easy java Magazine
Edição 45
DEVMEDIA

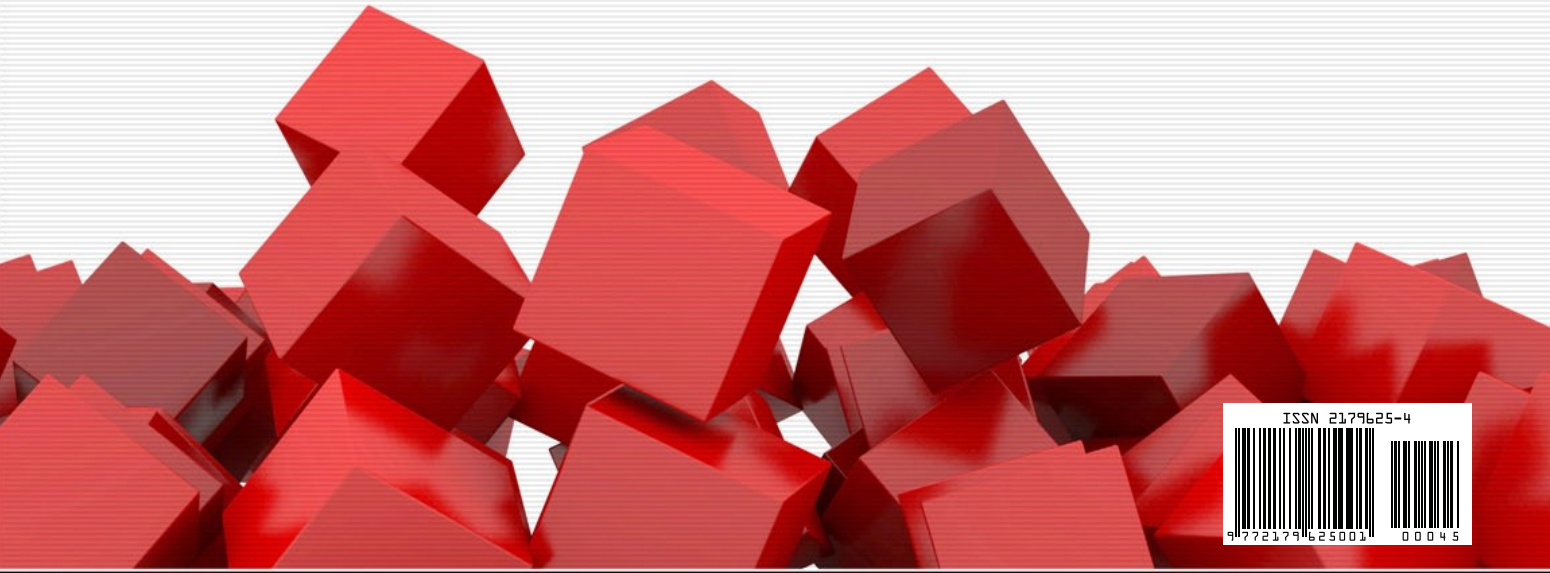
DESAFIO:
Desenvolva aplicações com o Play! Framework
Implementando sistemas web com qualidade e facilidade

Programando com a Java 3D
Aprenda a manipular formas geométricas utilizando a Java3D

Programação Orientada a Objetos
Código ágil e otimizado com os poderosos recursos do Java

SOLID

Conheça os cinco princípios de um código OO



ISSN 2179625-4

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 45 • 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

Java 3D: Criando e movimentando formas geométricas em Java

Aprenda a movimentar formas geométricas utilizando a Java3D

Fique por dentro

Este artigo apresenta uma introdução aos conceitos necessários para movimentar formas geométricas utilizando as ferramentas da API Java3D, explicando a estrutura que deve ser criada para montar a cena e os objetos de transformação necessários para obter a movimentação dos gráficos tridimensionais. Junto com as explicações, alguns exemplos demonstrando os conceitos abordados serão analisados para facilitar a sua compreensão.

A biblioteca Java3D possibilita escrever aplicações com gráficos tridimensionais, fornecendo as ferramentas para criar, manipular e visualizar geometrias 3D. Nesta biblioteca, a organização dos dados que representam os elementos que compõem a imagem é baseada em grafos. Isto quer dizer que os elementos da cena tridimensional são organizados em uma estrutura hierárquica de dados, que recebe o nome de grafo de cena. A visão geral de como criar e manipular as geometrias em um grafo de cena é apresentada no artigo "Primeiros passos com a Java3D", publicado na easy Java Magazine 36. Como apresentado neste primeiro artigo, as informações que representam um objeto tridimensional são chamadas de modelo geométrico, geometria ou simplesmente modelo.

O grafo de cena é utilizado para organizar os elementos que formam o que se chama de universo. O universo para uma aplicação gráfica, ou mais especificamente para uma cena tridimensional, deve conter todas as informações necessárias para a geração da(s) imagem(ens) desejada(s). Em um grafo de cena estão presentes, além dos dados que representam as geometrias dos elementos que devem aparecer na imagem, elementos que representam ações, ou movimentos, que alteram a forma ou a posição das geometrias definidas. Estas alterações podem representar um simples deslocamento utilizado para posicionar corretamente o modelo, ou também rotações e transformações que alterem a forma do modelo geométrico no qual está sendo aplicada.

Neste artigo serão discutidas as transformações de deslocamento e rotação juntamente com esquemas que permitem criar movimentos, além de demonstrar como permitir ao usuário controlar algumas destas transformações, fazendo com que ele controle o ponto de visão ou a posição da forma no espaço virtual criado pela aplicação. No restante do artigo os desenhos tridimensionais serão chamados de formas, geometrias ou ainda modelos, já o termo objeto será utilizado apenas para denotar instâncias de classes Java, definidas pelas APIs abordadas ou nos exemplos.

Posicionando as formas

Para posicionar uma forma geométrica no espaço é necessário indicar o ponto que esta ocupa em relação à origem do sistema de coordenadas, ou seja, define-se qual é a distância da forma ao ponto (0, 0, 0). Para realizar isso na Java3D, o objeto que representa a forma que se deseja posicionar precisa estar associado a um objeto de transformação que implementa uma translação. Isto é realizado pela classe **TransformGoup**, na qual deve ser especificada a transformação a ser realizada e as geometrias nas quais esta será aplicada. A translação é especificada chamando o método **setTranslation()** da classe **Transform3D**, que recebe como parâmetro um objeto do tipo **Vector3f** que representa o vetor de translação a que a forma será submetida, ou seja, o quanto será deslocada em cada um dos eixos ordenados (x, y, z).

No exemplo apresentado na **Listagem 1**, uma série de formas são posicionadas em cada um dos eixos, sendo colocadas esferas no eixo *x*, cones no eixo *y* e cilindros no eixo *z*. As formas criadas têm tamanho de cinco centímetros, o que pode ser observado nas

chamadas de seus respectivos construtores, que são os primeiros comandos em cada um dos três laços presentes no código. Cada uma destas formas é posicionada a cada 10 centímetros nos eixos ordenados, o que está especificado nos construtores dos objetos do tipo **Vector3f**, utilizados para especificar a translação a ser realizada em cada uma das geometrias declaradas. O resultado deste exemplo é mostrado na **Figura 1**, porém a visualização da disposição dos cilindros é prejudicada porque o eixo *z* representa a profundidade em relação ao plano de visão do dispositivo de saída, no caso o monitor.

Uma maneira de melhorar a visualização dos cilindros é alterar a disposição dos eixos utilizados no modelo. Isto pode ser realizado permitindo ao usuário movimentar o ponto de visão em volta da origem do sistema. Na **Listagem 2** são acrescentadas as chamadas para acrescentar este comportamento. Para permitir que o usuário manipule o ponto de visão é necessário alterar as características do objeto **ViewingPlatform**, que está vinculado ao universo. A referência a este objeto é fornecida chamando o método **getViewingPlatform()** da classe **SimpleUniverse**.

O **ViewingPlatform** deve ser associado a um objeto de comportamento para permitir ao usuário manipular o ponto de visão. Um objeto de comportamento faz a associação entre um grafo de cena, ou uma parte deste, e uma ação que deve ser executada (comportamento) pelos modelos que fazem parte do grafo. Um objeto do tipo **ViewingPlatform** define o ponto de observação, ou seja, de onde se está olhando para os modelos que compõem a cena, o tamanho da janela de visualização e também características de como a imagem é gerada a partir do grafo de cena.

A classe **OrbitBehavior**, por sua vez, movimenta um elemento quando o mouse é arrastado com um botão pressionado, incluindo

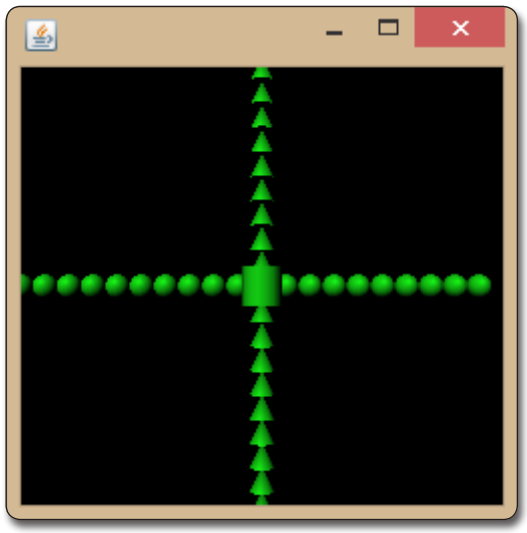


Figura 1. Resultado da execução do código da **Listagem 1**

Listagem 1. Criação de um conjunto de geometrias ao longo dos eixos ordenados.

```
public class Eixos3
{
 public Eixos3()
 {
  // Cria o universo virtual
  SimpleUniverse universo = new SimpleUniverse();
  // Cria a raiz do grafo de cena
  BranchGroup grupo = new BranchGroup();

  // Eixo X formado por esferas
  // Coordenadas do eixo X variando de -1 metro até 1 metro em intervalos de 10 cm
  for (float x = -1.0f; x <= 1.0f; x = x + 0.1f)
  {
   Sphere esfera = new Sphere(0.05f);
   TransformGroup tg = new TransformGroup();
   Transform3D transform = new Transform3D();
   transform.setTranslation(new Vector3f(x, .0f, .0f));
   tg.setTransform(transform);
   tg.addChild(esfera);
   // Adiciona a esfera transladada ao grafo de cena
   grupo.addChild(tg);
  }

  // Eixo Y formado por cones
  // Coordenadas do eixo Y variando de -1 metro até 1 metro em intervalos de 10 cm
  for (float y = -1.0f; y <= 1.0f; y = y + 0.1f)
  {
   Cone cone = new Cone(0.05f, 0.1f);
   TransformGroup tg = new TransformGroup();
   Transform3D transform = new Transform3D();
   transform.setTranslation(new Vector3f(.0f, y, .0f));;
   tg.setTransform(transform);
   tg.addChild(cone);
   grupo.addChild(tg);
  }

  // Eixo Z formado por cilindros
  // Coordenadas do eixo Z variando de -1 metro até 1 metro em intervalos de 10 cm
  for (float z = -1.0f; z <= 1.0f; z = z + 0.1f)
  {
   Cylinder cilindro = new Cylinder(0.05f, 0.1f);
   TransformGroup tg = new TransformGroup();
   Transform3D transform = new Transform3D();
   transform.setTranslation(new Vector3f(.0f, .0f, z));;
   tg.setTransform(transform);
   tg.addChild(cilindro);
   grupo.addChild(tg);
  }

  // Cria uma luz direcional verde que brilha por 100m, a partir da origem
  Color3f luz1Cor = new Color3f(.1f, 1.4f, .1f); // luz verde
  BoundingSphere limite = new BoundingSphere(new Point3d(0.0, 0.0, 0.0), 100.0);
  Vector3f luz1Direcao = new Vector3f(4.0f, -7.0f, -12.0f);
  DirectionalLight luz1   = new DirectionalLight(luz1Cor, luz1Direcao);
  luz1.setInfluencingBounds(limite);
  grupo.addChild(luz1);

  // Olha na direção dos eixos
  universo.getViewingPlatform().setNominalViewingTransform();

   // Adiciona o grupo ao grafo de cena
   universo.addBranchGraph(grupo);
  }

  public static void main(String[] args)
  {
   new Eixos3();
  }
}
```

as ações de rotação, zoom e translação. Assim é possível permitir que o usuário gire, desloque, aproxime ou afaste o elemento que está controlando. O construtor desta classe recebe como primeiro parâmetro o **Canvas** no qual o comportamento deve ser adicionado (a classe **Canvas** representa a área de desenho que a Java3D está utilizando para mostrar os modelos geométricos), e como segundo parâmetro um conjunto de flags, ou sinais, para configurar os comportamentos desejados, podendo-se ativar ou desativar a rotação, a translação e o zoom. Por padrão, todas estas ações estão ativas.

No exemplo da **Listagem 2**, o método **getCanvas()** da classe **SimpleUniverse** é utilizado para obter a referência ao **Canvas** e o flag **REVERSE_ALL** é utilizado para inverter todos os movimentos associados ao mouse, para que a movimentação seja mais intuitiva, pois com o comportamento padrão, ao se mover o mouse para baixo, por exemplo, uma rotação para cima é executada. Ao aplicar este flag, a rotação segue a direção do movimento do mouse.

Uma observação importante é que um objeto do tipo **OrbitBehavior** precisa ser informado sobre a região na qual os comportamentos definidos por ele são válidos. Com este intuito, um objeto do tipo **BoundingSphere**, com centro em (0, 0, 0) e raio 100 é instanciado e fornecido como parâmetro ao método **setSchedulingBonds()**.

Estes passos configuram o objeto **orbit**, que agora precisa ser associado ao **viewingPlatform** chamando-se o método **setViewPlatformBehavior()**. Com isto feito, o usuário pode modificar o ponto de visão do universo ao pressionar o botão esquerdo do mouse e arrastar. Isto faz com que a câmera possa rodar ao redor do ponto (0, 0, 0), aproximar ou afastar a câmera ou ainda transladar em qualquer direção. Na **Figura 2** pode ser visto um resultado da rotação aplicada aos eixos cartesianos.

Listagem 2. Permitir ao usuário mudar o ponto de visão.

```
// Olha na direção dos eixos
universo.getViewingPlatform().setNominalViewingTransform();
ViewingPlatform viewingPlatform = universo.getViewingPlatform();
OrbitBehavior orbit = new OrbitBehavior(universo.getCanvas(), OrbitBehavior.
REVERSE_ALL);
BoundingSphere bounds = new BoundingSphere(new Point3d(0.0, 0.0, 0.0),
100.0);
orbit.setSchedulingBounds(bounds);
viewingPlatform.setViewPlatformBehavior(orbit);

// Adiciona o grupo ao grafo de cena
universo.addBranchGraph(grupo);
```

Movimento a partir da rotação

Os objetos que definem comportamentos na Java3D também podem ser utilizados para especificar movimentos para os modelos ou partes do grafo de cena. No próximo exemplo, apresentado na **Listagem 3**, esta característica é demonstrada associando-se um comportamento de rotação a um cubo para fazer com que o mesmo gire em um de seus eixos.

