



Edição 43

 DEV MEDIA

Encapsulamento no Java:
Primeiros passos
Conheça um dos principais conceitos
da Orientação a Objetos

RMI na prática
Desenvolvendo aplicações
com objetos distribuídos

Dicas de qualidade
Programando código limpo
e de fácil manutenção

EXPLORANDO A JPA

Aprenda a
persistir dados
em Java



ISSN 2179625-4



DESENVOLVA PARA CORPORAÇÕES OU PARA O DIA A DIA COM JAVA



FORMAÇÃO DESENVOLVEDOR **JAVA**

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

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

CERTIFICAÇÕES:

➤ Oracle Certified Professional, Java SE 6 Programmer

➤ Oracle Certified Professional, Java EE 5 Web Component Developer



Sumário

Conteúdo sobre Boas Práticas

06 – Encapsulamento em Java: Primeiros passos

[José Fernandes A. Júnior]

Conteúdo sobre Boas Práticas

14 – Dicas de qualidade para o código Java

[Toyoaki Ejiri]

Artigo no estilo Solução Completa

21 – Java Persistence API (JPA): Explorando o EntityManager

[Andrei de Oliveira Tognolo]

Artigo no estilo Solução Completa

28 – Remote Method Invocation: RMI na prática

[John Soldera]

CURSOS ONLINE

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



CONHEÇA ALGUNS DOS CURSOS:

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



Para mais informações :

www.devmedia.com.br/curso/javamagazine

(21) 3382-5038



Edição 43 • 2014 • ISSN 2173625-4



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

EXPEDIENTE

Editor

Eduardo Spínola (eduspinola@gmail.com)

Consultor Técnico Davi Costa (davigc_08@hotmail.com)

Produção

Jornalista Responsável Kaline Dolabella - JP24185

Capa e Diagramação Romulo Araujo

Atendimento ao leitor

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

Publicidade

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

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

Fale com o Editor!

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

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



EDUARDO OLIVEIRA SPÍNOLA

eduspinola.wordpress.com

@eduspinola / @Java_Magazine

CURSOS ONLINE

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

Feita para Desenvolvedores de Software e DBAs



CONHEÇA OS CURSOS MAIS RECENTES:

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

Para mais informações :

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



DEVMEDIA



CRIANDO APLICAÇÕES INOVADORAS QUE VÃO MUDAR O MUNDO?

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

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

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

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

Na suite ToolsCloud você tem a sua disposição:

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

Acompanhe a Toolscloud:

 toolscloud@toolscloud.com



twitter.com/toolscloud



Encapsulamento em Java: Primeiros passos

Aumentando a segurança e a qualidade do código através do uso correto de encapsulamento

O que é o encapsulamento? Talvez a coisa mais importante que devamos saber sobre encapsulamento é a diferença que existe entre ele e a ocultação de informações, devido à grande confusão que sempre existiu entre estas duas definições. A ocultação de informações é considerada parte do encapsulamento, mas se fizermos uma pesquisa na internet, podemos encontrar a seguinte definição para encapsulamento: Um mecanismo da linguagem de programação para restringir o acesso a alguns componentes dos objetos, escondendo os dados de uma classe e tornando-os disponíveis somente através de métodos.

Na verdade, o mecanismo para restringir o acesso a alguns dos componentes do objeto é a definição de ocultação de informações. O encapsulamento é um conceito da Programação Orientada a Objetos onde o estado de objetos (as variáveis da classe) e seus comportamentos (os métodos da classe) são agrupados em conjuntos segundo o seu grau de relação.

Assim sendo, o propósito do encapsulamento é o de organizar os dados que sejam relacionados, agrupando-os (encapsulando-os) em objetos (classes), reduzindo as colisões de nomes de variáveis (dado que variáveis com o mesmo nome estarão em namespaces distintos) e, da mesma forma, reunindo métodos relacionados às suas propriedades (ou variáveis de classe). Este padrão ajuda a manter um programa com centenas ou milhares de linhas de código mais legível e fácil de trabalhar e manter.

Para facilitar a compreensão e visualização do que é o encapsulamento, vamos ver um simples exemplo. Suponha que temos um programa que trabalha com informações de seres vivos, como: homem e cachorro. Para estes dois seres vivos (objetos) existem características (atributos) em comum, como nome, idade e peso, e características (atributos) que são específicas a cada um, como número de identidade para o homem e raça para o cachorro. Há também operações comuns para ambos, como andar, ou específicas a cada um, como falar para

Fique por dentro

Este artigo descreve como você pode combinar a programação orientada a objetos e conceitos da linguagem Java para escrever programas mais fáceis de manter e com um bom design de código. Vamos explicar o que é o encapsulamento e qual a diferença entre este e a ocultação de informações. Em seguida apresentaremos um exemplo de como podemos utilizar com sucesso o encapsulamento numa classe que simula, de forma simplista, um elevador, adicionando lógica para tornar sua operação a mais segura possível.

o homem e latir para o cachorro. Uma possível representação das classes desses objetos pode ser a demonstrada na **Figura 1**.

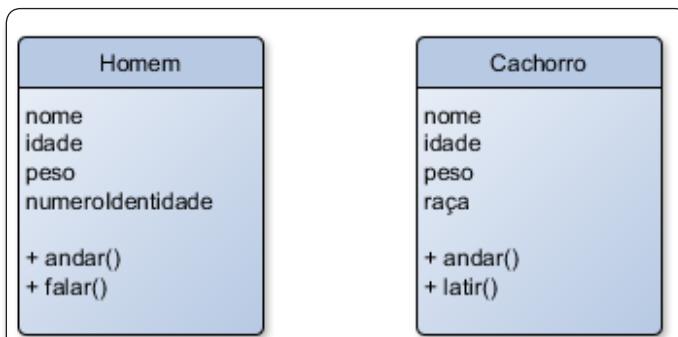


Figura 1. Representação das classes Homem e Cachorro

Como podemos notar, estas classes encapsulam os dados relacionados a cada objeto de forma que possamos acessar cada um deles sem conflito, por estarem cada um em seu domínio. Ou seja, para saber a idade de um homem, podemos perguntar por **Homem.idade** e, da mesma forma, para saber a idade de um cachorro, podemos perguntar por **Cachorro.idade**. Os dois atributos têm o mesmo nome, mas cada um tem o seu próprio domínio. O mesmo pode ser dito sobre as operações de cada uma das classes. Isto é encapsulamento!

Por outro lado, o conceito de ocultação de informações é mais do que esconder, ou ocultar os dados, é também o critério primário para a modularização de sistemas, que deve levar em consideração a ocultação de decisões de design que são suscetíveis a mudanças. Protegendo as informações desta maneira, retiramos a exigência de um conhecimento íntimo do projeto por parte dos clientes, de forma a poderem usar qualquer módulo como se fosse uma caixa preta, ou seja, sem saber o que está lá dentro, mas apenas o que entra e o que sai dela. Assim, o cliente não tem que saber ou se preocupar sobre como é feita a lógica de determinados métodos; apenas deve chamá-los e utilizar os seus resultados.

Por exemplo, seguindo a representação anterior, suponha que gostaríamos de obter o IMC (Índice de Massa Corporal) para um homem e para um cachorro. Para isto, teríamos que adicionar mais um campo em cada classe: altura para o homem e estatura para o cachorro. Além disso, tanto a classe **Homem**, quanto a classe **Cachorro**, teriam um método de nome **calcularIMC()** que retornaria o valor do IMC para qualquer um dos objetos. No caso do homem, o método **calcularIMC()** faria a conta: $IMC = peso/altura^2$. No caso do cachorro, usando uma aproximação sugerida por vários autores, teríamos: $IMC = peso/estatura^2$. O cliente (quem chama o método) não tem à disposição o campo IMC. Este valor é calculado segundo os valores de outros campos dos objetos. O cliente não tem que saber como o cálculo é feito ou, se o cálculo passou a ser feito de outra forma (para ter mais precisão, por exemplo). Para ele, não deve haver qualquer diferença. Ele apenas tem que saber que cada método trará o valor do IMC para qualquer um dos objetos (homem ou cachorro).

Modificadores de visibilidade

Vamos ver agora como alterar o nível de acesso que podemos dar aos métodos e atributos das classes do nosso projeto para mudar a visibilidade destes para os outros artefatos de código (classes, atributos e métodos) do projeto. Métodos e atributos podem ter modificadores como **public** e **private** para indicar o nível de acesso que outras classes terão dos mesmos. A **Tabela 1** mostra qual o nível de acesso que existe para os membros de uma classe de acordo com seu modificador de acesso.

Modificador	Classe	Package	Subclasse	Todos
public	SIM	SIM	SIM	SIM
protected	SIM	SIM	SIM	NÃO
sem modificador	SIM	SIM	NÃO	NÃO
private	SIM	NÃO	NÃO	NÃO

Tabela 1. Níveis de acesso dos membros de uma classe segundo seu modificador

Se observarmos a primeira coluna de dados (a coluna Classe), veremos que uma classe tem sempre acesso a todos os seus membros, não importa qual o modificador de acesso destes. A segunda coluna nos diz que todas as classes que pertencem ao mesmo pacote de uma outra classe têm acesso a todos os seus membros, exceto os privados. Por sua vez, a terceira coluna nos diz que as

subclasses de uma determinada classe têm acesso aos membros públicos e protegidos da classe estendida. Por fim, a última coluna nos diz que todas as outras classes (que não sejam uma classe que pertença ao mesmo pacote, ou uma subclass) têm acesso a todos os membros públicos de qualquer classe, e apenas a estes.

São estes os modificadores de visibilidade que existem na linguagem Java e que irão nos permitir aumentar a qualidade e segurança de nosso código quando bem utilizados, e já vamos explicar melhor o porquê.

Uma dica importante para ajudar na escolha de que nível de acesso usar é: seja o mais restritivo possível! Isto é, use sempre o modificador **private** para os membros de uma classe, a não ser que exista uma boa razão para não o fazer. Lembre-se que para que possamos implementar encapsulamento em nosso código (contando com a proteção da ocultação de informações), teremos que ter todos os membros da classe com acesso privado, dando o acesso externo a estes através de métodos públicos. Esta dica apenas não é válida para as constantes. Como os valores das constantes não podem ser alterados, permitir que tenham o seu acesso público não causa qualquer problema de segurança ao código.

Potenciais problemas com atributos públicos

O que significa dizer termos os atributos da classe públicos? Significa, como vimos na **Tabela 1**, que qualquer classe, de qualquer pacote do programa, pode alterar os valores dos atributos públicos dessa classe livremente, sem qualquer tipo de restrição! Isso implica que um cliente que não saiba as regras de negócio de um determinado módulo do programa pode introduzir valores inválidos, provocando erros de execução. Vejamos um pequeno exemplo para ilustrar alguns dos potenciais problemas que podemos introduzir em nosso código quando temos todos os atributos com visibilidade pública. O exemplo demonstrado na **Listagem 1** rocura simular de forma muito simplista o funcionamento de um elevador.

No código temos a classe **Elevator**, que representa o nosso elevador. Podemos abrir e fechar a porta do elevador através da variável **doorOpen**, saber em que andar o elevador se encontra através da variável **currentFloor** e saber qual o peso do elevador através da variável **weight**. Também temos algumas constantes como **WEIGHT_CAPACITY**, que nos indica qual a capacidade máxima de peso que o elevador suporta, **TOP_FLOOR** para saber qual é o andar mais alto do prédio e **BOTTOM_FLOOR**, para saber qual é o andar mais baixo.

Como a classe **Elevator** não implementa encapsulamento, é possível alterar os valores dos seus atributos livremente e de maneiras indesejáveis. No código da classe de teste **TestElevator**, podemos ver (com a ajuda dos comentários) que ações foram tomadas no elevador. A porta foi aberta para a entrada de passageiros, e depois foi fechada para que o elevador pudesse andar. Depois, o elevador recebeu o comando para andar um andar para baixo, apesar de já estar no andar mais baixo do prédio, ou seja, foi provocado um erro. Posteriormente foi indicado para o elevador subir um andar e “saltar” até o 27º andar sem passar por todos

Encapsulamento em Java: Primeiros passos

os andares que se encontram entre o andar corrente e o 27º andar do prédio, mesmo sabendo que o andar mais alto do prédio é o 20º, provocando outro erro.

Como podemos constatar, o programa deste elevador não é seguro. O que nos garante que, com esta interface de utilização, um utilizador não abra a porta e faça o elevador andar de porta aberta? Nada! Também nada nos garante que o elevador seja encaminhado apenas para andares válidos, nem que o peso no interior do elevador seja menor ou igual ao peso máximo estipulado quando estiver em funcionamento.

Listagem 1. Exemplo de código com atributos públicos.

```
public class Elevator {  
    public boolean doorOpen = false;  
    public int currentFloor = 0;  
    public int weight = 0;  
  
    public static final int TOP_FLOOR = 20;  
    public static final int BOTTOM_FLOOR = 0;  
    public static final int WEIGHT_CAPACITY = 400;  
}  
  
public class TestElevator {  
  
    public static void main(String[] args) {  
        // Cria uma instância de um elevador.  
        Elevator elevator = new Elevator();  
  
        // Abre a porta para entrada de passageiros.  
        elevator.doorOpen = true;  
  
        // Fecha a porta após entrada de passageiros.  
        elevator.doorOpen = false;  
  
        // Desce um andar (o elevador estava no andar 0 e não existe andares abaixo  
        // deste).  
        elevator.currentFloor--;  
        System.out.println("Andar corrente: " + elevator.currentFloor);  
  
        // Sobe um andar.  
        elevator.currentFloor++;  
        System.out.println("Andar corrente: " + elevator.currentFloor);  
  
        // Salta para o 27º andar (o prédio só tem 20 andares).  
        elevator.currentFloor = 27;  
        System.out.println("Andar corrente: " + elevator.currentFloor);  
    }  
}
```

O modificador private

O modificador privado permite que os atributos e operações de uma determinada classe se tornem inacessíveis a outras classes. Para tornar privado um atributo ou método de uma classe, inserimos o modificador **private** na frente do atributo ou método do qual pretendemos alterar a visibilidade. No entanto, se tornarmos os membros de uma classe privados, temos que ter em mente que não teremos como alterar o estado de uma instância desta classe a não ser que criemos métodos visíveis ao exterior desta que nos permitam fazer isso, ou seja, temos que criar uma interface pública de utilização.

Para aplicarmos com sucesso o conceito de encapsulamento em nosso código, não basta apenas tornar os atributos da classe privados e simplesmente criar métodos públicos para retornar e alterar diretamente estes mesmos valores (os chamados *getters* e *setters* que falaremos mais à frente), temos que adicionar lógica ao nosso código de forma a tornar as operações seguras, garantindo que não seja possível atribuir valores indevidos às nossas variáveis.

Dito isso, vamos seguir com o nosso exemplo e alterar os modificadores de visibilidade de nossas variáveis da classe **Elevator** para privado, mas deixando as constantes públicas. Deste modo, teremos o resultado demonstrado pela **Listagem 2**.

Ao implementar estas mudanças a classe **TestElevator** deixará

Listagem 2. Alteração dos modificadores de visibilidade das variáveis da classe Elevator para private.

```
public class Elevator {  
  
    private boolean doorOpen = false;  
    private int currentFloor = 0;  
    private int weight = 0;  
  
    public static final int TOP_FLOOR = 20;  
    public static final int BOTTOM_FLOOR = 0;  
    public static final int WEIGHT_CAPACITY = 400;  
}
```

de compilar, pois esta tenta alterar valores de atributos privados da classe **Elevator**. Nestas condições a classe **Elevator** não é muito útil, porque não existem maneiras de modificar os valores dos seus atributos.

Implementando uma interface pública

O que podemos fazer então para tornar útil a nossa classe **Elevator**? A resposta é simples: criando uma interface pública para ela. Mas o que é a interface pública de uma classe? As variáveis e métodos públicos de uma classe são comumente referenciados como a interface da classe, porque são os únicos elementos que outras classes podem utilizar para alterar o estado dos objetos instanciados a partir dela. E um dos objetivos das linguagens orientadas a objetos é o de nos incentivar a criar esta interface pública, de forma que a implementação dos métodos possa mudar sem afetar a interface. Assim, quem chama os métodos de uma classe precisa saber apenas o quê o método faz. Como o método faz o seu trabalho não é importante. Isso permite que os métodos das classes possam ser alterados de forma a melhorar o seu desempenho, sua segurança, introduzir chamadas a novas bases de dados, introduzir novos algoritmos, etc., e a chamada a ele continue sendo feita exatamente da mesma maneira.

Getters e Setters

Quando tornamos os atributos de uma classe privados, temos que criar uma maneira de poder acessá-los se acharmos que eles devem ser obtidos e/ou alterados por outras classes clientes desta (classes que chamam métodos desta classe). Uma forma de fazer

isso é criando métodos públicos que nos permitam retornar (get) ou definir (set) os valores desses atributos, os conhecidos *getters* e *setters*. Eles são tão utilizados que existem atalhos nos IDEs para criar estes métodos de forma automática.

Vamos então alterar o nosso código de forma que ele volte a ser compilável. Para atingir esse objetivo, implemente os *getters* e *setters* da classe **Elevator**. Para não termos que escrever todos os métodos manualmente, vamos aproveitar a ajuda do Eclipse e gerar os métodos automaticamente, com o atalho *Alt+Shift+S* e depois selecionando a opção de menu *Generate Getters and Setters....*. A Figura 2 mostra esta janela.

