



Edição 140 :: R\$ 14,90

 DEVMEDIA

Elabore projetos com a Arquitetura Orientada a Eventos  
Aprenda a evoluir a sua arquitetura de integração

Construa ambientes de alta disponibilidade  
Desenvolvendo as regras de negócio

Gerenciamento de dependências  
Conheça as principais ferramentas

# REFATORAÇÃO DE PROJETOS

Com boas práticas e  
padrões de projeto

Dominando coleções e Generics  
Como escolher a melhor opção  
e implementar código seguro

Big Data e MapReduce  
Criando uma solução  
completa com Hadoop



# DESENVOLVA PARA CORPORAÇÕES OU PARA O DIA A DIA COM JAVA

## FORMAÇÃO DESENVOLVEDOR JAVA

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

### CERTIFICAÇÕES:

➤ Oracle Certified Professional, Java SE 6 Programmer

➤ Oracle Certified Professional, Java EE 5 Web Component Developer





Edição 140 • 2015 • ISSN 1676-8361



MVP

Assine agora e tenha acesso a todo o conteúdo da DevMedia:  
[www.devmedia.com.br/mvp](http://www.devmedia.com.br/mvp)

## EXPEDIENTE

### Editor

Eduardo Spínola ([eduspinola@gmail.com](mailto:eduspinola@gmail.com))

**Consultor Técnico** Diogo Souza ([diogosouzac@gmail.com](mailto:diogosouzac@gmail.com))

### Produção

**Jornalista Responsável** Kaline Dolabella - JP24185

**Capa** Romulo Araújo

**Diagramação** Janete Feitosa

### Distribuição

FC Comercial e Distribuidora S.A.

Rua Teodoro da Silva, 907, Grajaú - RJ

CEP 20563-900, (21) 3879-7766 - (21) 2577-6362

### 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](http://www.devmedia.com.br/central), ou se preferir entre em contato conosco através do telefone 21 3382-5038.

### Publicidade

[publicidade@devmedia.com.br](mailto: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.

*Java, o logotipo da xícara de café Java e todas as marcas e logotipos baseados em/ou referentes a Java são marcas comerciais ou marcas registradas da Sun Microsystems, Inc. nos Estados Unidos e em outros países.*

### Fale com o Editor!

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

Se você estiver interessado em publicar um artigo na revista ou no site 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](http://eduspinola.wordpress.com)

@eduspinola / @Java\_Magazine

## CURSOS ONLINE

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



### CONHEÇA ALGUNS DOS CURSOS:

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

Para mais informações :

[www.devmedia.com.br/curso/javamagazine](http://www.devmedia.com.br/curso/javamagazine)

(21) 3382-5038



# Sumário

Artigo no estilo Curso

## 06 – Criando uma aplicação corporativa em Java – Parte 4

[ Bruno Rafael Sant'Ana ]

## 16 – Dominando o Java Collections Framework e Generics

[ José Fernandes A. Júnior ]

Conteúdo sobre Boas Práticas, Artigo no estilo Mentoring

## 26 – Refatoração utilizando padrões de projeto e boas práticas

[ Alex Radavelli ]

Conteúdo sobre Boas Práticas

## 34 – Elaborando projetos com a Arquitetura Orientada a Eventos

[ Ualter Azambuja Junior ]

Conteúdo sobre Boas Práticas

## 52 – Gerenciamento de dependências no Java

[ Bruno F.M. Attorre ]

Conteúdo sobre Novidades, Artigo no estilo Solução Completa

## 62 – Big Data: MapReduce na prática

[ Cláudio Martins e Wesley Louzeiro Mota ]



Dê seu feedback sobre esta edição!

A Java Magazine tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link:

[www.devmedia.com.br/javamagazine/feedback](http://www.devmedia.com.br/javamagazine/feedback)



## CRIANDO APLICAÇÕES INOVADORAS QUE VÃO MUDAR O MUNDO?

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

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

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

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

Na suite ToolsCloud você tem a sua disposição:

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

Acompanhe a Toolscloud:



[toolscloud@toolscloud.com](mailto:toolscloud@toolscloud.com)



[twitter.com/toolscloud](http://twitter.com/toolscloud)



# Criando uma aplicação corporativa em Java – Parte 4

Conheça mais recursos do JSF e desenvolva as regras de negócio da aplicação

ESTE ARTIGO FAZ PARTE DE UM CURSO

Neste quarto e último artigo da série desenvolvemos as duas principais funcionalidades do sistema gerenciador de bibliotecas: o empréstimo e a devolução de livros. Citamos tais funcionalidades como as mais importantes porque são elas que agregam maior valor a um software destinado a bibliotecas. A implementação desses dois itens será feita por meio de páginas JSF e managed beans.

Ainda nesse artigo, serão explicadas as fases do ciclo de vida do processamento de uma requisição feita a uma página JSF. Para complementar esse tópico, também mostraremos um exemplo prático de como implementar a interface **PhaseListener**, da API do JavaServer Faces, o que é bastante útil quando precisamos executar algum código antes ou depois de alguma fase do ciclo de vida da requisição.

Por fim, também é válido citar que passaremos rapidamente pela nova API de data e hora do Java 8, a qual será adotada para simplificar a manipulação de datas em todas as funcionalidades relacionadas da nossa aplicação.

## Criando a funcionalidade para empréstimo de livros

Após toda a construção realizada até aqui, o nosso próximo passo é implementar a funcionalidade para empréstimo de livros. Iniciaremos pela página *emprestimo.xhtml*, que tem seu código apresentado na **Listagem 1**. Tal página, apresentada na **Figura 1**, é composta por um formulário que irá registrar o empréstimo, salvando suas informações no banco de dados. Para isso, é possível no formulário escolher o livro que será emprestado e para qual leitor é este empréstimo.

## Fique por dentro

Este artigo é útil por apresentar informações importantes sobre o JavaServer Faces e concluir o desenvolvimento da aplicação exemplo. Como um dos destaques, citamos a análise sobre as fases do ciclo de vida do processamento de uma requisição a uma página JSF. Além disso, vamos demonstrar como adotar a nova API de Data e Hora na prática, recurso lançado com a versão 8 do Java e que já começa a ser amplamente adotado pelo mercado. A partir de todo o conteúdo exposto nesta série, o leitor terá um ótimo material de referência para iniciar o desenvolvimento de suas aplicações e se aprofundar ainda mais no assunto.

do suas informações no banco de dados. Para isso, é possível no formulário escolher o livro que será emprestado e para qual leitor é este empréstimo.

Para apresentar todos os livros disponíveis para empréstimo, usamos os componentes **h:selectOneMenu**, que irá gerar um elemento HTML do tipo **select**, e **f:selectItems**, que irá gerar elementos HTML do tipo **option**. Através do componente **h:selectOneMenu** conseguimos associar o valor do item selecionado com uma propriedade do managed bean (nesse caso, a propriedade **idLivro**). Com relação ao componente **f:selectItems**, para que o leitor entenda como ele foi utilizado no código da página *emprestimo.xhtml*, veremos para que servem seus atributos, que são explicados a seguir:

- **value** – colocamos nesse atributo a EL `#{$emprestimoBean.livrosDisponiveis}`, para indicar que os livros que devem ser listados devem ser provindos da propriedade **livrosDisponiveis** do managed bean. Em outras palavras, o método **getLivrosDisponiveis()** – presente no bean – será invocado e irá retornar uma lista de livros, que serão exibidos na tela;

- **itemValue** – define o que será setado na propriedade **idLivro** do bean. Como podemos notar na EL `#{livro.id}`, será setado o **id** do livro selecionado;

The screenshot shows a web application interface for managing book loans. At the top, there's a header with the title 'Easy - sistema de biblioteca' and a log-in status 'Logado como: teste'. Below the header is a main content area titled 'Emprestimo'. Inside, there's a form with several input fields: a dropdown for selecting a book ('Escolher livro a emprestar'), a dropdown for selecting a reader ('Leitor'), two date inputs ('Data do empréstimo' and 'Data prevista para devolução'), and a command button labeled 'Realizar Emprestimo'. At the bottom left of the form, there's a link 'Voltar ao Menu'.

**Figura 1.** Conteúdo da página emprestimo.xhtml

#### Listagem 1. Código da página emprestimo.xhtml.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:lnsh="http://java.sun.com/jsf/html"
      xmlns:ui="http://java.sun.com/jsf/facelets">
<ui:composition template="/WEB-INF/_template.xhtml">
    <ui:define name="corpo">
        <h:form>
            <h2>
                <h:outputText value="Empréstimo" />
            </h2>
            <fieldset>
                <legend>Dados do Empréstimo</legend>
                <h:outputLabel value="Escolher livro a emprestar:" for="livro" />
                <h:selectOneMenu id="livro" value="#{emprestimoBean.idLivro}">
                    <f:selectItems value="#{emprestimoBean.livrosDisponiveis}"
                                   var="livro" itemValue="#{livro.id}" itemLabel="#{livro.nome}" />
                </h:selectOneMenu>
                <h:outputLabel value="Leitor:" for="leitor" />
                <h:selectOneMenu id="leitor" value="#{emprestimoBean.idLeitor}">
                    <f:selectItems value="#{leitorBean.leitores}"
                                   var="leitor" itemValue="#{leitor.id}" itemLabel="#{leitor.nome}" />
                </h:selectOneMenu>
                <h:outputLabel value="Data do empréstimo:" for="dataEmprestimo" />
                <h:inputText id="dataEmprestimo" value="#{emprestimoBean
                    .dataEmprestimoFormatada}" disabled="true" />
                <h:outputLabel value="Data prevista para devolução:" for="dataPrevista" />
                <h:inputText id="dataPrevista" value="#{emprestimoBean
                    .dataPrevistaFormatada}" disabled="true" />
                <h:commandButton value="Realizar Empréstimo" action=
                    "#{emprestimoBean.realizarEmprestimo}" />
            </fieldset>
        </h:form>
    </ui:define>
</ui:composition>
</html>
```

- **itemLabel** – define o que de fato será exibido para o usuário. Ao observarmos a EL `#{livro.nome}`, fica fácil presumir que serão apresentados apenas os nomes dos livros na listagem.

Para que sejam listados todos os leitores, também usamos as mesmas tags **h:selectOneMenu** e **f:selectItems** na página *emprestimo.xhtml*. Já do lado do managed bean, de forma análoga ao que fizemos com o **idLivro**, também criamos uma propriedade **idLeitor** para receber o **id** do leitor selecionado.

Logo abaixo do componente que lista os leitores serão exibidas a data do empréstimo e a data prevista para devolução do livro. Para isso, usamos o componente **h:inputText**, porém configuramos seu atributo **disabled** para **true** para que o usuário do sistema não possa alterar essas datas, que serão geradas automaticamente.

O último componente do formulário é um **h:commandButton**, que irá acionar o método **realizarEmprestimo()** do managed bean, quando clicado.

#### Listagem 2. Código do managed bean LeitorBean.

```
package br.com.javamagazine.mb;

import java.util.List;
import javax.enterprise.context.RequestScoped;
import javax.inject.Inject;
import javax.inject.Named;
import br.com.javamagazine.dao.LeitorDao;
import br.com.javamagazine.entidades.Leitor;
import br.com.javamagazine.interceptadores.Transacional;

@Named
@RequestScoped
public class LeitorBean {

    //restante da implementação omitida

    @Inject
    private LeitorDao leitorDao;
    private List<Leitor> leitores;

    @Transacional
    public List<Leitor> getLeitores(){
        if(leitores == null){
            leitores = leitorDao.listarLeitores();
        }
        return leitores;
    }

    //restante da implementação omitida
}
```

Como podemos verificar na **Listagem 1**, é utilizada a tag **f:selectItems** para que sejam exibidos os nomes dos leitores e no atributo **value** dessa tag é colocada a EL `#{leitorBean.leitores}`. Então, para que essa página funcione, o managed bean **LeitorBean** precisa ser criado. O código desse bean pode ser visto na **Listagem 2**. Note que deixamos essa classe com o mínimo de código necessário, permanecendo apenas o método **getLeitores()**, que irá recuperar a lista com todos os leitores do banco de dados através de uma classe DAO.

Analisaremos agora o código do managed bean **EmprestimoBean**, presente na **Listagem 3**. No construtor dessa classe são invocados os métodos setters no objeto que representa o empréstimo para que sejam definidas a data do empréstimo e a data prevista para devolução do livro. A data do empréstimo é a data atual e a data prevista para devolução é obtida a partir da adição de sete dias à data atual. Com o intuito de simplificar o nosso trabalho, observe que estamos usando a nova API de data e hora do Java 8. Com ela fica bem mais simples somar sete dias à data atual, pois a API provê uma interface fluente que possibilita o encadeamento de chamadas de métodos, o que geralmente torna o código mais fácil de ler.

Para que essas duas datas sejam mostradas ao usuário no formato dd/MM/yyyy, usamos a classe **DateTimeFormatter** – também disponível a partir do Java 8 – para formatá-las. Ambas as datas serão formatadas como **String** e armazenadas nas variáveis **dataEmprestimoFormatada** e **dataPrevistaFormatada**.

Para que essas datas sejam exibidas ao usuário, na página *emprestimo.xhtml* são utilizados dois componentes do tipo **h:inputText** que têm como valor as seguintes ELs: **#{emprestimoBean.dataEmprestimoFormatada}** e **#{emprestimoBean.dataPrevistaFormatada}**.

O principal método desse bean é o **realizarEmprestimo()**, que contém a lógica referente ao empréstimo de livros. Observe que colocamos a anotação **@Transacional** para que ele seja interceptado pelo nosso **TransacionalInterceptor** e seja executado dentro de uma transação. Antes de persistirmos o empréstimo, no entanto, precisamos relacioná-lo com o livro que está sendo emprestado, com o leitor que está tomando o livro emprestado e com o funcionário da biblioteca que está operando o sistema. Conforme explicado na listagem anterior, a classe **EmprestimoBean** possui duas propriedades, **idLivro** e **idLeitor**, que armazenam os ids dos objetos, e usamos esses ids para recuperar tanto o livro quanto o leitor do banco de dados e assim associá-los ao empréstimo.

Como existe um relacionamento entre o usuário logado e o funcionário ao qual o usuário pertence, recuperamos o funcionário da biblioteca através do usuário logado e também o associamos ao empréstimo. Feitas as associações entre os objetos, persistimos o objeto que representa o empréstimo e, por fim, a lista de livros disponíveis é atualizada. Ao final, também é adicionada uma mensagem indicando que a operação foi realizada com sucesso, a ser exibida na página.

## Criando a funcionalidade para devolução de livros

Uma das partes mais importantes do sistema, sem dúvidas, é a funcionalidade para devolução de livros. A página *devolucao.xhtml* (vide **Listagem 4**) juntamente com o managed bean **DevolucaoBean** (vide **Listagem 5**) dão forma a esta funcionalidade.

Explicaremos essa parte específica do sistema em duas etapas. Primeiro, iremos analisar os pontos mais importantes da página *devolucao.xhtml*, e depois, passaremos à análise da classe **DevolucaoBean**.

### Listagem 3. Código do managed bean EmprestimoBean.

```
package br.com.javamagazine.mb;

import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.util.List;
import javax.enterprise.context.RequestScoped;
import javax.inject.Inject;
import javax.inject.Named;
// restante dos imports omitidos

@Named
@RequestScoped
public class EmprestimoBean {

    @Inject
    private EmprestimoDao emprestimoDao;
    private Emprestimo emprestimo = new Emprestimo();
    @Inject
    private LivroDao livroDao;
    private Integer idLivro;
    private List<Livro> livrosDisponiveis;
    @Inject
    private LeitorDao leitorDao;
    private Integer idLeitor;
    @Inject
    private UsuarioLogadoBean usuarioLogadoBean;
    String dataEmprestimoFormatada;
    String dataPrevistaFormatada;

    EmprestimoBean(){
        emprestimo.setDataEmprestimo(LocalDate.now());
        emprestimo.setDataPrevista(LocalDate.now().plusDays(7));
        DateTimeFormatter formatador = DateTimeFormatter.ofPattern("dd/MM/yyyy");
        dataEmprestimoFormatada = emprestimo.getDataEmprestimo()
            .format(formatador);
        dataPrevistaFormatada = emprestimo.getDataPrevista().format(formatador);
    }

    @Transactional
    public void realizarEmprestimo(){
        Livro livro = livroDao.pesquisarPorId(idLivro);
        Leitor leitor = leitorDao.pesquisarPorId(idLeitor);
        FuncionarioBiblioteca funcionarioBiblioteca = usuarioLogadoBean
            .getUsuario().getFuncionarioBiblioteca();
        emprestimo.setLivro(livro);
        livro.getEmprestimos().add(emprestimo);
        emprestimo.setLeitor(leitor);
        emprestimo.setFuncionarioBiblioteca(funcionarioBiblioteca);
        emprestimoDao.inserir(emprestimo);
        livrosDisponiveis = livroDao.listarLivrosDisponiveisParaEmprestimo();
        MensagemUtil.addMensagemInformativa("Sucesso - ",
            "Empréstimo realizado com sucesso!");
    }

    @Transactional
    public List<Livro> getLivrosDisponiveis() {
        if(livrosDisponiveis == null){
            livrosDisponiveis = livroDao.listarLivrosDisponiveisParaEmprestimo();
        }
        return livrosDisponiveis;
    }

    //demais métodos get e set omitidos...
}
```

Na página *devolucao.xhtml* (apresentada na **Figura 2**), por meio dos componentes **h:selectOneMenu** e **f:selectItems** – explicados anteriormente –, é exibida uma lista de livros que se encontram emprestados e são passíveis de devolução. Logo no início dessa lista, adicionamos uma opção em branco através do componente **f:selectItem**. Pelo fato desta opção em branco já vir selecionada ao carregar a página, o usuário é obrigado a selecionar na lista o livro que será devolvido, o que dispara a execução do método **alteradoLivroSelecionado()** presente no managed bean. Mas porque esse método é invocado quando escolhemos algum livro? Veremos mais detalhes adiante.

O mecanismo descrito no parágrafo anterior – executar um determinado método a cada vez que se escolhe um livro da lista – foi especificado na tag **f:ajax**. Para isso, no atributo **listener** dessa tag foi colocada a EL `#{}{devolucaoBean.alteradoLivroSelecionado}`. Assim, toda vez que escolhemos um item na lista de livros emprestados, o método **alteradoLivroSelecionado()** do nosso bean será invocado. Logo após, o formulário da página será atualizado (renderizado novamente), pois informamos seu id no atributo **render** (`render="formDevolucao"`) da tag **f:ajax**.

Com esse mecanismo, ao selecionarmos qualquer livro na lista, os demais campos do formulário serão preenchidos automaticamente e o operador do sistema poderá visualizar os dados relacionados àquele empréstimo, como o nome

do leitor que pegou o livro emprestado, o nome do funcionário da biblioteca que realizou o empréstimo, a data do empréstimo e a data prevista para devolução e se existe multa a ser paga por atraso. Observe ainda que a edição dos valores dos campos é desabilitada através do atributo **disabled="true"**.

A partir desse ponto, passaremos a analisar o managed bean **DevolucaoBean**. Por se tratar de uma classe grande, quebramos seu código em diferentes **Listagens** (da 5 a 12).

**Figura 2.** Conteúdo da página *devolucao.xhtml*

**Listagem 4.** Código da página *devolucao.xhtml*.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ui="http://java.sun.com/jsf/faces">

<ui:composition template="/WEB-INF/_template.xhtml">
  <ui:define name="corpo">
    <h:form id="formDevolucao">
      <h2>
        <h:outputText value="Devolução" />
      </h2>
      <fieldset>
        <legend>Dados da Devolução</legend>
        <h:outputLabel value="Escolher livro a devolver:" for="livro" />
        <h:selectOneMenu id="livro" value="#{devolucaoBean.idLivro}">
          <f:selectItem itemLabel="" itemValue="#{null}" />
          <f:selectItems value="#{devolucaoBean.livrosEmprestados}" var="livro"
            itemValue="#{livro.id}" itemLabel="#{livro.nome}" />
          <f:ajax listener="#{devolucaoBean.alteradoLivroSelecionado}"
            render="formDevolucao" />
        </h:selectOneMenu>
        <h:outputLabel value="Emprestado para:" for="emprestadoPara" />
```

```
<h:inputText id="emprestadoPara" value="#{devolucaoBean.emprestimo
  .leitor.nome}" disabled="true" />

<h:outputLabel value="Emprestado por:" for="emprestadoPor" />
<h:inputText id="emprestadoPor" value="#{devolucaoBean.emprestimo
  .funcionarioBiblioteca.nome}" disabled="true" />

<h:outputLabel value="Data do empréstimo:" for="dataEmprestimo" />
<h:inputText id="dataEmprestimo" value="#{devolucaoBean
  .dataEmprestimoFormatada}" disabled="true" />

<h:outputLabel value="Data prevista para devolução:" for="dataPrevista" />
<h:inputText id="dataPrevista" value="#{devolucaoBean
  .dataPrevistaFormatada}" disabled="true" />

<h:outputLabel value="Multa por atraso:" for="multa" />
<h:inputText id="multa" value="#{devolucaoBean.multa}" disabled="true" />

<h:commandButton value="Realizar Devolução"
  action="#{devolucaoBean.realizarDevolucao}" />
</fieldset>
</h:form>
</ui:define>
</ui:composition>
</html>
```

# Criando uma aplicação corporativa em Java – Parte 4

Listagem 5. Código do managed bean DevolucaoBean.

```
package br.com.javamagazine.mb;

import java.math.BigDecimal;
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.util.List;
import javax.enterprise.context.RequestScoped;
import javax.inject.Inject;
import javax.inject.Named;
// restante dos imports omitidos

@Named
@RequestScoped
public class DevolucaoBean {
    @Inject
    private LivroDao livroDao;
    private Integer idLivro;
    private List<Livro> livrosEmprestados;
    @Inject
    private EmprestimoDao emprestimoDao;
    private Emprestimo emprestimo = new Emprestimo();
    private String dataEmprestimoFormatada;
    private String dataPrevistaFormatada;

    private String multa = FormattedorDeNumeros.
    formatarBigDecimalComoMoeda(new BigDecimal(0.0d));

    //Código do método alteradoLivroSelecionado(), apresentado na Listagem 6

    //Código do método realizarDevolucao(), apresentado na Listagem 7

    //Código do método setarEmprestimo(), apresentado na Listagem 8

    //Código do método getLivroSelecionado(), apresentado na Listagem 9

    //Código do método getEmprestimoDoLivroSelecionado(), apresentado
    //na Listagem 10

    //Código do método limparCampos(), apresentado na Listagem 11

    //Código do método getLivrosEmprestados(), apresentado na Listagem 12

    //Métodos de acesso omitidos
}
```

Esse bean possui alguns métodos importantes que foram extraídos e serão explicados à parte.

Começaremos as explicações pelo método **alteradoLivroSelecionado()**, presente na **Listagem 6**. Este método será invocado quando o usuário do sistema escolher algum livro na lista de livros emprestados, que aparece na página *devolucao.xhtml*. Em outras palavras, o método é chamado toda vez que alteramos a seleção, ou seja, quando mudamos o livro que está selecionado. Dentro dele é verificado se o conteúdo da variável **idLivro** está preenchido, o que significa que algum livro foi selecionado. Se o conteúdo da variável for nulo, significa que a opção em branco da lista foi selecionada. Se algum livro foi selecionado, o método **setarEmprestimo()** é chamado para atribuir o empréstimo vigente – representado por um objeto – relacionado àquele livro à variável de instância **emprestimo**.

Caso exista mais de um empréstimo vinculado ao livro, o que vale é aquele que ainda não tem data de devolução, pois essa data só será atualizada quando a devolução for concretizada. Ainda nesse método são recuperadas a data do empréstimo e a data prevista para devolução, formatadas e atribuídas às variáveis de instância **dataEmprestimoFormatada** e **dataPrevistaFormatada**, para que possam ser exibidas na página. Em seguida é avaliado se a data atual é maior que a data prevista para o empréstimo, e nesse caso é gerada uma multa de R\$ 15,00. Para mantermos o sistema simples, a multa não será persistida. Apenas será mostrada na tela para que seja cobrado algum valor pelo atraso.

Listagem 6. Código do método alteradoLivroSelecionado().

```
public void alteradoLivroSelecionado(){
    if(idLivro != null){
        setarEmprestimo();
        DateTimeFormatter formatador = DateTimeFormatter.ofPattern("dd/MM/yyyy");
        dataEmprestimoFormatada = emprestimo.getDataEmprestimo()
        .format(formatador);
        dataPrevistaFormatada = emprestimo.getDataPrevista().format(formatador);
        LocalDate dataAtual = LocalDate.now();
        if(dataAtual.isAfter(emprestimo.getDataPrevista())){
            multa = FormattedorDeNumeros.formatarBigDecimalComoMoeda
            (new BigDecimal(15.0d));
        }
    }else{
        limparCampos();
    }
}
```

O método principal da classe **DevolucaoBean** é o **realizarDevolução()**, exposto na **Listagem 7**. Esse método primeiramente invoca o **setarEmprestimo()**, que recupera o empréstimo associado ao livro que foi selecionado para ser devolvido e o atribui à variável de instância **emprestimo**. Nesse momento é atualizada a data de devolução desse empréstimo com a data atual. Em seguida essa alteração no empréstimo é persistida no banco de dados através de uma classe DAO. Na sequência, a lista dos livros emprestados é atualizada para que o livro que acabou de ser devolvido não apareça mais nela. Por fim, os campos são limpos através de uma chamada ao método **limparCampos()** e é adicionada uma mensagem informando ao usuário que a devolução foi realizada com sucesso. Repare também que o método é marcado com a anotação **@Transacional**, que criamos no segundo artigo da série.

Dando continuidade às explicações dos métodos, veremos a seguir como o **setarEmprestimo()** foi implementado – vide **Listagem 8**. No início ocorre uma chamada a **getLivroSelecionado()**, que retorna um objeto que representa o livro selecionado pelo usuário do sistema. Na próxima linha de código, o livro selecionado é passado como argumento para o método **getEmprestimoDoLivroSelecionado()**, que faz exatamente o que seu nome sugere e retorna o objeto empréstimo vinculado ao livro. Antes do método terminar, o objeto empréstimo que foi retornado é atribuído à variável membro **emprestimo**, para que possa ser utilizado em outros lugares da classe.

#### Listagem 7. Código do método realizarDevolucao().

```
@Transacional
public void realizarDevolucao(){
    setarEmprestimo();
    emprestimo.setDataDevolucao(LocalDate.now());
    emprestimoDao.atualizar(emprestimo);
    livrosEmprestados = livroDao.listarLivrosEmprestados();
    limparCampos();
    MensagemUtil.addMensagemInformativa("Sucesso -",
        "Devolução realizada com sucesso!");
}
```

#### Listagem 8. Código do método setarEmprestimo().

```
private void setarEmprestimo(){
    Livro livroSelecionado = getLivroSelecionado();
    Emprestimo emprestimoDoLivroSelecionado =
        getEmprestimoDoLivroSelecionado(livroSelecionado);
    emprestimo = emprestimoDoLivroSelecionado;
}
```

Note que no parágrafo anterior mencionamos o método `getLivroSelecionado()`. Então vejamos agora a sua implementação ([Listagem 9](#)). Para entendermos a lógica desse método é necessário relembrar que quando o usuário do sistema escolhe um livro, armazenamos seu id na variável `idLivro`. Como temos o id do livro que foi selecionado, iteramos na lista de livros emprestados procurando pelo livro que tenha o mesmo id, e quando o encontramos retornamos esse livro.

#### Listagem 9. Código do método getLivroSelecionado().

```
private Livro getLivroSelecionado(){
    for(Livro livro : getLivrosEmprestados()){
        if(livro.getId() == idLivro){
            return livro;
        }
    }
    return null;
}
```

Seguindo com as explicações, analisaremos o código do método `getEmprestimoDoLivroSelecionado()`, apresentado na [Listagem 10](#). Visando um melhor entendimento, antes de explicarmos o código desse método, é válido mencionar que em nosso sistema um livro pode estar relacionado a vários empréstimos e o último empréstimo realizado é aquele cuja data de devolução está nula. Essa data permanece nula até o momento em que o livro é efetivamente devolvido, e só então ela é atualizada. Dito isso, passemos à implementação do método em questão. Como podemos verificar, seu código consiste em iterar sobre os empréstimos do livro que foi selecionado pelo usuário e retornar o último empréstimo realizado, que seria o empréstimo vigente.

