easy java Magazine

Edição 44

DEVMEDIA

ISSN 2179625-4

DESAFIO: Configurando um cluster com Tomcat e NGinx
Aprenda a criar um cluster com balanceamento de carga

Desmistificando quatro Design Patterns
Utilizando o Singleton, Abstract Factory, Decorator e Strategy

POO: reusabilidade e eficiência em seu código
Código ágil e otimizado com os poderosos recursos do Java

DATAS NO JAVA 8

Aprenda como manipular a Date and Time API

DESENVOLVA PARA CORPORAÇÕES OU PARA O DIA A DIA COM JAVA

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!

FORMAÇÃO DESENVOLVEDOR JAVA

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

CERTIFICAÇÕES:

- Oracle Certified Professional, Java SE 6 Programmer
- Oracle Certified Professional, Java EE 5 Web Component Developer

infnet TRAINING

Central de Matrículas: 21 2122-8800 | www.infnet.edu.br

Sumário

Edição 44 • 2014 • ISSN 2173625-4

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 a vontade para entrar em contato com os editores e dar a sua sugestão!

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

EDUARDO OLIVEIRA SPÍNOLA

eduspinola.wordpress.com

@eduspinola / @Java_Magazine

summa
Technology + Business
Premium Partner

CRIANDO APLICAÇÕES INOVADORAS QUE VÃO MUDAR O MUNDO?

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

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

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

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

Na suite ToolsCloud você tem a sua disposição:

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

Acompanhe a Toolscloud:

toolscloud@toolscloud.com

twitter.com/toolscloud

ToolsCloud

toolscloud.com

Design Patterns: Aprenda a utilizar Singleton, Abstract Factory, Decorator e Strategy

Fique por dentro

Um dos principais problemas na adoção de padrões de projeto orientados a objetos é justamente decidir em que problemas eles devem ser aplicados. Neste artigo exploramos os fundamentos de padrões de projeto, explicando seus conceitos essenciais e abordando a utilização de quatro padrões do catálogo GoF para satisfazer alguns requisitos básicos em uma aplicação exemplo.

Ao realizarmos uma busca, encontraremos diversas definições para a palavra "padrão". Essas definições, em geral, partem de um conceito central que é a busca por regularidade, repetição ou possibilidade de reprodução, mesmo que a repetição não seja exata ou perfeita. Com esses pequenos conceitos em mente, podemos imaginar que um padrão é algo que fornece um gabarito para a realização de determinada ação ou tarefa. É um ponto de partida reproduzível, mas que pode ser estendido de acordo com o problema em que queremos aplicá-lo.

Na Engenharia de Software, o termo "Padrão de Projeto" (ou *Design Pattern*, em inglês) foi utilizado pela primeira vez por Kent Beck e Ward Cunningham, que se propuseram a experimentar as ideias sobre uma linguagem de padrões para projetos de construção civil que Christopher Alexander havia implementado na década anterior. Alexander imaginava que os usuários sabiam mais sobre os prédios em que desejavam viver do que os arquitetos que os projetavam e, a partir dessa ideia central, desenvolveu padrões que possibilitariam a qualquer profissional projetar e construir prédios que se adequassem às necessidades mais comuns das pessoas.

Beck e Cunningham (que mais tarde ganhariam notoriedade pela sua participação na criação dos conceitos de Extreme Programming e do *Agile*) deram início à formalização do conceito de padrões de projeto, mas o termo ficou realmente popular com a publicação em 1994 do livro "Design Patterns: Elements of Reusable Object-Oriented Software" (sem tradução no Brasil, embora edições posteriores possam ser encontradas em Português), de autoria da chamada "Gang-of-Four" (GoF), formada inicialmente por Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides (falecido em 2005).

Esse livro, que se tornou um clássico no ensino de programação em todo o mundo, dedica seus dois primeiros capítulos à exploração do potencial da programação orientada a objetos enquanto paradigma da computação. Na sequência, os autores descrevem os 23 padrões de projeto que viriam a compor o não menos clássico catálogo GoF.

A partir desse momento, com o estabelecimento de um catálogo consistente em uma publicação importante para a Ciência da Computação, os padrões de projeto passam a ser um conceito plenamente estabelecido na Engenharia de Software para designar soluções reusáveis para problemas recorrentes de projeto de software em um dado contexto, constituindo um conjunto de boas práticas de desenvolvimento orientado a objetos.

Com isso, diversos catálogos foram elaborados e diversos tipos de padrões foram desenvolvidos (vide **BOX 1**). Classificações apareceram para facilitar a organização dos padrões em seus respectivos catálogos e, claro, novas publicações surgiram a respeito do tema.

Uma vez que os exemplos demonstrados no livro da Gang-of-Four foram desenvolvidos utilizando a linguagem C++, não demorou até que as comunidades dedicadas a outras linguagens fizessem suas versões dos padrões do catálogo. Diversos livros exploram a aplicação dos padrões do catálogo GoF em Java, dentre os quais gosto de destacar "*Java Design Patterns*", de James Cooper – por seu caráter seminal em explorar a aplicação do catálogo GoF em Java e "*Core J2EE Design Patterns: Best Practices and Design*

Strategies" (*Core J2EE Design Patterns*: As Melhores Práticas e Estratégias de Design), de Dan Malks, Deepak Alur e John Crupi – por sua abordagem pedagógica em relação ao tema.

BOX 1. Catálogos de Padrões

Catálogos são repositórios de padrões e, em geral, são mantidos por seus autores ou por grupos dos quais os autores dos padrões fazem parte. Em geral, os catálogos são relacionados a domínios de aplicação – ou contextos. Podemos destacar os populares catálogos Gang-of Four (GoF) e o General Responsibility Assignment Software Patterns (GRASP).

Em sua documentação, um padrão de projeto precisa explicar o motivo pelo qual as principais situações em que ele se aplica são problemáticas e, especialmente, por que a solução proposta pode ser considerada adequada. Para o pessoal da Gang-of-Four, os problemas mais comuns em projetos de software surgem de conflitos, ou seja, de situações em que o objetivo vai de encontro com a ferramenta utilizada para alcançá-lo (por exemplo, curar um paciente vs. o risco de determinados remédios poderem matá-lo). Esses fatores devem fazer parte da documentação do padrão e, em determinados casos, os autores explicam as suas principais aplicações.

Da mesma forma, se um padrão é utilizado recorrentemente de maneira não efetiva, ele se torna um Anti-padrão (vide **BOX 2**). Sendo assim, podemos assumir que o principal objetivo dos padrões é contribuir para a solução de problemas recorrentes de programação. No entanto, é preciso destacar que um padrão não é a solução em si, mas um "caminho" comum no qual a solução pode ser encontrada. O lado bom dessa história é que, um padrão catalogado significa que ele já foi usado e testado, o que fornece certa segurança em sua aplicação.

BOX 2. Anti-padrões (Anti-patterns)

Por diversas vezes, principalmente quando um determinado padrão passa a ser utilizado sem a devida avaliação, seu uso se torna contra-produtivo, tornando-o o que, no jargão do desenvolvimento, se chama "Anti-padrão". Em geral, seu uso traz mais problemas do que benefícios, havendo alternativas disponíveis e documentadas para solucionar o problema em que ele vem sendo aplicado. O primeiro passo para evitar Anti-padrões é reconhecê-los.

Ainda assim, cabe ao desenvolvedor descobrir qual caminho seguir, ou seja, que padrão pode ser utilizado na solução de seu problema. Isso não é uma decisão trivial. Mesmo programadores experientes se deparam com dificuldades ao fazer a associação direta de um problema com determinado padrão de projeto, embora alguns problemas sejam tão clássicos que já estejam associados a determinados padrões.

É nesse sentido que esse artigo foi elaborado, na tentativa de contribuir com uma demonstração do uso de alguns padrões de projeto sob um ponto de vista prático, aplicado em problemas que, embora comuns, não sejam os mais convencionais em que esses padrões são normalmente utilizados. Aqui, vamos buscar alternativas para o uso de quatro padrões do catálogo GoF e demonstraremos em exemplos como os padrões foram aplicados e os resultados obtidos.

O Catálogo GoF

O catálogo GoF possui atualmente 23 padrões, classificados em três tipos, como segue:

- **Padrões criacionais:** esse tipo de padrão abstrai o processo de instanciação, separando a criação, composição e representação dos objetos. Padrões criacionais fazem uso de herança para derivar classes que são instanciadas e delegam sua instanciação a outros objetos;
- **Padrões estruturais:** têm como foco a composição entre objetos. Nesses padrões, a herança é usada para compor interfaces e definir como objetos se relacionam para obter novas funcionalidades;
- **Padrões comportamentais:** são os padrões criados para descrever a maneira como os objetos se relacionam. Tratam das interações entre os objetos e da divisão de responsabilidades por entre as classes. Nesses padrões, as ações que se deseja obter de um objeto são abstraídas e, mudando o objeto ou a classe, o algoritmo usado para implementar um comportamento também pode ser modificado.

A **Figura 1** demonstra como os padrões GoF estão organizados.

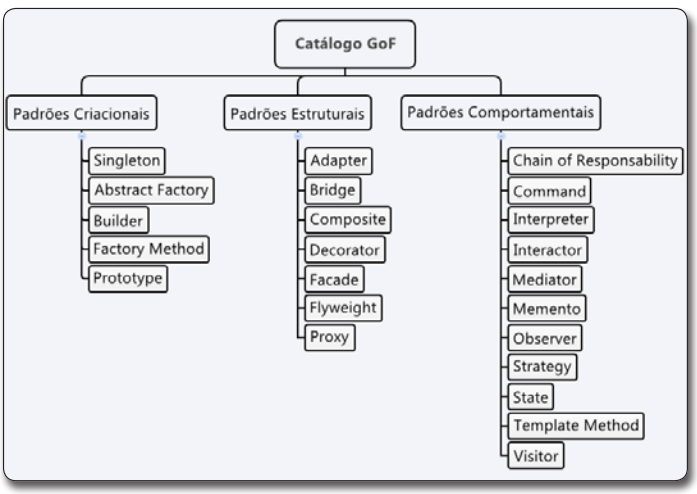

Figura 1. Classificação dos padrões no catálogo GoF

Nesse artigo escolhemos explorar quatro padrões, sendo dois padrões criacionais, um padrão estrutural e um padrão comportamental. Escolhemos os padrões sem nenhum critério especial, exceto pelo fato de serem os padrões mais abrangentes e, portanto, que acabaram se tornando muito populares entre os desenvolvedores. Os padrões criacionais escolhidos foram *Singleton* e *Abstract Factory*. O padrão estrutural escolhido foi o *Decorator*, enquanto o padrão comportamental escolhido foi o *Strategy*.

Os padrões *Singleton* e *Abstract Factory* são muito relacionados a bancos de dados. Enquanto o primeiro é fartamente utilizado no desenvolvimento de aplicações para o gerenciamento de *pools* de conexões, o segundo tem uma forte relação com o padrão

arquitetural Data Access Objects (DAO), amplamente adotado para separar as regras de negócio das regras de acesso na persistência em bancos de dados relacionais.

Por outro lado, o padrão *Decorator* se tornou muito popular a partir do uso de Anotações (vide **BOX 3**), não só em Java, mas em outras arquiteturas que possuem recursos semelhantes, enquanto o padrão *Strategy* é bastante utilizado para criar famílias de algoritmos.

BOX 3. Anotações (Annotations)

Anotações são recursos para a declaração de metadados – dados que descrevem outros dados – úteis para localizar dependências, configurações ou para fazer verificações lógicas. Essas definições serão, então, interpretadas pelo compilador para realizar uma determinada tarefa.

Implementando um exemplo do uso de padrões

Nas próximas subseções, os padrões serão explorados e exemplos de seu uso serão demonstrados. Vamos começar descrevendo um cenário para, em seguida, implementar soluções para as regras de negócio estabelecidas, usando os padrões que escolhemos para demonstrar nesse artigo.

Neste cenário, uma aplicação Java vai ser desenvolvida para dar suporte ao trabalho de pessoas na secretaria acadêmica de uma escola ou universidade. Não vamos demonstrar no exemplo apresentado nesse artigo o desenvolvimento da aplicação inteira. Vamos focar em algumas regras de negócio específicas e nos problemas existentes nos requisitos que devem ser implementados para atendê-las, como segue:

- **Perfis de usuários:** existem três perfis de usuários (Administrador, Estudante e Professor). O Administrador é o topo da hierarquia de usuários, tendo abaixo de si Estudante e Professor, ambos no mesmo nível, embora com funcionalidades específicas. Além disso, ambos podem receber, dependendo da situação, novas atribuições;
- **Login:** todos os usuários do sistema precisam se logar, mas somente um usuário com o perfil administrador pode estar logado por vez.

É natural que o desenvolvedor, ao se deparar com essas regras, pense instantaneamente em algoritmos capazes de satisfazê-las, no entanto, por se tratarem de problemas recorrentes, alguns padrões já apresentam formas otimizadas de implementá-las. Nesses casos, utilizar padrões pode acelerar o desenvolvimento, reduzindo o esforço de programação e, consequentemente, o custo de projeto, como veremos a seguir.

Padrões criacionais: Singleton e Abstract Factory

O padrão Singleton foi criado basicamente para que se pudesse controlar a quantidade de instâncias que uma determinada classe disponibilizaria. Mais do que isso, como seu próprio nome diz (single, que significa "único"), esse padrão deve permitir que apenas uma instância de uma determinada classe esteja ativa por vez.

Adaptações desse padrão foram utilizadas nos mais diversos tipos de aplicação, dentre os quais vale destacar o seu uso para gerenciar filas e *pools* de conexões com bancos de dados. Embora seja de simples implementação e fartamente utilizado – e talvez justamente por isso – esse padrão é constantemente referido como um Anti-padrão.

Sendo assim, seu uso é, por diversas vezes, visto com ressalvas, principalmente por ser usado para inserir restrições desnecessárias, controlando muitas vezes a instanciação de classes em contextos ou momentos desnecessários.

Ainda assim, seu uso é bastante difundido, especialmente combinado com outros padrões, como em conjunto com o próprio *Abstract Factory* – que será demonstrado nesse artigo – ou em *Facades* – que em geral são *Singletons*, uma vez que comumente apenas uma instância de cada *Facade* é desejada.

Em nosso estudo, vamos adotar o padrão Singleton de uma maneira um pouco diferente, embora mais simples: vamos utilizá-lo para satisfazer o requisito de que apenas um usuário do tipo Administrador pode estar logado por vez.

Sua implementação parte de um raciocínio bastante simples, que é senão o único, a principal forma de evitar que uma classe seja instanciada em Java: restringir o acesso ao construtor. Para fazer isso, basta que o modificador de acesso do construtor seja privado, como mostra a **Listagem 1**, que apresenta a primeira parte do código da classe **Administrador**.

Listagem 1. Código-fonte de Administrador.

```
package br.com.devmedia.gestaoacademica.model;

public class Administrador {

 private static Administrador instancia = null;

 private Administrador() {}

}
```

Repare que, enquanto o construtor de **Administrador** é **private**, temos um objeto chamado **instancia** sendo declarado. Esse objeto, do tipo **Administrador**, é justamente a única instância que será retornada toda vez que um objeto desse tipo for necessário. A razão pela qual esse objeto é estático é para que ele possa ser obtido sem a necessidade de uma outra instância de **Administrador** para disponibilizá-lo.

No entanto, o problema ainda não está resolvido. Na verdade, da maneira como está implementado até o momento, não há como conseguir nenhuma instância de **Administrador**. Isso será resolvido por meio da criação de um método – público e estático, chamado **getInstancia()** – que será responsável por retornar o objeto chamado **instancia** visto anteriormente na **Listagem 1**. Sendo assim, a **Listagem 2** mostra o código completo da classe **Administrador**.

O código de **getInstancia()** faz um teste simples, em que só atribui um novo valor a **instancia** caso seu valor corrente seja

nulo, ou seja, uma vez que o objeto exista, ele próprio – e somente ele – será retornado.

Com isso, um dos problemas – garantir que apenas um **Administrador** esteja ativo no sistema – está resolvido. Podemos partir agora para os problemas seguintes, onde aplicaremos outros padrões.

Listagem 2. Código-fonte de Administrador.

```
package br.com.devmedia.gestaoacademica.model;

public class Administrador {

private static Administrador instancia = null;

private Administrador() {}

public static Administrador getInstancia(){
      if (instancia == null) {
         instancia = new Administrador();
      }
      return instancia;
   }

}
```

Fábricas (*Factories*) de objetos se tornaram muito populares em Java por simplificarem a implementação de hierarquias de objetos, garantindo um bom encapsulamento de funcionalidades, tornando possível criar objetos sem expor a lógica implementada para sua criação, normalmente a partir do uso de uma interface comum. Como mostra a **Figura 1**, o catálogo GoF possui duas factories: *Abstract Factory* e *Factory Method* (que por sua vez, pode ser simplificado, se tornando o padrão que muitos chamam de *Factory Pattern* ou simplesmente *Factory*).

