easy java Magazine

Edição 54

DEVMEDIA

DESAFIO: Qualidade de código com DDD e TDD
Conheça padrões e boas práticas para desenvolver como um profissional

Estruturas de dados com classes genéricas
Saiba como escrever um código seguro e flexível com Generics

PERSISTÊNCIA COM HIBERNATE

Aprenda a criar relacionamentos

ISSN 2179625-4

summa
Technology + Business
Premium Partner

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

Acompanhe a Toolscloud:

toolscloud@toolscloud.com

twitter.com/toolscloud

ToolsCloud

toolscloud.com

Edição 54 • 2015 • ISSN 2173625-4

EXPEDIENTE

Editor
Eduardo Spínola (*eduspinola@gmail.com*)

Consultor Técnico Diogo Souza (diogosouzac@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 a 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

Sumário

Estruturas de dados com classes genéricas

Aprenda a utilizar classes genéricas em Java para facilitar o uso e aumentar a segurança de classes que implementam estruturas de dados

No artigo "Programando com estruturas de dados e padrões de projeto", publicado na Easy Java Magazine número 49, a estrutura de dados que utiliza uma sequência de nós auto ligados, conhecida como lista encadeada, foi descrita. No mesmo texto apresentou-se também o padrão *Iterator*, que permite a navegação, ou iteração, pela lista sem que seja necessário conhecer ou ter acesso aos detalhes da sua implementação, que pode ser, como no exemplo, por encadeamentos duplos, por encadeamentos simples ou mesmo utilizando vetores.

As listas são estruturas de dados muito flexíveis, permitindo que novos elementos sejam incluídos em qualquer posição, e também que sejam removidos de qualquer posição. Estas características já foram explicadas e demonstradas anteriormente, e o foco agora é em como melhorar a usabilidade da classe que as implementa.

Uma situação que pode ocorrer ao utilizar uma estrutura de dados como a que foi definia, e cuja estrutura está resumida na **Listagem 1**, é a inclusão, por acidente, de mais de um tipo de dado na mesma lista. Uma situação como esta é possível porque no código, para que seja viável armazenar referências a quaisquer objetos, os dados foram declarados como sendo do tipo **Object**. Como em Java todas as classes são estendidas direta ou indiretamente da classe **Object**, é possível referenciar qualquer objeto como sendo deste tipo, o que permite, também, que a classe de lista funcione adequadamente sem que seja necessário alterar seu código para cada tipo com o qual se desejar trabalhar. Isto porque a lista não necessita saber qual é o tipo que está armazenando, mas apenas de uma referência ao objeto para que possa responder às consultas realizadas.

Como já exposto, esta característica, que permite e facilita a implementação, também é uma fonte de problemas, pois como qualquer tipo de objeto pode ser referenciado como um **Object**, é possível que inadvertidamente sejam incluídas na mesma lista referências a objetos de tipos diferentes, causando, assim, comportamentos estranhos ou mesmo falhas graves na execução do sistema.

Nas **Listagens 2** e **3** é apresentado um exemplo de utilização desta classe de lista explicitando este problema e a sua correspondente saída, com a inclusão de três valores do tipo **String** e três valores do tipo **int**. No exemplo, não foram realizadas operações que geram exceções, mas na saída apresentada pelo teste é possível perceber a diferença entre os tipos de dados armazenados.

Uma falha grave ocorreria se, por exemplo, fosse necessário realizar alguma operação específica com os dados, como o cálculo da soma de todos os valores armazenados, assumindo que a lista deveria conter apenas valores inteiros.

Fique por dentro

Este artigo é útil por apresentar como utilizar classes genéricas, ou classes parametrizadas, para codificar uma estrutura de dados, de maneira que o compilador consiga realizar a checagem dos tipos que nela forem armazenados. Além disso, algumas características mais avançadas das classes genéricas, como as que fazem uso de conceitos como o relacionamento de herança entre classes, serão explicadas e demonstradas, de forma que, em um processo incremental, problemas como a inclusão inadvertida de elementos de tipos distintos na estrutura de dados; uso de hierarquias de classes para definir um conjunto de diferentes tipos de elementos que podem ser manipulados na estrutura; e também a especificação de um comportamento padronizado para os dados armazenados serão expostos e as suas soluções discutidas com base nos diferentes exemplos de código para cada uma destas situações. A partir desse conteúdo o leitor será capaz de desenvolver um código mais seguro e flexível, requisitos fundamentais para alcançar um código de qualidade.

Listagem 1. Classe para a estrutura de dados lista duplamente encadeada.

```
public class ListaDupEncadeada {
  // A classe que representa os nós da lista utiliza a classe Object para definir
  //a referência ao dado associado à posição da lista que este nó representa.
  private class No{
    // Referência ao próximo elemento da lista
    No proximo;
    // Referência ao elemento anterior na lista
    No anterior;
    // Referência ao dado armazenado no nó atual da lista
    Object dado;

    // Constrói um nó para armazenar um objeto na lista
    No(Object obj) { ... }
    // Constrói um nó para armazenar um objeto na lista, indicando quais os
    // nós anterior e próximo
    No(Object obj, No prox, No ant) { ... }
  }
  // Implementação da interface Iterator, que define os métodos de navegação na lista
  private class IteratorLista implements Iterator {
    // Referência ao nó apontado pelo iterador da lista durante a navegação
    No noAtual;

    // Retorna o dado associado ao nó atual. Caso o iterador não seja válido
    // (noAtual é nulo), retorna um objeto nulo
    public Object dado() { ... }
    // Coloca o iterador no próximo elemento da lista e retorna o dado
    // associado ao nó atual. Caso o iterador não seja válido (noAtual é
    // nulo), retorna um objeto nulo
    public Object proximo() { ... }
    // Coloca o iterador no elemento anterior da lista e retorna o dado
    // associado ao nó atual. Caso o iterador não seja válido (noAtual é
    // nulo), retorna um objeto nulo
    public Object anterior() { ... }
    // Verifica se existe um nó após o nó atual
    public boolean temProximo() { ... }
    // Verifica se existe um nó antes do nó atual
    public boolean temAnterior() { ... }
    // Verifica se o nó atual existe e é válido
    public boolean eValido() { ... }
  }

  // Nó inicial (primeiro) da lista encadeada
  No        inicio;
  // Nó final (último) da lista encadeada
  No        fim;
  // Tamanho da lista encadeada (número de nós na lista)
  int       tamanho;

  // Construtor
  public ListaDupEncadeada() { ... }
  // Retorna uma instância de Iterator para o primeiro nó da lista
  public Iterator iteradorInicio() { ... }
  // Retorna uma instância de Iterator para o último nó da lista
  public Iterator iteradorFim() { ... }
  // Retorna o tamanho da lista
  public int getTamanho() { ... }
  // Insere um novo nó no início da lista
  public int insereInicio(Object obj { ... }
  // Insere um novo nó no final da lista
  public int insereFim(Object obj { ... }
  // Insere um novo nó na posição indicada
  public int insere(Object obj, int pos) { ... }
  // Remove o nó do início da lista - retira o primeiro elemento
  // Retorna o objeto armazenado no primeiro nó da lista - que foi removido.
  public Object removeInicio(){ ... }
  // Remove o nó do final da lista - retira o último elemento
  // Retorna o objeto do último nó da lista - que foi removido.
  public Object removeFim(){ ... }
  // Remove um nó do meio da lista - retira o elemento da posição indicada
  // Retorna o objeto armazenado na posição indicada - que foi removida.
  public Object remove(int pos) { ... }
  // Retorna o objeto armazenado na posição indicada, sem remover o mesmo da lista
  public Object consulta(int pos) { ... }
}
```

Listagem 2. Teste da classe de lista explicitando a inclusão de diferentes tipos de dados.

```
public class TesteLista {
  public static void main(String[] args)              {
    // Lista encadeada
    ListaDupEncadeada lista = new ListaDupEncadeada();
    int val;

    // Strings a serem inseridas na lista
    String a = "str teste 01";
    String c = "str teste 03";
    String e = "str teste 05";

    // Inteiros a serem inseridos na lista
    int b = 2;
    int d = 4;
    int f = 6;

    // Mostra o tamanho atual da lista
    System.out.println("Tamanho: " + lista.getTamanho());

    // Insere uma String no início da lista
    val = lista.insereInicio(a);
    // Mostra o conteúdo da primeira posição da lista
    System.out.println("Inserindo: \"" + lista.consulta(1) + "\" -> Tamanho: " + val);
    // Insere um inteiro no início da lista
    val = lista.insereInicio(b);
    System.out.println("Inserindo: \"" + lista.consulta(1) + "\" -> Tamanho: " + val);
    // Insere uma String no início da lista
    val = lista.insereInicio(c);
    System.out.println("Inserindo: \"" + lista.consulta(1) + "\" -> Tamanho: " + val);
    // Insere um inteiro no início da lista
    val = lista.insereInicio(d);
    System.out.println("Inserindo: \"" + lista.consulta(1) + "\" -> Tamanho: " + val);
    // Insere uma String no início da lista
    val = lista.insereInicio(e);
    System.out.println("Inserindo: \"" + lista.consulta(1) + "\" -> Tamanho: " + val);
    // Insere um inteiro no início da lista
    val = lista.insereInicio(f);
    System.out.println("Inserindo: \"" + lista.consulta(1) + "\" -> Tamanho: " + val);

    // Mostra o tamanho atual da lista
    System.out.println("\nTamanho: " + lista.getTamanho());

    System.out.println("");

    // Iterador utilizado para percorrer a lista encadeada
    Iterator iterador = lista.iteradorInicio();

    // Contador de posições
    int i = 1;
    // Laço para percorrer a lista utilizando o iterador
    while (iterador.eValido()) {
      // Mostra o valor armazenado em cada posição da lista
      System.out.println("Posição " + i++ + ": -> \t" + iterador.dado());
      iterador.proximo();
    }

    System.out.println("");
    // Laço para remover todos os elementos da lista
    while(lista.getTamanho() > 0) {
      System.out.println("Removido do início: \"" + lista.removeInicio() +
      "\" -> Tamanho: " + lista.getTamanho());
    }
  }
}
```

Listagem 3. Saída do teste explicitando a inclusão de diferentes tipos de dados.

```
Tamanho: 0
Inserindo: str teste 01              -> Tamanho: 1
Inserindo: 2 -> Tamanho: 2
Inserindo: str teste 03              -> Tamanho: 3
Inserindo: 4 -> Tamanho: 4
Inserindo: str teste 05              -> Tamanho: 5
Inserindo: 6 -> Tamanho: 6

Tamanho: 6

Posição 1: -> 6
Posição 2: -> str teste 05
Posição 3: -> 4
Posição 4: -> str teste 03
Posição 5: -> 2
Posição 6: -> str teste 01

Removido do início: 6 Tamanho: 5
Removido do início: str teste 05      Tamanho: 4
Removido do início: 4 Tamanho: 3
Removido do início: str teste 03      Tamanho: 2
Removido do início: 2 Tamanho: 1
Removido do início: str teste 01      Tamanho: 0
```

Métodos genéricos

No exemplo da **Listagem 2** é possível perceber que não há nenhuma indicação, por parte do compilador e nem mesmo durante a execução do programa, de que existe um possível erro no código. Isto porque não foram realizadas operações incompatíveis entre os tipos de dados. Caso os elementos da lista fossem utilizados em um cálculo de média, no entanto, um erro de execução ocorreria e uma exceção seria lançada. Para evitar este tipo de situação, é possível informar ao compilador Java que um tipo específico de objetos será adotado. Este recurso da linguagem pode ser definido em nível de métodos ou classes, sendo chamado de **Métodos Genéricos** e **Classes Genéricas**.

Genéricos permitem que um método possa ser declarado para tratar tipos diferentes de objetos, que serão definidos no momento de sua utilização. Por exemplo, ao se trabalhar com vetores de diferentes tipos, quando for necessário mostrar o conteúdo dos mesmos no console, um conjunto de métodos sobrecarregados, sendo um para cada tipo de classe presente nas definições, deve ser implementado. Portanto, ao se declarar vetores para armazenar, por exemplo, valores inteiros, ou reais ou caracteres, três métodos sobrecarregados devem ser escritos de modo a poder tratar corretamente as diferenças entre os tipos de dados, um para cada tipo de vetor que está presente no código.

O uso de métodos genéricos facilita o desenvolvimento nestes casos porque a única alteração necessária entre as diferentes versões do método é a declaração dos tipos de dados utilizados, ou seja, qual é o tipo dos objetos que estão sendo manipulados em uma determinada chamada ao método. Isto fica bem claro no exemplo da **Listagem 4**, no qual três métodos sobrecarregados são implementados para mostrar o conteúdo de vetores dos tipos **Integer**, **Float** e **Character**. É importante notar que não foram declarados vetores a partir dos tipos primitivos **int**, **float** e **char** porque métodos genéricos não podem fazer uso de tipos primitivos.

Em cada um dos métodos repete-se o mesmo código para iterar sobre os elementos dos vetores de cada tipo. No laço, cada elemento dos vetores tem sua representação textual enviada ao console a partir da chamada implícita ao método **toString()** da classe **Object** e que é sobrescrito nas classes que empacotam os tipos de dados primitivos, utilizadas no código. O resultado é apresentado na **Listagem 5**.

Listagem 4. Exemplo de métodos sobrecarregados para mostrar o conteúdo de vetores de diferentes tipos.

```
public class ExemploSemGenerico
{
  // Método sobrecarregado que mostra a representação em formato
  //de texto do conteúdo de um vetor de números inteiros
  public static void mostra(Integer[] vet) {
    System.out.printf("[ ");
    for (Integer e : vet)
      System.out.printf("%s ", e);
    System.out.printf("]\n");
  }

  // Método sobrecarregado que mostra a representação em formato
  //de texto do conteúdo de um vetor de números reais do tipo float
  public static void mostra(Float[] vet) {
    System.out.printf("[ ");
    for (Float e : vet)
      System.out.printf("%s ", e);
    System.out.printf("]\n");
  }

  // Método sobrecarregado que mostra a representação em formato
  //de texto do conteúdo de um vetor de caracteres
  public static void mostra(Character[] vet) {
    System.out.printf("[ ");
    for (Character e : vet)
      System.out.printf("%s ", e);
    System.out.printf("]\n");
  }

  // Programa que utiliza métodos sobrecarregados para mostrar
  //o conteúdo de vetores de números inteiros, números reais e caracteres
  public static void main(String[] args) {
    Integer vetI[] = { 1, 2, 3, 4, 5 };
    Float vetF[] = { 0.12345f, 0.23456f, 0.34567f, 0.45678f, 0.56789f };
    Character vetC[] = {'A','B','C','D','E'};

    System.out.println("Para realizar operações similares em dados diferentes,
    solução utilizando sobrecarga:");
    System.out.println("Mostrando um vetor de inteiros:");
    mostra(vetI);
    System.out.println("Mostrando um vetor de números reais:");
    mostra(vetF);
    System.out.println("Mostrando um vetor de caracteres:");
    mostra(vetC);
  }
}
```

Listagem 5. Saída do exemplo com métodos sobrecarregados.

```
Para realizar operações similares em dados diferentes, solução utilizando sobrecarga:
Mostrando um vetor de inteiros:
[ 1 2 3 4 5 ]
Mostrando um vetor de números reais:
[ 0.12345 0.23456 0.34567 0.45678 0.56789 ]
Mostrando um vetor de caracteres:
[ A B C D E ]
```

O mais importante no caso de métodos sobrecarregados é lembrar que o compilador os diferencia pela sua lista de parâmetros. Portanto, ao encontrar as chamadas a cada um dos métodos no programa principal, o compilador realiza uma busca dentre os métodos que possuem o mesmo nome para identificar qual possui os parâmetros que correspondem à chamada sendo executada.

Uma vez que o compilador consegue identificar métodos sobrecarregados pela sua lista de parâmetros, e considerando que a única diferença entre os métodos do exemplo são os tipos de dados utilizados, é fácil pensar em uma simplificação, na qual se substitui o nome do tipo, ou seja, da classe que o método está utilizando, por um identificador que deve assumir uma nova classe a cada chamada ao método. Com isso, não só o valor do parâmetro, mas também o seu tipo, são especificados no momento da chamada ao método, deixando a tarefa de repetir o código para o compilador, eliminando uma boa quantidade de métodos repetidos em situações similares às descritas no exemplo anterior.

A sintaxe para especificar este identificador é informar o seu nome entre os sinais de maior e menor antes de definir o retorno do método, e utilizar este nome onde for necessário em todo o código do método, como ilustrado na **Listagem 6**, que implementa o mesmo exemplo da **Listagem 4**. Nesta nova versão do exemplo fica bem clara uma das vantagens de se utilizar métodos genéricos: a redução no tamanho do código.

Listagem 6. Exemplo de método genérico para mostrar o conteúdo de vetores de diferentes tipos.

```
public class ExemploMetodoGenerico
{
  // Método genérico que mostra a representação em formato de texto
  //do conteúdo de um vetor cujo tipo será identificado em tempo de compilação
  public static <TIPO> void mostra(TIPO[] vet)
  {
    System.out.printf("[ ");
    for (TIPO e : vet)
      System.out.printf("%s ", e);
    System.out.printf("]\n");
  }

  // Programa que utiliza um método genérico para mostrar o conteúdo
  //de vetores de números inteiros, números reais e caracteres
  public static void main(String[] args)
  {
    Integer vetI[] = { 1, 2, 3, 4, 5 };
    Float vetF[] = { 0.12345f, 0.23456f, 0.34567f, 0.45678f, 0.56789f };
    Character vetC[] = {'A','B','C','D','E'};

    System.out.println("Para realizar operações similares em dados diferentes,
    solução utilizando método genérico:");

    System.out.println("Mostrando um vetor de inteiros:");
    mostra(vetI);

    System.out.println("Mostrando um vetor de números reais:");
    mostra(vetF);

    System.out.println("Mostrando um vetor de caracteres:");
    mostra(vetC);
  }
}
```

Uma segunda vantagem é a diminuição de erros de codificação por falta de atenção ao se modificar os métodos, uma vez que é fácil e comum esquecer de alterar alguma das ocorrências dos nomes de tipos de uma cópia sobrecarregada para outra do método.