Outro ponto que deve ser previsto pelo desenvolvedor é a situação em que o usuário seleciona a opção em branco disponível na

lista de livros da página. Caso isso ocorra, os valores dos demais campos do formulário também devem ficar em branco e nesse caso `limparCampos()` é chamado para que esse resultado seja alcançado. Veja o código desse método na [Listagem 11](#).

Para finalizarmos a análise dos métodos de `DevolucaoBean`, veremos a implementação de `getLivrosEmprestados()`. A [Listagem 12](#) exibe o código desse método que, como veremos, é bastante simples. Através de uma chamada a `listarLivrosEmprestados()`, do DAO, é recuperada a lista de livros emprestados e em seguida ela é atribuída à variável membro `livrosEmprestados`. Note ainda que para que não seja feita uma nova invocação ao método do DAO caso a variável `livrosEmprestados` já esteja preenchida com a lista, foi colocado um `if`.

#### Listagem 10. Código do método getEmprestimoDoLivroSelecionado().

```
private Emprestimo getEmprestimoDoLivroSelecionado(Livro livroSelecionado){
    for(Emprestimo emprestimo : livroSelecionado.getEmprestimos()){
        if(emprestimo.getDataDevolucao() == null){
            return emprestimo;
        }
    }
    return null;
}
```

#### Listagem 11. Código do método limparCampos().

```
private void limparCampos(){
    emprestimo = new Emprestimo();
    dataEmprestimoFormatada = "";
    dataPrevistaFormatada = "";
    multa = FormatadorDeNumeros.formatarBigDecimalComoMoeda(new
        BigDecimal(0.0d));
}
```

#### Listagem 12. Código do método getLivrosEmprestados().

```
@Transacional
public List<Livro> getLivrosEmprestados() {
    if(livrosEmprestados == null){
        livrosEmprestados = livroDao.listarLivrosEmprestados();
    }
    return livrosEmprestados;
}
```

Antes de encerrarmos esse tópico sobre devolução de livros, é interessante citar que nos locais onde é necessário formatar o valor da multa, o managed bean `DevolucaoBean` faz uso do método `formatarBigDecimalComoMoeda()`, presente na classe `FormatadorDeNumeros`. Esse método recebe um objeto do tipo `BigDecimal` e retorna uma `String` que representa o valor desse `BigDecimal` no formato da moeda do Brasil, acrescentando para isso o R\$. Por exemplo, ao passarmos para o método `new BigDecimal(15.0d)`, receberemos de volta "R\$ 15,00". A [Listagem 13](#) mostra o código da classe e do método em questão.

**Listagem 13.** Código da classe para formatação de números.

```
package br.com.javamagazine.util;

import java.math.BigDecimal;
import java.text.DecimalFormat;
import java.text.DecimalFormatSymbols;
import java.util.Locale;
public class FormatadorDeNumeros {

    public static String formatarBigDecimalComoMoeda(BigDecimal numero){
        Locale localeBrasil = new Locale("pt","BR");
        DecimalFormatSymbols moedaReal = new DecimalFormatSymbols
            (localeBrasil);
        DecimalFormat df = new DecimalFormat("#,##,##,##.00", moedaReal);
        return df.format(numero);
    }
}
```

## Evitando acessos indevidos com PhaseListener

A aplicação está quase finalizada, porém, nesse momento, ela tem um problema que precisa ser corrigido. Da forma que está, ela permite que o usuário acesse qualquer página através da URL mesmo sem estar logado. E mesmo que o usuário tenha se logado, se ele não for administrador, não deveria poder acessar a página de ca-

dastro de funcionários e usuários, mas por enquanto ele consegue. Para resolver esse problema dos acessos indevidos, vamos criar uma classe que implemente a interface **PhaseListener**, da API do JSF. Antes, no entanto, precisamos saber quais são as fases do ciclo de vida do processamento de uma requisição feita a uma página JSF e entender cada uma delas, já que o uso da interface **PhaseListener** tem relação direta com essas fases.

A Figura 3 mostra as seis fases do ciclo de vida. São elas:

- **Restore view:** é nessa fase que a árvore de componentes UI, chamada de view, é criada. No entanto, ela só será criada caso seja a requisição inicial à página JSF, caso contrário a view já existirá e apenas será restaurada;
- **Apply request values:** nessa fase os valores vindos por parâmetros na requisição são atribuídos a cada componente da árvore;
- **Process validations:** é nesta etapa que são processados todos os validadores associados aos componentes da árvore;
- **Update model values:** se não ocorreu nenhum erro de validação nem conversão nas etapas anteriores, isso significa que os valores são válidos. Então são passados dos componentes para as propriedades do managed bean, o que ocorre neste momento;
- **Invoke application:** depois do modelo ter sido atualizado na fase anterior, nessa fase são tratados os eventos disparados pelo

# CURSOS ONLINE

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



### CONHEÇA OS CURSOS MAIS RECENTES:

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

Para mais informações :

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



**DEV**MEDIA

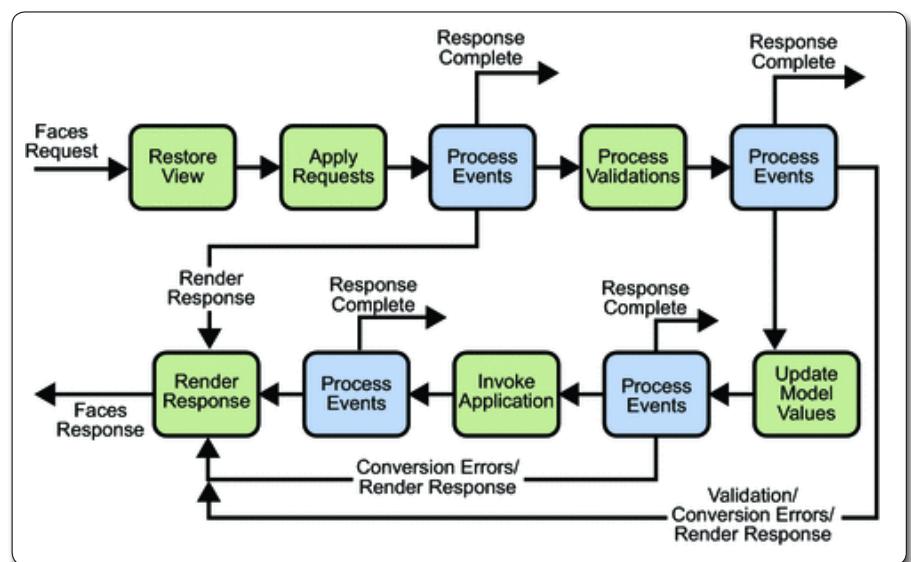
usuário (por exemplo, se ele clicou em algum botão) e executadas as regras de negócio;

- **Render response:** por fim, nesta fase será renderizada a resposta para o usuário.

Uma classe que implemente a interface **PhaseListener** será notificada antes e depois do processamento de cada uma dessas fases do ciclo de vida de uma requisição JSF, ou podemos escolher apenas a fase que nos interessa. Para o nosso exemplo, só interessa sermos notificados antes e depois do processamento da fase **Restore view**. Sendo assim, criaremos uma classe chamada **Autorizador**, que implementará **PhaseListener** (veja seu código na **Listagem 14**).

Note que sobreescrivemos os métodos **beforePhase()** e **afterPhase()**, porém todo o nosso código ficou no **afterPhase()** e deixamos o **beforePhase()** vazio. Isso porque antes da fase **Restore view**, a chamada ao método **context.getViewRoot()** retorna **null**, e não conseguíramos utilizar **context.getViewRoot().getViewId()** para descobrir a página que o usuário estaria tentando acessar. E também foi sobreescrito o método **getPhaseId()**, onde podemos informar qual fase do ciclo de vida nos interessa para sermos notificados.

Dentro do método **afterPhase()** está toda a lógica que previne os acessos indevidos. Primeiramente, verificamos se o usuário já se logou no sistema. Caso ele esteja logado e seja administrador, poderá acessar qualquer página, mas caso não seja administrador, não poderá acessar o cadastro de funcionários e usuário e se ele tentar, será direcionado para a página **menu\_principal.xhtml**.



**Figura 3.** Ciclo de vida de uma requisição JSF

**Listagem 14.** Código do PhaseListener Autorizador.

```
package br.com.javamagazine.listeners;

import javax.faces.context.FacesContext;
import javax.faces.event.PhaseEvent;
import javax.faces.event.PhaseId;
import javax.faces.event.PhaseListener;
import javax.inject.Inject;

import br.com.javamagazine.mb.FuncionarioEUsuarioVerificadorBean;
import br.com.javamagazine.mb.UsuarioLogadoBean;
import br.com.javamagazine.util.Navegador;

public class Autorizador implements PhaseListener {

    private static final long serialVersionUID = 1L;
    @Inject
    private UsuarioLogadoBean usuarioLogado;
    @Inject
    FacesContext context;
    @Inject
    private Navegador navegador;
    @Inject
    private FuncionarioEUsuarioVerificadorBean verificadorBean;

    @Override
    public void afterPhase(PhaseEvent event) {
        if(usuarioLogado.isLogado()){

            if("/funcionario_biblioteca.xhtml".equals(context.getViewRoot().getViewId()) && !usuarioLogado.getUsuario().isAdmin()){
                navegador.redirecionar("menu_principal");
            }
            else{
                if("/login.xhtml".equals(context.getViewRoot().getViewId())){
                    return;
                }
                if("/funcionario_biblioteca.xhtml".equals(context.getViewRoot().getViewId()) && !verificadorBean.existeFuncionarioEUsuarioCadastrado()){
                    return;
                }

                navegador.redirecionar("login");
            }
        }
    }

    @Override
    public void beforePhase(PhaseEvent event) {
        // não é necessário fazer nada antes da fase Restore View
    }

    @Override
    public PhaseId getPhaseId() {
        return PhaseId.RESTORE_VIEW;
    }
}
```

A outra situação ocorre se o usuário ainda não estiver autenticado. Nesse caso, ele não poderá acessar nenhuma página do sistema, com exceção da página de login, obviamente, e da página de cadastro de funcionários e usuários, desde que ainda não exista nenhum funcionário e nenhum usuário cadastrados. Se ele tentar acessar qualquer página diferente dessas duas, será direcionado para a página de login.

Por fim, para que os métodos da classe **Autorizador** sejam realmente chamados antes e depois da fase Restore view, devemos registrá-la no arquivo *faces-config.xml*, conforme o código da **Listagem 15**. Assim concluímos a implementação da aplicação exemplo, que agora já se encontra 100% funcional.

**Listagem 15.** Código do arquivo faces-config.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config
    xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
    http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig_2_2.xsd"
    version="2.2">

<lifecycle>
    <phase-listener>
        br.com.javamagazine.listeners.Autorizador
    </phase-listener>
</lifecycle>

</faces-config>
```

Após essas últimas explicações, o leitor estará apto a iniciar o desenvolvimento de aplicações web usando JSF. Ainda assim, existem muitos assuntos igualmente interessantes que devem ser estudados pelo leitor para que seja possível desenvolver uma aplicação completa. Visando simplificar/direcionar este caminho, aconselhamos estudos mais aprofundados sobre JPA, CDI e o próprio JSF, todos já explorados com grande riqueza de detalhes em artigos publicados em outras edições da Java Magazine.

Bons estudos!

## Autor



### Bruno Rafael Sant'Ana

[bruno.santana.ti@gmail.com](mailto:bruno.santana.ti@gmail.com)

Graduado em Análise e Desenvolvimento de Sistemas pelo SENAC. Possui as certificações OCJP e OCWCD. Atualmente trabalha na Samsung com desenvolvimento Java e atua como CTO na startup Vesteer. Entusiasta de linguagens de programação e tecnologia.



## Links:

### Aplicação reduzida (igual a desenvolvida no artigo) no GitHub.

<https://github.com/brunosantanati/javamagazine-app-reduzida>

### Aplicação completa no GitHub.

<https://github.com/brunosantanati/javamagazine-app-completa>

### Endereço para download do JDK 8.

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

### Endereço para download do Eclipse Luna.

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

### Endereço para download JBoss WildFly.

<http://wildfly.org/downloads/>

### Endereço para download do instalador do MySQL.

<http://dev.mysql.com/downloads/installer/>

### Endereço para download do driver do MySQL.

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

## Você gostou deste artigo?

Dê seu voto em [www.devmedia.com.br/javamagazine/feedback](http://www.devmedia.com.br/javamagazine/feedback)

Ajude-nos a manter a qualidade da revista!



# MVP

R\$ 1.000.000,00  
INVESTIDOS EM CONTEÚDO  
NOS ÚLTIMOS 12 MESES.

APlique esse investimento  
na sua carreira...

E mostre ao mercado  
quanto você vale!

CONFIRA TODO O MATERIAL  
QUE VOCÊ TERÁ ACESSO:

- + de **9.000** video-aulas
- + de **290** cursos online
- + de **13.000** artigos
- DEVMEDIA API's**  
consumido + de **500.000** vezes



POR APENAS  
**R\$ 69,90\*** mensais

\*Tempo mínimo de assinatura: 12 meses.



PRA QUEM QUER EXIGIR  
MAIS DO MERCADO!

 **DEVMEDIA**

# Dominando o Java Collections Framework e Generics

Conheça a API de coleções do Java e entenda o papel dos genéricos na criação de código seguro

É muito comum escrevermos programas que se comunicem passando informações através de objetos. Mas quando precisamos enviar vários objetos para serem processados, torna-se dispendioso fazer o seu envio de forma individual. Melhor seria fazer um único envio de vários objetos agrupados e receber uma única resposta, ou uma série de respostas agrupadas em um único objeto. Pensando nisso, a plataforma Java nos disponibiliza um framework de Collections.

Uma coleção, também designada de container, é um simples objeto que agrupa múltiplos objetos (conhecidos como elementos da coleção) numa só unidade. Estas coleções são utilizadas para guardar, retornar, manipular e transmitir dados agregados de maneira simples e estão disponíveis no framework de Collections do Java, que possui uma arquitetura unificada e criada para permitir que coleções sejam manipuladas de forma independente dos detalhes de suas implementações.

As principais vantagens do framework de Collections são:

- Reduz o esforço de programação, fornecendo estruturas de dados e algoritmos de modo que não tenhamos que escrevê-los;
- Por fornecer implementações de estruturas de dados e algoritmos de alto desempenho, viabiliza um aumento de performance;
- Proporciona interoperabilidade entre APIs não relacionadas, ao estabelecer uma linguagem comum para passar informações através de coleções;
- Reduz o esforço necessário para aprender sobre as APIs, já que as coleções que nos são disponibilizadas são comuns em várias linguagens de programação;

## Fique por dentro

Na linguagem Java existem diversos objetos que servem como containers de outros objetos, prontos para serem utilizados, facilitando a manipulação e a passagem de informação entre métodos e aplicações. Estes containers, ou coleções de objetos, possuem funcionalidades que nos permitem executar as operações básicas que precisamos para o seu manuseio, como a inserção, a remoção e a ordenação de dados. Uma característica muito importante destas estruturas de dados é a flexibilidade que existe sobre o formato dos dados que podem ser manuseados, pois não ficamos limitados a guardar um determinado tipo específico de objeto. Por outro lado, de forma a podermos utilizar vários tipos de objetos de maneira segura, a linguagem Java introduziu o conceito de genéricos, que nos permite fazer operações com objetos de vários tipos com “type safety” em tempo de compilação. Com base nisso, vamos ver neste artigo como está construída e como podemos utilizar a plataforma de coleções e genéricos da linguagem Java, recursos empregados em qualquer aplicação.

- Promove a reutilização de software, com interfaces padrão para coleções e algoritmos de dados.

## A API Collections

Para podermos usufruir de todas estas vantagens, temos disponível na linguagem Java vários tipos de objetos que implementam diversos tipos diferentes de coleções. Cada tipo de coleção tem suas características específicas e, de forma a organizá-las, a API de coleções do Java foi dividida em interfaces que agrupam as classes das implementações em um dos seguintes tipos:

- **Collection:** É a raiz da hierarquia das coleções. Define um agrupamento de objetos, denominados de elementos. As suas implementações determinam quando existe ou não ordenação e quando é possível ou não fazer a inserção duplicada de elementos;
- **Set:** Uma coleção onde não são permitidas inserções de elementos repetidos e onde seus elementos não são ordenados. Deste modo, não existe pesquisa através de índices;
- **List:** Uma coleção ordenada, onde são permitidas inserções duplicadas. Os elementos podem ser inseridos em uma posição específica, o que permite fazer pesquisas através de índices. Na sua essência, pode ser visto como um array de tamanho variável;
- **Queue:** Esta interface define uma estrutura de fila, que tipicamente (mas não necessariamente) guarda seus elementos na ordem em que foram inseridos. Ela define o sistema conhecido como *"first-in, first out"*, onde os elementos são inseridos no fim da fila e removidos do seu início;
- **Deque:** Do inglês *"Double Ended Queue"*, representa uma fila com duas terminações. Enquanto uma queue apenas permite inserções no fim da fila e remoções do seu início, um deque permite que inserções e remoções sejam feitas no início e no fim da fila, ou seja, um objeto que implemente Deque pode ser usado tanto como uma fila FIFO (first-in, first-out), quanto uma fila LIFO (last-in, first-out).

## A interface Map

A interface Map representa um tipo abstrato de coleções de objetos e uma instância de uma classe que implemente esta interface é um objeto que mapeia chaves a valores, ou seja, para uma determinada chave (única) existe um valor (único) associado. Assim sendo, um Map não pode ter chaves repetidas e cada chave está associada a, no máximo, um valor.

Mas, se Map é vista como uma coleção, então porque não estende a interface Collection? Esta é uma opção de desenho dos desenvolvedores da linguagem Java, que consideraram que Maps não são coleções e coleções não são Maps. Maps são pares de chave-valor, que não se encaixam na definição de coleções, que são constituídas por elementos simples. No entanto, ao mesmo tempo, podem ser vistas como coleções de chaves/valores, ou pares, e este fato é refletido nas suas operações de coleções, a saber: **keySet()**, que retorna uma estrutura Set com as chaves de mapeamento dos objetos contidos na estrutura Map; **entrySet()**, que retorna uma estrutura Set com os pares de chave-valor dos mapeamentos do objeto Map; e **values()**, que retorna uma estrutura do tipo Collection com todos os valores do mapeamento.

A Listagem 1 mostra um exemplo de código que cria um objeto Hashtable, que implementa a interface Map e mostra as operações de coleções que Map possui com seus respectivos outputs.

Temos ainda duas interfaces de coleção, a **SortedList** e a **SortedMap**, que são meramente versões classificadas/ordenadas das interfaces Set e Map, respectivamente (neste ponto há uma “confusão” sobre a palavra ordenada que será explicada a seguir). Não vamos detalhar estas interfaces porque o mais importante agora é entender o conceito de cada estrutura (Set, List, Queue, Deque e Map) e não as suas variações.

A Figura 1 nos mostra a hierarquia das principais interfaces que implementam Collection e a interface Map.

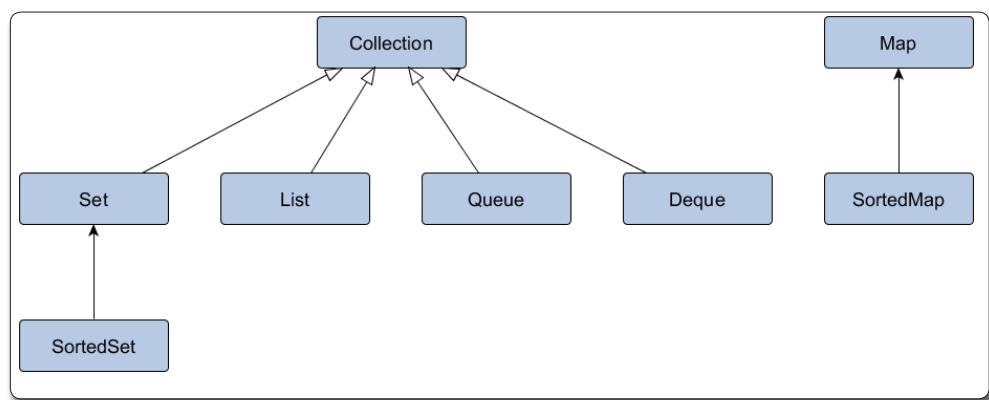
## Ordenado e classificado

Ao explicar as estruturas **SortedList** e **SortedMap**, surgiu a questão sobre um problema de tradução entre o português e o inglês que muitas vezes causa confusão nas pessoas, o que nem sempre é explicado da melhor forma nas documentações em português. Anteriormente foi comentado que as implementações de Set não são ordenadas e as de List são, e posteriormente foi dito que **SortedList** e **SortedMap** são implementações classificadas/ordenadas das estruturas List e Map. Mas, por que dissemos classificadas/ordenadas e não apenas ordenadas?

**Listagem 1.** Exemplo com operações de coleções que existem num objeto do tipo Map.

```

01 public static void testHashTable() {
02     Map<String, String> m = new Hashtable<>();
03     m.put("J", "José");
04     m.put("M", "Miguel");
05     m.put("B", "Bruno");
06     m.put("F", "Fernando");
07     m.put("A", "Alex");
08
09     System.out.println("Key set: " + m.keySet().toString());
10    System.out.println("Entry set: " + m.entrySet().toString());
11    System.out.println("Values: " + m.values().toString());
12 }
-- Output:
Key set: [A, J, F, B, M]
Entry set: [A=Alex, J=José, F=Fernando, B=Bruno, M=Miguel]
Values: [Alex, José, Fernando, Bruno, Miguel]
```



**Figura 1.** Hierarquia das principais interfaces que implementam Collection

Este problema de linguagem acontece porque no inglês existem palavras que significam coisas distintas, mas que são traduzidas da mesma forma para o português. No caso do nosso problema, as palavras são *ordered* e *sorted*. Ambas podem ser traduzidas para ordenado em português. De forma a tentarmos resolver este conflito, vamos traduzir *ordered* para ordenado e *sorted* para classificado, para entendermos o que cada uma significa.

Quando dissermos que uma estrutura é ordenada, é porque a ordem de inserção dos elementos é mantida, como acontece nas listas e arrays. Quando dissermos que uma estrutura é classificada, queremos dizer que um elemento será inserido numa determinada posição da estrutura de acordo com a regra de ordenação da mesma, por exemplo: por ordem (classificação) alfabética, numérica, etc.

Vejamos um exemplo para ajudar na compreensão destas diferenças. A **Listagem 2** mostra a iteração sobre os elementos de uma lista, sobre os elementos de um **HashSet** e sobre os elementos de um **Set**. No fim, é mostrado o output dos testes realizados para validarmos os resultados.

Como podemos ver pela saída gerada ao executarmos esse código, a lista manteve a ordem dos elementos que foram inseridos no input; logo, é uma estrutura de dados ordenada, ou que mantém a ordem. O **HashSet**, por sua vez, não é uma estrutura ordenada. Isso quer dizer que se fizermos uma iteração sobre os dados que foram inseridos, não teremos como garantir a sua ordem, como visto no exemplo, pois esta depende do valor do hash de cada objeto, assim como do tamanho do array de dispersão. Por último, **TreeSet** ordena os dados segundo um algoritmo binário

de balanceamento, o que faz com que a ordem de inserção não seja mantida. Desta forma, é feita uma nova ordenação dos seus dados, o que chamaremos de classificação dos elementos. Assim, vamos denominar este tipo de estrutura de dados de estruturas ordenadas/classificadas. No caso do exemplo, a classificação/ordenação dos elementos foi feita por ordem alfabética porque criamos uma **TreeSet** de **Strings** (**Set<String>**), mas podemos implementar uma árvore com qualquer tipo de classificação/ordenação que pretendermos.

## Implementações das coleções

Agora que já vimos as interfaces do framework Collections, vamos conhecer as classes que implementam estas interfaces no Java. Como você poderá notar, temos disponíveis diversas implementações e essas classes tipicamente têm os nomes no formato *<estilo de implementação><interface>*. Por exemplo, a classe **ArrayList** possui o estilo de implementação em **Array** e implementa a interface **List**. A **Tabela 1** sumariza estas classes.

As implementações de uso geral, ou seja, aquelas que respondem melhor a um maior número de casos de uso, que podemos ver na **Tabela 1**, suportam todas as operações opcionais das interfaces e não têm restrições sobre os tipos de elementos que podem conter (vamos falar sobre este assunto mais à frente, no tópico sobre Generics). Estas implementações não são sincronizadas, ou seja, não suportam o uso concorrente por mais que uma thread de forma segura, mas a classe **Collections** contém wrappers de sincronização que podem ser usados para torná-las sincronizadas, caso seja necessário (veremos um exemplo mais à frente).

**Listagem 2.** Exemplos para distinção entre as palavras sorted e ordered.