Como o próprio nome diz, as factories foram criadas para tornar possível criar famílias de objetos, muito úteis quando é necessário ter flexibilidade em adicionar novos tipos de objetos a uma determinada hierarquia, como para criar linhas de produtos, por exemplo.

Sendo assim, esse padrão foi muito utilizado para compor arquiteturas para a camada de persistência, em que é necessário ter escalabilidade suficiente para persistir diversos tipos de objetos sem revelar seu comportamento. O padrão arquitetural DAO se beneficiou bastante dessa abordagem.

O padrão Factory Method se baseia na criação de uma classe abstrata e um conjunto de subclasses concretas que são responsáveis por criar os objetos da família. Já em e uma Abstract Factory, sua implementação consiste em uma interface responsável pela criação de uma fábrica de objetos relacionados, sem especificar explicitamente suas classes. Sendo assim, ela se difere de sua "irmã" Factory Method por ser uma "fábrica de fábricas" e também por, nesse caso, ser dado a uma camada "cliente" a responsabilidade de gerenciar a criação dos objetos.

No nosso exemplo, vamos utilizar o padrão *Abstract Factory* para implementar a hierarquia de usuários da aplicação, que deve ser escalável o suficiente para receber novos tipos de usuários na medida em que o software evoluir.

A **Figura 2** ilustra o *Abstract Factory*, já exibindo as classes utilizadas no desenvolvimento de nossa aplicação de exemplo.

O primeiro passo é a criação da interface **Usuario**, como mostra a **Listagem 3**. Como podemos ver, essa interface define o método **efetuarLogin()**.

Em seguida, vamos fazer com que as classes concretas, que são representações dos usuários do sistema (Administrador, Estudante e Professor), implementem essa interface, como mostram as **Listagens 4**, **5** e **6**.

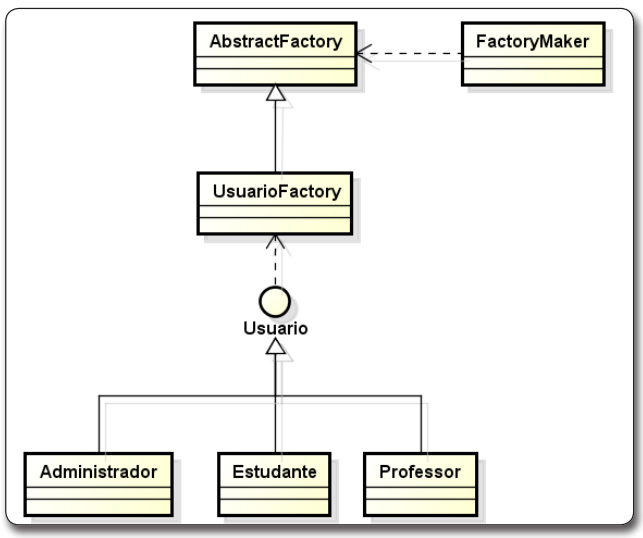

Figura 2. Padrão Abstract Factory para a aplicação de exemplo

Listagem 3. Código-fonte da interface Usuario.

```
package br.com.devmedia.gestaoacademica.model;

public interface Usuario {

   public void efetuarLogin();
}
```

Listagem 4. Código-fonte de Administrador.

```
package br.com.devmedia.gestaoacademica.model;

public class Administrador implements Usuario{

   private static Administrador instancia = null;

   private Administrador() {}

   public static Administrador getInstancia(){
      if (instancia == null) {
         instancia = new Administrador();
      }
      System.out.println(":: INSTÂNCIA ÚNICA DE ADMINISTRADOR OBTIDA ::");
      return instancia;
   }

   public void efetuarLogin(){
      System.out.println(":: LOGIN DE ADMINISTRADOR ::");
   }

}
```

Listagem 5. Código-fonte de Estudante.

```
package br.com.devmedia.gestaoacademica.model;

public class Estudante implements Usuario {

   public void efetuarLogin() {
      System.out.println(":: LOGIN DE ESTUDANTE ::");
   }

}
```

Listagem 6. Código-fonte de Professor.

```
package br.com.devmedia.gestaoacademica.model;

public class Professor implements Usuario {

   public void efetuarLogin() {
      System.out.println(":: LOGIN DE PROFESSOR ::");
   }

}
```

Listagem 7. Código-fonte de AbstractFactory.

```
package br.com.devmedia.gestaoacademica.model;

public abstract class AbstractFactory {

   abstract Usuario getUsuario(String usuario);

}
```

Listagem 8. Código-fonte de UsuarioFactory.

```
package br.com.devmedia.gestaoacademica.model;

public class UsuarioFactory extends AbstractFactory {

   public Usuario getUsuario(String usuario){
      if(usuario.equalsIgnoreCase("ADMINISTRADOR")){
         System.out.println(":: FABRICADO OBJETO DO TIPO ADMINISTRADOR ::");
         return Administrador.getInstancia();
      }else if(usuario.equalsIgnoreCase("ESTUDANTE")){
         System.out.println(":: FABRICADO OBJETO DO TIPO ESTUDANTE ::");
         return new Estudante();
      }else if(usuario.equalsIgnoreCase("PROFESSOR")){
         System.out.println(":: FABRICADO OBJETO DO TIPO PROFESSOR ::");
         return new Professor();
      }
      return null;
   }

}
```

Agora, vamos criar a classe abstrata – a fábrica propriamente dita – que define a forma pela qual os objetos são instanciados. Como mostra a **Listagem 7**, essa classe possui métodos abstratos que, quando implementados pelas subclasses determinarão como serão criadas instâncias dos objetos de sua hierarquia.

Feito isso, agora criaremos a classe concreta que fabrica os objetos que implementam a interface **Usuario** – **UsuarioFactory**, como mostra a **Listagem 8**.

Repare que ao retornar uma instância de **Administrador**, precisamos ter o cuidado de utilizar seu método **getInstancia()**, uma vez que se trata de um *Singleton*.

Usando um procedimento semelhante vamos criar a "fábrica de fábricas", embora no contexto em que se insere o exemplo demonstrado nesse artigo só tenhamos um tipo de objeto no nível mais alto da hierarquia – **Usuario**. Ainda assim, é importante desenvolvê-la para que o exemplo fique completo. Veja seu código na **Listagem 9**.

Listagem 9. Código-fonte de FactoryMaker.

```
package br.com.devmedia.gestaoacademica.model;

public class FactoryMaker {

   public static AbstractFactory getFactory(String fabrica){
      if(fabrica.equalsIgnoreCase("USUARIO")){
         return new UsuarioFactory();
      }
      return null;
   }

}
```

Isso resolve o segundo problema, que é garantir que os usuários do sistema estejam dispostos numa hierarquia que os torne independentes. Dessa forma, quaisquer características que especializem os objetos dessas classes podem ser implementadas de maneira particular, não interferindo na forma como estes se relacionam e são instanciados. Por exemplo, poderíamos fazer com que o método **efetuarLogin()** carregasse os privilégios para cada tipo de usuário de maneira diferente, e, da mesma forma, criar novos tipos de usuários com a mesma funcionalidade sem afetar o projeto.

Padrões estruturais: Decorator

Estender a funcionalidade de um objeto pode ser feito em tempo de compilação utilizando, por exemplo, herança. No entanto, em alguns casos, pode ser necessário estender as funcionalidades de um objeto de forma dinâmica, em tempo de execução, dependendo do contexto em que o objeto esteja sendo utilizado. O mesmo vale para remover funcionalidades de um objeto em determinado contexto.

Esse tipo de característica é muito explorado no projeto de interfaces gráficas, por exemplo, para tornar possível que determinadas janelas tenham funcionalidades que outras não possuem ou para que, de acordo com a posição em que ela esteja, funcionalidades sejam removidas ou adicionadas.

O padrão Decorator permite que o comportamento de um determinado objeto seja modificado tanto em tempo de compilação (de forma estática) quanto em tempo de execução (de forma dinâmica) sem que o comportamento de outros objetos da mesma classe seja afetado. Isso é feito por meio da implementação de uma classe Decorator que envolva a classe cujo comportamento precisa ser modificado. A **Figura 3** descreve a implementação do Decorator.

No nosso caso, partiremos da interface **Usuario** e de suas implementações, já criadas anteriormente, para implementar a

Abstract Factory. Sendo assim, basta criarmos o *Decorator* abstrato implementando a interface **Usuario**, que é efetivamente de onde derivam os objetos a serem "decorados". O código-fonte da classe **UsuarioDecorator** pode ser visto na **Listagem 10**.

Em seguida, vamos implementar o *Decorator* concreto, ou seja, a classe em que a modificação de comportamento do objeto vai ser efetivamente implementada. Essa modificação de comportamento é a famigerada "decoração" que o objeto pode receber dinâmica ou estaticamente, como pode ser visto na **Listagem 11**. No nosso caso, a decoração que criamos está relacionada com tornar um usuário um "Super Usuário".

Agora, basta utilizar o Decorator para decorar os objetos que desejarmos. Aplicado ao nosso exemplo, isso pode ser utilizado para satisfazer ao requisito que impõe que, dependendo do contexto, determinados usuários podem ter acesso a privilégios de administração do sistema.

Alguns padrões estruturais se confundem com padrões comportamentais na medida em que interferem na forma como objetos se comportam – e o Decorator é um caso desse tipo. Na seção seguinte, trataremos dessa distinção e demonstraremos o uso do padrão *Strategy*, um importante padrão comportamental.

Listagem 10. Código-fonte de UsuarioDecorator.

```
package br.com.devmedia.gestaoacademica.model;

public abstract class UsuarioDecorator implements Usuario{

   protected Usuario usuarioDecorado;

   public UsuarioDecorator(Usuario usuarioDecorado){
      this.usuarioDecorado = usuarioDecorado;
   }

   public void efetuarLogin(){
      usuarioDecorado.efetuarLogin();
   }

}
```

Listagem 11. Código-fonte de SuperUsuarioDecorator.

```
package br.com.devmedia.gestaoacademica.model;

public class SuperUsuarioDecorator extends UsuarioDecorator {

   public SuperUsuarioDecorator (Usuario usuarioDecorado){
      super(usuarioDecorado);
   }

   public void efetuarLogin(){
      usuarioDecorado.efetuarLogin();
      tornarSuper(usuarioDecorado);
   }

   private void tornarSuper(Usuario usuarioDecorado){
      System.out.println(":: ESSE USUÁRIO AGORA TEM PRIVILÉGIOS DE SUPER
      USUÁRIO ::");
   }

}
```

Padrões comportamentais: Strategy

Caso durante a modelagem os engenheiros de software definam duas classes com as mesmas características estruturais, à primeira vista, estas classes podem ser condensadas em uma única classe. No entanto, em algumas situações a única coisa que diferencia certas classes é o comportamento de seus objetos. Nesses casos, pode ser uma boa ideia isolar esses algoritmos que determinam as distinções de comportamento em classes separadas para, assim, selecionar o comportamento desejado – em tempo de execução.

O padrão Decorator trata desse problema, no entanto, os comportamentos podem ser herdados pelas subclasses das classes de negócio, o que, algumas vezes não é desejável.

Nesse sentido, o padrão Strategy é utilizado para definir famílias de algoritmos encapsulados e intercambiáveis entre objetos de cada família. Com isso, o comportamento de uma classe não será herdado por suas subclasses e, assim, o algoritmo pode variar de acordo com o cliente que o utilizar.

Por exemplo, considere a operação "efetuar cadastro". Tanto **Administrador** quanto **Estudante** e **Professor** possuem essa funcionalidade, embora naturalmente sejam implementadas de forma diferente de acordo com as características de cada classe. Dessa forma, o caminho natural seria colocar essas operações nas subclasses, no entanto, se a hierarquia descer de nível, ou seja, se **Administrador**, **Estudante** e **Professor** tiverem subclasses, esses comportamentos precisarão ser implementados em cada subclasse sucessivamente.

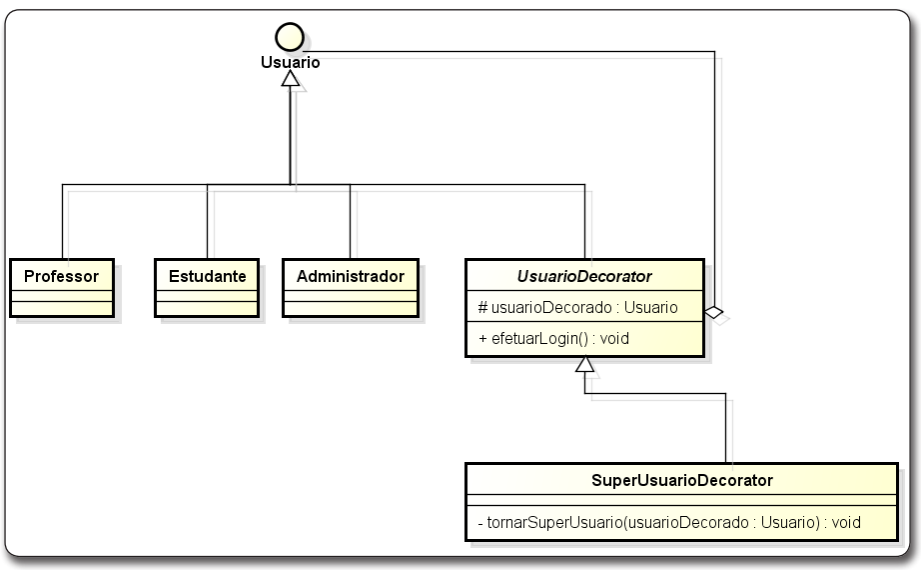

Figura 3. Utilização do padrão Decorator na aplicação de exemplo

O que o padrão Strategy sugere para resolver esse problema é definir o comportamento em interfaces e suas respectivas implementações e utilizar composição ao invés de herança, desacoplando o comportamento da classe que o utiliza. Com isso, o comportamento pode ser modificado sem afetar as classes, que por sua vez podem trocar de comportamento sem a necessidade de grandes alterações no código. A **Figura 4** demonstra a implementação do padrão *Strategy*.

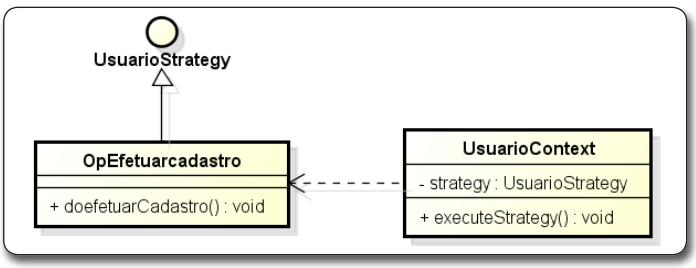

Figura 4. Diagrama de classes que ilustra o padrão Strategy

No nosso exemplo, vamos encapsular um algoritmo que, hipoteticamente, seria parte da operação **efetuarCadastro()**. O primeiro passo para implementar esse requisito utilizando o padrão Strategy seria criar uma interface, que no nosso caso, chamaremos de **UsuarioStrategy**, como mostra a **Listagem 12**. Essa interface será comum a todos os algoritmos suportados pela aplicação, ou seja, aqueles algoritmos encapsulados que serão adicionados em tempo de execução ao comportamento das classes.

Listagem 12. Código-fonte da interface UsuarioStrategy.

```
package br.com.devmedia.gestaoacademica.model;

public interface UsuarioStrategy {

   public void doEfetuarCadastro();

}
```

Repare que temos o método **doEfetuarCadastro()** declarado na interface **UsuarioStrategy**. Na sequência, então, implementaremos essa interface, criando classes concretas que serão responsáveis por encapsular o comportamento desse método. No nosso caso, apenas uma classe é necessária, pois estamos encapsulando apenas o comportamento do método que efetua o cadastro do usuário. Isso pode ser visto na **Listagem 13**.

Listagem 13. Código-fonte da OpEfetuarCadastro.

```
package br.com.devmedia.gestaoacademica.model;

public class OpEfetuarCadastro implements UsuarioStrategy {

   public void doEfetuarCadastro(){
      System.out.println(":: USUÁRIO CADASTRADO ::");
   }
}
```

Vamos agora criar a classe de contexto. Essa classe usará a interface **UsuarioStrategy** para chamar o algoritmo definido na classe concreta, no nosso caso, **OpEfetuarCadastro**. Sendo assim, objetos dessa classe de contexto receberão as requisições do cliente e delegarão essas requisições ao objeto Strategy, ou seja, objetos da classe **OpEfetuarCadastro** são instanciados pelo cliente e passados para o contexto. Feito isso, o cliente só precisa interagir com o contexto. O código da classe de contexto – **UsuarioContext** – pode ser visto na **Listagem 14**.

Listagem 14. Código-fonte de UsuarioContext.

```
package br.com.devmedia.gestaoacademica.model;

public class UsuarioContext {

   private UsuarioStrategy strategy;

   public UsuarioContext(UsuarioStrategy strategy){
      this.strategy = strategy;
   }

   public void executeStrategy(){
      strategy.doEfetuarCadastro();
   }

}
```

Com isso, a aplicação do padrão Strategy está finalizada, bem como todos os demais padrões que vimos nesse artigo. Na próxima subseção, implementaremos a camada de cliente, que conforme citado durante a explicação do exemplo, será responsável por acessar as camadas da aplicação e pode ser utilizada, portanto, para verificar seu funcionamento.