Comparando os exemplos das **Listagens 4** e **6**, as alterações no código do método **mostra()** são apenas a substituição do nome das classes dos parâmetros, que são diferentes para cada versão sobrecarregada, por um **parâmetro de tipo**. No caso foi utilizado o identificador **<TIPO>**, e a sua declaração antes do tipo de retorno do método. No corpo do programa principal, no método **main()**, não houve alteração. Portanto, o emprego de um método genérico é similar à codificação de métodos sobrecarregados. Além disso, o resultado deste código é exatamente o mesmo que o apresentado pelo código da **Listagem 4**.

Classes genéricas

Uma vez entendido o conceito de métodos genéricos, pode-se pensar da mesma forma em relação às classes, ou seja, se um conjunto de métodos sobrecarregados podem ser escritos como um método genérico, então uma classe que é projetada para lidar com diferentes tipos de classes também pode se beneficiar desta característica. Este é exatamente o caso com a classe que implementa a lista encadeada da **Listagem 1**, pois ela deve armazenar diferentes tipos de objetos.

Para que a classe de lista seja genérica é preciso informar um nome para o tipo de dado com o qual a classe deve trabalhar, de maneira similar à declaração de um método genérico. Isto é realizado no momento da declaração da classe, da seguinte maneira: **public class ListaDupEncadeada<E>**. Com a inclusão deste nome, o compilador é informado que a classe **ListaDupEncadeada** utiliza uma classe de objetos que será definida em tempo de compilação, e caso mais de uma declaração diferente seja feita, uma versão da classe para cada tipo de objeto será fornecida pelo compilador, conforme for necessário.

Uma vez que o tipo de objeto que será armazenado está sendo informado, a classe interna que representa cada um dos nós da lista pode especificar esta classe genérica no lugar de utilizar a classe **Object**, ou seja, a declaração do atributo **Object dado;** passa a ser **E dado;** e os parâmetros nos dois construtores também passam de **Object obj** para **E obj**, conforme mostrado no código da **Listagem 7**.

Lembre-se que a classe **Object** também é utilizada como tipo de parâmetro e de retorno nos métodos de inserção, remoção e consulta a elementos da lista, e nestes métodos a mesma adequação deve ser feita, como expõe o código da **Listagem 8**.

Com estas alterações, a classe de lista continua sem identificar qual é a classe que será armazenada em seus nós, porém no momento da compilação este tipo é definido e, portanto, o compilador pode tirar proveito disso para realizar as validações necessárias, tornando assim a classe mais segura e evitando que situações como a ilustrada no exemplo da **Listagem 2** ocorram, pois ao se tentar incluir um objeto que não seja do tipo correto, o compilador identifica a situação e gera um erro.

Para utilizar a versão genérica da classe de lista, é necessário informar, ao declarar um novo objeto, qual é o tipo a ser armazenado na lista antes do nome da variável que a representa. O tipo é informado entre os sinais de maior e menor, como na declaração da classe. O mesmo deve ser feito no momento de chamar o construtor da classe genérica. Portanto, a declaração de uma nova lista encadeada para armazenar objetos do tipo **Integer** é escrita da seguinte forma: **ListaDupEncadeada <Integer> nomeDaLista;**. E a instanciação de um novo objeto deste tipo, chamando o construtor da classe informando o tipo a ser armazenado, fica assim: **ListaDupEncadeada<Integer>();**. Estas são as únicas alterações necessárias para a utilização da classe apresentada na **Listagem 8**.

Listagem 7. Nova versão da classe interna que representa os nós da lista.

```
// A classe que representa os nós da lista utiliza o parâmetro de tipo <E> para
//definir a referência ao dado associado à posição da lista que este nó representa.
private class No{
   No        proximo;
   No        anterior;
   // Referência ao dado armazenado no nó atual da lista
   // O tipo do dado agora é um parâmetro passado à classe em tempo de compilação
   E         dado;

   // Constrói um nó para armazenar um objeto do tipo <E> na lista
   No(E obj) { ... }

   // Constrói um nó para armazenar um objeto do tipo <E> na lista,
   //indicando quais os nós anterior e próximo
   No(E obj, No prox, No ant) { ... }
}
```

Listagem 8. Nova versão da classe que implementa a lista duplamente encadeada de forma genérica.

```
// Classe que implementa a estrutura de dados Lista Duplamente Encadeada.
// A classe ListaDupEcadeada recebe um parâmetro informando qual é o tipo de
//dado que será armazenado em seus nós.
public class ListaDupEncadeada<E> {
   private class No { ... }

   // Implementação da interface Iterator, que define os métodos de navegação
   //na lista
   private class IteratorLista implements Iterator { ... }

   No        inicio;
   No        fim;
   int       tamanho;

   public ListaDupEncadeada() { ... }
   public Iterator iteradorInicio() { ... }
   public Iterator iteradorFim() { ... }
   public int getTamanho() { ... }
   public int insereInicio(E obj { ... }
   public int insereFim(E obj { ... }
   public int insere(E obj, int pos) { ... }
   public E removeInicio(){ ... }
   public E removeFim(){ ... }
   public E remove(int pos) { ... }
   public E consulta(int pos) { ... }
}
```

As chamadas a todos os métodos da classe permanecem exatamente como na versão anterior. A exceção se dá pela já comentada verificação de tipos realizada no momento da compilação, e que pode resultar em erros de compilação caso um tipo diferente do que foi especificado seja utilizado em alguma das chamadas aos métodos da classe **ListaDupEncadeada**.

Na **Listagem 9** está o código de uma classe de teste que utiliza a versão genérica da lista para armazenar um conjunto de objetos do tipo **String**. O programa consiste em um laço de inserção com cinco iterações, um laço de consulta, que utiliza um **Iterator** para navegar pela lista, e por último um bloco onde os elementos da lista são removidos. Note ainda que no laço de inserção de elementos existe uma linha de código comentada com o comando para inserir um valor inteiro na lista. Caso um comando como este esteja presente no código, o compilador Java consegue identificar que o valor passado como parâmetro ao método não é do tipo correto e informa o erro, diferente do que ocorreu no exemplo da **Listagem 2**, no qual elementos do tipo **String** e do tipo **int** foram inseridos na mesma lista. O resultado produzido por este programa de teste da **Listagem 9** pode ser verificado na **Listagem 10**.

Listagem 9. Teste com a classe de lista encadeada genérica.

```
public class TesteLista {
   public static void main(String[] args) {
      ListaDupEncadeada<String> lista = new ListaDupEncadeada<String>();

      System.out.println("Tamanho: " + lista.getTamanho());
      for(int i = 1; i <= 5; i++) {
         int val = lista.insereInicio(String.format("str teste %d", i));
         System.out.println("Inserindo: \"" +lista.consulta(1)+ "\" -> Tamanho: "+val);
         // Erro de compilação ao tentar inserir um objeto que não seja do tipo String
         // lista.insereInicio(i);
      }

      System.out.println("\nTamanho: " + lista.getTamanho());

      System.out.println("\nIterando sobre a lista:\n");
      // Iterador utilizado para percorrer a lista encadeada
      Iterator iterador = lista.iteradorInicio();

      // Contador de posições
      int c = 1;
      // Laço para percorrer a lista utilizando o iterador
      while (iterador.eValido()) {
         // Mostrar o valor armazenado em cada posição da lista
         System.out.println("Posição " + c++ + ": -> " + iterador.dado());
         iterador.proximo();
      }

      System.out.println("");
      for(int i = 1; i <= 5; i++) {
         System.out.println("Removido do início: \"" + lista.removeInicio() +
         "\" -> Tamanho: " + lista.getTamanho());
      }
   }
}
```

Listagem 10. Resultado do teste com a classe de lista encadeada genérica.

```
Tamanho: 0
Inserindo: "str teste 1" -> Tamanho: 1
Inserindo: "str teste 2" -> Tamanho: 2
Inserindo: "str teste 3" -> Tamanho: 3
Inserindo: "str teste 4" -> Tamanho: 4
Inserindo: "str teste 5" -> Tamanho: 5

Tamanho: 5

Iterando sobre a lista:

Posição 1: -> str teste 5
Posição 2: -> str teste 4
Posição 3: -> str teste 3
Posição 4: -> str teste 2
Posição 5: -> str teste 1

Removido do início: "str teste 5" -> Tamanho: 4
Removido do início: "str teste 4" -> Tamanho: 3
Removido do início: "str teste 3" -> Tamanho: 2
Removido do início: "str teste 2" -> Tamanho: 1
Removido do início: "str teste 1" -> Tamanho: 0
```

Superclasses como parâmetros de tipo

Pelo resultado apresentado no exemplo anterior, fica evidente a melhora na segurança do código ao se utilizar uma classe genérica para implementar uma estrutura de dados. Entretanto, surge uma dúvida: e se for necessário armazenar objetos distintos na lista, é possível? Sim, mas desde que os objetos estejam relacionados em uma hierarquia. Deste modo, se existir uma classe ou uma interface que represente todos os objetos a serem armazenados, basta que se utilize esta superclasse, ou a interface, no momento de instanciar a lista, que a classe genérica tratará corretamente todos os objetos, cada um como sendo uma instância da superclasse, ou da interface informada.

Nas **Listagens 11** a **13** são apresentadas as classes de uma hierarquia, com a interface **FormaGeometrica** no topo e as classes **Quadrado** e **Retangulo** a implementando. Esta interface define que todas as figuras geométricas devem implementar os métodos **float perimetro()** e **float area()**. A partir disso, qualquer figura geométrica poderá ser consultada a respeito de sua área e de seu perímetro.

Para a classe **Quadrado**, vide **Listagem 12**, é definido um atributo que representa o comprimento de seu lado, utilizado nos métodos para calcular o valor do perímetro e da sua área, conforme solicitado pela interface de formas geométricas. Já para a classe **Retangulo**, apresentada na **Listagem 13**, são definidos dois atributos: um representando a sua largura e outro à sua altura. Ambos utilizados nos métodos definidos pela interface para calcular o perímetro e a área da figura geométrica correspondente.

Com esta hierarquia de classes definida, é possível então instanciar uma lista encadeada de formas geométricas para armazenar tanto objetos do tipo **Quadrado** quanto objetos do tipo **Retangulo**, informando como tipo para lista a interface **FormaGeometrica**, como expõe o código da **Listagem 14**. Neste programa de teste, objetos das classes **Quadrado** e **Retangulo** são inseridos alternadamente na lista. Feito isso, utiliza-se um objeto **Iterator** para que todos os elementos sejam consultados e os seus perímetros calculados e exibidos no console, e na sequência, reinicializando o **Iterator**, novamente todos os objetos da lista são consultados para exibir o valor de suas áreas. Para encerrar o teste, todos os elementos da lista são removidos e sua representação textual é apresentada no console, conforme o resultado exibido na **Listagem 15**.

No código da **Listagem 14** os objetos retornados pelos métodos de navegação do **Iterator** precisam de um cast, ou seja, é necessário informar ao compilador o tipo de objeto que está armazenado na lista. Isto acontece tanto no laço para mostrar os perímetros quanto no laço para mostrar as áreas das formas geométricas, da seguinte forma: **((FormaGeometrica) iterador.dado())**.

O cast é necessário porque nem a interface **Iterator**, e nem a sua implementação, a classe **IteratorLista**, foram alteradas para utilizar o tipo de objeto **<E>** especificado como parâmetro.

Listagem 11. Interface comum para todas as figuras geométricas.

```
// Uma interface simples para objetos que representam formas geométricas.
// Para todos os objetos classificados como formas geométricas, os métodos que
// retornam os valores de seus perímetros e de suas áreas devem estar implementados.
public interface FormaGeometrica {
   // Retorna o valor do perímetro da forma geométrica
   public float perimetro();
   // Retorna o valor da área da forma geométrica
   public float area();
}
```

Listagem 12. Classe Quadrado. Implementa a interface FormaGeometrica e possui um atributo representando o comprimento do lado.

```
// Implementação da interface FormaGeometrica que representa um
// Quadrado utilizando um atributo com o valor do comprimento do lado.
public class Quadrado implements FormaGeometrica {
   // Comprimento do lado do quadrado
   private float lado;

   // Retorna o valor do lado do quadrado
   public float getLado() { return lado; }

   // Altera o valor do lado do quadrado
   public void setLado(float lado) { this.lado = lado; }

   // Construtor
   public Quadrado(float lado) { this.lado = lado; }

   // Retorna o valor do perímetro da forma geométrica
   @Override
   public float perimetro() { return lado * 4.0f; }

   // Retorna o valor da área da forma geométrica
   @Override
   public float area() { return lado * lado; }

   // Retorna a representação textual do Quadrado
   @Override
   public String toString() { return String.format("Quadrado com lado de %.2f", lado); }
}
```

Herança com classes genéricas

Para resolver o problema exposto anteriormente, não é suficiente alterar a classe que implementa a interface **Iterator**.

Isto porque ao utilizá-la para navegar na lista, invocando, por exemplo, o método **Iterator.dado()** ou o método **Iterator.proximo()**, a referência que é utilizada é para um objeto do tipo **Iterator**, ou seja, mesmo que internamente a classe **ListaDupEncadeada** instancie e trabalhe com um objeto do tipo **IteratorLista**, no código cliente, que utilizar a lista, não há como trabalhar com esta implementação da interface **Iterator**, pois a mesma é uma classe interna e privada, conforme o código da **Listagem 8**. Esta arquitetura, que corresponde ao encapsulamento das características internas da lista encadeada, é adotada justamente para proteger o seu conteúdo de manipulações externas indevidas, prevenindo erros e facilitando a manutenção do código.

Listagem 13. Classe Retangulo. Implementa a interface FormaGeometrica e possui dois atributos representando o comprimento e a largura.

```
// Implementação da interface FormaGeometrica que representa um
//Retangulo utilizando dois atributos
// com os valores dos comprimentos de altura e largura.
public class Retangulo implements FormaGeometrica
{
  // Comprimento da largura do retângulo
  private float largura;
  // Comprimento da altura do retângulo
  private float altura;

  // Retorna o valor da largura do retângulo
  public float getLargura() { return largura; }

  // Altera o valor da largura do retângulo
  public void setLargura(float largura) { this.largura = largura; }

  // Retorna o valor da altura do retângulo
  public float getAltura() { return altura; }

  // Altera o valor da altura do retângulo
  public void setAltura(float altura) { this.altura = altura; }

  // Construtor
  public Retangulo(float largura, float altura) {
    this.largura = largura;
    this.altura = altura;
  }

  // Retorna o valor do perímetro da forma geométrica
  @Override
  public float perimetro() { return 2.0f * largura + 2.0f * altura; }

  // Retorna o valor da área da forma geométrica
  @Override
  public float area() { return largura * altura; }

  // Retorna a representação textual do Retangulo
  @Override
  public String toString() {
    return String.format("Retângulo com largura de %.2f e altura de %.2f.", largura, altura);
  }
}
```

Listagem 14. Classe de teste da lista encadeada genérica que armazena formas geométricas.

```
public class TesteListaFiguraGeometrica {
  public static void main(String[] args) {
    // Lista encadeada genérica para armazenar FormaGeometrica
    ListaDupEncadeada<FormaGeometrica> lista = new ListaDupEncadeada
    <FormaGeometrica>();
    int val;

    // Mostra o tamanho atual da lista
    System.out.println("Tamanho: " + lista.getTamanho());

    // Laço de inserção
    for(int i = 1; i <= 3; i++) {
      // Instancia e insere um Quadrado no início da lista
      val = lista.insereInicio(new Quadrado(2.25f * i));
      // Mostra o conteúdo da primeira posição da lista
      System.out.println("Inserindo: \""+lista.consulta(1)+"\" -> Tamanho: "+val);
      // Instancia e insere um Retangulo no início da lista
      val = lista.insereInicio(new Retangulo(3.567f * i, 5.951f + i));
      // Mostra o conteúdo da primeira posição da lista
      System.out.println("Inserindo: \""+lista.consulta(1)+"\" -> Tamanho: "+val);
    }

    // Mostra o tamanho atual da lista
    System.out.println("\nTamanho: " + lista.getTamanho());

    System.out.println("\nIterando sobre a lista:\n");
    System.out.println("Perímetros:");

    // Iterador utilizado para percorrer a lista encadeada
    Iterator iterador = lista.iteradorInicio();

    // Contador de posições
    int c = 1;

    // Laço para percorrer a lista utilizando o iterador
    while (iterador.eValido()) {
      // Mostra o perímetro de cada FiguraGeometrica armazenada na lista
      // É necessário realizar um cast porque a interface Iterator não conhece
      // o tipo que está armazenado na lista
      System.out.println("Posição " + c++ + ": -> \t" + ((FormaGeometrica)iterador.
      dado()).perimetro());
      iterador.proximo();
    }

    System.out.println("Áreas:");
    // Iterador utilizado para percorrer a lista encadeada
    iterador = lista.iteradorInicio();

    // Contador de posições
    c = 1;
    // Laço para percorrer a lista utilizando o iterador
    while (iterador.eValido()) {
      // Mostra a área de cada FiguraGeometrica armazenada na lista
      System.out.println("Posição " + c++ + ": -> \t" + ((FormaGeometrica)iterador.
      dado()).area());
      iterador.proximo();
    }

    System.out.println("");
    // Laço para remover todos os elementos da lista
    while(lista.getTamanho() > 0) {
      System.out.println("Removido do início: \"" + lista.removeInicio() +
      "\" -> Tamanho: " + lista.getTamanho());
    }
  }
}
```