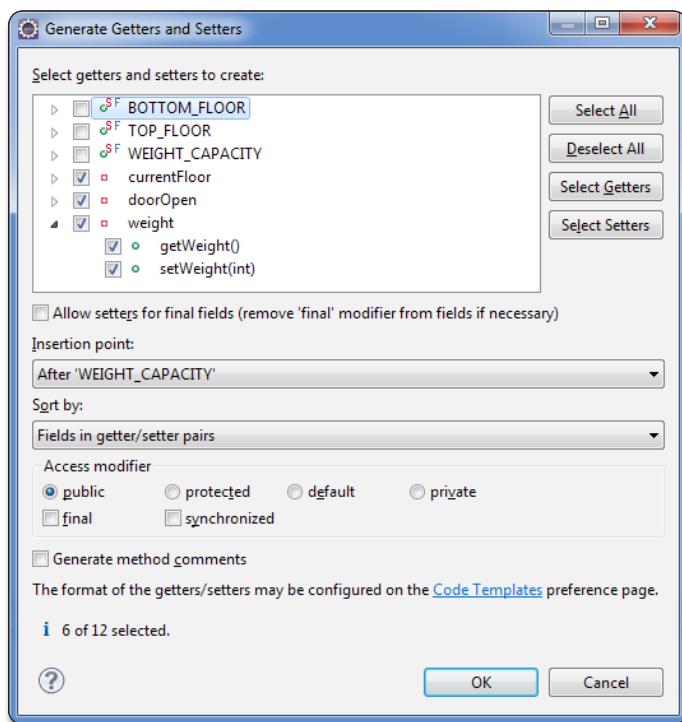


Figura 2. Gerando os métodos getters e setters no Eclipse

Nesta janela podemos escolher as variáveis para as quais queremos criar os métodos públicos. Neste exemplo vamos selecionar todas as variáveis privadas. Após clicarmos no botão *OK* teremos gerado métodos para retornar e atribuir os valores das variáveis privadas de nossa classe.

Apesar de já termos criado uma interface pública para utilização da nossa classe, temos ainda que realizar algumas alterações no código da classe de teste **TestElevator**. Ao invés de acessarmos diretamente as variáveis da classe **Elevator**, vamos alterar estes acessos de forma a utilizar os métodos que acabamos de criar. A Listagem 3 mostra o resultado final destas alterações.

Como podemos observar, o IDE gera automaticamente o nome dos métodos com um “get” ou “set” na frente do nome da variável de classe, com exceção para os métodos que retornam variáveis booleanas, onde ao invés do “get” é utilizado o “is”.

Na Listagem 3 podemos ver que os métodos públicos foram gerados de forma a podermos cometer os mesmos erros que já

Listagem 3. Geração de getters e setters para a classe **Elevator** e alteração da classe de teste para os utilizar.

```
public class Elevator {
    private boolean doorOpen = false;
    private int currentFloor = 0;
    private int weight = 0;

    public static final int TOP_FLOOR = 20;
    public static final int BOTTOM_FLOOR = 0;
    public static final int WEIGHT_CAPACITY = 400;

    public boolean isDoorOpen() {
        return doorOpen;
    }

    public void setDoorOpen(boolean doorOpen) {
        this.doorOpen = doorOpen;
    }

    public int getCurrentFloor() {
        return currentFloor;
    }

    public void setCurrentFloor(int currentFloor) {
        this.currentFloor = currentFloor;
    }

    public int getWeight() {
        return weight;
    }

    public void setWeight(int weight) {
        this.weight = weight;
    }

    public class TestElevator {
        public static void main(String[] args) {
            // Cria uma instância de um elevador.
            Elevator elevator = new Elevator();

            // Abre a porta para entrada de passageiros.
            elevator.setDoorOpen(true);

            // Fecha a porta após entrada de passageiros.
            elevator.setDoorOpen(false);

            // Desce um andar (o elevador estava no andar 0 e não existe andares abaixo deste).
            elevator.setCurrentFloor(elevator.getCurrentFloor() - 1);
            System.out.println("Andar corrente: " + elevator.getCurrentFloor());

            // Sobe um andar.
            elevator.setCurrentFloor(elevator.getCurrentFloor() + 1);
            System.out.println("Andar corrente: " + elevator.getCurrentFloor());

            // Salta para o 27º andar (o prédio só tem 20 andares).
            elevator.setCurrentFloor(27);
            System.out.println("Andar corrente: " + elevator.getCurrentFloor());
        }
    }
}
```

Encapsulamento em Java: Primeiros passos

cometíamos quando as variáveis eram públicas, com a diferença de estarmos usando métodos ao invés de alterar os valores das variáveis de classe diretamente. Os métodos gerados não possuem “inteligência”, isto é, não possuem qualquer lógica/validação relacionada às regras de negócio da aplicação para nos retornar ou atribuir os valores das variáveis da classe, simplesmente retornam e atribuem os valores. Como mencionado, temos ainda que introduzir lógica ao nosso código de forma a tornar as operações seguras, fazendo validações de modo a garantir que os valores corretos sejam atribuídos.

Implementando o encapsulamento

Agora que já alteramos a visibilidade dos atributos e criamos uma interface pública, vamos analisar um caso de cada vez para adicionar a segurança no código que possa nos garantir o correto funcionamento do elevador. Por exemplo, o que podemos validar quando um cliente tenta fechar ou abrir a porta do elevador? Uma primeira validação pode ser a de verificar se a porta já se encontra na mesma posição que a do comando enviado; ou seja, se o cliente pedir para fechar a porta e a porta já se encontra fechada, o elevador não deve fazer nada (o mesmo para a porta aberta). Outra validação que podemos fazer é a de verificar se o peso máximo suportado pelo elevador não foi ultrapassado quando alguém tenta fechar a porta do mesmo. Se assim for, o elevador deve manter a porta aberta até que o peso diminua para dentro dos limites. Por fim, uma melhoria de legibilidade do código que podemos fazer é tornar o método **setDoorOpen()** (método que recebe um booleano) privado e criar dois métodos públicos que sejam mais claros quanto à sua funcionalidade: **openDoor()**, para abrir a porta e **closeDoor()**, para fechar a porta. A **Listagem 4** mostra as alterações no código relativas à introdução de lógica nos métodos de abrir e fechar a porta do elevador.

Como podemos ver pelo código, temos uma interface pública onde, para o cliente, existem três operações:

- Perguntar se a porta está aberta;
- A operação de abrir a porta;
- A operação de fechar a porta.

Como a porta abre, ou fecha, e qualquer validação que exista nestas operações, deve ser totalmente transparente para o cliente. Ele não precisa saber como estas operações são feitas, quer apenas que elas sejam feitas com sucesso e segurança, e é isso que estamos proporcionando com o nosso código.

Agora, o que podemos fazer em relação ao movimento do elevador, quando apertamos o botão do número do andar para onde queremos ir? Obviamente, se um prédio tem 20 andares, o elevador não terá os botões de 21º, 22º, ou 23º andares (segurança introduzida por hardware). Mas, quem nos garante que quem programou os botões dos elevadores não cometeu um erro e fez com que o botão 3, ao invés de mandar o elevador para o 3º andar, manda-o para o 30º andar? Ninguém! Então o melhor é validarmos que estamos sempre dentro dos nossos limites. E o que deveria acontecer quando apertarmos o botão do mesmo andar

onde já nos encontramos? Nada. Mas só podemos garantir esse comportamento se programarmos o elevador para não fazer nada nestas condições. Caso contrário, o elevador poderia, por exemplo, fechar as portas e só depois validar que já se encontra no mesmo andar para o qual foi comandado para ir, e apenas depois disso abrir as portas novamente.

Listagem 4. Lógica por trás das operações de abertura e fechamento das portas do elevador.

```
public boolean isDoorOpen() {
    return _doorOpen;
}

public boolean openDoor() {
    return setDoorOpen(true);
}

public boolean closeDoor() {
    return setDoorOpen(false);
}

private boolean setDoorOpen(boolean doorOpen) {
    String doorPosition = doorOpen ? "aberta": "fechada";

    // Verifica se a posição da porta não é a mesma que a posição atual.
    if (_doorOpen == doorOpen) {
        System.out.println("- A porta já se encontra " + doorPosition);
        return true;
    }

    // Se for para fechar a porta, verifica se o peso do elevador não excede o peso
    // máximo permitido.
    if (!doorOpen && _weight > WEIGHT_CAPACITY) {
        System.out.println("- Elevador com mais peso que o permitido!");
        return false;
    }

    _doorOpen = doorOpen;
    System.out.println("- Porta " + doorPosition);

    return true;
}
```

Também podemos validar se a porta se encontra fechada antes do elevador entrar em movimento, caso contrário colocaríamos em risco a segurança das pessoas em seu interior. Além disso, mais uma melhoria do código é a de assegurar que o elevador ande um andar de cada vez, para que possa (futuramente) ser adicionada a lógica de onde parar se alguém o estiver chamando do lado de fora. O código com estas validações pode ser visto na **Listagem 5**.

Nesse caso, para o cliente, só existem duas operações: saber em que andar o elevador se encontra (para poder colocar esta informação em um placar eletrônico, por exemplo), e dizer para onde o elevador deve ir. Mais uma vez deve-se ressaltar que toda a lógica por trás da implementação dos métodos deve ser transparente para o cliente, informações que ele não precisa saber.

Por fim, temos os métodos relacionados ao controle do peso que está no elevador. Tais métodos foram codificados de modo simples. Por medida de segurança, é importante sabermos se a capacidade

Listagem 5. Lógica por trás das operações de movimento do elevador.

```
public int getCurrentFloor() {
    return _currentFloor;
}

private void goUp() {
    _currentFloor++;
    System.out.println(" - Andar: " + _currentFloor);
}

private void goDown() {
    _currentFloor--;
    System.out.println(" - Andar: " + _currentFloor);
}

public void setCurrentFloor(int currentFloor) {
    // Verifica se o piso indicado não é menor que o andar mais baixo do prédio.
    if (currentFloor < BOTTOM_FLOOR) {
        System.out.println(" - Número de andar errado! O andar mínimo é " + BOTTOM_FLOOR);
        return;
    }
    // Verifica se o piso indicado não é maior que o andar mais alto do prédio.
    if (currentFloor > TOP_FLOOR) {
        System.out.println(" - Número de andar errado! O andar mais alto é " + TOP_FLOOR);
        return;
    }
    // Verifica se o piso indicado não é o mesmo que o piso corrente.
    if (currentFloor == _currentFloor) {
        System.out.println(" - O elevador já se encontra no andar indicado.");
        return;
    }
    // Verifica se a porta está fechada antes de entrar em movimento.
    if (_doorOpen) {
        System.out.println(" - Fechando a porta...");
        if (!closeDoor()) {
            System.out.println(" - O elevador não pode andar com a porta aberta!");
            return;
        }
    }
    // Anda 1 andar de cada vez.
    if (_currentFloor < currentFloor) {
        System.out.println(" - Subindo...");
        while (_currentFloor != currentFloor) {
            goUp();
        }
    } else {
        System.out.println(" - Descendo...");
        while (_currentFloor != currentFloor) {
            goDown();
        }
    }
}
```

do elevador está sendo respeitada. Sendo assim, podemos perguntar qual o peso do elevador, através do método `getWeight()`, e dizer qual o peso que está nele, através do método `setWeight()`. Em um cenário real estes dados seriam medidos através de sensores (no nosso caso inserimos os dados manualmente). A **Listagem 6** mostra o código para esses métodos.

Listagem 6. Getter e Setter da operação de peso do elevador.

```
public int getWeight() {
    return _weight;
}

public void setWeight(int weight) {
    _weight = weight;
    System.out.println(" - Peso do elevador: " + _weight + " Kg");
}
```

Pronto! Agora podemos testar a segurança do nosso elevador. Para isso, vamos enviar alguns comandos válidos e outros inválidos e verificar como o elevador se comporta. A **Listagem 7** mostra a nossa classe de teste, onde tentamos subir e descer com o elevador para andares inexistentes, andar no elevador com mais peso do que o permitido e andar no elevador com a porta aberta.

Finalizada a execução do teste, vamos analisar os resultados gerados pelo código da **Listagem 7** e ver como o programa se comportou. A saída da classe `Elevator` foi configurada com um hífen em azul no início da linha para diferenciar da saída originada pela classe de testes. A **Listagem 8** mostra o resultado.

Como podemos verificar, conseguimos movimentar o elevador com uma série de validações de segurança, tendo implementado

Não perca tempo reinventando a roda!

COBREBEMX

Componente completo para sua Cobrança por Boleto Bancário e Débito em Conta Corrente

Mais de 40 exemplos em diversas linguagens de programação

Geração e leitura de arquivos (remessa e retorno) nos padrões FEBRABAN e CNAB

Testes e Downloads gratuitos em nosso site

ACESSE E CONHEÇA O COMPONENTE EM:
WWW.COBBREM.COM

Encapsulamento em Java: Primeiros passos

Listagem 7. Código da classe de teste do elevador.

```
public class TestElevator {  
  
    public static void main(String[] args) {  
        // Cria uma instância de um elevador.  
        Elevator elevator = new Elevator();  
  
        System.out.println("Abrir a porta para entrada de passageiros.");  
        elevator.openDoor();  
  
        // A entrada de passageiros aumenta o peso do elevador (um sensor do elevador se  
        // encarregaria  
        // de enviar estes dados).  
        System.out.println("Entraram pessoas no elevador.");  
        elevator.setWeight(elevator.getWeight() + 300);  
  
        // Fecha a porta após entrada de passageiros apertando o botão de fechar a porta.  
        System.out.println("Foi acionado o botão de fechar a porta.");  
        elevator.closeDoor();  
  
        // Desce um andar (o elevador estava no andar 0 e não existe andares abaixo deste).  
        // O botão de descer um andar não existe no elevador, mas estamos simulando o  
        // envio de um  
        // comando errado.  
        System.out.println("Enviado comando para descer 1 andar.");  
        elevator.setCurrentFloor(elevator.getCurrentFloor() - 1);  
  
        // Sobe um andar.  
        System.out.println("Enviado comando para subir 1 andar.");  
        elevator.setCurrentFloor(elevator.getCurrentFloor() + 1);  
  
        // Salta para o 27º andar (o prédio só tem 20 andares).  
        // Mais uma vez, o botão do 27º andar não existiria no elevador, mas devemos nos  
        // assegurar  
        // que um comando mal enviado não afete o seu correto funcionamento.  
        System.out.println("Enviado comando para subir até o 27º andar.");  
  
        // Sobe até o 10º do prédio.  
        System.out.println("Enviado comando para subir até o 10º andar.");  
        elevator.setCurrentFloor(10);  
  
        // Abre a porta do elevador.  
        System.out.println("Enviado comando para abrir a porta.");  
        elevator.openDoor();  
        System.out.println("Enviado comando para abrir a porta.");  
        elevator.openDoor();  
  
        // A entrada de passageiros aumenta o peso do elevador.  
        System.out.println("Entraram pessoas no elevador.");  
        elevator.setWeight(elevator.getWeight() + 280);  
  
        // Desce até o 7º andar do prédio (a porta está aberta, mas fecha automaticamente,  
        // como  
        // nos elevadores que conhecemos).  
        System.out.println("Enviado comando para ir para o 7º andar.");  
        elevator.setCurrentFloor(7);  
  
        // Passageiros saindo do elevador.  
        System.out.println("Sairam pessoas do elevador.");  
        elevator.setWeight(elevator.getWeight() - 180);  
  
        // Desce até o 7º andar do prédio (a porta está aberta).  
        System.out.println("Enviado comando para ir para o 7º andar.");  
        elevator.setCurrentFloor(7);  
  
        // Abre a porta para as pessoas descerem.  
        System.out.println("Enviado comando para abrir a porta.");  
        elevator.openDoor();  
    }  
}
```

Listagem 8. Output gerado pelos testes feitos na classe TestElevator.

```
Abrir a porta para entrada de passageiros.  
- Porta aberta  
Entraram pessoas no elevador.  
- Peso do elevador: 300 Kg.  
Foi acionado o botão de fechar a porta.  
- Porta fechada  
Enviado comando para descer 1 andar.  
- Número de andar errado! O andar mínimo é 0  
Enviado comando para subir 1 andar.  
- Subindo...  
- Andar: 1  
Enviado comando para subir até o 27º andar.  
- Número de andar errado! O andar mais alto é 20  
Enviado comando para subir até o 10º andar.  
- Subindo...  
- Andar: 2  
- Andar: 3  
- Andar: 4  
- Andar: 5  
- Andar: 6  
- Andar: 7  
- Andar: 8  
- Andar: 9  
- Andar: 10  
Enviado comando para abrir a porta.  
- Porta aberta  
Enviado comando para abrir a porta.  
- A porta já se encontra aberta  
Entraram pessoas no elevador.  
- Peso do elevador: 580 Kg.  
Enviado comando para ir para o 7º andar.  
- Fechando a porta...  
- Elevador com mais peso que o permitido!  
- O elevador não pode andar com a porta aberta!  
Sairam pessoas do elevador.  
- Peso do elevador: 400 Kg.  
Enviado comando para ir para o 7º andar.  
- Fechando a porta...  
- Porta fechada  
- Descendo...  
- Andar: 9  
- Andar: 8  
- Andar: 7  
Enviado comando para abrir a porta.  
- Porta aberta
```

com sucesso o encapsulamento de nossa classe. Além disso, criamos uma interface pública simples, acessível a todos e com uma lógica totalmente local (ou seja, as validações são feitas na própria classe), utilizando variáveis privadas (protegidas contra modificações indesejáveis).