Para que o movimento de rotação seja percebido, uma sequência de imagens deve ser gerada, sendo em cada uma delas o cubo desenhado em uma posição ligeiramente diferente. Isto é realizado alterando-se a posição do cubo aplicando-se sobre o mesmo uma transformação de rotação. Portanto, para cada frame uma nova rotação do cubo deve ser calculada e aplicada.

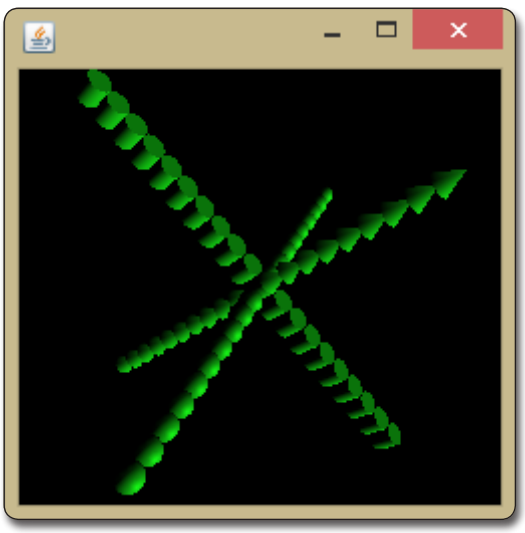


Figura 2. Rotação dos eixos usando OrbitBehavior na **Listagem 2**

O primeiro passo para permitir que um modelo possua um comportamento no grafo de cena é criar um ramo do tipo **TransformGroup** e informar que o mesmo será modificado em tempo de execução, com o flag **ALLOW_TRANSFORM_WRITE**. Se isto não for informado, a Java3D assume que o grupo é estático e as transformações aplicadas neste ramo já estão prontas, ou seja, não é permitido alterar os seus valores em tempo de execução, o que não possibilita uma sequência de transformações nos modelos, impossibilitando a visualização de movimentos.

No ramo do grafo de cena do tipo **TransformGroup** podem ser adicionados modelos e comportamentos, sendo os comportamentos aplicados a todas as geometrias pertencentes ao ramo do grafo de cena. Para rotacionar um cubo, precisa-se então criar e adicionar um cubo e uma transformação de rotação ao grupo. Para exibir um cubo na cena, basta instanciar um objeto do tipo **ColorCube** com o tamanho desejado e inclui-lo no grafo.

O controle do movimento de rotação é realizado por um objeto da classe **RotationInterpolator**, que é uma especialização de **Behavior**. Para configurar a rotação é necessário fornecer ao construtor de **RotationInterpolator** as informações sobre o número de vezes que a rotação será aplicada e qual o tempo de cada rotação, sobre qual grupo a rotação será aplicada, uma matriz de transformação e os ângulos mínimo e máximo da rotação em radianos.

A classe **Alpha** fornece os métodos para converter um valor de tempo em um valor no intervalo [0, 1], o que possibilita controlar a velocidade de rotação do cubo informando em quanto tempo, em milissegundos, deve ser completada uma volta. O primeiro parâmetro do construtor da classe **Alpha** indica quantas vezes o laço será repetido, ou neste caso, quantas rotações o cubo deve

realizar. O valor *-1* indica que não há um limite estabelecido e o laço será repetido indefinidamente. O segundo parâmetro, por sua vez, informa o tempo, em milissegundos, que cada iteração deve demorar.

Listagem 3. Associando a uma geometria um movimento de rotação.

```
public class RotacionarCubo{

 public RotacionarCubo(){
  SimpleUniverse universo = new SimpleUniverse();
  BranchGroup grupo = new BranchGroup();

  // Cria o nó Transformgroup e inicializa-o com a matriz identidade.
  TransformGroup transform = new TransformGroup();
  // Habilita a capacidade de escrita TRANSFORM_WRITE, para que o objeto
  // de comportamento possa modificá-lo em tempo de execução.
  transform.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
  // Adiciona o TransformGroup à raiz do grafo.
  grupo.addChild(transform);

  // Cria um cubo de 40cm de lado e adiciona-o ao grafo de cena.
  transform.addChild(new ColorCube(0.4));

  // Cria a matriz de transformação 3D
  Transform3D matTransf = new Transform3D();
  // Parâmetros da transformação:
  // Laço infinito, com duração de quatro segundos por rotação
  Alpha alphaRot = new Alpha(-1, 4000);
  // Cria interpolador de rotação
  RotationInterpolator rotacionar = new RotationInterpolator(alphaRot, trans-
form,
   matTransf, 0.0f, (float) Math.PI * 2.0f);
  BoundingSphere limites = new BoundingSphere(new Point3d(0.0, 0.0, 0.0),
100.0);
  rotacionar.setSchedulingBounds(limites);
  grupo.addChild(rotacionar);

  universo.getViewingPlatform().setNominalViewingTransform();
  ViewingPlatform viewingPlatform = universo.getViewingPlatform();
  OrbitBehavior orbit = new OrbitBehavior(universo.getCanvas(), OrbitBehavior.
REVERSE_ALL);
  BoundingSphere bounds = new BoundingSphere(new Point3d(0.0, 0.0, 0.0),
100.0);
  orbit.setSchedulingBounds(bounds);
  viewingPlatform.setViewPlatformBehavior(orbit);
  universo.addBranchGraph(grupo);
 }

 public static void main(String[] args){
  new RotacionarCubo();
 }
}
```

Os intervalos de rotação serão calculados pela classe **RotationInterpolator** de acordo com o número de frames que serão exibidos durante o tempo de rotação. Os parâmetros para a rotação são especificados em uma matriz de transformação, representada pelo objeto do tipo **Transform3D**, que recebe os valores da transformação aplicada em cada um dos frames a serem exibidos.

Por último é necessário especificar os limites onde estas transformações são válidas, ou seja, qual é a região do universo que será afetada pelo comportamento definido e associado ao grupo de transformações. Para que o usuário possa visualizar a rotação do cubo de diferentes ângulos, o controle sobre o ponto de visualização é adicionado como no último exemplo, permitindo que a câmera seja movimentada em volta do cubo. Na **Figura 3** tem-se um momento da rotação do cubo com o ângulo de visualização alterado para permitir a visão da sua face superior.

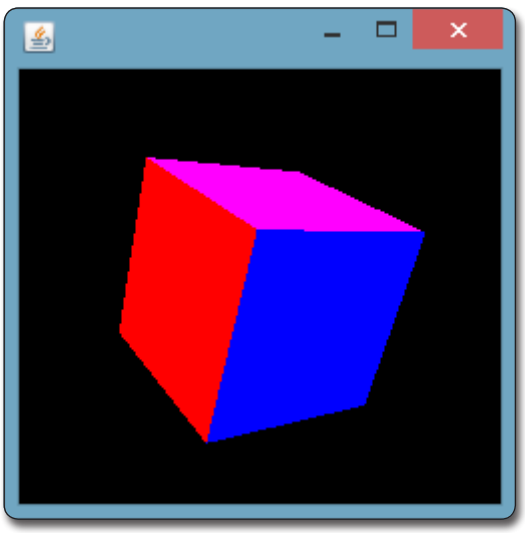


Figura 3. Resultado do código da **Listagem 3**

Movimentando e alterando a geometria

As transformações aplicadas em um grupo podem ser calculadas como no exemplo anterior ou podem ser uma resposta a um estímulo externo, como um comando do usuário, ou ainda uma combinação de ambas as situações. Assim, parte do movimento de um modelo pode ser controlado pelo usuário, como no caso da direção de um personagem em um jogo, combinado com o cálculo, digamos, de sua velocidade. Para exemplificar esta situação, o próximo exemplo apresenta uma bola que se movimenta na vertical, do topo até a base da janela, e o usuário controla a posição horizontal da mesma com as teclas 'a' e 's' (veja o código na **Listagem 4**).

A criação do grafo de cena para este exemplo é similar à do exemplo anterior, trocando-se o cubo por uma esfera e com a adição de uma fonte de iluminação direcional. O ramo do grafo correspondente à transformação é criado da mesma maneira que no exemplo de rotação do cubo.

O controle de posicionamento da bola é realizado pela alteração dos valores das variáveis de instância **altura**, **posX** e **sentido**. Estas variáveis são alteradas nos eventos gerados a partir do teclado para o posicionamento horizontal e nos eventos gerados a partir de um objeto do tipo **Timer** para o posicionamento vertical.

Como neste caso o código de alteração, ou seja, as transformações aplicadas ao modelo, dependem de fatores externos (eventos de teclado e **Timer**) e não apenas dos parâmetros definidos no código, os objetos **TransformGroup** e **Transform3D** são declarados como atributos da classe, permitindo assim que seus valores sejam alterados pelos métodos de tratamento de eventos, modificando a transformação aplicada à bola de acordo com os comandos do usuário.

A posição da bola na imagem é controlada por dois elementos: um correspondente ao posicionamento horizontal, controlado pelo usuário, e outro correspondente ao posicionamento vertical, controlado por um temporizador, para indicar os instantes nos quais a posição vertical deve ser alterada para cima ou para baixo, alterando-se o valor da variável **altura**. Este temporizador é iniciado ou parado no método **actionPerformed()** juntamente com o evento do próprio temporizador, que ocorre a cada 30

Listagem 4. Exemplo que implementa uma bola quicando.

```
public class BolaSaltando extends JFrame implements ActionListener, KeyListener{

 private Button ini = new Button("Iniciar");
 private TransformGroup transformGroup;
 private Transform3D matrizTransf  = new Transform3D();
 private float altura = 0.0f;
 private float sentido = 1.0f;  // subindo ou descendo
 private Timer tempo;
 private float posX = 0.0f;

 public BranchGroup criarGrafoCena()
 {
  // Cria a raiz do grafo de cena
  BranchGroup grupo = new BranchGroup();
  transformGroup = new TransformGroup();
  transformGroup.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
  grupo.addChild(transformGroup);

  // Cria uma esfera e a adiciona ao grafo de cena
  Sphere esfera = new Sphere(0.25f);
  transformGroup = new TransformGroup();
  transformGroup.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
  Transform3D pos1 = new Transform3D();
  pos1.setTranslation(new Vector3f(0.0f, 0.0f, 0.0f));
  transformGroup.setTransform(pos1);
  transformGroup.addChild(esfera);
  grupo.addChild(transformGroup);

  // Cria a luz vermelha e a adiciona ao grafo
  BoundingSphere limite = new BoundingSphere(new Point3d(0.0, 0.0, 0.0), 100.0);
  Color3f luz1Cor = new Color3f(1.0f, 0.0f, 0.2f);
  Vector3f luz1Direcao = new Vector3f(4.0f, -7.0f, -12.0f);
  DirectionalLight luz1  = new DirectionalLight(luz1Cor, luz1Direcao);
  luz1.setInfluencingBounds(limite);
  grupo.addChild(luz1);

  // Configura a luz ambiente e a adiciona ao grafo
  Color3f luzAmbienteCor = new Color3f(1.0f, 1.0f, 1.0f);
  AmbientLight luzAmbiente = new AmbientLight(luzAmbienteCor);
  luzAmbiente.setInfluencingBounds(limite);
  grupo.addChild(luzAmbiente);

  return grupo;
 }

 public BolaSaltando() {
  // Configura o ambiente do Applet
  setLayout(new BorderLayout());
  GraphicsConfiguration config = SimpleUniverse.getPreferredConfiguration();
  Canvas3D canvas3d = new Canvas3D(config);
  add("Center", canvas3d);
  canvas3d.addKeyListener(this);
  tempo = new Timer(30, this);

  Panel painel = new Panel();
  painel.add(ini);
  add("North", painel);
  ini.addActionListener(this);
  ini.addKeyListener(this);

  // Cria uma cena e a adiciona ao universo virtual
  BranchGroup cena = criarGrafoCena();
  SimpleUniverse universo = new SimpleUniverse(canvas3d);
  universo.getViewingPlatform().setNominalViewingTransform();
  universo.addBranchGraph(cena);
 }

 // chamado quando uma tecla é pressionada.
 public void keyPressed(KeyEvent e){
  //Verifica se a tecla pressionada foi ´s´
  if (e.getKeyChar() == 's') {
   if(posX < 1f)
    posX = posX + .01f;
  }
  // Verifica se a tecla pressionada foi ´a´
  if (e.getKeyChar() == 'a') {
   if(posX > -1f)
    posX = posX - .01f;
  }
 }

 // chamado quando uma tecla é liberada.
 public void keyReleased(KeyEvent e){
 }

 // chamado quando uma tecla é teclada
 public void keyTyped(KeyEvent e){
 }

 public void actionPerformed(ActionEvent e){
  // inicia o temporizador quando o botão é clicado
  if (e.getSource() == ini){
   if (!tempo.isRunning()) {
    ini.setLabel("Parar");
    tempo.start();
   } else {
    ini.setLabel("Iniciar");
    tempo.stop();
   }
  } else {
   altura += .03 * sentido;
   if (Math.abs(altura * 2) >= 1)
    sentido = -1.0f * sentido;
   if (altura < -0.4f) {
    matrizTransf.setScale(new Vector3d(1.0, .8, 1.0));
   } else {
    matrizTransf.setScale(new Vector3d(1.0, 1.0, 1.0));
   }
   matrizTransf.setTranslation(new Vector3f(posX, altura, 0.0f));
   transformGroup.setTransform(matrizTransf);
  }
 }

 public static void main(String[] args){
  System.out.println("Programa Iniciado");
  BolaSaltando bolaSaltando = new BolaSaltando();
  bolaSaltando.addKeyListener(bolaSaltando);
  bolaSaltando.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  bolaSaltando.setSize(256, 256);
  bolaSaltando.setVisible(true);
 }
}
```

milissegundos, conforme a sua configuração (**tempo = Timer(30, this);**). Quando ocorre o evento do temporizador, a posição vertical da esfera é atualizada e os limites de seu movimento são verificados. Diferente do caso da rotação do cubo no exemplo anterior, quando se utilizou um interpolador, o movimento da bola sofre uma alteração em seu sentido dependendo de sua posição. Por isto é necessário utilizar o temporizador e controlar o movimento diretamente.

Na movimentação vertical da bola, caso o limite superior seja atingido, o sentido do movimento é invertido. Já no caso do limite inferior, antes de inverter o movimento, uma verificação adicional é realizada. Assim, caso o limite esteja se aproximando, uma nova transformação é adicionada à bola, alterando a sua forma de maneira a achatar a esfera e simular o choque da bola com o solo. Este efeito é conseguido com uma transformação de escala, na qual apenas o componente vertical da bola é alterado. Isto é conseguido a partir dos parâmetros (1.0, 0.8 e 1.0) passados ao construtor do objeto **Vector3d**, parâmetros do método **setScale()**. Estes parâmetros indicam o quanto cada dimensão deve ser alterada. O valor 1 indica que não há alteração, um valor menor do que 1 indica uma redução do tamanho do modelo e um valor maior do que um indica um aumento do tamanho do modelo.