Listagem 15. Resultado do teste com a classe de lista encadeada genérica e a hierarquia de formas geométricas.

```
Tamanho: 0
Inserindo: "Quadrado com lado de 2,25." -> Tamanho: 1
Inserindo: "Retângulo com largura de 3,57 e altura de 6,95." -> Tamanho: 2
Inserindo: "Quadrado com lado de 4,50." -> Tamanho: 3
Inserindo: "Retângulo com largura de 7,13 e altura de 7,95." -> Tamanho: 4
Inserindo: "Quadrado com lado de 6,75." -> Tamanho: 5
Inserindo: "Retângulo com largura de 10,70 e altura de 8,95." -> Tamanho: 6

Tamanho: 6

Iterando sobre a lista:

Perímetros:
Posição 1: ->          39.304
Posição 2: ->          27.0
Posição 3: ->          30.17
Posição 4: ->          18.0
Posição 5: ->          21.036
Posição 6: ->          9.0
Áreas:
Posição 1: ->          95.78465
Posição 2: ->          45.5625
Posição 3: ->          56.722435
Posição 4: ->          20.25
Posição 5: ->          24.794218
Posição 6: ->          5.0625

Removido do início: "Retângulo com largura de 10,70 e altura de 8,95." ->
Tamanho: 5
Removido do início: "Quadrado com lado de 6,75." -> Tamanho: 4
Removido do início: "Retângulo com largura de 7,13 e altura de 7,95." ->
Tamanho: 3
Removido do início: "Quadrado com lado de 4,50." -> Tamanho: 2
Removido do início: "Retângulo com largura de 3,57 e altura de 6,95." ->
Tamanho: 1
Removido do início: "Quadrado com lado de 2,25." -> Tamanho: 0
```

Portanto, para não alterar o encapsulamento da lista, nem modificar o padrão **Iterator**, é necessário que a interface **Iterator** seja alterada de maneira que ela também tome conhecimento do tipo de dado que está armazenado na estrutura de dados; neste exemplo, a lista duplamente encadeada. Esta alteração é apresentada na **Listagem 16**, na qual é incluído o parâmetro de tipo **<D>** na declaração da interface e nas assinaturas dos métodos as referências à classe **Object** são substituídas por referências a objetos do tipo **<D>**.

Uma vez alterada a interface para que receba um parâmetro de tipo, todas as classes que a implementam precisam informar o tipo em sua declaração, ou seja, ao se declarar que uma classe implementa a interface **Iterator<D>**, é necessário também informar qual será o tipo de dado utilizado, por exemplo: **public class Exemplo implements Iterator<String>**. Nesta declaração está sendo informado que a classe **Exemplo** implementa a interface **Iterator<D>** e que o parâmetro de tipo a ser utilizado na implementação da interface é a classe **String**.

No caso da classe privada **IteratorLista**, que está implementada como uma classe interna da classe **ListaDupEncadeada<E>**, o tipo a ser utilizado deve ser o mesmo que o da própria lista, o tipo **<E>**. Assim, a declaração é escrita da seguinte maneira: **private class IteratorLista implements Iterator<E>**. Com isto se especifica que a classe **IteratorLista** implementa a interface **Iterator<D>** e o parâmetro de tipo a ser utilizado em sua implementação é o mesmo que foi recebido pela classe **ListaDupEncadeada<E>**. Portanto, no código de **IteratorLista**, sempre que for necessário referenciar um objeto armazenado na lista, deve-se utilizar o tipo **<E>**. O código resultante destas modificações pode ser verificado na **Listagem 17**.

Listagem 16. Versão genérica da interface Iterator.

```
// Interface que define as operações que podem ser realizadas para navegar
//por uma estrutura de dados, e que recebe como parâmetro o tipo de
// dado <D> que está armazenado nesta estrutura.
public interface Iterator<D> {
   // Retorna o dado associado ao nó atual
   public D dado();
   // Coloca o iterador no próximo elemento da lista e retorna o dado
   //associado ao nó atual
   public D proximo();
   // Coloca o iterador no elemento anterior da lista e retorna o dado
   //associado ao nó atual
   public D anterior();
   // Verifica se existe um nó após o nó atual
   public boolean temProximo();
   // Verifica se existe um nó antes do nó atual
   public boolean temAnterior();
   // Verifica se o iterador é válido
   public boolean eValido();
}
```

Listagem 17. Classe IteratorLista que implementa a versão genérica da interface Iterator.

```
// Implementação da interface Iterator, que define os métodos de navegação
// na lista. A interface Iterator trabalha com o tipo de dado <E>, o mesmo
// que é armazenado na lista encadeada.
private class IteratorLista implements Iterator<E>
{
   // Referência ao nó apontado pelo iterador da lista durante a navegação.
   No          noAtual;
   // Retorna o dado associado ao nó atual. Caso o iterador não seja válido
   // (noAtual é nulo), retorna um objeto nulo.
   public E dado() {
        if (noAtual == null)
        return null;
      E obj = noAtual.dado;
      return obj;
   }
   // Coloca o iterador no próximo elemento da lista e retorna o dado
   // associado ao nó atual. Caso o iterador não seja válido (noAtual é
   // nulo), retorna um objeto nulo.
   public E proximo() {
      if (noAtual == null)
         return null;
      E obj = noAtual.dado;
      noAtual = noAtual.proximo;
      return obj;
   }
   // Coloca o iterador no elemento anterior da lista e retorna o dado
   // associado ao nó atual. Caso o iterador não seja válido (noAtual é
   // nulo), retorna um objeto nulo.
   public E anterior() {
      if (noAtual == null)
         return null;
      E obj = noAtual.dado;
      noAtual = noAtual.anterior;
      return obj;
   }
   // Verifica se existe um nó após o nó atual.
   public boolean temProximo() { ... }
   // Verifica se existe um nó antes do nó atual.
   public boolean temAnterior() { ... }
   // Verifica se o nó atual existe e é válido.
   public boolean eValido() { ... }
}
```

A utilização desta nova versão da lista encadeada não é muito diferente da que foi apresentada na **Listagem 14**, sendo necessário apenas, no momento de declarar o objeto **Iterator**, informar como parâmetro qual é o tipo de dado a ser utilizado, que deve ser o mesmo informado para a classe de lista, como apresentado na **Listagem 18**. Como resultado desta parametrização do **Iterator**, não é mais necessário realizar a conversão de tipo nos objetos devolvidos nas chamadas aos métodos da interface, como o **Iterator.dado()**, que também está nos trechos de código da **Listagem 18**. O resultado produzido com estas alterações é exatamente o mesmo do teste anterior e que foi mostrado na **Listagem 15**.

Listagem 18. Classe de teste da lista encadeada genérica que armazena formas geométricas utilizando a versão genérica da interface Iterator.

```
public class TesteListaFiguraGeometrica
{
  public static void main(String[] args)
  {
    // Lista encadeada genérica para armazenar FormaGeometrica
    ListaDupEncadeada<FormaGeometrica> lista = new ListaDupEncadeada
    <FormaGeometrica>();
    int val;

    ...

    // Iterador utilizado para percorrer a lista encadeada, cujo conteúdo são objetos
    // do tipo FormaGeometrica
    Iterator<FormaGeometrica> iterador = lista.iteradorInicio();

    ...

    while (iterador.eValido()) {
      System.out.println("Posição " + c++ + ": -> \t"+iterador.dado().perimetro());
      iterador.proximo();
    }

    ...

    while (iterador.eValido()) {
      System.out.println("Posição " + c++ + ": -> \t"+iterador.dado().area());
      iterador.proximo();
    }

    ...
  }
}
```

Algumas vezes, diferente do que foi apresentado no exemplo anterior, ao se trabalhar com uma estrutura de dados podem acontecer situações nas quais os dados que devem ser armazenados são produzidos a partir de um processo cujo resultado é atribuído sempre a uma mesma referência. Uma situação similar a esta é apresentada a partir de uma alteração no teste das **Listagens 14** e **18**, resultando no código da **Listagem 19**. Neste novo teste são declarados um objeto **Quadrado** e outro **Retangulo** e no laço de inserção estes objetos são configurados com novos valores e em seguida armazenados na lista. O resultado desta alteração pode ser visto na **Listagem 20**.

Neste caso, apesar de a inserção de dados aparentemente estar correta, as consultas e as remoções da lista mostram sempre os mesmos valores. Isto ocorre porque no momento da inserção foram fornecidas ao método de inserção várias referências ao mesmo objeto, ou seja, apenas uma instância de **Quadrado** e uma de **Retangulo** foram instanciadas, antes do início do laço de inserção. Portanto, quando as chamadas ao método **ListaDupEncadeada.isereInicio()** foram realizadas, o mesmo objeto foi colocado na lista, uma vez para cada iteração do laço **while**.

Listagem 19. Classe de teste reutilizando as declarações dos objetos inseridos.

```
public class TesteListaFiguraGeometrica
{
  public static void main(String[] args)
  {
    // Lista encadeada genérica para armazenar FormaGeometrica
    ListaDupEncadeada<FormaGeometrica> lista = new ListaDupEncadeada
    <FormaGeometrica>();
    int val;

    // Instancia um Quandrado
    Quadrado quadrado = new Quadrado(0.0f);
    // Instancia um Retangulo
    Retangulo retangulo = new Retangulo(0.0f, 0.0f);

    // Mostra o tamanho atual da lista
    System.out.println("Tamanho: " + lista.getTamanho());

    // Laço de inserção
    for(int i = 1; i <= 3; i++) {
      // Configura o valor do lado da Quadrado
      quadrado.setLado(2.25f * i);

      // Configura os valores da altura e largura do Retangulo
      retangulo.setAltura(3.33f * i);
      retangulo.setLargura(5.68f * i);

      // Insere o Quandrado no início da lista
      val = lista.insereInicio(quadrado);
      // Mostra o conteúdo da primeira posição da lista
      System.out.println("Inserindo: \""+lista.consulta(1)+"\" -> Tamanho: "+val);
      // Insere o Retangulo no início da lista
      val = lista.insereInicio(retangulo);
      // Mostra o conteúdo da primeira posição da lista
      System.out.println("Inserindo: \""+lista.consulta(1)+"\" -> Tamanho: "+val);
    }

    ...

  }
}
```

Especificando uma interface mínima a um parâmetro de tipo

A situação descrita no exemplo das **Listagens 19** e **20** pode ser resolvida atribuindo-se mais uma característica à classe **ListaDupEncadeada**, de maneira que ao realizar a inserção de um novo objeto na lista, o próprio método de inserção deve garantir que o objeto inserido seja único, criando uma nova cópia do mesmo. Com este mesmo propósito a classe **Object**, que é superclasse de todas as classes declaradas em Java, define o método **Object.clone()**, porém este método é declarado como sendo **protected**.

Quando se deseja esta característica, o método **Object.clone()** deve ser sobrescrito e sua visibilidade alterada para **public**. Esta configuração, no entanto, não fornece uma garantia de que o

método esteja implementado corretamente para os objetos que serão inseridos na lista.

Uma maneira de garantir que esta funcionalidade esteja disponível é especificar uma superclasse abstrata ou uma interface que declare o método **clone()** com acesso **public**, assegurando que seja possível utilizá-lo na implementação da lista em questão. Isto é feito especificando-se um relacionamento de herança para o parâmetro de tipo da classe que necessita da característica. Neste caso o que se precisa é que os objetos a serem incluídos na lista encadeada obrigatoriamente implementem o método **Object .clone()** ou similar.

Listagem 20. Resultado do teste reutilizando as declarações dos objetos inseridos.

```
Tamanho: 0
Inserindo: "Quadrado com lado de 2,25." -> Tamanho: 1
Inserindo: "Retângulo com largura de 5,68 e altura de 3,33." -> Tamanho: 2
Inserindo: "Quadrado com lado de 4,50." -> Tamanho: 3
Inserindo: "Retângulo com largura de 11,36 e altura de 6,66." -> Tamanho: 4
Inserindo: "Quadrado com lado de 6,75." -> Tamanho: 5
Inserindo: "Retângulo com largura de 17,04 e altura de 9,99." -> Tamanho: 6

Tamanho: 6

Iterando sobre a lista:

Perímetros:
Posição 1: ->          54.059998
Posição 2: ->          27.0
Posição 3: ->          54.059998
Posição 4: ->          27.0
Posição 5: ->          54.059998
Posição 6: ->          27.0
Áreas:
Posição 1: ->          170.22958
Posição 2: ->          45.5625
Posição 3: ->          170.22958
Posição 4: ->          45.5625
Posição 5: ->          170.22958
Posição 6: ->          45.5625

Removido do início: "Retângulo com largura de 17,04 e altura de 9,99." -> Tamanho: 5
Removido do início: "Quadrado com lado de 6,75." -> Tamanho: 4
Removido do início: "Retângulo com largura de 17,04 e altura de 9,99." -> Tamanho: 3
Removido do início: "Quadrado com lado de 6,75." -> Tamanho: 2
Removido do início: "Retângulo com largura de 17,04 e altura de 9,99." -> Tamanho: 1
Removido do início: "Quadrado com lado de 6,75." -> Tamanho: 0
```

Esta obrigatoriedade pode ser conseguida utilizando-se mais uma interface, que declara uma versão pública do método **Object .clone()**. A linguagem Java especifica também que toda classe que necessita criar cópias de si mesma, além de fornecer uma implementação pública do método **clone()**, deve implementar a interface **Cloneable**, definida no pacote **java.lang**. Deste modo, para atender às especificações da linguagem, a nova interface também deve estender **Cloneable**, resultando no código apresentado na **Listagem 21**.

Uma vez definida a interface com o comportamento necessário aos objetos que serão manipulados pela lista, é preciso alterar a declaração da classe **ListaDupEncadeada** para que seja obrigatório que todos os parâmetros de tipo fornecidos implementem a interface **Clonavel** e, consequentemente, forneçam o método público **Clonavel.clone()**. A sintaxe Java que permite especificar a superclasse ou interface de um parâmetro de tipo é o uso da palavra-chave **extends**, indicando que o parâmetro deve ser uma subclasse da classe informada ou implementar a interface especificada, da seguinte forma: **public class ListaDupEncadeada <E extends Clonavel>**.

Listagem 21. Interface utilizada para especificar um comportamento padrão aos objetos que serão armazenados na lista.

```
// Interface que define um método a ser utilizado pelos objetos para criar clones (cópias) de si mesmos
public interface Clonavel extends Cloneable
{
   // Método para criar uma cópia do objeto
   public Clonavel clone();
}
```

O uso desta declaração informa ao compilador Java que a classe **ListaDupEncadeada** recebe um parâmetro de tipo, e que este parâmetro de tipo deve, obrigatoriamente, implementar a interface **Clonavel**. Esta declaração possibilita que os métodos da classe **ListaDupEncadeada** façam uso dos métodos definidos na interface **Clonavel**, que no caso é o método **clone()**.

A partir destas atualizações, é possível no método **ListaDupEncadeada.insere()**, que realiza todas as inserções na lista, clonar os objetos que são enviados para a lista, permitindo então que a reutilização das declarações de objetos no momento de inseri-los na lista não crie várias referências ao mesmo objeto. Para isso, basta que todos os nós instanciados recebam um clone do objeto enviado ao método **insere()**, como está na **Listagem 22**.

A alteração no parâmetro de tipo da lista encadeada inviabiliza o uso da interface **FormaGeometrica** como foi inicialmente concebida, pois neste novo exemplo, ela não implementa o mínimo exigido pela lista, que é o comportamento especificado por **Clonavel**. Para permitir que as formas geométricas possam ser armazenadas na lista, é preciso modificar a interface de acordo, como no código da **Listagem 23**, de modo que as classes que implementam **FormaGeometrica** devam também fornecer o comportamento especificado por **Clonavel**.

Para terminar, é necessário que as classes **Quadrado** e **Retangulo** forneçam suas respectivas implementações para o método **clone()**, para atender as especificações das interfaces **FormaGeometrica** e **Clonavel**, conforme as **Listagens 24** e **25**.

A partir destas modificações na classe **ListaDupEncadeada** e as novas versões das classes da hierarquia para formas geométricas, o teste da **Listagem 19** vai apresentar o mesmo resultado da **Listagem 15**, como era o esperado.