Uma classe que implemente com sucesso o encapsulamento não pode permitir que o objeto criado execute operações inválidas. Para alcançar esse objetivo podemos revogar o acesso direto aos atributos da classe e alguns dos seus métodos utilizando o modificador privado e criar métodos públicos (a interface pública) que contenham a lógica necessária para fazer com que todas as operações possíveis de serem executadas possam ser invocadas com segurança.

Foi isso o que vimos com o nosso estudo de caso do elevador. A partir dele, facilmente poderíamos adicionar mais lógica de modo a criar o comportamento de um elevador real em nosso programa como, por exemplo, fazer com que a porta se abra automaticamente sempre que ele chegue ao seu andar de destino. Experimente!

Depois da leitura desse artigo fica a pergunta: o seu código implementa encapsulamento com sucesso? Com este conteúdo você adquiriu os conhecimentos necessários para implementar encapsulamento e tornar o seu código ainda mais seguro. Sendo assim, mãos à obra e caso seja necessário, boa refatoração!

Autor



José Fernandes A. Júnior

jfajunior@gmail.com

Mestre em Engenharia Informática pela Faculdade de Ciências e Tecnologias da Universidade Nova de Lisboa (FCT-UNL).

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



Links:

Encapsulation is not information hiding.

<http://www.javaworld.com/article/2075271/core-java/encapsulation-is-not-information-hiding.html>

Conhecimento faz diferença!

The advertisement features several magazine covers from the 'Engenharia de Software' series. One cover highlights 'Gerência de Configuração' (Configuration Management) with a red figure and puzzle pieces. Another cover shows a hand holding a small plant, with the title 'Evolução do Software' (Software Evolution). A third cover features a robot and the title 'Automação de Testes' (Testing Automation). A large red starburst graphic in the bottom right corner contains the text '+ de 290 vídeos para assinantes' (More than 290 videos for subscribers).

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

Faça um upgrade em sua carreira

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



DEV MEDIA

Dicas de qualidade para o código Java

Programando código limpo e de fácil manutenção

O assunto melhores práticas pode ser aplicado em diversas etapas de um projeto de software, como na fase de desenvolvimento, testes, documentação, gerenciamento do projeto, etc. Neste artigo, este assunto será voltado para o ponto de vista do desenvolvedor. Assim, serão exibidas algumas das principais práticas para se obter um código limpo, de fácil compreensão.

Como você já deve ter ouvido falar, escrever um bom código é extremamente importante, pois com o tempo um código mal estruturado pode levar a uma complexidade excessiva no desenvolvimento de novos módulos e na manutenção, criando um projeto em que desenvolvedores não queiram trabalhar ou tenham medo de alterar algo e fazer uma funcionalidade não relacionada parar de funcionar.

Outro ponto muito importante em se criar um bom código desde o início é que geralmente a estrutura inicial do projeto é replicada, sofrendo apenas algumas alterações para acomodar uma nova funcionalidade. Por exemplo, se o projeto já possui um mecanismo de geração de relatório, infelizmente, mesmo que não seja um mecanismo adequado, um novo desenvolvedor ao ser solicitado para adicionar um novo relatório, provavelmente irá apenas copiar o relatório existente, alterando alguns parâmetros, aumentando a quantidade de código não muito bem escrito. Existem muitos motivos para que isso aconteça: prazos curtos, backlogs grandes, desmotivação, inexperiência, além de muitos outros.

Independentemente do motivo, um código ruim, mesmo que sem bugs, muito provavelmente irá gerar problemas de manutenção, que farão com que o custo a longo prazo provavelmente não justifique a pressa inicial, agravando ainda mais o backlog, a desmotivação e os prazos.

Como mencionado, existem inúmeras práticas que tornam um código melhor. Neste artigo, serão abordados os tópicos de não repetição de código, testes unitários, comentários e tratamento de exceções.

Para que o conteúdo seja bem compreendido, é interessante que o leitor conheça pelo menos conceitos

Fique por dentro

Neste artigo serão dadas algumas dicas básicas para que o desenvolvedor possa criar um software mais robusto e fácil de manter. Serão abordadas algumas práticas de desenvolvimento que podem até mesmo parecerem óbvias, como comentários, tratamento de exceções, testes unitários e a não repetição de código, porém que se aplicadas de forma correta, são muito úteis e podem fazer a diferença entre um código de referência com boa manutibilidade, reutilizável, e algo que pode se tornar um pesadelo no momento que alguma alteração ou correção, pelo próprio criador da funcionalidade ou outro desenvolvedor, seja necessária.

básicos do framework JUnit, utilizado como ferramenta para os testes unitários. Pode ser adotada qualquer ferramenta de testes unitários, porém no exemplo fornecido o JUnit foi o escolhido. O uso deste framework é recomendável porque possui um grande número de usuários, integração nativa com as principais IDEs (Eclipse e NetBeans) e é de uso muito fácil, apresentando uma barra verde para cada teste correto e uma vermelha para cada teste com erro.

É interessante também o conhecimento básico de como utilizar Javadoc, que é a forma padrão de documentação através de comentários em código Java, sendo utilizado pelo próprio Java para documentar sua API. Este também possui integração nativa com as principais IDEs e é uma ferramenta completa para documentação do fonte.

Evite código repetido

Copiar e colar código ou escrever códigos diferentes com uma mesma funcionalidade é algo que muitos desenvolvedores fazem, podendo ser encontrado em muitos projetos. Como mencionado, isto pode acontecer por vários motivos, como: falta de tempo, por medo, preguiça, por não perceber que é possível alterar algo para que se este torne genérico, desconhecimento de que algo semelhante já existe no projeto, entre outros motivos.

De qualquer forma, sempre que uma funcionalidade estiver implementada mais de uma vez, a correção de bugs e implementação de melhorias se torna muito mais complicada, pois tudo que

precisar ser alterado em um trecho do fonte, também precisará ser alterado no outro.

Como um exemplo bem simples, imagine que um desenvolvedor criou um método utilitário para verificar se as entradas de texto das telas estão em branco ou nulas, conforme a **Listagem 1**, e que este método seja utilizado na tela em que este desenvolvedor esteja trabalhando, no caso representada pela classe **Tela1**.

Imagine agora que outro desenvolvedor não tenha conhecimento deste método utilitário de validação e resolva criar em sua classe o mesmo tipo de validação para a tela que está desenvolvendo, representada pela classe **Tela2**, na **Listagem 2**.

Listagem 1. Método utilitário para validação de campo String vazio.

```
public class ValidadorFormulario {  
    /**  
     * @param texto  
     *      - o texto a ser validado  
     * @return true caso o texto seja válido, ou seja, não está vazio  
     */  
    public static boolean validarEntradaTextoVazia(String texto) {  
        return texto != null && !texto.isEmpty();  
    }  
  
    public class Tela1 {  
        // Campo da tela a ser validado  
        private String nome;  
  
        public void validarTela() {  
            if (!ValidadorFormulario.validarEntradaTextoVazia(nome)) {  
                // tratar erro  
            }  
        }  
    }  
}
```

Listagem 2. Validação de campo String vazio sem utilizar método utilitário.

```
public class Tela2 {  
    // Campo da tela a ser validado  
    private String nome;  
  
    public void validarTela() {  
        if (nome == null || nome.isEmpty()) {  
            // tratamento do erro  
        }  
    }  
}
```

Este código não possui problemas e realiza bem o trabalho desejado, possuindo o mesmo efeito final que a validação criada para a classe da **Tela1**. Porém, imagine que a equipe de requisitos altere a checagem dos campos de texto, por exemplo, adicionando mais uma regra à validação, fazendo com que campos que contenham somente espaços sejam considerados como campos em branco também. O desenvolvedor poderia simplesmente alterar o método **validarEntradaTextoVazia()**, deixando-o como exibido na **Listagem 3**.

No entanto, caso ele se esqueça de alterar também a validação efetuada na classe da **Tela2**, esta tela possuirá um comportamento diferente das demais que fazem uso do utilitário **Vali-**

dadorFormulario e caso não existam testes automatizados para efetuar esta checagem em todas as telas e a equipe de testes não efetue novamente um teste em todas telas a cada nova alteração, este erro poderia chegar até o cliente.

Com um exemplo bem simples, foi possível demonstrar um dos principais problemas em possuir código duplicado no projeto. O conceito de não repetir código é de extrema importância no desenvolvimento de software, e para que seja possível aplicar esta prática com sucesso, cada funcionalidade dentro de um sistema deve estar em um local único e bem definido, tornando possível descobrir de forma fácil o que já existe no sistema. Este conceito está presente em diversos pontos do desenvolvimento de software, como no uso de templates em telas, bibliotecas e outros.

Listagem 3. Nova regra adicionada ao método de validação.

```
public class ValidadorFormulario {  
    /**  
     * @param texto  
     *      - o texto a ser validado  
     * @return true caso o texto seja válido, ou seja, não está vazio  
     */  
    public static boolean validarEntradaTextoVazia(String texto) {  
        return texto != null && !texto.trim().isEmpty();  
    }  
}
```

Testes unitários

Testes unitários (testes específicos para cada método público de uma classe a fim de garantir que cada método funciona da forma desejada) são sempre lembrados pelos desenvolvedores como sendo algo importante, porém se o prazo para o desenvolvimento estiver curto, um dos primeiros itens a serem eliminados do projeto são eles. Isto geralmente ocorre porque os testes são algo que acabam não sendo visíveis na entrega de um produto para o cliente.

Geralmente adota-se a filosofia de que depois que a implementação estiver finalizada e entregue, pode-se fazer os testes unitários com calma. No entanto, isto dificilmente funciona, pois geralmente após o término, ou mesmo antes do término de um projeto com prazo curto, o desenvolvedor acaba recendo outro projeto, muitas vezes com prazo curto também. Por este motivo, o uso do TDD (ver **BOX 1**) deve ser fortemente considerado, mesmo em projetos com o prazo apertado. Caso a equipe não esteja confortável com o uso do TDD, pode-se considerar escrever o teste logo após a criação de uma classe, por exemplo, desde que os testes unitários não sejam esquecidos.

BOX 1. TDD

O TDD (Test Driven Development), em resumo, consiste em criar o teste unitário antes de desenvolver uma funcionalidade. Assim, primeiramente o desenvolvedor cria o teste para certa funcionalidade (initialmente o teste falha, pois a implementação nem mesmo existe ainda), depois implementa a funcionalidade da forma mais simples possível para que o teste passe e como terceiro passo refatora a implementação para que esta se torne mais robusta. Estes passos são seguidos para cada método/classe até que a nova funcionalidade esteja inteiramente implementada, resultando em um código bem estruturado e com uma boa cobertura de testes.

A importância destes testes não está apenas no fato de garantir que certo trecho de código esteja funcionando da maneira correta. Outros efeitos colaterais dos testes unitários são que eles também servem como uma forma de documentação de como os métodos públicos podem ou devem ser utilizados e principalmente, permitem que o código seja alterado por um desenvolvedor que não conheça profundamente os detalhes do fonte, pois se as alterações não quebrarem os testes unitários e estes estiverem bem escritos, tem-se um bom indicador de que a alteração efetuada não afetou de forma negativa o código já existente. Utilizando a classe desenvolvida na **Listagem 1**, pode-se criar os testes unitários exibidos na **Listagem 4**.

Ao efetuar a alteração no método validador (modificação que faz com que espaços em branco sejam considerados como campo vazio), pode-se atualizar a classe de testes unitários para inserir um novo teste (no caso do TDD, primeiro seria alterada a classe de testes antes de alterar a implementação) e assim verificar se a nova implementação também está correta. O novo método de teste é exibido na **Listagem 5**.

Listagem 4. Teste unitário para método validador de Strings vazias.

```
public class TesteValidacaoFormulario {  
    @Test  
    public void testeValidarEntradaTextoVazia() {  
        Assert.assertFalse(ValidadorFormulario.validarEntradaTextoVazia(""));  
    }  
  
    @Test  
    public void testeValidarEntradaTextoVaziaNull() {  
        Assert.assertFalse(ValidadorFormulario.validarEntradaTextoVazia(null));  
    }  
  
    @Test  
    public void testeValidarEntradaTextoVaziaPreenchida() {  
        Assert.assertTrue(ValidadorFormulario.validarEntradaTextoVazia("Pedro"));  
    }  
}
```

Listagem 5. Novo teste para verificar alteração feita no método validador de Strings vazias.

```
    @Test  
    public void testeValidarEntradaTextoVaziaBranco() {  
        Assert.assertFalse(ValidadorFormulario.validarEntradaTextoVazia(" "));  
    }
```

Caso todos os testes passem, podemos garantir, com uma razoável confiança, que a alteração no fonte não afetou o comportamento pré-existente e que a nova implementação também está funcionando.

Em um código simples como este, o uso de testes unitários pode parecer desnecessário, porém imagine um cenário mais complexo como, por exemplo, ter que adicionar uma nova regra de cálculo, em um mecanismo de cálculo de rotas de entrega de mercadorias. Sem uma forma rápida e fácil de testar, uma alteração no fonte poderia impactar negativamente no código existente e nem ser notada.

Comentários

A ideia de que quanto mais comentários em um fonte melhor o entendimento do mesmo, não é exatamente a melhor a ser adotada. A presença de muitos comentários, principalmente no meio de métodos, pode acabar poluindo o fonte, não agregando valor real e gerando um maior trabalho de manutenção em futuras alterações, o que são coisas exatamente opostas ao que um bom comentário deve gerar.

O ideal é que utilizando nomes de variáveis, métodos e classes apropriados, o fonte não necessite de tantos comentários, pois, através do uso de uma nomenclatura descriptiva, é possível ler o fonte e facilmente entender o que está escrito. Um exemplo desta prática pode ser observado na **Listagem 6**. A classe **Rota** possui dois métodos que executam a mesma coisa, porém note como é mais fácil entender o que o segundo método faz quando comparado ao primeiro.

Além de nomes apropriados, outra forma de documentação que faz com que os comentários sejam dispensáveis é o uso de testes unitários, como mencionado anteriormente, no tópico “Testes unitários”. Porém, existem casos em que o código de um método ou uma classe, por exemplo, pode não estar tão claro ou fácil de entender, mesmo utilizando uma nomenclatura adequada, ou também, em momentos em que seja necessário documentar certas suposições. Nesses casos é recomendável o uso de comentários que sejam sucintos, mas que expliquem o necessário, não descrevendo o que o código faz linha a linha, e sim explicando o que o código pretende atingir de uma forma mais geral, como no exemplo da **Listagem 7**.

Listagem 6. Exemplo do uso de nomes descriptivos que auxiliam na leitura do fonte.

```
public class Rota {  
    public int calcular(Local o, Local d) {  
        // implementação  
        return -1;  
    }  
  
    public int calcularCustoDaRota(Local origem, Local destino) {  
        // implementação  
        return -1;  
    }  
}
```

Listagem 7. Exemplo de uso recomendado de comentários.

```
public int calcularCustoDaRota(Local origem, Local destino) {  
    // implementação  
  
    // Neste trecho, através do uso do algoritmo XYZ, será calculada o custo  
    // da rota entre a origem e o destino  
  
    // implementação  
  
    return -1;  
}
```

A documentação através de Javadocs também é altamente recomendada, como no exemplo exibido na **Listagem 8**. Contudo, seu uso não exclui a forma de comentários explicativos mencionada

anteriormente, sendo, portanto, um complemento a estes, já que os Javadocs aparecem antes de classes, interfaces, atributos e métodos, e não dentro de métodos, como os comentários explicativos.

Listagem 8. Exemplo do uso de Javadocs.

```
/*
 * Um GeradorImagem é capaz de transformar uma lista de pontos em uma
 * imagem.
 *
 * @author Pedro
 *
 */
public interface GeradorImagen {
/*
 * Método responsável por gerar uma imagem a partir de uma lista de pontos
 * aleatórios.
 *
 * @param pontos
 * - lista de pontos que serão utilizados para gerar a imagem
 * @return imagem gerada
 */
public byte[] gerarImagen(List<Point> pontos);
}
```

Por possuir integração com a maioria das IDEs (Eclipse, NetBeans, etc.), quando um método, por exemplo, possui comentários em Javadoc, apenas posicionando o mouse sobre o método (**Figura 1**), podemos ver a descrição sobre ele. Outra vantagem ao adotar Javadoc é que é possível, de forma automática, gerar uma documentação baseada nele em formato HTML.

O uso do Javadoc se torna mais importante caso o código sendo desenvolvido seja uma biblioteca a ser utilizada por outros projetos, facilitando bastante o uso desta biblioteca quando classes, interfaces e métodos públicos estão devidamente comentados.

Tratamento de exceções

O correto tratamento de exceções também é uma tarefa de extrema importância. Com um tratamento de exceções adequado, o diagnóstico de um bug em desenvolvimento e produção pode

se tornar muito mais fácil, ao passo que com um tratamento inadequado, o diagnóstico de certo erro pode se tornar uma tarefa trabalhosa de tentativa e erro e muito debug.

Como regra geral, checked exceptions devem ser criadas apenas quando é possível que através do tratamento do erro o sistema se recupere do problema. Checked exceptions são aquelas exceções que herdam da classe `java.lang.Exception` e que necessariamente devem ser tratadas em um bloco `try-catch` ou devem ser lançadas utilizando `throw`. Como exemplo de uso desse tipo de exceção, pode-se citar: o erro na entrada de dados do programa, o erro em alguma operação de entrada/saída e outros em que o usuário ou o próprio sistema possa intervir para corrigir o problema.

