o resultado da execução da classe `TesteEnvers2`, note que é realizada uma busca por todos os funcionários cadastrados na linha 1. Como trata-se de uma busca e nenhuma alteração é realizada nas tabelas auditadas, nenhum *insert* será registrado nas tabelas versionadas.

Na linha 2, nota-se que foi criada uma entrada na tabela `projeto`. Esta ação gerou – na linha 12 – a adição de um registro na tabela `Projeto_AUD`. Esse registro contém os campos da tabela original, o tipo de operação e o código da revisão.

Em seguida, a atualização realizada na tabela *Funcionario* (vide linhas 3, 4, 5 e 6) também gerou a inclusão de registros em *Funcionario_AUD*. As linhas 13, 14, 15 e 16 mostram os *inserts* realizados nesta tabela.

Note ainda que a tabela *trabalha* também sofreu alterações. Foram inseridos quatro registros, nas linhas 7, 8, 9 e 10. Em virtude dessa alteração, a tabela versionada *Trabalha_AUD* também recebeu inserções, como expõe as linhas 17, 18, 19 e 20.

Listagem 11. Código da classe de teste 2.

```

01. package com.jm.teste;
02.
03. //imports omitidos
04.
05. public class TesteEnvers2 {
06.
07.     public static void main(String[] args) throws ParseException {
08.
09.         FuncionarioDAO funcionarioDao = new FuncionarioDAO();
10.         List<Funcionario> funcionarios = funcionarioDao.listarFuncionarios();
11.
12.         ProjetoDAO projetoDao = new ProjetoDAO();
13.
14.         Date dataInicio = Date.from(LocalDate.of(2014, 12, 25).atStartOfDay()).
15.             atZone(Zonelid.systemDefault()).toInstant());
16.         Date dataFim = Date.from(LocalDate.of(2015 , 5, 25).atStartOfDay()).
17.             atZone(Zonelid.systemDefault()).toInstant());
18.
19.         Projeto projeto = new Projeto("Nono Digito", dataInicio, dataFim,
20.             funcionarios);
21.         projetoDao.adicionarProjeto(projeto);
22.     }
23. }
```

Listagem 12. Saída da execução da classe TesteEnvers2.

01. Hibernate: select this_.codigo as codigo1_0_0_, this_.cargo as cargo2_0_0_, this_.cpf as cpf3_0_0_, this_.endereco as endereco4_0_0_, this_.nome as nome5_0_0_, this_.salario as salario6_0_0_ from Funcionario this_
02. Hibernate: insert into Projeto (data_Fim, data_Inicio, nome) values (?, ?, ?)
03. Hibernate: update Funcionario set cargo=?, cpf=?, endereco=?, nome=?, salario=? where codigo=?
04. Hibernate: update Funcionario set cargo=?, cpf=?, endereco=?, nome=?, salario=? where codigo=?
05. Hibernate: update Funcionario set cargo=?, cpf=?, endereco=?, nome=?, salario=? where codigo=?
06. Hibernate: update Funcionario set cargo=?, cpf=?, endereco=?, nome=?, salario=? where codigo=?
07. Hibernate: insert into trabalha (cod_projeto, cod_funcionario) values (?, ?)
08. Hibernate: insert into trabalha (cod_projeto, cod_funcionario) values (?, ?)
09. Hibernate: insert into trabalha (cod_projeto, cod_funcionario) values (?, ?)
10. Hibernate: insert into trabalha (cod_projeto, cod_funcionario) values (?, ?)
11. Hibernate: insert into REVINFO (REVTSTMP) values (?)
12. Hibernate: insert into Projeto_AUD (REVTYPE, data_Fim, data_Inicio, nome, codigo, REV) values (?, ?, ?, ?, ?, ?)
13. Hibernate: insert into Funcionario_AUD (REVTYPE, cargo, cpf, endereco, nome, salario, codigo, REV) values (?, ?, ?, ?, ?, ?, ?, ?)
14. Hibernate: insert into Funcionario_AUD (REVTYPE, cargo, cpf, endereco, nome, salario, codigo, REV) values (?, ?, ?, ?, ?, ?, ?, ?)
15. Hibernate: insert into Funcionario_AUD (REVTYPE, cargo, cpf, endereco, nome, salario, codigo, REV) values (?, ?, ?, ?, ?, ?, ?, ?)
16. Hibernate: insert into Funcionario_AUD (REVTYPE, cargo, cpf, endereco, nome, salario, codigo, REV) values (?, ?, ?, ?, ?, ?, ?, ?)
17. Hibernate: insert into trabalha_AUD (REVTYPE, REV, cod_projeto, cod_funcionario) values (?, ?, ?, ?)
18. Hibernate: insert into trabalha_AUD (REVTYPE, REV, cod_projeto, cod_funcionario) values (?, ?, ?, ?)
19. Hibernate: insert into trabalha_AUD (REVTYPE, REV, cod_projeto, cod_funcionario) values (?, ?, ?, ?)
20. Hibernate: insert into trabalha_AUD (REVTYPE, REV, cod_projeto, cod_funcionario) values (?, ?, ?, ?)