O exemplo demonstrado aqui levou em consideração a possibilidade de incluir, repetidamente, referências a um mesmo objeto, mas as mesmas ideias podem ser utilizadas em casos nos quais se deseja outras funcionalidades como, por exemplo, realizar a

Listagem 22. Classe ListaDupEncadeada que insere clones dos objetos.

```
// Classe que implementa a estrutura de dados Lista Duplamente Encadeada.
// A classe ListaDupEcadeada recebe um parâmetro <E> informando qual é o tipo de
// dado que será armazenado em seus nós. Uma classe para poder ser armazenada na
// lista deve, obrigatoriamente, implementar a interface Clonavel.
public class ListaDupEncadeada<E extends Clonavel>
{
  private class No { ... }
  private class IteratorLista implements Iterator <E> { ... }

  No      inicio;
  No      fim;
  int     tamanho;

  public Iterator<E> iteradorInicio() { ... }
  public Iterator<E> iteradorFim() { ... }
  public ListaDupEncadeada_v06() { ... }
  public int getTamanho() { ... }
  public int insereInicio(E obj) { ... }
  public int insereFim(E obj) { ... }

  // insere um novo nó na posição indicada.
  public int insere(E obj, int pos)
  {
    No no;

    if (obj == null)
      return tamanho;

    // posição solicitada fora do intervalo
    if (pos < 1 || pos > tamanho + 1)
      return tamanho;

    // primeiro elemento - lista vazia - ou (tamanho == 0)
    if (inicio == null)
    {
      no = new No((E)obj.clone());
      inicio = fim = no;
    }
    else
    // já existem elementos na lista
    {
      // inserir no início da lista
      if (pos == 1)
      {
        no = new No((E)obj.clone(), inicio, null);
        inicio.anterior = no;
        inicio = no;
      }
      else
        // inserir no final da lista
        if (pos == tamanho + 1)
        {
          no = new No((E)obj.clone(), null, fim);
          fim.proximo = no;
          fim = no;
        }
        else
        // inserir no meio da lista
        {
          No aux = inicio;
          while (pos > 1)
          {
            aux = aux.proximo;
            pos--;
          }
          // inserir na posição de aux
          no = new No((E)obj.clone(), aux, aux.anterior);
          aux.anterior.proximo = no;
          aux.anterior = no;
        }
    }
    tamanho++;
    return tamanho;
  }
  public Object removeInicio() { ... }
  public Object removeFim() { ... }
  public E remove(int pos) { ... }
  public E consulta(int pos)
}
```

Listagem 23. Nova versão da interface comum para todas as figuras geométricas.

```
// Uma interface simples para objetos que representam formas geométricas e que
// também implementa a interface Clonavel, de modo que as classes que
// implementem esta interface devam implementar também o método para criar
// uma cópia de si mesmo.
// Para todos os objetos classificados como formas geométricas, os métodos que
// retornam os valores de seus perímetros e de suas áreas devem estar implementados.
public interface FormaGeometrica extends Clonavel
{
  // Retorna o valor do perímetro da forma geométrica
  public float perimetro();
  // Retorna o valor da área da forma geométrica
  public float area();
}
```

Listagem 24. Nova versão da classe Quadrado, que implementa também a interface Clonavel.

```
// Implementação da interface FormaGeometrica que representa um Quadrado
// utilizando um atributo com o valor do comprimento do lado, e que também
// implementa a interface Clonavel, de modo que as classes que implementem
// esta interface devam implementar também o método para criar uma cópia
// de si mesmo.
public class Quadrado implements FormaGeometrica
{
  ...
  // Método para criar uma cópia do objeto
  @Override
  public Quadrado clone() {
    Quadrado novo = new Quadrado(this.lado);
    return novo;
  }
}
```

comparação entre objetos já armazenados na estrutura de dados e os que estão sendo inseridos, para a realização da inclusão ordenada de elementos, sendo que neste caso, uma interface especificando um método de comparação deveria ser utilizada.

A utilização de uma classe que implementa uma estrutura de dados, como a lista encadeada, para apresentar os conceitos de classes genéricas facilita o aprendizado, uma vez que é fácil entender que uma lista deva poder armazenar muitos tipos diferentes de dados. Também é necessário que alguma segurança seja fornecida ao programador, pois como o primeiro exemplo demonstrou de uma forma simples e direta, podem ocorrer situações nas quais dados diferentes sejam fornecidos para serem armazenados, não

sendo o desejado, e, portanto, causando falhas no sistema. Por outro lado, a solução deve ser flexível o suficiente para permitir que classes relacionadas entre si possam ser utilizadas, como no caso das formas geométricas.

Listagem 25. Nova versão da classe Retangulo, que implementa também a interface Clonavel.

```
// Implementação da interface FormaGeometrica que representa um Retangulo
// utilizando dois atributos com os valores dos comprimentos de altura e largura,
// e que também implementa a interface Clonavel, de modo que as classes que
// implementem esta interface devam implementar também o método
// para criar uma cópia de si mesmo.
public class Retangulo implements FormaGeometrica
{
   ...

   // Método para criar uma cópia do objeto
   @Override
   public Retangulo clone() {
      Retangulo novo = new Retangulo(this.largura, this.altura);
      return novo;
   }
}
```

Estas situações são as ideais para o uso de classes genéricas, possibilitando assim que o programador escreva apenas uma classe e esta se adapte corretamente aos diferentes tipos de dados que, porventura, sejam necessários na solução final, sem que seja preciso alterar a classe original para adaptá-la aos dados de cada problema no qual a classe de lista será empregada.

É importante lembrar que os exemplos apresentados neste artigo expõem soluções similares a classes que estão disponíveis em APIs, como algumas da própria linguagem Java, como as classes **ArrayList** e **LinkedList** do pacote **java.util**, que implementam a estrutura de lista, ou ainda a classe **Stack** do mesmo pacote, que implementa a estrutura de pilha.

Contudo, para o bom uso das estruturas de dados, o programador deve entender como funcionam, compreender as técnicas e padrões que são utilizados em suas implementações e principalmente quais são as características disponíveis nas classes fornecidas pelas APIs, para tirar melhor proveito das mesmas ou optar por criar a sua própria versão, caso alguma funcionalidade específica não esteja contemplada.

O bom entendimento das características da linguagem são importantes não apenas para aproveitar melhor uma API, mas para reconhecer também quando uma determinada funcionalidade não está presente na mesma, justificando assim a implementação de suas próprias classes de estruturas de dados.

Autor

Miguel Diogenes Matrakas
mdmatrakas@yahoo.com.br
Mestre em Informática pela PUC-PR e doutorando em Métodos Numéricos em Engenharia, pela UFPR (Universidade Federal do Paraná). Trabalha como professor de Java há quatro anos nas Faculdades Anglo-Americano de Foz do Iguaçu.

Como criar relacionamentos entre entidades com Hibernate

Aprenda nesse artigo a mapear associações dos tipos OneToMany e ManyToMany e criar relacionamentos entre entidades com Hibernate

Os profissionais de TI que trabalham com a orientação a objetos através do Java geralmente optam pelo JPA ou Hibernate ao implementar a persistência em suas aplicações. Neste artigo, daremos vez ao Hibernate, solução que simplifica e facilita o armazenamento e a recuperação de informações e que, assim como a JPA, adota o Mapeamento Objeto-Relacional (*Object/Relational Mapping* – ORM), uma técnica que abstrai as diferenças entre o modelo relacional e o modelo OO.

As principais vantagens do ORM são viabilizar a persistência dos dados, permitir que a aplicação permaneça totalmente orientada a objetos e possibilitar que possíveis mudanças na base de dados impliquem em um menor impacto sobre a aplicação, tendo em vista que apenas os objetos relacionados com as tabelas alteradas precisarão de mudanças.

Um dos principais recursos que o Hibernate oferece é a opção de criar mapeamentos entre modelos de objetos e modelos relacionais através de anotações. A partir desse mecanismo podemos especificar, no código, as mais diversas associações existentes dentro de um banco de dados.

Dentre as vantagens, o uso de anotações nos permite um código mais intuitivo do que os arquivos baseados em XML e ainda supre algumas das suas limitações, tais como: facilidade de visualização das configurações, checagem do código da aplicação em tempo de compilação e possibilidade de *refactoring*.

Antes de iniciar a codificação, no entanto, é comum a desenvolvedores realizar a modelagem do banco de dados. Nesse momento, muitos consideram como etapa mais complexa a necessidade de utilizar relacionamentos dos tipos "1:N" (um-para-muitos) e "N:N" (muitos-para-muitos). Essas opções, bastante presentes no mundo real, normalmente são um pouco mais difíceis de especificar porque envolvem um número maior de variáveis a serem consideradas, como a necessidade de definir mais uma tabela, o modo como os dados devem ser carregados do banco, entre outras.

Fique por dentro

Este artigo é útil por apresentar como estabelecer relacionamentos múltiplos entre entidades com o Hibernate, requisito presente em qualquer aplicação que envolva persistência de dados. Para isso, aprenderemos de forma didática e simples como funcionam as associações OneToMany e ManyToMany, importantes opções para a persistência de dados. Primeiramente, abordaremos cada um desses relacionamentos através de uma análise geral. Em seguida, realizaremos a configuração de um ambiente de desenvolvimento com Maven, Eclipse e MySQL e, por fim, ensinaremos a modelagem de um cenário que faz uso de cada uma das opções mencionadas.

Com base nisso, começaremos este artigo abordando as características dos relacionamentos um-para-muitos e muitos-para-muitos. Em seguida, veremos como modelar um sistema de cadastro considerando cada uma das associações mencionadas. Para tanto, serão empregados, além da linguagem 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 para abstrair a camada de persistência, viabilizando a interface entre a aplicação e o MySQL.

Associações um-para-muitos

O relacionamento de cardinalidade um-para-muitos (ou muitos-para-um), representado pela notação 1:N (lê-se um para N), é

usado quando uma entidade A pode se relacionar com uma ou mais entidades B e B pode estar relacionado a apenas uma entidade A. Vejamos um exemplo: um time de futebol pode ter vários jogadores, mas um jogador só pode estar relacionado a um time. A **Figura 1** demonstra mais um exemplo desse tipo de relacionamento. Neste caso, um Setor pode ter vários Empregados trabalhando nele, e um Empregado só pode estar vinculado a um Setor.

Ao adotar o Hibernate, o uso desse tipo de relacionamento em nossos projetos implica na utilização das anotações **@OneToMany** e **@ManyToOne**, como veremos adiante.

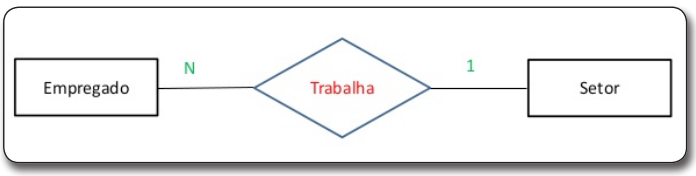

Figura 1. Associação do tipo 1:N

Associações muitos-para-muitos

O relacionamento de cardinalidade muitos-para-muitos, representado pela notação N:N (lê-se N para N), é usado quando uma entidade A pode se relacionar com uma ou mais entidades B e uma entidade B pode se relacionar com uma ou mais entidades A. Para simplificar a compreensão, vejamos as seguintes situações:

- Um usuário do sistema pode fazer parte de vários grupos de usuários e em um grupo pode haver diversos usuário;
- Um autor pode publicar vários livros e um livro pode ser escrito por vários autores.

Para estabelecer esse tipo de relacionamento, devemos ter três tabelas, sendo a terceira responsável por manter a ligação entre as outras duas. Para isso, é preciso que as duas primeiras tabelas contenham uma coluna que seja chave primária.

A **Figura 2** mostra um exemplo desse tipo de relacionamento. Neste caso, um Empregado pode desenvolver um ou mais Projetos e um Projeto é desenvolvido por um ou mais Empregados.

As colunas que são chaves primárias na primeira e na segunda tabela devem aparecer na terceira como chaves estrangeiras. Assim, esta tabela terá duas chaves estrangeiras, que formam uma chave primária composta.

A anotação que representa esse tipo de associação no Hibernate é a **@ManyToMany**.

Figura 2. Associação do tipo N:N

Configuração do ambiente

Após conhecer as principais informações sobre os relacionamentos que abordaremos, vamos partir para a parte prática e desenvolver uma aplicação que possibilite ao leitor aprender como implementar cada uma destas opções a partir da utilização do framework Hibernate. Antes de iniciar o desenvolvimento, vamos configurar o ambiente que utilizaremos para implementar a aplicação exemplo.

Preparando o ambiente de desenvolvimento

Como ambiente de desenvolvimento, optamos por utilizar o Eclipse, por se tratar da solução mais flexível e utilizada no mercado. Na seção **Links** encontra-se o endereço para download desta IDE. Aqui, 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. Na seção **Links** também está disponível o endereço onde pode ser obtido este JDK.

Após baixar o arquivo do Eclipse adequado ao seu sistema, descompacte-o no local de sua preferência. Em nosso caso, 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 da construção (build) de projetos Java. De acordo com o próprio site do Maven, este tem os seguintes objetivos: prover um padrão para o desenvolvimento de aplicações; impor uma estrutura para organizar o projeto; controlar a versão e artefatos; e possibilitar o compartilhamento de bibliotecas entre vários projetos. Como grande destaque, a principal facilidade ofertada pelo Maven é o gerenciamento de dependências e este será o nosso motivador para adotá-lo em parceria com a IDE Eclipse.

Na seção **Links** encontra-se 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 ela. Por fim, descompacte esse arquivo.

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

Para visualizarmos esta instalação, com o Eclipse aberto, acesse o menu *Window > Preferences* e escolha a opção *Maven > Installations*, conforme a **Figura 3**. Perceba a presença da versão "embarcada". Porém, vamos utilizar a versão do Maven que baixamos, por ser uma versão mais atual.

Para adicionar o Maven ao Eclipse, ainda nesta janela, 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 4**, e esta versão é adicionada ao IDE. Sendo assim, apenas 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: os comandos em SQL, como indica o nome; excelente desempenho e estabilidade; suporte a praticamente qualquer sistema operacional (portabilidade); interface gráfica com boa usabilidade; etc. Na seção **Links** encontra-se o endereço para download e mais informações. A versão adotada neste artigo foi a Community Server 5.6.22.

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

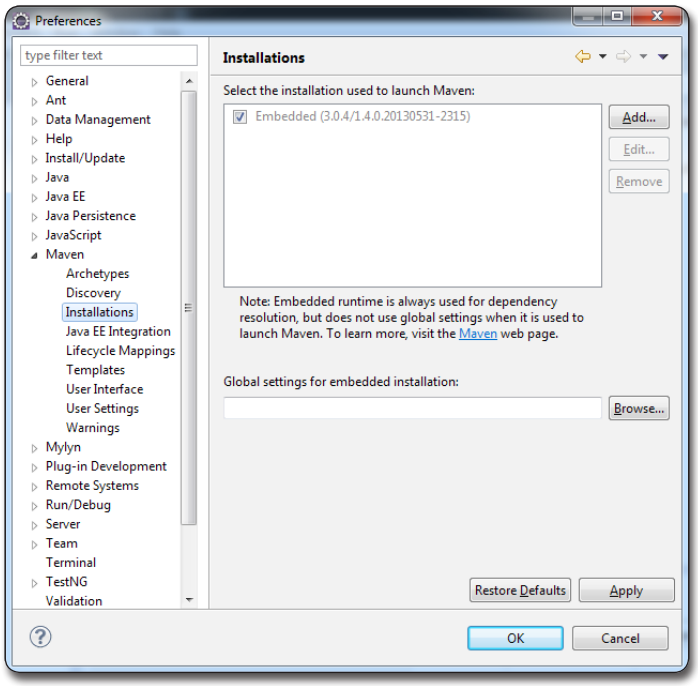

Figura 3. Instalação oferecida do Maven no Eclipse

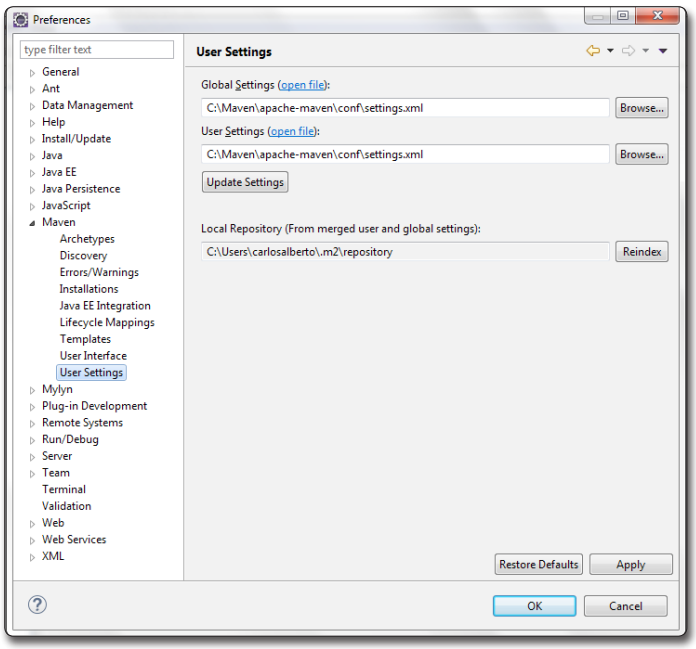

Figura 4. Adicionando o Maven ao Eclipse

1. Clique duas vezes sobre o arquivo. Com isso, uma nova janela será exibida, solicitando que o usuário aguarde enquanto o sistema operacional configura o instalador do SGBD;
2. Após o sistema operacional configurar o instalador, 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* para podermos customizar a instalação. 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 5**) pede para definir a senha do administrador. Portanto, informe uma senha e clique em *Next*;

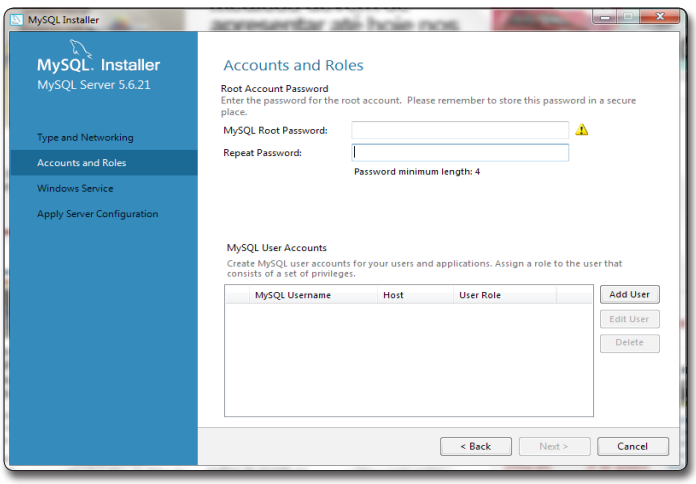

Figura 5. Definindo a senha do administrador

10. Feito isso, a próxima tela permite ao usuário configurar o MySQL como um serviço do Sistema Operacional Windows. Para tal, 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 6**;
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 (vide **Figura 7**). 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*.