```
01 public static void testList() {
02     List<String> l = new LinkedList<>();
03     l.add("José");
04     l.add("Miguel");
05     l.add("Bruno");
06     l.add("Fernando");
07     l.add("Alex");
08
09     Iterator<String> i = l.iterator();
10    System.out.print("List: ");
11    while (i.hasNext()) {
12        System.out.print(i.next() + " ");
13    }
14    System.out.println();
15 }
16
17 public static void testHashSet() {
18     Set<String> s = new HashSet<>();
19     s.add("José");
20     s.add("Miguel");
21     s.add("Bruno");
22     s.add("Fernando");
23     s.add("Alex");
24
25     Iterator<String> i = s.iterator();
26     System.out.print("HashSet: ");
27     while (i.hasNext()) {
28         System.out.print(i.next() + " ");
29     }
30     System.out.println();
31 }
32
33 public static void testTreeSet() {
34     Set<String> s = new TreeSet<>();
35     s.add("José");
36     s.add("Miguel");
37     s.add("Bruno");
38     s.add("Fernando");
39     s.add("Alex");
40
41     Iterator<String> i = s.iterator();
42     System.out.print("TreeSet: ");
43     while (i.hasNext()) {
44         System.out.print(i.next() + " ");
45     }
46     System.out.println();
47 }
48 public static void main(String[] args) {
49     System.out.println("Input: José Miguel Bruno Fernando Alex");
50     testList();
51     testHashSet();
52     testTreeSet();
53 }
```

-- Output:  
Input: José Miguel Bruno Fernando Alex  
List: José Miguel Bruno Fernando Alex  
HashSet: José Miguel Fernando Bruno Alex  
TreeSet: Alex Bruno Fernando José Miguel

Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Tabela 1. Implementações das interfaces do framework Collections

#### Nota

Como os tipos Deque e List implementam Queue e possuem os conceitos de fila, não há necessidade de ter as implementações de Queue na Tabela 1.

## Como escolher o tipo de coleção mais adequado?

Tão importante quanto conhecer os tipos de coleções que temos disponíveis, é saber qual o tipo de coleção mais apropriado para o que queremos fazer. Para definir o tipo de coleção que devemos usar, nos baseamos nas suas características de ordenação e a capacidade de inserção de elementos repetidos.

Uma importante regra a ter em mente é: Não devemos escolher uma classe que implemente recursos que não sejam necessários para o que pretendemos! É provável que tenhamos que pagar por esses recursos com custos que nos causem perda de eficiência. Então, devemos levar em consideração apenas os recursos que realmente necessitamos. Por exemplo, algumas coleções ordenam automaticamente os dados no momento em que são adicionados. Se precisarmos que os dados sejam ordenados, geralmente é mais eficiente escolher uma coleção que já insira os dados de forma ordenada do que usar uma coleção sem ordenação e depois efetuar a ordenação no momento que desejarmos. Por outro lado, se não precisarmos que os dados estejam ordenados, então é mais eficiente adotarmos uma coleção que não seja ordenada.

Com essas informações, vejamos novamente o tópico “A API Collections” no início deste artigo. Se precisarmos de uma coleção que não permita a inserção de elementos duplicados e onde seus elementos não são inseridos de forma ordenada, optaremos por uma implementação do tipo genérico **Set**. Por outro lado, se precisarmos de uma coleção que permita a inserção de elementos repetidos e onde os seus elementos são inseridos de maneira a manter a ordem de inserção, optaremos por uma implementação do tipo **List**. No caso de precisarmos de uma coleção ordenada em fila, onde queiramos usar uma implementação do tipo “first-in, first-out”, podemos optar pelos tipos **Queue** ou **Deque**. No caso de precisarmos manipular elementos no fim da fila, então teremos que optar por uma implementação de **Deque**. Por último, caso pretendamos guardar dados do tipo chave-valor, usaremos uma implementação do tipo **Map**.

Até aqui a escolha parece bastante simples e clara. Mas tendo escolhido qual o tipo de coleção que precisamos, como escolher qual a melhor implementação deste tipo?

## Que implementação de List escolher?

No caso das implementações de **List**, a sua melhor implementação para uso geral é a **ArrayList**, devido a sua melhor performance em relação à **LinkedList**. Vamos analisar cada uma das estruturas individualmente para entender o porquê.

Um **ArrayList**, como o próprio nome indica, é uma lista em array. Sendo assim, podemos ter acesso direto aos seus elementos através do índice e podemos adicionar e remover elementos de forma eficiente apenas no fim da fila, porém, caso seja necessário aumentar o seu tamanho, o custo será alto, já que todo o array será copiado para um novo com maiores dimensões.

Já uma **LinkedList**, como o nome indica, é uma lista ligada de elementos. Para ser mais preciso, a implementação é feita como uma lista duplamente ligada, ou seja, cada elemento sabe quem é o próximo e quem é o anterior. Sendo assim, temos que as inserções e remoções são mais caras em termos de performance, já que cada inserção/remoção exige um maior número de operações em cada elemento a ser guardado/removido, pois é necessário informar a cada célula qual o elemento que lhe sucede e qual o seu elemento anterior. No caso das listas ligadas, o primeiro e último elementos podem ser acessados de maneira direta, mas os restantes terão um custo linear, já que para acessarmos um elemento na posição N da lista, temos que passar por todos os elementos de 0 até N-1, ou seja, o acesso não pode ser feito diretamente através do índice. O maior benefício desta estrutura de dados vem do fato de que as listas ligadas têm um crescimento dinâmico, isto é, podem crescer indefinidamente, sem a necessidade de realocação dos dados, e as inserções e remoções feitas no meio da lista também são mais rápidas, pelo mesmo motivo. A Tabela 2 faz um resumo da complexidade das principais operações em uma lista, comparando as duas estruturas: **ArrayList** e **LinkedList**.

Operação	ArrayList	LinkedList
get(int index)	O(1)	O(n)
add(E element)	O(1)	O(1)
add(int index, E element)	O(n)	O(n)
remove(int index)	O(n)	O(n)
Iterator.remove()	O(n)	O(1)
ListIterator.add(E element)	O(n)	O(1)

Tabela 2. Complexidade das operações em uma lista

Novamente, num caso de uso geral, um **ArrayList** conseguirá satisfazer nossas necessidades de maneira mais eficiente. Por outro lado, no caso de não sabermos antecipadamente o número aproximado de elementos que a lista terá, existindo a possibilidade de a lista ter que crescer muito e várias vezes, ou no caso de necessitarmos de alterações constantes no meio da lista, a melhor opção será uma **LinkedList**.

Também devemos ter atenção ao fato que nenhuma das listas mencionadas são sincronizadas.

Isso quer dizer que, no caso de acesso concorrente por mais que uma thread, devemos optar por uma implementação que garanta esse acesso com segurança ou, como informado anteriormente, utilizar um wrapper de sincronização fornecido pela classe **Collections** na estrutura de lista escolhida, como nos mostra o código a seguir:

```
List<String> l = Collections.synchronizedList(new ArrayList<String>());
```

Um wrapper nos permite adotar uma estrutura que já conhecemos, e com isso evita o trabalho de termos que procurar por uma nova implementação para um caso específico.

## Que implementação de Set escolher?

No caso da estrutura de dados do tipo **Set**, a melhor implementação para uso geral é a **LinkedHashSet**, por ser mais flexível que uma **HashSet** e mais rápida que uma **TreeSet**.

No entanto, vamos começar nossa análise pela **HashSet**. Esta collection é implementada usando uma **Hashtable** e pode ser vista, basicamente, como uma **Hashtable** onde só existem chaves. Logo, não existem elementos duplicados. Por ser baseada nesta estrutura de dados que é organizada por um array de dispersão, os seus elementos não são ordenados. Ademais, os métodos **add()**, **remove()** e **contains()** são de complexidade constante  $O(1)$ , o que é o melhor caso em termos de eficiência.

Os objetos do tipo **TreeSet** são implementados numa estrutura de árvore binária balanceada, utilizando o algoritmo de árvores preta-vermelha, para ser mais preciso. Os seus elementos são ordenados e, por ser uma árvore ordenada binária, significa que os métodos básicos **add()**, **remove()** e **contains()** possuem complexidade logarítmica  $O(\log(n))$ , o que é pior que o caso de complexidade constante  $O(1)$ , mas melhor que o caso de complexidade linear  $O(N)$ . Apesar de ter uma maior complexidade que a estrutura anterior, ela nos oferece funcionalidades para lidar com conjuntos classificados, como retornar o primeiro ou último elemento da classificação e listar os dados de forma classificada, da maneira que achamos mais conveniente através da implementação da interface **Comparable** ou através de um **Comparator**.

Por fim, uma **LinkedHashSet** é implementada com uma **Hashtable** e uma lista duplamente ligada (**LinkedList**) sobre os elementos contidos em sua **Hashtable**. Isso quer dizer que temos a vantagem de ter a complexidade constante na utilização dos métodos básicos, como temos na **HashSet** (com um ligeiro comprometimento devido à adição da lista ligada), mas ao mesmo tempo conseguimos manter a ordem de inserção dos objetos através da ligação feita pela **LinkedList**. Por estar num meio termo entre a eficiência e a flexibilidade de utilização, esta é a implementação que geralmente satisfaz mais casos de uso para estruturas do tipo **Set**.

## Que implementação de Queue escolher?

O caso da **Queue** é semelhante ao da **List**, como podemos ver pela **Tabela 1**. Assim sendo, a melhor implementação para uso geral de filas é a **ArrayDeque**, devido a sua melhor performance

sobre **LinkedList**, pelos mesmos motivos que foram explicados no caso da escolha da melhor estrutura que implementa a interface **List**. **ArrayDeque** é uma estrutura baseada em array (mais eficiente) e a **LinkedList** é uma estrutura de lista duplamente ligada (mais dinâmica, mas menos eficiente).

## Que implementação de Map escolher?

Do mesmo modo que foi explicado nas implementações de **Queue**, o caso da interface **Map** é muito semelhante ao da escolha da implementação de estruturas **Set**, como podemos verificar na **Tabela 1**. Deste modo, para estruturas de dados do tipo **Map**, a melhor implementação para uso geral é a **LinkedHashMap**, por ser mais flexível que uma **HashMap** e mais rápida que uma **TreeMap**. Para mais detalhes sobre esta explicação, veja o tópico “Que implementação de Set escolher?”.

## Ordenação de coleções de dados

Uma das ações mais comuns que temos que fazer com uma coleção de dados é ordenar os elementos nela contidos. Para nos ajudar neste sentido, a linguagem Java disponibiliza duas opções para ordenarmos uma coleção: através da implementação da interface **Comparable**, por parte dos objetos a serem guardados numa coleção, ou através da implementação da interface **Comparator**, num objeto de comparação que serve apenas para este fim.

A interface **Comparable** transmite ordenação natural para classes que a implementam. Esta interface especifica uma relação de ordem, que também pode ser usada para substituir a ordenação natural.

Vejamos um exemplo para ilustrar essa informação. Para isso, suponha que precisamos de uma classe para representar um estudante, onde apenas necessitamos saber qual o seu nome, a idade e uma nota que o estudante tenha no estabelecimento de ensino. Neste cenário, sabemos que iremos manusear coleções de estudantes e que precisaremos listar os estudantes destas coleções pela ordem alfabética dos seus nomes. Dito isso, a **Listagem 3** apresenta um exemplo de implementação desta classe.

Como podemos verificar, a classe **Student** implementa a interface **Comparable**. Sendo assim, deve implementar o método **compareTo()**, que é quem viabiliza a ordenação natural a instâncias desta classe. Neste exemplo, este método apenas usa o **compareTo()** da classe **String**, já que esta classe também implementa a interface **Comparable**. Logo, temos que a ordenação natural de um estudante é estabelecida pela ordem alfabética do seu nome.

Vamos ver o que isso quer dizer observando o resultado desta implementação através de um método de testes que cria uma coleção ordenada e depois itera sobre seus elementos. Para isso, adotamos a opção **TreeSet**, já que precisamos apenas ordenar e iterar sobre a lista. A **Listagem 4** mostra o nosso código de teste e o respectivo output.

Ótimo! Como podemos constatar, inserimos os alunos aleatoriamente, com os nomes não ordenados e o output nos mostra a lista de alunos ordenados alfabeticamente pelos nomes, mas o que aconteceu com os alunos com o mesmo nome?

Por que aparece apenas um aluno com o nome Bruno? Se relembrarmos o que foi dito sobre as coleções de **Set**, vamos ver que **Set** é “Uma coleção onde não são permitidas inserções de elementos repetidos”. Mas o que é um elemento repetido?

**Listagem 3.** Classe que representa um estudante e implementa a interface Comparable.

```

01 package Collections;
02
03 public class Student implements Comparable<Student> {
04     private String _name;
05     private int _age;
06     private int _score;
07
08     public Student(String name, int age, int score) {
09         _name = name;
10         _age = age;
11         _score = score;
12     }
13
14     @Override
15     public String toString() {
16         return "Name:" + _name + " Age:" + _age + " Score:" + _score;
17     }
18
19     @Override
20     public int compareTo(Student student) {
21         return _name.compareTo(student.getName());
22     }
23
24     public String getName() {
25         return _name;
26     }
27
28     public int getAge() {
29         return _age;
30     }
31
32     public int getScore() {
33         return _score;
34     }
35
36 }
```

**Listagem 4.** Código de teste para iteração sobre uma lista ordenada de alunos.

```

01 public static void testComparable() {
02     Set<Student> set = new TreeSet<>();
03     set.add(new Student("José", 34, 100));
04     set.add(new Student("Miguel", 32, 94));
05     set.add(new Student("Bruno", 36, 56));
06     set.add(new Student("Bruno", 30, 56));
07     set.add(new Student("Bruno", 26, 82));
08     set.add(new Student("Fernando", 30, 80));
09     set.add(new Student("Alex", 28, 78));
10
11     Iterator<Student> it = set.iterator();
12     while (it.hasNext()) {
13         System.out.println(it.next());
14     }
15 }

-- Output:
Name: Alex  Age: 28 Score: 78
Name: Bruno  Age: 36 Score: 56
Name: Fernando Age: 30 Score: 80
Name: José  Age: 34 Score: 100
Name: Miguel  Age: 32 Score: 94
```

Os três objetos com nome de *Bruno* não são o mesmo objeto, mas num **Set**, um objeto é igual ao outro se o método **compareTo()** da classe de elementos que são inseridos na coleção retornar o valor 0. Como estamos apenas comparando os nomes dos alunos, todos os objetos **Student** com o mesmo nome serão considerados como sendo o mesmo elemento a ser inserido. Como as coleções **Set** não permitem inserções duplicadas, apenas o primeiro objeto com o nome *Bruno* é inserido.

Deste modo, vamos melhorar o nosso algoritmo de ordenação para fazer com que alunos com o mesmo nome não sejam considerados iguais. Para isso, iremos comparar todos os seus membros de classe, ou seja, vamos comparar os alunos por nome, idade e pontuação. A ordem com que as regras do algoritmo de comparação são feitas é muito importante, pois ela vai decidir qual a ordenação no output da iteração na lista.

Decidimos então que um aluno deve ser ordenado primeiro pelo seu nome, em ordem alfabética, depois pela sua nota, em ordem decrescente, ou seja, da maior nota para a menor, e depois, por sua idade, por ordem crescente. A **Listagem 5** nos mostra a alteração que precisamos fazer no método **compareTo()** da classe **Student** para conseguirmos adicionar estas regras e seu respectivo output.

**Listagem 5.** Alteração das regras do método **compareTo()** da classe **Student**.

```

01     @Override
02     public int compareTo(Student student) {
03         if (!_name.equalsIgnoreCase(student.getName())) {
04             return _name.compareTo(student.getName());
05         }
06         else if (_score != student.getScore()) {
07             return student.getScore() - _score;
08         }
09         else {
10             return _age - student.getAge();
11         }
12     }
```

```
-- Output:
Name: Alex  Age: 28 Score: 78
Name: Bruno  Age: 26 Score: 82
Name: Bruno  Age: 30 Score: 56
Name: Bruno  Age: 36 Score: 56
Name: Fernando Age: 30 Score: 80
Name: José  Age: 34 Score: 100
Name: Miguel  Age: 32 Score: 94
```

Como podemos verificar, agora temos todos os alunos inseridos na ordem que desejávamos. Para alcançarmos este resultado, alteramos o método **compareTo()** para validarmos se um campo é diferente do mesmo campo de outro objeto da mesma classe e, se for, então retornamos o resultado de sua comparação.

Em primeiro lugar, ordenamos o nome através do método **compareTo()** da classe **String**, que, como vimos, nos ordena as strings por ordem alfabética crescente. Em segundo lugar, compararmos as notas dos alunos, devolvendo a nota do aluno a ser comparado menos a nota do aluno atual. Se a nota do aluno a ser comparado for maior que a nota do aluno atual, então o resultado será um

# Dominando o Java Collections Framework e Generics

número positivo, ou seja, este aluno virá depois que o aluno comparado. Explicando de outra maneira, se o resultado do método **compareTo()** for igual a 0, isso significa que os dois objetos são iguais. Se o resultado for menor do que 0, significa que o objeto é “menor” (vem antes) que o objeto comparado, e se o resultado for maior do que 0, significa que o objeto é “maior” (vem depois) que o objeto comparado.

Caso analisemos com atenção a comparação da idade do aluno, vamos notar que invertemos a ordem de comparação para conseguirmos um resultado por ordem crescente, ou seja, se a idade do aluno atual for menor que a idade do aluno a ser comparado, então o resultado será negativo, o que quer dizer que o aluno atual é “menor” (vem antes) que o aluno a ser comparado. Para alterarmos a ordem de crescente para decrescente, basta modificar a ordem da subtração, assim como foi feito para a comparação da idade dos alunos.

Esta ordem criada na classe **Student**, através da implementação da interface **Comparable** e do método **compareTo()**, é chamada de ordem natural do objeto. Mas se quisermos ordenar objetos que não implementem a interface **Comparable**, ou se quisermos alterar a ordem natural do objeto que temos em uma coleção, o que podemos fazer? Para estes casos existe a interface **Comparator**. Um comparator representa uma relação de ordem que pode ser passada para um método de ordenação. Quando implementamos a interface **Comparator**, temos que escrever o método **compare()**, muito semelhante ao que acontece com a interface **Comparable** e o seu método **compareTo()**.

A **Listagem 6** nos mostra o código de uma classe para comparação de idade de alunos que implementa a interface **Comparator** e seu método **compare()**.

**Listagem 6.** Exemplo de classe para comparação da idade dos alunos.

```
01 public class CompareStudentAge implements Comparator<Object> {  
02     public int compare(Object o1, Object o2) {  
03         return ((Student) o1).getAge() - ((Student) o2).getAge();  
04     }  
05 }
```

Note que ela é muito simples e pode ser usada como comparador de objetos em coleções, quer estes objetos implementem a interface **Comparable** ou não. Tendo desenvolvido a nossa nova classe de comparação, vamos alterar o nosso método de testes para usar o novo objeto criado. A **Listagem 7** nos mostra o que fazer para usarmos o nosso comparador de idades de alunos e seu respectivo output.

Como podemos notar, conseguimos ordenar uma listagem de alunos somente pela idade e de forma crescente, sobrepondo a ordenação natural da classe **Student**. No entanto, mais uma vez não temos uma listagem com todos os alunos inseridos na coleção. Isso acontece porque existem alunos com a mesma idade, o que torna os objetos iguais para o nosso comparador, fazendo com que as repetições não sejam inseridas no objeto **Set**.

**Listagem 7.** Exemplo de utilização da classe para comparação da idade dos alunos.

```
01 public static void testComparable() {  
02     Comparator<Object> csa = new CompareStudentAge();  
03     Set<Student> set = new TreeSet<>(csa);  
04  
05     set.add(new Student("José", 34, 100));  
06     set.add(new Student("Miguel", 32, 94));  
07     set.add(new Student("Bruno", 36, 56));  
08     set.add(new Student("Bruno", 30, 56));  
09     set.add(new Student("Bruno", 26, 82));  
10     set.add(new Student("Fernando", 30, 80));  
11     set.add(new Student("Alex", 28, 78));  
12  
13     Iterator<Student> it = set.iterator();  
14     while (it.hasNext()) {  
15         System.out.println(it.next());  
16     }  
17 }
```

-- Output:

```
Name: Bruno  Age: 26 Score: 82  
Name: Alex   Age: 28 Score: 78  
Name: Bruno  Age: 30 Score: 56  
Name: Miguel  Age: 32 Score: 94  
Name: José   Age: 34 Score: 100  
Name: Bruno  Age: 36 Score: 56
```

Como poderíamos resolver este problema em uma situação real? Uma maneira simples e elegante seria adicionar um identificador a cada aluno e inclui-lo em todas as comparações. Assim, nenhum aluno será igual ao outro e todos os alunos sempre serão listados, independente do campo de comparação que utilizarmos.

Vamos então inserir um identificador na classe **Student**, adicionando **private int \_id** às nossas variáveis de classe, refazer a nossa classe de ordenação por idade e comparar com o resultado anterior. A **Listagem 8** apresenta a nova classe de comparação.

**Listagem 8.** Nova classe para comparação da idade de alunos.

```
01 public class CompareStudentAge implements Comparator<Object> {  
02     public int compare(Object o1, Object o2) {  
03         Student s1 = (Student) o1;  
04         Student s2 = (Student) o2;  
05         if (s1.getAge() != s2.getAge()) {  
06             return s1.getAge() - s2.getAge();  
07         }  
08         else {  
09             return s1.getId() - s2.getId();  
10         }  
11     }  
12 }
```

-- Output:

```
Id: 5, Name: Bruno,  Age: 26, Score: 82  
Id: 7, Name: Alex,   Age: 28, Score: 78  
Id: 4, Name: Bruno,  Age: 30, Score: 56  
Id: 6, Name: Fernando, Age: 30, Score: 80  
Id: 2, Name: Miguel,  Age: 32, Score: 94  
Id: 1, Name: José,   Age: 34, Score: 100  
Id: 3, Name: Bruno,  Age: 36, Score: 56
```

Conforme constatado nos resultados obtidos, também expostos nesta listagem, agora temos a listagem completa dos alunos (note que o aluno Fernando voltou a fazer parte do output). Assim, não teremos mais o problema de remoção não intencional de objetos de nossas coleções.

No caso de querermos implementar diferentes tipos de comparações, temos a liberdade de criar quantas classes de comparação desejarmos e as passarmos para nossas coleções para ordená-las da maneira pretendida. Como exercício para colocar o conhecimento adquirido em prática, experimente criar uma classe para ordenação das notas dos alunos, da mesma forma que foi feito para a classe de ordenação da idade, mas com a ordenação decrescente das notas e crescente dos identificadores, e compare os resultados com os valores da **Listagem 9**.

**Listagem 9.** Resultado da ordenação dos alunos por ordem decrescente das notas e crescente dos ids.

```
Id: 1, Name: José, Age: 34, Score: 100
Id: 2, Name: Miguel, Age: 32, Score: 94
Id: 5, Name: Bruno, Age: 26, Score: 82
Id: 6, Name: Fernando, Age: 30, Score: 80
Id: 7, Name: Alex, Age: 28, Score: 78
Id: 3, Name: Bruno, Age: 36, Score: 56
Id: 4, Name: Bruno, Age: 30, Score: 56
```

## Genéricos

Para melhorar o conhecimento adquirido sobre como usar coleções, vamos falar agora sobre um estilo de programação que nos ajuda a trabalhar com coleções de maneira mais flexível e segura: os genéricos! Este recurso adiciona estabilidade em nosso código por fazer com que a maior parte de nossos bugs sejam detectáveis em tempo de compilação. Mas, afinal, o que são genéricos? E o que é programação genérica?

A programação genérica é um estilo de programação na qual os algoritmos são escritos com tipos **a serem especificados posteriormente**, ou seja, são instanciados posteriormente para um tipo específico fornecido como parâmetro. Genéricos são uma facilidade da programação genérica que foi adicionado à linguagem Java na versão 5.0. Este tipo de programação permite que um método opere sobre objetos de vários tipos, fornecendo “type safety” em tempo de compilação e promovendo a eliminação da necessidade de casts. Sem genéricos é possível escrever código com erros de tipificação de objetos que passem pela fase de compilação com sucesso, sendo detectados apenas em tempo de execução. E, como sabemos, corrigir erros de compilação é mais fácil do que corrigir erros de execução.

Para entendermos o problema da não utilização de genéricos em nosso código e qual a motivação para adotá-los, vejamos um exemplo simples, mostrado na **Listagem 10**, que também apresenta o seu respectivo output.

Como podemos constatar, criamos uma lista de objetos onde inserimos primeiro um valor numérico e, em seguida, uma **String**. No momento de visualizar os objetos da lista, fizemos o cast correto do primeiro elemento, mas tivemos um problema no

segundo cast (linha 10), onde tentamos ler uma **String** como se fosse um inteiro. Apesar disso, o programa compila corretamente, sem qualquer problema, mas em tempo de execução teremos uma exceção do tipo **ClassCastException**.

**Listagem 10.** Exemplo de código sem erros de compilação, mas com erro de runtime.

```
01 public static void testNonGenericList() {
02     List list = new ArrayList();
03     list.add(41005);
04     list.add("lasmin");
05
06     Integer i = (Integer)list.get(0);
07     System.out.println("First item:" + i);
08
09 // Código que não causa erros de compilação, mas causa um erro em runtime!
10    i = (Integer)list.get(1);
11    System.out.println("Second item:" + i);
12 }
```

-- Output:

```
First item: 41005
Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot
be cast to java.lang.Integer
at TestClassCollectionsAndGenerics.testNonGenericList(TestClassCollections
AndGenerics.java:10)
at TestClassCollectionsAndGenerics.main(TestClassCollectionsAndGenerics.java)
```

Este é um simples exemplo do problema que podemos evitar usando uma lista genérica. No caso da listagem anterior, temos uma lista com apenas dois elementos, mas imagine um caso mais complexo. Por exemplo: suponha que temos uma aplicação que recebe uma lista de objetos para processar, tendo definido que a lista deve conter apenas objetos do tipo inteiro, mas de tamanho indeterminado (10.000, ou 1.000.000 de elementos). E se nos enviarem elementos que não sejam inteiros? Ok, podemos nos precaver deste problema perguntando qual o tipo do elemento da lista antes de processá-lo, utilizando o operador **instanceof**, através de um comando como **if (list.get(i) instanceof Integer)** ..., mas adicionariam um pouco mais de complexidade ao código, perdendo alguma performance e remediando um problema, mas não impedindo realmente que ele aconteça. Então, vamos conhecer uma melhor solução, com genéricos, como nos mostra a **Listagem 11**.

**Listagem 11.** Exemplo de leitura em uma lista genérica.

```
01 public static void testGenericList() {
02     List<Integer> list = new ArrayList<>();
03     list.add(41005);
04     list.add("lasmin");
05
06     Integer i = list.get(0);
07     System.out.println(i);
08
09     i = list.get(1);
10     System.out.println(i);
11 }
12 }
```

-- Erro:

```
The method add(Integer) in the type List<Integer> is not applicable for the
arguments (String) - line 4
```

Desta vez, criamos uma lista de inteiros. Ao fazer isso, o compilador Java sabe que só é possível inserir e retornar inteiros da lista. Deste modo, a tentativa de inserir uma String, como acontece na linha 4, gera o erro de compilação apresentado no fim da listagem, informando que o método `add()` para uma lista de inteiros não é aplicável a argumentos do tipo `String`. Como o compilador nos garante que só existirão inteiros na lista, não existe a necessidade de casts no momento da leitura dos seus valores, como podemos confirmar nas linhas 7 e 10. Assim, não temos que remediar o problema perguntando os tipos de elementos que estão na lista antes de os ler. Em vez disso, impedimos que o problema possa acontecer, criando uma restrição sobre o que pode ser inserido na lista.

## Como escrever uma classe genérica?

Agora que já sabemos quais as vantagens de se utilizar classes genéricas, vamos aprender como escrever uma. Para isso, criaremos primeiro uma classe da maneira “tradicional” e depois vamos transformá-la numa classe genérica para entendermos quais as diferenças.

Seja `Box` uma classe que representa uma caixa que pode guardar apenas um objeto de cada vez e da qual podemos retornar o objeto que está dentro dela, vejamos como escrever um possível código para esta classe na sua forma tradicional (vide [Listagem 12](#)).

**Listagem 12.** Classe que representa uma caixa que guarda um objeto.

```
01 public class Box {  
02     private Object object;  
03  
04     public void set(Object object) {  
05         this.object = object;  
06     }  
07  
08     public Object get() {  
09         return object;  
10    }  
11}
```

Como podemos verificar, uma caixa pode conter um objeto de qualquer tipo. Isso quer dizer que quando retornarmos o objeto da caixa, teremos que perguntar que tipo de objeto ela contém antes de fazermos o cast para o seu tipo específico e assim podermos trabalhar com ele. Para entendermos melhor este fato, analisemos a [Listagem 13](#), que nos mostra um exemplo de utilização segura da classe `Box` e o output gerado pelo código.

Para verificarmos as diferenças entre as implementações no modelo tradicional e com genéricos, vamos reescrever a classe `Box` de forma a transformá-la numa classe genérica e assim podermos usufruir da segurança do type safety. A [Listagem 14](#) nos mostra a implementação dessa classe, a qual chamamos de `GenericBox`.

Por regra, uma classe genérica é definida no formato `NomeDaClasse<T1, T2, T3, ..., Tn>`, onde `T` representa um tipo de parâmetro da classe. Em `GenericBox`, introduzimos apenas uma variável do tipo `T`, que será o parâmetro recebido pela classe e que irá determinar o tipo de objeto que ela poderá usar.

**Listagem 13.** Código exemplo utilizando a classe `Box`.

```
01 public static void testBox() {  
02     Box box = new Box();  
03  
04     box.set(1324);  
05     Object o = box.get();  
06     if (o instanceof Integer) {  
07         Integer i = (Integer) o;  
08         System.out.println("Box has: " + i);  
09     }  
10  
11     box.set("Another object");  
12     o = box.get();  
13     if (o instanceof String) {  
14         String s = (String) o;  
15         System.out.println("Box has: " + s);  
16     }  
17 }
```

-- Output

```
Box has: 1324  
Box has: Another object
```

**Listagem 14.** Código da classe `Box` na sua forma genérica – `GenericBox`.