O código da **Listagem 9** ilustra esta situação. Neste exemplo foi utilizada uma exceção do próprio Java, que também herda da classe `java.lang.Exception`. Como o usuário pode intervir para corrigir o problema, uma mensagem apropriada é exibida para que ele tome uma ação. Vale notar que neste exemplo, o modo ideal para tratar a exceção provavelmente não seria exibir a mensagem diretamente no bloco `catch`, porém está desta forma apenas para ilustrar o uso do tratamento da exceção.

Listagem 9. Exemplo de tratamento de uma checked exception.

```
public class LeitorArquivo {

    public FileReader abrirArquivo(String nomeArquivo) {
        File arquivo = new File(nomeArquivo);
        try {
            return new FileReader(arquivo);
        } catch (FileNotFoundException e) {
            System.out.println("Arquivo não encontrado, fornecer caminho correto.");
            return null;
        }
    }
}
```

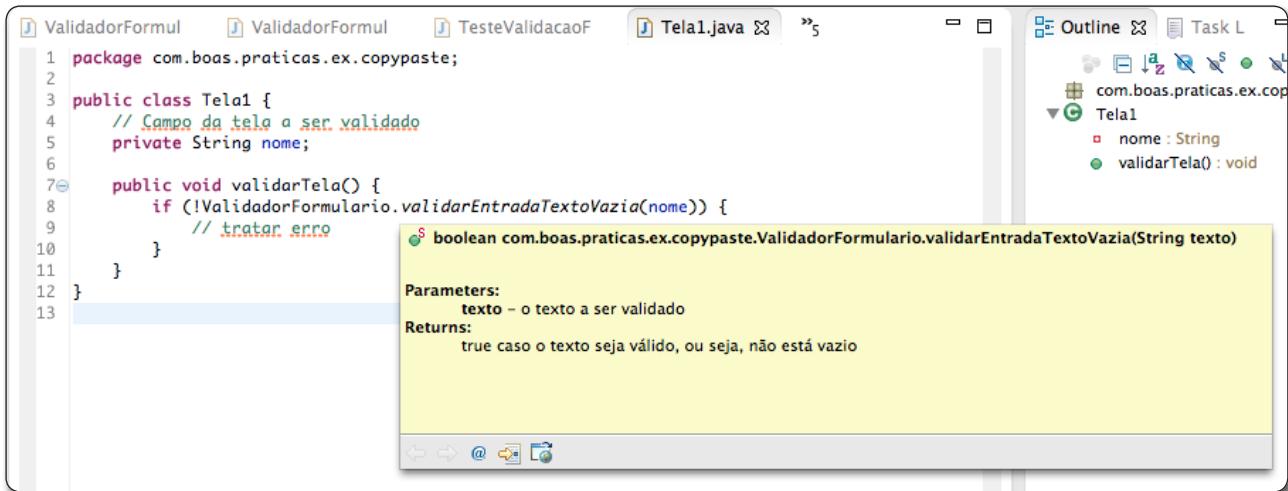


Figura 1. Eclipse exibindo Javadoc do método `validarEntradaVazia()`

As exceções lançadas que geralmente um sistema não consegue se recuperar como, por exemplo, um erro ao executar uma query no banco de dados, ou um erro de codificação em que um método de uma variável nula é acessado (gerando o famoso `java.lang.NullPointerException`), ou uma classe que se encontra em um estado inconsistente é utilizada, devem ser criadas como unchecked exceptions. Este tipo de exceção herda da classe `java.lang.RuntimeException` e não existe obrigatoriedade por parte da linguagem de tratá-la em um bloco try-catch nem de lançá-la. Como regra geral, as `RuntimeExceptions` devem ser tratadas apenas na camada de apresentação, para informar o usuário que algo deu errado e logar o erro de forma apropriada conforme exibido na **Listagem 10**. Neste exemplo, caso as variáveis **origem** e **destino** sejam iguais, uma `RuntimeException` é lançada, pois considera-se que a origem e o destino não podem ser iguais. Este exemplo supõe que o próprio sistema escolha um valor para estas variáveis, isto é, que o usuário não possua controle sobre isso, e que o método `tracarRota()` seja acionado diretamente por um evento da tela. Novamente, a forma ideal de tratar a exceção provavelmente não seria exibir a mensagem desta forma diretamente no bloco `catch`, porém ela foi empregada no exemplo para ilustrar o uso do tratamento da exceção.

Listagem 10. Exemplo de uso de uma `RuntimeException`.

```
public class Rota {  
  
    public int calcularCustoDaRota(Local origem, Local destino) {  
        if (origem.equals(destino))  
            throw new IllegalArgumentException(  
                "origem e destino não podem ser iguais");  
  
        // implementação  
        return -1;  
    }  
  
}  
  
public class CamadaApresentacao {  
  
    public void tracarRota() {  
        try {  
            // implementação ...  
            new Rota().calcularCustoDaRota(getOrigem(), getDestino());  
            // implementação ...  
        } catch (Exception e) {  
            e.printStackTrace();  
            System.out.println("Ocorreu um erro ao traçar a rota,  
                favor enviar o log ao suporte.");  
        }  
    }  
  
    // implementação ...  
}
```

```
public class ErroCalculoCustoRota extends RuntimeException {  
  
    private static final long serialVersionUID = 1703318755539012005L;  
  
    private final Local origem;  
    private final Local destino;  
  
    public ErroCalculoCustoRota(String mensagem, Local origem, Local destino) {  
        super(mensagem);  
        this.origem = origem;  
        this.destino = destino;  
    }  
  
    public Local getOrigem() {  
        return origem;  
    }  
  
    public Local getDestino() {  
        return destino;  
    }  
  
}  
  
public class Rota {  
    public int calcularCustoDaRota(Local origem, Local destino) {  
        if (origem.equals(destino))  
            throw new ErroCalculoCustoRota(  
                "origem e destino não podem ser iguais", origem, destino);  
  
        // implementação  
        return -1;  
    }  
  
}  
  
public class CamadaApresentacao {  
    public void tracarRota() {  
        try {  
            // implementação ...  
            new Rota().calcularCustoDaRota(getOrigem(), getDestino());  
            // implementação ...  
        } catch (ErroCalculoCustoRota erroCalculoCustoRota) {  
            erroCalculoCustoRota.printStackTrace();  
            System.out.println("Ocorreu um erro ao calcular  
                o custo da rota, favor enviar o log ao suporte.");  
            System.out.println("Local de origem: " + erroCalculoCustoRota.getOrigem());  
            System.out.println("Local de destino: " + erroCalculoCustoRota.getDestino());  
        } catch (Exception e) {  
            e.printStackTrace();  
            System.out.println("Ocorreu um erro ao traçar a rota,  
                favor enviar o log ao suporte.");  
        }  
    }  
  
    // implementação ...  
}
```

Outra prática que recomenda-se adotar é a de utilizar exceções já existentes sempre que possível (como realizado na **Listagem 10**), visando evitar a duplicação de código e também facilitar a compreensão do erro, já que as exceções mais comuns, como `IllegalArgumentException` e `NullPointerException`, geralmente já têm

seu significado conhecido pelos desenvolvedores. A vantagem no caso da criação de novas exceções é a de que pode-se adicionar novos atributos e métodos, tornando a exceção mais expressiva, como ocorre na **Listagem 11**. Neste exemplo, com a criação de uma nova exceção, é possível adicionar ao log mais detalhes sobre o erro, como o local de origem e destino que deram origem ao problema.

Listagem 11. Exemplo de uma nova exceção.

```
public class ErroCalculoCustoRota extends RuntimeException {  
  
    private static final long serialVersionUID = 1703318755539012005L;  
  
    private final Local origem;  
    private final Local destino;  
  
    public ErroCalculoCustoRota(String mensagem, Local origem, Local destino) {  
        super(mensagem);  
        this.origem = origem;  
        this.destino = destino;  
    }  
  
    public Local getOrigem() {  
        return origem;  
    }  
  
    public Local getDestino() {  
        return destino;  
    }  
  
}
```

Além da criação de novas exceções desnecessárias, algo que também deve ser evitado ao máximo e que pode causar muita dor de cabeça é a captura de exceções sem efetuar nenhum tratamento, como exibido na **Listagem 12**. Este tipo de código pode tornar o diagnóstico do problema difícil, pois acaba camuflando o problema ao não tratar de forma correta a exceção.

Outra prática comum que deve ser evitada é o tratamento do mesmo erro em mais de um ponto do código. É comum capturar uma checked exception e transformá-la em unchecked exception quando não há uma forma razoável de se recuperar de algumas exceções como, por exemplo, erros relacionados ao banco de dados ou problemas de rede. Nestes casos, alguns desenvolvedores podem acabar imprimindo o stack trace mais de uma vez, uma no momento em que a checked exception é capturada e outra no momento em que a unchecked exception é capturada. Isto é ruim porque gera logs duplicados, poluindo o log e atrapalhando o diagnóstico, como pode se observar na **Listagem 13**.

Listagem 12. Exemplo de captura de exceção sem tratamento.

```
public class CamadaApresentacao {  
  
    public void tracarRota() {  
        try {  
            // implementação ...  
            new Rota().calcularCustoDaRota(getOrigem(), getDestino());  
            // implementação ...  
        } catch (Exception e) {  
        }  
        // implementação ...  
    }  
}
```

Listagem 13. Exemplo de tratamento incorreto de uma exceção.

```
public class TesteTratamentoDuploExcecao {  
  
    private void salvarDados() throws SQLException {  
        throw new SQLException(); // criado apenas para simular a exceção  
    }  
  
    public void executarTransacaoSalvar() {  
        try {  
            salvarDados();  
        } catch (SQLException e) {  
            e.printStackTrace(); // stack trace não deveria ser exibido neste  
            // ponto pois a exceção sera relançada como  
            // uma unchecked exception  
            throw new RuntimeException(e);  
        }  
    }  
  
    public static void main(String[] args) {  
        try {  
            new TesteTratamentoDuploExcecao().executarTransacaoSalvar();  
        } catch (Exception e) {  
            e.printStackTrace(); // tratamento de todas exceções lançadas para  
            // camada do usuário  
        }  
    }  
}
```

Quando ocorrer a exceção no método `salvarDados()` da **Listagem 13**, o log de erro apresentará de forma duplicada o lançamento da `SQLException` (linhas 01 e 08), tornando a leitura do erro confusa, como pode-se observar na **Listagem 14**.

Listagem 14. Exemplo de stack trace duplicado devido ao tratamento incorreto da exceção.

```
01 java.sql.SQLException  
02 at com.boas.praticas.ex.excecao.TesteTratamentoDuploExcecao.salvarDados  
    (TesteTratamentoDuploExcecao.java:7)  
03 at com.boas.praticas.ex.excecao.TesteTratamentoDuploExcecao.executarTransacaoSalvar  
    (TesteTratamentoDuploExcecao.java:12)  
04 at com.boas.praticas.ex.excecao.TesteTratamentoDuploExcecao.main  
    (TesteTratamentoDuploExcecao.java:23)  
05 java.lang.RuntimeException: java.sql.SQLException  
06     at com.boas.praticas.ex.excecao.TesteTratamentoDuploExcecao.  
executarTransacaoSalvar  
    (TesteTratamentoDuploExcecao.java:17)  
07 at com.boas.praticas.ex.excecao.TesteTratamentoDuploExcecao.main  
    (TesteTratamentoDuploExcecao.java:23)  
08 Caused by: java.sql.SQLException  
09 at com.boas.praticas.ex.excecao.TesteTratamentoDuploExcecao.salvarDados  
    (TesteTratamentoDuploExcecao.java:7)  
10 at com.boas.praticas.ex.excecao.TesteTratamentoDuploExcecao.executarTransacaoSalvar  
    (TesteTratamentoDuploExcecao.java:12)  
11 ... 1 more
```

Caso o stack trace não tivesse sido impresso na captura da `SQLException`, o log ficaria como na **Listagem 15**, em que é exibida apenas uma vez a `SQLException` e o método que deu a origem à exceção (`salvarDados()`). Comparando os dois stack traces, é mais fácil perceber através da **Listagem 15** que a `RuntimeException` teve como origem uma `SQLException` (linha 01), e que o método com erro foi o `salvarDados()` (linha 05), que lançou a exceção `SQLException` (linha 04).

Listagem 15. Exemplo de stack trace quando não há duplicação no tratamento do erro.

```
01 java.lang.RuntimeException: java.sql.SQLException  
02 at com.boas.praticas.ex.excecao.TesteTratamentoDuploExcecao.executarTransacaoSalvar  
    (TesteTratamentoDuploExcecao.java:17)  
03 at com.boas.praticas.ex.excecao.TesteTratamentoDuploExcecao.main  
    (TesteTratamentoDuploExcecao.java:23)  
04 Caused by: java.sql.SQLException  
05 at com.boas.praticas.ex.excecao.TesteTratamentoDuploExcecao.salvarDados  
    (TesteTratamentoDuploExcecao.java:7)  
06 at com.boas.praticas.ex.excecao.TesteTratamentoDuploExcecao.executarTransacaoSalvar  
    (TesteTratamentoDuploExcecao.java:12)  
07 ... 1 more
```

As técnicas exibidas neste artigo são apenas para que o leitor comece a pensar e estudar sobre a qualidade do código que escreve e como isto pode afetar a equipe de desenvolvedores.

Dicas de qualidade para o código Java

Estas práticas e muitas outras podem ser utilizadas para se alcançar um código bem escrito. Infelizmente, desenvolvedores com menos experiência podem não perceber rapidamente a importância de se adotar boas práticas. Geralmente esta percepção ocorre quando ele, ao ler seu próprio código ou o código de seus colegas, nota que a legibilidade deste pode ser melhorada. Portanto, como diferencial, não espere que isso aconteça, pois a depender do estágio do projeto pode ser muito difícil obter a qualidade esperada.

Uma das principais diferenças entre um código legível, bem estruturado, de boa manutenção, e um código difícil de ser entendido e alterado, na maioria das vezes, consiste em pequenos detalhes como os demonstrados no artigo. Quando deixados de lado, tais detalhes podem se tornar um problema para a própria equipe de desenvolvimento e também para o cliente, pois com um fonte de difícil manutenção, a correção de bugs e inclusão de novas funcionalidades se tornam sempre mais complexas.

Autor



Toyoaki Ejiri

toyoaki.ejiri@gmail.com

Formado em Engenharia da Computação pela POLI-USP, trabalha com Java há 7 anos. Possui as certificações SCJP e SCWCD.



Links:

Site do JUnit.

<http://junit.org/>

Javadoc Tool Home Page.

<http://www.oracle.com/technetwork/java/javase/documentation/javadoc-137458.html>

FÓRUM DEVMEDIA

O lugar perfeito para você ficar por dentro
de tudo o que acontece nas tecnologias do
mercado atual

No fórum da DevMedia você irá encontrar uma equipe disponível e altamente qualificada com consultores e colaboradores prontos para te ajudar a qualquer hora e sobre qualquer assunto. Temos as salas de Java, .NET, Delphi, Banco de Dados, Engenharia de Software, PHP, Java Script, Web Design, Automação comercial, Ruby on Rails e muito mais!



ACESSE AGORA
www.devmedia.com.br/forum

Java Persistence API (JPA): Explorando o EntityManager

Entenda como funciona o EntityManager e o ciclo de vida das entidades na JPA

AJPA (*Java Persistence API*), especificação Java EE para ORM (mapeamento objeto relacional), tem sido amplamente adotada nos mais diversos sistemas que necessitam de persistência de dados. Esta poderosa API traz uma grande gama de recursos e facilidades, mas exige um estudo aprofundado se estivermos buscando compreender os métodos disponibilizados por ela e extrair o máximo de suas funcionalidades.

Com base nisso, este artigo visa guiar leitores que já tiveram algum contato com a API a um entendimento mais amplo do ciclo de vida das entidades e o funcionamento do PersistenceContext que, como veremos no decorrer desse estudo, é essencial para a compreensão dos métodos do EntityManager aqui estudados: `persist()`, `merge()` e `find()`.

Por ser uma API bastante extensa, esse artigo não tem como objetivo cobrir todas as suas particularidades, mas sim focar nos conceitos e nomenclaturas que irão auxiliar o leitor em seus futuros estudos. Para alcançarmos tal objetivo, os conceitos apresentados serão acompanhados de exemplos, construídos passo a passo, visando encorajar o leitor a codificar juntamente com a leitura. Esses exemplos serão construídos em um projeto criado no Eclipse, mas que facilmente pode ser adaptado para sua IDE de preferência. Neste projeto iremos utilizar o Hibernate como implementação da JPA e o HSQLDB como banco de dados, por ser livre de instalação e configuração. A configuração da JPA, assim como a do Hibernate, será ilustrada aqui, contudo o detalhamento mais profundo da configuração destes está além do escopo desse artigo.

Os exemplos aqui apresentados são baseados na manipulação de uma entidade bastante simples chamada Pessoa, que veremos em detalhes nos próximos tópicos. Nestes exemplos nem sempre iremos exibir a classe inteira na listagem, focando apenas no trecho de código que exemplifica o conceito abordado e que seja suficiente para o leitor entender seu funcionamento.