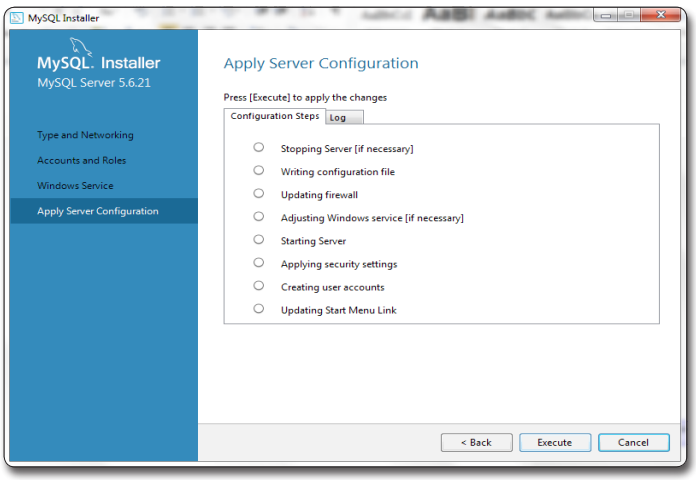

Figura 6. Aplicando as configurações definidas no servidor

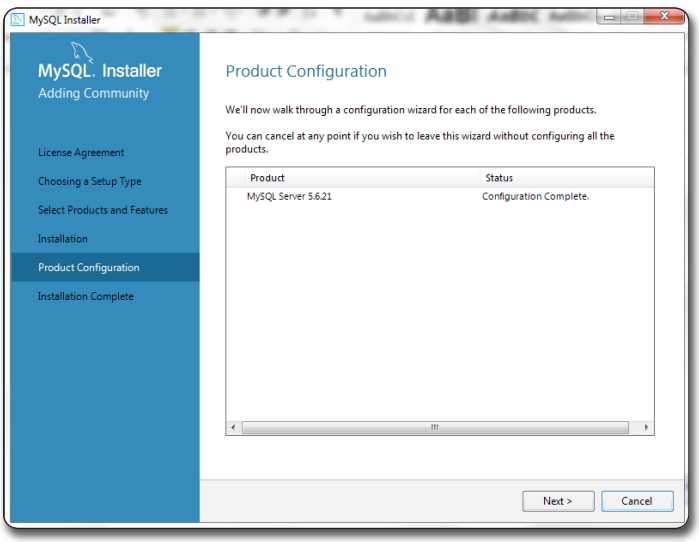

Figura 7. Configuração do MySQL finalizada

Modelagem do Sistema

Agora que temos o banco de dados instalado, assim como o Eclipse e o Maven integrados, vamos partir para a modelagem do software.

Como vamos construir um exemplo simples, suponha que no domínio do nosso sistema existam as seguintes entidades: **Funcionario**, **Departamento**, **Projeto** e **Trabalha**. Neste cenário, um **Funcionario** pode trabalhar em vários **Projetos**, assim como um **Projeto** pode possuir vários **Funcionarios**. Deste modo, a tabela *trabalha* aparece como resultado da relação muitos-para-muitos entre as tabelas *funcionario* e *projeto* e armazenará a chave primária de cada uma delas.

Outra relação que temos ocorre entre as tabelas *departamento* e *funcionario*. Como um **Departamento** pode ter vários **Funcionarios** e um **Funcionario** pode estar associado a apenas um **Departamento**, esse relacionamento é do tipo um-para-muitos.

As informações que cada **Departamento** carregará são: código e nome. Já um **Funcionario** conterá um código, um nome, um cargo, um salário, um CPF e um endereço. Por fim, um **Projeto** possui os atributos código, nome, data de início e data de término.

A **Figura 8** mostra o relacionamento existente (diagrama entidade relacionamento) entre as tabelas do banco de dados.

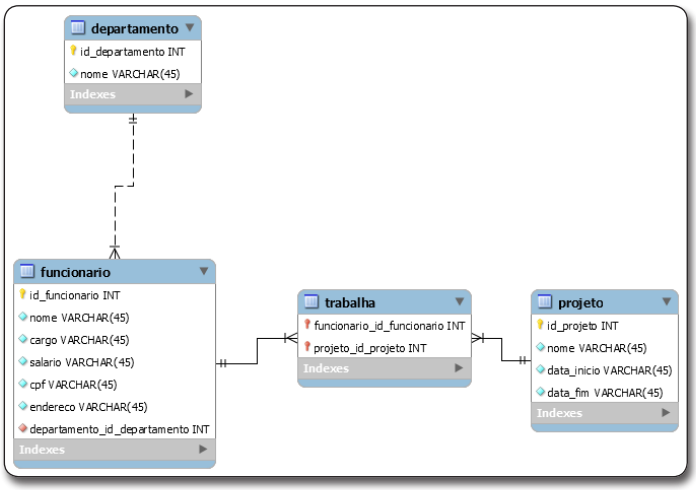

Figura 8. Diagrama Entidade Relacionamento

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.

A tabela *trabalha*, criada a partir da linha 33, estabelece a ligação entre as tabelas *departamento* e *projeto*. Ela contém um par de colunas, *id_funcionario_fk* (linha 35) e *id_projeto_fk* (linha 36), onde cada uma dessas aponta para uma chave primária de uma das tabelas que fazem parte do relacionamento.

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* > *Maven 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 opções:

- **Group ID** (identificador da empresa/grupo ao qual o projeto pertence): **br.com.devmedia**;
- **Artifact ID** (nome do projeto): **hibernate-relations**;
- **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 departamento
05.  (
06.    id_departamento INT NOT NULL auto_increment,
07.    nome  VARCHAR(100) NOT NULL,
08.    PRIMARY KEY (id_departamento)
09.  );
10.
11. CREATE TABLE funcionario
12.  (
13.    id_funcionario  INT NOT NULL auto_increment,
14.    nome            VARCHAR(100) NOT NULL,
15.    cargo           VARCHAR(30) NOT NULL,
16.    salario         DOUBLE NOT NULL,
17.    cpf             VARCHAR(20) NOT NULL,
18.    endereco        VARCHAR (100) NOT NULL,
19.    id_departamento_fk INT,
20.    PRIMARY KEY (id_funcionario),
21.    CONSTRAINT fk_departamento FOREIGN KEY (id_departamento_fk)
       REFERENCES departamento (id_departamento)
22.  );
23.
24. CREATE TABLE projeto
25.  (
26.    id_projeto  INT NOT NULL auto_increment,
27.    nome        VARCHAR(100) NOT NULL,
28.    data_inicio DATE,
29.    data_fim    DATE,
30.    PRIMARY KEY (id_projeto)
31.  );
32.
33. CREATE TABLE trabalha
34.  (
35.    id_funcionario_fk INT,
36.    id_projeto_fk    INT,
37.    PRIMARY KEY (id_funcionario_fk, id_projeto_fk),
38.    CONSTRAINT fk_projeto FOREIGN KEY (id_projeto_fk)
       REFERENCES projeto (id_projeto),
39.    CONSTRAINT fk_funcionario FOREIGN KEY (id_funcionario_fk)
       REFERENCES funcionario (id_funcionario)
40.  );
```

Após preencher todos os dados mencionados, clique em *Finish*.

Perceba 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 com as configurações do Maven. O projeto passará, então, a ter a estrutura característica do Maven, como demonstra a **Figura 9**.

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 testes 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.

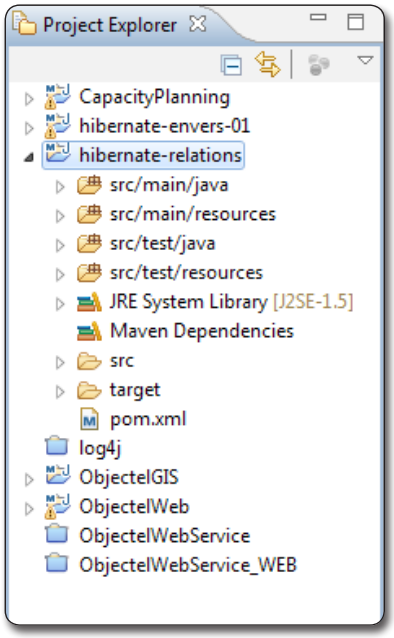

Figura 9. Estrutura do projeto Maven

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: o Hibernate Core e o driver do MySQL. Em seguida, clique em *Ok*.

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

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-relations</artifactId>
05.   <version>0.0.1-SNAPSHOT</version>
06.   <dependencies>
07.     <dependency>
08.       <groupId>org.hibernate</groupId>
09.       <artifactId>hibernate-core</artifactId>
10.       <version>4.3.8.Final</version>
11.     </dependency>
12.     <dependency>
13.       <groupId>mysql</groupId>
14.       <artifactId>mysql-connector-java</artifactId>
15.       <version>5.1.34</version>
16.     </dependency>
17.   </dependencies>
18. </project>
```

Configuradas as dependências, clique com o botão direito do mouse sobre o projeto e selecione novamente a opção *Maven* > *Update Project*. Com isso, todas as bibliotecas definidas serão adicionadas.

Implementando a relação 1:N

Nosso próximo passo é criar e mapear as entidades. 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 do banco de dados.

Classes desse tipo são classes Java simples, definidas como POJOs, contendo todos os seus atributos encapsulados através dos métodos de acesso get e set e também disponibilizando 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-relations*, 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**.

Listagem 3. Código da classe Funcionario.

```
01. package com.jm.entidade;
02.
03. //imports omitidos
04.
05. @Entity
06. @Table(name = "funcionario", schema = "empresabd")
07. public class Funcionario implements Serializable {
08.
09.     private static final long serialVersionUID = 1L;
10.
11.     @Id
12.     @GeneratedValue(strategy = GenerationType.IDENTITY)
13.     @Column(name = "id_funcionario")
14.     private int id;
15.
16.     @Column(name = "nome")
17.     private String nome;
18.
19.     @Column(name = "cargo")
20.     private String cargo;
21.
22.     @Column(name = "salario")
23.     private float salario;
24.
25.     @Column(name = "cpf")
26.     private String cpf;
27.
28.     @Column(name = "endereco")
29.     private String endereco;
30.
31.     @ManyToOne(fetch = FetchType.LAZY)
32.     @JoinColumn(name = "id_departamento_fk")
33.     private Departamento departamento;
34.
35.     //métodos get e set omitidos...
36. }
```

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

Já a anotação **@id**, presente na linha 11, é utilizada para indicar qual atributo da classe anotada com **@Entity** será mapeado como chave primária da tabela. Uma anotação que geralmente acompanha **@id** é a **@GeneratedValue**, presente na linha 12. 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

Além destas, podemos perceber as anotações **@Table** e **@Column**, que são usadas para personalizar o nome das tabelas e das colunas, respectivamente, bastando, para isso, informar o nome entre parênteses.

A anotação **@ManyToOne**, vista na linha 31, sinaliza a existência de um relacionamento entre tabelas. Neste código, significa que múltiplas instâncias de uma entidade **Funcionario** podem ser relacionadas a uma simples instância de uma entidade **Departamento**. Ainda nesta anotação, a propriedade **FetchType.LAZY** tem o objetivo de tornar a aplicação mais rápida e performática, poupando a memória e outros recursos do servidor. Sua utilização faz com que determinados objetos sejam carregados do banco de dados apenas quando forem solicitados.

Na linha 32, **@JoinColumn** é utilizada para nomearmos a coluna que possui a chave-estrangeira requerida por esta associação. Se nada for especificado, será utilizado o nome do campo. Em nosso exemplo, a coluna **id_departamento_fk** foi definida como chave-estrangeira da tabela *funcionario*.

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

A diferença dessa classe para a classe **Funcionario** é a presença, na linha 19, da anotação **@OneToMany** e do atributo **mappedBy**. Essa anotação indica quem é o dono do relacionamento. Em uma associação bidirecional, como a que estamos utilizando (onde qualquer um dos lados da relação consegue acessar o outro), temos que definir o lado mais forte, que geralmente é o lado *many*. Nesse caso, como um departamento pode ter muitos funcionários, e um funcionário pode estar associado a apenas um departamento, o dono da relação é a entidade *departamento*.

O atributo **mappedBy** informa o nome da variável de instância proprietária do relacionamento.

Configurando a aplicação

Antes de mapearmos o relacionamento de cardinalidade N:N, vamos primeiro testar a implementação do relacionamento 1:N que acabamos de realizar. Para isso, precisamos configurar a aplicação para informar ao Hibernate onde está nosso banco de dados e como se conectar a ele.

As configurações do Hibernate, por padrão, devem ser adicionadas em um arquivo XML, de nome *hibernate.cfg.xml*.

Para criar este arquivo, clique com o botão direito sobre o projeto, selecione *Novo* > *Documento XML* e o chame de *hibernate.cfg*. Seu conteúdo deve ser semelhante ao da **Listagem 5**.

Listagem 4. Código da classe Departamento.

```
01. package com.jm.entidade;
02.
03. //imports omitidos
04.
05. @Entity
06. @Table(name = "departamento", schema = "empresabd")
07. public class Departamento implements Serializable {
08.
09.    private static final long serialVersionUID = 1L;
10.
11.    @Id
12.    @GeneratedValue(strategy = GenerationType.IDENTITY)
13.    @Column(name = "id_departamento")
14.    private int id;
15.
16.    @Column(name = "nome")
17.    private String nome;
18.
19.    @OneToMany(fetch = FetchType.LAZY, mappedBy = "departamento")
20.    private Set<Funcionario> funcionarios;
21.
22.    //métodos get e set omitidos...
23. }
```

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
           .MySQLDialect</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.Funcionario" />
12.       <mapping class="com.jm.entidade.Departamento" />
13.    </session-factory>
14. </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 a tabelas do banco de dados.

Conexão com o banco de dados

Após as configurações do banco de dados, vamos implementar 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 com.jm.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.
22.  return sessionFactory;
23. }
24.}
```

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**. Estas instâncias são utilizadas para viabilizar a execução das tarefas de persistência do framework.

Voltando à classe **HibernateUtil**, esta invoca o método estático **getSessionFactory()**, visto na linha 9, para oferecer sempre a mesma instância de **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, construção esta que é feita com base no arquivo de configuração do Hibernate.

Persistindo dados no banco

Com as classes mapeadas, o arquivo de configuração e a classe auxiliar implementados, codificaremos as classes que serão responsáveis por viabilizar todas as operações de persistência disponibilizadas pela aplicação. Para manter uma boa organização do código, crie essas classes no pacote **com.jm.acessobd**.

Para definir esse pacote, clique com o botão direito do mouse sobre o projeto *hibernate-relations*, escolha *New* > *Package* e informe **com.jm.acessobd** como nome.

Feito isso, clique sobre este pacote com o botão direito, escolha *New* > *Class* e dê o nome à classe de **FuncionarioDAO**. O código fonte desta deve ficar semelhante ao apresentado na **Listagem 7**.

Na linha 9, dentro do construtor de **FuncionarioDAO**, o método **getSessionFactory()** é invocado para criação de uma **SessionFactory**. Como este objeto é pesado e lento de se criar, por armazenar os mapeamentos e configurações do Hibernate, deve ser criado apenas uma vez.

Já na linha 12, o método **adicionarFuncionario()** recebe como parâmetro os dados de um novo funcionário e realiza a inserção deste no banco de dados. Para isso, na linha 13 é obtida uma sessão a partir de um **SessionFactory**. Um objeto do tipo **Session** pode ser compreendido 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, estes métodos possuem a mesma estrutura em

Listagem 7. Código da classe FuncionarioDAO.

```
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(Funcionario.class, codigoFuncionario);
56.       funcionario.setSalario(salario);
57.       session.update(funcionario);
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 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, codigoFuncionario);
75.       session.delete(funcionario);
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. }
```

relação ao método já explicado. As diferenças encontram-se nas linhas 37, 57 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 57, realiza a atualização do objeto passado como parâmetro, assim como o método **delete()**, chamado na linha 75, remove o funcionário passado como parâmetro da base de dados.

Depois de **FuncionarioDAO**, devemos criar a classe **DepartamentoDAO**, cujo código é apresentado na **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 **Departamento**.

Testando a implementação 1:N

Nesse momento, com todo o projeto estruturado, podemos partir para os testes com o intuito de ver o resultado da implementação dentro do banco de dados. Para isso, vamos criar a classe de testes **Teste1N** (vide **Listagem 9**). Seu objetivo é associar alguns funcionários a determinado departamento da empresa. Visando manter a organização do código, crie esta classe no pacote **com.jm.teste**.

Na linha 13, uma instância da classe **Departamento** é inserida no banco de dados através da operação **adicionarDepartamento()**. Em seguida, entre as linhas 15 e 16, são criados dois objetos do tipo **Funcionario**, que são vinculados ao departamento definido anteriormente por meio das linhas 30 e 31. Por fim, estes funcionários são inseridos no banco de dados ao executar as linhas 34 e 35.

Resultado da execução

Finalizada a execução da classe de teste, o banco de dados terá a situação mostrada na **Figura 10**. Ao realizar uma consulta na tabela *departamento*, podemos confirmar a criação do departamento "Recursos Humanos", cujo identificador é igual a 1.

Listagem 8. Código da classe DepartamentoDAO.