Por uma questão de simplicidade, a transformação de escala é aplicada na esfera como um evento único, porém este comportamento não é o correto. O ideal seria a utilização de outra variável de controle para que a transformação de escala seja realizada gradualmente, da mesma maneira que a transformação de translação (posicionamento).

Esta transformação de escala é aplicada quando a posição da bola se aproxima do limite inferior estabelecido para o movimento, porém a transformação de translação, que controla o posicionamento da bola depende dos eventos de teclado e do temporizador. A matriz de transformação correspondente à translação depende dos valores das variáveis **altura** e **posX**. A **altura** da bola depende do tempo transcorrido, já a sua posição **horizontal** depende do controle do usuário. Sendo assim, a atualização da variável que controla a posição horizontal da bola é realizada no método de tratamento de eventos do teclado **keyPressed()**, no qual é verificada qual tecla foi pressionada e a variável **posX** é incrementada ou decrementada de acordo, enquanto a mesma não atingir os limites estabelecidos.

O resultado dos movimentos associados à bola pode ser visualizado na **Figura 4**, na qual a sequência de imagens representa a bola movendo-se para baixo na imagem da esquerda, depois a bola aproximando-se da base, na imagem do centro, e por último a bola aproximando-se do limite vertical superior, no qual não é aplicada a transformação de escala.

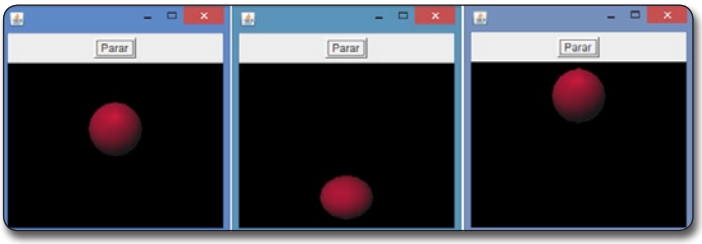


Figura 4. Resultado da execução do código da **Listagem 4**

A partir do conteúdo explorado forma-se uma base para o início do estudo sobre o desenvolvimento de aplicações que fazem a síntese de imagens (geram imagens) com base na definição de modelos geométricos. Como você poderá constatar, os métodos de controle utilizados para posicionar os modelos podem ser incrementados para que transformações mais suaves sejam conseguidas, ou uma combinação mais complexa de parâmetros pode ser utilizada para controlar o movimento de modelos que podem ser parte de um jogo ou de uma simulação, por exemplo.

Estas técnicas, em conjunto, podem ser adotadas em diferentes ramos do grafo de cena, o que permite um controle ainda mais detalhado dos elementos, viabilizando, por exemplo, que dois usuários controlem personagens diferentes em um jogo. Sendo assim, continue explorando esta API. A sua criatividade ditará o limite para ela quanto à criação de cenários e objetos 3D.

Autor

Miguel Diogenes Matrakas
mdmatrakas@yahoo.com.br

É Mestre em Informática pela PUC-PR e doutorando em Métodos Numéricos em Engenharia, pela UFPR (Universidade Federal do Paraná). Trabalha como professor de Java há quatro anos nas Faculdades Anglo-Americano de Foz do Iguaçu.

Links:

Site oficial para download da biblioteca Java 3D.
www.oracle.com/technetwork/java/javase/tech/index-jsp-138252.html

Instruções de instalação da Java3D.
http://download.java.net/media/java3d/builds/release/1.5.1/README-download.html

Site do grupo de desenvolvimento da Java3D.
https://java3d.java.net/

Qualidade no código com os princípios S.O.L.I.D.

Como criar um design sólido através dos cinco princípios da Orientação a Objetos

É muito comum, no nosso dia a dia de desenvolvimento, nos depararmos com sistemas detentores de um código não reutilizável, com classes fortemente acopladas e não extensíveis. Com isso, tanto o trabalho de alterar funcionalidades já existentes quanto o de adicionar novas se torna uma tarefa muito complexa, dispondo de demasiado tempo e esforço.

Um sistema, quando implementado dentro de um cenário como o exposto, se torna frágil e nos remete a uma série de situações de erro. Erros se apresentam em qualquer módulo do projeto quando outra área do código é alterada. Problemas como classes com mais de uma responsabilidade, métodos não extensíveis e interfaces poluídas consistem em falhas que podem ser de fato reduzidas com a prática do desenvolvimento sólido.

Uma implementação sólida é aquela em que as responsabilidades são bem definidas e distribuídas, e os comportamentos existentes podem ser reutilizados e facilmente estendidos para outras funcionalidades. Com o intuito de ajudar o desenvolvedor a alcançar essa meta, foi criado o acrônimo S.O.L.I.D. Este acrônimo é simplesmente a junção de cinco princípios que, quando adotados, se tornam garantia de um código mais robusto e flexível.

Com base nisso, o objetivo deste artigo é apresentar de forma prática e abrangente o uso destes princípios, visando levar ao desenvolvedor o conhecimento necessário para começar a aplicá-los em seu ambiente de programação, seja em projetos novos ou em legados.

Fique por dentro

Este artigo tem por objetivo fornecer algumas propostas para o aperfeiçoamento do uso da Orientação a Objetos, através de uma breve apresentação do tema e da análise dos princípios S.O.L.I.D., defendidos por Robert C. Martin. Princípios que, se seguidos, aumentarão de forma significativa a simplicidade e a qualidade do código, características que trazem como consequência uma maior facilidade para a manutenção do sistema.

O Paradigma da Orientação a Objetos

O verdadeiro Design de Software é aquele em que toda a arquitetura pensada consiste na produção de código reutilizável, coeso, desenvolvido de forma legível e organizada. Isto é o que pregam as boas práticas da Orientação a Objetos: trazer benefícios reais para o desenvolvimento de softwares.

O paradigma da orientação a objetos nos apresenta uma coleção de diversos agentes interconectados. A estes agentes damos o nome de objetos e a cada um deles são atribuídas responsabilidades, ou seja, eles ficam responsáveis por executar tarefas específicas. Através dessa interação entre os objetos tarefas computacionais são realizadas.

Mas como realizar essa interação de forma limpa, legível e organizada conforme descrito anteriormente? Para desenvolver um sistema possuidor de um código com baixo acoplamento, devidamente modularizado, com suas responsabilidades bem definidas, Robert C. Martin, por volta dos anos 90, compilou cinco princípios da orientação a objetos que visam nos ajudar com a tarefa de obter um código mais sólido (S.O.L.I.D.).

Martin, mais conhecido como Uncle Bob, é um grande nome na comunidade de desenvolvimento de software, trabalhando na área desde 1970. Fundador e presidente da Object Mentor Inc. e autor de livros como *Clean Code – A Handbook of Agile Software Craftsmanship* e *Clean Coder – A Code of Conduct for Professional Programmers*. Martin, em sua publicação sobre SOLID, chama nossa atenção para quatro perguntas. O que de fato é um design orientado a objetos? Do que ele trata? Quais são seus benefícios? E o quanto irá nos custar sua implementação? Essas perguntas podem parecer simplórias numa época em que todos os desenvolvedores de software estão usando alguma linguagem orientada a objetos, seja ela de qualquer tipo. No entanto, são perguntas importantes, segundo ele, tendo em vista que a maioria de nós usa essas linguagens sem saber o porquê e sem saber como tirar o maior benefício delas.

Já a criação do acrônimo S.O.L.I.D. propriamente dito, foi introduzida por Michael Feathers, igualmente integrante da Object

Mentor Inc. e autor do livro *Working Effectively with Legacy Code*. Feathers, na busca por facilitar a absorção da ideia, percebeu que a junção das cinco letras iniciais dos princípios, a saber: *Single Responsability Principle*, *Open Closed Principle*, *Liskov Substitution Principle*, *Interface Segregation Principle* e *Dependency Inversion Principle*, formavam a palavra Solid, que nos remete aos objetivos implícitos dos princípios, isto é, um desenvolvimento mais sólido.

Os cinco princípios da OO

Muitos dos profissionais envolvidos com o desenvolvimento de software orientado a objetos possuem o conhecimento acadêmico sobre acoplamento, coesão e encapsulamento. Mas esses mesmos profissionais, em sua maioria, não têm ideia de como alcançar esses objetivos, conseguindo um código com alta coesão, baixo acoplamento e forte encapsulamento. É disso que trata o S.O.L.I.D, que de forma simples pode ser entendido como um guia que traz diretrizes a serem seguidas para o alcance desses objetivos. São elas:

• **Single Responsability Principle (SRP)** – *Princípio da Responsabilidade Única*. Esse princípio nos diz que toda classe deve possuir uma única responsabilidade. Assim, podemos ter um código mais coeso, simples e reutilizável, de forma que consigamos evitar diversas mudanças sendo propagadas ao longo do projeto;

• **Open Closed Principle (OCP)** – *Princípio do Aberto Fechado*. Com base nesse princípio, o comportamento de nossas classes deve ser extensível para outras classes, sem que o mesmo sofra quaisquer modificações para isso. Os comportamentos poderão ser estendidos seja por herança, composição ou através do uso de interfaces;

• **Liskov Substitution Principle (LSP)** – *Princípio da Substituição de Liskov*. Esse princípio nos alerta sobre o cuidado no uso da Herança. Ele afirma que devemos ser capazes de usar uma classe derivada no lugar de uma classe mãe, e essa derivação deve possuir a capacidade de se comportar da mesma forma;

• **Interface Segregation Principle (ISP)** – *Princípio da Segregação de Interfaces*. Esse princípio afirma que não se deve forçar os clientes a dependerem de interfaces que não utilizarão de fato. Ou seja, nossos módulos deverão possuir poucos comportamentos e apenas aqueles que realmente serão utilizados. Interfaces com muitos comportamentos se distribuem ao longo do sistema e acabam por dificultar futuras manutenções;

• **Dependency Inversion Principle (DIP)** – *Princípio da Inversão de Dependências*. Esse princípio nos diz que devemos depender apenas de classes abstratas e não de concretas. Classes abstratas quase nunca mudam. Assim, o código fica desacoplado, facilitando futuras mudanças no comportamento delas.

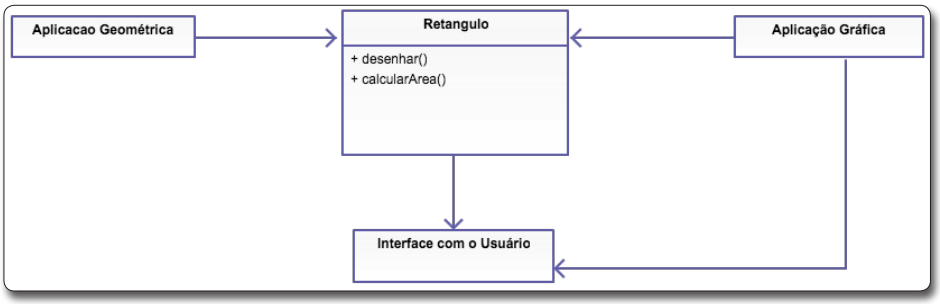


Figura 1. Excesso de responsabilidades para a classe Retangulo

Tais princípios, se colocados em prática de forma concreta, se tornam garantia de um código de evolução fácil e de mudanças simples, que serão feitas em pontos específicos do nosso sistema. Então, como aplicar esses princípios no nosso dia a dia e tornar essas práticas reais dentro de nossos projetos? Este é o tema que será abordado com exemplos ao longo desse artigo.

Princípio da Responsabilidade Única

Como o próprio nome já diz, nossas classes devem possuir uma única função, isto é, elas têm de ser responsáveis apenas pela execução do objetivo para o qual foram criadas. Quando uma classe possui diversas responsabilidades, cria-se um forte acoplamento interno, pois certamente suas funções estarão dependentes umas das outras, ao passo que qualquer alteração em um desses métodos pode causar erros em outros, tornando sua manutenção trabalhosa.

Esse tipo de acoplamento nos leva a um projeto frágil, que tende a apresentar diversos erros inesperados quando alterados. Para exemplificar melhor o problema aqui apresentado, vamos considerar o design da **Figura 1**, onde uma classe **Retangulo** possui dois métodos. Um método irá simplesmente desenhar um retângulo na tela e o outro calcular a sua área.

Como pode ser observado, duas aplicações fazem uso da classe **Retangulo**, sendo uma delas a Aplicação Geométrica, que apenas fará uso do método **calcularArea()** e assim obterá o resultado esperado, e a outra a Aplicação Gráfica, que não apenas calcula a área, mas também apresenta a figura desenhada via interface com o usuário.

Este design claramente viola a SRP (*Single Responsability Principle*), pois a classe **Retangulo** possui duas responsabilidades: calcular a área e desenhar a figura. Caso qualquer alteração na Aplicação Gráfica resulte em mudanças na classe **Retangulo**, consequentemente isso forçará uma nova compilação, novos testes e um novo deploy da Aplicação Geométrica, pois existe um acoplamento entre esses sistemas através da mesma classe.

Para resolver esse problema devemos separar completamente as responsabilidades em classes diferentes, como mostrado na **Figura 2**. Este novo design separa a responsabilidade do cálculo geométrico em uma nova classe, chamada **RetanguloGeometrico**. Agora qualquer mudança feita no processo de como o retângulo é apresentado para o usuário não irá interferir no funcionamento da Aplicação Geométrica.

Princípio do Aberto Fechado

Mudanças sempre irão acontecer em nossos sistemas, pois eles precisam mudar

e evoluir. Em virtude disso, surge o desafio de criar um projeto estável mediante essas mudanças. Vejamos o que apresenta o *Open Closed Principle* (OCP) em relação a essa meta.

O princípio do OCP prega a criação de comportamentos em nossas classes que possam ser estendidos para outras quando houver necessidade de mudanças por parte dos requisitos. Isto porque alterar comportamentos já em funcionamento demanda um grande esforço, já que todas as dependências do sistema que possuem interação com as funcionalidades a serem alteradas precisariam ser novamente testadas.

Sendo assim, nossas entidades, módulos e funções devem estar *abertas*, de modo que permitam suas extensões, mas *fechadas* para modificações. Na **Figura 3** podemos ver no exemplo de nossa aplicação gráfica como ela viola o conceito do *OCP*. Responsável por desenhar diferentes formas geométricas, do jeito que o design se encontra, podemos perceber que seríamos obrigados a alterar o código da classe **EditorGrafico** cada vez que uma nova forma fosse adicionada à lógica.