```
01 public class GenericBox<T> {  
02     private T t;  
03  
04     public void set(T t) {  
05         this.t = t;  
06     }  
07  
08     public T get() {  
09         return t;  
10    }  
11}
```

Comparando a classe `Box` com `GenericBox`, podemos notar que todas as ocorrências de `Object` foram substituídas pela variável de tipo `T`, que poderá ser especificada com qualquer tipo não primitivo que desejarmos. Esta mesma técnica pode ser usada para criar interfaces genéricas.

Para compreendermos o resultado desta alteração, vejamos como podemos utilizar a classe `GenericBox` na [Listagem 15](#).

Com a nova classe, somos obrigados a identificar, no momento da sua instanciação, que tipo de objeto ela (`GenericBox`) pode guardar. Deste modo seremos impedidos de guardar objetos que sejam de um tipo diferente do previamente acordado e, se tentarmos burlar esse acordo, teremos um erro de compilação e não conseguiremos executar o programa. Portanto, ganhamos a segurança do “type safety” e não temos mais que nos preocupar com casts aos objetos guardados pela caixa.

## Convenções de nomeação para parâmetros de classes genéricas

Uma ressalva importante é que `T` não é uma palavra-chave do Java. Portanto, a classe `GenericBox` poderia ser escrita da mesma forma como se encontra na [Listagem 16](#), sem qualquer tipo de problema.

**Listagem 15.** Código que demonstra a utilização da classe GenericBox.

```

01 public static void testGenericBox() {
02     GenericBox<Integer> box = new GenericBox<>();
03
04     box.set(1324);
05     Integer i = box.get();
06     System.out.println("Box has:" + i);
07
08     box.set(5678);
09     i = box.get();
10     System.out.println("Box has:" + i);
11 }
```

-- Output  
Box has: 1324  
Box has: 5678

**Listagem 16.** Alteração da variável de tipo T para A.

```

01 public class GenericBox<A> {
02     private A a;
03
04     public void set(A a) {
05         this.a = a;
06     }
07
08     public A get() {
09         return a;
10    }
11 }
```

Por que então usamos o T e não outra letra qualquer do alfabeto? Por convenção! E isso é algo muito importante e que não pode ser negligenciado. As convenções melhoram a leitura do código e sua manutenção. T vem de *Type*, ou Tipo, traduzindo do inglês, e poderá ser visto em muitas classes genéricas que pertencem às bibliotecas core do Java.

Sendo assim, é importante conhecermos os padrões usados na linguagem Java para que possamos seguir a mesma nomenclatura, não causando confusões sobre que tipo de parâmetro desejamos em nossas classes genéricas.

A seguir apresentamos os nomes convencionados para os diferentes tipos de parâmetros genéricos:

- E – Element (muito usado pela Java Collections Framework): Elemento;
- K – Key: Chave;
- N – Number: Número;
- T – Type: Tipo;
- V – Value: Valor;
- S, U, V etc. – 2º, 3º, 4º tipos.

#### Nota

Sempre que for escrever uma classe genérica, siga estas regras de convenção de nomenclatura para obter um código mais legível e de fácil manutenção.

Quanto mais complexo é um projeto de software, mais difícil torna-se encontrar os seus bugs. Sendo assim, um planejamento cuidadoso e uma série de testes bem executados podem ajudar a reduzir o número destes, mas não nos garante a sua eliminação por completo.

Felizmente, alguns bugs são mais fáceis de detectar do que outros. Erros de compilação, por exemplo, podem ser detectados precocemente e podemos usar as mensagens que nos são dadas pelo compilador para chegar facilmente à causa do problema e corrigi-lo. Erros de runtime, por outro lado, podem ser muito mais problemáticos, já que não surgem imediatamente e, quando ocorrem, podem acontecer em um ponto distante da real causa do problema, tendo se propagado no código.

Os genéricos adicionam estabilidade ao nosso código por possibilitar que mais bugs sejam detectados em tempo de compilação, tornando mais fácil e rápida a sua correção. Com a adoção dos genéricos na API Java Collections, além de termos disponível uma série de classes e interfaces que tornam mais simples a tarefa de manipulação de coleções de objetos, obtemos mais segurança para o nosso código. Fundamentada na segurança e na flexibilidade, esta API nos oferece um código maduro e estável, com algoritmos de excelente desempenho nas implementações de cada interface. E por serem implementações de estruturas comuns a diversas linguagens de programação orientadas a objetos, a sua utilização também viabiliza um código mais legível e fácil de manter.

Um sólido conhecimento do framework de Collections do Java nos permitirá escolher as melhores estruturas de dados para cada situação, nos ajudando a aprimorar o nosso código em termos de reutilização, desempenho, legibilidade e segurança.

#### Autor



**José Fernandes A. Júnior**

jfjunior@gmail.com



É Mestre em Engenharia Informática pela Faculdade de Ciências e Tecnologias da Universidade Nova de Lisboa (FCT-UNL). Trabalha com Java há 10 anos, mas vem atuando com diversas linguagens, em diversos ramos e áreas, desde core engines de empresas de telecomunicação, até aplicações móveis para Android e iOS. Trabalhou como formador autorizado da Sun Microsystems nos cursos de Java, Unix, Shell Programming e JCAPS. Possui as certificações de Java Swing, Enterprise Java Beans, Java Composite Application Platform Suite (JCAPS), Certificado de Aptidão Profissional (CAP) e Titanium Certified App Developer (TCAD).

#### Links:

**The Java Tutorials: Collections.**

<http://docs.oracle.com/javase/tutorial/collections/interfaces/index.html>

**Choosing the right Collection.**

<http://www.javapractices.com/topic/TopicAction.do?Id=65>

**Generics (Updated).**

<http://docs.oracle.com/javase/tutorial/java/generics/index.html>

#### Você gostou deste artigo?

Dê seu voto em [www.devmedia.com.br/javamagazine/feedback](http://www.devmedia.com.br/javamagazine/feedback)

Ajude-nos a manter a qualidade da revista!



# Refatoração com padrões de projeto e boas práticas

Identifique e refatore códigos mal escritos utilizando padrões de projetos e boas práticas

ESTE ARTIGO É DO TIPO MENTORING

SAIBA MAIS: [WWW.DEVMEDIA.COM.BR/MENTORING-SAIBAMAI](http://WWW.DEVMEDIA.COM.BR/MENTORING-SAIBAMAI)

Projetar e desenvolver um software de qualidade não é uma tarefa trivial. Muitos softwares levam incontáveis meses para serem desenvolvidos, gerando altos custos e jamais chegam a ser entregues. Um dos fatores mais comuns para o insucesso na criação de softwares é o código mal escrito, também conhecido como *code smell*.

Por isso, é importante que a equipe de desenvolvimento se preocupe desde o início do projeto em entregar um software de qualidade, de modo a evitar futuros problemas. No entanto, se um software é entregue com trechos de código mal escritos, ele fatalmente apresentará problemas em tempo de manutenção e até mesmo para o incremento de novas funcionalidades. Neste ponto, o software pode tornar-se inviável, pois um código ruim tende a gerar mais código ruim, além de diminuir a produtividade no desenvolvimento. Este é o momento de identificar os pontos de código mal escrito e refatorá-los.

Para auxiliar e orientar os desenvolvedores neste processo, existem diversos padrões de projetos e boas práticas de programação que podem ser aplicados como base para a refatoração de códigos mal escritos. No entanto, antes de iniciar a refatoração de um código, deve-se levar em consideração qual a característica que torna o código ruim e avaliar qual o padrão ou boa prática que melhor se encaixa para refatoração de tal código.

## Cenário

A manutenção de um software compreende a maior parte de sua vida útil e também costuma ser mais trabalhosa do que o seu desenvolvimento. Portanto, um código mal escrito pode trazer muitos problemas durante o desenvolvimento de novas funcionalidades e manutenção de um software. Neste contexto, este artigo é útil para a identificação de códigos mal escritos, que podem trazer problemas em tempo de manutenção, e também para a aplicação de refatoração nestes códigos, utilizando para isso alguns padrões de projetos e boas práticas.

Diante disso, o objetivo deste artigo é mostrar como identificar códigos mal escritos e como refatorar estes códigos utilizando padrões de projetos e boas práticas de programação. Deste modo, durante o artigo serão mostrados exemplos de códigos mal escritos, a identificação do mesmo e a aplicação da refatoração, apresentando, por fim, os ganhos obtidos.

## Padrões de Projeto

O foco principal deste artigo não é esmiuçar em detalhes os conceitos dos padrões de projetos. No entanto, é importante ter conhecimento das características dos padrões utilizados no decorrer do artigo, de modo que se possa tomar a decisão de qual padrão utilizar em determinadas situações.

Um padrão é composto basicamente por uma solução robusta para problemas comuns. Em outras palavras, um padrão é uma solução que pode ser facilmente adaptada para resolver problemas semelhantes, em diferentes contextos.

Existem basicamente duas maneiras de se utilizar padrões em um projeto de software. Uma delas é a utilização dos padrões já

no momento em que o software está sendo projetado, ou seja, na modelagem do software, antes mesmo do código começar a ser escrito. A outra opção, que será abordada neste artigo, é através da refatoração de código, onde os padrões são utilizados com o objetivo de eliminar trechos de código mal escritos, mas sem modificar seu comportamento.

O livro *Design Patterns: Elements of Reusable Object-Oriented Software*, da Gang of Four (vide **BOX 1**), lista vinte e três padrões de projetos, divididos em três categorias – padrões de projeto criacionais, padrões de projeto estruturais e padrões de projeto comportamentais.

#### Box 1. Gang of Four

Em 1994 os autores Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides publicaram o famoso livro *Design Patterns: Elements of Reusable Object-Oriented Software* (Padrões de Projeto: Soluções Reutilizáveis de Software Orientado a Objetos) com o objetivo de conceituar alguns padrões de projeto para o desenvolvimento de softwares orientados a objetos. Depois disso, estes autores ficaram conhecidos como Gang of Four, ou ainda, GoF.

O livro busca documentar soluções recorrentes para tipos de problemas parecidos, que se agrupam em contextos semelhantes. Os vinte e três padrões abordados no livro estão divididos em três categorias: padrões criacionais, estruturais e comportamentais.

### Padrões de Projeto Criacionais

Os padrões de projeto criacionais, como o próprio nome indica, são padrões utilizados para a criação de objetos. A ideia é separar a lógica de criação de modo que a mesma possa ser encapsulada e reaproveitada facilmente. A seguir são listados alguns exemplos de padrões de projeto criacionais:

- *Abstract Factory*;
- *Builder*;
- *Factory Method*;
- *Prototype*;
- *Singleton*.

### Padrões de Projeto Estruturais

Os padrões de projeto estruturais focam na organização das classes e objetos de um sistema. O propósito é evitar o alto acoplamento entre as classes, modularizando o sistema em componentes com responsabilidades específicas. A seguir são listados alguns exemplos de padrões de projeto estruturais:

- *Adapter*;
- *Bridge*;
- *Composite*;
- *Decorator*;
- *Facade*;
- *Flyweight*;
- *Proxy*.

### Padrões de Projeto Comportamentais

Os padrões de projeto comportamentais definem como as responsabilidades são atribuídas entre as classes do sistema, ou seja, atuam sobre o comportamento das classes.

O objetivo é facilitar a comunicação entre as classes distribuindo as responsabilidades. A seguir são listados alguns exemplos de padrões de projetos comportamentais:

- *Chain of Responsibility*;
- *Command*;
- *Interpreter*;
- *Iterator*;
- *Mediator*;
- *Observer*;
- *State*;
- *Strategy*;
- *Template Method*;
- *Visitor*.

### Boas práticas

Escrever um código limpo e legível não é fácil. O código deve ser frequentemente revisado e refatorado. Para isso, o desenvolvedor deve partir do princípio de que um código sempre pode melhorar, por mais limpo e bem escrito que possa parecer. Com este objetivo, existem diversas boas práticas de programação que, se utilizadas, podem melhorar muito a legibilidade do código de um projeto, facilitando a manutenção e a evolução do mesmo. Lembre-se: um código fácil de ler é um código fácil de manter e evoluir.

O famoso livro *Clean Code*, do autor Robert C. Martin, também conhecido como Uncle Bob, é uma leitura ótima neste sentido, e apresenta conceitos e exemplos interessantíssimos no que diz respeito a código limpo e legível e sua importância para o sucesso dos projetos. O livro também faz uma abordagem interessante sobre os princípios S.O.L.I.D (vide **BOX 2**).

Veremos agora alguns exemplos de boas práticas de programação e como elas podem ajudar na refatoração de códigos mal escritos.

#### Box 2. Princípios S.O.L.I.D

Os Princípios SOLID representam cinco princípios de design de softwares orientados a objetos. Abordados inicialmente por Robert C. Martin, são considerados um conjunto de boas práticas de programação que visa separar responsabilidades e diminuir o acoplamento entre classes, melhorando a legibilidade e a organização do código. A seguir, apresentamos um breve resumo de cada um dos cinco princípios:

1. Single Responsibility Principle: cada classe ou método de um projeto deve ter apenas uma responsabilidade, ou seja, deve fazer apenas uma coisa. Ele deve fazer da melhor forma possível;
2. Open Closed Principle: princípio do aberto/fechado. Diz que as classes do projeto podem ter seu comportamento facilmente estendido, seja por herança, interfaces ou composição. Por outro lado, as classes não devem ser abertas para pequenas modificações;
3. Liskov Substitution Principle: é uma extensão do princípio aberto/fechado e diz que uma classe base deve poder ser substituída por qualquer uma de suas classes derivadas;
4. Interface Segregation Principle: define que uma interface não pode fazer com que uma classe implemente métodos que não sejam dela. Além disso, as interfaces devem ter poucos comportamentos, evitando o alto acoplamento e procurando manter a coesão;
5. Dependency Inversion Principle: este princípio diz que as classes de um nível mais alto de abstração não podem depender de seus detalhes de implementação, ou seja, devem ser independentes.

# Refatoração com padrões de projeto e boas práticas

## Nomes Significativos

Para o desenvolvimento de um projeto, utilizamos vários pacotes, classes, métodos e variáveis. São esses elementos que compõem um projeto e o desenvolvedor é o responsável por dar nome a eles. Diante disso, é muito importante que o desenvolvedor escolha nomes que expressem realmente o que aquela classe ou variável representa dentro do contexto do projeto, com o objetivo de facilitar a leitura do código.

Infelizmente, não é difícil encontrar classes com incontáveis linhas de código, com métodos gigantescos e se deparar com nomes de variáveis que não fazem sentido algum. Isso dificulta, e muito, a manutenção do código, pois consome um tempo bem maior de leitura e interpretação para realmente compreender o que determinada variável ou método faz, ou deveria fazer. Por isso, é preferível perder um pouco de tempo definindo bons nomes para suas classes, variáveis e métodos, do que perder muito tempo depois tentando entender o que cada um destes elementos significa dentro do contexto do projeto.

## Métodos curtos

Os métodos são os elementos do projeto responsáveis por definir o comportamento dos objetos. Eles devem ser curtos e fáceis de entender e, principalmente, devem seguir o princípio da responsabilidade única. Devem fazer uma coisa só e bem feita. Um método muito grande indica que ele provavelmente está fazendo mais de uma coisa. Isso é péssimo para a manutenção por dois motivos: um método extenso é difícil de ler e entender e praticamente impossível de ser reutilizado.

Quando um desenvolvedor se deparar com um código extenso, deve refatorá-lo, extraíndo métodos menores que tenham apenas uma responsabilidade e dando um nome que represente o que estes métodos fazem.

## Métodos com poucos parâmetros

Procure sempre criar métodos que recebam poucos parâmetros. De preferência, nenhum, e no máximo, três. Mais que isso pode ser um indício de que seu método está fazendo mais coisas do que deveria fazer, quebrando o princípio da responsabilidade única. Métodos com muitos parâmetros são também mais difíceis de serem reutilizados.

Caso seja necessário criar métodos com mais de três parâmetros, avalie a possibilidade de encapsular estes parâmetros em um objeto, aumentando assim a possibilidade de reuso.

## Comentários

Comentários devem ser evitados. Uncle Bob diz que um comentário é uma tentativa de explicar um código confuso e mal escrito. Além disso, existe um problema maior ainda com comentários. Imagine que um desenvolvedor escreveu um código confuso para implementar uma regra de negócio e utilizou um comentário para explicar o que o código faz. Pouco tempo depois, a regra de negócio muda e outro desenvolvedor recebe a tarefa de aplicar a alteração no código. Ele o faz, porém, não altera o comentário. Mais tarde,

deve ser agregada uma nova funcionalidade no sistema e aquele trecho de código será impactado. Por fim, um terceiro desenvolvedor recebe a demanda e se depara com um código confuso e que faz uma coisa diferente do que diz o comentário.

No entanto, existem alguns tipos de comentários que podem ser utilizados. É o caso dos *Javadocs*, que podem ser definidos em métodos públicos e classes que sejam disponibilizadas como bibliotecas para uso em outros módulos ou projetos.

Analizando as vantagens e desvantagens, o ideal é evitar os comentários. Afinal, um código bem escrito e fácil de entender não precisa de comentários.

## Refatorando na prática

Para demonstrar na prática a refatoração utilizando padrões de projeto e boas práticas de programação, será utilizado um exemplo de um projeto que requer uma refatoração de código, com o objetivo de melhorar a legibilidade do mesmo e facilitar as futuras manutenções e evoluções que se façam necessárias.

## Entendendo o problema

Suponha que um desenvolvedor recebe a tarefa de refatorar um módulo de um projeto que realiza o cálculo da conta de utilização de um estacionamento. As tarifas para o cálculo seguem os valores da **Tabela 1**.

Tempo	Carro de Passeio	Caminhonete
Até ½ hora	R\$ 5,00	R\$ 7,00
Até 1 hora	R\$ 8,00	R\$ 10,00
Hora adicional	R\$ 3,00	R\$ 5,00
Diária	R\$ 20,00	R\$ 30,00

**Tabela 1.** Valores de tarifas do estacionamento

As regras para o cálculo do valor dessa conta são as seguintes:

- Para os veículos que permanecerem no estacionamento por até meia hora, o valor da conta será correspondente ao da linha **Até ½ hora** da referida tabela;
- Para os veículos que permanecerem no estacionamento por mais de meia hora, até uma hora, o valor da conta será correspondente ao da linha **Até 1 hora**;
- Para os veículos que permanecerem no estacionamento por mais de uma hora, as regras são as seguintes:

- Para os veículos que permanecerem no estacionamento até seis horas, o valor da conta será o valor da linha **Até 1 hora**, mais o valor da linha **Hora Adicional** da **Tabela 1**, multiplicado pelo número de horas adicionais de utilização;
- Para os veículos que permanecerem no estacionamento por mais de seis horas, o valor da conta será correspondente ao valor da linha **Diária**;
- Para os veículos do tipo carro de passeio, devem ser utilizados os valores da coluna **Carro de Passeio**;
- Para veículos do tipo caminhonete, devem ser utilizados os valores da coluna **Caminhonete**;

- Para gerar a conta da utilização do estacionamento, devem ser considerados a hora de entrada do veículo, a hora de saída do veículo e o tipo do veículo.

### **Identificando código mal escrito**

Depois de ler os requisitos do módulo de cálculo da conta de utilização do estacionamento, o próximo passo do desenvolvedor é começar a leitura do código a ser refatorado. O código da classe responsável por calcular o valor da conta do estacionamento está representado na **Listagem 1**.

**Listagem 1.** Classe Conta, contendo a lógica para cálculo do valor da conta do estacionamento.

```

01 public class Conta {
02
03     private long entrada;
04
05     private long saida;
06
07     private Veiculo veiculo;
08
09     public Conta(long entrada, long saida, Veiculo veiculo) {
10         this.entrada = entrada;
11         this.saida = saida;
12         this.veiculo = veiculo;
13     }
14
15     public Double gerarConta() {
16
17         //Obtém o período de utilização em minutos
18         long periodo = (saida - entrada) / 1000 / 60;
19
20         if (veiculo.getTipoVeiculo().equals(TipoVeiculo.CARRO_PASSEIO)) {
21             // gera conta para carros de passeio
22             if (periodo <= 30) {
23                 return 5.0;
24             } else if (periodo > 30 && periodo <= 60) {
25                 return 8.0;
26             } else if (periodo > 60 && periodo / 60 <= 6) {
27                 //Obtém horas adicionais de utilização
28                 long horas = (periodo - 1) / 60;
29                 return horas * 3.0 + 8.0;
30             } else {
31                 return 20.0;
32             }
33         } else if (veiculo.getTipoVeiculo().equals(TipoVeiculo.CAMINHONETE)) {
34             // gera conta para caminhonetes
35             if (periodo <= 30) {
36                 return 7.0;
37             } else if (periodo > 30 && periodo <= 60) {
38                 return 10.0;
39             } else if (periodo > 60 && periodo / 60 <= 6) {
40                 //Obtém horas adicionais de utilização
41                 long horas = (periodo - 1) / 60;
42                 return horas * 5.0 + 10.0;
43             } else {
44                 return 30.0;
45             }
46         }
47         return null;
48     }
49 }
```

Analisando o código, pode-se observar que o método responsável por gerar a conta do estacionamento é confuso e de difícil leitura. Note que o método é grande, indicando a quebra do princípio da responsabilidade única, além de apresentar uma alta complexidade ciclomática, que pode ser observada pela grande quantidade de condicionais. É possível também observar duplicação de código, pois o trecho de código que implementa as regras de cobrança para carros de passeio é praticamente igual ao trecho que implementa as regras para caminhonetes, com exceção das tarifas, que mudam conforme o tipo do veículo. O código também está cheio de números soltos e contém alguns comentários que pouco acrescentam, além de varáveis de nome pouco expressivo. Outro aspecto importante é que o método pode retornar `null`, e retornar `null` nunca é uma boa ideia, pois obriga a todos os clientes deste método tratarem um possível `NullPointerException`.

Apesar de todos os problemas identificados, o código compila e atende às regras especificadas. Porém, há um perigo muito grande com códigos desse tipo. Imagine que em determinado momento, o estacionamento passe a trabalhar com veículos de outros tipos, como motocicletas, vans, ônibus e caminhões. Para cada novo tipo de veículo, teria de ser adicionado um novo condicional, replicando mais código, para atender o cálculo de acordo com as tarifas do novo tipo de veículo. E se o estacionamento passasse a contar com planos de pernoite e mensalidade, o método precisaria de mais condicionais dentro do tratamento de cada tipo de veículo para implementar as novas regras.

Neste cenário, o código tende a crescer de forma exponencial, chegando a um ponto em que fique inviável manter ou evoluir o projeto. Lembre-se que código ruim gera código ruim.

Para completar o código do projeto, a classe que representa um veículo é apresentada na **Listagem 2** e a **Listagem 3** apresenta uma enumeração contendo os tipos de veículos.

**Listagem 2.** Classe que representa um veículo.

```

01 public class Veiculo {
02
03     private String placa;
04
05     private String modelo;
06
07     private TipoVeiculo tipoVeiculo;
08
09     // gets e sets omitidos
10 }
```

**Listagem 3.** Enumeração com os tipos de veículos.

```

01 public enum TipoVeiculo {
02     CARRO_PASSEIO,
03     CAMINHONETE;
04 }
```

A **Figura 1** representa o diagrama de classes da versão atual do módulo de cálculo da conta de utilização do estacionamento. Com o intuito de facilitar a compreensão do conteúdo apresentado a partir deste ponto, conforme o código for sendo refatorado, o diagrama será evoluído, para ilustrar a utilização de alguns padrões de projetos e boas práticas na refatoração.

**Listagem 4.** Aplicando boas práticas na refatoração.

```

01 public Double gerarConta() {
02
03     long periodoEmMinutos = obtemPeriodoDeUtilizacaoEmMinutos();
04
05     //lógica omitida...
06
07     throw new IllegalStateException("Tipo de veículo indefinido");
08 }
09
10 private long obtemPeriodoDeUtilizacaoEmMinutos() {
11     return (saida - entrada) / 1000 / 60;
12 }
```

## Aplicando boas práticas na refatoração

Feita a análise e a identificação dos problemas no código, o desenvolvedor deve agora começar a refatoração. O primeiro passo é aplicar boas práticas para eliminar alguns problemas levantados. Como é possível observar na **Listagem 1**, o método responsável por gerar a conta do estacionamento começa com uma linha um pouco

confusa para obter o período que o veículo utilizou o estacionamento. No entanto, o ideal seria eliminar o comentário, extrair a lógica para obter o período de utilização em um método privado e renomear a variável para indicar que o período armazenado é em minutos.

Outro ponto a se observar, é que o método pode retornar **null**, o que nunca é uma boa ideia. Analisando o código, pode-se constatar que o método só retorna **null** se o veículo não tiver nenhum valor definido para o atributo que representa o tipo de veículo ou se o valor for diferente de **CAMINHONETE** ou **CARRO\_PASSEIO**. Neste caso, o método pode lançar uma exceção indicando que o estado do objeto é inválido. A **Listagem 4** mostra como fica o código após a refatoração.

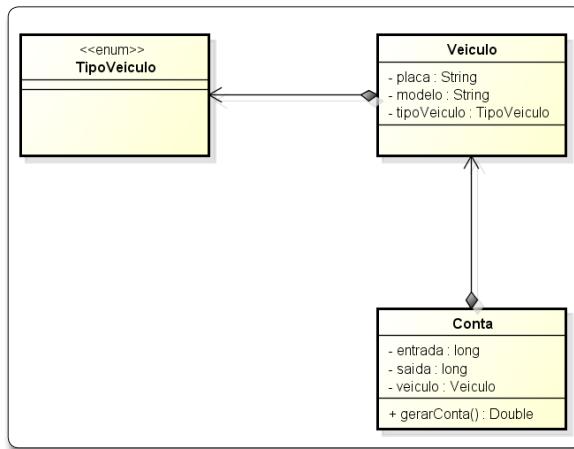
Note que o código pertinente à lógica de geração da conta do estacionamento foi omitido dessa listagem, pois neste ponto da refatoração ele ainda não foi alterado.

## Aplicando o padrão Strategy na refatoração

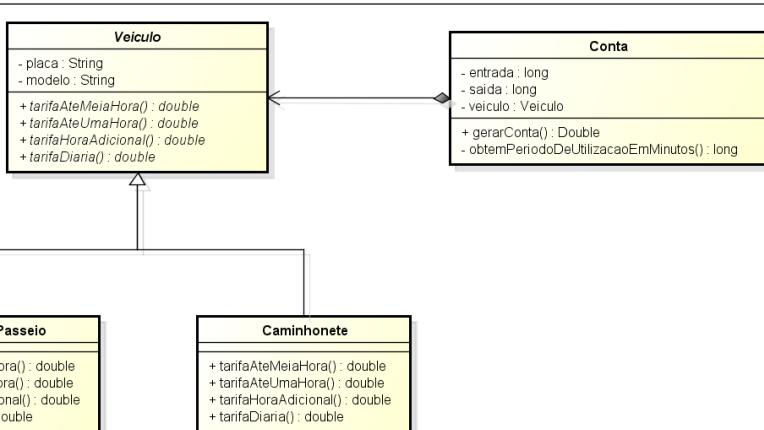
Agora, vamos começar a eliminar o código replicado e alguns condicionais. Uma boa abordagem para a eliminação de código duplicado e dos condicionais é a utilização do padrão **Strategy**. A ideia é utilizar composição e uma estrutura de herança para criar subclasses que especializem o comportamento dos tipos de veículo e onde a execução possa ser delegada para estas subclasses em tempo de execução. Neste caso, a classe que representa um veículo passará a ser uma classe abstrata e possuirá duas subclasses, uma para representar um carro de passeio e outra para representar uma caminhonete.

Para dar início à refatoração utilizando o padrão **Strategy**, a classe **Veiculo** será modificada e terá agora métodos abstratos para recuperar o valor da tarifa, conforme o período que o veículo permanecer no estacionamento. A **Figura 2** ilustra a evolução do diagrama de classes para atender as alterações.

Este modelo permite que, caso a aplicação passe a suportar novos tipos de veículos, não seja necessário criar mais código duplicado e métodos extensos. Basta criar uma nova subclasse de veículo



**Figura 1.** Diagrama de classes do módulo de cálculo da conta de utilização do estacionamento



**Figura 2.** Diagrama de classes utilizando Strategy na refatoração

e implementar os métodos necessários. Em outras palavras, a modelagem segue o princípio aberto para expansão e fechado para alteração.

Com a alteração, o método responsável por gerar a conta de utilização do estacionamento não precisa mais duplicar código para cada tipo de veículo. A **Listagem 5** apresenta o código da classe responsável pela geração da conta após a refatoração utilizando o padrão *Strategy*. Note que o código melhorou, mas ainda apresenta um grande número de condicionais para calcular o valor da conta de acordo com tempo. Para resolver isso, será aplicado outro padrão de projeto, abordado mais à frente.

**Listagem 5.** Classe responsável pela geração da conta, após a refatoração com *Strategy*.