```
01.  package com.jm.acessobd;
02.
03.  //imports omitidos
04.
05.  public class DepartamentoDAO {
06.     private static SessionFactory factory;
07.
08.     public DepartamentoDAO() {
09.        factory = HibernateUtil.getSessionFactory();
10.    }
11.
12.     public Integer adicionarDepartamento(Departamento departamento) {
13.        Session session = factory.openSession();
14.        Transaction tx = null;
15.        Integer cod_dep = null;
16.        try {
17.           tx = session.beginTransaction();
18.           cod_dep = (Integer) session.save(departamento);
19.           tx.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_dep;
29.     }
30.
31.     public List<Departamento>  listarDepartamentos() {
32.        Session session = factory.openSession();
33.        Transaction tx = null;
34.        List<Departamento> departamentos = null;
35.        try {
36.           tx = session.beginTransaction();
37.           departamentos = session.createCriteria(Departamento.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 departamentos;
48.     }
49.
50.     public void atualizarDepartamento(Integer cod, String nome) {
51.        Session session = factory.openSession();
52.        Transaction tx = null;
53.        try {
54.           tx = session.beginTransaction();
55.           Departamento departamento = (Departamento)
              session.get(Departamento.class, cod);
56.           departamento.setNome(nome);
57.           session.update(departamento);
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 apagarDepartamento(Integer cod) {
70.        Session session = factory.openSession();
71.        Transaction tx = null;
72.        try {
73.           tx = session.beginTransaction();
74.           Departamento departamento = (Departamento)
              session.get(Departamento.class, cod);
75.           session.delete(departamento);
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.  }
```

Já na tabela *funcionario*, observe que foram inseridos dois registros. Observe ainda a chave estrangeira *id_departamento_fk*. Esta sinaliza a que departamento cada funcionário está associado.

Implementando a relação N:N

Após implementarmos a associação 1:N, vamos mapear a tabela *projeto*, que em conjunto com a tabela *funcionario* mantém uma relação do tipo muitos-para-muitos.

Listagem 9. Código da classe de teste 1N.

```
01. package com.jm.teste;
02.
03. //imports omitidos
04.
05. public class Teste1N {
06.
07.    public static void main(String args[]) {
08.
09.        Departamento departamento = new Departamento();
10.        departamento.setNome("Recursos Humanos");
11.
12.        DepartamentoDAO departamentoDAO = new DepartamentoDAO();
13.        departamentoDAO.adicionarDepartamento(departamento);
14.
15.        Funcionario func1 = new Funcionario();
16.        Funcionario func2 = new Funcionario();
17.
18.        func1.setNome("Carlos Martins");
19.        func1.setCargo("Analista de Sistemas");
20.        func1.setEndereco("Rua da Pitangueira");
21.        func1.setCpf("06693829393");
22.        func1.setSalario(4000);
23.
24.        func2.setNome("Camila Rodrigues");
25.        func2.setCargo("Esteticista");
26.        func2.setEndereco("Rua Floriano Peixoto");
27.        func2.setCpf("06543490322");
28.        func2.setSalario(4566);
29.
30.        func1.setDepartamento(departamento);
31.        func2.setDepartamento(departamento);
32.
33.        FuncionarioDAO funcionarioDAO = new FuncionarioDAO();
34.        funcionarioDAO.adicionarFuncionario(func1);
35.        funcionarioDAO.adicionarFuncionario(func2);
36.     }
37. }
```

Os passos para estabelecer esse mapeamento são idênticos aos realizados com as classes **Funcionario** e **Departamento**. A **Listagem 10** apresenta o código da classe **Projeto**.

Listagem 10. Código da classe Projeto.

```
01. package com.jm.teste;
02.
03. //imports omitidos...
04.
05. @Entity
06. @Table(name = "projeto", schema = "empresabd")
07. public class Projeto implements Serializable {
08.
09.    private static final long serialVersionUID = 1L;
10.
11.    @Id
12.    @GeneratedValue(strategy = GenerationType.IDENTITY)
13.    @Column(name = "id_projeto")
14.    private Long id;
15.
16.    @Column(name = "nome")
17.    private String nome;
18.
19.    @Column(name = "data_inicio")
20.    private Date dataInicio;
21.
22.    @Column(name = "data_fim")
23.    private Date dataFim;
24.
25.    @ManyToMany(fetch = FetchType.LAZY, mappedBy = "projetos")
26.    private Set<Funcionario> funcionarios;
27.
28.    //sets e gets omitidos
29. }
```

A diferença em relação às outras classes mapeadas aparece na linha 27, que expõe o uso da anotação **@ManyToMany**.

Após implementar esta classe, precisamos realizar algumas alterações na configuração do Hibernate e no código do projeto para que a associação N:N seja mapeada corretamente entre as tabelas *projeto* e *funcionario*. Primeiramente, devemos mapear a classe **Projeto** no arquivo de configuração (*hibernate.cfg.xml*), conforme o código a seguir (tal código deve ser inserido no final do arquivo):

```
<mapping class="com.jm.entidade.Projeto" />
```

```
mysql> use empresabd;
Database changed
mysql> select * from funcionario;
+----------------+-----------------+----------------------+---------+-------------+----------------------+---------------------+
| id_funcionario | nome            | cargo                | salario | cpf         | endereco             | id_departamento_fk  |
+----------------+-----------------+----------------------+---------+-------------+----------------------+---------------------+
|              1 | Carlos Martins  | Analista de Sistemas |    4000 | 06693829393 | Rua da Pitangueira   |                   1 |
|              2 | Camila Rodrigues| Esteticista          |    4566 | 06543490322 | Rua Floriano Peixoto |                   1 |
+----------------+-----------------+----------------------+---------+-------------+----------------------+---------------------+
2 rows in set (0.00 sec)

mysql> select * from departamento;
+-----------------+-----------------+
| id_departamento | nome            |
+-----------------+-----------------+
|               1 | Recursos Humanos |
+-----------------+-----------------+
1 row in set (0.00 sec)

mysql> _
```

Figura 10. Resultado da execução da classe de teste

Feito isso, altere a classe **Funcionario** para inserir o código exposto na **Listagem 11**. Neste bloco de código, a anotação **@ManyToMany**, presente na linha 1, define o relacionamento do tipo muitos para muitos. Já a anotação **@JoinTable** (vide linha 2) cria uma tabela adicional para mapear quais funcionários estão relacionados a quais projetos e vice-versa. O atributo **name** define o nome da tabela criada; neste caso, "trabalha". Por fim, na linha 3 temos uma lista que faz referência a projetos, caracterizando o relacionamento N:N.

Listagem 11. Mapeamento da relação muitos-para-muitos na classe Funcionario.

```
01. @ManyToMany(fetch = FetchType.LAZY, cascade = CascadeType.ALL)
02. @JoinTable(name = "trabalha", catalog = "empresabd",
    joinColumns = { @JoinColumn(name = "id_funcionario_fk", nullable = false) },
    inverseJoinColumns = { @JoinColumn(name = "id_projeto_fk",
    nullable = false) })
03. private Set<Projeto> projetos;
```

Persistindo dados no banco

Agora, vamos implementar a classe que viabilizará todas as operações de persistência sobre a entidade *projeto*. Essa classe, de nome **ProjetoDAO**, deve ser criada no mesmo pacote que **DepartamentoDAO** e **FuncionarioDAO**, **com.jm.acessobd**. Veja seu código na **Listagem 12**.

A diferença desse código em relação aos DAOs criados para as classes **Funcionario** e **Departamento** está no tipo de objeto que as operações manipulam, que nesse caso é do tipo **Projeto**.

Testando a implementação N:N

Agora que todas as configurações foram realizadas, testaremos a relação muitos-para-muitos. Para tanto, crie a classe de testes **TesteNN** no pacote **com.jm.teste**. Essa classe cria dois projetos e depois associa alguns funcionários a eles. O código da classe de teste pode ser visto na **Listagem 13**.

Com o objetivo de associar os funcionários que serão criados

Listagem 12. Código da classe ProjetoDAO.

```
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_proj = (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_proj;
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. }
```

a um departamento, na linha 10 é recuperado o departamento cadastrado anteriormente. Em seguida, são criados dois projetos: o primeiro de nome "Nono Digito" e o segundo chamado "Decimo Digito". Nas linhas 19 e 20 esses projetos são adicionados a uma lista para que possam ser associados aos Funcionários que serão criados. Após a criação dos funcionários, eles são associados aos projetos, e nas linhas 31 e 39 são vinculados ao departamento recuperado. Por fim, nas linhas 42 e 43 os funcionários são internalizados no banco de dados.

Resultado da execução

Encerrada a execução da classe **TesteNN**, o banco de dados estará conforme a **Figura 11**. Na tabela *funcionario*, agora temos dois novos registros, assim como na tabela *projeto*.

```
mysql> use empresabd;
Database changed
mysql> select * from departamento;
+-----------------+-----------------+
| id_departamento | nome            |
+-----------------+-----------------+
|               1 | Recursos Humanos |
+-----------------+-----------------+
1 row in set (0.00 sec)

mysql> select * from funcionario;
+---------------+-----------------+----------------------+---------+-------------+----------------------+--------------------+
| id_funcionario | nome            | cargo                | salario | cpf         | endereco             | id_departamento_fk |
+---------------+-----------------+----------------------+---------+-------------+----------------------+--------------------+
|              1 | Carlos Martins  | Analista de Sistemas |    4000 | 06693829393 | Rua da Pitangueira   |                  1 |
|              2 | Camila Rodrigues | Esteticista          |    4566 | 06543490322 | Rua Floriano Peixoto |                  1 |
|              3 | Yujo Rodrigues  | Analista de Sistemas |    8000 | 06693844493 | Rua da Semente       |                  1 |
|              4 | Daniel Cioqueta | Analista de Sistemas |    4522 | 02243455322 | Rua Afonso Pena      |                  1 |
+---------------+-----------------+----------------------+---------+-------------+----------------------+--------------------+
4 rows in set (0.00 sec)

mysql> select * from projeto;
+------------+--------------+-------------+----------+
| id_projeto | nome         | data_inicio | data_fim |
+------------+--------------+-------------+----------+
|          1 | Nono Digito  | NULL        | NULL     |
|          2 | Decimo Digito | NULL        | NULL     |
+------------+--------------+-------------+----------+
2 rows in set (0.00 sec)

mysql> select * from trabalha;
+-------------------+---------------+
| id_funcionario_fk | id_projeto_fk |
+-------------------+---------------+
|                 3 |             1 |
|                 4 |             1 |
|                 3 |             2 |
|                 4 |             2 |
+-------------------+---------------+
4 rows in set (0.00 sec)

mysql> _
```

Figura 11. Resultado da execução da classe de teste

Listagem 13. Código da classe de teste TesteNN.