Testando a aplicação

O primeiro passo para implementar a nossa camada cliente e verificar o funcionamento dos padrões que acabamos de implementar é criar a classe **UsuarioCliente**. Essa classe deverá ter um método **main()**, que conterá o código das chamadas aos métodos que desejamos testar, como mostra a **Listagem 15**.

Para ilustrar bem o exemplo, testamos cada um dos padrões aplicados separadamente, como pode ser visto nos comentários no código na **Listagem 15**. Iniciamos pelo padrão *Singleton*, e depois obtemos um objeto do tipo **Administrador** por meio da *Abstract Factory*. Repare no uso da "fábrica de fábricas" – no trecho de código **FactoryMaker.getFactory("USUARIO")**. No nosso exemplo, só existe um tipo de fábrica – de **Usuario** – mas ainda assim o padrão foi utilizado em sua forma completa.

Na sequência, "decoramos" o objeto usuário, que se torna – hipoteticamente – super usuário. Finalmente, adicionamos o algoritmo de cadastro, que foi encapsulado por meio do padrão *Strategy* ao objeto **Usuario**.

Para verificar o resultado, basta executar a classe **UsuarioCliente** e observar na saída padrão – no nosso caso, no *prompt* de comando – os textos que serão impressos.

Listagem 15. Código-fonte de UsuarioCliente.

```
package br.com.devmedia.gestaoacademica.model;

public class UsuarioCliente {

   public static void main(String[] args) {

      //Testando o Singleton
      Administrador administrador = Administrador.getInstancia();

      //Testando a Abstract Factory
      AbstractFactory factory = FactoryMaker.getFactory("USUARIO");
      Usuario usuario = factory.getUsuario("ADMINISTRADOR");
      usuario.efetuarLogin();

      //Testando o Decorator
      usuario = new SuperUsuarioDecorator(new Professor());
      usuario.efetuarLogin();

      //Testando o Strategy
      UsuarioContext context = new UsuarioContext(new OpEfetuarCadastro());

         context.executeStrategy();

   }

}
```

O conceito de padrões de projeto de software, embora plenamente adotado na comunidade de desenvolvedores, também é alvo de críticas de todos os tipos. Alguns afirmam que a construção de catálogos como o GoF é perda de tempo, uma vez que problemas comuns devem ser solucionados de forma igualmente comuns. Outros defendem que a maioria dos padrões catalogados poderia ser simplificada ou simplesmente eliminada.

Há ainda alguns teóricos que afirmam que os padrões não se aplicam de maneira equivalente em todas as linguagens que dão suporte ao paradigma da orientação a objetos, ou que poderiam ser incluídos como funcionalidades em determinadas linguagens.

Mesmo que todas essas afirmações sejam verdadeiras, elas não ressaltam de forma alguma que a catalogação de padrões e seu respectivo uso pelos desenvolvedores envolve qualquer prejuízo na tarefa de programação. Na verdade, o que pode ser inferido é que, como qualquer trabalho, o uso inapropriado das ferramentas disponíveis pode aumentar a complexidade do trabalho e, dessa forma, trazer prejuízos concretos ao projeto.

Richard P. Gabriel, cientista da computação reconhecido por seu trabalho com a linguagem Lisp, autor do livro "Patterns of Software: Tales from the Software Community" ("Padrões de Software: Contos da Comunidade de Software", sem tradução no Brasil), diz que o catálogo GoF "ajuda os perdedores a perderem menos".

Vista de forma um tanto irônica, a afirmação de Gabriel sinaliza que, independente da crítica, o estabelecimento de padrões sempre contribuiu para o desenvolvimento de qualquer campo e, desde que utilizados de forma correta, os padrões devem contribuir decisivamente para o aprimoramento de qualquer projeto de software.

Nesse sentido, esse artigo não esgota, de forma alguma, o tema. Existem outros padrões importantes no catálogo GoF e há outros catálogos que merecem ser explorados, dada a sua contribuição para a área de desenvolvimento de software. No entanto, nossa análise contribui para demonstrar que, mais importante do que aplicar um determinado padrão, devemos compreendê-los em sua essência para, assim, decidirmos quando utilizá-lo. Mais importante do que os padrões em si, é desenvolver a capacidade de identificar qual padrão melhor se adequa ao problema que se apresenta.

Autor

Alessandro Jatobá
jatoba@jatoba.org
É Mestre em Ciência da Computação pela UFRJ, Doutorando em Engenharia também pela UFRJ com Doutorado-Sanduíche pelo Departamento de Engenharia e Projeto de Sistemas da Universidade de Waterloo, no Canadá. Trabalha com Java há mais de 15 anos, a maior parte deles liderando equipes de desenvolvedores em projetos Web. Também é professor universitário lecionando temas ligados ao desenvolvimento orientado a objetos.

Links:

Wiki sobre o Gang-of-Four.
http://c2.com/cgi/wiki?GangOfFour

Artigo de Obi Ezechukwu sobre Anti-padrões.
http://www.javaworld.com/author/Obi-Ezechukwu/

Date and Time API: Manipulando datas em Java

Manipule datas facilmente com a Date and Time API

Por muitos anos, na linguagem Java tivemos dificuldades com a manipulação de datas e horas. Como primeiro recurso disponível, tivemos a classe **java.util.Date**, mas que rapidamente passou a ser considerada complexa e bastante limitada. Em 1998, a IBM desenvolveu e disponibilizou a classe **java.util.Calendar**, com recursos como internacionalização, métodos para alterar individualmente o dia, o mês ou o ano de uma data, variáveis estáticas pré-definidas com a descrição dos dias da semana e dos meses do ano, entre outros que eram inexistentes na classe **Date**. A classe **Calendar**, embora tenha trazido maior flexibilidade à manipulação de datas, ainda era considerada complexa por grande parte dos programadores.

Como solução para **Date** e **Calendar**, em 2005 o projeto Joda Time foi lançado. Ele consiste de uma biblioteca externa ao JDK/JRE, criada por Stephen Colebourne, que rapidamente foi adotada por muitos desenvolvedores como a principal API para datas no Java.

Durante a idealização do Java 8, avaliou-se como necessário definir uma API nativa que correspondesse às necessidades dos desenvolvedores em relação à manipulação de datas. Como o projeto Joda Time foi bem aceito pela comunidade, optou-se por toma-lo como uma das principais referências. Stephen Colebourne veio então a ser um dos líderes deste novo projeto, que foi lançado no Java 8 como a especificação JSR-310, Date and Time API (apelidada de Java Time).

Segundo Colebourne, o grande problema encontrado nas classes **Date** e **Calendar** é que elas são mutáveis, ou seja, é possível alterar externamente o valor de um atributo desse tipo. Outro problema citado por Colebourne é que os anos iniciam a partir de 1900 e os meses a partir de zero. Por exemplo, o mês de Janeiro tem valor correspondente à zero (**0**), o mês de Fevereiro a um (**1**) e assim sucessivamente até dezembro, que tem o valor **11**. Além disso, há uma complexidade na manipulação de datas criadas a partir das classes **Date** e **Calendar** que não poderia ser corrigida sem descaracterizar completamente estas classes. Por exemplo, os métodos da classe **Calendar** não são considerados intuitivos, já que os nomes dados a eles muitas vezes não deixa clara as suas funções. Por sua vez, a classe **Date** já possui grande parte de seus métodos marcados como **deprecated** (descontinuado) e a inserção de novos métodos poderia deixar a classe mais confusa.

Assim sendo, foi necessário pensar em uma nova API, a qual deu origem ao pacote **java.time** e a classes como **LocalDate**, **LocalTime**, **Period**, **Instant**, **ZoneDateTime**, entre outras.

Como característica principal, todas as classes da API Java Time são imutáveis, corrigindo um problema que, conforme Colebourne, era grave nas classes **Date** e **Calendar**. Outro ponto positivo é que esta API fornece uma manipulação muito mais simples para trabalhar com objetos que armazenam apenas datas, apenas horas ou ambas simultaneamente. Por exemplo, você poderia armazenar o aniversário de alguém a partir da classe **MonthDay**. Um objeto desse tipo vai armazenar apenas os valores correspondentes ao mês e ao dia. E ainda temos as classes **Year** e **YearMonth**, para armazenar apenas o ano ou o ano e o mês em um mesmo objeto.

Dito isso, reforça-se que no decorrer deste artigo o leitor vai ter a chance de conhecer os recursos da API Java Time. Como esta API é nativa do Java 8, é preciso que o JDK e o JRE estejam ambos atualizados para a versão 8 do Java.

Fique por dentro

Este artigo apresenta ao leitor a JSR-310, uma especificação para o tratamento e manipulação de datas lançada com o Java 8. A Date and Time API, ou simplesmente Java Time, se mostra um grande recurso para substituir o uso das classes Date e Calendar, pois as classes dessa API têm métodos simples e intuitivos para oferecer uma melhor experiência a seus usuários. Sendo assim, esse tema é útil para todos os desenvolvedores que ainda não tiveram a chance de conhecer essa nova API e têm interesse em atualizar seus conhecimentos, viabilizando com isso a adoção das facilidades oferecidas por mais uma inovação do Java.

Nota

As JSRs (Java Specification Requests) são documentos formais que descrevem as especificações e tecnologias propostas que se pretende adicionar à plataforma Java. Estas propostas são formuladas e aprovadas por membros do JCP (Java Community Process).

Usando as classes LocalDate, LocalTime e LocalDateTime

É provável que as classes **LocalDate**, **LocalTime** e **LocalDateTime** se tornem as mais utilizadas por você ao lidar com essa API. Com elas podemos capturar a data e a hora atual do sistema, inserir e redefinir uma data ou horário por meio de métodos específicos, entre outras possibilidades. Os parâmetros destes métodos podem representar um dia, um mês, horas, minutos, segundos ou nanosegundos. Na **Listagem 1** é apresentado como capturar a data atual, a hora e a data, e apenas a hora, com o método **now()**.

Listagem 1. Capturando apenas a data, apenas a hora, e a data e a hora.

```
LocalDate dataAtual = LocalDate.now();
System.out.println("Data: " + dataAtual);

LocalTime horaAtual = LocalTime.now();
System.out.println("Hora: " + horaAtual);

LocalDateTime dataHoraAtual = LocalDateTime.now();
System.out.println("Data e Hora: " + dataHoraAtual);
```

A saída no console após a execução desse código é:

```
Data: 2014-08-21
Hora: 17:48:07.097
Data e Hora: 2014-08-21T17:48:07.097
```

Analisando o resultado exposto no console é possível notar que cada uma dessas classes retorna valores distintos. Assim, é mais fácil obter os valores pretendidos. Além disso, note que as classes envolvidas neste processo não usam instâncias, já que são formadas por métodos estáticos e pelo padrão Factory Method.

Definindo um time-zone

Um time-zone, ou fuso horário, define a hora em uma determinada região da Terra. Levando em consideração esse conceito, o método **now()** – da classe **LocalDateTime** – vai retornar a hora conforme o fuso horário da região em que a aplicação está sendo executada. Por exemplo, se a aplicação é web, será levado em consideração o fuso do local onde ela está hospedada. Caso haja a necessidade de definir um time-zone que não seja o padrão, o método **now()** possui duas sobrecargas para isto.

A diferença entre estas sobrecargas é o parâmetro atribuído ao método **now()**, que pode ser um objeto do tipo **java.time.Clock** ou um objeto do tipo **java.time.ZoneId**. Na **Listagem 2** temos um exemplo que retorna a data e a hora do sistema local, e em seguida a data e a hora no Japão e em Atenas, na Grécia.

Listagem 2. Recuperando data e hora a partir de diferentes time-zones.

```
LocalDateTime datHoraAtual = LocalDateTime.now();
System.out.println("Data atual local: " + datHoraAtual);

LocalDateTime dataHoraAtualJapao = LocalDateTime.now(Clock.system(ZoneId
.of("Japan")));
System.out.println("Data atual no Japão: " + dataHoraAtualJapao);

LocalDateTime dataHoraAtualAtenas = LocalDateTime.now(ZoneId.of("Europe/
Athens"));
System.out.println("Data atual no Atenas: " + dataHoraAtualAtenas);
```

A saída no console após a execução desse código é:

```
Data atual local: 2014-08-21T18:09:24.271
Data atual no Japão: 2014-08-22T06:09:24.272
Data atual no Atenas: 2014-08-22T00:09:24.273
```

Observe que o método **now()** – da classe **LocalDateTime** – foi usado com três assinaturas distintas. A primeira delas retorna a data e a hora do sistema local a partir da assinatura padrão do método. Para recuperar a data e a hora no Japão, foi usado o método estático **system()** da classe **java.time.Clock**. Este método recebe como parâmetro um objeto do tipo **java.time.ZoneId**, configurado com o time-zone desejado. Para adicionar o time-zone, utiliza-se o método **ZoneId.of()** atribuído de uma **String** que representa o local do fuso horário. A terceira assinatura do método **now()** recebeu como parâmetro um objeto do tipo **ZoneId** ao invés de um objeto do tipo **Clock**, como demonstra o exemplo de código que busca a data e a hora em Atenas.

Analisando os valores impressos no console é possível notar as diferenças entre as datas e os horários de cada um dos três locais. Para saber quais time-zones podem ser utilizados como parâmetro do método **of()**, a classe **ZoneId** provê o método **getAvailableZoneIds()**. Este retorna uma lista com todos os time-zones disponíveis na API Java Time.

Nota

Os fusos horários, ou time-zones, foram estabelecidos através de uma reunião composta por representantes de 25 países em Washington (EUA), em 1884. Nessa ocasião foi realizada uma divisão do globo terrestre em 24 fusos horários. No total, o Brasil é dividido em quatro zonas horárias, que são: UTC-2, UTC -3 (hora oficial de Brasília), UTC -4 e UTC -5.

Alterando datas e horas

Outro recurso simples da API Java Time são os métodos providos para a atualização de datas e horas, como demonstra o exemplo na **Listagem 3**. Uma série de métodos está disponível para adicionar ou subtrair dias, meses e anos de datas, como também horas, minutos, segundos e nanosegundos de valores temporais. Entre esses métodos, temos alguns como: **plusDays()**, **plusMonths()**, **plusMinutes()** e **plusHours()** para adição; e **minusDays()**, **minusMonths()**, **minusHours()** e **minusMinutes()** para subtração de valores.

Listagem 3. Incremento para datas e horas.

```
LocalDateTime dataHora = LocalDateTime.now();
System.out.println("Data e Hora atuais: " + dataHora);

dataHora = dataHora.plusMonths(1).plusDays(4);
System.out.println("Adicionando 1 mês e 4 dias " + dataHora);

dataHora = dataHora.plusHours(2).plusMinutes(10);
System.out.println("Adicionando 2 horas e 10 minutos " + dataHora);
```

A saída no console após a execução desse código é a seguinte:

```
Data e Hora atuais: 2014-08-21T18:41:08.522
Adicionando 1 mês e 4 dias 2014-09-25T18:41:08.522
Adicionando 2 horas e 10 minutos 2014-09-25T20:51:08.522
```

O código atribui à variável **dataHora** a data e hora atual do sistema, imprimindo estes valores no console. Em seguida, esta variável recebe um incremento de 1 mês e 4 dias sobre a data e hora capturadas na linha anterior.

Como os objetos da Java Time são imutáveis, para modificar o valor de uma variável é necessário atribuir a ela a alteração realizada, ou seja, não basta executar **dataHora.plusMonths(1).plusDays(4)**, é preciso atribuir o retorno deste processo à variável, conforme o código: **dataHota = dataHora.plusMonths(1).plusDays(4)**.

Com isso, veja na saída do console que a data passou de **2014-08-21** para **2014-09-25**. O mesmo processo é realizado para adicionar um novo horário à data já modificada. No entanto, para atualizar a hora e os minutos, utilizamos os métodos **plusHours()** e **plusMinutes()**.

Voltando à saída do console, note que a hora **18:41:08.522** foi alterada para **20:51:08.522**. Caso o interesse seja de subtrair a data ou o horário, utilize os métodos que iniciam pelo nome **minus**, como alguns dos já citados.

Enquanto os métodos do tipo **plus** e **minus** adicionam ou subtraem valores de uma data, podemos determinar um ano específico usando o método **withYear()**. Este método deve ser atribuído com um valor do tipo **int** que representa o ano desejado, como mostra o exemplo da **Listagem 4**.

Listagem 4. Definindo um ano específico com o método withYear().

```
LocalDate data = LocalDate.now();
System.out.println("Data atual: " + data);

data = data.withYear(2015);
System.out.println("Ano alterado: " + data);
```

A saída no console da execução desse código é a seguinte:

```
Data atual: 2014-08-21
Ano alterado: 2015-08-21
```

Nesse código, uma data é atribuída à variável **data**, através do método **LocalDate.now()**, e depois é impressa no console. Em seguida, essa data tem o seu ano modificado para 2015, por meio do método **withYear()**. A saída no console comprova a alteração. Além disso, há a possibilidade de alterar o mês, com o método **withMonth()**, o dia do mês, com **withDayOfMonth()**, e o dia do ano, com **withDayOfYear()**, conforme a **Listagem 5**.