Fique por dentro

Este artigo será útil para desenvolvedores que já têm contato com a JPA, mas que buscam um entendimento mais abrangente do ciclo de vida das entidades e do PersistenceContext a fim de explorar o máximo dessa API. O entendimento mais abrangente desses conceitos tem como objetivo propiciar um uso mais consciente dos recursos da API, evitando erros comuns e facilitando a resolução de eventuais problemas que surjam no dia a dia.

Além disto, trataremos aqui sobre conceitos não triviais, principalmente para leitores sem contato prévio com a JPA. Desta forma, o leitor que sentir a necessidade de um embasamento inicial maior, recomendamos a leitura do artigo “Persistindo dados em Java com JPA”, publicado na Easy Java Magazine 36.

Criando e configurando o projeto exemplo

Para construirmos os exemplos que irão auxiliar no entendimento dos conceitos discutidos, vamos utilizar o Eclipse como IDE, mas o leitor deve se sentir livre para utilizar a ferramenta de desenvolvimento que esteja mais habituado.

Criando um novo projeto no Eclipse

No momento de inicialização do Eclipse precisamos, primeiramente, selecionar o workspace em que iremos trabalhar. O workspace é um diretório onde serão armazenadas informações dos nossos projetos e preferências do Eclipse. Você pode utilizar o diretório sugerido por ele, se preferir.

Para construirmos nossa aplicação de exemplo precisamos de um novo projeto, que pode ser criado a partir do menu *File > New > Java Project*. Feito isso, defina um nome para o seu projeto. Se preferir, altere o diretório onde seu projeto será armazenado. Por fim, clique em *Finish*.

Dentro do nosso novo projeto temos um diretório com o nome *src*. É neste diretório que iremos criar nossas classes e armazenar nossos arquivos de configuração.

Java Persistence API (JPA): Explorando o EntityManager

Adicionando a JPA e o Hibernate no classpath

O primeiro passo é fazer o download do Hibernate no site oficial do framework, que pode ser consultado na seção [Links](#). A última versão disponível até o momento da escrita deste artigo é a 4.3.5.

Após a descompactação do arquivo (que pode ser baixado nos formatos ZIP ou TGZ), é necessário adicionar os JARs que se encontram no diretório *required* ao classpath da sua aplicação. Além desses, é necessário também adicionar o JAR que se encontra no diretório *jpa*.

Para adicionar esses JARs ao seu projeto no Eclipse, clique com o botão direito no projeto e selecione *Build Path > Configure Build Path*. Na aba *Libraries*, clique em *Add External JARs*. Depois de selecionar os arquivos indicados, a aba *Libraries* do *Build Path* do seu projeto deve se parecer com a **Figura 1**.

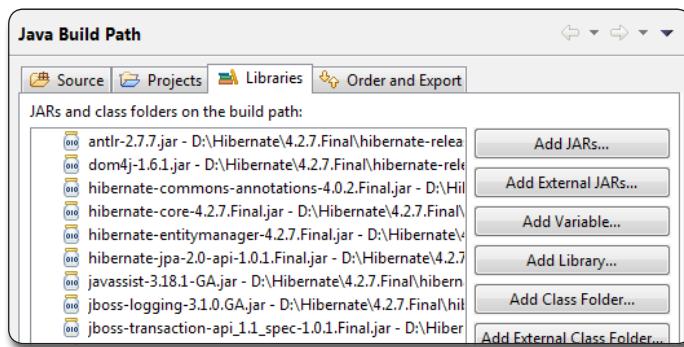


Figura 1. Adicionando os JARs do Hibernate ao projeto no Eclipse

Adicionando o HSQLDB ao classpath

Precisamos agora configurar o banco de dados que iremos utilizar. Para facilitar esta etapa, vamos optar por um banco que armazena os dados em memória, chamado HSQLDB. O fato do banco ser em memória implica que a cada vez que executarmos nossas classes de teste, teremos uma nova instância (vazia) do banco. Para fazer o download do HSQLDB, acesse o site oficial, que pode ser consultado na seção [Links](#). A última versão disponível até o momento da escrita deste artigo é a 2.3.2.

Após a descompactação do arquivo, haverá um arquivo chamado **hsqldb.jar** no diretório *lib*, e este deve ser adicionado ao classpath da sua aplicação. O processo de adicionar esse JAR ao seu projeto no Eclipse é análogo ao apresentado no subtópico anterior.

Configurando o persistence.xml

Toda aplicação que utiliza JPA necessita de um arquivo de configuração com o nome de *persistence.xml*. Esse arquivo contém as configurações gerais da sua aplicação, e deve estar no diretório *META-INF*. Este diretório, por sua vez, tem de estar presente na raiz do classpath da sua aplicação, como ilustrado na **Figura 2**.

O arquivo que vamos utilizar em nossa aplicação de exemplo se encontra na **Listagem 1**. Esse arquivo de configuração está definindo um *persistence-unit* com o nome de "exemplo-pu".

É através desse nome que iremos referenciar essas configurações na nossa aplicação.

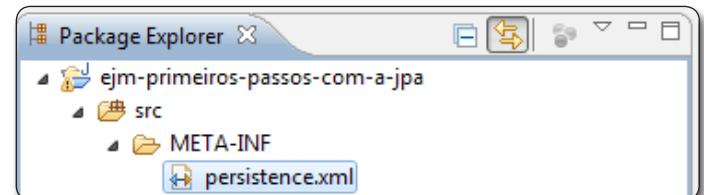


Figura 2. Caminho onde deve ficar o arquivo *persistence.xml*

Listagem 1. Arquivo de configuração *META-INF/persistence.xml*.

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
  http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
  version="2.0">

<persistence-unit name="exemplo-pu" transaction-type="RESOURCE_LOCAL">
<properties>
  <property name="hibernate.hbm2ddl.auto" value="create"/>
  <property name="hibernate.show_sql" value="true"/>
  <property name="hibernate.dialect"
    value="org.hibernate.dialect.HSQLDialect"/>

  <property name="javax.persistence.jdbc.driver"
    value="org.hsqldb.jdbc.JDBCDriver"/>
  <property name="javax.persistence.jdbc.url" value="jdbc:hsqldb:mem:"/>
  <property name="javax.persistence.jdbc.user" value="sa"/>
  <property name="javax.persistence.jdbc.password" value=""/>
</properties>
</persistence-unit>
</persistence>
```

Além do nome do *persistence-unit*, definimos também o *transaction-type* como *RESOURCE_LOCAL*. Esta propriedade define que a JPA será utilizada no modo *standalone*, ou seja, não haverá um container Java EE gerenciando as transações, como veremos no próximo tópico. As demais propriedades presentes nesse arquivo não serão detalhes nesse artigo, mas podem ser encontrados no artigo "Persistindo dados em Java com JPA", publicado na Easy Java Magazine 36.

Criando a entidade Pessoa

Nos exemplos que construiremos no restante desse artigo, iremos utilizar a entidade *Pessoa*, cujo código pode ser observado na **Listagem 2**. Essa entidade, que é bastante simples por motivos didáticos, faz o mapeamento da tabela *Pessoa*, que contém os campos *id* e *nome*. O valor do atributo *id* será gerado automaticamente por uma *sequence*, pois assim está indicado na anotação *@GeneratedValue* que acompanha sua declaração.

Obtendo e utilizando um EntityManager

O *EntityManager* é a interface pela qual manipulamos as entidades da nossa aplicação. Podemos entendê-la como a ponte entre dois mundos: o orientado a objetos e o banco de dados. Essa manipulação das entidades é feita através de métodos, alguns

dos quais estudaremos a seguir, a saber: **persist()**, **merge()** e **find()**. Entretanto, antes de analisarmos os detalhes de cada método, iremos entender como obter uma instância da interface **EntityManager**. A seguir, veremos duas formas de obtermos um **EntityManager**: uma utilizada em ambientes Java EE e outra em ambientes Java SE.

Listagem 2. Código da classe Pessoa.

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Pessoa {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    private Long id;

    private String nome;

    // gets e sets omitidos...
}
```

Em aplicações Java EE, o container gerencia o **EntityManager** de forma transparente para a aplicação, que não necessita se preocupar com sua criação e controle de transação. Somente necessita injetar a dependência do mesmo, através de uma anotação, para que possa utilizá-lo. Por outro lado, em ambientes Java SE (*standalone*), onde não temos, necessariamente, um container, o **EntityManager** é gerenciado pela própria aplicação e dessa forma sua criação necessita ser feita de forma explícita, como veremos a seguir. Como os conceitos apresentados nesse artigo são aplicáveis a ambos os cenários, iremos optar por gerenciar o **EntityManager** pela aplicação, pois assim não necessitamos de um container, o que facilita a execução dos exemplos apresentados aqui.

Para obter um **EntityManager** gerenciado pela aplicação, precisamos de uma instância de **EntityManagerFactory**, que pode ser obtida através da classe **Persistence**. Essa sequência de chamadas para obter um **EntityManager** está ilustrada na **Figura 3**.

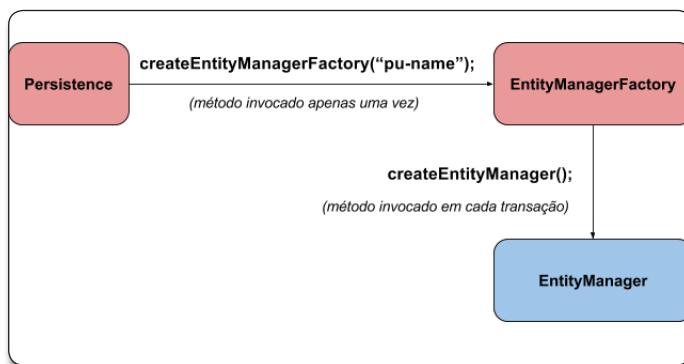


Figura 3. Obtendo um EntityManager

Dessa forma, o primeiro passo é obter um **EntityManagerFactory** a partir do método **Persistence.createEntityManagerFactory()**, como mostra a **Listagem 3**. Este método recebe como argumento o nome do **persistence-unit** que informamos no arquivo *persistence.xml*. Em nosso exemplo foi atribuído a ele o nome **exemplo-pu**.

No momento da chamada desse método o Hibernate varrerá o JAR onde se encontra o arquivo *persistence.xml* procurando por classes que tenham a annotation **@Entity**. Esse processo pode levar alguns segundos, dependendo do número de entidades que sua aplicação tenha. No entanto, apesar do tempo gasto pelo método **createEntityManagerFactory()**, essa chamada não caracteriza um problema de performance, pois podemos invocá-la apenas uma vez durante toda a execução da aplicação e armazenar seu retorno em uma variável estática.

Com o **EntityManagerFactory** em mãos, podemos então criar instâncias do **EntityManager**, com as quais iremos manipular nossas entidades. O processo de criar o **EntityManager** está exemplificado na **Listagem 4**.

Listagem 3. Obtendo um EntityManagerFactory.

```
public class JPAUtil {

    private static EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("exemplo-pu");

}
```

Listagem 4. Obtendo um EntityManager.

```
public class EntityManagerUtil {

    private static EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("exemplo-pu");

    public static EntityManager em() {
        return emf.createEntityManager();
    }
}
```

Quando optamos por deixar o container gerenciar o **EntityManager**, precisamos informar à JPA que o container irá controlar as transações, ficando transparente para a aplicação esse controle. Esse comportamento deve ser sinalizado à JPA no arquivo *persistence.xml*, através da propriedade **transaction-type**, onde atribuímos o valor **JTA** (*Java Transaction API*). Por outro lado, quando o **EntityManager** é gerenciado pela aplicação, é necessário tratar as transações de forma explícita, ou seja, a aplicação fica responsável por abrir, commitar e eventualmente fazer rollback das transações manualmente.

Para sinalizar que controlaremos as transações pela aplicação, iremos utilizar, novamente, a propriedade **transaction-type**, mas agora atribuindo o valor **RESOURCE_LOCAL**, como vimos na **Listagem 1**. Nesse cenário, o controle das transações pela aplicação pode ser feito como exemplificado na **Listagem 5**, onde um novo

registro da entidade **Pessoa** é inserido no banco de dados. Esse trecho de código faz uso do método **begin()** para indicar o início da transação e do método **commit()** para indicar seu fim. Podemos ainda notar o método **rollback()** sendo invocado quando alguma exceção ocorrer, informando que todas as operações realizadas desde o início da transação devem ser desfeitas.

Listagem 5. Controle explícito de transação (ExemploControleTransacao.java).

```
1. EntityManager em = EntityManagerUtil.em();
2. EntityTransaction tx = null;
3.
4. try{
5.     tx = em.getTransaction();
6.     tx.begin();
7.
8.     Pessoa pessoa = new Pessoa();
9.     pessoa.setNome("Andrei");
10.
11.    em.persist(pessoa);
12.
13.    tx.commit();
14. } catch (RuntimeException e) {
15.     if (tx != null && tx.isActive()) {
16.         tx.rollback();
17.     }
18. } finally {
19.     em.close();
20. }
```

O método **persist()**, que será detalhado nos próximos tópicos, é utilizado para informar ao **EntityManager** que o objeto passado como parâmetro deve ser persistido no banco de dados. Assim, ao analisarmos esse exemplo podemos, prematuramente, concluir que a execução do método **persist()** realiza uma operação de **INSERT** no banco de dados. Entretanto, como veremos nos próximos capítulos, esse não é o comportamento desse método. Para compreendermos seu funcionamento, primeiramente precisamos entender o conceito de **PersistenceContext**.

PersistenceContext

No momento de criação do **EntityManager**, um **PersistenceContext** (contexto de persistência) é associado ao mesmo. O **PersistenceContext** é constituído por um conjunto de entidades, no qual existe a garantia de que o identificador (valor do atributo anotado com **@Id**) de cada objeto seja único. Os objetos que pertencem a esse conjunto de entidades são denominados *managed*, e serão utilizados pelo **EntityManager** quando o método **commit()**, da interface **EntityTransaction**, for invocado. Após a invocação desse método, irá ocorrer a operação de **sincronização** que, baseada nas entidades que estão no **PersistenceContext**, executa instruções SQL necessárias para persistir as alterações no banco de dados.

Nota

A operação de sincronização, que é realizada no método **commit()**, também pode ser invocada através do método **flush()**, presente na interface **EntityManager**.

Inserindo entidades no PersistenceContext

Uma vez que sabemos que somente as entidades associadas ao **PersistenceContext** serão persistidas no banco de dados, precisamos de uma maneira de realizar esta associação. Quando criamos uma nova instância de uma entidade, esta não é automaticamente associada ao **PersistenceContext**, e assim não será persistida. Uma entidade que ainda não foi associada é classificada como entidade *new* e pode ser relacionada ao **PersistenceContext** através do método **persist()**, já demonstrado na **Listagem 5**. A invocação deste método cria a ligação entre a entidade e o **PersistenceContext** e ela, que antes era classificada como *new*, se torna *managed*. O fluxo envolvendo a chamada dos métodos **persist()** e **commit()** pode ser observado na **Figura 4**.

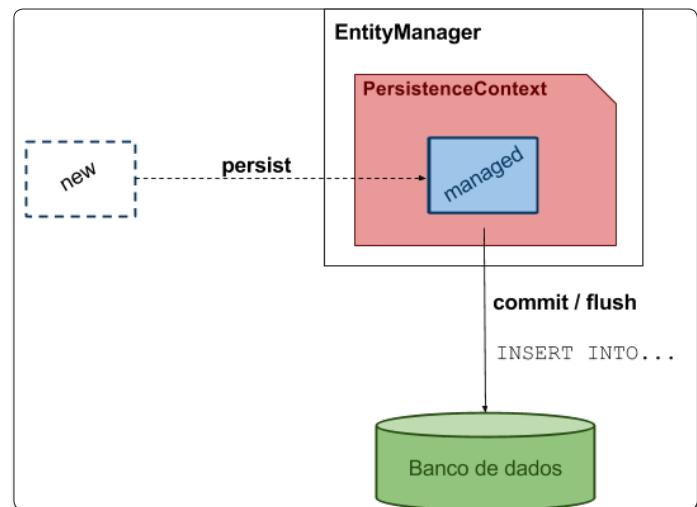


Figura 4. Fluxo das chamadas aos métodos **persist()** e **commit()**

A partir da **Figura 4** podemos observar que a instrução de **INSERT** somente é executada quando invocamos o método **commit()**, e não no momento da chamada do método **persist()**, como poderíamos erroneamente supor. Como forma de verificar esse comportamento, vamos alterar o código da **Listagem 5**, incluindo alguns logs, como mostra a **Listagem 6**. Ao executarmos essa classe, teremos a saída presente na **Listagem 7**, onde podemos notar que a instrução de **INSERT** foi executada no método de **commit()**. É importante ressaltar que as instruções executadas no banco de dados foram exibidas somente por que a propriedade **hibernate.show_sql** está com valor **true** no **persistence.xml**.

Entidades detached

Após a utilização do **EntityManager**, devemos fechá-lo através do método **close()**. Essa operação irá fechar, por consequência, o **PersistenceContext**. Dessa maneira, as entidades associadas ao mesmo, que antes eram *managed*, passam a ser *detached*.

Alterações feitas em entidades *detached* não são propagadas para o banco de dados. No entanto, é comum precisarmos alterar uma entidade que já está persistida no banco dados, mas não está mais presente no **EntityManager**. Por conta disso, uma entidade *detached*

ched pode ser associada a outro **EntityManager** com a invocação do método **merge()**, cuja utilização é exemplificada na **Listagem 8**. Nesse código, criamos uma nova **Pessoa** com o nome “Andrei” e utilizamos o método **persist()**, alterando seu status de *new* para *managed*. Com a finalização do primeiro **EntityManager**, a entidade passa de *managed* para *detached*.