```
01. package com.jm.teste;
02.
03. //imports omitidos
04.
05. public class TesteNN {
06.
07.     DepartamentoDAO departamentoDao = new DepartamentoDAO();
08.       List<Departamento> departamentos = departamentoDao
          .listarDepartamentos();
09.
10.       Departamento departamento = departamentos.get(0);
11.
12.       Projeto projeto1 = new Projeto();
13.       projeto1.setNome("Nono Digito");
14.
15.       Projeto projeto2 = new Projeto();
16.       projeto2.setNome("Decimo Digito");
17.
18.       List<Projeto> projetos = new ArrayList<Projeto>();
19.       projetos.add(projeto1);
20.       projetos.add(projeto2);
21.
22.       Funcionario func1 = new Funcionario();
23.       Funcionario func2 = new Funcionario();
24.
25.       func1.setNome("Yujo Rodrigues");
26.       func1.setCargo("Analista de Sistemas");
27.       func1.setEndereco("Rua da Semente");
28.       func1.setCpf("06693844493");
29.       func1.setSalario(8000);
30.       func1.setProjetos(projetos);
31.       func1.setDepartamento(departamento);
32.
33.       func2.setNome("Daniel Cioqueta");
34.       func2.setCargo("Analista de Sistemas");
35.       func2.setEndereco("Rua Afonso Pena");
36.       func2.setCpf("02243455322");
37.       func2.setSalario(4522);
38.       func2.setProjetos(projetos);
39.       func2.setDepartamento(departamento);
40.
41.       FuncionarioDAO funcionarioDAO = new FuncionarioDAO();
42.       funcionarioDAO.adicionarFuncionario(func1);
43.       funcionarioDAO.adicionarFuncionario(func2);
44.   }
45. }
```

Ademais, verificamos também a tabela *trabalha*, que armazena as chaves primárias de *funcionario* e *projeto*, viabilizando assim o relacionamento entre elas.

Além de ser um ótimo framework open source para estabelecer o mapeamento objeto relacional, o Hibernate também é a solução mais adotada atualmente. A estabilidade em que se encontra o projeto e as facilidades que ele oferece ao programador estão entre suas principais características.

O Hibernate faz com que a camada de persistência seja desenvolvida com muita simplicidade e liberdade, tanto por dar mais legibilidade ao código, o que é conseguido por meio de anotações, por exemplo, quanto pela portabilidade oferecida, que abstrai as particularidades de cada SGBD.

Autor

Carlos Alberto Silva
casilvamg@hotmail.com
Formado em Ciência 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.

Autor

Lucas de Oliveira Pires
pires.info@gmail.com
Formado em Sistemas de Informação pelo Centro Universitário do Triângulo (Unitri) com especialização em Análise e Desenvolvimento de Sistemas Aplicados à Gestão Empresarial (IFTM). Trabalha na empresa Algar Telecom como Analista de Desenvolvimento.

Links e Livro:

Site oficial do Hibernate.
http://hibernate.org /

Site oficial do MySQL.
http://www.mysql.com/

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

Documentação da anotação @OneToMany.
https://docs.oracle.com/javaee/6/api/javax/persistence/OneToMany.html

Documentação da anotação @ManyToMany.
http://docs.oracle.com/javaee/6/api/javax/persistence/ManyToMany.html

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

Qualidade de código com DDD e TDD

Veja nesse artigo como aplicar as boas práticas e padrões do Domain-Driven Design e as técnicas de Test-Driven Development

Os processos de desenvolvimento e manutenção dos sistemas de grande parte das empresas possuem custos elevados por conta da alta complexidade do problema que os mesmos se propõem a resolver e da baixa qualidade do código e/ou demais artefatos. Dentre os principais fatores responsáveis pela baixa qualidade, podemos citar a documentação desatualizada em relação ao código-fonte, este que não reflete o domínio do negócio, dificuldades em entender o código, duplicidade, pouca ou nenhuma cobertura de testes unitários e de integração, baixa coesão e alto acoplamento entre módulos e classes.

Como solução para os problemas citados, este artigo apresenta a metodologia Domain-Driven Design, proposta por Eric Evans, no livro "Domain-Driven Design: Atacando as Complexidades no Coração do Software". Com o mesmo intuito, será apresentada a técnica de desenvolvimento e implementação de sistemas Test-Driven Development, que se baseia na construção de softwares através de ciclos de teste – implementação – refatoração. O TDD tem como criador e principal incentivador o engenheiro americano Kent Beck, também criador da metodologia ágil XP. Por fim, serão expostos os benefícios e ganhos, a médio e longo prazo, de se utilizar DDD e TDD no design e codificação de sistemas.

Fique por dentro

Este artigo é útil para arquitetos, programadores, analistas de sistemas e líderes de projetos que buscam desenvolver sistemas de alta qualidade tendo em vista um código-fonte que reflita o domínio e as regras de negócio das empresas.

Levando em consideração esse cenário, o principal objetivo deste artigo é apresentar a metodologia de projeto DDD e a técnica de desenvolvimento de software TDD, que visam melhorar a qualidade do software e diminuir os altos custos no desenvolvimento, manutenção e evolução.

Domain-Driven Design

Fundamentado na experiência de mais de 20 anos de Erick Evans no desenvolvimento de sistemas, o DDD é uma abordagem que reúne um conjunto de boas práticas, padrões, ferramentas e recursos da orientação a objetos que têm como objetivo a construção e desenvolvimento de sistemas de acordo com o domínio e regras de negócio do cliente. Além disso, questões relacionadas ao processo de desenvolvimento, como a necessidade de um estreito relacionamento entre a equipe de programadores e os especialistas do domínio, também são tratadas pela abordagem.

O principal conceito do DDD é o modelo. Este expressa o domínio e negócio do cliente e pode ser criado utilizando desenhos, fluxogramas, diagramas, etc. O importante é que ele represente o negócio do cliente.

Como principais componentes do DDD, podemos listar: a linguagem onipresente, a arquitetura em camadas e os padrões.

Com o intuito de mostrar essa abordagem na prática, será utilizado como exemplo um sistema de reservas de churrasqueira e salão de festas de um condomínio, cujo modelo é apresentado logo a seguir, através do diagrama de classes da UML (vide **Figura 1**).

Sistema de reservas da churrasqueira e do salão de festas

O condomínio Parque Novo Mundo, formado por um conjunto de seis prédios, tem aproximadamente 1.200 moradores e uma excelente estrutura para festas e comemorações, sendo oito churrasqueiras e seis salões de festa em sua área comum.

Entretanto, por conta da grande procura por parte dos moradores para reservar estes espaços e da não utilização de um software para auxiliar administradores e moradores no procedimento de reserva destas áreas, o processo de agendamento e controle se torna custoso e propenso a erros.

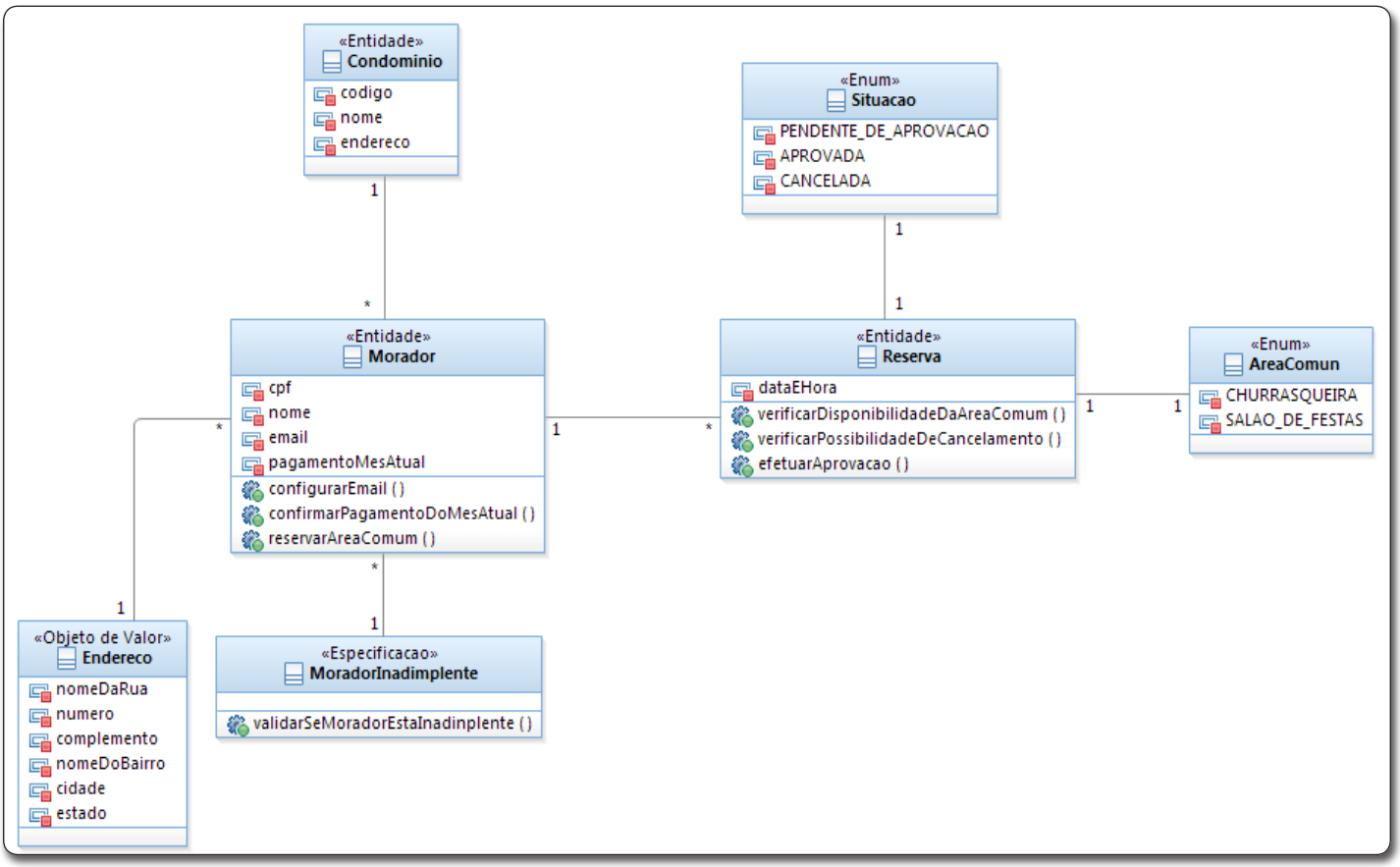

Figura 1. Modelo de domínio do condomínio para o sistema de reservas da churrasqueira e salão

Visando possibilitar aos moradores do condomínio uma maior facilidade na reserva destas áreas e um melhor controle por parte dos administradores, foi solicitada a uma empresa de desenvolvimento de software a implementação de um sistema de reservas, a partir do qual um morador poderá realizar a reserva da churrasqueira ou salão na data desejada caso ele esteja em dia com as taxas do condomínio e o ambiente já não esteja alocado. Solicitada a reserva, ela ficará pendente aguardando a aprovação do administrador, que verificará a disponibilidade do local. Ademais, concluída essa etapa, o morador poderá cancelar o pedido com até três dias de antecedência.

Linguagem onipresente

A linguagem onipresente ou ubiquitous language é uma das características do DDD que permite a clientes, gerentes de projeto, analistas de sistemas, arquitetos e desenvolvedores "falarem a mesma língua", evitando assim que membros da equipe utilizem termos que não reflitam o domínio do negócio do sistema.

A não utilização da linguagem onipresente impede e dificulta a troca de informações, ideias e conhecimentos, resultando na implementação de um código-fonte que não reflete o negócio e, com isso, provoca uma maior dificuldade na manutenção, evolução e correção de defeitos, o que, consequentemente, eleva os custos.

Os termos e conceitos da Linguagem Onipresente devem ser utilizados em todos os artefatos do software, incluindo: documentos, diagramas, código-fonte e em todas as comunicações feitas entre os envolvidos na construção do sistema.

O DDD usa o modelo de domínio para representar de forma gráfica o relacionamento entre os componentes do sistema, e adota a linguagem onipresente na criação dos nomes de classes, atributos e métodos, com o objetivo de ser possível expressar o negócio do cliente sem ambiguidades. Dessa forma, o modelo se torna uma ferramenta valiosa de comunicação entre equipe técnica e especialistas do negócio, contendo os termos e conceitos que estão sendo e/ou serão empregados no projeto.

Um recurso que pode ser empregado para criar o modelo de domínio é o diagrama de classes da UML, como já demonstrado. O ideal é que ele seja construído e evoluído com a presença de pelo menos um integrante da equipe técnica (Arquiteto, Programador ou Analista de Sistema) e especialistas do negócio, pois se o mesmo for criado sem a presença de uma pessoa da equipe técnica, corre-se o risco de se construir um modelo que não é implementável, e caso a construção aconteça sem a presença de especialistas do negócio, ele poderá se tornar base para a construção de um software com código-fonte que não reflete o domínio do negócio.

Arquitetura em camadas

A abordagem do DDD propõe que os componentes do software sejam organizados em uma arquitetura em camadas (vide **Figura 2**) de acordo com as suas responsabilidades, permitindo assim que cada camada do sistema tenha a sua especialidade.

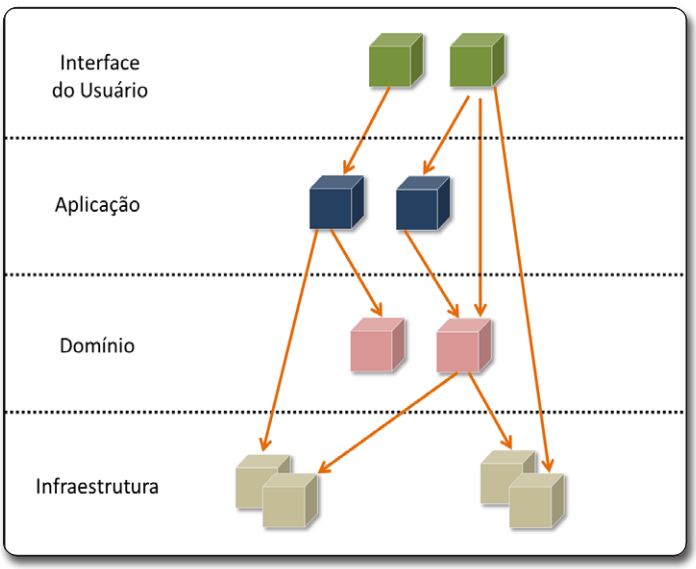

Figura 2. Arquitetura em camadas

A principal ideia dessa divisão arquitetural é que componentes pertencentes a uma camada dependam apenas de outros da mesma ou de qualquer outra abaixo, sendo cada camada responsável por um aspecto do software. Isto possibilita, por exemplo, que as regras de negócio de um componente da camada de domínio de uma aplicação que tenha uma interface web JSF (componente da camada de interface do usuário) possam ser reutilizadas por outra interface de usuário, por exemplo, mobile, centralizando em um único ponto as regras de negócio.

Caso um componente da camada de infraestrutura dependa de outro, por exemplo, da camada de domínio, esse componente estaria altamente acoplado a um determinado domínio, dificultando a sua reutilização.

O DDD propõe a divisão arquitetural através das seguintes camadas:

• **Interface do usuário ou camada de apresentação:** Possui componentes responsáveis por mostrar informações e interpretar comandos do usuário. Não devem ser colocados nessa camada componentes de software relacionados a regras de negócio e domínio. Podemos citar como exemplos para esta camada: JSP, JSF, XHTML, HTML, JavaScript, Managed Beans e Actions do Struts;

• **Aplicação:** Camada que possui componentes responsáveis por coordenar e solicitar tarefas para a camada de domínio. Deverá ser fina e não conter implementações de regras de negócios e processos. Possui relação direta com o domínio de negócio da aplicação. Um bom exemplo para essa camada seria a classe **GerenciadorDeReservas** (vide **Listagem 1**), onde o método **efetuarReserva()** apenas coordena chamadas aos métodos **validarSeMoradorEstaInadimplente()**, **verificarDisponibilidadeDaAreaComum()** e **efetuarAprovacao()**, que estão implementados respectivamente nas entidades **Morador** e **Reserva**, localizadas logo depois na camada de domínio. Perceba que a classe **GerenciadorDeReservas** não contém detalhes referentes à implementação das regras de negócio;

Listagem 1. Serviço do Domínio responsável pela validação das regras de negócio e efetivação de uma reserva.

```
@Stateless
public class GerenciadorDeReservas {

  @PersistenceContext
  EntityManager entityManager;

  @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
  public void efetuarReserva(Reserva reserva)
  throws  MoradorEncontraseInadimplenteException,
          AreaComumIndisponivelException {
    Morador morador = reserva.obterMorador();

    //Entidade da camada de domínio valida se o morador encontra-se inadimplente.
    morador.validarSeMoradorEstaInadimplente();

    //Entidade da camada de domínio valida se a área encontra-se disponível.
    reserva.verificarDisponibilidadeDaAreaComum();

    //Após serem feitas as validações acima, a reserva será aprovada.
      reserva.efetuarAprovacao();

    //Realizando a efetivação da reserva. Armazenando no Banco de Dados.
    entityManager.persist(reserva);
  }
}
```

• **Domínio:** Camada composta por classes, objetos, serviços e funções responsáveis pela implementação das regras de negócio e processos diretamente relacionados com o domínio do software. O DDD propõe que seja criada uma camada de domínio com componentes que utilizem as melhores técnicas e boas práticas da orientação a objetos, como herança, polimorfismo, agregação, especialização, encapsulamento, etc. Os objetos e componentes dessa camada representam o coração do sistema, o que justifica o porquê de grande parte das recomendações, boas práticas e padrões do DDD se concentrarem nela;

• **Infraestrutura:** Camada responsável por recursos técnicos de infraestrutura, recursos que não estão relacionados diretamente ao domínio de negócio da aplicação. Os componentes dessa camada podem ser utilizados e acessados por qualquer camada superior. Como exemplos, podemos citar: serviços de persistência de dados, envio de e-mail, geração de relatórios e planilhas, envio e recebimento de mensagens, gerenciamento e controle de fuso horário, gerenciamento de perfis e permissões de usuários, etc.

Padrões

Para organizar os componentes da linguagem onipresente representados através do modelo de domínio da aplicação, o

DDD propõe um conjunto de padrões. A seguir apresentamos os principais:

1. Entidades: São objetos definidos não pelos seus atributos e propriedades, mas sim pela sua identidade. Possui como identificador um código, que deve ser único. Veja como exemplo o objeto **morador**, criado pela fábrica **ConstrutorDeMoradores** (apresentada na **Listagem 2**). Este possui como identificador o atributo **cpf**;

2. Objetos de Valor: Ao contrário das entidades, os objetos de valor não possuem nenhuma identidade, sendo definidos por seus atributos. Devem ser tratados como imutáveis e possuir um design simples e limpo. Um exemplo de objeto de valor seria a classe **Endereco** (vide **Figura 1**) do sistema de reservas;

3. Agregados: O padrão Agregados permite agrupar as entidades e objetos de valor e identificar uma entidade como raiz do agregado, para que, objetos e serviços externos que precisarem manipular qualquer informação das entidades e objetos de valor façam isso através da raiz do agregado. Um exemplo seria o relacionamento entre a entidade **Morador** e o objeto de valor **Endereco**, onde **Morador** seria a raiz do agregado, pois não existe **Morador** sem **Endereco**;

4. Fábricas: As fábricas se mostram muito úteis na criação de agregados e objetos que possuem uma estrutura complexa, com muitos relacionamentos entre entidades e objetos de valor. De acordo com o DDD, todas as regras e códigos para a criação de entidades e objetos de valor que compõem um agregado são de responsabilidade das fábricas. Um exemplo de Fábrica seria a classe **ConstrutorDeMoradores** (vide **Listagem 2**), responsável por construir e validar as regras de criação de um objeto da entidade **Morador**, evitando, por exemplo, que se cadastre um **Morador** sem **Endereco**;

5. Especificação: O padrão Especificação tem como principal objetivo verificar se um determinado objeto atende a um conjunto de regras. Podemos citar como exemplo de especificação **MoradorInadimplente** (vide **Figura 1**), que verifica se um objeto **Morador** está inadimplente;

6. Repositório: O padrão Repositório é implementado por meio de classes responsáveis por gerenciar o ciclo de vida de objetos, através de operações de inclusão, remoção, alteração e buscas em determinado meio de armazenamento, seja ele um sistema gerenciador de banco de dados, diretório, arquivos, etc. O Java possui alguns frameworks, como Hibernate e TopLink, que permitem realizar o mapeamento de um objeto com um banco de dados relacional e então manipular esses objetos no banco de dados;

7. Serviços: Serviços são classes responsáveis por executar determinadas tarefas que estejam diretamente relacionadas ao domínio da aplicação e podem estar presentes nas camadas de aplicativo, domínio e infraestrutura. Quando essas tarefas não se encaixam em nenhuma entidade ou objeto de valor, o DDD recomenda a criação de serviços. Um exemplo de serviço seria a classe **GerenciadorDeReservas** (vide **Listagem 1**), onde o método **efetuarReserva()** solicita às entidades **Morador** e **Reserva** a validação das regras de negócio relacionadas e concretiza a efetivação da reserva;

8. Módulos: O conceito de Módulos propõe que entidades, objetos de valor, especificações, repositórios e serviços sejam organizados e separados de acordo com o domínio do negócio, ao contrário da maioria dos projetos, nos quais as classes, objetos, serviços e repositórios da aplicação são separados e organizados, geralmente, de acordo com a tecnologia, padrão de projeto ou framework, colocando-se, por exemplo, todos os managed beans de um projeto que usa JSF em um pacote. Vale ressaltar que os módulos devem ser independentes, procurando manter o baixo acoplamento e a alta coesão entre classes, objetos, serviços e outros componentes.

Listagem 2. Fábrica responsável pela criação de objetos da entidade Morador.