```
01 public class Conta {  
02  
03     private long entrada;  
04  
05     private long saida;  
06  
07     private Veiculo veiculo;  
08  
09     public Double gerarConta() {  
10  
11         long periodoEmMinutos = obtemPeriodoDeUtilizacaoEmMinutos();  
12  
13         if (periodoEmMinutos <= 30) {  
14             return veiculo.tarifaAteMeiaHora();  
15         } else if (periodoEmMinutos > 30 && periodoEmMinutos <= 60) {  
16             return veiculo.tarifaAteUmaHora();  
17         } else if (periodoEmMinutos > 60 && periodoEmMinutos / 60 <= 6) {  
18             // Obtém horas adicionais de utilização  
19             long horas = (periodoEmMinutos - 1) / 60;  
20             return horas * veiculo.tarifaHoraAdicional() + veiculo.tarifaAteUmaHora();  
21         } else {  
22             return veiculo.tarifaDiaria();  
23         }  
24     }  
25  
26     private long obtemPeriodoDeUtilizacaoEmMinutos() {  
27         return (saida - entrada) / 1000 / 60;  
28     }  
29  
30 }
```

Para aplicar o padrão *Strategy*, foi alterada a classe que representa um veículo, de modo que ela fosse especializada em duas novas subclasses. A **Listagem 6** apresenta a nova estrutura da classe que representa um veículo, contendo agora apenas atributos pertinentes aos veículos em geral e métodos abstratos que contêm comportamentos que variam conforme o período de utilização do estacionamento. A **Listagem 7** apresenta a subclasse que representa um carro de passeio e a **Listagem 8** apresenta a subclasse que representa uma caminhonete. Nas classes **CarroDePasseio** e **Caminhonete**, os métodos abstratos da classe **Veiculo** são implementados de modo a especializar a lógica das tarifas por tempo de utilização de acordo com cada tipo específico de veículo.

**Listagem 6.** Classe que representa um veículo, após a refatoração com *Strategy*.

```
01 public abstract class Veiculo {  
02  
03     private String placa;  
04     private String modelo;  
05  
06     public abstract double tarifaAteMeiaHora();  
07     public abstract double tarifaAteUmaHora();  
08     public abstract double tarifaHoraAdicional();  
09     public abstract double tarifaDiaria();  
10  
11     // gets e sets omitidos  
12 }
```

**Listagem 7.** Classe que representa um carro de passeio.

```
01 public class CarroDePasseio extends Veiculo {  
02  
03     @Override  
04     public double tarifaAteMeiaHora() {  
05         return 5.0;  
06     }  
07  
08     @Override  
09     public double tarifaAteUmaHora() {  
10         return 8.0;  
11     }  
12  
13     @Override  
14     public double tarifaHoraAdicional() {  
15         return 3.0;  
16     }  
17  
18     @Override  
19     public double tarifaDiaria() {  
20         return 20.0;  
21     }  
22  
23     // Demais atributos e métodos específicos da classe omitidos...  
24 }
```

**Listagem 8.** Classe que representa uma caminhonete.

```
01 public class Caminhonete extends Veiculo {  
02  
03     @Override  
04     public double tarifaAteMeiaHora() {  
05         return 7.0;  
06     }  
07  
08     @Override  
09     public double tarifaAteUmaHora() {  
10         return 10.0;  
11     }  
12  
13     @Override  
14     public double tarifaHoraAdicional() {  
15         return 5.0;  
16     }  
17  
18     @Override  
19     public double tarifaDiaria() {  
20         return 30.0;  
21     }  
22  
23     // Demais atributos e métodos específicos da classe omitidos...  
24 }
```

# Refatoração com padrões de projeto e boas práticas

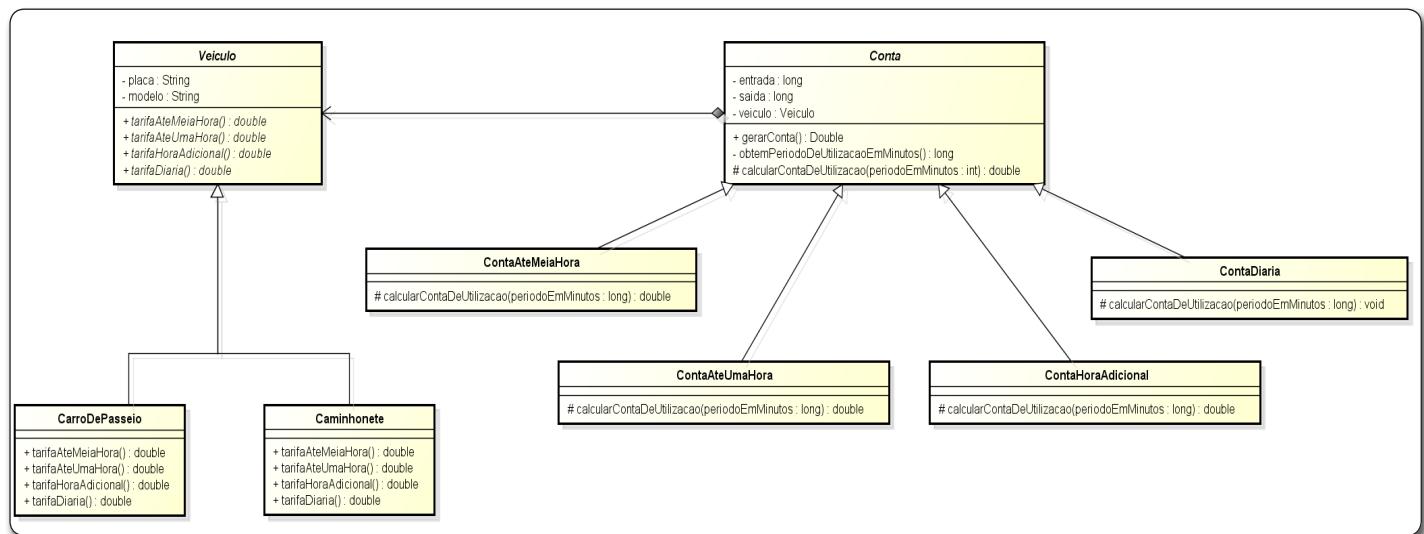


Figura 3. Diagrama de classes utilizando Template Method na refatoração

## Aplicando o padrão Template Method na refatoração

Finalmente, será demonstrada a aplicação do padrão *Template Method*, para reduzir o restante da lógica condicional da classe que gera a conta do estacionamento. Este padrão consiste na utilização de herança para variar partes de um algoritmo. Basicamente, a classe principal possui uma lógica genérica, onde apenas algumas partes mudam. E esses pontos do algoritmo que variam são especializados nas subclasses.

No exemplo da geração da conta do estacionamento, temos uma lógica genérica que obtém o tempo de utilização e, de acordo com as tarifas do tipo do veículo, calcula o valor da conta sobre o tempo que o veículo permaneceu no estacionamento. Porém, a tarifa varia para cada faixa de tempo de utilização. Com essa leitura, pode-se delegar o cálculo para subclasses da classe **Conta**, permitindo o isolamento da lógica do cálculo para classes específicas, facilitando a manutenção e melhorando a legibilidade e organização do código. Para tanto, a classe **Conta** deve passar a ser abstrata, forçando a implementação do cálculo do valor da conta nas subclasses específicas para cada faixa de tempo.

A Figura 3 ilustra como fica o diagrama de classes do módulo de cálculo do valor da conta de utilização do estacionamento com a aplicação do *Template Method* após a refatoração do código.

A partir dessa refatoração, além da eliminação dos condicionais no código para gerar a conta de utilização do estacionamento, resolvemos também o problema do crescimento da complexidade ciclomática do código, em caso da adição de novas regras de tarifa como, por exemplo, valores para pernoites e mensalidades.

Para a utilização do *Template Method*, foram criadas quatro subclasses para a classe **Conta**, uma para cada tarifa entre as faixas de valores definidas na Tabela 1. Assim, caso alguma nova tarifa seja criada, basta criar uma nova subclass e implementar nela a regra representando a tarifa. Com isso, o código fica isolado e cada classe tem apenas a responsabilidade de implementar a regra específica daquela tarifa.

Dito isso, a seguir serão apresentadas as subclasses criadas para implementar as regras de cada tarifa. A Listagem 9 ilustra a subclass responsável por calcular a tarifa de veículos que permanecem no estacionamento até meia hora. A Listagem 10 apresenta a classe para veículos que permanecem de meia hora até uma hora. A Listagem 11 mostra a classe com o cálculo para veículos que ficam até seis horas, onde temos o cálculo de horas adicionais de utilização.

Listagem 9. Classe que representa o cálculo de tarifa para até meia hora de utilização.

```
01 public class ContaAteMeiaHora extends Conta {  
02  
03     @Override  
04     protected double calcularContaDeUtilizacao(long periodoEmMinutos) {  
05         return getVeiculo().tarifaAtéMeiaHora();  
06     }  
07 }
```

Listagem 10. Classe que representa o cálculo de tarifa para mais de meia hora, até uma hora de utilização.

```
01 public class ContaAteUmaHora extends Conta {  
02  
03     @Override  
04     protected double calcularContaDeUtilizacao(long periodoEmMinutos) {  
05         return getVeiculo().tarifaAtéUmaHora();  
06     }  
07  
08 }
```

Listagem 11. Classe que representa o cálculo de tarifa para mais de uma hora, até seis horas de utilização.

```
01 public class ContaHoraAdicional extends Conta {  
02  
03     @Override  
04     protected double calcularContaDeUtilizacao(long periodoEmMinutos) {  
05         long horasAdicionais = (periodoEmMinutos - 1) / 60;  
06         return horasAdicionais * getVeiculo().tarifaHoraAdicional() + getVeiculo()  
07             .tarifaAtéUmaHora();  
08     }  
09 }
```

E, por fim, a **Listagem 12** apresenta a classe com o cálculo de veículos que ficam mais de seis horas no estacionamento, configurando a cobrança de uma diária.

Encerrando nosso estudo, a **Listagem 13** apresenta a classe **Conta**, responsável pela geração do valor da conta de utilização do estacionamento, após a aplicação de boas práticas e dos padrões *Strategy* e *Template Method* para refatoração do código. Observe que a classe ficou limpa, com pouco código, fácil de ler e entender e, consequentemente, fácil de manter e evoluir.

Enfim, o código agora está preparado para suportar novas regras de tarifas e novos tipos de veículos, de modo a crescer de forma organizada, sem duplicação de código e sem adição de condicionais.

**Listagem 12.** Classe que representa o cálculo de tarifa para mais de seis horas de utilização.

```
01 public class ContaDiaria extends Conta {  
02  
03     @Override  
04     protected double calcularContaDeUtilizacao(long periodoEmMinutos) {  
05         return getVeiculo().tarifaDiaria();  
06     }  
07  
08 }
```

**Listagem 13.** Classe Conta, após a refatoração com boas práticas e padrões de projetos.

```
01 public abstract class Conta {  
02  
03     private long entrada;  
04  
05     private long saida;  
06  
07     private Veiculo veiculo;  
08  
09     public double gerarConta() {  
10         long periodoEmMinutos = obtemPeriodoDeUtilizacaoEmMinutos();  
11         return calcularContaDeUtilizacao(periodoEmMinutos);  
12     }  
13  
14     private long obtemPeriodoDeUtilizacaoEmMinutos() {  
15         return (saida - entrada) / 1000 / 60;  
16     }  
17  
18     protected abstract double calcularContaDeUtilizacao(long periodoEmMinutos);  
19  
20     protected Veiculo getVeiculo() {  
21         return veiculo;  
22     }  
23  
24 }
```

O desenvolvedor de software conta hoje com uma vasta documentação de padrões de projetos consolidados, que podem ajudar a resolver problemas de maneira limpa e elegante. A adoção de boas práticas também ajuda muito na criação de códigos bem escritos. Mas como ponto de partida, é indispensável que o desenvolvedor tenha como uma das suas principais metas escrever o código pensando na manutenção e na evolução do mesmo, de

forma que outro profissional consiga ler e entender rapidamente o que o código escrito faz.

Além disso, a refatoração deve ser contínua, partindo do princípio que, por mais que um código pareça limpo e bem escrito, sempre pode ser melhorado. Boas práticas como nomes significativos e extração de métodos para encapsular responsabilidades em métodos menores podem ser aplicadas constantemente nos projetos. Sendo assim, sempre que você começar a trabalhar em alguma nova funcionalidade ou melhoria, procure tentar aplicar boas práticas para deixar o código melhor do que quando você começou a trabalhar nele. Se você encontrar um método com um nome pouco intuitivo, altere esse nome. Caso se depare com um método muito grande e confuso, quebre-o em métodos menores e mais coesos. Se encontrar um comentário desatualizado, remova-o do código.

Os padrões de projeto também podem e devem ser utilizados, tanto em tempo de modelagem, quanto em tempo de refatoração. Se você recebe uma demanda para evoluir um módulo de um sistema e se depara com um código confuso, com trechos duplicados e excesso de lógica condicional, considere a aplicação de algum padrão para refatorar o código e torná-lo mais simples, antes de adicionar mais complexidade ao mesmo. Talvez você até perca algum tempo neste momento, mas com certeza será recompensado no médio e longo prazo.

O importante é conseguir identificar o código mal escrito e evitar que aquele código cresça de maneira desordenada e confusa, a ponto de tornar a evolução ou manutenção do projeto impraticável. Neste caso, o profissional deve parar de produzir código e avaliar se algum padrão de projeto se encaixa na refatoração do mesmo, para permitir que o código possa evoluir de maneira clara e organizada.

## Autor



**Alex Radavelli**

*alexradavelli@gmail.com*

Desenvolvedor Java EE com mais de cinco anos de experiência.

Bacharel em Ciência da Computação e cursando MBA em Gerenciamento de Projetos. Possui a certificação Oracle Certified Associate, Java SE 7 Programmer.



## Links:

**Artigo sobre os princípios SOLID.**

*<http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>*

## Você gostou deste artigo?

Dê seu voto em *[www.devmedia.com.br/javamagazine/feedback](http://www.devmedia.com.br/javamagazine/feedback)*

Ajude-nos a manter a qualidade da revista!



# Elaborando projetos com a Arquitetura Orientada a Eventos

Evolua sua arquitetura de integração com os conceitos da Event Driven Architecture

Uma das maneiras de tentar estabelecer uma comunicação organizada e controlada entre os componentes que formam o conjunto de sistemas de uma empresa/departamento seria utilizar a ocorrência de um fato, ou seja, um evento, pois desta forma teríamos a comunicação entre as partes mantida com um baixo acoplamento, onde emissores e assinantes estariam ligados apenas pela criação destes eventos.

Além dos eventos definirem o momento em que a integração pode ocorrer, possuem as informações necessárias que serão trocadas entre as peças, efetivando as integrações. No acontecimento de um evento, ações podem ser tomadas pelos interessados sem a necessidade de qualquer tipo de atuação por parte do gerador do evento.

Mas dentro do nosso contexto de tecnologia: o que são estes eventos? A resposta é: depende! Apesar de a resposta parecer uma tentativa de ludibriar alguém que você gostaria de convencer, ela realmente depende do contexto envolvido, ou seja, do modelo do domínio de negócio que estamos nos referindo. O evento é algo relevante para o negócio da empresa, podendo ser: a ocorrência de uma venda, o cadastro de um novo cliente, a desistência de uma assinatura de revista, saques em locais geograficamente distantes do mesmo correntista, ou o valor acumulado de pedidos realizados dentro de um período.

A definição do que pode ser considerado um evento está muito mais orientada ao negócio da empresa do que a decisões técnicas. É claro que é possível também estabelecer eventos de natureza técnica como, por exemplo: o tempo médio da execução de transações

## Fique por dentro

Neste artigo apresentaremos um exemplo prático para demonstrar o que é e como pode ser concebida uma Arquitetura Orientada a Eventos. Para isso, implementaremos todas as partes relevantes de uma arquitetura completa de integração, incluindo componentes que compõem a Arquitetura Orientada Serviços (SOA). Em seguida, complementaremos nosso estudo abordando algumas ferramentas que ajudam a conceber a EDA (Event Driven Architecture).

Portanto, este artigo é útil para profissionais envolvidos em ambientes nos quais há necessidade de comunicação entre os participantes (sistemas e componentes) que formam todo o conjunto de sistemas de software da empresa.

ou a quantidade de execuções de um serviço no mês. Porém, na maioria dos cenários, ao estabelecer uma arquitetura orientada a eventos, a prioridade fica para a definição e captura de eventos do negócio em questão, para que a empresa tenha mais benefícios sobre o orçamento investido em tecnologia. Os eventos de natureza técnica (como tempo de resposta) ganham maior prioridade quando afetam diretamente os negócios da empresa.

Em uma arquitetura orientada a eventos, colocamos obviamente como ator principal o evento, estabelecendo assim o elo que causará a integração entre os sistemas e componentes. Vale ressaltar ainda que um evento é autocontido, isto é, carrega consigo a informação de quando está preparado para ser disparado e quais informações deve conter.

O que de fato a Arquitetura Orientada a Eventos (EDA – *Event Driven Architecture*) deve promover é uma estrutura que possa estabelecer o alicerce de apoio à integração entre os sistemas e/ou

componentes do ambiente – usufruindo da ocorrência de evento –, de forma que ainda se mantenha um baixo acoplamento entre os participantes.

Obviamente, assim como a definição de um modelo de domínio para a criação de uma aplicação, a modelagem de quais eventos serão tratados deve fazer parte da análise da solução. Principalmente porque estes eventos serão utilizados por vários sistemas distintos da corporação, o que sugere colocá-los como parte do modelo canônico, junto com as entidades de negócio e mensagens comuns utilizadas pelos diferentes sistemas da empresa.

Dentro do ambiente de uma empresa com vários sistemas distintos, existem potenciais emissores e assinantes destes eventos. Neste cenário, os emissores de eventos possuem as informações e o contexto associado ao disparo, fazendo o evento existir de forma consistente e válida para consumo por todos os interessados.

A complexidade para a definição de uma estrutura que possa auxiliar o controle e organização entre emissores e assinantes de eventos, estabelecendo uma Arquitetura Orientada a Eventos, vai depender da maturidade de integração existente na empresa. É nesse momento que a adoção e estabelecimento de uma Arquitetura Orientada a Serviços mostra suas vantagens. Ao temos um mediador responsável por tratar as integrações, que centraliza os serviços existentes no ambiente, criar uma arquitetura para eventos se torna mais fácil.

Pelo barramento de serviços podemos identificar os serviços expostos de sistemas que são potenciais emissores (origem de eventos), deixando a responsabilidade de gerenciamento dos eventos ao barramento de serviços. É possível até ir além. Ao realizar o cruzamento de eventos de um ou mais sistemas, pode-se estabelecer a criação de eventos mais complexos, que são derivados do cruzamento de informações de outros, realizando o conceito de processamento de eventos complexos (CEP) – vide **BOX 1**.

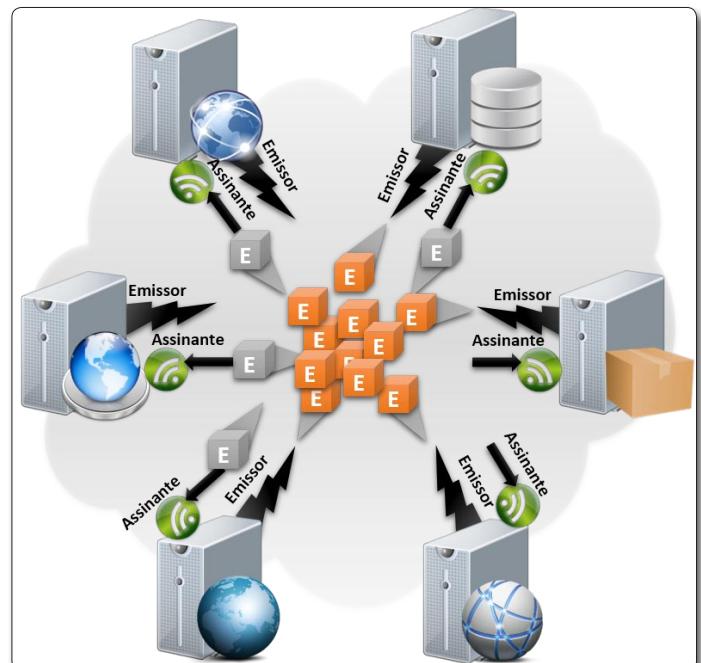
#### BOX 1. Complex Event Processing

Conceito que consiste na captura de informações no formato de eventos significativos que acontecem na organização. Possibilita a realização de ações em tempo real como, por exemplo, identificar possíveis fraudes em andamento durante transações financeiras.

Os eventos são identificados e disparados durante a análise da “corrente” (*stream*) de informações que trafega no barramento durante as execuções de serviços que ocorrem nas transações de negócio da empresa. Neste cenário de integrações é fácil perceber a simplificação que se obtém possuindo um ambiente SOA bem organizado, pois o controle das integrações já deve existir. O que falta é aproveitar a oportunidade deste fluxo de informações para identificar, criar e disparar eventos. Por outro lado, em um ambiente onde as integrações entre sistemas não possuem um controle consistente e o desenho destas integrações é normalmente definido em salas de reuniões em que apenas duas partes estão presentes, aumenta a complexidade para estabelecer uma arquitetura orientada a eventos. O descontrole sobre a integração acaba se espalhando, resultando em: eventos duplicados, eventos

específicos por sistema, desconhecimento de assinantes e acoplamento mais alto entre as partes.

Na **Figura 1** podemos analisar uma ilustração de como seria um ambiente previamente concebido estabelecendo uma arquitetura orientada a eventos sem nenhuma organização e controle sobre as integrações.



**Figura 1.** Integração entre sistemas utilizando eventos sem o apoio do barramento de serviços

Observe que os eventos são emitidos diretamente entre os sistemas do ambiente por causa da falta de um mediador que tenha o conhecimento e controle de todos que necessitam se integrar no ambiente. Mesmo ainda sendo possível, o esforço será maior para estabelecer e gerenciar este cenário.

Já na **Figura 2** podemos observar uma ilustração de uma Arquitetura Orientada a Eventos funcionando com o apoio de um barramento de serviços de uma arquitetura SOA. Neste cenário, o barramento é responsável por centralizar a emissão dos eventos do ambiente, realizando a inspeção do fluxo de informações que está trafegando durante a execução dos serviços expostos pelos sistemas. Assim que identificado e criado um evento pelo barramento, os assinantes podem, em seguida, recebê-lo.

A forma como o barramento de serviços analisa os *streams* de informação, identifica e lança os eventos, é uma definição associada ao desenho da Arquitetura Orientada a Eventos. Apesar de facilitar, não é obrigatório possuir já estabelecida uma arquitetura orientada a serviços para concretizar a arquitetura orientada a eventos. As duas arquiteturas podem se complementar. Portanto, se o SOA já estiver presente, é possível evoluir sua arquitetura de integração de serviços para se beneficiar dos eventos. Enfim, SOA e EDA podem coexistir, mas não é necessário possuir SOA para implantar EDA, e vice-versa.

# Elaborando projetos com a Arquitetura Orientada a Eventos

A característica comum de um ambiente SOA é a existência de consumidores e provedores, ao passo que em EDA possuímos assinantes e emissores. Como um diferencial, na arquitetura EDA os emissores não sabem quem são os seus assinantes, viabilizando assim uma integração de acoplamento baixo.

Na implementação de uma Arquitetura Orientada a Eventos, uma das opções mais simples seria utilizar filas de mensagerias (normalmente disponíveis em ferramentas ESB), através de tópicos, colocando em ação o padrão de comunicação editor/assini-

nante (publish/subscriber). A outra opção seria utilizar ferramentas especializadas no gerenciamento e tratamento de eventos, como o Esper, que iremos analisar em seguida no nosso cenário de exemplo.

## Arquitetura Orientada a Eventos na prática

O nosso exemplo de Arquitetura Orientada a Eventos provém de um cenário simples de e-commerce, onde uma loja on-line fornece seus produtos. Apesar de ser um cenário simples de compras via internet, implementamos o fluxo de integração das

informações desde o início do cenário transacional, passando pelo sistema de vendas, barramento de serviços, processamento de eventos, chegando até a visualização do monitoramento das atividades de negócios. Desta maneira, podemos ter uma melhor demonstração prática do papel dos eventos dentro desta arquitetura.

A visão lógica e completa da arquitetura implementada nesta solução pode ser observada na **Figura 3**.

Nesta solução temos os clientes realizando os pedidos no site da empresa ou com representantes de vendas via telefone, o que é ilustrado nos quadros número um e dois. As compras são registradas através do serviço *Registrar Pedido*, que está exposto no Barramento de Serviços Corporativos (ESB) da empresa (vide quadro número três). O barramento realiza o roteamento do request para o sistema atual, responsável na empresa por manusear e gerenciar as transações de vendas (vide quadro número quatro).

Até este momento observamos que não temos nada muito diferente de uma Arquitetura Orientada a Serviços, onde cada sistema realiza a sua tarefa (motivo pelo qual foi criado) e as integrações ocorrem de forma controlada pelo Barramento de Serviços. Neste são tratados todos os aspectos de integração, desonerando os sistemas de tarefas como: transformações, enriquecimento, mapeamento, roteamento, orquestração, protocolos e adaptadores de tecnologia.

Os pedidos irão para o sistema denominado SalesCenter, que após confirmar com sucesso o registro da compra, eventos de interesse da empresa nesta transação de registro de compra serão capturados e lançados pelo ESB para o componente que realiza o processamento de eventos (vide quadro número cinco). Aqueles que tenham alguma relevância informacional serão, também, armazenados em uma base de dados (vide quadro número seis). O sistema de monitoramento, denominado MiniJS-BAM, faz uso destes dados para apresentar informações das atividades de negócios que estão ocorrendo na empresa simultaneamente à ocorrência dos fatos (vide quadros de número sete e oito).



Figura 2. Integração entre sistemas utilizando eventos com o apoio de um barramento de serviços (SOA)

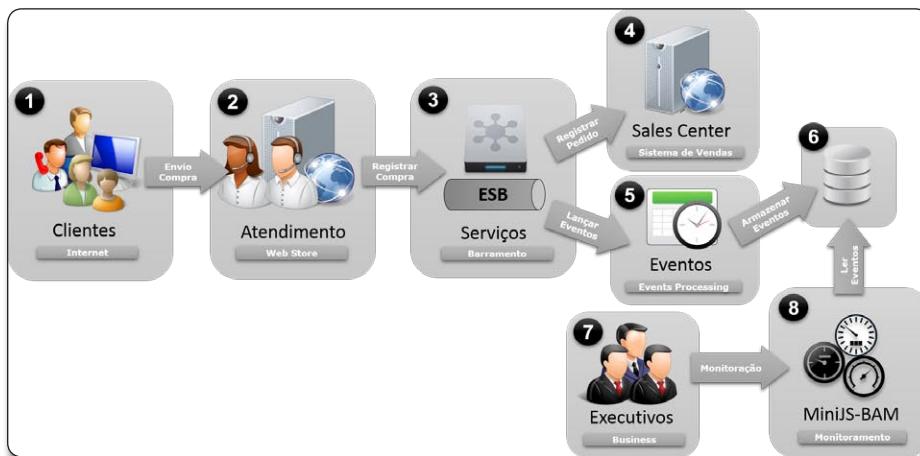
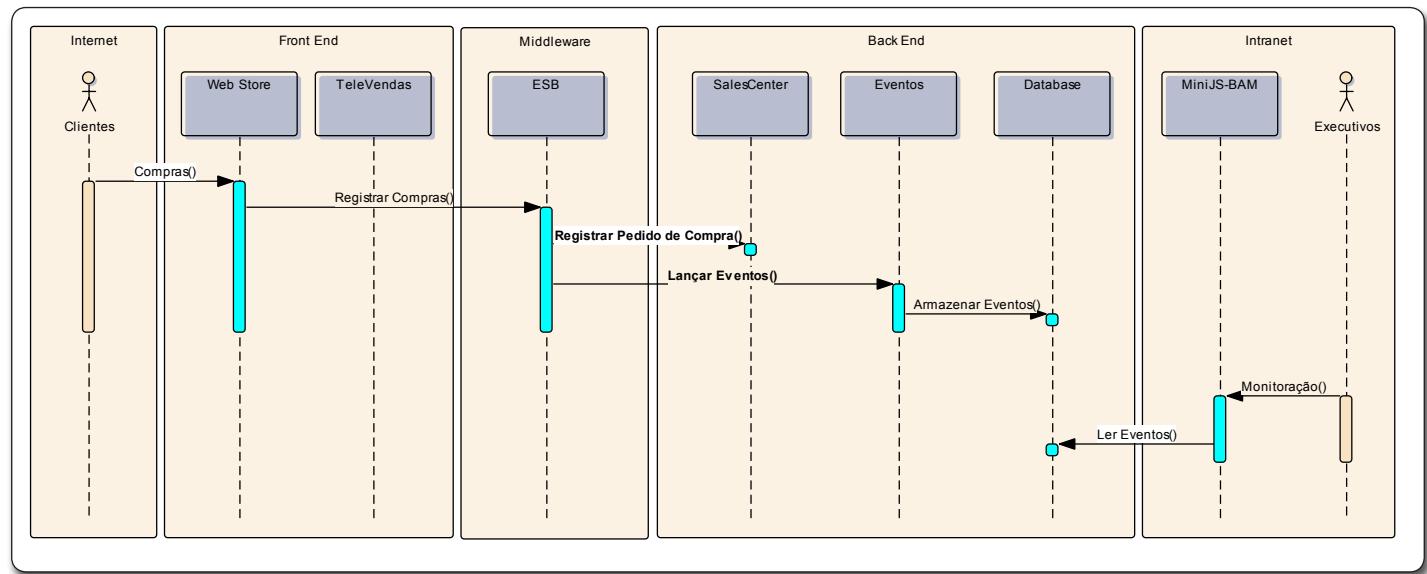


Figura 3. Visão holística da arquitetura lógica da solução de vendas on-line



**Figura 4.** Visão dinâmica da solução de vendas on-line

<b>Mule ESB (Anypoint Studio)</b>	Possui a função de prover o Barramento de Serviço, conhecido pelo acrônimo em inglês ESB (Enterprise Service Bus). Este componente é responsável por estabelecer o barramento dos fluxos de mensagens de integrações entre consumidores e provedores de serviços. A versão utilizada é a Community Edition 4.1.1, representada nesta solução pelo IDE Anypoint Studio, que possui uma versão reduzida do Mule ESB embutido para a execução dos fluxos.
<b>Apache TomEE+</b>	Servidor de aplicação Java certificado para o Web Profile Java EE 6 da Apache, com alguns adicionais, por exemplo, em vez de termos disponível o EJB Lite, como diz a especificação Web Profile Java EE, temos uma implementação completa da API EJB. Utilizaremos este servidor de aplicação para fazer a implantação das nossas soluções Java, o SalesCenter e o MiniJS-BAM. O Apache TomEE+ é praticamente o OpenEJB com "esteroides". Ele nasceu e cresceu deste projeto. A versão utilizada na solução é a 1.7.1.
<b>MySQL</b>	Banco de dados utilizado como repositório das informações capturadas durante o processamento dos eventos pelo Esper. Assim, mais tarde, ferramentas e/ou aplicações interessadas em mostrar estas informações em uma perspectiva analítica, podem fazê-lo. Nesta base de dados armazenamos o resultado final do processamento dos eventos, apenas para que nossa solução MiniJS-BAM possa usufruir e montar seu cockpit. Veremos mais informações sobre isso no nosso exemplo.
<b>Esper</b>	Componente de software para trabalhar com o processamento do fluxo de eventos, criado pela empresa EsperTech. Iremos analisar em mais detalhes esta ferramenta a seguir.

**Tabela 1.** Tecnologias utilizadas na implementação da solução de vendas on-line

A apresentação provida pelo sistema de monitoramento ocorre na forma de gráficos que, juntos, caracterizam uma espécie de *cockpit*. A partir dele, os executivos da empresa podem ter a visão do que acontece nos negócios de sua corporação – neste caso em tempo real –, o que pode trazer mais segurança e precisão para a tomada de decisões, pois imagine como seria difícil pilotar um Boeing 777-200 sem nenhum painel de informações: sem saber a altitude, velocidade vertical, velocidade em relação ao solo, velocidade do ar, inclinação, posição dos flaps e a proa, e ainda assim ter que conduzir um “negócio” de U\$ 269.500.000,00.

Na **Figura 4** podemos verificar em uma visão mais convencional, utilizando UML, a dinâmica de funcionamento desta solução. Observe que o fluxo das mensagens inicia no momento em que o cliente realiza a compra. Em um dado momento após a realização da compra, mais tarde ou mesmo simultaneamente, os interessados em analisar as informações coletadas pelos eventos acessam via Intranet o MiniJS-BAM.

Para implantar o nosso exemplo de arquitetura orientada a eventos, na **Tabela 1** temos a lista das tecnologias seleciona-

das e o seu propósito básico perante a implementação desta solução.

A representação da arquitetura desta solução, agora traduzindo do conceito para a tecnologia implementada, pode ser observada na **Figura 5**. Nesta tradução, temos cada um dos elementos que utilizamos na composição do cenário.

A visão da implantação final de nossa solução pode ser vista no Diagrama de Deployment na **Figura 6**. Este digrama é muito útil para conciliar de forma organizada os dois mundos: software e infraestrutura. Os nossos componentes de software (WAR e EAR, neste caso), com as necessidades de infraestrutura (Apache TomEE, Mule ESB e o MySQL). Infelizmente é pouco usado na prática.

Um detalhe importante que podemos verificar nesse modelo de implantação e que passa despercebido em nossos modelos dinâmicos apresentados é a presença do JAR com o nome de *Domain Model – Canonical*. Neste componente estão presentes todos os objetos que representam as entidades, mensagens e eventos comuns do modelo de domínio da empresa em questão. Portanto, desta forma, mantemos o modelo canônico como um componente

# Elaborando projetos com a Arquitetura Orientada a Eventos

o qual todos os demais podem (na verdade, deveriam) utilizar em suas soluções como dependência externa.

Contido neste componente canônico, dentre outros, há dois elementos comuns que ultrapassam a fronteira das aplicações da empresa, que são os objetos **Order** e **Customer**. Estes, respectivamente, representam as entidades Pedido de Compra e Cliente, pertencentes ao domínio de negócio da corporação. Na **Listagem 1** são demonstrados estes objetos. Perceba que eles possuem anotações JPA e JAXB, pois assim levam consigo as definições de como

devem ser utilizados pelos mecanismos de persistência e serialização/deserialização REST/XML. Como estas anotações não são intrusivas, no final as classes continuam sendo simples objetos POJO que podem ser utilizados por qualquer classe Java, sem a necessidade da presença de um container, como um EJB para JPA.

Para controle dos projetos da organização, utilizamos a ferramenta Maven. Assim, este componente do modelo canônico estará presente no repositório Maven para ser utilizado pelos demais projetos de forma organizada e centralizada.

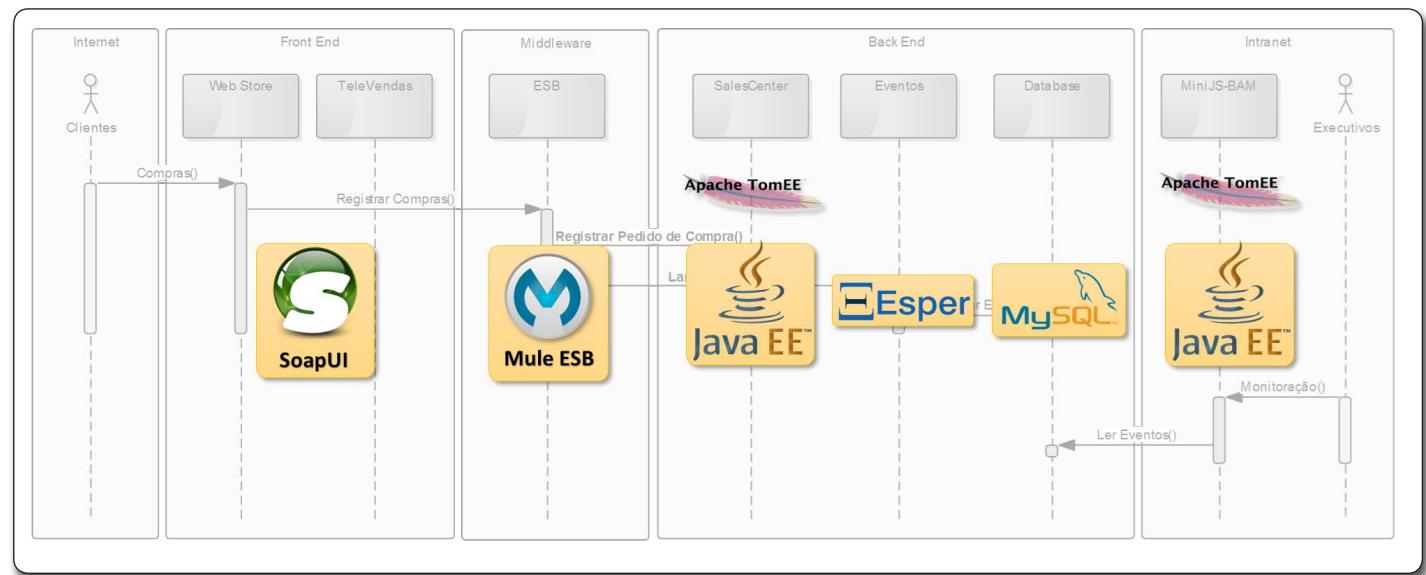


Figura 5. Elementos de tecnologia utilizados na concepção da solução

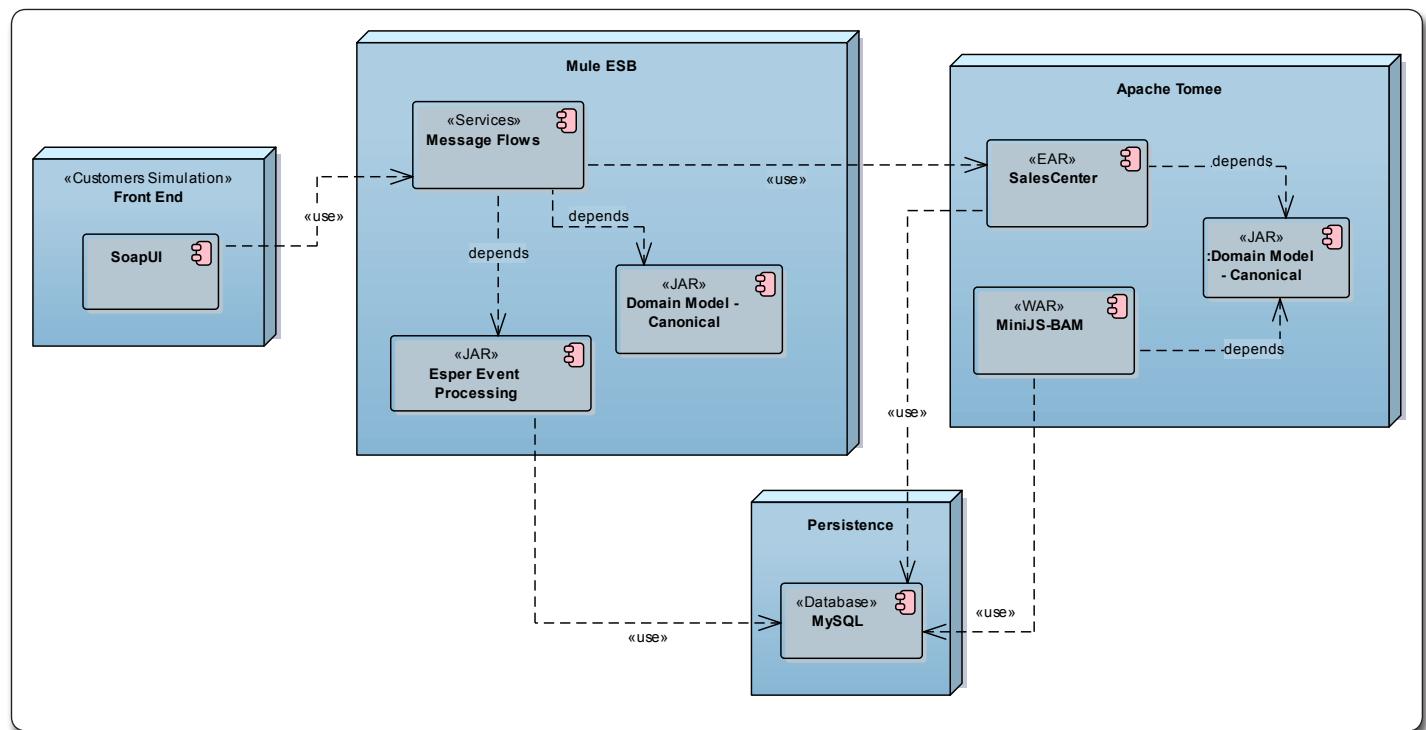


Figura 6. Diagrama de deployment da solução

**Listagem 1.** Exemplo de dois objetos canônicos contidos no componente Domain Model – Canonical.