Listagem 5. Alterando datas com os métodos do tipo with.

```
LocalDate data = LocalDate.now();
System.out.println("Data atual: " + data);

data = data.withMonth(10);
System.out.println("Alterando para o mês 10: " + data);

data = data.withDayOfMonth(29);
System.out.println("Alterando para o dia 29 do mês atual: " + data);

data = data.withDayOfYear(25);
System.out.println("Alterando para o 25° dia do ano: " + data);
```

A execução desse código tem como resultado no console:

```
Data atual: 2014-08-21
Alterando para o mês 10: 2014-10-21
Alterando para o dia 29 do mês atual: 2014-10-29
Alterando para o 25° dia do ano: 2014-01-25
```

Veja que a data atual é **2014-08-21**. A partir disso, usando o método **withMonth()** a data foi alterada para o mês de Outubro (**10**). Logo após, com o método **withDayOfMonth()**, o dia do mês passou de **21** para **29**. Por fim, a data foi modifica para o vigésimo quinto dia do ano, resultando em **2014-01-25**.

Nas classes **LocalTime** e **LocalDateTime** você encontrará métodos do tipo **with** para atribuir valores temporais para horas (**withHour()**), para minutos (**withMinute()**), para segundos (**withSecond()**) e nanosegundos (**withNano()**).

Criando datas e horas a partir de parâmetros

Para criar datas e horas a partir de parâmetros, os objetos **LocalDate**, **LocalTime** e **LocalDateTime** podem ser inicializados com o método **now()**. Deste modo tais objetos são definidos com a data e hora corrente do sistema operacional. Contudo, existe uma forma de inicializar estes objetos com valores referentes a uma data e/ou hora por meio de parâmetros adicionados ao método **of()**. Esta é uma opção útil para criar objetos que devam receber valores que não sejam a data e/ou hora corrente do sistema operacional, mas sim aqueles atribuídos, por exemplo, via interface gráfica de um aplicativo de dados cadastrais. A **Listagem 6** demonstra como usar o método **of()**.

A saída no console para esse código será:

```
Data pré-definida: 2014-12-25T15:35:15
```

O código tem uma lista de variáveis que representam, individualmente, os valores que formam uma data e uma hora.

Tais parâmetros são adicionados ao método **of()** de **LocalDateTime** para formar a data e o horário impressos na saída do console. O método **of()** está presente também na classe **LocalDate**, mas limitado a valores de datas e também na classe **LocalTime**, mas limitado a valores relacionados a horas.

Listagem 6. Criando datas a partir do método of().

```
int ano = 2014;
int mes = 12;
int dia = 25;
int hora = 15;
int min = 35;
int sec = 15;

LocalDateTime localDateTime = LocalDateTime.of(ano, mes, dia, hora, min, sec);
System.out.println("Data pré-definida: " + localDateTime);
```

Comparando datas e horas

Outro recurso interessante da API Java Time é a possibilidade de testar se uma data é anterior, posterior ou igual a outra. Para isso, a API fornece os métodos **isBefore()**, **isAfter()** e **isEqual()**, os quais têm como retorno um valor booleano. Um exemplo de como utilizar esses métodos é descrito na **Listagem 7**. Veja que as primeiras duas variáveis, **anoNovo** e **natal**, são do tipo **LocalDate**, e três comparações são realizadas entre elas.

O método **isBefore()** testa se o valor da data contida na variável **anoNovo** é anterior ao valor da data contida em **natal**. O método **isAfter()** testa se o valor de **anoNovo** é posterior ao valor de **natal**. Já o método **isEqual()** faz um teste de igualdade. Neste exemplo, as saídas no console serão, respectivamente: **true**, **false** e **false**.

Listagem 7. Comparando datas e horários.

```
LocalDate anoNovo = LocalDate.of(2014, 1, 1);
LocalDate natal = LocalDate.of(2014, 12, 25);

System.out.println(anoNovo.isBefore(natal));
System.out.println(anoNovo.isAfter(natal));
System.out.println(anoNovo.isEqual(natal));

LocalTime almoco = LocalTime.of(12, 0);
LocalTime jantar = LocalTime.of(20, 30);

System.out.println(almoco.isBefore(jantar));
System.out.println(almoco.isAfter(jantar));
System.out.println(almoco.equals(jantar));
```

Já a classe **LocalTime** não possui o método **isEqual()** para a comparação de igualdade. Ao invés disso, o método **equals()** da classe **java.lang.Object** foi sobrescrito para realizar a comparação entre objetos desse tipo, como as variáveis **almoco** e **jantar** na **Listagem 7**. A saída no console para o método **isBefore()** será **true** e para os métodos **isAfter()** e **equals()** será **false**.

A classe **LocalDateTime** é caracterizada por trabalhar com data e hora simultaneamente, conforme mostra a **Listagem 8**. Neste código o objeto **anoNovoAlmoco**, do tipo **LocalDateTime**, foi criado a partir de uma sobrecarga do método **of()** que recebe as variáveis **anoNovo** (**LocalDate**) e **almoco** (**LocalTime**). Já o objeto **natalJantar** foi criado com os valores de **natal** e **jantar**. A comparação entre essas variáveis foi realizada com o uso dos métodos **isBefore()**, **isAfter()** e **isEqual()** e o resultado obtido foi o seguinte: **true**, **false** e **false**.

Listagem 8. Comparando objetos do tipo LocalDateTime.

```
LocalDate anoNovo = LocalDate.of(2014, 1, 1);
LocalDate natal = LocalDate.of(2014, 12, 25);

LocalTime almoco = LocalTime.of(12, 0);
LocalTime jantar = LocalTime.of(20, 30);

LocalDateTime anoNovoAlmoco = LocalDateTime.of(anoNovo, almoco);
LocalDateTime natalJantar = LocalDateTime.of(natal, jantar);

System.out.println(anoNovoAlmoco.isBefore(natalJantar));
System.out.println(anoNovoAlmoco.isAfter(natalJantar));
System.out.println(anoNovoAlmoco.isEqual(natalJantar));
```

Comparando datas e horas em diferentes time-zones

Sabemos que existem diversos time-zones e, por conta disso, uma comparação de datas e horários entre eles não pode ser realizada com a classe **LocalDateTime**, já que seu método **of()**, para criar uma data/hora, não aceita como parâmetro a adição de um fuso horário. Para esse tipo de comparação a classe a ser utilizada é a **java.time.ZoneDateTime**. Deste modo, consegue-se verificar, por exemplo, se a data e a hora de algum lugar da América do Sul correspondem com a data e a hora de algum lugar da América do Norte, Europa, Ásia ou Oceania.

Na **Listagem 9** temos um exemplo de uma data sendo comparada com o método **isEqual()**. Esta verificação vai levar em consideração o time-zone da cidade de São Paulo com o time-zone da cidade de Melbourne, Austrália.

Listagem 9. Comparando datas entre São Paulo e Melbourne.

```
ZonedDateTime saoPaulo =
    ZonedDateTime.of(2014, 8, 30, 18, 0, 0, 0, ZoneId.of("America/Sao_Paulo"));
ZonedDateTime melbourne =
    ZonedDateTime.of(2014, 8, 30, 18, 0, 0, 0, ZoneId.of("Australia/Melbourne"));
System.out.println(saoPaulo.isEqual(melbourne));
```

O retorno da comparação entre as variáveis **saoPaulo** e **melbourne**, que possuem a data/hora **2014-08-30 18:00:00**, vai ser **false**. Isso porque, embora os valores sejam idênticos, os time-zones atribuídos às variáveis são diferentes.

Para que a comparação entre datas/horas de diferentes time-zones resulte **true**, é necessário ajustar as datas e/ou horários levando em consideração a diferença de fusos horários entre os locais. Por exemplo, o fuso horário de Melbourne está treze horas à frente do fuso de São Paulo. Assim, devemos ajustar o horário adicionando à variável **melbourne** treze horas e também um dia a mais na data, como indica a **Listagem 10**.

Listagem 10. Ajustando o fuso horário entre São Paulo e Dubai.

```
ZonedDateTime saoPaulo =
    ZonedDateTime.of(2014, 8, 30, 18, 0, 0, 0, ZoneId.of("America/Sao_Paulo"));

ZonedDateTime melbourne =
    ZonedDateTime.of(2014, 8, 31, 7, 0, 0, 0, ZoneId.of("Australia/Melbourne"));

System.out.println(saoPaulo.isEqual(melbourne));
```

A comparação realizada não avalia se os valores referentes à data e hora são iguais entre as variáveis, mas sim se eles correspondem ao mesmo período no tempo em relação aos diferentes fusos horários. Caso esse período no tempo seja idêntico, então o retorno será **true**, mesmo que visualmente as datas/horários sejam diferentes.

Trabalhando com as classes Year, YearMonth e MonthDay

As classes **MonthDay**, **YearMonth** e **Year** estão presentes no pacote **java.time** e retornam, a partir de uma data, valores como o mês, o dia e o ano. É possível, por exemplo, recuperar o mês de uma data e formatar seu retorno com **java.time.format.TextStyle**, como também definir o idioma em que o retorno será exibido, por meio da classe **java.util.Locale**. Veja na **Listagem 11** como proceder.

Listagem 11. Usando a classe YearMonth.

```
LocalDate data = LocalDate.of(1975, 12, 19);

YearMonth mesDoAno = YearMonth.from(data);

System.out.println(  mesDoAno.getMonth() + "/" + mesDoAno.getYear() );

System.out.println(
    mesDoAno.getMonth().getDisplayName(TextStyle.FULL, Locale.getDefault())
    + "/" + mesDoAno.getYear()
);

System.out.println(
    mesDoAno.getMonth().getDisplayName(TextStyle.SHORT, Locale.getDefault())
    + "/" + mesDoAno.getYear()
);

System.out.println(
    mesDoAno.getMonth().getDisplayName(TextStyle.NARROW,
    Locale.getDefault()) + "/" + mesDoAno.getYear()
);
```

A saída no console será:

```
DECEMBER/1975
Dezembro/1975
dez/1975
D/1975
```

No código foi adicionada a data "19/12/1975" à variável **data** da classe **LocalDate**. A partir disso, com a classe **YearMonth** consegue-se recuperar o mês e o ano lá contidos. Para isso, é preciso atribuir o objeto **data** como parâmetro do método **YearMonth.from()**. Este método inicializará a variável **mesDoAno**, do tipo **YearMonth**, e por meio dos métodos **getYear()** e **getMonth()** conseguimos recuperar o ano e o mês atribuídos. Para formatar a saída a ser impressa no console, o método **getMonth()** fornece acesso ao método **getDisplayName()**. Este método exige que dois parâmetros sejam adicionados: um do tipo **TextStyle** e outro do tipo **Locale**.

O valor atribuído a **TextStyle** define o formato no qual o mês será impresso no console, podendo ser a descrição completa do mês (**TextStyle.FULL**), o mês abreviado (**TextStyle.SHORT**) ou apenas a primeira letra (**TextStyle.NARROW**). Para imprimir o mês em forma de número, você pode usar os estilos **TextStyle.FULL_STANDALONE**, **TextStyle.SHORT_STANDALONE** ou **TextStyle.NARROW_STANDALONE**, entretanto, não há nenhuma diferença entre eles.

Enquanto **TextStyle** define a forma como o mês será impresso no console, o parâmetro **Locale** vai definir a idioma que será utilizado para isso. Quando se atribui **Locale.getDefault()**, será adotado o idioma padrão de onde o sistema está sendo executado. Contudo, por meio de constantes da classe **Lacale** podemos optar por outros idiomas, como o Inglês (**Locale.ENGLISH**), o Chinês (**Locale.CHINA**), o Francês (**Locale.FRENCH**), entre outros.

Outra forma para recuperar o ano e o mês em uma data é utilizando as classes **Year** e **MonthDay**. Esta última possibilita ainda que seja retornado o dia. Este processo pode ser observado na **Listagem 12**, na qual a variável **data** é passada como parâmetro para o método **Year.from()**. Note que a saída no console exibe o ano recuperado pelo método **from()** da classe **Year**. Já a classe **MonthDay** fornece o método **getDayOfMonth()**, que tem como objetivo recuperar o dia de uma data, e o método **getMonth()**, que recupera o mês.

Listagem 12. Exemplo de uso das classes Year e MonthDay.

```
LocalDate data = LocalDate.of(1975, 12, 19);

System.out.println( "Ano: " + Year.from(data) );

MonthDay diaDoMes = MonthDay.from(data);
System.out.println( diaDoMes.getDayOfMonth() + "/" + diaDoMes.getMonth() );
```

A saída no console após a execução do trecho de código da **Listagem 12** é:

```
Ano: 1975
19/DECEMBER
```

Criando datas a partir de Year, YearMonth e MonthDay

Vimos que as classes **Year**, **YearMonth** e **MonthDay** têm métodos que retornam o dia, o mês e o ano de uma data. Contudo, estas classes ainda possibilitam que instâncias do tipo **LocalDate** ou **LocalDateTime** sejam criadas com datas e horas definidas por

meio de parâmetros. Para isso, temos os seguintes métodos disponíveis: **atMonth()**, **atDay()** e **atMonthDay()** para a classe **Year**; **atDay()** e **atEndOfMonth()** para a classe **YearMonth**; e **atYear()** para a classe **MonthDay**. Veja na **Listagem 13** um exemplo de como criar uma data usando alguns destes métodos.

Listagem 13. Gerando datas com os métodos at.

```
LocalDate data = Year.of(2010).atMonth(9).atDay(5);
System.out.println(data);
```

A saída impressa no console será a data **2010-09-05**, resultante dos valores adicionados aos métodos da classe **Year**. Com esta classe é permitido ainda atribuir um horário através do método **atTime()**. Porém, diferentemente de **atDay()**, que retorna um objeto **LocalDate**, **atTime()** tem como retorno um objeto do tipo **LocalDateTime**, conforme mostra a **Listagem 14**.

Listagem 14. Gerando uma hora com o método atTime().

```
LocalDateTime dataHora = Year.of(2010).atMonth(9).atDay(5).atTime(14, 20, 30);
System.out.println(dataHora);
```

A saída no console da execução desse código será a seguinte:

```
2010-09-05T14:20:30
```

Enums para os meses e dias da semana

Já vimos que a API Java Time fornece a formatação do estilo de texto com a enumeração **TextStyle**, a partir da qual podemos determinar o modo como a descrição do mês será retornada para o usuário. Além desta, existem outros dois tipos de enumeração, que são: **DayOfWeek**, para a descrição do dia da semana; e **Month**, para a descrição dos meses. Assim, ao invés de adicionar um valor numérico para um determinado mês, se atribui a descrição desse mês com o parâmetro **Month** correspondente ao mês desejado. No exemplo da **Listagem 15** pode-se observar como definir Setembro como sendo o mês desejado.

Listagem 15. Criando uma data com o auxílio do enum Month.

```
LocalDate localDate = LocalDate.of(2014, 9, 19);
System.out.println("Saída 1: " + localDate);

localDate = LocalDate.of(2014, Month.SEPTEMBER, 19);
System.out.println("Saída 2: " + localDate);
```

A partir desse código, a saída no console será:

```
Saída 1: 2014-09-19
Saída 2: 2014-09-19
```

Veja que as datas impressas são idênticas. A diferença está na forma como foram atribuídas ao método **of()**. Nesse ponto vale ressaltar que todos os métodos da API Java Time que aceitam o mês como parâmetro são sobrecarregados para aceitar ou o valor numérico do mês ou um enumerado da classe **Month**.

O enum **DayOfWeek**, por outro lado, é usado para representar os dias da semana. Em um exemplo simples, **DayOfWeek** poderia ser aplicado para comparar se o dia da semana de uma data corresponde ao mesmo dia da semana de outra data. Esta comparação pode ser realizada a partir da classe **LocalDate**, que fornece o método **getDayOfWeek()**, o qual tem como retorno o tipo **DayOfWeek**. Deste modo, em um teste lógico, poderíamos verificar se uma data armazenada em um objeto **LocalDate** corresponde a um domingo, a uma segunda-feira, a uma terça-feira, ou qualquer outro dia da semana.