Na **Listagem 1** vemos uma representação em código sobre o design exposto que fere de forma clara o princípio do *OCP*. Como pode ser visto no método **desenharForma()** da classe **EditorGrafico**, o método possui *ifs* encadeados que, através de um tipo informado, definirão qual forma será desenhada. Com isso, para cada nova forma que necessite ser desenhada haverá mais um **if** para representá-la.

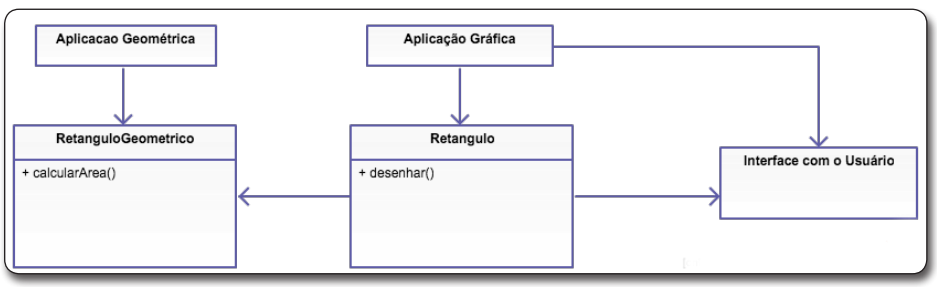


Figura 2. Responsabilidades separadas corretamente, respeitando o SRP

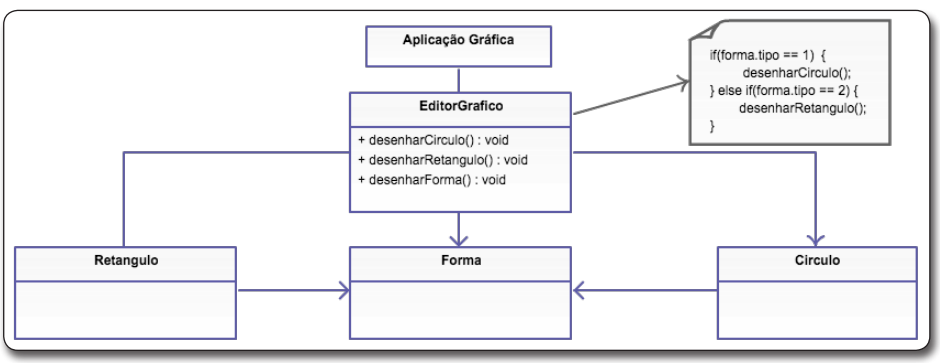


Figura 3. A classe EditorGrafico possui comportamentos não extensíveis

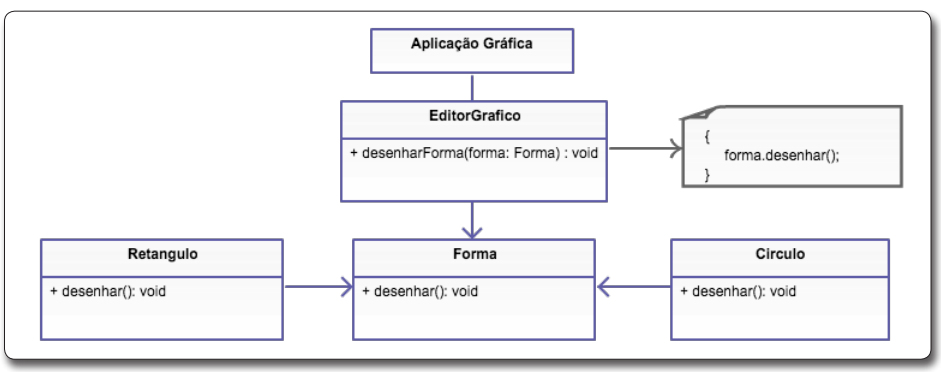


Figura 4. Implementando o princípio da OCP

Na **Figura 4** apresentamos um exemplo de design que irá suportar o *OCP*. Nesse design temos o método abstrato **desenhar()** na classe **Forma** sinalizando que toda classe do tipo forma terá a capacidade de desenhar. Assim, tanto a classe **Circulo**, quanto a classe **Retangulo**, bem como as demais formas que venham a surgir, se tornam elas próprias as responsáveis por executar a ação de desenhar, não sendo mais necessária quaisquer alterações na classe **EditorGrafico**.

Como podemos verificar, a classe **EditorGrafico** agora recebe apenas a forma que será desenhada e executa o seu método **desenhar()**. Observe a representação desse design em código na **Listagem 2**. Ao criarmos o método abstrato **desenhar()** na classe **Forma**, garantimos que todas as classes que a estendam possuam essa capacidade. Assim, o método **desenharForma()** da classe **EditorGrafico** fica apenas com a responsabilidade de receber um objeto do tipo **Forma** como parâmetro e, através dele, invocar seu método **desenhar()**.

A partir do momento em que a responsabilidade de desenhar é movida para as classes concretas, o risco de afetar funcionalidades antigas quando novas são criadas é reduzido consideravelmente.

Princípio da Substituição de Liskov

Esse princípio nos alerta para o cuidado com o uso da herança. Certamente este é um recurso poderoso e é justamente por isso que devemos empregá-lo com responsabilidade.

A herança é adotada quando se deseja reutilizar um determinado código de outra classe, ou seja, reaproveitar o comportamento existente. No entanto, ao usar herança, é muito comum disponibilizar para as classes filhas diversos métodos e atributos que as mesmas não irão utilizar e que, portanto, não precisariam ter acesso.

O próprio James Gosling (pai da linguagem Java) chama a nossa atenção para o uso indevido da herança, afirmando: "Você deve evitar implementar herança sempre que possível.".

Listagem 1. OCP – Mal exemplo de codificação.

```
public class EditorGrafico {

    public void desenharForma(Forma forma) {
        if(forma.tipo == 1) {
            desenharCirculo(forma);
        }
        else if(forma.tipo ==2){
            desenharRetangulo(forma);
                                                    }
    }

    public void desenharCirculo(Circulo circulo) {...}

    public void desenharRetangulo(Retangulo retângulo) {...}

}

public class Forma {
    int tipo;
}

public class Circulo extends Forma {

    Circulo() {
        super.tipo = 1;
    }
}

public class Retangulo extends Forma {

    Retangulo() {
        super.tipo = 2;
    }
}
```

Listagem 2. OCP – O bom exemplo de codificação.

```
public class EditorGrafico {

    public void desenharForma(Forma forma) {
        forma.desenhar();
    }
}

public class Forma {
    abstract void desenhar();
}

public void Retangulo extends Forma {
    public void desenhar() {
        //desenhar o Retangulo
    }
}

public void Circulo extends Forma {
    public void desenhar() {
        //desenhar o Circulo
    }
}
```

Ao herdarmos explicitamente uma classe concreta, ficamos presos a uma implementação específica, a classe filha passa a conhecer de forma exagerada o código interno da sua classe mãe. Este procedimento nos deixa acoplados demais à superclasse. Dessa forma, podemos notar claramente uma quebra no encapsulamento a partir do momento em que uma subclasse fica dependente do comportamento da superclasse para realizar suas operações.

Isso nos mostra a fragilidade desse tipo de implementação, pois se o código da classe mãe sofrer quaisquer mudanças, suas classes filhas poderão necessitar de mudanças também. Essa fragilidade pode ser encontrada dentro da implementação do próprio Java, como é o exemplo da classe **Properties**, que estende a classe **Hashtable**. Podemos passar para **Hashtable** qualquer objeto como chave e valor, desde que o objeto não esteja nulo. Já no caso da classe **Properties**, esta deveria receber apenas objetos do tipo **String** para chave e valor.

Nesta classe, até temos métodos como **getProperty()** e **setProperty()**, responsáveis por obter e fornecer valores do tipo **String**. Contudo, métodos como **put()** e **putAll()**, originários de **Hashtable**, continuam disponíveis em **Properties**, permitindo que valores de qualquer tipo sejam passados para a mesma. Vejamos um exemplo na **Listagem 3**.

Listagem 3. Exemplo de violação do LSP.

```
Hashtable matricula = new Properties();

matricula.put("Marcio", "3542");
matricula.put("Marcelo", "4433");

// um pequeno engano na próxima linha:
//         3224 ao invés de "3224"

matricula.put("Joaquim", 3224);
((Properties)matricula).list(System.out);

Este código resultará em uma exceção:

Marcio=3542
Exception in thread "main" java.lang.ClassCastException:
      java.lang.Integer cannot be cast to java.lang.String
      at java.util.Properties.list(Unknown Source)
 at UseProperties.main(UseProperties.java:9)
```

A motivação para o uso da Herança é a reutilização de código. No entanto, devido aos problemas apresentados, esta deve ser evitada sempre que possível. Felizmente, existe uma alternativa mais eficaz: a composição. Com ela conseguimos reaproveitamento de código sem quebrar o encapsulamento de nossas classes.

No caso da classe **Properties**, para que ela reutilize os comportamentos existentes em **Hashtable** empregando a composição, bastaria criar um atributo privado do tipo **Hashtable** e, com isso, o método **setProperty()** seria o responsável por delegar a invocação a **hashtable.put()**. Dessa forma somente os métodos necessários seriam expostos.

Um relacionamento baseado em herança traz mais dificuldades no momento de mudanças referentes à interface de uma superclasse. Isso porque mudanças no código da classe mãe podem vir a quebrar o código implementado em qualquer uma de suas subclasses. Por exemplo, em uma determinada classe **Trabalhador**, na qual seja necessária uma alteração no tipo de retorno de um dos métodos públicos, resultaria em uma quebra do código que invoca este método em qualquer classe que seja uma referência do tipo **Trabalhador** ou qualquer subclasse de **Trabalhador** que

subscreva este método. Como consequência disso, as classes que possuem ligação direta não irão compilar até que as alterações para equiparar o tipo de retorno dos métodos subscritos a partir da superclasse sejam feitas em cada um deles. Já a composição nos fornece uma abordagem com um código mais fácil de mudar, pois ao contrário da herança, mudanças realizadas no código da classe final (antes classe mãe) não necessariamente resultariam em mudanças na classe de entrada (antes subclasse), tornando assim o encapsulamento mais forte.

Nas **Listagens 4**, **5** e **6** podemos ver uma amostra do benefício adquirido com o uso da composição. Na **Listagem 4** apresentamos o exemplo descrito anteriormente, que diz respeito à classe **Trabalhador**. Esta é herdada pela classe **Funcionario**, que tem como cliente a classe **ExemploUm**; e esta última, por sua vez, irá executar o programa para obter a quantidade de horas trabalhadas de determinado funcionário.

Listagem 4. Exemplo de uma classe usando herança.

```
public class Trabalhador {

    public int horasTrabalhadas() {
        system.out.println("Contabilizando horas trabalhadas ");
        return 8;
    }
}

public class Funcionario extends Trabalhador {
}

public class ExemploUm {

    public static void main(String[] args) {

        Funcionario funcionario = new Funcionario();
        int horas = funcionario.horasTrabalhadas();
    }
}
```

Ao executarmos o código da classe **ExemploUm** iremos obter como resposta a quantidade de horas e a mensagem impressa "*Contabilizando horas trabalhadas*". Isso porque **Funcionario** reutiliza o método **horasTrabalhadas()** ao herdá-lo de **Trabalhador**. Mas, e se em algum momento no futuro houver a necessidade de mudar o valor de retorno do tipo **int** do método **horasTrabalhadas()** para o tipo **Horas**? Isso resultaria em uma quebra de código da classe cliente **ExemploUm**, mesmo que sua referência direta seja apenas para com a classe **Funcionario**. Vejamos essa demonstração na **Listagem 5**.

Já na **Listagem 6** podemos ver como a composição provê uma alternativa para **Funcionario** no momento de obter a quantidade de horas trabalhadas. Ao invés de estender **Trabalhador**, **Funcionario** pode possuir uma referência para determinada instância de **Trabalhador** e definir seu próprio método **horasTrabalhadas()**.

Na abordagem da composição, a classe de "entrada", antiga subclasse, agora passa a ter sua própria implementação de método, que fará uma chamada explícita dele mesmo, porém, correspon-

Listagem 5. Exemplo da fragilidade no uso da Herança.

```
public class Horas {

    private int trabalhadas;

    public Horas(int trabalhadas) {
        this.trabalhadas = trabalhadas;
    }

    public int getTrabalhadas() {
      return trabalhadas;
    }
}

public class Trabalhador {

    public Horas horasTrabalhadas() {
        system.out.println("Contabilizando horas trabalhadas");
        return new Horas(8);
  }
}

//Funcionario compila e continua funcionando
public class Funcionario extends Trabalhador {
}

//Esta implementação não irá mais compilar
public class ExemploUm {

    public static void main(String[] args) {

        Funcionario funcionario = new Funcionario();
        int horas = funcionario.horasTrabalhadas();
    }
}
```

Listagem 6. Exemplo do uso da composição para favorecer LSP.

```
public class Horas {
    private int trabalhadas;
    public Horas(int trabalhadas) {
        this.trabalhadas = trabalhadas;
    }

    public int getTrabalhadas() {
        return trabalhadas;
    }
}

public class Trabalhador {

    public Horas horasTrabalhadas() {
        system.out.println("Contabilizando horas trabalhadas");
        return new Horas(8);
    }
}

public class Funcionario {

    private Trabalhador trabalhador = new Trabalhador();

    public int horasTrabalhadas() {
        Horas horas = trabalhador.horasTrabalhadas();
        return horas.getTrabalhadas();
    }
}

public class ExemploUm {
    public static void main(String[] args) {
        Funcionario funcionario = new Funcionario();
        int horas = funcionario.horasTrabalhadas();
    }
}
```

dente ao método da classe "final", antiga superclasse. Para essa chamada explícita do método damos o nome de "delegação". Essa abordagem provê um forte encapsulamento, pois alterações na classe final não necessariamente resultarão em quebra de código na classe de entrada.

Princípio da Segregação de Interfaces

No momento de projetar o design de alguma aplicação devemos ter bastante atenção com a criação dos módulos. Estes devem sempre ser os mais abstratos possíveis, pois à medida que o sistema evolui novos módulos se tornam necessários.

Diante da necessidade de estender a aplicação, haverá o caso de reaproveitar determinadas interfaces. No entanto, para isso, estas deverão ser consideravelmente abstratas, pois nossas classes clientes não devem ser forçadas a implementar métodos que não serão utilizados por elas. Quando uma interface propaga métodos utilizados por algumas classes, mas não aproveitados por outras, ocorre a subscrição forçada. A interfaces como estas damos o nome de interfaces poluídas.

É justamente disso que trata o *Princípio da Segregação de Interfaces* (ISP), o qual afirma que os clientes não devem ser obrigados a implementar interfaces que não utilizarão. Ao invés de uma interface poluída, o ideal seria fazer uso de várias interfaces baseadas em pequenos grupos de métodos, em que cada um deles atenderia a um determinado sub-módulo.