The screenshot shows the MySQL 5.6 Command Line Client interface. The command line shows the following output:

```

MySQL 5.6 Command Line Client - Unicode
Enter password: *****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 8
Server version: 5.6.21-1-log MySQL Community Server (GPL)

Copyright (c) 2000, 2014, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> use empresabd;
Database changed
mysql> select * from Funcionario_audit;
+-----+-----+-----+-----+-----+-----+
| codigo | REV | REVTYPE | cargo | cpf | endereco | nome | salario |
+-----+-----+-----+-----+-----+-----+
| 1 | 1 | 0 | Analista de TI | 06493084765 | Rua Rio Paranaiba | Clenio Rocha | 5600 |
| 2 | 2 | 0 | Analista de TI | 06433055765 | Rua Abadia dos Dourados | Carlos Martins | 4300 |
| 3 | 3 | 0 | Analista de TI | 06493084444 | Rua Rio das Ostras | Daniel Cioqueta | 4200 |
| 4 | 4 | 0 | Gerente de Projetos | 07777775765 | Rua dos Patos | Yujo Rodrigues | 4110 |
+-----+-----+-----+-----+-----+-----+
4 rows in set (0.11 sec)

mysql> select * from revinfo;
+-----+-----+
| REV | REVSTAMP |
+-----+-----+
| 1 | 1430338743469 |
| 2 | 1430338743599 |
| 3 | 1430338743689 |
| 4 | 1430338743899 |
+-----+-----+
4 rows in set (0.00 sec)

mysql>
```

Figura 9. Visão das tabelas alteradas após a execução da classe de teste

Hibernate Envers: Auditoria de dados em Java

```
mysql> use empresabd;
Database changed
mysql> select * from Funcionario_aud;
+-----+-----+-----+-----+-----+
| codig | REV | REVTYPE | cargo | cpf   |
+-----+-----+-----+-----+-----+
| 1    | 1   | 0     | Analista de TI | 06433084765 |
| 1    | 5   | 1     | Analista de TI | 06433084765 |
| 2    | 2   | 0     | Analista de TI | 06433055765 |
| 2    | 5   | 1     | Analista de TI | 06433055765 |
| 3    | 3   | 0     | Analista de TI | 06433084444 |
| 3    | 5   | 1     | Analista de TI | 06433084444 |
| 4    | 4   | 0     | Gerente de Projetos | 0777775765 |
| 4    | 5   | 1     | Gerente de Projetos | 0777775765 |
+-----+-----+-----+-----+-----+
8 rows in set (0.00 sec)

mysql> select * from projeto;
+-----+-----+-----+-----+
| codig | nome | data_inicio | data_fim |
+-----+-----+-----+-----+
| 1    | Nono Dígito | 2014-12-25 | 2015-05-25 |
+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> select * from projeto_aud;
+-----+-----+-----+-----+-----+
| codig | REV | REVTYPE | data_fim | data_inicio |
+-----+-----+-----+-----+-----+
| 1    | 5   | 0     | 2015-05-25 00:00:00 | 2014-12-25 00:00:00 |
+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> select * from trabalho;
+-----+-----+
| cod_funcionario | cod_projeto |
+-----+-----+
| 1               | 1           |
| 2               | 1           |
| 3               | 1           |
| 4               | 1           |
+-----+-----+
4 rows in set (0.00 sec)

mysql> select * from trabalho_aud;
+-----+-----+-----+-----+
| REV | cod_projeto | cod_funcionario | REVTYPE |
+-----+-----+-----+-----+
| 5   | 1           | 1             | 0       |
| 5   | 1           | 2             | 0       |
| 5   | 1           | 3             | 0       |
| 5   | 1           | 4             | 0       |
+-----+-----+-----+-----+
4 rows in set (0.00 sec)
```

Figura 10. Visão das tabelas após a execução da classe TesteEnvers2

Por fim, a tabela *REVINFO* (vide linha 11) recebeu a inclusão de um registro que contém o identificador da nova revisão e o momento exato em que ela ocorreu.

A Figura 10 mostra a visão das tabelas do banco após a execução da classe *TesteEnvers2*.

O Hibernate Envers é um poderosíssimo recurso para realização de auditoria que torna desnecessário, por exemplo, criar uma estrutura específica para as auditorias diretamente no SGBD, como Stored Procedures ou triggers, recursos muito utilizados nos sistemas atuais.

Diante disso, no desenvolvimento de projetos que requerem auditoria, certamente o desenvolvedor poderá incorporar o Envers para o tratamento deste requisito, minimizando os impactos por ser um *framework* transparente, baseado em revisões e, principalmente, por não ser intrusivo.

Além de tudo isso, pode tornar o trabalho muito mais produtivo com o mínimo de alterações no código, pois como vimos, basta a inclusão de algumas anotações e poucas configurações para que o framework funcione.

Autor



Carlos Alberto Silva

casilvamg@hotmail.com



É formado em Ciéncia da Computação pela Universidade Federal de Uberlândia (UFU), com especialização em Desenvolvimento Java pelo Centro Universitário do Triângulo (UNITRI) e em Análise e Desenvolvimento de Sistemas Aplicados a Gestão Empresarial pelo Instituto Federal do Triângulo Mineiro (IFTM). Trabalha atualmente na empresa Algar Telecom como Analista de TI. Possui as seguintes certificações: OCJP, OCWCD e ITIL.

Links:

Site oficial do Hibernate Envers.

<http://hibernate.org/orm/envers/>

Site oficial do MySQL.

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

Endereço para download do driver do MySQL.

<http://dev.mysql.com/downloads/connector/j/>

Site oficial do Eclipse.

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

Endereço para download do JDK.

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

Endereço para download do Maven.

<http://maven.apache.org/download.html>

Christian Bauer, Gavin King. Livro Java Persistence com Hibernate, 2005.

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