```
@Entity
@Table(name="Customer")
@XmlRootElement
public class Customer {

    @Id @GeneratedValue(strategy = GenerationType.AUTO)
    private Integer id;
    private String name;
    private String city;
    private String country;

    public Customer(Integer id, String name, String city, String country) {
        super();
        this.id = id;
        this.name = name;
        this.city = city;
        this.country = country;
    }

    // Setters and Getters
}

@Entity
@Table(name="Orders")
@XmlRootElement
public class Order implements Serializable {

    @Id @GeneratedValue(strategy = GenerationType.AUTO)
    private Integer number;

    @ManyToOne(cascade={CascadeType.PERSIST})
    @JoinColumn(name="id_customer")
    private Customer customer;

    @OneToMany(cascade={CascadeType.PERSIST}, fetch = FetchType.EAGER)
    @JoinColumn(name="id_order")
    private List<OrderItem> itens;

    @Temporal(TemporalType.TIMESTAMP)
    @Column(name ="Date")
    private Date dateTime;

    public Order(Customer customer, List<OrderItem> itens) {
        super();
        this.customer = customer;
        this.itens = itens;
    }

    // Setters and Getters
}
```

## Esper – O componente de processamento dos eventos

Um componente importante que deve ser destacado em nossa solução é a ferramenta Esper, pois é esta que nos fornece os recursos para trabalharmos com o processamento e interpretação dos eventos lançados durante as transações. O Esper é uma aplicação criada em 2006 pela empresa EsperTech que possui diferentes versões disponíveis. Dentre essas versões, algumas são mais robustas e requerem o pagamento de licenças. No entanto, garantem recursos mais avançados, como aspectos de alta disponibilidade. Felizmente, a empresa disponibiliza também uma versão livre, a qual adotaremos em nossa solução. Esta ferramenta é direcio-

nada especialmente para trabalhar com soluções que envolvam a Arquitetura Orientada a Eventos (EDA).

O Esper permite a análise de eventos em séries, ou executar a análise fazendo a correlação entre um grupo de eventos e, a partir disto, realizar alguma ação no momento da captura destes eventos lançados. Neste contexto, o evento é um grupo de informações que são originárias de uma ou mais transações. E estas informações, contidas nos eventos, devem ser relevantes para o negócio da empresa, algo que tenha semântica dentro do seu cenário de negócio. Por exemplo, na lista a seguir temos alguns eventos que podem ser de interesse da empresa:

- Quantidade de pedidos por minuto maior que R\$ 2.000,00;
- Quantidade de pedidos nas principais cidades da região Sudeste;
- Valor total de pedidos nos últimos 60 minutos;
- Valor total do maior pedido realizado nos últimos 60 minutos.

A identificação do evento está relacionada ao cenário transacional onde se originam os dados, sendo deste cenário extraídas as visões que a área de negócio possui interesse em analisar. No entanto, é possível não só capturar eventos que sejam relevantes ao negócio, como também lançar eventos com visões de perspectiva mais técnica como, por exemplo: o tempo de execução de uma operação. Porém, para conseguir trabalhar com estes eventos de natureza técnica, as informações devem ser capturadas durante a execução das transações de negócio, pois informações de negócio são naturalmente tratadas, mas os dados técnicos nem sempre são capturados durante as transações, tornando mais tarde praticamente impossível o lançamento de eventos de natureza técnica.

### EPL – Event Processing Language

O Esper possui uma linguagem específica de domínio (DSL) para o tratamento de eventos. À primeira vista, bem semelhante ao SQL para bancos de dados. Ela é chamada de EPL, acrônimo do inglês *Event Processing Language*. Através desta linguagem filtramos o fluxo de dados transacional detectando os eventos alvos de interesse, simultaneamente à sua ocorrência. No site do Esper temos disponível uma referência completa da linguagem EPL, inclusive uma área onde é possível realizar testes de declarações EPL de forma on-line. Os endereços estão disponíveis na seção [Links](#).

Sendo baseada em SQL, vários dos comandos da EPL são muito semelhantes ou exatamente iguais ao da linguagem padrão de bancos de dados, tais como: SELECT, FROM, WHERE, GROUP BY e ORDER BY.

Alguns exemplos da sintaxe e funcionamento desta linguagem estão ilustrados na **Listagem 2**.

Para o Esper, os eventos são representados como classes POJO, que detêm todas as informações que estes devem encapsular. No exemplo da **Listagem 1** temos os seguintes eventos: **StockTickEvent** e **OrderEvent**, que nada mais seriam do que as classes demonstradas na **Listagem 3**. Desta forma, podemos fazer uso de características da orientação a objetos, como herança e polimorfismo, para conceber uma cadeia de eventos que represente de forma eficiente o domínio de negócio da empresa.

# Elaborando projetos com a Arquitetura Orientada a Eventos

Vale ressaltar novamente a grande utilidade que o conceito de Canônico pode trazer. Para criar um padrão de comunicação entre os sistemas da empresa, entidades e mensagens podem ser padronizadas dentro do barramento de serviços, onde os sistemas se integram “falando” a mesma língua. E da mesma maneira que podemos padronizar entidades comuns (por exemplo: Cliente),

**Listagem 2.** Exemplos de declarações da linguagem específica de domínio EPL da ferramenta Esper.

```
// Busca o símbolo, o preço e o total do volume dos preços de movimentações da
// ação da IBM dentro
// dos últimos 60 segundos
select symbol, price, sum(volume) from StockTickEvent(symbol='IBM').win:time(60 sec)

// Busca a média e o total do preço nas movimentações da ação da IBM dentro
// dos últimos 10 elementos
select avg(price) as avgPrice, sum(price) as sumPrice from
StockTickEvent(symbol='IBM').win:length(10) where symbol='IBM'

// Busca o total das movimentações de ações dentro dos últimos 30 segundos,
// quando o volume for maior que 100
select count(*) from StockTickEvent.win:time(30 sec) where volume > 100 group
by symbol

// Busca a cada 60 segundos o total dos preços dos Pedidos
// nos últimos 30 minutos
select sum(price) from OrderEvent.win:time(30 min) output snapshot every
60 seconds

// Busca a cada 1.5 minutos as informações das movimentações de ações
// dentro das últimas cinco movimentações ocorridas
select * from StockTickEvent.win:length(5) output every 1.5 minutes
```

**Listagem 3.** Objetos POJO que representam para o Esper as entidades de eventos.

```
public class StockTickEvent {
    private String symbol;
    private float price;
    private int volume;
    // setters e getters
}

public class OrderEvent {
    private float price;
    // setters e getters
}
```

podemos padronizar eventos (por exemplo: Quantidade Total de Ordem de Compra no dia), pois tais eventos também são de uso corporativo, ou seja, não pertencem apenas a uma aplicação, e sim à corporação.

Em nossa solução é importante observar que o Esper atua como um filtro, coletando apenas informações que são alvos de interesse. Apenas estes dados, no formato de eventos, são direcionados para o armazenamento na base de dados MySQL. Portanto, hipoteticamente, se 100 milhões de transações passarem pelo Barramento de Serviços, apenas uma fração destes dados, já processados no formato de eventos, serão selecionados e lançados pelo Esper como objeto de análise. A quantidade de eventos, é claro, dependerá do desenho concebido para a solução em questão, o que definirá também a existência da característica de integração por eventos da arquitetura orientada a eventos. A **Figura 7** representa uma ilustração do fluxo de dados deste cenário.

## Implementação da solução de Arquitetura Orientada a Eventos

A partir de agora iremos abordar a implementação, características e principal propósito de cada uma das camadas e componentes que fazem parte da nossa solução exemplo de Arquitetura Orientada a Eventos. Dentro de cada tópico, todo o código fonte relevante à principal função do componente será apresentado e explicado.

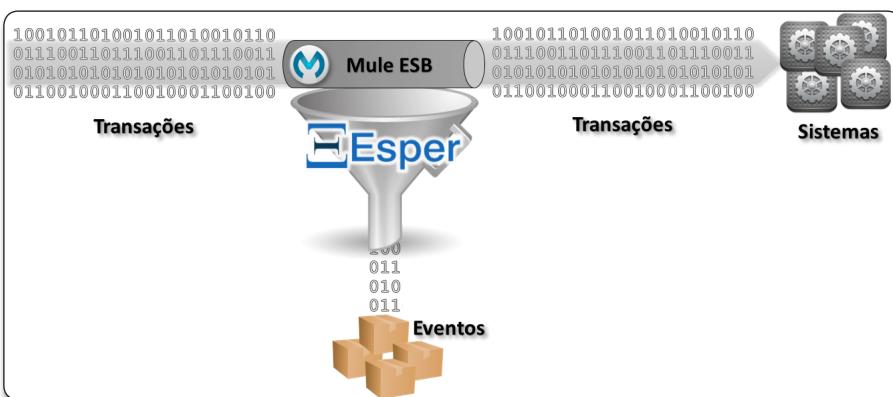
### Front End – SoapUI

Para representar o cliente navegando no site e realizando as compras, utilizamos a ferramenta SoapUI, que simula a execução destas chamadas. Como adendo, podemos aproveitar alguns recursos como, por exemplo, simular uma quantidade ininterrupta de chamadas ao nosso sistema de vendas. Esta ferramenta irá implementar a ação de enviar pedidos para o Barramento de Serviços Corporativos (ESB) da empresa.

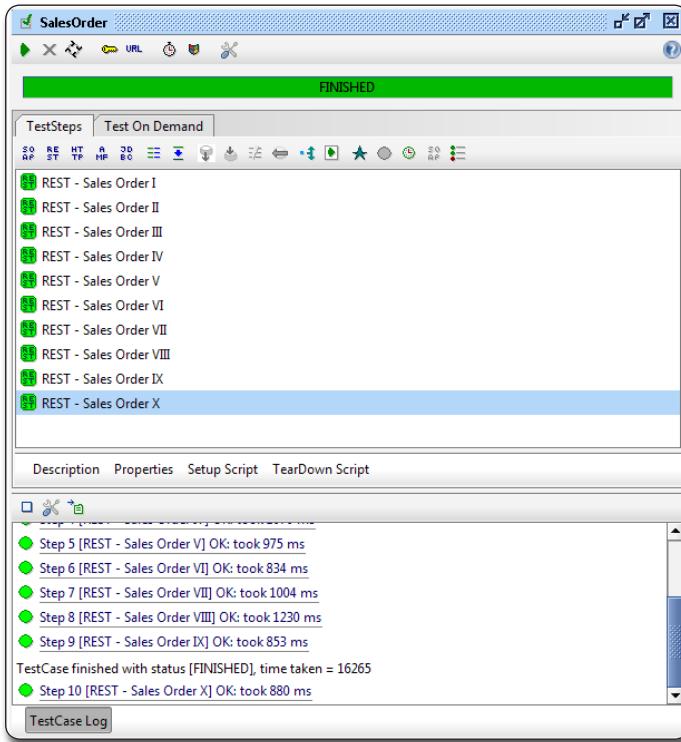
Na ferramenta SoapUI criamos um caso de teste onde são organizados os requests que simulam as chamadas de clientes. Estes requests se apresentam no formato JSON e possuem todas as informações de um registro de compra realizado por um cliente. Utilizamos o formato JSON porque o Mule ESB (Barramento de Serviços) fornece um Serviço de Registro de Pedido via protocolo REST. Na **Listagem 4** podemos conferir o conteúdo desta mensagem enviada para o ESB.

Para garantir que as execuções dos requests do caso de teste ocorram com sucesso, criamos *assertions* no SoapUI para estes requests, conferindo informações básicas que deveriam estar presentes na resposta. Nestes *assertions* validamos ao menos:

- Que o status do código de retorno HTTP seja 200 – OK;
- Que o conteúdo da resposta contenha o texto “orderNumber”;
- Utilizamos uma declaração XPath procurando pelo **orderNumber** na resposta e conferindo que seu valor seja maior que zero.



**Figura 7.** Cenário do fluxo de dados e a filtragem dos eventos pelo Esper



**Figura 8.** Caso de teste SoapUI com dez requests de criação de pedidos de compra

**Listagem 4.** Mensagem REST no SoapUI que simula um request de registro de uma compra.

```
{
  "order": [
    {
      "customer": {
        "id": 1,
        "name": "Ulalter",
        "city": "Sao Paulo",
        "country": "Brasil"
      },
      "itens": [
        {
          "product": {
            "id": 1,
            "description": "66 Audio BTS",
            "price": 12.30,
            "unit": "UNITARIO"
          },
          "quantity": 5
        },
        {
          "product": {
            "id": 2,
            "description": "MP3 PLAYER",
            "price": 250,
            "unit": "UNITARIO"
          },
          "quantity": 2
        },
        {
          "product": {
            "id": 3,
            "description": "BATTERIES",
            "price": 5,
            "unit": "UNITARIO"
          },
          "quantity": 4
        }
      ]
    }
  ]
}
```

Dentro deste caso de teste temos dez requests que serão executados. Eles possuem exatamente a mesma estrutura. Apenas alteramos os valores dos atributos para simular diferentes pedidos sendo realizados. Podemos observar a interface da ferramenta SoapUI executando estes requests de registro de compra na **Figura 8**.

#### Middleware – Mule ESB

No barramento temos exposto o serviço que permite o registro de um pedido de compras no sistema de vendas SalesCenter. Para isto, criamos um Fluxo de Mensagem conforme observamos na **Figura 9**. Nesta imagem marcamos cada um dos elementos do fluxo de mensagens com um número, para que passo a passo possamos explicar o objetivo de cada componente.

No componente de **número um**, rotulado de “HTTP/REST”, criamos um canal de entrada HTTP que recebe os pedidos em protocolo REST. Para este componente HTTP, atribuímos o endereço `http://localhost:9081/mule` para receber os pedidos.

No componente de **número dois**, rotulado de “Receive Sales Order”, está a classe Java que expõe efetivamente o endpoint REST, realizando o suporte de implementação para o componente HTTP anterior. Esta classe Java utiliza a API JAX-RS para expor um serviço RESTful, onde no método `receiveSalesOrder()` recebemos o request com o Pedido de Compra encapsulado. Este código é apresentado na **Listagem 5**.

**Listagem 5.** Classe Java que implementa o serviço RESTful (via API JAX-RS) para recebimento dos registros dos pedidos de compras.