Para exemplificar a ideia do que seria uma interface poluída, observe a **Listagem 7**. Esse código nos mostra uma situação em que a classe **Gerente** é responsável por gerenciar trabalhadores de dois tipos: trabalhadores básicos e trabalhadores excepcionais, que são aqueles que conseguem desempenhar mais atividades no mesmo espaço de tempo que os básicos. Ambos os tipos, além de trabalharem, necessitam do seu devido intervalo para almoço. Porém a empresa evoluiu, e com isso surgiu a ideia de adicionar robôs para também atuarem como trabalhadores. No entanto, diferentemente dos demais trabalhadores, os robôs não precisam se alimentar.

Mediante o design apresentado, surge o seguinte problema: as classes referentes aos robôs também precisariam implementar a interface **ITrabalhador**, já que a mesma é o contrato que define o comportamento de todos os tipos de trabalhadores da empresa. Contudo, ao procedermos dessa forma, os robôs também iriam adquirir o método **comer()** sem necessidade.

Ao manter o design dessa forma, alguns comportamentos não desejados podem acontecer, como resultados incorretos apresentados por um relatório, ao analisar, por exemplo, a quantidade de refeições para os trabalhadores.

Para resolver o problema de design da **Listagem 7**, de acordo com a ISP, devemos separar a interface **ITrabalhador** em duas outras interfaces, tornando assim a implementação realmente flexível. Vejamos como fica o código modificado na **Listagem 8**.

No novo design foi criada a interface **IAlimentavel** para contemplar o método **comer()**, que antes pertencia à interface **ITrabalhador**. Agora, as classes referentes aos trabalhadores básicos e excepcionais

Listagem 7. Exemplo de violação da ISP.

```
public interface ITrabalhador {
    public void trabalhar();
    public void comer();
}
public class Trabalhador implements ITrabalhador {
    public void trabalhar() {
        // trabalhando...
    }

    public void comer() {
        // comendo...
    }
}
public class TrabalhadorExcepcional implements ITrabalhador {
    public void trabalhar() {
        // trabalhando...
    }

    public void comer() {
        // comendo...
    }
}
public class Gerente {
    ITrabalhador trabalhador;
    public void setTrabalhador(ITrabalhador trabalhador) {
        this.trabalhador = trabalhador;
    }
    public void gerenciar() {
        trabalhador.trabalhar();
    }
}
```

Listagem 8. Exemplo de design flexível ao utilizar ISP.

```
public interface ITrabalhador {
    public void trabalhar();
}

public interface IAlimentavel {
    public void comer();
}

public class Trabalhador implements ITrabalhador, IAlimentavel {

    public void trabalhar() {
        // trabalhando...
    }

    public void comer() {
      // comendo...
    }
}

public class TrabalhadorExcepcional implements ITrabalhador, IAlimentavel {

    public void trabalhar() {
        // trabalhando...
    }

    public void comer() {
        // comendo...
    }
}

public class Robo implements ITrabalhador {

    public void trabalhar() {
        // trabalhando...
    }
}
```

implementam duas interfaces: **ITrabalhador**, que ficou apenas com o método **trabalhar()**; e **IAlimentavel**, que possui apenas o método **comer()**. Quanto às classes dos robôs, elas só precisam implementar a interface **ITrabalhador**, pois estes só irão trabalhar.

A partir desse novo design a classe **Robo** não é mais forçada a implementar o método **comer()**, de forma que caso seja necessário criar alguma função específica para robôs, basta definir uma nova interface; por exemplo, **IRecarregavel**, para **recarregar()** os robôs após um longo dia de trabalho.

Princípio da Inversão de Dependências

A ideia deste princípio de design é representada através da concepção de que classes de alto nível não devem depender de classes de baixo nível, isto é, elas não devem estar ligadas de forma direta, mas sim através de alguma abstração.

Em todo desenvolvimento de aplicações, é considerado que classes de baixo nível são responsáveis por implementar operações primárias e básicas, como leitura de arquivos, acesso ao disco, uso de protocolos de rede, etc. Já as classes de alto nível são aquelas responsáveis por encapsular as lógicas complexas do sistema, como os fluxos do negócio. Nas arquiteturas de sistemas convencionais, componentes de baixo nível são criados para serem consumidos por componentes de alto nível, ou seja, as classes de mais alto nível são projetadas dependendo diretamente das de mais baixo nível para executar alguma tarefa.

Esta forte dependência com os componentes de baixo nível, no entanto, acaba por limitar as oportunidades de reutilização dos componentes de alto nível. Para evitar esse tipo de problema, uma nova camada abstrata pode ser introduzida entre os níveis, separando as classes de alto nível das de baixo nível. Vale lembrar, contudo, que a camada abstrata não deve ser criada a partir dos componentes de baixo nível, e sim o contrário: os componentes devem ser criados a partir da camada abstrata.

Ainda de acordo com este princípio, o caminho para um design de componentes estruturados corretamente seria iniciar o desenvolvimento a partir das classes de alto nível para as de baixo nível. Dessa forma teríamos: *Classes de Alto Nível* > *Camada Abstrata* > *Classes de Baixo Nível*.

O exemplo da **Listagem 9** mostra uma implementação que viola a ideia do IDP. Note que continuamos seguindo o mesmo caso dos trabalhadores de uma empresa. Neste código, temos a classe **Gerente** como sendo um componente de alto nível e a classe **Trabalhador** como sendo de baixo nível. No entanto, agora a empresa resolveu separar as classes trabalhadoras em trabalhadores comuns e trabalhadores excepcionais, e para isso será necessário criar uma nova classe, que chamaremos de **TrabalhadorExcepcional**.

Tomaremos como base que a classe **Gerente**, no exemplo proposto, seja a classe de alto nível, pois ela será responsável por interagir com todas as classes que representarão os diferentes tipos de trabalhadores. De acordo com a implementação apresentada na **Listagem 9**, teremos que alterar a classe **Gerente** para adicionar o novo tipo de trabalhador.

Listagem 9. Exemplo de violação do DIP.

```
public class Trabalhador {

      public void trabalhar() {
          //Trabalhando...
      }
}

public class Gerente {

      Trabalhador trabalhador;

      public void setTrabalhador(Trabalhador trabalhador) {
            this.trabalhador = trabalhador;
      }

      public void gerenciar() {
            trabalhador.trabalhar();
      }
}

public class TrabalhadorExcepcional {

    public void trabalhar() {
            //Trabalhar muito mais
      }
}
```

Listagem 10. Exemplo correto de uso do DIP.

```
public interface ITrabalhador {
      public void trabalhar();
}

public class Trabalhador implements ITrabalhador {
      public void trabalhar() {
            //trabalhando...
      }
}

public class TrabalhadorExcepcional implements ITrabalhador {
      public void trabalhar() {
            //trabalhando...
      }
}

public class Gerente {

      ITrabalhador trabalhador;

      public void setTrabalhador(ITrabalhador trabalhador) {
            this.trabalhador = trabalhador;
      }

      public void gerenciar() {
            trabalhador.trabalhar();
      }
}
```

Da forma como o design se encontra, para cada novo tipo de trabalhador uma nova referência será necessária, assim como uma nova chamada para o método **trabalhar()**, exclusivo de cada tipo de trabalhador, deverá ser criada dentro do método **gerenciar()** da classe **Gerente**.

Devido a essas alterações, podemos causar algum dano não previsto em alguma funcionalidade já existente da classe **Gerente**.

Nota

Ao respeitar o Princípio da Inversão de Dependência (DIP) é possível reduzir o acoplamento e criar objetos reutilizáveis de forma mais flexível.

Para resolver esse problema de design, devemos alterar essa classe para que ela deixe de trabalhar diretamente com a classe concreta **Trabalhador** e passe a ter uma interação mais abstrata. Para isso, criaremos a interface **ITrabalhador** e esta deverá ser implementada pela classe **Trabalhador**. A partir desta solução poderemos atuar com novos tipos de trabalhadores, bastando para isso que as novas classes também implementem **ITrabalhador**.

Assim, salvo que alguma alteração por parte da lógica de negócio seja precisa, nenhuma outra mudança será necessária em nossa classe **Gerente** quando novos trabalhadores forem adicionados. Vejamos como fica essa implementação na **Listagem 10**.

Um design de qualidade pode facilitar e muito a implementação de códigos reutilizáveis e coesos. No entanto, para que isso ocorra, também é importante que o código seja claro e desprovido de funcionalidades desnecessárias.

Para alcançar esse cenário, diversos são os conceitos que o desenvolvedor precisa compreender. Entre esses conceitos estão os princípios S.O.L.I.D., que certamente fazem grande diferença na hora de implementarmos o projeto com qualidade.

Autor

Marcio de S. Justo de Oliveira
marcio.justo@gmail.com
É formado em Criação e Gestão de Ambientes Internet pela faculdade Estácio de Sá. Trabalha com Java há oito anos, tendo desenvolvido sistemas para grandes clientes como Caixa Econômica Federal, Petrobras e Bradesco Seguros. Atualmente é desenvolvedor sênior na Capgemini – Consulting, Technology and Outsourcing.

Links:

Principles Of OOD.
http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

Artima Developer.
http://www.artima.com/designtechniques/

Desenvolvendo software sólido - Java Magazine 79.
http://www.devmedia.com.br/desenvolvendo-software-solido-java-magazine-79/16799

Livros
Introdução à Arquitetura e Design de Software, escrito por Paulo Silveira, Guilherme Silveira, Sérgio Lopes, Guilherme Moreira, Nico Steppat e Fábio Kung.

Programação Orientada a Objetos (POO): reusabilidade e eficiência em seu código – Parte 2

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

ESTE ARTIGO FAZ PARTE DE UM CURSO

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.

No primeiro artigo da série, introduzimos o tema Programação Orientada a Objetos e iniciamos a explicação sobre os conceitos básicas desse paradigma. Aprendemos o que são classes e objetos, conhecemos os modificadores de acesso e também o conceito de herança, usando classes e interfaces.

Neste artigo continuaremos nosso estudo sobre herança, agora através de classes abstratas, e depois aprenderemos sobre encapsulamento e polimorfismo, cobrindo assim os principais conceitos da Orientação a Objetos na prática com Java.

Herança usando classes abstratas

Classes abstratas são classes que não podem ser instanciadas diretamente e são definidas através da palavra chave **abstract**, empregada como modificador da classe. Além disso, uma classe abstrata se comporta de forma parecida com uma interface, pois uma classe abstrata pode conter métodos abstratos (sem implementação) usando ponto-e-vírgula no lugar do corpo do método e a palavra-chave **abstract** como modificador destes.

É importante notar que uma subclasse não abstrata de uma classe abstrata pode ser instanciada normalmente, pois ela é obrigada a implementar todos os métodos abstratos da classe mãe. Como exemplo de classe abstrata, é proposta a classe **Produto** na **Listagem 1**, que tem a função de manter o nome e o setor do produto, incluindo a quantidade em estoque.

Como pode-se observar na linha 3, a classe **Produto** é declarada como sendo abstrata pelo uso da palavra-chave **abstract**. Essa classe contém três atributos (**nome**, **setor** e **estoque** – linhas 5 a 7), de modo que os dois primeiros são informados no construtor (linhas 9 a 13), e o estoque inicia com zero. Os *getters* e *setters* para ambos os atributos são definidos nas linhas 15 a 29.

Já nas linhas 31 a 33 é definido o método **toString()**, que retorna uma descrição textual do produto, de forma a retornar uma mensagem com o **nome** e o **estoque**. Na linha 35 é declarado o método **ajustaEstoque()**, que pode ser utilizado pelas classes filhas para ajustar a variável **estoque**.

Ao final, na linha 39, é declarada a assinatura do método abstrato **entradaEstoque()**, e na linha 41, o método **saidaEstoque()**. Observe que ambos são declarados com a palavra-chave **abstract** e não contêm a declaração do corpo.

Listagem 1. Código da classe abstrata Produto.

```
01. package pkg.mercado;
02.
03. public abstract class Produto {
04.
05.  private String nome;
06.  private String setor;
07.  private int estoque;
08.
09.  public Produto(String nome, String setor) {
10.   this.nome = nome;
11.   this.setor = setor;
12.   this.estoque = 0;
13.  }
14.
15.  public String getNome() {
16.   return nome;
17.  }
18.
19.  public void setNome(String nome) {
20.   this.nome = nome;
21.  }
22.
23.  public String getSetor() {
24.   return setor;
25.  }
26.
27.  public void setSetor(String setor) {
28.   this.setor = setor;
29.  }
30.
31.  public String toString() {
32.   return "Produto nome = [" + nome + "] estoque = [" + estoque + "]";
33.  }
34.
35.  public void ajustaEstoque(int quantidade) {
36.   estoque = estoque + quantidade;
37.  }
38.
39.  public abstract void entradaEstoque(int quantidade);
40.
41.  public abstract void saidaEstoque(int quantidade);
42. }
```

Como a classe **Produto** não pode ser instanciada, ela não tem utilidade sozinha. Portanto, ela precisa ser especializada. Com o intuito de representar uma nova versão de **Produto**, criamos a classe **ProdutoFisico**, responsável por descrever qualquer produto na loja física (veja a **Listagem 2**).

Note que a classe **ProdutoFisico** estende **Produto** (linha 3) e define um construtor (linhas 5 a 7), informando para a superclasse os atributos **nome** e **setor** usando **super** (linha 6). O método **entradaEstoque()** é definido nas linhas 9 a 13, sendo herdado de **Produto**, e tem a função de adicionar produtos ao estoque e imprimir na tela uma descrição textual da operação.

De forma semelhante, o método **saidaEstoque()** – codificado nas linhas 15 a 19 – remove uma certa quantidade de produtos do estoque e imprime na tela uma descrição da operação. Para mostrar o funcionamento de **ProdutoFisico**, codificamos na **Listagem 3** um exemplo onde são criados dois produtos físicos distintos, em seguida é adicionada uma quantidade em estoque e depois retirada outra quantidade. Para facilitar a compreensão durante a execução, a cada passo é impresso na tela o estoque corrente.