Listagem 6. Verificando a instrução de INSERT no commit (ExemploVerificarInstrucaoInsert.java).

```

1. tx.begin();
2.
3. Pessoa pessoa = new Pessoa();
4. pessoa.setNome("Andrei");
5.
6. System.out.println("Antes persist");
7. em.persist(pessoa);
8. System.out.println("Depois persist");
9.
10. System.out.println("Antes commit");
11. tx.commit();
12. System.out.println("Depois commit");

```

Listagem 7. Trecho do console ao executar o código da Listagem 6.

```

(...) Antes persist
Hibernate: call next value for hibernate_sequence
Depois persist
Antes commit
Hibernate: insert into Pessoa (nome, id) values (?, ?)
Depois commit
(...)

```

Nota

A especificação JPA não determinada o momento em que a instrução de **INSERT** deve ser executada. Ela pode ser executada no momento da execução do **persist()** ou do **commit()**. No Hibernate, isso irá depender da estratégia utilizada para gerar o identificador (atributo anotado com `@Id`).

Em seguida à finalização do primeiro **EntityManager**, alteraremos o nome da pessoa para “Andrei Tognolo”, criamos um novo **EntityManager** e invocamos o método **merge()**, associando a entidade ao novo **PersistenceContext** e alterando seu status de *detached* para *managed* novamente. Feito isso, o próximo método a ser invocado é o **commit()**, da transação do segundo **EntityManager**. Como nesse momento a entidade está no **PersistenceContext**, suas alterações são persistidas no banco de dados por intermédio da instrução **UPDATE**. O log de saída da execução desse trecho de código pode ser observado na **Listagem 9**.

Ao olharmos atentamente para o log de saída na **Listagem 9**, iremos observar uma instrução de **SELECT** executada durante a chamada do método **merge()**. Para entendermos o motivo desse comportamento, primeiramente vamos analisar como recuperar entidades do banco de dados e voltaremos a esse detalhe no subtópico “Outras características do método **merge()**”. O fluxo de execução da **Listagem 8** pode ser visualizado na **Figura 5**.

Listagem 8. Exemplo de uso do método **merge()** – ExemploMerge.java.

```

1. Pessoa p = new Pessoa();
2. p.setNome("Andrei");
3.
4. EntityManager em1 = EntityManagerUtil.em();
5. EntityTransaction tx1 = em1.getTransaction();
6. tx1.begin();
7. em1.persist(p);
8. tx1.commit();
9. em1.close();
10.
11. p.setNome("Andrei Tognolo");
12.
13. EntityManager em2 = EntityManagerUtil.em();
14. EntityTransaction tx2 = em2.getTransaction();
15. tx2.begin();
16. System.out.println("Antes merge");
17. em2.merge(p);
18. System.out.println("Depois merge");
19. System.out.println("Antes commit");
20. tx2.commit();
21. System.out.println("Depois commit");
22. em2.close();

```

Listagem 9. Trecho do console ao executar o código da Listagem 8.

```

(...) Antes merge
Hibernate: select (...) from Pessoa pessoa0_ where pessoa0_.id=?
Depois merge
Antes commit
Hibernate: update Pessoa set nome=? where id=?
Depois commit
(...)

```

Nota

O trecho apresentado na **Listagem 8** faz uso de dois **EntityManager** pois, para exemplificar o comportamento do método **merge()**, utiliza-se duas transações. Métodos que trabalham com uma única transação necessitam apenas de um **EntityManager**.

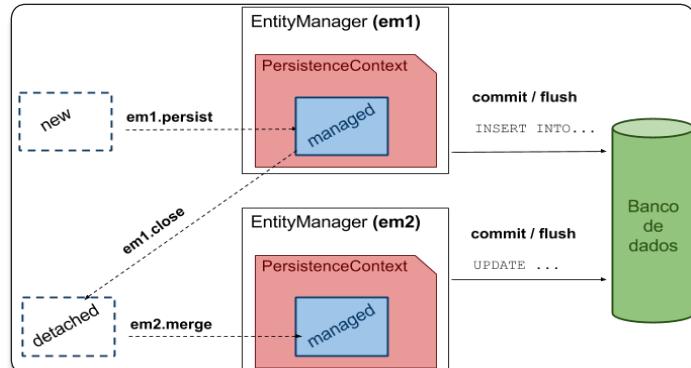


Figura 5. Fluxo de execução da **Listagem 8**

Recuperando entidades do banco de dados

A JPA provê uma série de opções para recuperarmos registros do banco de dados e transformá-los em objetos novamente. Aqui, iremos focar apenas no método **find()**, visto que seu entendimento pode ser utilizado como base para deduzir o comportamento dos

Java Persistence API (JPA): Explorando o EntityManager

outros métodos de busca. Sua assinatura mais simples se encontra na **Listagem 10**, onde o primeiro parâmetro é a classe da entidade que estamos buscando e o segundo é o seu identificador (atributo anotado com `@Id`).

O retorno do método `find()` é o objeto que representa o registro no banco de dados. É importante ressaltar que esse objeto retornado é uma instância *managed*, ou seja, ele está automaticamente associado ao `PersistenceContext`, e assim, qualquer alteração no objeto será propagada novamente para o banco de dados, caso o método `commit()` seja invocado. Para exemplificar esse comportamento, vamos analisar a **Listagem 11** e o resultado de sua execução, presente na **Listagem 12**. Veja que na linha 15 estamos recuperando uma `Pessoa` do banco através do método `find()`, alteramos então seu nome e em seguida invocamos o método `commit()`. Como dissemos, o método `find()` retorna uma instância *managed*, e por conta disso, na execução do método `commit()` a alteração feita no objeto é propagada para o banco de dados, através da instrução UPDATE, como podemos verificar no console de saída, exibido na **Listagem 12**.

Listagem 10. Assinatura do método `find()`.

```
<T> T find(Class<T> entityClass, Object primaryKey)
```

Listagem 11. Entidades recuperadas com `find()` são automaticamente associadas ao `PersistenceContext` – ExemploFind.java.

```
1. Pessoa p = new Pessoa();
2. p.setNome("Andrei");
3.
4. EntityManager em1 = EntityManagerUtil.em();
5. EntityTransaction tx1 = em1.getTransaction();
6. tx1.begin();
7. em1.persist(p);
8. tx1.commit();
9. em1.close();
10.
11. EntityManager em2 = EntityManagerUtil.em();
12. EntityTransaction tx2 = em2.getTransaction();
13. tx2.begin();
14.
15. Pessoa pessoaPersistida = em2.find(Pessoa.class, p.getId());
16. pessoaPersistida.setNome("Andrei Tognolo");
17.
18. System.out.println("Antes commmit");
19. tx2.commit();
20. System.out.println("Depois commmit");
21. em2.close();
```

Listagem 12. Trecho do console ao executar o código da **Listagem 11**.

```
(...)
Hibernate: select (...) from Pessoa pessoa0_ where pessoa0_.id=?
Antes commmit
Hibernate: update Pessoa set nome=? where id=?
Depois commmit
(...)
```

É necessário executar o `merge()` após o `find()`?

O entendimento mais amplo do funcionamento do `PersistenceContext` evita que muitas vezes sejam criados códigos que não estão aderentes aos conceitos estipulados pela JPA. Um exemplo

comum de utilização incorreta dos métodos é a invocação do método `merge()` após obter uma entidade pelo método `find()`, como representado na **Listagem 13**. O método `merge()`, nesse contexto, não está gerando nenhum efeito, pois a execução do método `find()` retornou uma entidade *managed* e assim suas alterações já serão propagadas para o banco de dados no `commit()`. Esse equívoco se dá, principalmente, pelo fato do método `merge()` ser facilmente confundido com a operação de UPDATE.

Listagem 13. Executando `merge()` após o `find()`, um equívoco bastante comum.

```
1. Pessoa p = em.find(Pessoa.class, 100);
2. p.setNome("Andrei Tognolo");
3.
4. em.merge(p);
5. tx.commit();
```

Nota

Quando o `merge()` é executado e a entidade em questão já está no `PersistenceContext`, a operação é ignorada.

Preservando a unicidade de identificadores no método `find()`

Como já foi dito, o `PersistenceContext` é constituído por um conjunto de entidades no qual existe a garantia de que o identificador de cada objeto seja único. Tendo essa garantia em mente, vamos analisar a **Listagem 14**, onde o método `find()` é invocado duas vezes e seu resultado é atribuído a duas variáveis distintas. Após as duas invocações, seria razoável supor que foram criados dois objetos na memória, mas uma análise mais detalhada nos mostrará que apenas um objeto foi criado. Note que, na linha 5, fazemos uma comparação entre as duas variáveis e pelo log de saída apresentado na **Listagem 15** podemos verificar que elas apontam para o mesmo objeto.

Listagem 14. Executando o método `find()` duas vezes sequencialmente – ExemploUnicidadedidentificador.java.

```
(...)
1. System.out.println("Antes de executar primeiro find");
2. Pessoa p1 = em.find(Pessoa.class, id);
3. System.out.println("Antes de executar segundo find");
4. Pessoa p2 = em.find(Pessoa.class, id);
5. System.out.println("p1 == p2 ? " + (p1 == p2));
(...)
```

Listagem 15. Trecho do console ao executar o código da **Listagem 14**.

```
(...)
Antes de executar primeiro find
Hibernate: select (...) from Pessoa pessoa0_ where pessoa0_.id=?
Antes de executar segundo find
Hibernate: select (...) from Pessoa pessoa0_ where pessoa0_.id=?
p1 == p2 ? true
(...)
```

A **Listagem 15** nos mostra também outro importante comportamento do `PersistenceContext`. Podemos notar que somente uma instrução SELECT foi executada no banco de dados, o que

nos permite concluir que o método `find()`, antes de consultar o banco de dados, verifica se a entidade em questão já está presente no `PersistenceContext`, evitando consultas possivelmente desnecessárias. Por conta desse comportamento, dizemos que o `PersistenceContext` tem a função de *cache*, porém restrito a apenas um `EntityManager`. Por conta disso, é denominado como *cache* de primeiro nível. Para criarmos um *cache* que seja compartilhado por mais de um `EntityManager` precisamos utilizar o *cache* de segundo nível, que é associado ao `EntityManagerFactory`. Apesar de ser um assunto de extrema relevância, o estudo de *cache* de segundo nível está além do escopo deste artigo.

Mais detalhes dos métodos de persistência

Além do comportamento já apresentado dos métodos `persist()` e `merge()`, vamos analisar a seguir outras características que podem ajudar o leitor a ter uma compreensão mais amplo sobre eles.

Outras características do método `persist()`

A função do método `persist()` é inserir uma nova entidade no `PersistenceContext`, alterando seu status de *new* para *managed*. Sua assinatura é apresentada na [Listagem 16](#), onde podemos notar que ele tem retorno `void`, pois diferente do que veremos no `merge()`, o próprio objeto passado como parâmetro será inserido no `PersistenceContext`.

Assim, se fizermos alterações em um objeto que foi passado como parâmetro para o método `persist()`, essas alterações serão persistidas no momento da execução do `commit`.

Listagem 16. Assinatura do método `persist()`.

```
void persist(java.lang.Object entity)
```

Outras características do método `merge()`

A função do `merge()` é associar ao `PersistenceContext` uma entidade que uma vez já esteve associada, alterando seu status de *detached* para *managed*. Para isto, o Hibernate realiza duas operações: primeiramente, ele cria um novo objeto com os dados recuperados do banco de dados e a insere no `PersistenceContext`, assim como o comportamento do método `find()`; e após isso, ele atualiza esse novo objeto com os dados da entidade passada por parâmetro. Como consequência desse comportamento, o objeto passado como parâmetro não é inserido no `PersistenceContext`, mas sim o novo objeto. Em seguida este novo objeto é retornado pelo método `merge()`, que explicita isso em sua assinatura, exposta na [Listagem 17](#).

Listagem 17. Assinatura do método `merge()`.

```
<T> T merge(T entity)
```

O comportamento de primeiramente recuperar a entidade do banco de dados e depois atualizar seus dados explica a instrução `SELECT` exibida antes do `UPDATE` na [Listagem 9](#).

A partir do conteúdo aqui abordado o leitor estará apto a entender, por si só, outros métodos relevantes da Java Persistence API, como por exemplo, o `remove()` da interface `EntityManager`. Um dos conceitos centrais para entendimento da JPA é o do ciclo de vida das entidades. O conhecimento mais aprofundado deste serve como base para uma boa compreensão de outros temas centrais da JPA, como o relacionamento entre entidades e as propriedades de *cascade* presentes nestas relações. Portanto, não pare os estudos sobre a JPA por aqui. A busca por novas informações tornará melhor a qualidade do código de seus aplicativos, principalmente no que diz respeito à persistência de dados.

Autor



Andrei de Oliveira Tognolo

andreitognolo@gmail.com

Bacharel em Ciéncia da Computação pela UNICAMP. Tem mais de sete anos de experiência com desenvolvimento de software, utilizando principalmente Java e JavaScript. Criador do Raiden JPA, uma implementação JPA extremamente rápida utilizada em ambientes de desenvolvimento.



Links:

Site oficial do framework Hibernate.

hibernate.org

Site oficial do banco de dados HSQLDB.

hsqldb.org

Página da especificação JPA 2.1.

download.oracle.com/otndocs/jcp/persistence-2_1-fr-eval-spec/index.html

Tutorial da Oracle sobre a Java Persistence API.

<http://docs.oracle.com/javaee/6/tutorial/doc/bnbpz.html>

Uma introdução à JPA.

<http://www.javaworld.com/article/2077817/java-se/understanding-jpa-part-1-the-object-oriented-paradigm-of-data-persistence.html>

Remote Method Invocation: RMI na prática

Desenvolvendo aplicações com objetos distribuídos

Em aplicações distribuídas, é comum organizar os processos em processo servidor e processo cliente, de forma que os processos possam estar localizados em plataformas diferentes, em computadores diferentes. Segundo esse modelo, o processo servidor oferece serviços aos processos clientes, porém a implementação do modelo de comunicação entre tais processos pode ser difícil devido à complexidade natural da utilização de *sockets* ou outras *streams* de comunicação de dados. A complexidade aumenta quando é necessário acessar ou enviar objetos pela internet, pois os mesmos apresentam atributos e métodos que precisam ser convertidos para a forma binária para serem enviados por uma *stream*.

A fim de facilitar a implementação de aplicações distribuídas entre diversas plataformas de sistemas operacionais ou hardware, foi criado o padrão CORBA (*Common Object Request Architecture*), disponibilizado nas linguagens mais adotadas. Sua implementação oferece recursos para serem utilizados para facilitar a comunicação entre objetos de aplicações diferentes, dessa forma, permitindo invocar métodos de objetos remotos, mover objetos entre computadores diferentes e outros.

CORBA é um padrão multilinguagem, mas existe outro padrão criado especificamente para a linguagem Java, chamado de Java RMI (*Java Remote Method Invocation*), onde são oferecidos recursos avançados para o desenvolvimento de aplicações com objetos distribuídos, de forma a disponibilizar classes e interfaces simples de usar, mas poderosas; deste modo, possibilitando que a utilização de tais recursos seja totalmente contida na linguagem Java, sem precisar de bibliotecas de terceiros e sem sobrecarregar os programas com sintaxe adicional, e facilitando em muito o desenvolvimento de aplicações com objetos distribuídos.

Geralmente em aplicações Java RMI, o servidor é usado para criar objetos remotos cujos métodos são invocados pelos clientes da mesma forma como se fossem objetos locais, simplificando bastante as aplicações que realizam comunicação entre objetos distribuídos. Dessa forma, o Java RMI esconde a complexidade inerente à transmissão de dados por *streams* pela internet.

Fique por dentro

Este artigo apresenta os principais recursos disponibilizados pelo Java RMI para realizar a invocação de métodos de objetos remotos, fazendo com que a comunicação entre aplicações Java localizadas em diferentes JVMs seja de simples implementação, mas muito eficiente, principalmente por que Java RMI cria uma abstração na qual o objeto remoto é acessado como se estivesse localizado na JVM local.

Através de sua forma natural e completamente integrada com a linguagem Java, o Java RMI se mostra como sendo um poderoso padrão que é utilizado no desenvolvimento de aplicações com objetos distribuídos, oferecendo uma abstração simples de comunicação entre processos, escondendo a complexidade do uso de sockets e outros mecanismos de baixo nível usados em aplicações em que objetos precisam receber ou enviar informações a outros objetos.

A fim de que os objetos remotos sejam encontrados, um registro de objetos remotos é utilizado, e ele possibilita vincular objetos a nomes no lado do servidor, operação chamada de *bind*. No lado do cliente, o registro é consultado para que se obtenham referências a objetos remotos.