```
public class ConstrutorDeMoradores {

   public static Morador construirMorador(String cpf, String nome,
                String email, String nomeDaRua,
                String numero, String complemento,
                String bairro, String cidade, String
          estado) throws CamposObrigatoriosException {

     if(cpf == null || nome == null || nomeDaRua == null ||
       numero == null || complemento == null || bairro == null ||
       cidade == null || estado == null)
        throw new CamposObrigatoriosException();

     Morador morador = new Morador();
     morador.atribuirCpf(cpf);
     morador.atribuirNome(nome);
     morador.configurarEmail(email);

     Endereco endereco = new Endereco(nomeDaRua, numero, complemento,
                bairro, cidade, estado);
     morador.cadastrarEndereco(endereco);

     return morador;
   }
}
```

Test-Driven Development (TDD)

A técnica de desenvolvimento de software TDD tem como premissa a construção de toda a estrutura e design do código-fonte guiada e orientada por classes de testes, tendo como principal objetivo a implementação de um código que seja limpo, autoexplicativo, de fácil manutenção, coeso, com pouca duplicidade e baixo acoplamento. Outra preocupação é evitar que se construa código desnecessário.

Conforme a **Figura 3**, o TDD se baseia no desenvolvimento iterativo e incremental, sendo aplicado através de três passos:

1. Escrever um teste falho;
2. Escrever uma classe que faça o teste passar de forma simples;
3. Refatorar a classe.

Os principais aspectos referentes ao TDD que serão abordados neste artigo são: testes de unidade, testes de integração, design e qualidade interna do código-fonte, suíte de testes automatizados e integração contínua e documentação para programadores.

Teste de Unidade

O teste unitário ou de unidade é o principal componente do TDD, sendo responsável por testar e validar a menor parte de uma classe: um método ou funcionalidade. Os testes unitários têm como vantagem o feedback imediato, já que são extremamente rápidos de serem executados por não possuírem integração com nenhum outro módulo ou parte do sistema.

O principal framework open source para testes unitários na plataforma Java é o JUnit, que oferece um conjunto de classes e anotações para facilitar a implementação e execução dos testes. Por estes e muitos outros motivos, recomendamos fortemente que você aprenda e domine os recursos desta solução.

As **Listagens 3** e **4** mostram, respectivamente, as classes de testes responsáveis pelo design e cobertura das funcionalidades relacionadas às entidades **Morador** e **Reserva**. Respeitando a proposta do TDD, tais classes de teste foram criadas antes da implementação de **Morador** e **Reserva**, contendo código apenas para demonstrar a implementação de testes unitários.

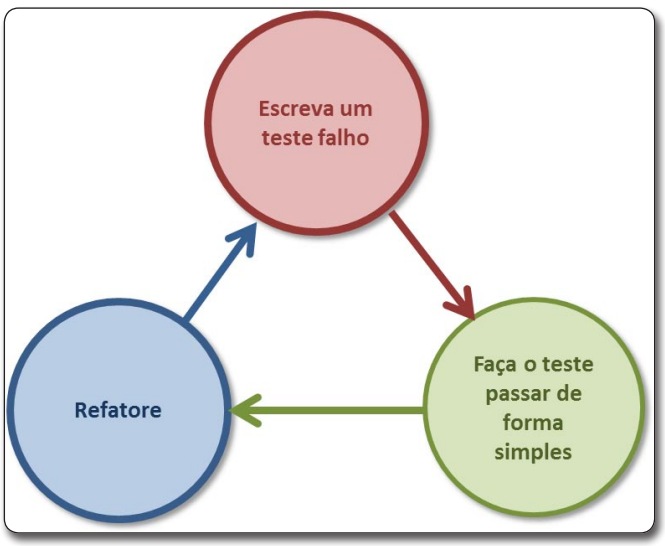

Figura 3. Ciclo de implementação com TDD

A classe de testes unitários **MoradorTest** realiza a cobertura das seguintes regras de negócio: verificação do morador Luciano Henrique, que encontra-se inadimplente, e da moradora Luciana Fonseca, que já realizou o pagamento do mês. Já os testes de unidade presentes em **ReservaTest** realizam a cobertura das seguintes regras: morador Luciano Henrique deseja verificar a disponibilidade do salão de festas para o dia 31 de dezembro de 2015 e a moradora Luciana efetua a reserva da churrasqueira com sucesso.

Teste de Integração

São testes que envolvem várias camadas da aplicação e utilizam infraestrutura e recursos similares aos do ambiente de produção como, por exemplo, a necessidade do servidor de aplicação e do banco de dados na execução dos testes. Por isso sua execução se torna mais lenta, se comparada com a dos testes de unidade.

Testes de integração são utilizados normalmente para avaliar classes de serviços, repositórios e DAOs, que são classes responsáveis por encapsular a complexidade de acesso ao banco de dados e realizar operações de inserção, remoção e buscas.

Tais testes também podem ser empregados para validar os mapeamentos objeto-relacionais, definidos, por exemplo, com Hibernate ou JPA, o que é feito a partir de operações básicas de inclusão e busca envolvendo os objetos mapeados.

Atualmente, muitos frameworks/bibliotecas facilitam a implementação dos testes de integração. Dentre eles podemos citar o JBoss Arquillian, que trouxe muita simplicidade e agilidade para o desenvolvimento desse tipo de teste, possibilitando a sua adoção junto ao TDD.

Listagem 3. Teste unitário da entidade Morador, criado utilizando TDD.

```
public class MoradorTest {

  @Test(expected = MoradorEncontraseInadimplenteException.class)
  public void verificarMoradorDoCondominioQueEncontraseInadimplente()
        throws MoradorEncontraseInadimplenteException,
        CamposObrigatoriosException {


    //Solicitando para a Fábrica (ConstrutorDeMoradores) a criação do Morador.
    Morador moradorLucianoHenrique = ConstrutorDeMoradores
        .construirMorador("123456789-12",
        "Luciano Henrique de Medeiros da Silva",null,
        "Rua Francisco José da Silva","90","apartamento 67D", "Vila Mariana",
        "São Paulo","SP");
        moradorLucianoHenrique.configurarEmail(
        "lucianohenriquemedeirosdasilva@tyu.com.br");

    //Realizando os testes unitários no objeto moradorLuciano.
    assertNotNull(moradorLucianoHenrique);
    moradorLucianoHenrique.validarSeMoradorEstaInadimplente();
  }
```

```
  @Test
  public void verificarMoradoraDoCondominioQueJaRealizouOPagamentoDoMes()
  throws  MoradorEncontraseInadimplenteException, CamposObrigatoriosException
  {
    //Solicitando para a Fábrica (ConstrutorDeMoradores) a criação do Morador.
    Morador moradoraLucianaFonseca = ConstrutorDeMoradores
          .construirMorador("987654321-21", "Luciana Fonseca",null,
          "Rua Francisco José da Silva","90","apartamento 156C",
          "Vila Mariana","São Paulo","SP");

    moradoraLucianaFonseca.configurarEmail(
          "lucianafonseca@pty.com.br");
    moradoraLucianaFonseca.confirmarPagamentoDoMesAtual();

    //Realizando os testes unitários no objeto moradoraLucianaFonseca.
    assertNotNull(moradoraLucianaFonseca);
    assertFalse(moradoraLucianaFonseca.validarSeMoradorEstaInadimplente());
  }
}
```

A **Listagem 5** mostra um exemplo de teste de integração do sistema exemplo que valida o cadastro de um morador com sucesso e a tentativa de cadastramento de um morador já cadastrado. Para isso, o teste de integração **GerenciadorDeMoradoresIT** valida o serviço de domínio **GerenciadorDeMoradores** (vide **Listagem 6**), que cuida da manutenção dos dados dos moradores no sistema. Para esses testes de integração, foi utilizado o framework JBoss Arquillian.

Design e qualidade interna do código-fonte

Com TDD tornam-se evidentes vários aspectos positivos relacionados ao design e qualidade das classes implementadas. Dentre eles podemos citar o baixo acoplamento, métodos e classes com poucas responsabilidades, código-fonte que reflete o domínio do negócio, podendo ser utilizado como documentação pelos programadores, não implementação de código desnecessário, dentre outros.

Listagem 4. Teste unitário da entidade Reserva, criado utilizando TDD.

```
public class ReservaTest {

  SimpleDateFormat format = new SimpleDateFormat("dd/MM/yyyy");

  @Test
  public void verificarDisponibilidadeDoSalaoDeFestasParaODia31DeDezembro
  De2015() throws ParseException,
    AreaComumIndisponivelException,
    CamposObrigatoriosException {
    //Solicitando para a Fábrica(ConstrutorDeMoradores) a criação do Morador.
    Morador moradorLucianoHenrique = ConstrutorDeMoradores
        .construirMorador("123456789-12", "Luciano Henrique de Medeiros
        da Silva",null,
        "Rua Francisco José da Silva","90","apartamento 67D", "Vila Mariana",
        "São Paulo","SP");
    moradorLucianoHenrique.configurarEmail(
        "lucianohenriquemedeirosdasilva@tyu.com.br");
    moradorLucianoHenrique.confirmarPagamentoDoMesAtual();
    moradorLucianoHenrique.reservarAreaComun(format.parse("31/12/2015"),
    AreaComum.SALAO_FESTAS);

    Reserva reserva = moradorLucianoHenrique.obterReserva();

    //Realizando os testes unitários no objeto reserva.
    assertNotNull(reserva.obterDataEHora() != null);
    assertTrue(reserva.verificarDisponibilidadeDaAreaComum());
  }
  @Test
  public void efetuarAAprovacaoDaReservaDaMoradoraLucianaComSucesso()
  throws ParseException, CamposObrigatoriosException, MoradorEncontrase
  InadimplenteException, AreaComumIndisponivelException {
    //Solicitando para a Fábrica(ConstrutorDeMoradores) a criação do Morador.
    Morador moradoraLucianaFonseca = ConstrutorDeMoradores
    .construirMorador("987654321-21", "Luciana Fonseca",null,
    "Rua Francisco José da Silva", "90","apartamento 156C ",
     "Vila Mariana","São Paulo","SP");
    moradoraLucianaFonseca.configurarEmail("lucianafonseca@pty.com.br");
    moradoraLucianaFonseca.confirmarPagamentoDoMesAtual();
    moradoraLucianaFonseca.reservarAreaComun(format.parse("15/02/2015"),
    AreaComum.CHURRASQUEIRA);

    Reserva reserva = moradoraLucianaFonseca.obterReserva();

    //Realizando os testes unitários no objeto reserva.
    assertFalse(moradoraLucianaFonseca.validarSeMoradorEstaInadimplente());
    assertNotNull(reserva.obterDataEHora() != null);

    assertTrue(reserva.verificarDisponibilidadeDaAreaComum());
    assertTrue(reserva.efetuarAprovacao());
  }
}
```

Listagem 5. Teste de integração do serviço de domínio GerenciadorDeMoradores.

```
@RunWith(Arquillian.class)
public class GerenciadorDeMoradoresIT {

  @EJB(mappedName="ejb/GerenciadorConfiguracoes#IGerenciadorDeMoradores")
  private IGerenciadorDeMoradores gerenciadorDeMoradores;

  @Test
  public void incluirMoradorComSucesso() throws CamposObrigatoriosException,

    MoradorJaCadastradoException {
    //Solicitando à Fábrica (ConstrutorDeMoradores) a criação do Morador.
    Morador moradorLucianoHenrique = ConstrutorDeMoradores
        .construirMorador("123456789-12",
          "Luciano Henrique de Medeiros da Silva",null,
          "Rua Francisco José da Silva","90","apartamento 67D",
          "Vila Mariana","São Paulo","SP");
    moradorLucianoHenrique.configurarEmail(
        "lucianohenriquemedeirosdasilva@tyu.com.br");

    //Solicitando ao Serviço de Domínio cadastrar o morador no sistema.
    Morador moradorCadastradoComSucesso = gerenciadorDeMoradores
    .cadastrarMorador(moradorLucianoHenrique);

    //Realizando os testes para verificar se o Morador foi incluído com
      //sucesso no sistema.
    assertTrue(moradorCadastradoComSucesso != null);
    assertTrue(moradorCadastradoComSucesso.obterNome()
        .equalsIgnoreCase("Luciano Henrique de Medeiros da Silva"));
    assertTrue(moradorCadastradoComSucesso.obterEmail()
        .equalsIgnoreCase("lucianohenriquemedeirosdasilva@tyu.com.br"));  }

  @Test(expected = MoradorJaCadastradoException.class)
  public void tentandoCadastrarMoradorJaCadastrado()
  throws CamposObrigatoriosException,  MoradorJaCadastradoException {

    //Solicitando à Fábrica (ConstrutorDeMoradores) a criação do Morador.
    Morador moradorLucianoHenrique = ConstrutorDeMoradores
        .construirMorador("123456789-12", "Luciano Henrique de Medeiros da
         Silva",null, "Rua Francisco José da Silva","90","apartamento 67D",
         "Vila Mariana","São Paulo","SP");

    moradorLucianoHenrique.configurarEmail(
        "lucianohenriquemedeirosdasilva@tyu.com.br");

    //Solicitando ao Serviço de Domínio cadastrar o Morador no sistema.
    Morador moradorCadastradoComSucesso = gerenciadorDeMoradores
    .cadastrarMorador(moradorLucianoHenrique);

    //Tentando cadastrar Morador novamente.
    Morador moradorLucianoHenriqueJaCadastrado = gerenciadorDeMoradores
    .cadastrarMorador(moradorLucianoHenrique);
  }
}
```

Listagem 6. Serviço de domínio GerenciadorDeMoradores.

```
@Stateless
public class GerenciadorDeMoradores implements IGerenciadorDeMoradores {

  @PersistenceContext
  EntityManager entityManager;

  @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
  public Morador cadastrarMorador(Morador morador)
      throws CamposObrigatoriosException, MoradorJaCadastradoException {

    morador.verificarCamposObrigatorios();
    morador.verificarMoradorJaCadastrado();

    return entityManager.merge(morador);
  }
}
```

Mesmo que o TDD seja uma ótima técnica de design, não significa que deixa de ser necessária a utilização de outras técnicas já conhecidas e consolidadas para se alcançar uma boa qualidade do código-fonte. Isto é, o uso de diagramas e modelos para o entendimento dos problemas, a utilização de design patterns para resolver problemas já conhecidos, a implementação de provas de conceito para tomar decisões arquiteturais, entre outras, devem continuar sendo adotadas.

Suíte de testes automatizados e integração contínua

A implementação e desenvolvimento de testes unitários e de integração faz com que seja criada uma extensa suíte de testes automatizada, que pode ser utilizada para monitorar todo o código-fonte a partir de cada commit realizado no servidor. Esta suíte de testes dará feedbacks constantes sobre o código que está sendo implementado e versionado. Diante disso, caso o desenvolvimento de uma nova funcionalidade gere algum erro em algo que já funcionava, a suíte de testes indicará o ponto exato do código que deixou de funcionar.

Outro fator para o qual a suíte de testes automatizados contribui é a realização de refatorações no sistema. Com este suporte, o desenvolvedor passa a ter mais segurança para realizar melhorias contínuas no código, efetuando mudanças sem medo de quebrar algo que já funcionava, o que proporciona um aumento significativo da qualidade do sistema como um todo.

Ao longo do projeto, no entanto, a suíte de testes se torna extensa e deste modo passa a ser custosa sua execução por completo a cada commit realizado pelos desenvolvedores. Visando contornar esse problema e manter a execução dos testes para garantir a "saúde" do projeto, a solução indicada é a adoção de um servidor de integração contínua, que de tempos em tempos pode verificar se existe algum erro de compilação e executar todos os testes unitários e de integração, notificando imediatamente os profissionais envolvidos e responsáveis caso algum código quebre ou algum teste falhe. Um dos servidores de integração contínua mais populares e utilizados por desenvolvedores Java é o Jenkins.

Documentação para programadores

Um dos maiores problemas para a compreensão de um sistema no momento de realizar tarefas como manutenção e evolução é a falta de documentação, ou pior ainda, encontrar uma grande quantidade de artefatos, diagramas e documentos desatualizados em relação ao código-fonte. Por conta da velocidade e dinâmica que alterações e novas funcionalidades são solicitadas, manter todos os artefatos do software atualizados tem se tornado uma tarefa complexa.

Diante disso, uma forma de facilitar o entendimento do software para a realização de manutenções e inclusões de novas funcionalidades, é buscar a geração de uma documentação mínima para o sistema e implementar um código-fonte auto-explicativo, com classes, métodos, atributos, serviços e repositórios que reflitam exatamente o domínio do negócio. Como importante complemento, os testes unitários e de integração desenvolvidos com TDD servirão como uma extensa e fiel documentação para o programador acoplada a todo o código de produção.

Em grande parte dos projetos que não adotam técnicas como DDD e TDD, assim como práticas que garantem uma boa qualidade aos artefatos gerados durante o planejamento, implementação e testes, os custos de manutenção, evolução e correção de defeitos se tornam mais altos, principalmente por conta da baixa qualidade do código-fonte e da documentação desatualizada.

A utilização de DDD em conjunto com TDD se mostra extremamente importante para o sucesso de sistemas a curto, médio e longo prazo, diminuindo consideravelmente os custos para evolução e correção de defeitos dos softwares, por conta da qualidade do código-fonte e da alta cobertura da suíte de testes.

Como demonstrado, é viável e pode ser incentivada a adoção de DDD e TDD simultaneamente no planejamento e implementação de sistemas, pois uma prática não anula a outra, sendo possível, portanto, aproveitar o melhor das técnicas que cada uma das abordagens propõe, levando em consideração sempre as necessidades e complexidades de cada software.

Saiba, sobretudo, que utilizar as técnicas de DDD e TDD requer disciplina e organização, pois exige que o desenvolvedor se preocupe muito mais com questões relacionadas a design, padrões de projeto e qualidade de código, pois o código passa a ser a principal documentação do projeto, devendo, portanto, poder ser lido e entendido por uma pessoa com conhecimentos básicos na linguagem de programação empregada.

Além disso, procure utilizar o DDD e o TDD de acordo com as características e necessidades do projeto, levando em consideração o conhecimento e experiência da equipe. Para isso, comece a empregar as técnicas aos poucos, introduzindo e experimentando as boas práticas em sessões de programação em par, por exemplo. Como outro diferencial, treine a sua equipe, organize *lunch talks* para disseminar as técnicas e demonstre a importância de se ter uma extensa suíte de testes a médio e longo prazo para todos os envolvidos no projeto.

Enfim, a partir de DDD e TDD os clientes e a equipe técnica ficarão mais satisfeitos não só com o software entregue, mas também durante todo o ciclo de vida do mesmo, possibilitando a redução de custos para o cliente associada à alta qualidade de código para o desenvolvedor.

Autor

Lincoln Fernandes Coelho
lincolnfcoelho@gmail.com
Bacharel em Ciência da Computação, trabalha com desenvolvimento de software há 12 anos, sendo especialista na plataforma Java, design, boas práticas e qualidade de código-fonte. Possui as certificações SCJP, SCWCD e SCBCD.

Referências e Links:

Domain-Driven Design: Atacando as Complexidades no Coração do Software. Eric Evans, Alta Books, 2003.

TDD – Desenvolvimento Guiado Por Testes. Kent Beck, Bookman, 2010.

Introdução à Arquitetura e Design de Software – Uma Visão Geral Sobre A Plataforma Java. Paulo Silveira, Guilherme Silveira, Sérgio Lopes, Guilherme Moreira, Nico Steppat, Fábio Kung, 2012.

UML Essencial – Um breve guia para a linguagem padrão de modelagem de objetos. Martin Fowler, 3° Edição, 2005

Refatoração – Aperfeiçoando o Projeto de Código Existente. Martin Fowler, Bookman, 2004.

Domain Driven Design – InfoQ.
http://www.infoq.com/domain-driven-design

Introduction to Test Driven Development (TDD).
http://www.agiledata.org/essays/tdd.html

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