Listagem 2. Especialização da classe abstrata Produto: ProdutoFisico.

```
01. package pkg.mercado;
02.
03. public class ProdutoFisico extends Produto {
04.
05.  public ProdutoFisico(String nome, String setor) {
06.   super(nome, setor);
07.  }
08.
09.  public void entradaEstoque(int quantidade) {
10.   ajustaEstoque(quantidade);
11.   System.out.println("Entrada de Estoque do Produto Físico " + getNome()
12.     + " quantidade = [" + quantidade + "]");
13.  }
14.
15.  public void saidaEstoque(int quantidade) {
16.   ajustaEstoque(-quantidade);
17.   System.out.println("Saída de Estoque do Produto Físico " + getNome()
18.       + " quantidade = [" + quantidade + "]");
19.  }
20. }
```

Listagem 3. Teste para as classes abstratas.

```
01. package pkg.mercado;
02.
03. public class TesteProdutos {
04.  public static void main(String[] args) {
05.
06.   ProdutoFisico p1 = new ProdutoFisico("Notebook", "Informática");
07.   ProdutoFisico q1 = new ProdutoFisico("Mesa", "Móveis");
08.
09.   p1.entradaEstoque(200);
10.   System.out.println(p1.toString());
11.   p1.saidaEstoque(50);
12.   System.out.println(p1.toString());
13.
14.   q1.entradaEstoque(1000);
15.   System.out.println(q1.toString());
16.   q1.saidaEstoque(500);
17.   System.out.println(q1.toString());
18.  }
19. }
```

Na linha 6 é criado o produto físico **notebook**. Em seguida, na linha 9, é dada uma entrada de estoque de 200 unidades, e por fim, na linha 11, são retiradas 50 unidades, levando a um total de 150 *notebooks*, o que é mostrado na **Listagem 4** nas quatro primeiras linhas.

Ainda na **Listagem 3**, linha 7, é criado um segundo produto, a **mesa**. Em seguida, na linha 14, é adicionado ao estoque 1000 mesas, e são retiradas 500 na linha 16, levando a um estoque final de 500 mesas, o que fecha com o total mostrado na **Listagem 4** nas quatro últimas linhas.

Encapsulamento

O conceito de encapsulamento é aplicado quando temos uma classe e queremos que certos atributos e métodos sejam manipuláveis apenas dentro da própria classe, de forma a esconder a

implementação interna desta, prevenindo a utilização incorreta da mesma por objetos externos. Para que isso seja possível, os métodos internos são definidos como sendo privados.

Além disso, é necessário implementar um conjunto de métodos que são acessíveis de fora da classe, formando uma espécie de interface de acesso externa. Tais métodos devem ser declarados como públicos e, portanto, acessíveis em qualquer classe da aplicação.

O encapsulamento é muito útil para esconder a implementação interna das classes e se encaixa muito bem na criação de bibliotecas de software, implicando na criação de uma API de acesso externa que esconde a complexidade do código.

Como exemplo de classe que segue o conceito de encapsulamento, foi apresentada a classe **Produto** na **Listagem 1**. Outra classe criada seguindo os conceitos de encapsulamento é a classe **Loja**, que tem sua primeira parte declarada na **Listagem 5**.

Listagem 4. Resultados no console para o teste envolvendo as classes Abstratas.

```
Entrada de Estoque do Produto Físico Notebook quantidade = [200]
Produto nome = [Notebook] estoque = [200]
Saída de Estoque do Produto Físico Notebook quantidade = [50]
Produto nome = [Notebook] estoque = [150]
Entrada de Estoque do Produto Físico Mesa quantidade = [1000]
Produto nome = [Mesa] estoque = [1000]
Saída de Estoque do Produto Físico Mesa quantidade = [500]
Produto nome = [Mesa] estoque = [500]
```

Listagem 5. Primeira parte da classe Loja.

```
01. package pkg.mercado;
02.
03. import java.util.ArrayList;
04.
05. public class Loja extends PessoaJuridica {
06.
07.  private String    apelidoLoja;
08.  private ArrayList funcionarios = new ArrayList();
09.  private ArrayList clientes    = new ArrayList();
10.
11.  public Loja(String nome, boolean habilitado, long CNPJ) {
12.   super(nome, habilitado, CNPJ);
13.  }
```

Repare que a classe **Loja** possui três atributos que seguem as regras do encapsulamento, **apelidoLoja**, **funcionarios** e **clientes**, declarados privados (linhas 7 a 9). Como **Loja** é uma **PessoaJuridica**, ela recebe no seu construtor os atributos **nome**, **habilitado** e **CNPJ** de **PessoaJuridica** (que são repassados ao construtor da superclasse) e seu atributo **apelidoLoja**. Na **Listagem 6** é apresentada a segunda parte da classe **Loja**.

Nesse código encontramos os métodos da API da classe **Loja**, invocados no acesso externo às funcionalidades desta classe. Primeiro temos o *getter* e o *setter* do atributo **apelidoLoja** e em seguida os métodos públicos **adicionarFuncionario()** e **adicionarCliente()**. Por último, temos o método público **realizarVenda()**, que recebe como parâmetro um vendedor, um cliente e um produto e chama os métodos privados **adicionarVenda()**, **calcularComissaoVendedor()** e **atualizarEstoque()**, declarados na **Listagem 7**.

Listagem 6. Segunda parte da classe Loja.

```
14.  public String getApelidoLoja() {
15.   return apelidoLoja;
16.  }
17.
18.  public void setApelidoLoja(String apelidoLoja) {
19.   this.apelidoLoja = apelidoLoja;
20.  }
21.
22.  public void adicionarFuncionario(Funcionario funcionario) {
23.   funcionarios.add(funcionario);
24.  }
25.
26.  public void adicionarCliente(Cliente cliente) {
27.   clientes.add(cliente);
28.  }
29.
30.  public void realizarVenda(Vendedor vendedor, Cliente cliente,
31.      Produto produto) {
32.   adicionarVenda(cliente);
33.   calcularComissaoVendedor(vendedor);
34.   atualizarEstoque(produto);
35.  }
```

Listagem 7. Terceira parte da classe Loja.

```
36.  private void adicionarVenda(Cliente cliente) {
37.   System.out.println("Compra adicionada para o cliente "
38.      + cliente.getNome());
39.  }
40.
41.  private void calcularComissaoVendedor(Vendedor vendedor) {
42.   System.out.println("Comissão calculada para o vendedor "
43.      + vendedor.getNome());
44.  }
45.
46.  private void atualizarEstoque(Produto produto) {
47.   System.out.println("Atualização de estoque para o produto "
48.      + produto.getNome());
49.  }
50. }
```

Observe que os métodos apresentados nessa listagem são privados, e, portanto pertencem à implementação interna da classe.

Polimorfismo

Quando são definidas várias especializações diferentes para uma mesma superclasse, ocorre que cada especialização pode reimplementar os métodos da superclasse de uma forma diferente. Assim, podemos ter uma coleção de objetos, mas cada um deles não precisa necessariamente implementar da mesma forma as operações da superclasse.

Dito isso, codificamos a classe **ProdutoVirtual** (**Listagem 8**), que implementa os métodos **entradaEstoque()** e **saidaEstoque()**, provenientes de **Produto** (**Listagem 1**).

Além disso, **ProdutoVirtual** contém um atributo chamado **link** (linha 5) e define o *getter* e o *setter* correspondentes (linhas 12 a 17), completando a declaração da classe.

O conceito de polimorfismo está diretamente relacionado à sobrecarga de métodos (*overloading* ou sobrescrita), que significa que uma classe filha pode sobrescrever a implementação dos métodos da classe mãe, permitindo aos objetos da classe

filha apresentarem uma implementação diferente nos métodos sobrecarregados.

Como exemplo, a classe **CreditoVirtual**, que representa uma moeda virtual, sobrescreve os métodos **entradaEstoque()** e **saidaEstoque()** de **ProdutoVirtual** na **Listagem 9**, especializando-os.

Listagem 8. Código da classe ProdutoVirtual.

```
01. package pkg.mercado;
02.
03. public class ProdutoVirtual extends Produto {
04.
05.  public String link;
06.
07.  public ProdutoVirtual(String nome, String setor, String link) {
08.   super(nome, setor);
09.   this.link = link;
10.  }
11.
12.  public String getLink() {
13.   return link;
14.  }
15.
16.  public void setLink(String link) {
17.   this.link = link;
18.  }
19.
20.  public void entradaEstoque(int quantidade) {
21.   ajustaEstoque(quantidade);
22.   System.out.println("Adição de Seriais de Produto Virtual " + getNome()
23.     + " quantidade = [" + quantidade + "]");
24.  }
25.
26.  public void saidaEstoque(int quantidade) {
27.   ajustaEstoque(-quantidade);
28.   System.out.println("Saída de Seriais de Produto Virtual " + getNome()
29.     + " quantidade = [" + quantidade + "]");
30.  }
31. }
```

Listagem 9. Código da classe CreditoVirtual.

```
01. package pkg.mercado;
02.
03. public class CreditoVirtual extends ProdutoVirtual {
04.
05.  public double valor;
06.
07.  public CreditoVirtual(String nome, String setor, String link, double valor) {
08.   super(nome, setor, link);
09.   this.valor = valor;
10.  }
11.
12.  public double getValor() {
13.   return valor;
14.  }
15.
16.  public void setValor(double valor) {
17.   this.valor = valor;
18.  }
19.
20.  public void entradaEstoque(int quantidade) {
21.   ajustaEstoque(quantidade);
22.   System.out.println("Adição de Créditos Virtuais " + getNome()
23.     + " quantidade = [" + quantidade + "]");
24.  }
25.
26.  public void saidaEstoque(int quantidade) {
27.   ajustaEstoque(-quantidade);
28.   System.out.println("Pagamento de Créditos Virtuais " + getNome()
29.     + " quantidade = [" + quantidade + "]");
30.  }
31. }
```

Note que nas linhas 20 a 30 dessa listagem é apresentada uma nova implementação dos métodos originários da classe **Produto**, ou seja, ocorre o *overloading*. A fim de mostrar o funcionamento do polimorfismo, é criada uma classe executável na **Listagem 10** que instancia cada uma das subclasses de **Produto** (introduzidas anteriormente) e realiza a entrada e saída de estoque para cada instância.

Listagem 10. Código da classe de teste TesteProdutosPoli.

```
01. package pkg.mercado;
02.
03. public class TesteProdutosPoli {
04.  public static void main(String[] args) {
05.
06.    ProdutoFisico p1 = new ProdutoFisico("Notebook", "Informática");
07.    ProdutoFisico q1 = new ProdutoFisico("Mesa", "Móveis");
08.    ProdutoVirtual r1 = new ProdutoVirtual("Editor de Imagens",
09.        "Loja Virtual", "www...");
10.    CreditoVirtual s1 = new CreditoVirtual("Editor de Texto",
11.        "Loja Virtual", "www...", 100);
12.
13.    p1.entradaEstoque(100);
14.    q1.entradaEstoque(200);
15.    r1.entradaEstoque(300);
16.    s1.entradaEstoque(400);
17.
18.    vendaProduto(p1);
19.    vendaProduto(q1);
20.    vendaProduto(r1);
21.    vendaProduto(s1);
22.  }
23.
24.  private static void vendaProduto(Produto p) {
25.    p.saidaEstoque(1);
26.    System.out.println(p.toString());
27.  }
28. }
```

Nesse exemplo são criados diversos produtos, sendo dois produtos físicos (linhas 6 e 7), um produto virtual (linha 8) e um produto de crédito virtual (linha 10). Em seguida, ocorre a entrada no estoque de uma quantidade distinta para cada produto (linhas 13 a 16), chamando o método **entradaEstoque()**. Por fim, ocorre a venda de uma unidade de cada um desses produtos criados (linhas 18 a 21), o que é feito chamando o método **vendaProduto()**, declarado na classe exemplo (**TesteProdutosPoli**).

O método **vendaProduto()** recebe um objeto da classe **Produto** como parâmetro para efetuar a venda de uma unidade dele (linha 25) e escreve o resultado na tela (linha 26). Como o parâmetro **produto** pode ser qualquer especialização da classe **Produto**, é importante observar que ele pode ser pode ser filho de **ProdutoFisico**, **ProdutoVirtual** ou **CreditoVirtual**. O resultado da execução desse código é mostrado na **Listagem 11**.

Podemos constatar nas quatro primeiras linhas a adição de cada produto ao estoque feita de uma maneira particular, usando o método **entradaEstoque()**. Nas linhas restantes ocorre a saída de cada produto, feita também de uma maneira particular (de acordo com cada implementação de **Produto**), usando o método **vendaProduto()**.

Listagem 11. Resultado da execução do exemplo da **Listagem 10**.

```
Entrada de Estoque do Produto Físico Notebook quantidade = [100]
Entrada de Estoque do Produto Físico Mesa quantidade = [200]
Adição de Seriais de Produto Virtual Editor de Imagens quantidade = [300]
Adição de Créditos Virtuais Editor de Texto quantidade = [400]
Saída de Estoque do Produto Físico Notebook quantidade = [1]
Produto nome = [Notebook] estoque = [99]
Saída de Estoque do Produto Físico Mesa quantidade = [1]
Produto nome = [Mesa] estoque = [199]
Saída de Seriais de Produto Virtual Editor de Imagens quantidade = [1]
Produto nome = [Editor de Imagens] estoque = [299]
Pagamento de Créditos Virtuais Editor de Texto quantidade = [1]
Produto nome = [Editor de Texto] estoque = [399]
```

Uma última observação é que se quisermos fazer com que um método não possa mais ser sobrescrito, pode-se utilizar a palavra-chave **final** como modificador do método. Além disso, **final** pode ser utilizado também como modificador de classe, para evitar que uma classe seja especializada, impedindo assim a relação de herança para novas classes.

Os conceitos da orientação a objetos são muito úteis para alavancar o desenvolvimento em projetos de software, pois em comparação com outros paradigmas, possibilitam maior reusabilidade de código, manutenção simplificada, facilidade para o crescimento da aplicação, entre outros. Enfim, a correta compreensão e utilização da herança, do encapsulamento e do polimorfismo leva a uma programação mais eficiente e simples, viabilizando um código limpo, o que é bom para você, para os desenvolvedores da empresa em que você atua e para o cliente.

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/

Desenvolva aplicações com Play! Framework

Aprenda a implementar sistemas web com qualidade e facilidade

Atualmente, obter informações precisas e em um curto espaço de tempo se tornou indispensável para que pequenas e grandes corporações sobrevivam e se mantenham atualizadas em relação à forte concorrência do mercado. Com isso, a tecnologia tem se tornado cada vez mais presente nas empresas, trazendo inovações que possibilitam suprir tal necessidade. Neste contexto, entre as inovações mais recentes e de grande destaque se encontra a tecnologia WebSocket.

WebSocket é um recurso que permite a comunicação em tempo real entre cliente e servidor através de uma única conexão TCP. A conexão é mantida durante todo o tempo e ambos podem se comunicar quantas vezes for necessário, não necessitando de solicitações adicionais (os famosos requests), resultando assim em ganho de performance para a aplicação. Geralmente esta solução é adotada em aplicações que requerem troca de informações e atualizações em tempo real, como é o caso de uma transmissão online, que ao inserir uma nova postagem precisa que esta seja notificada a todos instantaneamente, sem precisar que cada um dos clientes solicite esta atualização.

Como as corporações possuem a necessidade de obter soluções ágeis e com bom desempenho, os desenvolvedores também possuem a uma importante necessidade: dispor de boas soluções que permitam desenvolver projetos com praticidade e produtividade. Neste contexto surge o Play Framework, outra tecnologia que iremos abordar no decorrer do artigo.