O pacote java.rmi

O pacote `java.rmi` contém a implementação do Java RMI pertencente ao JDK e disponibiliza classes, interfaces e subpacotes para serem utilizados no desenvolvimento de aplicações com objetos distribuídos. Dentre estes recursos, os principais são:

- **Remote:** É uma interface que indica se um objeto possui métodos que podem ser invocados remotamente por outras JVMs;
- **MarshalledObject:** É uma classe, e o seu construtor recebe um objeto e o converte para um vetor de bytes (*marshalling*). Suporta também a reconstrução do objeto (*unmarshalling*);
- **Naming:** É uma classe que oferece métodos para armazenar e consultar referências a objetos remotos no registro de objetos remotos do Java RMI;
- **RMISecurityManager:** É uma classe utilizada por aplicações para verificar os requisitos de segurança para acessar classes descarregadas pela internet para serem executadas na máquina local;

- **AccessException:** É uma exceção que pode ser lançada por vários métodos da classe **Naming**, por exemplo, quando um servidor tenta se registrar no registro de objetos remotos e ocorre um erro;
- **AlreadyBoundException:** É uma exceção que ocorre quando se tenta registrar um nome no registro de objetos remotos, porém esse nome já se encontra registrado;
- **ConnectException:** É uma exceção que é lançada quando não é possível realizar uma conexão entre objetos remotos;
- **ConnectIOException:** É uma exceção que é lançada quando ocorre uma **IOException** na tentativa de conexão entre dois objetos remotos;
- **MarshallException:** É uma exceção que é lançada quando ocorre erro na operação de *marshalling*;
- **NoSuchObjectException:** É uma exceção que é lançada quando a aplicação tenta acessar um objeto remoto que não existe mais;
- **RemoteException:** É a superclasse comum a um grande número de exceções que podem ocorrer durante a invocação de um método remoto;
- **ServerException:** É uma exceção que é lançada no servidor se ocorrer erro no processamento de uma requisição de invocação de um método remoto por um cliente remoto;
- **UnmarshalException:** É uma exceção que é lançada quando ocorre erro na operação de *unmarshalling*;
- **java.rmi.activation:** É um pacote que oferece suporte à ativação de objetos remotos. Tais objetos são objetos que necessitam de acesso persistente e ininterrupto;
- **java.rmi.registry:** É um pacote que oferece suporte ao registro de objetos remotos do Java RMI. Suas principais classes são **LocateRegistry** e **Registry**;
- **java.rmi.server:** É um pacote que oferece classes e interfaces para a implementação do lado servidor do Java RMI. Suas principais classes são **UnicastRemoteObject**, **RemoteServer**, **RemoteObject**, entre outros.

A classe Naming

A classe **Naming** tem função primordial no controle do registro remoto RMI, sendo disponibilizada no pacote **java.rmi**. Ela é usada para armazenar e consultar referências a objetos remotos em um dado servidor. Seus principais métodos são:

- **static void bind(String name, Object obj):** Liga o objeto informado (**obj**) ao nome informado (**name**) no registro remoto RMI;
- **static String[] list(String name):** Retorna um vetor com os nomes encontrados no registro;
- **static Remote lookup(String name):** Retorna uma referência ao objeto remoto que está ligado ao nome informado como parâmetro;
- **static void rebind(String name, Object obj):** Liga o objeto informado (**obj**) ao nome informado (**name**) no registro remoto RMI, sobrescrevendo a ligação anterior que houver com o mesmo nome;
- **static void unbind(String name):** Elimina a ligação que existia do nome informado ao objeto que ele referenciava.

A classe LocateRegistry

É uma das principais classes presentes no pacote **java.rmi.registry**, pois ela é responsável por criar o registro RMI ou encontrá-lo em uma máquina virtual remota. Seus principais métodos são:

- **static Registry createRegistry(int port):** Cria um registro remoto RMI na máquina local usando a porta indicada;
- **static Registry getRegistry():** Retorna o registro remoto RMI presente na máquina local. Procura na porta padrão 1099;
- **static Registry getRegistry(int port):** Retorna o registro remoto RMI presente na máquina local na porta indicada;
- **static Registry getRegistry(String host):** Retorna o registro remoto RMI presente na máquina **host**. Procura na porta padrão 1099;
- **static Registry getRegistry(String host, int port):** Retorna o registro remoto RMI presente na máquina **host**. Procura na porta indicada.

A classe Registry

Situase no pacote **java.rmi.registry** e tem a função de manipular o registro remoto RMI, ligando ou desligando objetos remotos a nomes. Objetos da classe **Registry** podem ser obtidos quando realizadas consultas ao registro pela classe **LocateRegistry**. Os principais métodos de **Registry** são:

- **void bind(String nome, Remote object):** Liga o nome indicado (**name**) ao objeto remoto informado (**object**);
- **String[] list():** Retorna um vetor com todos os nomes presentes nesse registro;
- **Remote lookup(String name):** Consulta pelo objeto que esteja ligado ao nome indicado no registro;
- **void rebind(String nome, Remote object):** Sobrescreve a ligação do nome indicado, vinculando-o ao objeto remoto informado;
- **void unbind(String nome):** Elimina a ligação do nome indicado ao objeto que ele estava ligado no registro.

A interface Remote

A interface **Remote**, disponibilizada no pacote **java.rmi**, é usada para indicar objetos que contém métodos que são acessíveis remotamente por JVMs em clientes remotos. Para definir os métodos remotos é necessário que seja criada uma interface que contém a assinatura dos mesmos. Tal interface deve ser filha de **Remote**, garantindo assim, que o Java RMI considere essa interface como sendo uma interface de acesso remoto aos métodos do objeto remoto.

Portanto, para se criar um objeto no servidor cujos métodos sejam acessíveis remotamente, a classe desse objeto deve implementar uma interface filha de **Remote**, sendo que cada método dessa interface deve lançar a exceção **RemoteException**, pois podem ocorrer erros de comunicação no acesso remoto a tais métodos.

A interface **Remote** por si só não tem métodos, mas deve ser estendida diretamente ou indiretamente, caso contrário o objeto não poderá ser acessado remotamente. Adicionalmente, se um método de um objeto remoto retornar um objeto, este objeto deve estender a interface **java.io.Serializable**, pois de outra forma ocorrerá um erro na invocação deste método.

A classe UnicastRemoteObject

Para criar uma classe que seja acessível remotamente, pode-se fazer com que ela estenda **UnicastRemoteObject** (presente no pacote `java.rmi.server`), pois assim a classe remota ganha por herança as funcionalidades necessárias.

Adicionalmente, a classe remota deve oferecer uma interface de acesso aos clientes remotos, pela qual são determinadas as assinaturas dos métodos acessíveis remotamente. Com esse objetivo, a classe remota deve implementar uma interface filha de **Remote**, como comentado anteriormente; dessa forma, definindo precisamente as assinaturas dos métodos remotos que são utilizadas tanto no lado do cliente como no lado do servidor.

A seguir é apresentada uma aplicação exemplo que demonstra a utilização dos principais recursos do pacote `java.rmi`.

Aplicação exemplo – Lado do servidor

Para abordar o RMI na prática, o funcionamento das principais classes do Java RMI será apresentado no desenvolvimento de duas aplicações, uma com o papel de cliente e outra com o papel de servidor. A aplicação servidora tem a função de simular o controle dos eletrodomésticos de uma casa inteligente, disponibilizando funções de ligar ou desligar TV, som, ar condicionado e ainda de abrir ou fechar cortinas.

A aplicação cliente, por sua vez, tem a função de usar Java RMI para acessar tais recursos do servidor, possibilitando ao usuário da casa utilizar as funções de automação remotamente em qualquer plataforma que tenha uma implementação da JVM, através de acesso à internet ou estando na mesma rede local.

Primeiro, vamos à definição das funções da casa remota que são disponibilizadas no servidor. Isso é feito através da codificação da interface **InterfaceRemota**, apresentada na **Listagem 1**.

Listagem 1. Declaração da interface InterfaceRemota.

```
01. package pkg.rmi;
02.
03. import java.rmi.Remote;
04. import java.rmi.RemoteException;
05.
06. public interface InterfaceRemota extends Remote {
07.     public void ligarTV (int id) throws RemoteException;
08.     public void desligarTV (int id) throws RemoteException;
09.     public void ligarSom (int id) throws RemoteException;
10.     public void desligarSom (int id) throws RemoteException;
11.     public void abrirCortina (int id) throws RemoteException;
12.     public void fecharCortina (int id) throws RemoteException;
13.     public void ligarAr (int id) throws RemoteException;
14.     public void desligarAr (int id) throws RemoteException;
15.     public int getID() throws RemoteException;
16.     public void desconectarCliente(int idCliente) throws RemoteException;
17.     public ObjetoStatus getStatus() throws RemoteException;
18. }
```

Dessa forma, são definidos todos os métodos que são acessíveis remotamente, uma vez que a interface **InterfaceRemota** estende **Remote** e os seus métodos lançam **RemoteException**. Observando esse código, pode-se verificar que existem métodos para ligar

ou desligar TV, som e ar condicionado, além de abrir ou fechar cortinas.

Ainda analisando **InterfaceRemota**, o método **getID()** é responsável por retornar um identificador único ao cliente que recém se conectou no servidor (linha 15) e o método **desconectarCliente()** é responsável por sinalizar ao servidor que o cliente está sendo fechado (linha 16). Por fim, o método **getStatus()** é responsável por retornar o estado de cada eletrodoméstico da casa (incluindo as cortinas). Note que o estado é dado pela classe **ObjetoStatus**, apresentada na **Listagem 2**.

A classe **ObjetoStatus** tem a função de manter um registro do status dos eletrodomésticos da casa. Para isso, ela usa as variáveis **tvLigada**, **somLigado**, **cortinaAberta** e **arLigado** (linhas 7 a 10). Os **getters** e **setters** de tais atributos são definidos nas linhas 12 a 42. O método **toString()** próprio é definido de forma a retornar uma representação na forma de *string* desse objeto (linhas 44 a 68). Note que na criação da representação textual é utilizada a classe **StringBuilder** de forma a otimizar o desempenho e evitar a fragmentação da memória, causada nos casos em que a concatenação de strings é feita com o operador **+**.

Antes de apresentar a implementação dos métodos remotos, no entanto, é necessário apresentar a GUI do servidor, que tem a função de manter um log das operações recebidas, registrando a conexão e desconexão de clientes e o uso das funções de automação da casa inteligente, como apresentado na **Listagem 3**.

A GUI servidora é apresentada na **Figura 1** e mostra que um cliente se conectou, realizou a operação de ligar a TV e em seguida se desconectou.

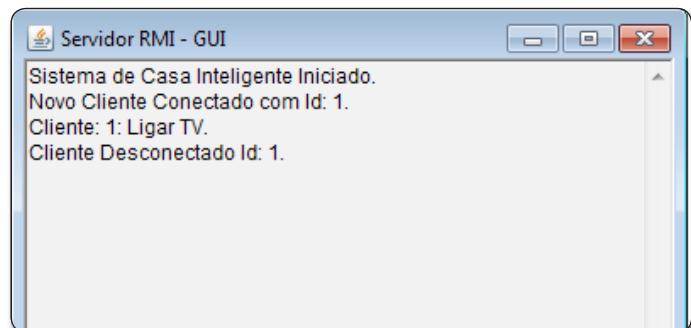


Figura 1. GUI no lado do servidor

Observe que a GUI servidora é uma interface gráfica que contém apenas um componente de texto, usado para o registro de mensagens. O método **imprimirMensagem()** recebe uma **String** e a escreve na tela (linhas 24 a 27). Neste método é importante notar o uso da palavra-chave **synchronized** (linha 25), de forma a definir um bloco sincronizado que é usado para evitar problemas de acesso concorrente ao componente de texto.

Agora, é apresentado o código do servidor na **Listagem 4**, onde a classe **ServidorRmi** (linha 8) estende **UnicastRemoteObject**, se tornando um objeto acessível remotamente, e implementa **InterfaceRemota**, assim, definindo os métodos que são remotos, e também suas implementações.

Listagem 2. Declaração da classe ObjetoStatus.

```
01. package pkg.rmi;
02.
03. import java.io.Serializable;
04.
05. public class ObjetoStatus implements Serializable {
06.
07.     private boolean tvLigada;
08.     private boolean somLigado;
09.     private boolean cortinaAberta;
10.     private boolean arLigado;
11.
12.     public boolean isTvLigada() {
13.         return tvLigada;
14.     }
15.
16.     public void setTvLigada(boolean tvLigada) {
17.         this.tvLigada = tvLigada;
18.     }
19.
20.     public boolean isSomLigado() {
21.         return somLigado;
22.     }
23.
24.     public void setSomLigado(boolean somLigado) {
25.         this.somLigado = somLigado;
26.     }
27.
28.     public boolean isCortinaAberta() {
29.         return cortinaAberta;
30.     }
31.
32.     public void setCortinaAberta(boolean cortinaAberta) {
33.         this.cortinaAberta = cortinaAberta;
34.     }
35.

36.     public boolean isArLigado() {
37.         return arLigado;
38.     }
39.
40.     public void setArLigado(boolean arLigado) {
41.         this.arLigado = arLigado;
42.     }
43.
44.     public String toString () {
45.         StringBuilder builder = new StringBuilder(200);
46.         builder.append("Status:TV\"");
47.         if (tvLigada)
48.             builder.append("Ligada");
49.         else
50.             builder.append("Desligada");
51.         builder.append("] Som\"");
52.         if (somLigado)
53.             builder.append("Ligado");
54.         else
55.             builder.append("Desligado");
56.         builder.append("] Cortina\"");
57.         if (cortinaAberta)
58.             builder.append("Aberta");
59.         else
60.             builder.append("Fechada");
61.         builder.append("] Ar\"");
62.         if (arLigado)
63.             builder.append("Ligado");
64.         else
65.             builder.append("Desligado");
66.         builder.append("]");
67.         return builder.toString();
68.     }
69. }
```

Listagem 3. Declaração da classe ServidorGui.

```
01. package pkg.rmi;
02.
03. import java.awt.Frame;
04. import java.awt.TextArea;
05. import java.awt.event.WindowAdapter;
06. import java.awt.event.WindowEvent;
07.
08. public class ServidorGui extends Frame {
09.
10.     private TextArea textArea = new TextArea();
11.
12.     public ServidorGui () {
13.         super("Servidor RMI - GUI");
14.         setSize(400, 500);
15.         add(textArea);
16.         textArea.setEditable(false);
17.         addWindowListener( new WindowAdapter() {
18.             public void windowClosing(WindowEvent we) {
19.                 System.exit(0);
20.             }
21.         });
22.     }
23.
24.     public void imprimirMensagem (String mensagem) {
25.         synchronized (this) {
26.             textArea.append(mensagem + "\r\n");
27.         }
28.     }
29. }
```

A GUI servidora é dada separadamente pela classe **ServidorGui**, e o servidor contém uma referência para acessá-la instanciada na linha 10. Também é mantido um objeto com os status dos eletrodomésticos (linha 11). A variável estática **id** (linha 12) registra o identificador do último cliente criado, e a cada vez que um cliente se conecta, o **id** é incrementado, como ocorre no método **getId()** (linhas 90 a 96), onde, através de um bloco sincronizado é gerado o novo **id**, que é escrito na tela e retornado para o cliente.

O método **main()** (linha 18) é o código executável que inicia a aplicação servidor. Primeiramente, ele cria um registro remoto RMI na porta 1099 (linha 21), em seguida, instancia o objeto remoto (linha 27) e o liga ao nome **RmiServer** (linha 28). Já nas linhas 30 e 31, é criada uma mensagem na GUI servidora informando que a aplicação foi iniciada e a mesma é exibida com o comando **setVisible()**.

O método **ligarTV()** é implementado nas linhas 34 a 39, originário da interface **InterfaceRemota**. Ele marca a TV como ligada (no objeto **status**, linha 36) e imprime uma mensagem na tela mostrando o **id** do cliente e a operação que foi realizada (linha 37). O mesmo padrão é repetido para os demais eletrodomésticos para ligar ou desligar até a linha 88.

No final da classe encontra-se o método **desconectarCliente()**. Este imprime na tela uma mensagem informando o **id** do cliente que foi desconectado (linhas 98 na 100). Por último, o método

Remote Method Invocation: RMI na prática

`getStatus()` tem a função de retornar para o cliente o status dos eletrodomésticos (linhas 102 a 104).

Com isso, foram apresentadas todas as classes localizadas no lado do servidor. A seguir serão apresentadas as classes que ficam localizadas no lado do cliente, que invoca os métodos remotos do objeto registrado com o nome **RmiServer**.

Aplicação exemplo – Lado do cliente

Apresentamos a aplicação servidora, porém ela não tem utilidade sem a existência da aplicação cliente, com a qual se comunica.

Neste exemplo a aplicação cliente é representada pela classe **ClienteRmi**, que contém um método `main()` com a função de obter uma referência ao objeto remoto **RmiServer** (linha 9) fazendo uma consulta ao registro remoto RMI no host *localhost* (máquina local) na porta padrão 1099. Observe que o servidor poderia estar localizado em outro computador. Portanto, para acessá-lo, basta informar o nome ou o endereço IP do computador remoto no local da palavra *localhost*.