Listagem 16. Exemplo comparando dias da semana.

```
LocalDate dt1 = LocalDate.of(2013, Month.APRIL, 6);
LocalDate dt2 = LocalDate.of(2014, Month.APRIL, 6);
LocalDate dt3 = LocalDate.of(2015, Month.APRIL, 6);

List<LocalDate> localDates = Arrays.asList(dt1, dt2, dt3);
for (LocalDate date : localDates) {
  switch (date.getDayOfWeek()) {
      case SATURDAY:
        System.out.println(date + " é Sábado");
        break;
      case SUNDAY:
        System.out.println(date + " é Domingo");
        break;
      default:
        System.out.println(
          date + " é " +
          date.getDayOfWeek().getDisplayName(TextStyle.FULL,
           Locale.getDefault()));
  }
}
```

A saída após a execução do código da **Listagem 16** é a seguinte:

```
2013-04-06 é  Sábado
2014-04-06 é  Domingo
2015-04-06 é  Segunda-feira
```

Veja que temos um código que cria uma lista com três datas. Essas datas têm em comum o dia e o mês, porém são de anos diferentes. Usando um teste condicional do tipo **switch-case**, podemos verificar, por exemplo, se os dias da semana destas datas são referentes a um sábado (**DayOfWeek.SATURDAY**) ou a um domingo (**DayOfWeek.SUNDAY**). E para o caso de alguma(s) destas datas corresponder(em) a algum(ns) dos demais dias da semana, a chamada **date.getDayOfWeek().getDisplayName()** foi implementada no bloco **default** para exibir esse(s) dia(s).

Com códigos simples como este podemos tratar de forma fácil e diferenciada cada um dos dias da semana, apenas os dias úteis, apenas os sábados e os domingos... enfim, as possibilidades são inúmeras.

Formatando datas

A formatação de datas também recebeu um tratamento especial na API Java Time, tornando muito mais fácil esse processo em relação ao que é necessário quando se trabalha com **Date** e **Calendar**. Por exemplo, para formatar uma data do tipo **LocalDateTime**, baste usar o método **format()**, atribuindo como parâmetro um objeto do tipo **java.time.format.DateTimeFormatter** com o formado desejado para a data.

Listagem 17. Exemplo de formatação de datas.

```
LocalDateTime dataHora = LocalDateTime.now();
System.out.println(dataHora);
System.out.println(dataHora.format(DateTimeFormatter.ofPattern("dd/MM/yyyy
hh:MM:ss")));
```

Ao executar o código da **Listagem 17** teremos como saída:

```
2014-08-22T17:39:05.759
22/08/2014 05:08:05
```

Nesse código um objeto do tipo **LocalDateTime** é criado a partir do método **now()** e duas saídas no console são esperadas. A primeira delas é uma saída natural, isto é, sem o tratamento do formato para exibição da data. Já para a segunda saída o método **format()** é utilizado. Este método recebe como parâmetro um objeto do tipo **DateTimeFormatter**, o qual pode ser obtido por meio do retorno do método **DateTimeFormatter.ofPattern()**. O método **ofPattern()** deve conter um parâmetro **String** que representa o formato desejado a ser apresentado na saída do console.

O Java Time traz também algumas formatações padrões que podem ser utilizadas. Para isso, basta substituir o método **ofPattern()** por **ofLocalizedDate()**. Este método deve receber como parâmetro o padrão desejado para a data a partir de um estilo fornecido por **java.time.format.FormatStyle**, como mostra o exemplo de código da **Listagem 18**.

Listagem 18. Formatando datas com FormatStyle.

```
LocalDate dataHora = LocalDate.now();
System.out.println(dataHora.format(DateTimeFormatter.
ofLocalizedDate(FormatStyle.FULL)));
System.out.println(dataHora.format(DateTimeFormatter.
ofLocalizedDate(FormatStyle.LONG)));
System.out.println(dataHora.format(DateTimeFormatter.
ofLocalizedDate(FormatStyle.MEDIUM)));
System.out.println(dataHora.format(DateTimeFormatter.
ofLocalizedDate(FormatStyle.SHORT)));
```

A saída no console para esse código será:

```
Sexta-feira, 22 de Agosto de 2014
22 de Agosto de 2014
22/08/2014
22/08/14
```

Calculando a diferença entre datas

No Java, sempre foi muito complicado calcular a diferença entre datas, pois as classes **Date** e **Calendar** não fornecem suporte para isto. Pensando nisso, na API de datas foram desenvolvidos métodos que viabilizam a realização de procedimentos desse tipo, facilitando bastante a vida do programador.

Trabalhando com o Enum ChronoUnit

Para demonstrar esse recurso na prática, vamos recuperar a diferença de dias e de meses entre duas datas usando o enum **java.time.temporal.ChronoUnit**. Em nosso exemplo, apresentado na **Listagem 19**, temos a variável **tiradentes**, com a data do feriado de Tiradentes (21 de Abril), e a variável **procRepublica**, com a data do feriado da Proclamação da República (de Novembro). Vamos então calcular a quantidade de dias e de meses entre essas datas.

Listagem 19. Calculando a diferença de dias e meses.

```
LocalDate tiradentes = LocalDate.of(2014, 4, 21);
LocalDate procReplublica = LocalDate.of(2014, 11, 15);

long dias = ChronoUnit.DAYS.between(tiradentes, procReplublica);
System.out.printf("São %s dias de diferença.", dias);

long meses = ChronoUnit.MONTHS.between(tiradentes, procReplublica);
System.out.printf("\nSão %s meses de diferença.", meses);
```

A saída no console é a seguinte:

```
São 208 dias de diferença.
São 6 meses de diferença.
```

Como pode ser verificado, o enum **ChronoUnit.DAYS** informa ao método **between()** que ele deve retornar a diferença de dias entre as duas datas passadas como parâmetro. Já **ChronoUnit.MONTHS** diz ao método **between()** para retornar o número de meses entre essas datas. Veja, na saída do console, que a mensagem informa que são **208** dias de diferença entre as duas datas. Já para meses, a diferença resultante é **6**. Com o uso de **ChronoUnit** consegue-se ainda recuperar a diferença entre datas em anos, décadas, horas, minutos, entre outras medidas de tempo.

Usando a classe Period

Há também outra forma de obter a diferença entre datas: usando a classe **java.time.Period** e seu método **between()**. Essa classe fornece métodos como: **getDays()**, para recuperar a quantidade de dias; **getMonths()**, para a quantidade de meses; e **getYears()**, para obter a quantidade de anos. Porém, diferentemente de **ChronoUnit**, a classe **Period** não retorna a diferença em horas, minutos ou segundos. Veja na **Listagem 20** um exemplo que calcula a diferença em anos, meses e dias entre os títulos de Campeão Brasileiro de 2009 e da Copa do Brasil de 2013 do Clube de Regatas do Flamengo.

Listagem 20. Calculando a diferença de datas com Period.

```
LocalDate brasileiro = LocalDate.of(2009, 12, 6);
LocalDate copaDoBrasil = LocalDate.of(2013, 11, 27);

Period dif = Period.between(brasileiro, copaDoBrasil);

System.out.printf("%s anos, %s meses e %s dias", dif.getYears(), dif.getMonths(),
dif.getDays() );
```

A saída no console será a seguinte:

```
3 anos, 11 meses e 21 dias
```

Este resultado exibe os valores referentes a cada unidade de tempo retornados pelos métodos **getYears()**, **getMonths()** e **getDays()** da classe **Period**.

Usando a classe Duration

Para o cálculo da quantidade de horas, minutos, segundos, nanosegundos e até dias entre duas datas, o Java Time provê também a classe **java.time.Duration**. Assim como em **Period** e em **ChronoUnit**, o método **between()** da classe **Duration** é o indicado para esta operação.

Dito isso, suponha que você deseja saber a quantidade de dias, horas, minutos e segundos que durou o horário de verão de 2013/2014. Com o código apresentado na **Listagem 21** rapidamente obtemos essas informações.

Listagem 21. Buscando a diferença de tempo com a classe Duration.

```
LocalDateTime inicio = LocalDateTime.of(2013, 10, 20, 0, 0, 0);
LocalDateTime fim = LocalDateTime.of(2014, 2, 16, 0, 0, 0);

Duration dur = Duration.between(inicio, fim);

System.out.printf(
"%s dias %s horas, %s minutos e %s segundos",
dur.toDays(), dur.toHours(), dur.toMinutes(), dur.getSeconds()
);
```

Sabendo que o horário de verão começou no dia 20 de Outubro de 2013, às 24h00min, e terminou em 16 de Fevereiro de 2014, às 24h00min, a saída no console será a seguinte:

```
119 dias 2856 horas, 171360 minutos e 10281600 segundos
```

Como se pode notar, o cálculo da diferença entre datas ficou extremamente fácil. A API Java Time simplificou esse processo de modo que basta fornecer as datas a um dos métodos **between()** para que o resultado seja calculado para você.

Com todos os recursos desta nova API lançada com o Java 8, programadores que desenvolvem sistemas com cadastros que envolvem agendamentos de consultas passam a ter uma forma muito mais simples de implementar métodos para este fim. A Java Time também facilita a implementação de sistemas que precisam obter a diferença de anos, meses ou dias para a realização de cálculos de taxas de juros e multas aplicadas sobre cobranças ou pagamentos de produtos e serviços, entre muitas outras possibilidades.

Embora o artigo tenha abordado um conteúdo bem abrangente sobre a API Java Time, é aconselhado que o leitor estude mais profundamente a documentação das classes que envolvem esta API, visando conhecer outras particularidades que possam não ter sido apresentadas e consequentemente ampliar seu domínio sobre o assunto.

Autor

Marcio Ballem de Souza
marcio@mballem.com
É Bacharel em Sistemas de Informação pelo Centro Universitário Franciscano em Santa Maria/RS. Tem experiência com desenvolvimento Delphi e Java em projetos para gestão pública e acadêmica. Possui certificação em Java, OCPJP 6. Mantém o blog www.mballem.com e http://twitter.com/mballem.

Links:

Página oficial com a descrição da JSR-310.
https://jcp.org/en/jsr/detail?id=310

Documentação do Java 8 e da API Java Time.
http://docs.oracle.com/javase/8/docs/api/index.html

Artigo sobre semelhanças entre a Joda Time e a Java Time.
http://tinyurl.com/lss86c3

Entrevista com Stephen Colebourne, líder do projeto Java Time.
http://tinyurl.com/qdt8c99

Site oficial da biblioteca Joda Time.
http://www.joda.org/joda-time/

POO em Java: reusabilidade e eficiência em seu código – Parte 1

Produza código correto, ágil e otimizado com os poderosos recursos da linguagem Java

Este artigo faz parte de um curso

Por ser uma linguagem orientada a objetos, todos os programas desenvolvidos em Java são compostos por um ou mais objetos que colaboram entre si para resolver um determinado problema. Como os conceitos do mundo real podem ser modelados na forma de objetos distintos e interdependentes, torna-se intuitivo desenvolver programas orientados a objetos para as mais diversas finalidades.

A correta produção de código orientado a objetos leva a muitos benefícios, como a reutilização de código, melhor organização deste, facilidade de manutenção, melhor escalabilidade da aplicação, entre outros.

Além do paradigma de programação orientado a objetos, existem muitos outros paradigmas propostos na literatura, aplicados a diferentes situações, como o paradigma lógico, para tratar de conhecimento usando regras lógicas, o paradigma funcional, que permite estruturar um programa na forma de chamadas a funções, o estruturado, que é baseado no conceito de sub-rotinas, entre outros.

O paradigma estruturado precede o paradigma orientado a objetos, e foi muito utilizado no passado, quando começaram a se tornar importantes os conceitos de função e sub-rotina. Porém, a programação estruturada não tem o mesmo poder de organização da informação e procedimentos quando comparada à programação orientada a objetos, que divide o programa em conceitos, ou seja, classes.

Entre as desvantagens da programação estruturada, destacam-se os fatos que ela leva a uma menor legibilidade do código e maior dificuldade de manutenção.

Fique por dentro

Apesar da grande maioria das organizações já adotar linguagens orientadas a objetos, em muitas destas a programação ainda ocorre na forma de grandes blocos de código, ignorando as vantagens da orientação a objetos. Sendo assim, esse artigo apresenta os principais recursos da linguagem Java relativos à programação orientada a objetos, habilitando o programador a obter código reutilizável e eficiente sem complicar suas implementações, superando metodologias antigas, onde os programas são caracterizados por serem uma coleção de blocos de código.

Sendo a sua alternativa natural, a orientação a objetos organiza toda a informação pertencente a um programa na forma de conceitos, onde cada conceito é representado por uma classe, e cada classe tem atributos e métodos. Por exemplo, em uma aplicação para um mercado, o conceito **Funcionário** pode ser definido pelos atributos **nome**, **data de nascimento** e **CPF**, além dos métodos **realizarAdmissao()** e **realizarDemisso()**, que correspondem às operações ligadas ao conceito **Funcionário**.

Uma vez que nós conheçamos com abrangência os recursos da programação orientada a objetos, estaremos aptos a criar programas ganhando principalmente em eficiência e em reutilização de código. Tais recursos são importantes tanto para o desenvolvedor como para o analista de software, pois o segundo também precisa dominá-los para realizar a modelagem do sistema usando classes, atributos e métodos.

Classes e Objetos

Primeiramente, vamos partir do básico na orientação a objetos da linguagem Java, que é a definição de classes e objetos. Para alguém que está iniciando na linguagem Java, tais conceitos podem se confundir, porém classes e objetos são coisas bem distintas.

Uma classe tem a função de representar conceitos, contendo atributos e métodos. Em outras palavras, uma classe define um padrão de criação de objetos que contém os mesmos atributos e métodos. Por exemplo, uma classe **Pessoa** pode ter o atributo **nome** e o método **setNome()**. Como consequência dessa especificação, cada objeto da classe **Pessoa** (chamado também de instância) representa uma pessoa distinta no sistema que tem um valor específico definido para a variável **nome** e que pode invocar o método **setNome()**.

Considere agora outro exemplo, onde precisamos criar uma classe referente ao conceito de cliente presente no mundo real, a que chamamos de **Cliente**. Supondo ainda que existem muitas outras classes que fazem parte do sistema, como **Funcionário**, **Fornecedor** e outras, pode ser muito útil definirmos uma estrutura hierárquica para organizar as classes, de forma que cada uma seja uma especialização de outra, como na **Figura 1**.

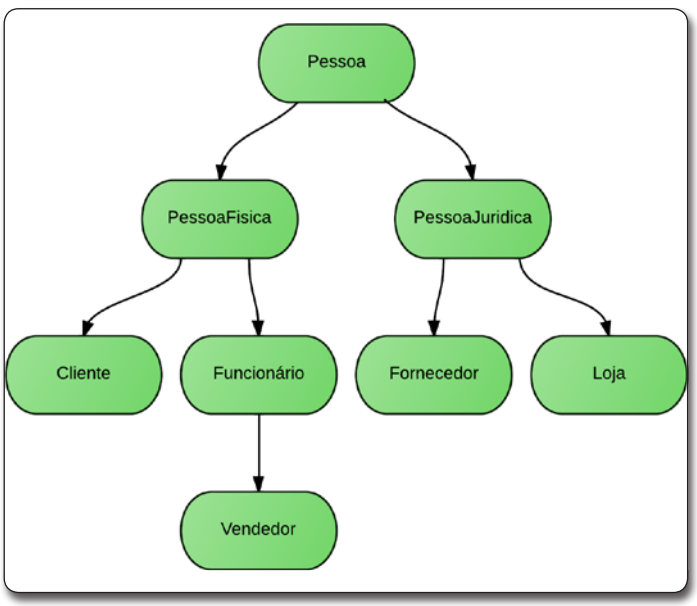

Figura 1. Diagrama de classe generalizando a modelagem de clientes

Em nossa estrutura hierárquica de classes, observamos que a classe **Pessoa** é a classe mãe de todas as outras. Abaixo dela, temos as classes **PessoaFisica** e **PessoaJuridica**, que especializam a classe **Pessoa**. Seguindo a mesma forma de organização, abaixo de **PessoaJuridica** temos as especializações **Fornecedor** e **Loja**, e abaixo de **PessoaFisica** temos as especializações **Cliente** e **Funcionário**. Por fim, considere **Vendedor** como uma especialização de **Funcionário**, pois todo vendedor é um funcionário no nosso sistema.

Como visto, o diagrama na **Figura 1** apresenta o conceito de herança (a ser definido mais adiante), um dos mecanismos básicos mais importantes oferecidos pela linguagem Java para o desenvolvimento orientado a objetos. Além deste, analisaremos ao longo do artigo outros recursos que facilitam a programação, aproveitando ao máximo a orientação a objetos.

Conceitos básicos da Orientação a Objetos

Além do mecanismo de herança, a linguagem Java oferece outros mecanismos para a programação orientada a objetos, que são os conceitos de encapsulamento e polimorfismo, definidos mais adiante. Esses conceitos formam a base da orientação a objetos, e a correta utilização deles leva a códigos simples, poderosos e eficientes, além de aplicações melhor elaboradas e uma manutenção simplificada. Vamos então conhecer como funciona o conceito de herança.

Herança entre classes

O conceito de herança é utilizado como uma forma implícita de reutilizar código, de forma que uma classe filha herda todos os atributos e métodos de uma classe mãe, o que é feito utilizando a palavra-chave **extends** na codificação da classe filha. É importante saber que atributos e métodos herdados da classe mãe são implícitos, ou seja, não são visíveis no código-fonte da classe filha, mas estão presentes e acessíveis conforme a visibilidade especificada na classe mãe.

Uma definição muito importante na linguagem Java é que **Object** é a superclasse de todas as classes. Portanto, o que ocorre quando se define uma classe sem informar a classe mãe, é que está sendo criada uma classe filha de **Object**. Na **Listagem 1** é apresentado um exemplo clássico de uma classe filha de **Object**.