O Play! vem chamando a atenção dos desenvolvedores justamente por facilitar o desenvolvimento de aplicações web em Java, não necessitando de muitas ferramentas e configurações complexas para a implementação da solução, nem mesmo sendo necessária a utilização de uma IDE. Além disso, o Play consome recursos mínimos de hardware, possibilitando assim seu uso em equipamentos inferiores. Dentre as diversas vantagens de se utilizar este framework, podemos destacar:

Fique por dentro

Neste artigo apresentaremos os recursos do simples e prático Play Framework. Para isso, demonstraremos na prática a criação de um chat, e como um importante complemento, utilizaremos conjuntamente as soluções do protocolo WebSocket para realizar a comunicação em tempo real entre a aplicação e o servidor.

- A facilidade na detecção e resolução de erros durante o processo de desenvolvimento;
- A possibilidade de utilização de qualquer biblioteca Java;
- O uso da arquitetura MVC/RESTful.

Sendo assim, com o intuito de demonstrar a praticidade de se desenvolver com os recursos fornecidos pelo Play!, e também visando de abordar um exemplo onde possamos fazer uso de WebSockets, desenvolveremos no decorrer deste artigo um chat.

Play Framework

O Play! é um framework open source utilizado para o desenvolvimento de aplicações Java e Scala. O seu objetivo é facilitar a implementação de soluções web visando a produtividade, para que todo processo seja ágil e o menos desgastante possível. Como grande vantagem está o fato de trabalhar com o processo de Hot Deployment, que possibilita ao desenvolvedor visualizar as alterações sem a necessidade de recompilar o projeto manualmente. Com Hot Deployment, o tempo gasto para recompilar o projeto e executá-lo novamente se resume apenas a um simples refresh na página. Outra característica fornecida por este framework é a possibilidade de se trabalhar com o sistema de rotas, que simplifica o mapeamento entre URLs e os controladores da aplicação.

O que é WebSocket?

Normalmente, quando uma aplicação web é acessada por um browser, uma solicitação HTTP é enviada para o servidor web responsável por hospedar a página. Em seguida, o servidor web analisa o pedido e envia a resposta. Em alguns casos, o tempo de resposta é excedido devido à grande quantidade de informações

a serem processadas, causando lentidão e transtornos (ver **Figura 1**).

O objetivo do WebSocket é viabilizar a comunicação assíncrona entre servidores e aplicações web que necessitam de informações em tempo real, como sites de previsões do tempo, sites de ações e investimentos, jogos online, etc.

Com o uso desta solução a comunicação se torna mais prática e com menores chances de causar lentidão, pois é utilizado um canal de comunicação bidirecional entre navegador e servidor, e então ambos podem enviar e receber diversas mensagens entre eles sem ter a necessidade de enviar múltiplas requisições.

Ao adotar o tradicional protocolo HTTP, dependendo da complexidade da aplicação, é necessário o envio e recebimento de diversas requisições, seja pela aplicação ou servidor. Além disso, realizar várias requisições resulta em queda de desempenho da aplicação e do servidor, exigindo maior processamento de informações. Com a utilização da API WebSocket, padronizada pela W3C (*Word Wide Web Consortium*), abre-se a possibilidade de implantarmos os recursos do protocolo WebSocket no desenvolvimento de soluções web, reduzindo assim consideravelmente o número de requisições trocadas.

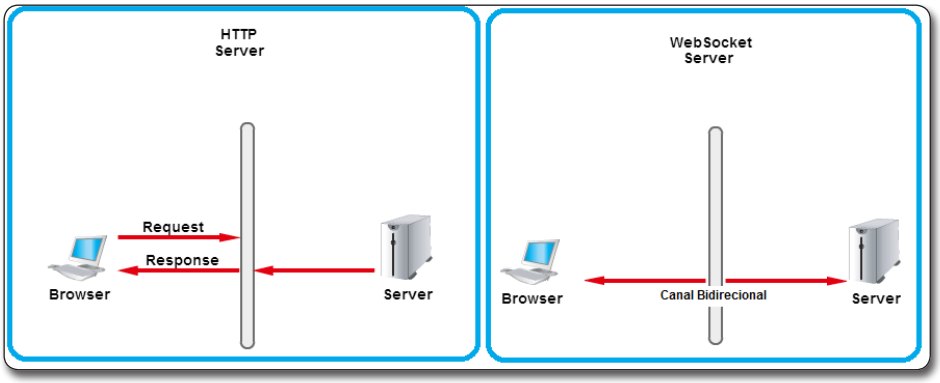


Figura 1. Comunicação entre cliente e servidor com HTTP e WebSocket

A aplicação exemplo

No decorrer do artigo iremos abordar o desenvolvimento de um chat. Para isso, vamos criar duas páginas HTML: a primeira sendo uma tela de login para que o usuário possa se autenticar e entrar na sala do chat, e a segunda para exibir esta sala, na qual o usuário poderá se comunicar com outros membros. No processo de autenticação da página inicial iremos realizar uma simples consulta no banco de dados para averiguar se as informações repassadas aos campos de login constam na tabela de usuários. Desta forma, precisamos configurar a conexão com o banco de dados (neste caso o MySQL) e adicionar a sua dependência ao arquivo *build.sbt* para que o Play! consiga se comunicar com ele. Também será necessário criar a classe que fará referência à tabela onde estão armazenados os registros de login.

Com o Play!, é disponibilizado o banco de dados H2, que fica armazenado internamente no framework. Porém, com a adoção desta opção, a cada vez que iniciarmos a aplicação as informações nele contidas serão perdidas, pois este banco é útil para ambientes de teste e como teremos a necessidade que as informações permaneçam armazenadas, adotaremos o MySQL.

Concluída essa etapa, começaremos a desenvolver a sala do chat. Para esta tarefa criaremos as classes **Chat**, **SalaChat** e **Login**, que terão suas funcionalidades explicadas mais adiante. Além disso, para realizar alguns procedimentos em nosso chat, como iniciar a seção do websocket e enviar mensagens para os usuários ao teclar *Enter*, utilizaremos recursos do JavaScript.

Inicialmente, é preciso que você configure seu ambiente de desenvolvimento com a versão 6 ou superior do Java e navegadores que suportem o protocolo WebSocket. Consulte na seção **Links** o site para verificar se o seu navegador é compatível.

Caso seu ambiente de desenvolvimento já esteja pronto, o próximo passo é realizar a instalação do Play! Para isso, basta baixar o framework no site oficial do projeto (veja o endereço na seção **Links**), extraí-lo em qualquer diretório e adicionar o caminho onde foi descompactado na variável *PATH*.

Criando o projeto Play!

Diferentemente de outras tecnologias que necessitam de uma IDE para que seja possível o desenvolvimento de um projeto, o Play! possibilita a criação da aplicação com a execução de apenas um comando a partir do Play Console, que vem juntamente com o pacote do framework.

Nota

O arquivo build.sbt contém instruções para o gerenciamento de um projeto com o Play! Framework, dentre elas; diretivas de versão e dependências para recursos adicionais, como por exemplo, a dependência do banco de dados MySQL.

Apesar disso, e devido à importância das IDEs para os projetos, o Play! oferece suporte ao NetBeans, Eclipse e IntelliJ IDEA. Assim, caso o desenvolvedor queira trabalhar com os recursos do Play! em uma destas IDEs, basta digitar no console os comandos *play eclipsify nomeProjeto* para adaptar o projeto ao Eclipse, *play netbeansify nomeProjeto* para adaptar ao NetBeans e *play idealize nomeProjeto* para adaptá-lo ao IntelliJ IDEA.

Dito isso, para criar o projeto, abra a linha de comando, acesse o diretório no qual foi extraído o framework, digite o comando *play new* juntamente com o nome do projeto e, em seguida, escolha a opção 2 (*Java Application*) para definir o tipo do projeto, conforme demonstra a **Figura 2**.

Quando criamos um novo projeto, é definida uma estrutura de diretórios para armazenar os arquivos da aplicação. Com o Play! Framework, o modelo MVC é definido como padrão. Assim, ao navegar pelos diretórios do projeto, na pasta *app* encontraremos três subdiretórios: *controllers*, *models* e *views*, responsáveis por armazenar os principais arquivos da aplicação. Tais arquivos irão definir toda a regra e a parte visual da aplicação.

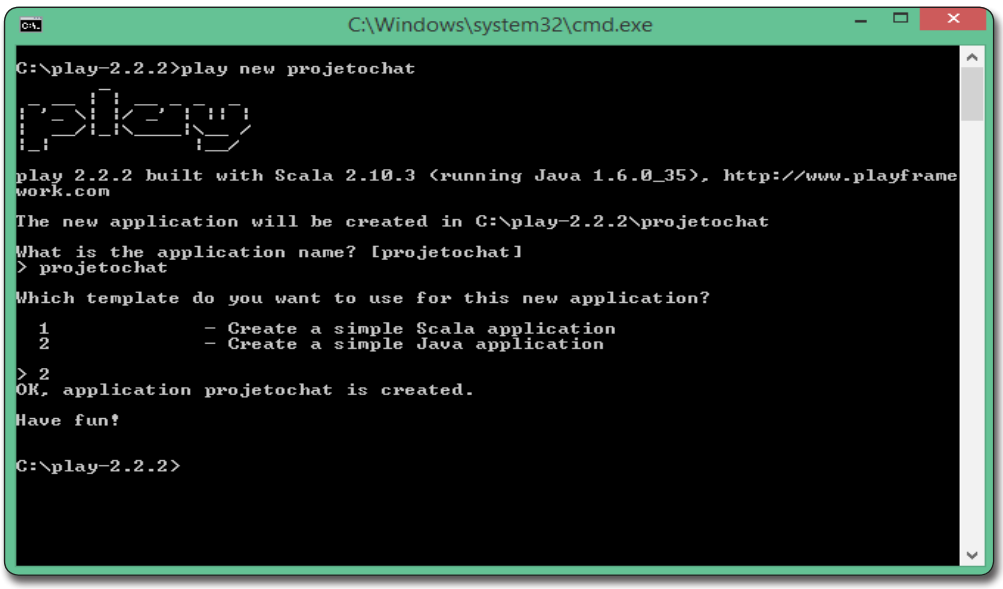


Figura 2. Comandos necessários para criação do projeto

Nesta organização, a parte visual deve ser armazenada no diretório *views*, as classes Java que contêm regras de negócio devem ser armazenadas na pasta *model* e, por último, na pasta *controller* estarão presentes os controladores que realizam a comunicação entre *model e view*.

Outro diretório importante e sobre o qual devemos compreender é o *conf*. Este armazena o arquivo *application.conf*, local onde são especificadas algumas configurações do sistema, tais como dados de conexão com o banco de dados, e o arquivo *routes*, que guarda a declaração das rotas, ou seja, neste arquivo são definidas todas as URLs que poderão ser acessadas pela camada view.

O arquivo *routes* por padrão já vem com duas rotas definidas, a saber:

1) a rota responsável por encaminhar a requisição do usuário ao acessar a aplicação para o método **index()**, localizado na classe **Application**, que também é criada automaticamente quando criamos o projeto Play!; e,

2) a rota que indica o caminho para a pasta *public*, local onde são armazenados arquivos de imagem, CSS e JavaScript.

Caso a sua tela esteja semelhante à **Figura 2**, já é possível acessar a aplicação pelo navegador. Assim, após executar o comando *play run* no Play! Console, acesse pelo navegador o link *localhost:9000/*. Caso apareça a mensagem "*Your new application is ready*", é sinal de que sua aplicação está em pleno funcionamento e portanto podemos dar continuidade ao desenvolvimento do nosso chat.

Estabelecendo a conexão com o banco de dados

Após instalar o Play! e criar a estrutura do projeto conforme explicado nos tópicos anteriores, localize o arquivo *build.sbt* no diretório raiz do projeto e insira a dependência do conector MySQL, conforme apresenta a **Listagem 1**. Com a dependência instalada, o Play! interpretará que a sua aplicação necessita deste conector e o instalará quando o projeto for executado.

Feito isso, o próximo passo é localizar o arquivo *application.conf* e adicionar as informações de conexão com o banco de dados, como driver de conexão, URL do JDBC, usuário e senha, conforme a **Listagem 2**.

Criando a classe de Login

Informados os dados de conexão no arquivo *application.conf*, criaremos agora a classe **Login**, que irá estender a classe **Model** para poder utilizar os recursos de persistência e consulta com o banco de dados que são disponibilizados pelo ebean (**BOX 1**). A classe **Login** irá conter todos os campos da tabela na qual iremos buscar as informações do usuário que deseja se conectar ao chat. Sendo assim, nesta classe vamos definir três atributos: **id**, **nome** e **senha**. Após isso, com o **Model.Finder**, recurso que faz parte da classe **Model** (linhas 23 e 24), criaremos o objeto **find**, que será utilizado para auxiliar na consulta. A **Listagem 3** apresenta o código da classe **Login**.

Listagem 1. Código do arquivo build.sbt.

```
01. name := "projetochat"
02.
03. version := "1.0-SNAPSHOT"
04.
05. libraryDependencies ++= Seq(
06.  javaJdbc,
07.  javaEbean,
08.  cache,
09.  "mysql" % "mysql-connector-java" % "5.0.8" )
11.
12. play.Project.playJavaSettings
```

Listagem 2. Adicionando informações de conexão com o banco de dados no arquivo application.conf.

```
01. db.default.url="jdbc:mysql://localhost/dbplay"
02. db.default.user=usuario
03. db.default.pass="senha"
04. db.default.logStatements=true
05. ebean.default="models.*"
```

BOX 1. Ebean

É um gerenciador de objetos relacionais utilizado pelo Play! e que adere o JPA como padrão. Para saber mais detalhes, verifique na seção **Links** a documentação do ebean.

Criando a sala do chat

Neste passo, iniciaremos a construção da sala do chat, codificando para isso a classe **SalaChat**, que será a estrutura principal do nosso projeto. Nela, iremos implementar o método responsável por incluir o usuário no chat e o método que realizará o envio de mensagens ao bate-papo. A **Listagem 4** exibe o código dessa classe.

Nessa listagem, utilizaremos as classes **UntypedActor**, **ActorSystem** e **ActorRef** providas pela API de concorrência Akka, que também é propriedade da TypeSafe, mesma empresa detentora dos direitos do Play. Esta API de concorrência adere ao modelo de atores (**BOX 2**), que visa aumentar o desempenho da aplicação e melhorar o tempo de resposta do sistema para o usuário.

BOX 2. Atores

Resumidamente, atores são definidos pela própria documentação do Akka como entidades que efetuam a comunicação entre si através da troca de mensagens.

Ao criarmos a classe **SalaChat**, estendemos a classe abstrata **UntypedActor** e, desta forma, possibilitamos a criação de um ator. Em seguida, criamos de fato o ator, na linha 3. Para isso, instanciamos a classe **ActorSystem**, que tem como objetivo criar e armazenar os atores, e logo após o inicializamos, através da chamada ao método **actorOf()**, na linha 4.