Na **Listagem 5**, a GUI cliente é instanciada na linha 10 e como ela é um *thread Runnable* (como visto na **Listagem 6**), inicia

Listagem 4. Declaração da classe ServidorRmi.

```
01. package pkg.rmi;
02.
03. import java.rmi.Naming;
04. import java.rmi.RemoteException;
05. import java.rmi.server.UnicastRemoteObject;
06. import java.rmi.registry.*;
07.
08. public class ServidorRmi extends UnicastRemoteObject implements
   InterfaceRemota {
09.
10.    private static ServidorGui gui = new ServidorGui();
11.    private ObjetoStatus status = new ObjetoStatus();
12.    private static int id = 0;
13.
14.    public ServidorRmi() throws RemoteException {
15.        super(0);
16.    }
17.
18.    public static void main(String args[]) throws Exception {
19.        System.out.println("Servidor RMI iniciado.");
20.        try {
21.            LocateRegistry.createRegistry(1099);
22.            System.out.println("Registro Java RMI criado.");
23.        } catch (RemoteException e) {
24.            System.out.println("Registro Java RMI já existente.");
25.        }
26.
27.        ServidorRmi servidor = new ServidorRmi();
28.        Naming.rebind("//localhost/RmiServer", servidor);
29.        System.out.println("RmiServer Registrado.");
30.        gui.imprimirMensagem("Sistema de Casa Inteligente Iniciado.");
31.        gui.setVisible(true);
32.    }
33.
34.    public void ligarTV(int idCliente) throws RemoteException {
35.        synchronized (this) {
36.            status.setTvLigada(true);
37.            gui.imprimirMensagem("Cliente:" + idCliente + ": Ligar TV");
38.        }
39.    }
40.
41.    public void desligarTV(int idCliente) throws RemoteException {
42.        synchronized (this) {
43.            status.setTvLigada(false);
44.            gui.imprimirMensagem("Cliente:" + idCliente + ": Desligar TV");
45.        }
46.    }
47.
48.    public void ligarSom(int idCliente) throws RemoteException {
49.        synchronized (this) {
50.            status.setSomLigado(true);
51.            gui.imprimirMensagem("Cliente:" + idCliente + ": Ligar Som.");
52.        }
53.    }
54.
55.    public void desligarSom(int idCliente) throws RemoteException {
56.        synchronized (this) {
57.            status.setSomLigado(false);
58.            gui.imprimirMensagem("Cliente:" + idCliente + ": Desligar Som.");
59.        }
60.    }
61.
62.    public void abrirCortina(int idCliente) throws RemoteException {
63.        synchronized (this) {
64.            status.setCortinaAberta(true);
65.            gui.imprimirMensagem("Cliente:" + idCliente + ": Abrir Cortina.");
66.        }
67.    }
68.
69.    public void fecharCortina(int idCliente) throws RemoteException {
70.        synchronized (this) {
71.            status.setCortinaAberta(false);
72.            gui.imprimirMensagem("Cliente:" + idCliente + ": Fechar Cortina.");
73.        }
74.    }
75.
76.    public void ligarAr(int idCliente) throws RemoteException {
77.        synchronized (this) {
78.            status.setArLigado(true);
79.            gui.imprimirMensagem("Cliente:" + idCliente + ": Ligar Ar.");
80.        }
81.    }
82.
83.    public void desligarAr(int idCliente) throws RemoteException {
84.        synchronized (this) {
85.            status.setArLigado(false);
86.            gui.imprimirMensagem("Cliente:" + idCliente + ": Desligar Ar.");
87.        }
88.    }
89.
90.    public int getId(){
91.        synchronized (this) {
92.            id++;
93.            gui.imprimirMensagem("Novo Cliente Conectado com Id:" + id + "");
94.            return id;
95.        }
96.    }
97.
98.    public void desconectarCliente(int idCliente) throws RemoteException {
99.        gui.imprimirMensagem("Cliente Desconectado Id:" + idCliente + "");
100.    }
101.
102.    public ObjetoStatus getStatus() throws RemoteException {
103.        return status;
104.    }
105.}
```

seu processamento concorrente na chamada do método **start()** (linha 11), e passa a ser visível na linha 12 para iteração com o usuário.

Listagem 5. Declaração da classe ClienteRmi.

```

01. package pkg.rmi;
02.
03. import java.rmi.Naming;
04.
05. public class ClienteRmi {
06.
07.     public static void main(String args[]) throws Exception {
08.
09.         InterfaceRemota servidor = (InterfaceRemota)
10.             Naming.lookup("//localhost/RmiServer");
11.         ClienteGui gui = new ClienteGui(servidor);
12.         new Thread(gui).start();
13.         gui.setVisible(true);
14.     }

```

A última classe pertencente à aplicação é a GUI do cliente, apresentada na **Listagem 6**. Esta GUI tem a função de mostrar na sua parte superior o status dos eletrodomésticos e na sua parte inferior, os botões, para realizar as operações de ativar ou desativar os mesmos. A fim de simplificar a sua implementação, a classe **ClienteGui** foi definida como sendo um frame AWT, pois estende **Frame**, e além disso, é um *thread*, pois implementa **Runnable**. Dessa forma, ela pode ficar obtendo o status atual dos eletrodomésticos localizados no servidor para informar ao usuário (cliente), pois tal informação é exibida na GUI, como será mostrado mais adiante.

A interface gráfica provida pela classe **ClienteGui** é apresentada na **Figura 2**, sendo separada em duas partes: status e operações. Na área de status, localizada na parte superior, nota-se que a TV já se encontra ligada, pois o usuário solicitou anteriormente a operação de ligar a TV, clicando no botão de ação correspondente na parte inferior, que contém as operações disponíveis.



Figura 2. GUI no lado do cliente

A classe **ClienteGui** apresenta diversos atributos de instância, onde o primeiro, chamado de **idCliente** (declarado na linha 17 da **Listagem 6**), tem a função de manter a identificação desse cliente junto ao servidor. Este atributo é inicializado somente mais tarde (linha 61 da **Listagem 6**), quando o servidor for acessado, pois tal valor é definido somente uma vez, já que se trata de um atributo **final**, como requerido mais adiante por ser acessado dentro de classes anônimas internas (dadas pelos *listeners*, como exemplo, o *listener* para ligar a TV, na linha 89 da **Listagem 7**). Pelos mesmos motivos, a variável **servidor** também é declarada como **final** (linha 18 da **Listagem 6**). Por fim, nas linhas 19 e 20 da **Listagem 6** são declarados os *labels* para serem exibidos na GUI.

A classe **ClienteGui** apresenta muitos outros atributos referentes a cada eletrodoméstico, sendo nas linhas 22 a 27 declarados os componentes da interface gráfica referentes à TV: um *label*, um botão de rádio para mostrar o status do aparelho e dois botões (um para ligar a TV e outro para desligar). Analogamente, cada outro eletrodoméstico contém componentes equivalentes que são instanciados até a linha 48.

Depois da declaração de todos os campos da interface gráfica, o construtor da GUI cliente é declarado, recebendo como parâmetro um objeto remoto servidor (linha 50), que é mantido como variável de instância para que possa ser acessível aos *listeners* (linha 52). Em seguida, o método **getId()** do servidor é consultado para obter o **id** desse cliente (linhas 53 a 61). Logo após, são adicionados os *labels* que organizam a interface gráfica em status e operações (linhas 66 e 67), de modo que suas posições são definidas nas linhas 68 e 69. Observe nas linhas 70 e 71 o uso do método **setFont()** da classe **Label**. Este modifica a fonte de exibição para exibir as palavras com tamanho maior.

Feito isso, são inseridos e posicionados os componentes gráficos que representam cada eletrodoméstico. A **Listagem 7** apresenta o restante do construtor, que inicia pelos componentes da TV e define de forma análoga os outros eletrodomésticos.

Observe que nas linhas 73 a 77 os campos da GUI referentes à TV são adicionados ao *frame* para exibição. Os botões de rádio correspondentes ao status da TV são desabilitados para edição pelo usuário nas linhas 78 e 79, pois eles têm apenas a finalidade de exibição de status (sendo atribuído aos botões de ação a realização de operações sobre os eletrodomésticos). Depois disso, nas linhas 80 a 84, os componentes da interface gráfica da TV são ajustados na tela com o comando **setBounds()**, definindo a posição e o tamanho de cada um.

O *listener* do botão de ligar TV é definido nas linhas 85 a 95, sendo uma classe anônima interna, pois a declaração do seu código-fonte é contida dentro da declaração de outra classe (**ClienteGui**), sem definição explícita de nome. No processamento do evento do *listener* é usado um bloco sincronizado (linha 88) a fim de evitar problemas de concorrência no acesso ao objeto servidor e aos campos da GUI. Na linha 89, o servidor é chamado para ligar a TV, invocando o método remoto **ligarTV()**, e nas duas próximas linhas, a mudança de status é aplicada à interface gráfica através do método **setState()**.

Remote Method Invocation: RMI na prática

do botão de rádio. De forma semelhante, nas linhas 96 a 106 é criado o *listener* para o evento de clique no botão de ação para desligar a TV, que invoca outro método remoto, o **desligarTV()**.

Até agora foi apresentada a criação dos componentes da interface gráfica para a TV. Adicionalmente, os componentes

gráficos dos demais eletrodomésticos da casa inteligente são declarados até a linha 212 dessa listagem, porém as linhas de código estão omitidas para evitar a repetição de instruções semelhantes. O código fonte completo está disponível para download.

Listagem 6. Primeira parte da declaração da classe ClienteGui.

```
01. package pkg.rmi;
02.
03. import java.awt.Button;
04. // alguns imports omitidos...
05. import java.rmi.RemoteException;
06.
07. private final int idCliente;
08. private final InterfaceRemota servidor;
09. private Label status = new Label("Status da Casa:");
10. private Label operacoes = new Label("Operações Disponíveis:");
11.
12. private Label labelTV = new Label("TV:");
13. private CheckboxGroup tv = new CheckboxGroup();
14. private Checkbox tv1 = new Checkbox("Ligado", tv, true);
15. private Checkbox tv2 = new Checkbox("Desligado", tv, true);
16. private Button ligarTV = new Button ("Ligar TV");
17. private Button desligarTV = new Button ("Desligar TV");
18.
19. private Label labelSom = new Label("Som:");
20. private CheckboxGroup som = new CheckboxGroup();
21. private Checkbox som1 = new Checkbox("Ligado", som, true);
22. private Checkbox som2 = new Checkbox("Desligado", som, true);
23. private Button ligarSom = new Button ("Ligar Som");
24. private Button desligarSom = new Button ("Desligar Som");
25.
26. private Label labelCortina = new Label("Cortina.");
27. private CheckboxGroup cortina = new CheckboxGroup();
28. private Checkbox cortina1 = new Checkbox("Aberta", cortina, true);
29. private Checkbox cortina2 = new Checkbox("Fechada", cortina, true);
30. private Button ligarCortina = new Button ("Abrir Cortina");
31.
32.
33.
34.
35.
36.
37.
38.
39.
40.
```

```
41. private Button desligarCortina = new Button ("Fechar Cortina");
42.
43. private Label labelAr = new Label("Ar Condicionado:");
44. private CheckboxGroup ar = new CheckboxGroup();
45. private Checkbox ar1 = new Checkbox("Ligado", ar, true);
46. private Checkbox ar2 = new Checkbox("Desligado", ar, true);
47. private Button ligarAr = new Button ("Ligar Ar");
48. private Button desligarAr = new Button ("Desligar Ar");
49.
50. public ClienteGui (final InterfaceRemota servidor) {
51.
52.     this.servidor = servidor;
53.     int id = 0;
54.     try {
55.         id = servidor.getId();
56.     } catch (RemoteException e) {
57.         e.printStackTrace();
58.         System.out.println("Erro de Conexão");
59.         System.exit(0);
60.     }
61.     idCliente = id;
62.     setTitle("Cliente id = " + idCliente);
63.
64.     setLayout(null);
65.     setSize(400, 440);
66.     add(status);
67.     add(operacoes);
68.     status.setBounds(50, 60, 300, 25);
69.     operacoes.setBounds(50, 230, 300, 25);
70.     status.setFont(new Font("Times", 1, 20));
71.     operacoes.setFont(new Font("Times", 1, 20));
72.
```

Listagem 7. Segunda parte da declaração da classe ClienteGui.

```
73.     add(labelTV);
74.     add(tv);
75.     add(tv2);
76.     add(ligarTV);
77.     add(desligarTV);
78.     tv1.setEnabled(false);
79.     tv2.setEnabled(false);
80.     labelTV .setBounds(50, 110, 100, 25);
81.     tv1 .setBounds(170, 110, 80, 25);
82.     tv2 .setBounds(260, 110, 100, 25);
83.     ligarTV .setBounds(70, 270, 100, 25);
84.     desligarTV.setBounds(200, 270, 100, 25);
85.     ligarTV.addActionListener(new ActionListener() {
86.         public void actionPerformed(ActionEvent evt) {
87.             try {
88.                 synchronized (this) {
89.                     servidor.ligarTV(idCliente);
90.                     tv1.setState(true);
91.                     tv2.setState(false);
92.                 }
93.             } catch (RemoteException e) {
94.                 e.printStackTrace();
95.             }});
96.     desligarTV.addActionListener(new ActionListener() {
```

```
97.         public void actionPerformed(ActionEvent evt) {
98.             try {
99.                 synchronized (this) {
100.                     servidor.desligarTV(idCliente);
101.                     tv1.setState(false);
102.                     tv2.setState(true);
103.                 }
104.             } catch (RemoteException e) {
105.                 e.printStackTrace();
106.             }});
107.
108. ...
109.         addWindowListener(new WindowAdapter() {
110.             public void windowClosing(WindowEvent we) {
111.                 try {
112.                     servidor.desconectarCliente(idCliente);
113.                 } catch (RemoteException e) {
114.                     System.out.println("Erro ao desconectar cliente: " + e.getMessage());
115.                 }
116.                 System.exit(0);
117.             }
118.         });
119.     });
120. }}
```

Após todo esse código, é declarado o *listener* para fechamento da GUI (linhas 214 a 222), que avisa o servidor que o cliente está sendo fechado invocando o método remoto **desconectarCliente()**. Na **Listagem 8** é apresentada a última parte da classe **ClienteGui**, o código a ser executado na forma de *thread*.

Listagem 8. Última parte da declaração da classe ClienteGui.

```

225. public void run() {
226.     while (true) {
227.         try {
228.             ObjetoStatus status = servidor.getStatus();
229.             tv1.setState(status.isTvLigada());
230.             tv2.setState(!status.isTvLigada());
231.             som1.setState(status.isSomLigado());
232.             som2.setState(!status.isSomLigado());
233.             cortina1.setState(status.isCortinaAberta());
234.             cortina2.setState(!status.isCortinaAberta());
235.             ar1.setState(status.isArLigado());
236.             ar2.setState(!status.isArLigado());
237.         } catch (RemoteException e) {
238.             System.out.println("Erro ao acessar status: " + e.getMessage());
239.         }
240.         synchronized (this) {
241.             try {
242.                 wait(1000);
243.             } catch (InterruptedException e) {
244.                 e.printStackTrace();
245.             }
246.         }
247.     }
248. }
249.

```

Veja que é declarado o método **run()** e como a classe **ClienteGui** implementa a interface **Runnable**, ela pode ser executada como *thread* (linha 11 da **Listagem 5**). Nesse caso, durante a execução de **run()**, o processamento fica preso em um laço infinito (linha 226), e a cada iteração o método remoto **getStatus()** é invocado para obter o status dos eletrodomésticos da casa.

Uma vez que o servidor tenha retornado o status dos eletrodomésticos, a interface gráfica é atualizada para refletir o status obtido (linhas 229 a 236), marcando cada botão de rádio de acordo com o status obtido de cada eletrodoméstico. Por fim, na linha 242, é chamado o método **wait()** para fazer o *thread* perder a CPU durante um segundo. Dessa forma, a cada iteração, ele busca o status, atualiza os botões de rádio e espera um segundo antes de iniciar a nova iteração.

Stubs e Skeletons

Em versões mais antigas do Java era necessário usar o compilador **rmic** para gerar as classes *stubs* e *skeletons*, as quais são classes implícitas que têm a função de realizar a comunicação entre servidor e cliente, de forma a implementar a interface **Remote** dos objetos remotos, adicionando as funcionalidades de comunicação.

Por definição, o *stub* é localizado no lado do cliente e o *skeleton* é localizado no lado do servidor. Felizmente, após a versão 5 do Java, não é mais necessário usar o comando **rmic** para gerar os *stubs*, pois tal tarefa é feita automaticamente pela JDK no processo de compilação do código fonte.

Porém, implicitamente, ainda são utilizados *stubs* e *skeletons*, e quando um cliente faz uma invocação de um método remoto, internamente, o *stub* é chamado, iniciando uma sequência de passos onde, primeiramente, cria-se uma conexão com a JVM remota que contém o objeto chamado. Em seguida, os dados da requisição são transformados em um vetor de bytes (*marshalling*) e transmitidos à JVM remota e o *stub* aguarda até o processamento do método terminar, que é quando ele recebe o retorno e realiza a operação de *unmarshalling*, e finalmente devolve o retorno do método ao cliente.

A maioria das aplicações utilizadas na atualidade contém algum tipo de comunicação em rede ou pela internet, demandando recursos de comunicação entre processos. Para solucionar este requisito, existem diversos métodos de comunicação que podem ser utilizados, como por exemplo, o protocolo HTTP para acesso a *websites* ou *sockets* para transmissão física de dados entre processos de plataformas diferentes.

Comparado com outros modelos de comunicação entre processos de plataformas diferentes, o modelo de comunicação baseado em objetos distribuídos é mais refinado, pois as clássicas diretivas de *sockets* para enviar e receber mensagens são substituídas pela invocação de métodos de objetos remotos. Sendo assim, explore a fundo os recursos do RMI, para que quando encontre requisitos nos quais ele possa ser empregado você já tenha domínio sobre o assunto e saiba empregá-lo de forma diferenciada.

Autor



John Soldera

johnsoldera@gmail.com

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



Links:

Javadoc da plataforma Java SE 7.

<http://docs.oracle.com/javase/7/docs/api/>

Site do projeto CORBA.

<http://www.corba.org/>

Documentação com informações sobre Java RMI.

<http://docs.oracle.com/javase/7/docs/api/java/rmi/package-summary.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