Listagem 1. Declaração da classe Pessoa.

```
01. package pkg.mercado;
02.
03. public class Pessoa {
04.
05.   protected String nome;
06.   protected boolean habilitado;
07.
08.   public Pessoa(String nome, boolean habilitado) {
09.     this.nome = nome;
10.     this.habilitado = habilitado;
11.   }
12.
13.   public String getNome() {
14.     return nome;
15.   }
16.
17.   public void setNome(String nome) {
18.     this.nome = nome;
19.   }
20.
21.   public boolean isHabilitado() {
22.     return habilitado;
23.   }
24.
25.   public void setHabilitado(boolean habilitado) {
26.     this.habilitado = habilitado;
27.   }
28.
29.   public String toString() {
30.     return "Pessoa, nome = [" + nome + "] habilitado = [" + habilitado + "]";
31.   }
32. }
```

Note que nas linhas 5 e 6 a classe **Pessoa** contém dois atributos: **nome** e **habilitado**. Como consequência, cada objeto dessa classe define valores próprios para ambos os atributos, o que é feito através do construtor, declarado nas linhas 8 a 11. O construtor apenas recebe os valores destes atributos e os armazena na instância usando o comando **this**.

A mesma classe poderia ser declarada de forma diferente, porém com exatamente a mesma funcionalidade. Veja um exemplo disto na **Listagem 2**.

Listagem 2. Declaração alternativa da classe Pessoa.

```
01. package pkg.mercado;
02.
03. public class Pessoa2 extends Object {
04.
05.  protected String nome;
06.  protected boolean habilitado;
07.
08.  public Pessoa2(String nome, boolean habilitado) {
09.   super();
10.   this.nome = nome;
11.   this.habilitado = habilitado;
12.  }
13.
14.  public String getNome() {
15.   return nome;
16.  }
17.
18.  public void setNome(String nome) {
19.   this.nome = nome;
20.  }
21.
22.  public boolean isHabilitado() {
23.   return habilitado;
24.  }
25.
26.  public void setHabilitado(boolean habilitado) {
27.   this.habilitado = habilitado;
28.  }
29.
30.  public String toString() {
31.   return "Pessoa, nome = [" + nome + "] habilitado = [" + habilitado + "]";
32.  }
33. }
```

Observe agora que a declaração da classe (linha 3) usa o comando **extends**. Ele serve para definir a relação de herança entre duas classes. Nesse exemplo, **extends** indica que a classe **Pessoa** herda todos os atributos e métodos da classe **Object**. Como mencionado anteriormente, **Object** já é, por *default*, a classe mãe de todas as classes e, portanto, essa relação de herança pode ser omitida (como na declaração mostrada na **Listagem 1**).

Uma exceção à regra da herança é que não se herdam construtores. Sendo assim, deve-se redef inir os construtores na declaração da classe filha. Uma observação muito importante é que o primeiro comando do construtor da classe filha deve ser, obrigatoriamente, a chamada ao construtor da classe mãe, usando **super**, como na linha 9 da **Listagem 2**.

Entretanto, pode-se omitir a chamada ao construtor da classe mãe, fazendo com que o Java implicitamente invoque o construtor *default* desta, que é o construtor sem parâmetros. Porém, nesse caso, se o construtor default não estiver definido na classe mãe, ocorrerá um erro de compilação.

Agora considere o exemplo da classe **PessoaFisica**, que especializa **Pessoa**. Para definir a relação de herança entre ambas, deve-se usar **extends Pessoa** na declaração de **PessoaFisica**. Como construtores não são herdados, deve-se definir explicitamente o construtor da classe **PessoaFisica**, porém dessa vez, sem usar o "truque" da chamada implícita ao construtor *default* da superclasse, visto que **Pessoa** não contém o construtor *default*. O código da classe **PessoaFisica** é apresentado na **Listagem 3**.

Listagem 3. Declaração da classe PessoaFisica.

```
01. package pkg.mercado;
02.
03. public class PessoaFisica extends Pessoa {
04.
05.  protected String dataNascimento;
06.  protected long CPF;
07.
08.  public PessoaFisica(String nome, boolean habilitado, String dataNascimento,
09.    long CPF) {
10.
11.   super(nome, habilitado);
12.   this.dataNascimento = dataNascimento;
13.   this.CPF = CPF;
14.  }
15.
16.  public String getDataNascimento() {
17.   return dataNascimento;
18.  }
19.
20.  public void setDataNascimento(String dataNascimento) {
21.   this.dataNascimento = dataNascimento;
22.  }
23.
24.  public long getCPF() {
25.   return CPF;
26.  }
27.
28.  public void setCPF(long CPF) {
29.   this.CPF = CPF;
30.  }
31.
32.  public String toString() {
33.   return "Pessoa Fisica, nome = [" + nome + "] habilitado = [" + habilitado
34.     + "] data nascimento = [" + dataNascimento + "] CPF = [" + CPF
35.     + "]";
36.  }
37. }
```

Observe que o comando **extends** foi usado na linha 3 para definir a relação de herança entre **Pessoa** e **PessoaFisica**.

Dessa forma, **PessoaFisica** herda todos os métodos e atributos da classe **Pessoa**, e com isso, tem os atributos **nome**, **habilitado**, **dataNascimento** e **CPF**, onde os dois primeiros são herdados da classe **Pessoa** e os dois últimos são próprios de **PessoaFisica**.

Além disso, são herdados todos os métodos da superclasse, fazendo com que **PessoaFisica** tenha os métodos **getNome()**, **setNome()**, **isHabilitado()** e **setHabilitado()**, herdados de **Pessoa**, e os métodos **getDataNascimento()**, **setDataNascimento()** e **getCPF()**, definidos na própria classe.

O construtor de **PessoaFisica** é definido nas linhas 8 a 14, recebendo como parâmetros: **nome**, **habilitado**, **dataNascimento** e **CPF**. E como seu primeiro comando, na linha 11, ocorre uma chamada ao construtor da superclasse:

```
super(nome, habilitado);
```

O uso do comando **super** realiza a chamada ao construtor da superclasse (definido na linha 8 da **Listagem 1**), informando os parâmetros **nome** e **habilitado**. Em seguida, são definidos os valores dos atributos **dataNascimento** e **CPF** em **PessoaFisica** (linhas 12 e 13 da **Listagem 3**). O restante da **Listagem 3** apresenta os *getters* e os *setters* dos atributos de **PessoaFisica**. Por fim, o método **toString()** é declarado na linha 32, e retorna uma descrição textual do objeto.

Seguindo a mesma lógica, na **Listagem 4** é declarada a classe **PessoaJuridica**.

Listagem 4. Declaração da classe PessoaJuridica.

```
01. package pkg.mercado;
02.
03. public class PessoaJuridica extends Pessoa {
04.
05.  protected long CNPJ;
06.
07.  public PessoaJuridica(String nome, boolean habilitado, long CNPJ) {
08.   super(nome, habilitado);
09.   this.CNPJ = CNPJ;
10.  }
11.
12.  public long getCNPJ() {
13.   return CNPJ;
14.  }
15.
16.  public void setCNPJ(long CNPJ) {
17.   this.CNPJ = CNPJ;
18.  }
19.  public String toString() {
20.   return "Pessoa Juridica, nome = [" + nome + "] habilitado = [" + habilitado
21.    + "] CNPJ = [" + CNPJ + "]";
22.  }
23. }
```

O mesmo padrão se repete em **PessoaJuridica**. Ela é declarada como filha de **Pessoa** usando o comando **extends** (linha 3). Além dos atributos herdados, ela contém o atributo **CNPJ** (linha 5). O seu construtor é declarado na linha 7, e recebe os parâmetros **nome**, **habilitado** e **CNPJ**, os quais, os dois primeiros são repassados à classe mãe usando o comando **super** para chamar o construtor da superclasse. Na linha 9, o valor do atributo **CNPJ** é definido, e o restante da classe define o *getter* e o *setter* para o atributo **CNPJ**.

Agora que temos diversas classes, vamos usá-las em um exemplo para criar objetos e escrever na tela uma descrição para cada um usando o método **toString()**.

A **Listagem 5** apresenta um exemplo em que são criados dois objetos da classe **Pessoa** (linhas 5 e 6), dois objetos da classe **PessoaFisica** (linhas 7 e 8) e dois objetos da classe **PessoaJuridica** (linhas 9 e 10). Observe que, na criação de cada objeto, foram informados todos os parâmetros necessários conforme declarado no construtor de cada classe. Em seguida, é impressa uma descrição textual de cada objeto criado (linhas 11 a 16) usando o método **toString()**. Como resultado da execução desse código, é impresso na tela do console o conteúdo da **Listagem 6**.

Listagem 5. Exemplo de criação de objetos.

```
01. package pkg.mercado;
02.
03. public class TestePessoas {
04.  public static void main (String[] args) {
05.  Pessoa p1 = new Pessoa("Carlos", true);
06.  Pessoa p2 = new Pessoa("Carina", true);
07.  PessoaFisica p3 = new PessoaFisica("Elias", true, "01/01/1980", 1111111);
08.  PessoaFisica p4 = new PessoaFisica("Carla", true, "01/01/1990", 2222222);
09.  PessoaJuridica p5 = new PessoaJuridica("Loja Teste1", false, 3333333);
10.  PessoaJuridica p6 = new PessoaJuridica("Loja Teste2", true, 4444444);
11.  System.out.println(p1.toString());
12.  System.out.println(p2.toString());
13.  System.out.println(p3.toString());
14.  System.out.println(p4.toString());
15.  System.out.println(p5.toString());
16.  System.out.println(p6.toString());
17.  }
18. }
```

Listagem 6. Resultado da execução.

```
Pessoa, nome = [Carlos] habilitado = [true]
Pessoa, nome = [Carina] habilitado = [true]
Pessoa Fisica, nome = [Elias] habilitado = [true] data nascimento = [01/01/1980]
CPF = [1111111]
Pessoa Fisica, nome = [Carla] habilitado = [true] data nascimento = [01/01/1990]
CPF = [2222222]
Pessoa Juridica, nome = [Loja Teste1] habilitado = [false] CNPJ = [3333333]
Pessoa Juridica, nome = [Loja Teste2] habilitado = [true] CNPJ = [4444444]
```

Pode-se notar que o método **toString()** de cada objeto escreveu uma descrição textual do mesmo, identificando se é pessoa, pessoa física ou pessoa jurídica. Portanto, pode-se verificar que o esquema apresentado para definir pessoa em termos de pessoa jurídica ou pessoa física funcionou adequadamente usando herança para reutilizar o código da classe **Pessoa** nas classes filhas **PessoaFisica** e **PessoaJuridica**.

Herança usando interfaces

Em Java, para não aumentar a complexidade do mecanismo de herança, não temos herança múltipla, ou seja, uma classe filha pode ter somente uma classe mãe, ao contrário de outras linguagens, como C++. A fim de prover uma solução alternativa, Java disponibiliza interfaces.

Uma interface tem a função de servir como se fosse uma classe mãe adicional, porém é regra que uma interface não pode conter a implementação de seus métodos (somente pode conter as assinaturas dos mesmos), pois a sua implementação deve ser definida por uma classe filha (ou seja, uma classe que implementa a interface).

Deste modo, uma interface é composta de assinaturas de métodos (somente o nome, retorno e parâmetros) e constantes (atributos

que sejam **final static**). Para fazer com que uma classe implemente uma interface é usado o comando **implements** (da mesma forma que se usa **extends**). Como resultado, a classe filha deve conter a implementação de todos os métodos definidos na interface (usando a mesma assinatura). Na **Listagem 7** é apresentado um exemplo de interface.

Observa-se que a interface **Investidor** define dois métodos, **realizarInvestimento()** e **terminarInvestimento()**, que devem ser implementados em qualquer classe que a implementar. Como pode-se verificar nas linhas 5 e 6, esses métodos contêm apenas um ponto-e-vírgula no lugar do corpo. Na **Listagem 8** é declarada a classe **Cliente** que implementa **Investidor**.

Listagem 7. Interface Investidor.

```
01. package pkg.mercado;
02.
03. public interface Investidor {
04.
05.  void realizarInvestimento();
06.  void terminarInvestimento();
07. }
```

Listagem 8. Código da classe Cliente, que implementa Investidor.

```
01. package pkg.mercado;
02.
03. public class Cliente extends PessoaFisica implements Investidor {
04.
05.  public Cliente(String nome, boolean habilitado, String dataNascimento,
     long CPF) {
06.   super(nome, habilitado, dataNascimento, CPF);
07.  }
08.
09.  public void realizarInvestimento() {
10.   System.out.println("Cliente " + nome + " realizou investimento");
11.  }
12.
13.  public void terminarInvestimento() {
14.   System.out.println("Cliente " + nome + " terminou investimento");
15.  }
16.
17.  public String toString() {
18.   return "Cliente, nome = [" + nome + "] habilitado = [" + habilitado
19.    + "] data nascimento = [" + dataNascimento + "] CPF = [" + CPF
20.    + "]";
21.  }
22. }
```

Note que a classe **Cliente** define a implementação dos métodos **realizarInvestimento()** e **terminarInvestimento()** nas linhas 9 a 15.

Ainda na linguagem Java existe a possibilidade de uma interface ser uma especialização de outra interface, o que é feito pela utilização do comando **extends**, como na **Listagem 9**, onde é definida a interface **Investidor2**.

Observe que a interface **Investidor2** define o método **emitirRelatorio()**, além dos métodos herdados de **Investidor**. Com o intuito de criar uma classe que implemente a interface **Investidor2**, é primeiro exposta a classe **Funcionario** na **Listagem 10**, filha de **PessoaFisica**, e na **Listagem 11** é apresentada a classe **Vendedor**, filha de **Funcionario** e que implementa **Investidor2**, uma das interfaces de nosso exemplo e que contém três métodos.

Listagem 9. Código da interface Investidor2.

```
01. package pkg.mercado;
02.
03. public interface Investidor2 extends Investidor {
04.  void emitirRelatorio();
05. }
```

Listagem 10. Código da classe Funcionario.

```
01. package pkg.mercado;
02.
03. public class Funcionario extends PessoaFisica {
04.
05.  protected String dataAdmissao;
06.  protected String dataDemissao;
07.  protected String localTrabalho;
08.
09.  public Funcionario(String nome, boolean habilitado, String dataNascimento,
10.     long CPF, String dataAdmissao, String dataDemissao,
11.     String localTrabalho) {
12.
13.   super(nome, habilitado, dataNascimento, CPF);
14.   this.dataAdmissao = dataAdmissao;
15.   this.dataDemissao = dataDemissao;
16.   this.localTrabalho = localTrabalho;
17.  }
18.
19.  public String getDataAdmissao() {
20.   return dataAdmissao;
21.  }
22.
23.  public void setDataAdmissao(String dataAdmissao) {
24.   this.dataAdmissao = dataAdmissao;
25.  }
26.
27.  public String getDataDemissao() {
28.   return dataDemissao;
29.  }
30.
31.  public void setDataDemissao(String dataDemissao) {
32.   this.dataDemissao = dataDemissao;
33.  }
34.
35.  public String getLocalTrabalho() {
36.   return localTrabalho;
37.  }
38.
39.  public void setLocalTrabalho(String localTrabalho) {
40.   this.localTrabalho = localTrabalho;
41.  }
42. }
```

Note que a classe **Vendedor** implementa o método definido na interface **Investidor2**, que (**emitirRelatorio()**) e os métodos definidos em **Investidor**, que são **realizarInvestimento()** e **terminarInvestimento()**. Para exemplificar a utilização das interfaces apresentadas, na **Listagem 12** é introduzido um teste que cria diversos objetos que são investidores.

Esse teste cria dois clientes e dois vendedores (linhas 5 a 9) e para cada um desses objetos chama o método **realizarInvestimento()** e o método **terminarInvestimento()**, independentemente se o objeto que invocou cada um desses métodos implementa **Investidor** ou

Investidor2, pois ambas interfaces definem ambos os métodos. O resultado da execução da classe **TesteInterfaces** pode ser visto na **Listagem 13**.