Feito isso, instanciamos o objeto **Map** para armazenar todos os usuários conectados no chat, definimos o método **entrarChat()**, responsável por viabilizar a entrada dos membros na sala e pelo envio de uma mensagem informando que determinado usuário se conectou.

Listagem 3. Código da classe Login.

```
01. package models;
02.
03. import javax.persistence.*;
04. import play.data.validation.Constraints;
05. import play.db.ebean.Model;
06.
07. @Entity
08. @Table(name = "login")
09. public class Login extends Model {
10.
11.    private static final long serialVersionUID = 1L;
12.
13.    @Id
14.    @Column(name = "id")
15.    private Long id;
16.
17.    @Column(name = "name")
18.    private String name;
19.
20.    @Column(name = "senha")
21.            private String senha;
22.
23.  public static Model.Finder<Long, Login> find = new Model.Finder
       <Long, Login>(
24.    Long.class, Login.class);
25.
26. }
```

Listagem 4. Código da classe SalaChat.

```
01. public abstract class SalaChat extends UntypedActor {
02.
03.    private static ActorSystem system = ActorSystem.create("chat");
04.    private static ActorRef defaultRoom = system.actorOf(Props.create
       (SalaChat.class), "chat");
05.
06.  private static Map<String, WebSocket.Out<JsonNode>>
     listaDeMembros = new HashMap<String,
07.   WebSocket.Out<JsonNode>>();
08.
09.
10.    public static void entrarChat(final String username,
       WebSocket.In<JsonNode> in,
11.          WebSocket.Out<JsonNode> out) throws Exception{
12.
13.       final DateFormat dateFormat = new SimpleDateFormat
          ("dd/MM/yyyy HH:mm");
14.       final Date date = new Date();
15.
16.    Acesso acs = new Acesso(username, out);
17.    Object obj = (Object)ask(defaultRoom, joined, 1000);
18.
19.    listaDeMembros.put(acs.username, acs.channel);
20.
21.    enviaMensagemTodos(acs.username, "Entrou no Chat. "
        + " - " + dateFormat.format(date));
22.
23.    in.onMessage(new Callback<JsonNode>() {
24.
25.    public void invoke(JsonNode event)            {
26.
27.   Conversa talkInstance = new Conversa(username ,event.get("text").asText() );
28.
29.
30.  enviaMensagemTodos (username, talkInstance.getMensagem() + " -
      "  + dateFormat.format(date));
31.
32.            }
33.          });
34.
35.       }
36
37.     public static void enviaMensagemTodos(String user, String text) {
38.      for(WebSocket.Out<JsonNode> lista: listaDeMembros.values()) {
39.          ObjectNode evento = Json.newObject();
40.          evento.put("user", user);
41.          evento.put("message", text);
42.          lista.write(evento);
43.       }
44.    }
45. }
```

Definimos também o método **enviaMensagemTodos()**, entre as linhas 47 e 44. Como o próprio nome indica, ele serve para enviar mensagens a todos os membros do chat.

Com a classe do ator definida na camada Model, nosso próximo passo é criar a classe **Chat** no diretório *app/controller* e nela escrevermos alguns métodos que serão executados diretamente da camada view. A **Listagem 5** apresenta o código da classe **Chat**.

O método **login()** terá o objetivo de abrir a página de login quando solicitado pela camada view. O método **abreSalaChat()**, por sua vez, irá exibir a sala do chat após o usuário logar no sistema.

Para que seja possível estabelecer a comunicação entre a camada View e a camada Controller, é preciso que os links que

Listagem 5. Código da classe Chat

```
package controllers;

package controllers;

import play.mvc.*;
import com.fasterxml.jackson.databind.JsonNode;
import views.html.*;
import models.*;
import play.api.data.Form.*;
import play.data.Form;
import views.html.helper.form;
import views.html.*;

public class Chat extends Controller {


  private static final Form<Login> loginForm = Form.form(Login.class);

  public static Result login() {
    return ok(login.render(loginForm));
  }

  public static Result abreSalaChat(String username) {
    return ok(chatRoom.render(username));

  }

  public static Result consultar() {
    Form<Login> form = loginForm.bindFromRequest();
    Login log = form.get();
    Login newLogin = new Login();

    if (!log.getName().equals(null) && !log.getSenha().equals(null)) {
      newLogin = Login.find.where().eq("name", log.getName())
          .eq("senha", log.getSenha()).findUnique();
    }

    if (log.getName().equals(newLogin.getName())
        && log.getSenha().equals(newLogin.getSenha())) {

      return Chat.abreSalaChat(log.getName());

    } else {

      return Chat.login();
    }

  }

  public static Result chatRoomJs(String username) {
    return ok(views.js.chatRoom.render(username));

  }

  public static WebSocket<JsonNode> entrarChat(final String username) {
    return new WebSocket<JsonNode>() {

      public void onReady(WebSocket.In<JsonNode> in,
          WebSocket.Out<JsonNode> out) {

        try {
          ChatRoom.entrarChat(username, in, out);
        } catch (Exception ex) {
          ex.printStackTrace();
        }
      }
    };
  }
}
```

irão chamar os métodos estejam devidamente configurados no arquivo *routes.conf*, o qual explicamos no início do artigo. Seu conteúdo é apresentado na **Listagem 6**.

Por último, o método **entrarChat()** é invocado quando qualquer usuário entra na sala chat. É no momento em que este método é acionado que o WebSocket é aberto. Neste momento, ressaltamos que a comunicação entre as camadas View e a classe **Chat** só será possível se a rota for declarada no arquivo *routes.conf*, conforme exibe a **Listagem 6**.

Listagem 6. Código do arquivo routes.conf

```
01. # Routes
02. # This file defines all application routes (Higher priority routes first)
03.
04. GET  / controllers.Chat.login()
05. GET  /room/inicializachat          controllers.Chat.inicializaChat(username)
06. GET  /room/abresalachat            controllers.Chat.abreSalaChat(username)
07. GET  /assets/javascripts/chatroom.js  controllers.Chat.chatRoomJs(username)
08. POST /consultalogin                                  controllers.Chat.consultar()
09.
10.
11. # Map static resources from the /public folder to the /assets URL path
12. GET    /assets/*file  controllers.Assets.at(path="/public", file)
```

No código apresentado estão especificadas as rotas, ou seja, são definidos os métodos que serão acionados quando houver uma solicitação da camada View a determinada URL. Em cada linha configuramos primeiramente se a chamada será realizada via GET ou POST, em seguida nomeamos a rota e por último especificamos o método que será executado ao acessar a rota declarada. Como exemplo de uso das rotas, no código da página *salachat.html*, que será apresentado mais adiante, utilizamos a rota descrita na linha 5 para a inicialização do chat.

Criando o layout da aplicação

Conforme mencionado, a parte visual do nosso chat será elaborada em duas telas. Na primeira delas teremos a tela de login, e na segunda teremos o chat propriamente dito. As **Listagens 7** e **8** apresentam o código das páginas de *login.html* e *salachat.html*, respectivamente.

No código da **Listagem 7**, definimos na linha 1 que a página de login será um formulário, e para a construção desse formulário informamos a classe **Login**, que contém os atributos que serão necessários, no caso: **id**, **nome** e **senha**. Deve-se informar a classe **Login** porque ao invocar o método **consultar()**, na linha 5, será feita a pesquisa dos dados informados nos campos de login (linhas 12 e 13) e assim verificar se estes constam no banco de dados. Ainda na linha 5, note que utilizamos a anotação **@helper** para indicar que o método **consultar()**, localizado em **Chat**, deverá ser acionado no momento em que o usuário pressionar o botão para entrar no chat, e nas linhas 12 e 13 utilizamos esta mesma anotação para especificar o tipo de dados para os campos de nome e senha. A página de login é apresentada na **Figura 3**.

Por sua vez, a **Listagem 8** apresenta o código da página *salachat.html*. Nela adicionamos o componente **textarea**, no qual o usuário poderá digitar o que deseja enviar aos demais membros do chat.

Listagem 7. Código da página login.html.

```
01.  @(loginForm : Form[models.Login])
02.
03.  <div>
04.    <div>
05.        @helper.form(action=routes.Chat.consultar()) {
06.      <form>
07.        <fieldset>
08.          <legend>Login</legend>
09.          <div>
10.            <div>
11.
12.                    @helper.inputText(loginForm("name"), '_label -> "Nome: ")
13.                @helper.inputPassword(loginForm("senha"), '_label -> "Senha: ")
14.
15.            </div>
16.          </div>
17.          }
18.
19.          <div>
20.            <label for="singlebutton"></label>
21.
22.            <div>
23.             <button id="singlebutton" name="singlebutton">Entrar</button>
24.            </div>
25.          </div>
26.      </form>
27.    </div>
28.  </div>
```

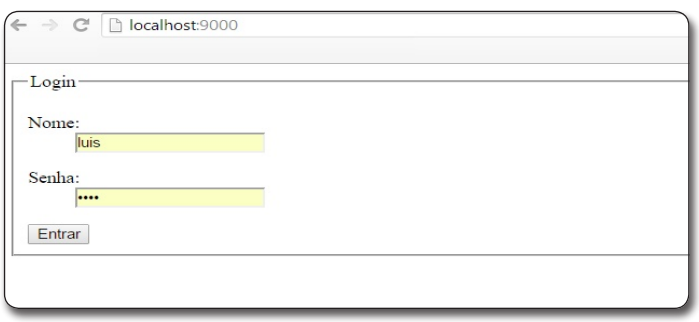


Figura 3. Exibição da página de login

Em seguida, informamos através da tag **<script>** a dependência ao arquivo *salachat.js*, onde implementamos algumas funções JavaScript que auxiliarão no funcionamento do chat. Tais funções serão úteis para inicializar o websocket e viabilizar o envio de mensagens para os demais membros com o pressionamento da tecla *Enter*, por exemplo. O conteúdo do arquivo *salachat.js* é apresentado na **Listagem 9**. A página da sala do chat é apresentada na **Figura 4.**

Nesse código, a conexão com o WebSocket é aberta ao instanciarmos a classe **WebSocket** na linha 5. Logo após, nas linhas 7 a 12, temos a função **enviaMensagem()**, que ao ser chamada irá enviar a mensagem digitada pelo usuário a todos os demais membros conectados ao chat. A função **enviaMensagemEnter()**, como o nome indica, possibilita o envio de mensagens ao pressionar a tecla *Enter*. Por último, temos a função **receiveEvent()**, que será invocada sempre que um novo usuário entrar na sala de chat. Após ser invocada, ela enviará uma mensagem avisando a entrada do novo usuário.

Listagem 8. Código da página salachat.html.

```
01. @(username: String)
02.
03. <html>
04.     <h1>Usuário logado: @username</h1>
05.
06.           <div id="onChat">
07.     <div id="main">
08.         <div id="messages"></div>
09.         <textarea id="talk"></textarea>
10.     </div>
11.   </div>
12.
13.   <div>
14.   <scriptsrc="@routes.Assets.at("javascripts/jquery-1.9.0.min.js")"
        type="text/javascript"></script>
15.   <script type="text/javascript" charset="utf-8" src="@routes.Chat.salachat
        (username)">
16. </script>
17.   </div>
18. </html>
```

Listagem 9. Código do arquivo salachat.js

```
01. @(username: String)
02.
03. $(function() {
04.
05.   var websocket = new WebSocket("@routes.Chat.inicializaChat(username).
       webSocketURL(request)")
06.
07.   var enviaMensagem = function() {
08.     websocket.send(JSON.stringify(
09.       {text: $("#talk").val()}
10.     ))
11.     $("#talk").val('')
12.   }
13.
14.   function enviaMensagemEnter(e) {
15.     e.which = e.which || e.keyCode;
16.     if(e.which == 13) {
17.       enviaMensagem()
18.     }
19.   }
20.
21.   var receiveEvent = function(event) {
22.     var data = JSON.parse(event.data)
23.
24.      $("#onChat").show()
25.
26.     var el = $('<div class="message"><span></span><p></p></div>')
27.     $("span", el).text(data.user)
28.     $("p", el).text(data.message)
29.     $(el).addClass(data.kind)
30.     if(data.user =='@username') $(el).addClass('me')
31.     $('#messages').append(el)
32.
33.   }
34.
35.  $("#talk").keypress(enviaMensagemEnter)
36.   websocket.onmessage = receiveEvent
37. })
```

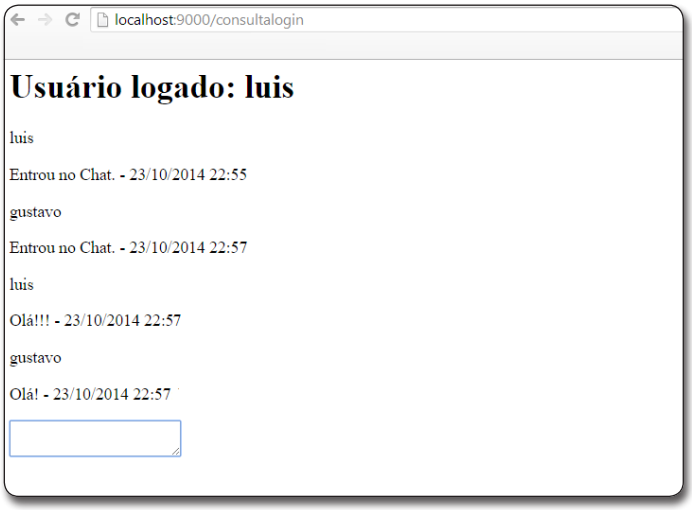


Figura 4. Exibição da sala do chat

O desenvolvimento de soluções web continua se reinventando. A cada dia surgem novas tecnologias simplificam a implementação destas soluções. A partir de agora, para começar a desenvolver aplicações real time, tarefa até então vista com bastante receio por muitos desenvolvedores, basta empregar tecnologias simples como WebSockets. E para garantir a agilidade e eficiência da programação, fazer uso de soluções como o Play Framework. Sendo assim, continue os estudos nestas tecnologias. Através delas você conseguirá criar sistemas web diferenciados para seus clientes.

Bons projetos!

Autor

Luis Gustavo Souza
os.luisgustavo@gmail.com

Cursa Sistemas de Informação pela Universidade de Franca, trabalhou como desenvolvedor no Grupo Amazonas e hoje é desenvolvedor Java na Universidade de Franca. Gosta de estudar e trabalhar com tecnologias open source e atualmente tem como objetivo obter a certificação OCA Java.

Links:

Download do Play Framework.
http://www.playframework.com/download

Informações de compatibilidade do WebSocket com os browsers.
https://www.websocket.org/echo.html

Exemplos de uso de WebSockets.
https://www.websocket.org/demos.html

Conteúdo sobre o Ebean.
https://www.playframework.com/documentation/2.0/JavaEbean

Documentação do Ebean.
http://www.avaje.org/ebean/documentation.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