Listagem 11. Código da classe Vendedor.

```
01. package pkg.mercado;
02.
03. public class Vendedor extends Funcionario implements Investidor2 {
04.
05.   protected Funcionario supervisor;
06.
07.   public Vendedor(String nome, boolean habilitado, String dataNascimento,
      long CPF,
08.   String dataAdmissao, String dataDemissao, String localTrabalho,
09.   Funcionario supervisor) {
10.
11.     super(nome, habilitado, dataNascimento, CPF, dataAdmissao, dataDemissao,
12.           localTrabalho);
13.
14.     this.supervisor = supervisor;
15.   }
16.
17.   public Funcionario getSupervisor() {
18.     return supervisor;
19.   }
20.
21.   public void setSupervisor(Funcionario supervisor) {
22.     this.supervisor = supervisor;
23.   }
24.
25.   public void realizarInvestimento() {
26.     System.out.println("Vendedor " + nome + " realizou investimento");
27.   }
28.
29.   public void terminarInvestimento() {
30.     System.out.println("Vendedor " + nome + " terminou investimento");
31.   }
32.
33.   public void emitirRelatorio() {
34.     System.out.println("Vendedor " + nome + " emitiu relatório");
35.   }
36. }
```

Listagem 12. Teste para as interfaces.

```
01. package pkg.mercado;
02.
03. public class TesteInterfaces {
04.   public static void main (String[] args) {
05.     Cliente  c1 = new Cliente("Cláudio", true, "01/01/1980", 1111111);
06.     Cliente  c2 = new Cliente("Amanda", false, "01/01/1985", 2222222);
07.     Funcionario supervisor= new Funcionario("Alberto", true, "01/01/1986",
        9999999, "01/01/1978", null, "Matriz");
08.     Vendedor c3 = new Vendedor("Juliano", true, "01/01/1985", 3333333,
        "01/01/2005", null, "Matriz", supervisor);
09.     Vendedor c4 = new Vendedor("Juliana", true, "01/01/1980", 4444444,
        "01/02/2005", null, "Matriz", supervisor);
10.     c1.realizarInvestimento();
11.     c1.terminarInvestimento();
12.     c2.realizarInvestimento();
13.     c2.terminarInvestimento();
14.     c3.realizarInvestimento();
15.     c3.terminarInvestimento();
16.     c4.realizarInvestimento();
17.     c4.terminarInvestimento();
18.   }
19. }
```

Listagem 13. Resultado da execução de TesteInterfaces.

```
Cliente Cláudio realizou investimento
Cliente Cláudio terminou investimento
Cliente Amanda realizou investimento
Cliente Amanda terminou investimento
Vendedor Juliano realizou investimento
Vendedor Juliano terminou investimento
Vendedor Juliana realizou investimento
Vendedor Juliana terminou investimento
```

Para finalizar nosso estudo sobre interfaces, ao contrário da herança entre classes, uma classe pode implementar quantas interfaces forem necessárias, desde que codifique todos os métodos de todas elas.

Influência dos modificadores de acesso na herança

Cada modificador de acesso (ou visibilidade) usado em métodos ou atributos na linguagem Java tem também um papel específico em relação à herança. São eles:

- **private:** um método ou atributo **private** somente pode ser acessado dentro da própria classe em que ele é declarado e não é acessível em suas subclasses;
- **protected:** um método ou objeto **protected** somente pode ser acessado dentro do mesmo pacote em que é declarada sua classe e é acessível nas suas subclasses;
- **friendly:** é quando não se declara nenhum modificador para o atributo ou método. É a mesma coisa que usar **protected**, porém um método ou atributo **friendly** não é acessível nas subclasses;
- **public:** o método ou atributo é visível para todas as classes e subclasses.

É importante saber que interfaces podem somente conter métodos públicos. Pode-se inclusive omitir o modificador de acesso numa interface, mas seus métodos continuarão sendo públicos.

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/

Como configurar um cluster com Tomcat e NGinx

Aprenda a criar um cluster com balanceamento de carga utilizando Tomcat, NGinx e uma aplicação Java

Fique por dentro

Este artigo aborda diversos conceitos relacionados à computação distribuída, oferecendo uma introdução aos principais tópicos da disciplina como base para a aplicação prática de um cluster, que será implementado com o servidor HTTP NGinx como balanceador de carga para três instâncias do Tomcat. Além disso, será feito o deploy de uma aplicação Java que faz uso de sessão nesse ambiente distribuído a fim de verificar o funcionamento da replicação de sessão em um ambiente clusterizado.

Com a popularização da computação em nuvem, que tem como base abordagens de computação distribuída, o conhecimento do funcionamento de um cluster é muito importante para desenvolver aplicações distribuídas de sucesso

A computação distribuída assumiu um papel muito importante como paradigma computacional para sistemas distribuídos. O aumento da prestação de serviços digitais em rede, a necessidade por mais poder de processamento, menor tempo de resposta, mais confiabilidade e o aumento do número de usuários são características que induziram e popularizaram a adoção de sistemas distribuídos, até mesmo em casos onde seu uso não se faz necessário, tamanha a popularização dessa arquitetura computacional.

A computação distribuída consiste em executar aplicações que cooperam entre si em máquinas distintas interligadas por uma rede de computadores, objetivando a melhoria do desempenho por meio de complexas infraestruturas computacionais.

Atingir o objetivo principal da computação distribuída implica o projeto de uma arquitetura específica para este fim. Geralmente complexas e bem elaboradas, as arquiteturas distribuídas envolvem muitos aspectos de hardware e software.

Em relação ao hardware, há preocupações sobre todos os dispositivos utilizados na abordagem, inclusive, e mais importante, sobre as características da rede, geralmente configuradas como clusters ou grids computacionais.

Cluster é um agrupamento físico de computadores considerando-se um pequeno limite geográfico. Podem ser formados por simples PCs conectados em rede que passam a atender às solicitações de terceiros como recursos computacionais. Em um cluster, cada máquina é chamada de nó e, geralmente, existe um nó mestre (*master*) que gerencia e divide as tarefas computacionais entre os demais nós, conhecidos como nós escravos (*slaves*). Uma das características mais importantes de um cluster é a tolerância a falhas, pois o cluster mantém seu funcionamento mesmo com a paralisação de alguns nós.

Grid computacional, por outro lado, é um agrupamento de clusters considerando-se uma grande área geográfica e possui as mesmas características dos clusters, embora o principal atributo de um grid seja permitir o compartilhamento de recursos (memória, processamento ou banda de rede), como, por exemplo, a plataforma Amazon Web Services o faz.

Em relação ao software, a maior preocupação é com o gerenciamento das tarefas sendo executadas em diferentes nós que compõem o sistema distribuído. Questões como balanceamento de carga, controle de sessões e compartilhamento e acesso concorrente aos dados são resolvidas por meio de software.

A plataforma Java fornece muitos mecanismos e APIs que auxiliam na criação de sistemas distribuídos. A lista de tecnologias no ecossistema Java com o propósito de resolver questões em ambientes de aplicações distribuídas é extensa e incluem RMI, JAX-RPC (SOAP) e EJBs.

Do ponto de vista da aplicação, um dos principais desafios em ambientes de computação distribuída é como as sessões são gerenciadas, pois cada requisição em um ambiente distribuído é geralmente redirecionada pelo load balancer para um nó de processamento diferente.

As próximas sessões sintetizarão a teoria e a prática acerca dos principais aspectos da computação distribuída com a tec-

nologia Java. Será mostrado como configurar um cluster com três Tomcats, o servidor HTTP NGinx como load balancer e uma aplicação web que faz uso de sessão para ilustrar o comportamento do compartilhamento de sessões em um ambiente distribuído. A **Figura 1** mostra a arquitetura do cluster utilizada neste artigo.

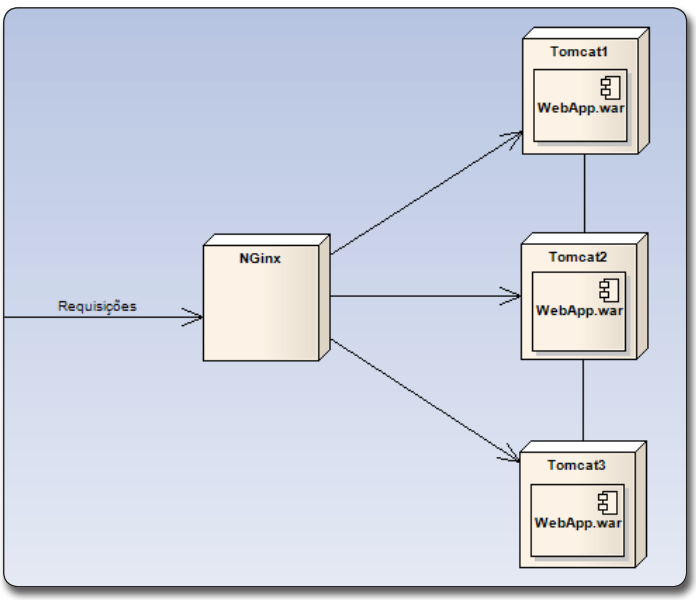

Figura 1. Arquitetura do cluster. NGinx como load balancer e três instâncias Tomcat

Java e clustering

Ao longo das últimas duas décadas Java foi a plataforma escolhida por muitas organizações para implementar aplicações críticas na web. Alta escalabilidade, performance e tolerância a falhas são requisitos imprescindíveis em uma aplicação crítica e esses requisitos são inerentes a arquiteturas distribuídas como cluster. A plataforma Java possui forte ligação com as técnicas de clustering. Os containers Java EE mais importantes, por exemplo, possuem características que incentivam e favorecem o uso de clusters.

A definição de cluster pelos fornecedores de servidores Java EE é a seguinte: um grupo de máquinas trabalhando juntas e oferecendo de forma transparente serviços como JNDI, EJB, JSP, HttpSession, etc. É uma definição vaga, mas proposital, já que cada fornecedor implementa clustering de uma forma diferente dos outros.

Apesar dessas diferenças, o objetivo é o mesmo, viabilizar um ambiente de alto desempenho e confiabilidade, os conceitos permanecem inalterados. Ao longo do artigo, com detalhes de configuração do Tomcat, será possível perceber qual o nível de diferenças entre os servidores de diferentes fornecedores.

Antes de continuar com o aprofundamento no tema, no entanto, convém entender alguns dos conceitos mais importantes relacionados à técnica de clustering: escalabilidade, disponibilidade, balanceamento de carga, tolerância a falhas, recuperação e idempotência.

Escalabilidade

Escalabilidade é a capacidade de atender a um número de usuários que cresce rapidamente. A forma mais comum de aumentar a escalabilidade é adicionar recursos como processador e memória. Ambientes distribuídos facilitam a escalabilidade ao permitir que vários servidores em um grupo dividam tarefas de processamento e atuem como um único servidor lógico.

Disponibilidade

Disponibilidade está relacionada a um determinado período de tempo em que o sistema permite atendimento às requisições. A disponibilidade pode ser bastante elevada em ambientes distribuídos devido ao número de servidores redundantes no cluster. Quando um servidor falha, outro servidor é acionado para atender a requisição.

Balanceamento de carga

Balanceamento de carga (*load balancing*) é uma das tecnologias mais importantes por trás da clusterização, pois possibilita a distribuição do trabalho computacional entre diferentes computadores. Um load balancer pode ser implementado de forma simples, como um servlet, ou ser um mecanismo parte de um servidor HTTP com recursos avançados de distribuição de tráfego, como os mecanismos do NGinx e Tomcat para balanceamento de carga.

Tolerância a falhas

Alta disponibilidade de dados não significa que os dados disponíveis sejam corretos. Em um ambiente de alta disponibilidade, em meio a diferentes requisições, em caso de falha de um servidor, o serviço continua disponível. Nesse cenário, uma requisição que estava sendo processada por um servidor que falhou pode não conter os dados corretos.

Grande parte das falhas em ambientes distribuídos são ocasionadas por inconsistência nos dados, sendo necessário diversos mecanismos para lidar com esse problema e aumentar a tolerância a falhas da aplicação.

Recuperação (Failover)

Intimamente relacionada à tolerância a falhas, a recuperação é a capacidade do sistema continuar o processamento quando o nó original que estava respondendo a solicitação falha. A recuperação é feita de maneira transparente quando ocorre um problema. A requisição é redirecionada para outro servidor que pode recuperar os dados da solicitação anterior ou recomeçar o processamento.

Idempotência

Métodos idempotentes são métodos que podem ser chamados diversas vezes com os mesmos argumentos e resultarão sempre nos mesmos resultados, sem alterar o estado do sistema. Um exemplo de método idempotente pode ser o **getUsername()** em uma sessão.

A idempotência é um valioso atributo em um ambiente distribuído, promovendo a consistência ao evitar a alteração de estado

do sistema a cada chamada a um método proveniente de um servidor diferente.

Em linhas gerais, clustering em Java é definido de forma básica como balanceamento de carga e recuperação, mas os atributos listados anteriormente são imprescindíveis para atingir um nível de qualidade e operação satisfatório.

NGinx como balanceador de carga

Balanceamento de carga é uma técnica que permite a distribuição do trabalho computacional entre diferentes computadores, redes, discos rígidos ou outros recursos computacionais com o objetivo de aumentar a eficiência na utilização dos mesmos e diminuir a sobrecarga. O balanceamento de carga em um cluster distribui as requisições dos clientes de forma uniforme entre todas as máquinas que compõem o cluster, garantindo assim o uso equilibrado de todos os recursos.

NGinx é um servidor HTTP e proxy reverso muito popular para realizar o load balancer devido a sua performance. Proxy reverso, por sua vez, é um servidor que fica a frente dos servidores web, recebe todas as requisições e decide por tratar ele mesmo a requisição ou encaminhá-la para um dos servidores web, podendo assim fazer o balanceamento de carga do cluster.

O load balancer é responsável por:

- **Algoritmos de balanceamento de carga:** algoritmo responsável por fazer com que a carga de trabalho seja distribuída de maneira igualitária entre todos os servidores do cluster. A maioria dos algoritmos são baseados no número de requisições enviadas para cada servidor, embora alguns load balancers possuam algoritmos sofisticados que verificam a real carga de trabalho do servidor antes de encaminhar a requisição para o mesmo. Além disso, alguns algoritmos fazem o controle de sessão, para que as requisições de um determinado navegador sejam redirecionadas sempre para o mesmo servidor;
- **Checagem dos nós do cluster:** o load balancer precisa monitorar todos os servidores que compõem o cluster e quando identificar uma falha não deve enviar requisições para o servidor que falhou. É responsabilidade do load balancer manter a verificação dos servidores que falharam até que voltem a funcionar. Quando o servidor voltar a funcionar, o load balancer deve retomar o envio requisições para o mesmo.

Configuração do cluster

O cluster ilustrado na **Figura 1**, formado por um proxy reverso e três Tomcats, será configurado em uma mesma máquina a fim de facilitar a instalação e focar a abordagem conceitual do artigo, cujo objetivo é mostrar o comportamento de uma aplicação que faz uso de sessão em um ambiente distribuído.

Os próximos tópicos mostrarão como configurar o ambiente, instalando os servidores Tomcat e configurando-os para que possam rodar juntos em um mesmo computador de forma a simular um cluster em localhost.

Configuração do Tomcat

Para rodar em uma mesma máquina, cada instância do Tomcat deve ter números de porta HTTP, porta AJP e server port diferentes. Essa alteração é feita no arquivo *server.xml*, localizado em *CATALINA_HOME/conf*.

A porta HTTP é alterada por meio da configuração de um conector (elemento **<Connector>**) no arquivo *server.xml* cujo atributo **protocol** seja **HTTP**. O padrão da porta HTTP é 8080, conforme a seguinte configuração:

```
<Connector port="8080" protocol="HTTP/1.1" connectionTimeout="20000"
redirectPort="8443" />
```

Como serão necessários três Tomcats, podem-se utilizar os valores 8081, 8082 e 8083 no atributo **port** de cada um dos Tomcat instalados.

A porta AJP também é configurada por meio de um conector no arquivo *server.xml* cujo atributo **protocol** seja **AJP**. A seguir é mostrada onde a porta AJP precisa ser alterada:

```
<Connector port="8009" protocol="AJP/1.3" redirectPort="8443" />
```

A **server port** também deve ser única para cada instância do Tomcat e deve ser especificada por meio da tag **Server** no arquivo *server.xml*, conforme a seguinte configuração:

```
<Server port="8005" shutdown="SHUTDOWN">
```

Tendo realizado essas configurações torna-se possível rodar as três instâncias do Tomcat na mesma máquina sem nenhum conflito de portas. O próximo passo para a configuração do cluster é a configuração do servidor proxy reverso.

Configuração do NGinx

O NGinx faz o papel de proxy reverso, conforme ilustrado na **Figura 1**, redirecionando as requisições para os diferentes servidores que compõem o cluster de forma balanceada. Sua instalação apresenta particularidades para cada sistema operacional e os detalhes de instalação podem ser conferidos no manual de instalação disponível na seção **Links**. Após instalar o NGinx, o próximo passo é configurá-lo como load balancer indicando os servidores que farão parte do cluster.

A configuração de um cluster no NGinx é feita por meio do arquivo *nginx.conf*, localizado em *NGINX_HOME/conf/nginx.conf*. Nesse arquivo, dentro do elemento **http**, é preciso criar um elemento **upstream** contendo os endereços dos servidores que vão compor o cluster. A **Listagem 1** mostra a configuração de um cluster chamado myCloud. Esse cluster é formado pelos três servidores Tomcat configurados no tópico anterior. Note como todos os endereços se referem à localhost (*127.0.0.1*) e mudam apenas a porta.

O upstream myCloud especificado no proxy reverso diz quais são os servidores que fazem parte do cluster. No entanto, quando o

NGinx recebe uma requisição, ele ainda não sabe o que fazer com ela. Sendo assim, o proxy reverso precisa ser configurado para que ele saiba o que fazer com a requisição recebida. Para esse caso é necessário dizer ao NGinx para redirecionar as requisições ao cluster myCloud. A **Listagem 2** mostra como configurar o redirecionamento das requisições à raiz do servidor para myCloud.

Esta configuração é feita por meio do elemento **location** que fica dentro do **server** em *nginx.conf*. O elemento **location** é seguido por uma barra invertida indicando a raiz e configurado com o subelemento **proxy_pass**, que indica o cluster para o qual o NGinx deve redirecionar as requisições à raiz. Deste modo, sempre que houver uma requisição à raiz do NGinx, ele irá redirecioná-la para myCloud.

Listagem 1. nginx.conf – configuração de um cluster no NGinx.

```
upstream myCloud{
   server 127.0.0.1:8081;
   server 127.0.0.1:8082;
   server 127.0.0.1:8083;
}
```

Listagem 2. Configuração no NGinx para redirecionar as requisições ao cluster.

```
location / {
   proxy_pass http://myCloud;
}
```

Após essas configurações, o ambiente ilustrado na **Figura 1** está pronto. Pode-se então iniciar todos os Tomcats, que responderão nas portas 8081, 8082 e 8083 e o NGinx, que por padrão responde na porta 80. Neste momento, qualquer requisição a localhost será redirecionada pelo load balancer para alguma das instâncias do Tomcat configuradas no cluster.

Com esta etapa concluída, as próximas seções apresentam os desafios de se desenvolver em plataformas distribuídas, as características das aplicações distribuídas e como a plataforma Java lida com essas questões.

Aplicações distribuídas

As aplicações distribuídas inserem novos desafios e preocupações em comparação a uma aplicação não distribuída. Questões como gerenciamento de sessões e acesso concorrente a dados são frequentes ao lidar com sistemas distribuídos.

Um dos principais desafios para uma aplicação distribuída é o acesso aos dados a partir dos diferentes servidores do cluster. A distribuição dos dados pelo cluster fica a cargo dos servidores que o compõem, que podem implementar essa característica de diferentes maneiras e de forma transparente para o desenvolvedor. Entretanto, é preciso compreender o que acontece com a aplicação em um ambiente distribuído de forma a escolher a melhor implementação e configuração adequada de acordo com os requisitos não funcionais do sistema.

O que acontece, por exemplo, com uma aplicação implantada em um cluster que precisa, em diferentes requisições, utilizar dados que estão na sessão? As **Listagens 3** e **4** apresentam dois arquivos JSP de uma aplicação web que usam sessão. O arquivo *put.jsp* coloca informações na sessão e estas informações são utilizadas pelo arquivo *get.jsp*. Esses dois arquivos ilustrarão o que acontece quando a aplicação lê e escreve dados na sessão em diferentes requisições.

Para implantar essa aplicação no cluster, deve-se fazer o deploy do arquivo WAR em todas as instâncias do Tomcat configuradas no load balancer. O acesso à página *put.jsp* coloca na sessão um atributo chamado **sessionAtribute** com o valor "Hello!", enquanto que o acesso à página *get.jsp* recupera da sessão o mesmo atributo, **sessionAtribute**, no entanto este é recuperado com o valor **null**.

Isso acontece devido à sessão não estar sendo compartilhada entre os nós do cluster, pois o mecanismo de replicação não está configurado para este cluster. Segue o que acontece quando a replicação não está habilitada: a primeira requisição ao arquivo *put.jsp* é redirecionada pelo load balancer para a instância Tomcat1, o atributo é colocado na sessão e mantida no Tomcat1. Já a segunda requisição, ao arquivo *get.jsp*, é redirecionada pelo load balancer para a instância Tomcat2, que não tem sessão e por isso a tentativa de recuperar alguma informação da mesma retorna **null**. A solução para esse problema em um ambiente distribuído é a replicação de sessão.

Listagem 3. Arquivo put.jsp, coloca uma informação na sessão.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
  pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
 "http://www.w3.org/TR/html4/loose.dtd">
<html>
 <head>
   <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
   <title>Cluster test</title>
 </head>
 <body>
   <%
    session.setAttribute("sessionAtribute", "Hello!");
   %>
   Atribute added: <%= session.getAttribute("sessionAtribute") %>
 </body>
</html>
```

Listagem 4. Arquivo get.jsp, utiliza uma informação na sessão.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
  pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
 "http://www.w3.org/TR/html4/loose.dtd">
<html>
 <head>
   <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
   <title>Cluster test</title>
 </head>
 <body>
   Atribute retrieved: <%= session.getAttribute("sessionAtribute") %>
 </body>
</html>
```

Replicação de sessão

A replicação de sessão é um mecanismo fornecido pelo servidor de aplicação para transferir dados, objetos ou eventos entre os servidores de um cluster. Com a replicação de sessão, todas as sessões em uma instância de um servidor de aplicação são replicadas para as demais, de forma que seja qual for o servidor utilizado pelo load balancer para o redirecionamento, o cliente encontrará sua sessão nesse servidor. A replicação de sessão também possibilita a alta disponibilidade, pois se um nó do cluster cai destruindo todas as sessões do mesmo, as sessões do cluster que caiu estarão disponíveis para os clientes nos outros nós.

Este mecanismo é viabilizado por meio de algoritmos. Atualmente, os diferentes servidores de aplicação implementam os algoritmos mais convenientes para a sua arquitetura, podendo ser alterados de acordo com as necessidades de performance e do tamanho do cluster. O padrão do Tomcat é o algoritmo "all-to-all replication", que replica as sessões por todos os nós do cluster, inclusive em nós onde a aplicação não foi implantada. O algoritmo "all-to-all" é eficiente em clusters com poucos nós e não é recomendado para clusters muito grandes. O **BOX 1** apresenta mais detalhes sobre os tipos de algoritmos de replicação.

BOX 1. Algoritmos de replicação de sessão

Algoritmos de replicação de dados são fundamentais para a implementação de clusters em arquiteturas distribuídas, pois garante a disseminação dos dados pelos diferentes nós que compõem o cluster. Os algoritmos de replicação são categorizados em dois grupos, "optimistic replication" e "pessimistic replication".

Técnicas otimistas ("optimistic replication") consistem em permitir divergências nas réplicas, fazendo a convergência depois de algum período de tempo ou apenas quando necessária. No momento da sincronização dos dados, pode haver inconsistências e, caso a aplicação não esteja preparada para lidar com as inconsistências, a intervenção humana é necessária.

As técnicas pessimistas ("pessimistic replication") tentam garantir a igualdade de todas as réplicas desde o início. O algoritmo padrão do Tomcat é o "all to all replication". Esse algoritmo é um exemplo de replicação pessimista, pois replica as sessões por todos os nós do cluster, de forma que todos os servidores permaneçam iguais durante todo o tempo. Esse algoritmo apresenta bom desempenho em pequenos clusters, mas é ineficiente em clusters maiores, devido ao alto tráfego de dados entre os nós ao replicarem os dados.

Configuração de replicação de sessão no cluster

Os tópicos anteriores mostraram a necessidade do compartilhamento de dados em um cluster, inclusive com um exemplo prático dos problemas que podem ocorrer caso esse compartilhamento não seja realizado entre os nós do cluster. Este tópico mostra como configurar a replicação de sessão no Tomcat por meio do arquivo *server.xml*. Para isso, essa configuração deve ser realizada em todas as instâncias que compõem o cluster.

Para o Tomcat trabalhar com replicação de sessão é necessário remover as tags de comentário do elemento **<Cluster>**, que deve estar dentro do elemento **<Engine>**. A **Listagem 5** mostra a configuração padrão do Tomcat.

Os itens mais importantes dessa configuração são:

- O endereço e a porta multicast (configurados respectivamente pelas propriedades **address** e **port** no elemento **<Membership>**), pois permitem a um membro enviar mensagens a todos os demais nós que compõem o cluster. O endereço é configurado por default com o valor *228.0.0.4* e a porta, com o valor *45564*. O endereço e a porta juntos determinam a associação do cluster;
- O listener **ClusterSessionListener**, pois é o responsável pelo recebimento das mensagens replicadas de outros membros do cluster.

Listagem 5. Configuração de replicação de sessão em cluster no Tomcat.

```
<Cluster className="org.apache.catalina.ha.tcp.SimpleTcpCluster"
    channelSendOptions="8">

    <Manager className="org.apache.catalina.ha.session.DeltaManager"
          expireSessionsOnShutdown="false"
          notifyListenersOnReplication="true"/>

     <Channel className="org.apache.catalina.tribes.group.GroupChannel">
      <Membership className="org.apache.catalina.tribes.membership.
        McastService"
            address="228.0.0.4"
            port="45564"
            frequency="500"
            dropTime="3000"/>
      <Receiver className="org.apache.catalina.tribes.transport.nio.NioReceiver"
            address="auto"
            port="4000"
            autoBind="100"
            selectorTimeout="5000"
            maxThreads="6"/>

      <Sender className="org.apache.catalina.tribes.transport.
        ReplicationTransmitter">
        <Transport className="org.apache.catalina.tribes.transport.nio.
          PooledParallelSender"/>
      </Sender>
      <Interceptor className="org.apache.catalina.tribes.group.interceptors.
        TcpFailureDetector"/>
      <Interceptor className="org.apache.catalina.tribes.group.interceptors.
        MessageDispatch15Interceptor"/>
     </Channel>

     <Valve className="org.apache.catalina.ha.tcp.ReplicationValve"
          filter=""/>
     <Valve className="org.apache.catalina.ha.session.JvmRouteBinderValve"/>

     <Deployer className="org.apache.catalina.ha.deploy.FarmWarDeployer"
          tempDir="/tmp/war-temp/"
          deployDir="/tmp/war-deploy/"
          watchDir="/tmp/war-listen/"
          watchEnabled="false"/>

     <ClusterListener className="org.apache.catalina.ha.session.
       JvmRouteSessionIDBinderListener"/>
     <ClusterListener className="org.apache.catalina.ha.session.
       ClusterSessionListener"/>
   </Cluster>
```

Além desses itens, o elemento **<Cluster>** mantém todas as propriedades necessárias para o correto funcionamento do cluster e da replicação de sessão, como definição das classes responsáveis pela replicação de deploy (abordado no tópico "Deploy com FarmWarDeploy") e as classes de listeners e interceptors para

replicação de sessão. O **BOX 2** apresenta mais detalhes sobre estas classes.

BOX 2. Listeners e interceptors do cluster

O Tomcat fornece listeners e interceptors para tratar eventos e requisições entre os servidores que compõem o cluster durante sua execução. A **Listagem 5** define os listeners JvmRouteSessionIDBinderListener e ClusterSessionListener por meio do elemento <ClusterListener> e os interceptors TcpFailureDetector e MessageDispatch15Interceptor por meio do elemento <Interceptor> dentro de <Channel>.

TcpFailureDetector intercepta mensagens não entregues em casos de timeout. MessageDispatch15Interceptor possibilita a comunicação assíncrona entre os nós do cluster. Ele intercepta a flag Channel.SEND_OPTIONS_ASYNCHRONOUS em uma mensagem e, caso a flag esteja presente, a mensagem é colocada em uma fila e uma resposta é enviada imediatamente ao solicitante avisando que a mensagem foi entregue.

ClusterSessionListener é responsável pelo recebimento das mensagens replicadas de outros membros do cluster e JvmRouteSessionIDBinderListener é responsável pelo recebimento de alterações no SessionID.

Configuração da aplicação para rodar no cluster

O último passo para ter um ambiente em cluster no qual a aplicação faça replicação de sessão é configurar as aplicações do cluster por meio da tag **<distributable>** no arquivo *web.xml*. Essa tag marca a aplicação como uma aplicação distribuída que suporta rodar em ambientes distribuídos.

Com essa tag, todas as novas sessões e cada alteração em sessões existentes serão replicadas para todos os membros do cluster, de acordo com as regras mostradas na **Listagem 5**, que são configuradas no arquivo *server.xml*, especificamente no elemento **<Cluster>**. Quando uma aplicação não é marcada como distribuída e é implantada em um ambiente de cluster, as alterações na sessão ocorrerão apenas em uma JVM e se o usuário for redirecionado, em uma requisição posterior, para uma JVM diferente, a sessão não será reconhecida.

Deploy com FarmWarDeploy

A classe **FarmWarDeployer**, do Tomcat, é a responsável por fazer o deploy/undeploy das aplicações no cluster. Tal mecanismo é configurado no elemento **<Deployer>** do arquivo de configuração *server.xml*, como apresentado na **Listagem 5**.

No tópico "Aplicações Distribuídas", o primeiro deploy no cluster foi realizado de forma manual, onde a aplicação teve de ser implantada em todos os nós um de cada vez. No entanto, não é mais necessário fazer o deploy individual em cada instância do Tomcat. Com a configuração do elemento **<Deployer>**, ao fazer o deploy em um dos nós, a aplicação será replicada em todos os

membros do cluster. Esse conceito de implantação replicada é chamado de "farming".

Funcionamento do cluster

Agora que o cluster foi configurado, é importante entender como ele funciona e o que acontece durante as requisições de uma forma mais detalhada. Sendo assim, esta seção explica o funcionamento do cluster a partir de uma série de eventos no ciclo de vida dos servidores e aplicações em um ambiente distribuído.

Para isso, analisaremos como o cluster é inicializado e como os servidores se comportam durante a inicialização. Logo após, será comentado como são tratadas as requisições e como acontece a replicação de sessão. O ciclo de vida do nosso cluster exemplo é o seguinte:

1. Tomcat1 é iniciado: O Tomcat1 é iniciado usando sua sequência padrão de inicialização, por meio do comando *startup*. Devido às configurações realizadas, um objeto cluster é associado ao host. Ao iniciar a aplicação, caso o arquivo *web.xml* contenha a tag **<distributable>**, o Tomcat solicita ao objeto cluster (objeto do tipo **SimpleTcpCluster**, conforme configurado pela propriedade **className** do elemento **<Cluster>** na **Listagem 5**) a criação de um gerenciador para contexto replicado, que inicia um serviço multicast e outro unicast para troca de informações entre os membros do cluster;

2. Tomcat2 e Tomcat3 são iniciados: os demais servidores são iniciados usando a mesma sequência padrão de inicialização do Tomcat. Entretanto, como o cluster já está em execução, os demais servidores (Tomcat2 e Tomcat3) iniciados, que também estão configurados para trabalhar de forma distribuída, são associados ao cluster. Nesse momento a arquitetura distribuída mostrada na **Figura 1** está em execução e os servidores recém-adicionados solicitam a algum servidor que já exista no cluster as informações de sessão para cada aplicação marcada como distribuída por meio da tag **<distributable>**;

3. Tomcat1 recebe uma requisição e cria a sessão Session1: a requisição é redirecionada para o Tomcat1 pelo load balancer e tratada da mesma forma como se não houvesse replicação de sessão. Quando a requisição é finalizada, antes de retornar para o usuário, a sessão é replicada para os outros membros do cluster;

4. Tomcat2 receber uma requisição da sessão Session1: devido à replicação de sessão, o Tomcat2 possui a Session1 e trata a requisição normalmente, fazendo a replicação entre os membros do cluster ao término da requisição.

Os passos mencionados ilustram o comportamento básico do cluster e o que acontece em meio a cenários de inicialização, requisições e replicações de sessões durante o ciclo de vida de uma aplicação.

Há ainda o cenário de falha em algum servidor membro do cluster. Quando um servidor cai, os demais membros do cluster recebem uma notificação do servidor que falhou e retiram o mesmo da lista de associação (*membership list*). Feito isso, o load balancer passa a redirecionar as requisições entre os servidores que continuam funcionando. Assim, verifica-se que uma das principais vantagens em um ambiente distribuído é a tolerância a falhas, pois se um membro do cluster falha, outros tomarão seu lugar e isso acontece de forma transparente ao requisitante.

A computação distribuída é totalmente diferente da abordagem standalone. A arquitetura de um sistema bem sucedido em um ambiente distribuído depende de um projeto pensado desde o início levando em consideração as características, vantagens e limitações de arquiteturas distribuídas.

Além das técnicas e ferramentas apresentadas neste artigo, há muito mais a explorar. Existe uma grande variedade de ferramentas e frameworks voltados para a computação distribuída a serem levados em consideração no momento de criar um modelo arquitetural para aplicações distribuídas e o conhecimento dos principais conceitos da computação distribuída é muito importante para fazer as escolhas adequadas de implementação.

Autor

Gabriel Novais Amorim
novais.amorim@gmail.com – blog.gabrielamorim.com
Tecnólogo em Análise e Desenvolvimento de Sistemas (Unifieo) e especialista (MBA) em Engenharia de Software (FIAP). Trabalha com desenvolvimento de software há cinco anos. Possui as certificações OCJP, OCEJWCD, OCEEJBD, OCEJWSD, IBM-OOAD, IBM-RUP, CompTIA Cloud Essentials e SOACP.

Links:

Instalação do Apache Tomcat.
http://tomcat.apache.org/tomcat-7.0-doc/appdev/installation.html

Site oficial do NGinx.
http://nginx.org/

Instalação do NGinx.
http://nginx.org/en/docs/install.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