```
@Path("/rest")
public class ReceiveSalesOrder {

  @POST
  @Path("/sales/order/")
  @Consumes({MediaType.APPLICATION_JSON})
  @Produces({MediaType.APPLICATION_JSON})
  public MessageRequest receiveSalesOrder(MessageRequest request) {
    br.com.ujr.isus.canonical.Order order = request.getOrder();

    return request;
  }

  @GET
  @Path("/hello/{nome}")
  @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
  public String hello(@PathParam("nome") String name, @QueryParam("param")
  String param) {
    return "Hello " + name + " " + param;
  }
}
```

No código podemos observar que fazemos uso de recursos fornecidos pela API JAX-RS utilizando a anotação `@Path("/rest")` na classe e a anotação `@Path("/sales/order/")` no método. Desta forma, definimos que a URL final de comunicação com o barramento será: `http://localhost:9081/mule/rest/sales/order`.

# Elaborando projetos com a Arquitetura Orientada a Eventos

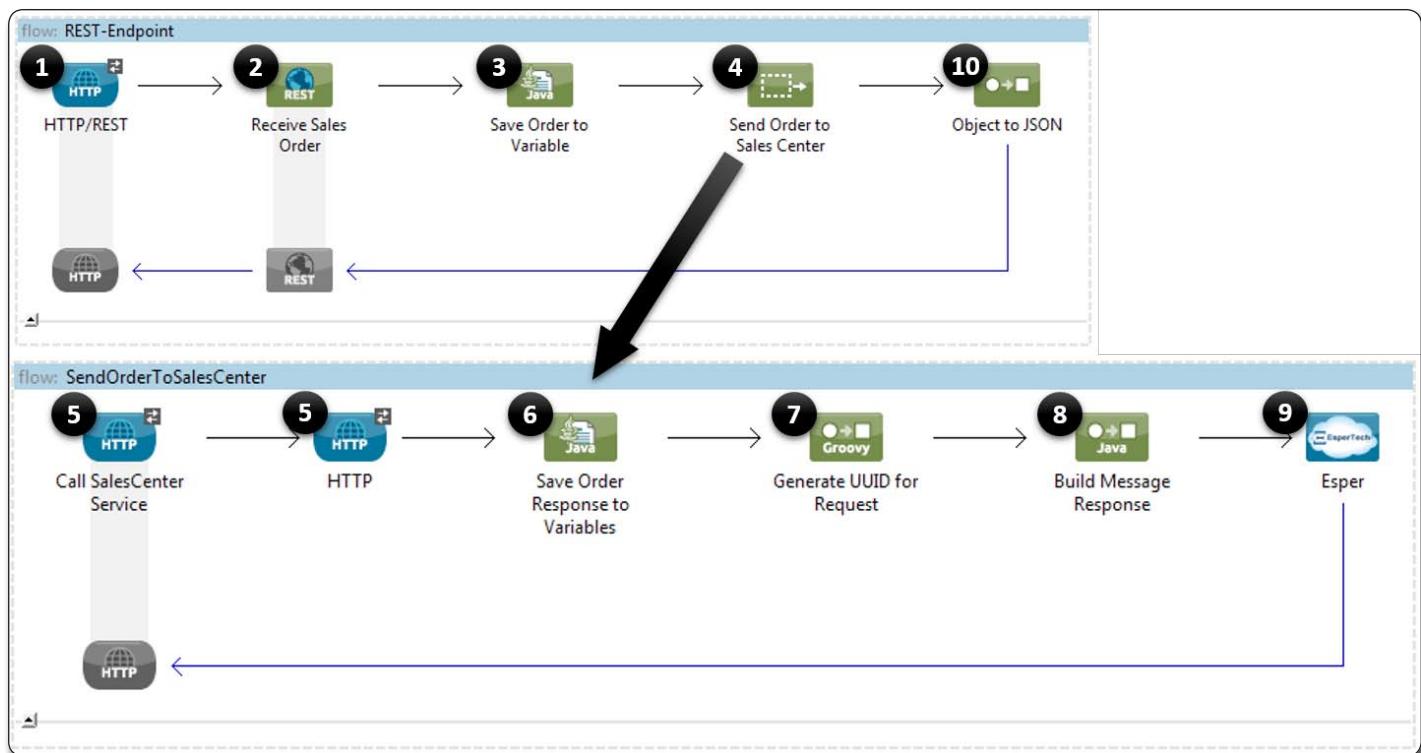


Figura 9. Fluxo de mensagem do serviço de registro de pedido de compras exposto no barramento

Esta URL é o endereço utilizado pelo caso de teste do SoapUI explicado no tópico anterior.

Na sequência do fluxo temos o componente de **número três**, de nome “Save Order to Variable”. Neste elemento temos uma classe Java a qual o Mule ESB executa durante o fluxo da mensagem, passando para ela todo o contexto de informações. Isto permite a realização de algum tratamento dos dados sendo trafegados utilizando código Java. A função deste componente, neste caso, é apenas armazenar em memória o objeto **Order** (Pedido de Compra) original que foi enviado no request executado pelo cliente. Para isso, ele converte a **String JSON** recebida para o objeto do tipo **Order** e, em seguida, o armazena como uma variável do Fluxo de Mensagem. O código da classe **SaveOrderToVariable** é apresentado na **Listagem 6**.

O componente de **número quatro** é uma referência a um subflow, um outro fluxo de mensagem que deve ser chamado neste momento. O objetivo deste subflow é realizar a chamada ao provedor do serviço, o sistema que registra os pedidos de compra.

Entrando no subflow, de nome *SendOrderToSalesCenter*, temos o componente de **número cinco**, que realiza a chamada HTTP/ REST ao sistema SalesCenter. Ele executa o request ao endereço <http://localhost:8083/SalesCenter-Web/rest/order/register/>, no qual está exposto o serviço de Registro de Compras no protocolo REST. Este web service RESTful encontra-se em um Servidor de Aplicação Java EE Apache TomEE+, o qual iremos verificar durante a análise da aplicação SalesCenter nos tópicos a seguir. No momento, desempenhando o papel de barramento de serviços, precisamos entender apenas que este é o provedor do serviço que registra a ordem de compra. Este serviço recebe o Pedido de Compra (Order) no protocolo REST e nos retorna um número gerado caso a transação tenha sido executada com sucesso.

Continuando no subflow *SendOrderToSalesCenter*, temos o componente de **número seis**, nomeado “Save Order Response to Variable”, outra classe Java que tem a oportunidade de interagir com a mensagem. Neste caso, não alteramos em nada a mensagem. Apenas a interceptamos para armazenar a resposta do SalesCenter em variáveis do fluxo de mensagem do Mule ESB. Na **Listagem 7** podemos observar o código desta classe.

As informações retornadas após a execução da transação do pedido de compras estão encapsuladas no objeto Java de nome **ResponseSaveOrder**, que pertence ao nosso modelo canônico, presente no JAR mencionado anteriormente.

**Listagem 6.** Classe Java executada pelo componente “Save Order to Variable” no fluxo de mensagem do ESB.

```
public class SaveOrderToVariable implements Callable {  
    @Override  
    public Object onCall(MuleEventContext eventContext) throws Exception {  
        String restOrder = eventContext.getMessage().getPayloadAsString();  
        ObjectMapper om = new ObjectMapper();  
        om.configure(Feature.UNWRAP_ROOT_VALUE, true);  
        Order order = om.readValue(restOrder, Order.class);  
        eventContext.getMessage().setInvocationProperty("Order", order);  
        return eventContext.getMessage().getPayload();  
    }  
}
```

Neste objeto temos a data e o número da ordem gerado, que armazenamos separadamente em variáveis do nosso fluxo de mensagem.

**Listagem 7.** Classe Java executada pelo componente “Save Order Response to Variable” no fluxo de mensagem do ESB.

```
public class SaveOrderResponseToVariables implements Callable{

    @Override
    public Object onCall(MuleEventContext eventContext) throws Exception {

        String jsonString = eventContext.getMessageAsString();
        ObjectMapper mapper = new ObjectMapper();
        mapper.configure(DeserializationFeature.UNWRAP_ROOT_VALUE, true);
        ResponseSaveOrder responseSaveOrder = mapper.readValue
        (jsonString, ResponseSaveOrder.class);

        Order order = (Order)eventContext.getMessage().getInvocationProperty("order");
        order.setDateTime(responseSaveOrder.getDate());

        eventContext.getMessage().setInvocationProperty("responseSaveOrder",
        responseSaveOrder);
        eventContext.getMessage().setInvocationProperty("orderNumber",
        String.valueOf(responseSaveOrder.getNumber()));
        eventContext.getMessage().setInvocationProperty
        ("date", responseSaveOrder.getDate().toString());

        return eventContext.getMessage().getPayload();
    }

}
```

O componente de **número sete** trata-se de um script Groovy, que gera um código aleatório, “praticamente” único, utilizando o padrão de identificação UUID. Retornamos este número ao consumidor do serviço de Registro de Pedido com uma espécie de *Ticket* que ele pode utilizar para referenciar mais tarde um determinado request feito ao serviço fornecido pelo Mule ESB, caso seja necessário. Rotulamos este componente de “Generate UUID for Request”.

Seguindo para o componente de **número oito**, rotulado de “Build Message Response”, encontramos um componente transformador de informações de mensagens, implementado com uma classe Java. Neste ponto, todas as informações trafegando pelo fluxo de mensagem serão passadas para uma classe Java, que irá reunir todas as informações consideradas relevantes que coletamos durante todo o fluxo para compor a mensagem de resposta para o nosso consumidor. Na **Listagem 8** temos o código da classe **CompoundMessageResponseObject**. Observe que no final retornamos o objeto **MessageResponse**, criado especialmente para ser a resposta deste fluxo de mensagens, fazendo parte do contrato do serviço exposto.

Neste ponto da análise do fluxo chegamos ao último componente do nosso subflow *SendOrderToSalesCenter*. Este realiza uma tarefa importante para a nossa arquitetura orientada a eventos. No entanto, do ponto de vista transacional, já temos tudo finalizado: o pedido foi gravado com sucesso no sistema SalesCenter e temos todas as informações para retornar ao consumidor.

**Listagem 8.** Classe Java utilizada no componente “Build Message Response” para compor a resposta do serviço de Registro de Compras.

```
public class CompoundMessageResponseObject extends AbstractMessage-
Transformer {

    @Override
    public Object transformMessage(MuleMessage message, String outputEncoding)
    throws TransformerException {

        String uuid = "";
        String orderNumber = "";
        String date = "";
        try {
            uuid = message.getPayloadAsString();
            orderNumber = message.getInvocationProperty("orderNumber");
            date = message.getInvocationProperty("date");
        } catch (Exception e) {
            e.printStackTrace();
        }
        return new MessageResponse(new Integer(orderNumber),uuid,date);
    }
}
```

Antes de retornar a resposta, no entanto, iremos realizar um passo que não impacta mais no término transacional deste fluxo de mensagens, que é lançar o evento ocorrido para a nossa instância do Esper. Isto é realizado através de um conector específico feito para a comunicação com o Esper. Estamos nos referindo ao componente de **número nove**, denominado Esper.

Este componente de conexão com o Esper não está presente na versão livre do Anypoint (Mule ESB) disponível para a comunidade. Sendo assim, é necessária a instalação deste conector executando os seguintes passos:

- No Anypoint Studio, navegue pelo menu seguindo o caminho: *Help > Install New Software*;
- Na janela seguinte, escolha a opção *Anypoint Connectors Update Site* no campo *Work with*;
- Procure pelo módulo *Mule Esper Module Mule Studio Extension* nas opções de Community.
- Na **Figura 10** podemos verificar o conector que deve ser instalado.

Após a instalação, precisamos criar uma configuração para o conector no Fluxo de Mensagem, a qual será utilizada por todas as instâncias do componente Esper, seja para lançar eventos ou para interceptá-los. Para isto, criamos um arquivo XML de nome *esper-config.xml*, apresentado na **Listagem 9**, e adicionamos este arquivo ao fluxo de mensagem como um Global Element.

Nas primeiras linhas desta configuração informamos ao Esper quais são os tipos de eventos que poderão existir, e como mencionado anteriormente, são apenas classes POJO Java. Em nosso exemplo, informamos que as seguintes classes são tipos de eventos que poderão ser lançados ou interceptados:

- **OrderPedido**;
- **TotalOrderByCity**;
- **TotalOrderByDate**;
- **TotalOrderByTimeFrame**.

# Elaborando projetos com a Arquitetura Orientada a Eventos

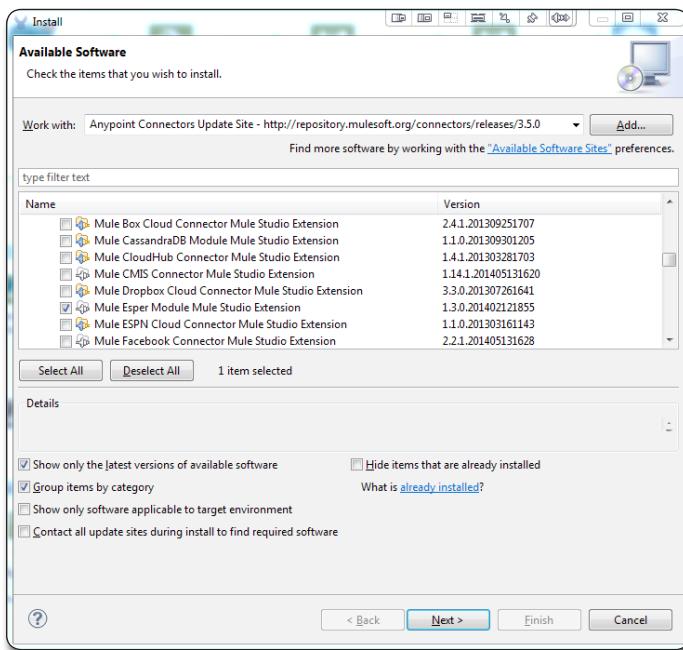


Figura 10. Instalação no AnyPoint Studio do conector para comunicação com o Esper

Em seguida, configuraremos um dos adaptadores de entrada e saída do Esper, que em nosso exemplo iremos utilizar apenas como adaptador de saída, sendo responsável pela conexão com a base de dados MySQL para armazenar as informações dos eventos capturados. Para tanto, configuraremos o adaptador **com.espertech.esperio.db.EsperIODBAdapterPlugin**. Na configuração deste plugin, preparamos uma conexão JDBC na seção **jdbc-connection** para a comunicação com o MySQL.

No arquivo de configuração do Esper, *esper-config.xml*, as seções que possuem as tags **<upsert>** são onde configuramos as execuções de Insert ou Update na base de dados dos eventos capturados pelos conectores Esper. Para cada um destes comandos upserts, temos que, no mínimo, prover as seguintes informações:

- **connection**: Conexão JDBC com o MySQL;
- **stream**: Tipo do evento capturado a ser armazenado;
- **name**: Nome para o Upset;
- **table-name**: Nome da tabela na base de dados MySQL que irá armazenar os dados do evento capturado em questão;
- **executor-name**: Configuração do executor que definirá a quantidade de threads utilizada para este Upset;
- **retry**: Quantidade de tentativas a serem feitas em caso de falha;

Listagem 9. Configuração da instância do Esper para os conectores do Mule ESB lançar ou interceptar eventos.

```
<esper-configuration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://www.espertech.com/schema/espertech"
  xsi:schemaLocation="
    http://www.espertech.com/schema/espertech
    http://www.espertech.com/schema/espertech/
    esper-configuration-2.0.xsd">
  <event-type name="OrderPedido" class="br.com.ujr.isus.canonical.Order"/>
  <event-type name="TotalOrderByCity" class="br.com.ujr.isus.canonical.events.TotalOrderByCity"/>
  <event-type name="TotalOrderByDate" class="br.com.ujr.isus.canonical.events.TotalOrderByDate"/>
  <event-type name="TotalOrderByTimeFrame" class="br.com.ujr.isus.canonical.events.TotalOrderByTimeFrame"/>
  <plugin-loader name="EsperIODBAdapter" class-name="com.espertech.esperio.db.EsperIODBAdapterPlugin">
    <config-xml>
      <esperio-db-configuration>
        <jdbc-connection name="esperEvents">
          <drivermanager-connection class-name="com.mysql.jdbc.Driver"
            url="jdbc:mysql://localhost:3306/dbevents" user="esper" password="esper" />
          <connection-settings auto-commit="true" catalog="dbevents"/>
        </jdbc-connection>

        <upsert connection="esperEvents" stream="TotalOrderByCity"
          name="UpdateTotalOrderByCityStream"
          table-name="TableTotalOrderByCityStream"
          executor-name="queue1" retry="3">
          <keys>
            <column property="city" column="CITY" type="varchar"/>
          </keys>
          <values>
            <column property="total" column="QTDE" type="integer"/>
          </values>
        </upsert>
        <upsert connection="esperEvents" stream="TotalOrderByDate"
          name="UpdateTotalOrderByDateStream"
          table-name="TableTotalOrderByDateStream"
          executor-name="queue1" retry="3">
          <keys>
            <column property="date" column="DATE" type="integer"/>
          </keys>
          <values>
            <column property="total" column="QTDE" type="integer"/>
          </values>
        </upsert>
      </esperio-db-configuration>
    </config-xml>
  </plugin-loader>
</esper-configuration>
```

```
  <table-name="TableTotalOrderByDateStream"
    executor-name="queue2" retry="3">
    <keys>
      <column property="date" column="DATE" type="integer"/>
    </keys>
    <values>
      <column property="total" column="QTDE" type="integer"/>
    </values>
  </upsert>
  <upsert connection="esperEvents" stream="TotalOrderByTimeFrame"
    name="UpdateTotalOrderByTimeFrameStream" table-name="TableTotalOrderByTimeFrameStream"
    executor-name="queue3" retry="3">
    <keys>
      <column property="timeFrame" column="TIMEFRAME" type="varchar"/>
    </keys>
    <values>
      <column property="qtdeTotal" column="QTDE" type="integer"/>
    </values>
    <values>
      <column property="valueTotal" column="VALUE" type="decimal"/>
    </values>
  </upsert>
  <executors>
    <executor name="queue1" threads="3"/>
  </executors>
  <executors>
    <executor name="queue2" threads="3"/>
  </executors>
  <executors>
    <executor name="queue3" threads="3"/>
  </executors>
  </esperio-db-configuration>
</config-xml>
</plugin-loader>
</esper-configuration>
```

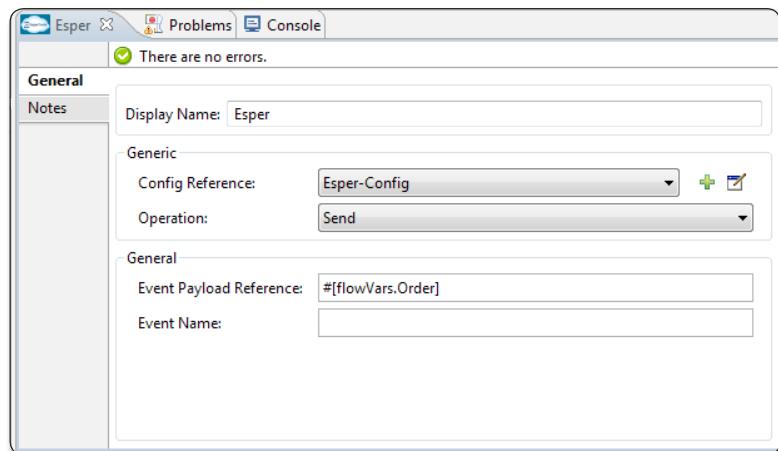
- keys:** Nesta seção devem ser informados todos os campos que formam a chave primária da tabela/evento. A informação destes campos será processada nas cláusulas WHERE pelo adaptador Esper, definindo por exemplo, se será uma instrução SQL de Update ou de Insert que deve ser executada;
- values:** Nesta seção informamos todo o restante das informações do evento, que estão também na tabela da base de dados.

Com o arquivo de configuração pronto, podemos preencher as propriedades do componente Esper. Na **Figura 11** observamos quais são os atributos necessários. Além da referência à configuração do Esper (*Config Reference*), escolhemos a operação (*Operation*) que deve ser realizada e qual será o payload enviado. Neste ponto do fluxo, estamos lançando um evento que se trata da ordem de compra realizada. Por isso, informamos a operação *Send*, e no Payload (*Event Payload Reference*) passamos o caminho onde está a informação (evento) a ser lançada. Em passos anteriores, armazenamos como uma variável do Fluxo de Mensagem a Ordem de Compra. Neste momento estamos enviando-a para o Esper utilizando a literal `#[flowVars.Order]`.

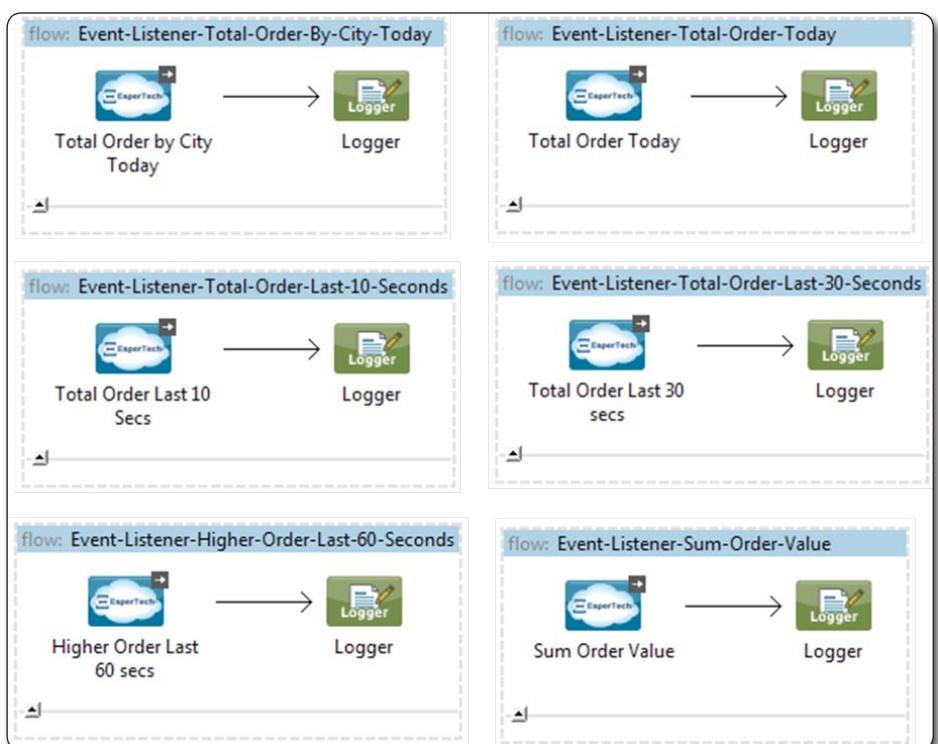
Neste passo do fluxo estamos apenas informando a ocorrência de um fato ao Esper (Ordem de Compra realizada). Veremos mais adiante como realizar a configuração dos filtros no Esper, os quais irão processar, identificar, criar e lançar determinados eventos de interesse da empresa.

No componente de **número dez**, para finalizar a transação de ordem de compra, retornamos ao fluxo principal, denominado “REST-Endpoint”. Aqui executamos o último passo antes de retornar a resposta para o consumidor. Neste componente, o objeto **MessageResponse** é transformado então em uma String JSON que será o retorno do request realizado no início pelo cliente ao serviço Registro de Compras.

Neste momento já temos realizada a transação de compra do cliente e a informação do evento Ordem de Compra para o Esper. Vamos verificar agora como requisiti-



**Figura 11.** Configuração do componente de comunicação Esper no fluxo de mensagem do Mule ESB



**Figura 12.** Fluxos de Mensagens contendo o componente Esper para processamento dos eventos

tamos ao Esper para interceptar, processar e lançar os eventos de nosso interesse.

Em nosso exemplo iremos trabalhar com seis eventos, que serão explicados em mais detalhes no tópico “Configuração de componentes Esper”. Para estes eventos serem lançados, criamos outros seis Fluxos de Mensagem muito simples no Mule ESB. Em cada um deles utilizamos o mesmo componente com atributos de configuração diferentes. Na **Figura 12** podemos verificar estes fluxos com apenas

um elemento do Esper configurado para processar eventos, seguido de um componente simples de Log.

As configurações das instâncias destes componentes Esper inseridos nos seis Fluxos de Mensagem são quase equivalentes. Praticamente apenas um atributo é distinto entre eles, e esta diferença está no comando EPL que realiza o processamento e lançamento do evento. A sintaxe e objetivo da EPL foram abordados no tópico “EPL – Event Processing Language”.

# Elaborando projetos com a Arquitetura Orientada a Eventos

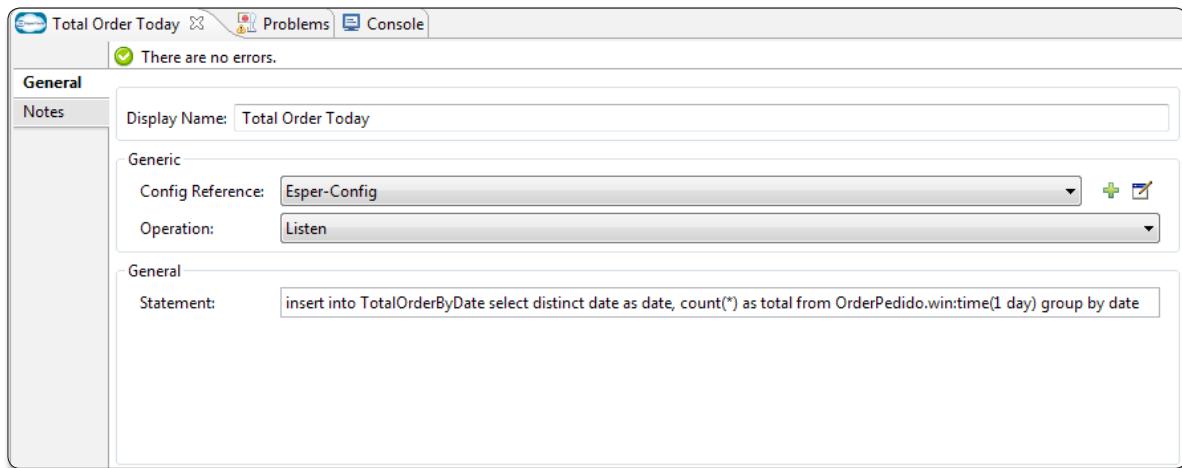


Figura 13. Componente Esper configurado para leitura dos eventos realizando o processamento das informações

Na Figura 13 podemos observar como é a configuração de um componente Esper que irá manipular os eventos que estamos interessados, realizando o processamento das informações enviadas.

Informamos no campo *Operation* que este componente irá realizar a inspeção dos eventos que são lançados ao Esper utilizando a opção *Listen*. A configuração mais relevante e que justifica a existência de seis diferentes componentes Esper é o atributo *Statement*, que recebe um comando EPL. Com esses comandos instruímos ao Esper o que deve ser processado para eventualmente lançar um determinado evento, se a condição no script EPL for encontrada.

## Configuração de componentes Esper

Para cada uma das configurações feitas nestes componentes Esper, vamos inspecionar e analisar as instruções EPL utilizadas:

- Componente: **Total Order By City Today**

- EPL:

- *insert into TotalOrderByCity select distinct customer.city as city, count(\*) as total from OrderPedido.win:time(1 day) group by customer.city.*

- Objetivo:

- Capturar o total de Pedidos por cidade dentro do período de um dia.

- Resultado:

- Conforme a configuração do plugin Esper de I/O de base de dados no arquivo *esper-config.xml*, as informações capturadas por esta EPL serão gravadas na tabela *TableTotalOrderByCityStream* do MySQL, no caso: cidade e total de pedidos, dentro da janela de um dia. Observe que estamos realizando a inserção de dados no evento/stream **TotalOrderByCity**, configurado no *esper-config.xml*. Nesta configuração, o Upsert que trata do evento **TotalOrderByCity** realizará a associação deste evento com a tabela da base de dados em que serão registradas as suas informações.

- Componente: **Total Order Last 10 Secs**

- EPL:

- *insert into TotalOrderByTimeFrame select '10secs' as timeFrame, sum(totalOrder) as valueTotal, count(\*) as qtdeTotal from OrderPedido.win:time(10 sec).*

- Objetivo:

- Capturar o total do valor e da quantidade de pedidos realizados dentro de uma janela de tempo dos últimos 10 segundos.

- Resultado:

- Conforme configuração realizada no Upsert que aponta para o evento/stream **TotalOderByTimeFrame**, as informações do valor total dos pedidos e quantidade de pedidos efetuados nos últimos 10 segundos serão gravadas na tabela *TableTotalOrderByTimeFrameStream*. Observe que colocamos a literal '10secs' como valor para o campo *timeFrame*. Assim rotulamos essa informação para gravação na tabela.

- Componente: **Total Order Last 30 Secs**

- EPL:

- *insert into TotalOrderByTimeFrame select '30secs' as timeFrame, sum(totalOrder) as valueTotal, count(\*) as qtdeTotal from OrderPedido.win:time(30 sec).*

- Objetivo:

- Capturar o total do valor e da quantidade de pedidos realizados dentro de uma janela de tempo dos últimos 30 segundos.

- Resultado:

- Neste componente escrevemos uma EPL que realiza a mesma tarefa do componente explicado anteriormente. Assim, o evento/stream, tabela e o resultado das informações capturadas são os mesmos. O que muda é que solicitamos uma janela de tempo diferente, de 30 segundos.

- Componente: **Total Order Today**

- EPL:

- *insert into TotalOrderByDate select distinct date as date, count(\*) as total from OrderPedido.win:time(1 day) group by date.*

- Objetivo:

- Capturar o total da quantidade de pedidos realizados diariamente.

- Resultado:

- Conforme a configuração realizada no Upsert que aponta para o evento/stream **TotalOrderByDate**, as informações da quantidade total de pedidos efetuados por dia serão gravadas na tabela *TableTotalOrderByDateStream*.

• Componente: **Higher Order Last 60 secs**

- EPL:

```
- insert into TotalOrderByTimeFrame select 'HigherOrder60secs' as timeFrame, max(totalOrder) as valueTotal, count(*) as qtdeTotal from OrderPedido.win:time(60 sec).
```

- Objetivo:

- Capturar o pedido de maior valor efetuado no último minuto.

- Resultado:

- Novamente utilizamos o mesmo evento/stream (**TotalOrderByTimeFrame**) para configurar um Upsert que irá gravar na tabela *TableTotalOrderByTimeFrameStream* as informações do evento. Nesta captura solicitamos o pedido de maior valor realizado nos últimos 60 segundos e o gravamos na tabela com o rótulo 'HigherOrder60secs' no campo *timeFrame*.

• Componente: **Sum Order Value**

- EPL:

```
- insert into TotalOrderByTimeFrame select 'SumOrderValue' as timeFrame, sum(totalOrder) as valueTotal, count(*) as qtdeTotal from OrderPedido.
```

- Objetivo:

- Capturar a soma de todos os valores e a quantidade de pedidos realizados.

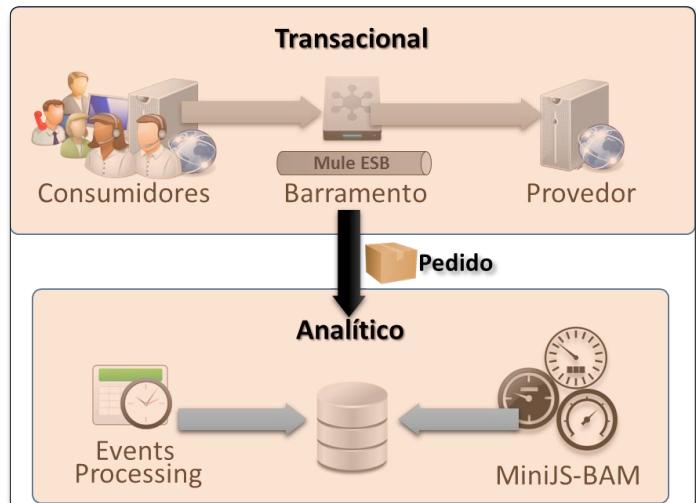
- Resultado:

- Como resultado, geramos o evento/stream **TotalOrderByTimeFrame**. No processamento da EPL, capturamos a soma do total de todos os pedidos e também a quantidade de pedidos realizados em todo o tempo decorrido desde o início da execução da aplicação. Para criarmos a identificação dos registros dos eventos gravados na tabela, utilizamos o rótulo 'SumOrderValue' como chave da unicidade destes registros de eventos. Assim, através do campo *timeFrame*, presente na tabela da base de dados, podemos identificar as informações capturadas com tal rótulo para o evento 'Soma do Valor e Quantidade Total de Ordens de Compra'.

A linguagem EPL fornece um acervo completo de recursos para a configuração do evento/stream a ser processado. Nestes exemplos vimos apenas uma pequena demonstração do que é capaz de prover esta linguagem de processamento de eventos do Esper.

Com isso, finalizamos a inspeção do Fluxo de Mensagem que expõe no barramento o Serviço de Registro de Pedidos para os consumidores. A parte transacional foi encerrada no momento em que o Pedido é gravado com sucesso pelo provedor do serviço, o sistema SalesCenter, que retorna a resposta ao Mule ESB.

Os componentes do Esper agem posteriormente, recebendo o evento Pedido, lançado na parte transacional, processando e filtrando os eventos/stream de interesse da empresa a serem capturados como, por exemplo, o total de pedidos nos últimos 30 segundos. Essa divisão entre o escopo transacional e o escopo analítico deste cenário pode ser observada na **Figura 14**.



**Figura 14.** Divisão entre os escopos transacional e analítico do cenário desta solução

### Back-End – SalesCenter

Por trás do barramento, portanto desacoplado de seus consumidores, está a aplicação atualmente responsável por gerenciar os pedidos de compras da empresa, conhecida como SalesCenter. Trata-se de uma aplicação Java EE distribuída implantada em um servidor de aplicação Apache TomEE+.

A aplicação é composta por um arquivo EAR contendo dois módulos:

- **SalesCenter-Services.JAR**: Módulo onde estão os serviços provados pela aplicação;
- **SalesCenter-Web.WAR**: Módulo web que provê as interfaces de Front-End Intranet necessárias aos usuários da aplicação. Em nosso cenário, este módulo não tem relevância, já que fazemos o uso apenas dos serviços expostos pela aplicação.

Para compreender melhor o papel desta aplicação, vamos analisar o código construído que representa os serviços providos pelo SalesCenter, os quais são acessados pelo Mule ESB em nosso estudo de caso. Comecemos pela interface criada para definir o contrato do que um serviço do SalesCenter deve prover. Esta interface está representada na **Listagem 10** e como podemos verificar, temos um código muito simples, com apenas um método que permite o registro de uma ordem de serviço.

Como próximo passo, iremos analisar a implementação desta interface, **ISalesCenterService**. A classe que a implementa tem como função executar a transação de registro do pedido, codificação esta que é realizada por um EJB Stateless, apresentado na **Listagem 11**. Este EJB recebe o request utilizando o protocolo REST

# Elaborando projetos com a Arquitetura Orientada a Eventos

(JAX-RS) com o objeto **Order** no formato JSON. Em seguida, com a referência a outro EJB, **SalesCenterFacade**, executa o registro do pedido e retorna a resposta ao cliente.

**Listagem 10.** Interface de definição do contrato de um serviço do SalesCenter.

```
public interface ISalesCenterService {  
    public ResponseSaveOrder placeOrder(Order order);  
}
```

**Listagem 11.** EJB Stateless que realiza o contrato de um serviço do SalesCenter.

```
@Stateless  
public class SalesCenterService implements ISalesCenterService {  
  
    private final static Logger LOGGER = LoggerFactory.getLogger(SalesCenterService.class);  
  
    @EJB  
    private SalesCenterFacade facade;  
  
    @POST  
    @Path("/order/register")  
    @Consumes(MediaType.APPLICATION_JSON)  
    @Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})  
    public ResponseSaveOrder placeOrder(Order order) {  
        LOGGER.info("Receiving order from customer: " + order.getCustomer().  
        .getName());  
        facade.registerSale(order);  
        Integer orderNumber = order.getNumber();  
        Date date = order.getDateTime();  
        LOGGER.info("Generated order #" + orderNumber.toString());  
        return new ResponseSaveOrder(orderNumber, date);  
    }  
  
    @GET  
    @Path("/hello/{name}")  
    public String ping(@PathParam("name") String name) {  
        return "Hello there," + name + "!";  
    }  
}
```

O próximo objeto que irá receber o request, seguindo o fluxo da aplicação SalesCenter, é o **SalesCenterFacade**, chamado pelo serviço **SalesCenterService**. O **SalesCenterFacade** é um EJB Stateless que centraliza todos os serviços disponíveis pelo modelo de domínio da aplicação. Dentre eles está o registro da ordem do pedido. Na **Listagem 12** temos o código fonte desta classe. Para colocar em exposição o serviço de Registro de Ordem de Compra, o disponibilizando para os consumidores, precisamos apenas utilizar o método **registerSale()** do **SalesCenterFacade**.

Neste EJB utilizamos o recurso de injeção de dependência usando a anotação **@Inject**, provido pela API CDI da plataforma Java EE, para realizar a associação do **SalesCenterFacade** com a instância da classe que implementa a interface **ISalesRepository**. Esta interface estabelece o contrato de persistência dos serviços do SalesCenter.

Através da anotação **@MySQL** presente em **SalesCenterFacade**, informamos ao container CDI que a implementação que estamos

interessados em receber nesta injeção contém um qualificador que deve ser considerado no processamento desta dependência. Nas **Listagens 13 e 14** temos, respectivamente, o código fonte da interface **ISalesRepository** e da classe **SaleRepositoryQualifiers**. Esta última possui os qualificadores criados para identificação precisa da classe alvo de nosso interesse que será instanciada para a injeção da dependência. Esta classe injetada é a que realiza o contrato de **ISalesRepository**.

**Listagem 12.** EJB que implementa o componente Facade da aplicação SalesCenter.

```
@Stateless  
public class SalesCenterFacade {  
  
    @Inject @MySQL  
    ISaleRepository repository;  
  
    public Order registerSale(Order order) {  
        return repository.save(order);  
    }  
  
    public boolean getStatusSale(Order order) {  
        return repository.checkStatus(order);  
    }  
  
    public boolean cancelSale(Order order) {  
        return repository.cancel(order);  
    }  
}
```

**Listagem 13.** Interface que estabelece as funções de persistência do SalesCenter.

```
public interface ISaleRepository {  
  
    public abstract Order save(Order sale);  
    public abstract boolean cancel(Order sale);  
    public abstract boolean checkStatus(Order sale);  
}
```

**Listagem 14.** Qualificadores criados para classificação de classes que implementam a interface **ISalesRepository**.

```
public class SaleRepositoryQualifiers {  
  
    @Qualifier  
    @Retention(RUNTIME)  
    @Target({TYPE, METHOD, FIELD, PARAMETER})  
    public @interface MySQL {}  
  
    @Qualifier  
    @Retention(RUNTIME)  
    @Target({TYPE, METHOD, FIELD, PARAMETER})  
    public @interface Fake {}  
}
```

Criamos estes qualificadores para a injeção de dois tipos de **ISaleRepository**: um “falso”, o qual não persiste as informações e retorna sempre as mesmas respostas inventadas (mocks), e a outra implementação sendo a “verdadeira” do **ISaleRepository**. Esta última implementação é a classe **MySQLDatabase**, que leva consigo a anotação **@MySQL**, que efetiva a gravação dos pedidos em uma base de dados MySQL. Esta classe é apresentada na **Listagem 15**.

**Listagem 15.** Classe que realiza o contrato de persistência do SalesCenter e registra as informações no MySQL.

```
@MySQL
public class MySQLDatabase implements ISaleRepository {

    private final static Logger LOGGER = LoggerFactory.getLogger(MySQLDatabase.class);

    public Order save(Order sale) {
        EntityManagerFactory factory = null;
        try {
            factory = Persistence.createEntityManagerFactory("SalesCenterJPA");
            EntityManager em = factory.createEntityManager();

            em.getTransaction().begin();
            /**
             * Persist Customer
             */
            checkPersistenceCustomer(sale, em);
            /**
             * Persist Products
             */
            checkPersistenceProduct(sale, em);
            sale.setDateTime(Calendar.getInstance().getTime());
            em.persist(sale);
            em.getTransaction().commit();
        } catch (Exception e) {
            LOGGER.error(e.getMessage(), e);
            throw new RuntimeException(e);
        } finally {
            factory.close();
        }
        return sale;
    }

    public boolean cancel(Order sale) {
        return false;
    }

    public boolean checkStatus(Order sale) {
        return false;
    }

    private void checkPersistenceProduct(Order sale, EntityManager em) {
        List<OrderItem> ordersItens = new ArrayList<OrderItem>();
        for(Iterator i = sale.getItens().iterator(); i.hasNext();) {
            OrderItem item = (OrderItem).next();
            if (item.getProduct().getId() != null) {
                Product product = em.find(Product.class, item.getProduct().getId());
                if (product == null) {
                    product = item.getProduct();
                    product.setId(null);
                    em.persist(product);
                }
                item.setProduct(product);
            } else {
                Product product = item.getProduct();
                em.persist(product);
                item.setProduct(product);
            }
            ordersItens.add(item);
        }
        sale.setItens(ordersItens);
    }

    private static void checkPersistenceCustomer(Order sale, EntityManager em) {
        if (sale.getCustomer().getId() != null) {
            Customer c = em.find(Customer.class, sale.getCustomer().getId());
            if (c == null) {
                c = sale.getCustomer();
                c.setId(null);
                em.persist(c);
            }
            sale.setCustomer(c);
        }
    }
}
```

Podemos observar que além de implementar e cumprir o contrato da interface **ISalesRepository**, **MySQLDatabase** contém a anotação do qualificador **@MySQL**. Esta classe grava as informações do pedido que chegam via o objeto **Order** utilizando a API JPA. Lembre-se que a classe **Order** provém do componente (JAR) que compõe o modelo canônico de nossa corporação, e estas classes estão com as configurações de anotações para uso do JPA.

Com isso chegamos ao final da análise da parte transacional de nossa solução. No próximo tópico vamos analisar um pequeno exemplo de monitoramento das atividades de negócio. Para isso, utilizaremos uma simples solução web – que faz o uso de JavaScript como Front-End – para ilustrar em tempo real a captura de eventos durante as transações.

### Monitoramento – MiniJS-BAM

Criamos esta simples aplicação web/JavaScript somente para tornar mais dinâmica e interessante a demonstração de nossa solução baseada em arquitetura orientada a eventos. De fato, é nessa parte que reside o “marketing” de todo o esforço, pois esta é a “janela” que é apresentada aos senhores de negócio, que investem seu orçamento em tecnologia.

É importante mencionar que esta mini aplicação de monitoramento de atividades de negócio não foi testada em ambiente produtivo com alta escala e sob alto estresse de carga. Como previamente apontado, é uma aplicação ilustrativa para demonstrar o resultado de nossa solução, criada apenas para este contexto. Para ambientes mais robustos e complexos, existem no mercado soluções prontas que são coesas às famílias de produtos de seus fabricantes.

Por exemplo: para a IBM Integration Bus (outrora conhecida como IBM WebSphere Message Broker) ferramenta ESB da IBM, temos a opção de utilizar o IBM Business Monitor como ferramenta de monitoração de atividades de negócio. As duas ferramentas possuem recursos que permitem a integração entre elas provida pelo fabricante. Assim, na maioria das vezes, temos apenas o trabalho de configurar a comunicação entre estas ferramentas e, é claro, talvez a parte mais difícil, implementar uma solução de arquitetura que faça uso intensivo destes Monitores de Negócio.

A aplicação MiniJS-BAM é composta por apenas um arquivo WAR (Web Archive) Java EE, que possui como principais componentes os seguintes itens:

# Elaborando projetos com a Arquitetura Orientada a Eventos

- **MiniJsBamController:** Servlet que recebe as requisições via protocolo HTTP, no formato REST, realiza as consultas e retorna os resultados no mesmo formato;
- **MiniJSBamServices:** Classe Java comum (POJO) que executa as consultas na base de dados MySQL via JDBC, respondendo as requisições enviadas pelo Servlet Controller.

O código fonte do **MiniJsBamController** é apresentado na **Listagem 16**. Esta classe desempenha o papel de controlador principal, que recebe as requisições dos clientes que desejam informações sobre eventos já capturados durante a execução de transações de negócios. Este servlet é disponibilizado com quatro diferentes padrões de URL para receber as chamadas dos clientes (podemos observar o atributo **urlPatterns** na anotação **@WebServlet**). Assim, conforme o padrão de URL utilizado pelo cliente no request, decidimos quais informações devemos disponibilizar na resposta.

No método principal do servlet recepcionamos as requisições via GET, analisamos a URI fornecida e, através dela, definimos qual ação do serviço de consulta deve ser executada.

Como o foco de nosso artigo é Arquitetura, o código fonte da classe **MiniJsBamServices** não é tão relevante para ser apresentado. Por isso seu conteúdo não está presente na forma de listagem, mas está disponível para download junto com toda a solução exemplo. A classe **MiniJsBamServices** apenas realiza

a comunicação com a base de dados, executando as queries de consulta nas tabelas em que foram gravadas as informações durante a captura dos eventos pelo Esper, e para isto ela utiliza a API JDBC.

## Solução em Ação

Com todas as peças de nossa solução ligadas e funcionando, podemos ver em ação o monitoramento executando enquanto os requests de pedidos chegam ao nosso barramento de serviços.

Os componentes e as soluções neles implantadas que devem estar funcionando são:

- Mule ESB – Responsável pela disponibilização do Serviço Registro de Pedidos;
- Apache TomEE+ – Responsável por prover o ambiente de execução Java EE ao Sistema de gerenciamento de pedidos (SalesCenter);
- MySQL – Base de dados responsável pelo registro dos eventos capturados pelo Esper no Mule ESB.

Para iniciar a simulação, abrimos o SoapUI, responsável por executar os requests dos clientes conforme explicado nos tópicos anteriores. Uma vez iniciado o cenário de testes na ferramenta, simultaneamente abrimos no navegador a nossa aplicação de monitoramento MiniJS-BAM. Desta forma, em tempo real, podemos acompanhar as informações capturadas nos eventos durante

**Listagem 16.** Servlet Controller da mini aplicação web MiniJS-BAM.

```
@Named
@WebServlet(name="MiniJsBamController",
    urlPatterns = {"/totalOrderByCities", "/totalOrderByDate",
    "/totalOrderByTimeFrame", "/ping"}, loadOnStartup = 1)
public class MiniJsBamController extends HttpServlet {

    private static final long serialVersionUID = 1L;

    private MiniJsBamServices services = new MiniJsBamServices();

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws IOException {
        String path = req.getServletPath();

        String jsonContent = "";
        switch (path) {
            case "/totalOrderByCities":
                jsonContent = this.transformToJson( this.services.getTotalOrderByCity() );
                break;
            case "/totalOrderByDate":
                int dateRequest = Integer.parseInt(req.getParameter("date"));
                jsonContent = this.transformToJson( this.services.getTotalOrderByDate(
                    dateRequest));
                break;
            case "/totalOrderByTimeFrame":
                String timeFrameRequest = req.getParameter("timeFrame");
                jsonContent = this.transformToJson(
                    (this.services.getTotalOrderByTimeFrame(timeFrameRequest)));
                break;
            case "/ping":
                jsonContent = this.ping();
        }
        resp.getWriter().write(jsonContent);
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws IOException {
        resp.getWriter().write("Sorry! We are working only with GET for the sake of
        simplicity, this is not a real project.");
    }

    private String transformToJson(Object obj) {
        String result = "";
        ObjectMapper mapper = new ObjectMapper();
        mapper.setSerializationInclusion(Include.NON_NULL);
        mapper.enable(SerializationFeature.INDENT_OUTPUT);
        mapper.enable(SerializationFeature.CLOSE_CLOSEABLE);
        mapper.configure(SerializationFeature.WRAP_ROOT_VALUE, false);

        try {
            result = mapper.writeValueAsString(obj);
        } catch (JsonProcessingException e) {
            throw new RuntimeException(e);
        }
        return result;
    }
}
```

o movimento das transações que passam pelo barramento de serviços. Na **Figura 15** apresentamos a tela da nossa monitoração de eventos.

Nesta imagem podemos observar que as informações que capturamos utilizando o EPL do Esper estão sendo mostradas em gráficos como, por exemplo, o total de pedidos diários, o total de pedidos diários nas principais cidades e o total de pedidos realizados nos últimos trinta segundos.

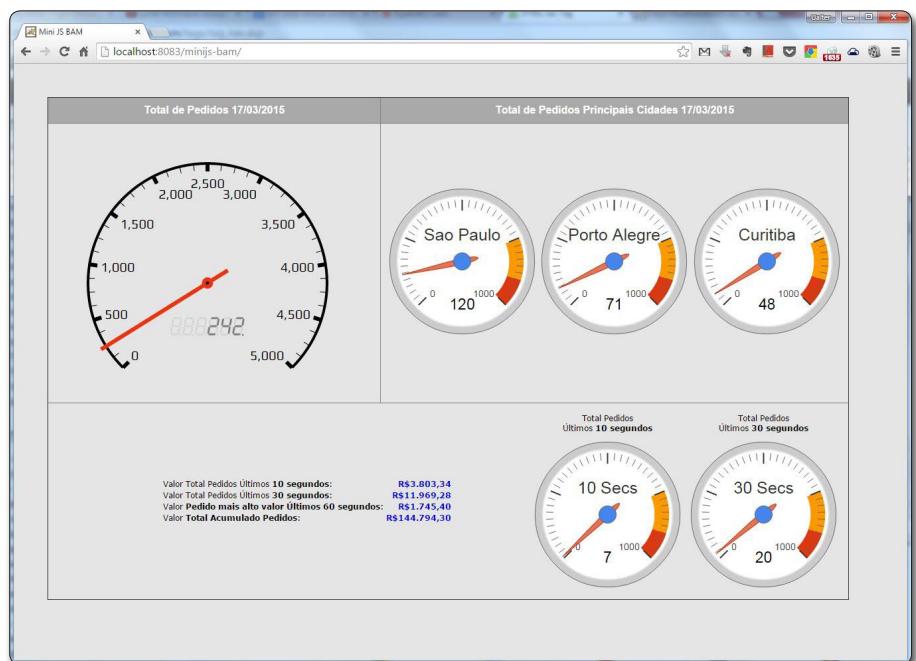
A integração entre sistemas sempre será um tema presente no contexto de tecnologia da informação. Dificilmente, dentro de uma corporação, haverá uma aplicação que não necessitará de informações de outros sistemas para complementar ou aprimorar o seu trabalho. Se aparentemente isso parecer existir, pode acreditar que existe uma planilha Excel em alguma pasta de algum funcionário da empresa, cheia de macros VBA, que está fazendo o trabalho de integração de informações que poderia/ deveria ser feito automaticamente pelos sistemas.

Não organizar e controlar a integração de sistemas nos leva a um ambiente muito mais complexo para realizar manutenções e também evoluir numa velocidade satisfatória para o negócio (*time-to-market*). Sem uma organização satisfatória, os custos são normalmente mais altos e os riscos maiores quando precisamos realizar alterações corretivas ou evolutivas nos sistemas do ambiente.

É neste ponto que a arquitetura tem um papel importante quanto a definições e gerenciamento de integrações de sistemas. Para auxiliar na implantação de uma arquitetura de integração, existem opções desde as mais simples, como adotar boas práticas e padrões de integrações, até opções para o estabelecimento de arquiteturas mais complexas, como SOA e EDA.

A Arquitetura Orientada a Eventos tem aspectos que são interessantes para realizar integrações, como: baixo acoplamento e disponibilização de informações simultaneamente à ocorrência de fatos. Fatos estes que são os eventos os quais a corporação, departamentos e/ou indivíduos, têm total interesse em conhecer e analisar.

Mas como sempre, toda definição deve ser baseada principalmente no bom senso. Ou seja, os benefícios devem compensar os custos e dificuldades de implantação de uma arquitetura complexa de integração. Por isso, somente após uma análise do ambiente, identificando seus problemas e carências, é que se pode afirmar se uma Arquitetura Orientada a Eventos será uma boa opção de escolha.



**Figura 15.** Tela de monitoração do MiniJS-BAM dos eventos capturados durante as transações

## Autor



**Walter Azambuja Junior**

[ualter.junior@gmail.com](mailto:ualter.junior@gmail.com)

Atua como profissional na área de tecnologia há dezenove anos, dos quais doze dedicados à plataforma Java no desenvolvimento de softwares e soluções de Arquitetura de Aplicação/Integração/Serviços. Trabalha como Arquiteto de TI para a GFT em São Paulo/SP, é pós-graduado em Tecnologia Orientada a Objetos e possui as certificações: SCJP, SCWCD e SCEA.



## Links:

**Comparação dos recursos que estão disponíveis entre as versões do Apache TomEE.**

<http://tomee.apache.org/comparison.html>

**Documentação de referência da linguagem específica de domínio EPL da ferramenta Esper.**

[http://esper.codehaus.org/esper-4.2.0/doc/reference/en/html/epl\\_clauses.html](http://esper.codehaus.org/esper-4.2.0/doc/reference/en/html/epl_clauses.html)

**Site da empresa EsperTech, desenvolvedora dos produtos Esper, EsperHA, Esper Enterprise Edition.**

<http://esper.codehaus.org/>

**Esper Online, área para teste e execução de declarações de código EPL.**

<http://esper-epl-tryout.appspot.com/epltryout/mainform.html>

## Você gostou deste artigo?

Dê seu voto em [www.devmedia.com.br/javamagazine/feedback](http://www.devmedia.com.br/javamagazine/feedback)

Ajude-nos a manter a qualidade da revista!



# Gerenciamento de dependências no Java

## Conheça as principais ferramentas para gerir as bibliotecas de seu projeto

**L**inguagens orientadas a objetos são, hoje em dia, dominantes na maioria dos sistemas corporativos em funcionamento. Com diversos benefícios como o baixo acoplamento e a divisão de responsabilidades, a criação e manutenção de aplicações projetadas a partir desse conceito se torna uma tarefa extremamente produtiva e, devido a isso, altamente atrativa para a indústria.

Essa produtividade se deve, principalmente, à capacidade de reuso de componentes criados seguindo os conceitos da orientação a objeto. Explicando um pouco mais, linguagens que seguem esse tipo de paradigma (orientado a objetos) tendem a permitir que o desenvolvedor crie módulos totalmente desacoplados, ou seja, sem nenhuma dependência entre si. Portanto, a solução de um problema maior pode ser dividida em diversas soluções menores que, em conjunto, resolvem o objetivo principal.

A vantagem disso é que, caso surja um novo projeto que seja semelhante a um já resolvido, essas pequenas soluções podem ser reaproveitadas, evitando duplicação de código e outros problemas.

No Java, o reuso de código pode ser feito através da utilização de pacotes JAR, arquivos compactados que contêm classes já desenvolvidas e que podem ser adicionados a um projeto. No entanto, para projetos de grande porte, a configuração manual dessas bibliotecas se torna extremamente complexa e custosa, especialmente do ponto de vista de manutenção da aplicação, o que pode gerar um oneroso trabalho na busca e inclusão de todos os arquivos necessários para o funcionamento de uma biblioteca.

Visando resolver esse problema, foram criadas algumas ferramentas chamadas de **Gerenciadores de dependências** que têm como objetivo gerenciar essas bibliotecas externas ao projeto. Em nosso artigo, pretendemos introduzir algumas delas, explicando seu funcionamento e mostrando suas vantagens e desvantagens. Também pretendemos, no decorrer da explicação,

### Fique por dentro

Este artigo será útil por apresentar as principais ferramentas que existem para gerenciar e adicionar dependências externas a projetos Java, mostrando as vantagens e desvantagens de cada uma e em quais cenários se encaixam melhor. Conheceremos o Maven, o Ivy e o Gradle, soluções de grande relevância por simplificar o controle da execução de tarefas repetitivas que podem tirar o foco do desenvolvedor no que realmente agrega valor ao projeto, as regras de negócio.

apresentar exemplos de situações em que o uso de dependências se faz necessário, mostrando sua utilidade no dia a dia.

### Reuso de componentes no Java

A aplicação dos conceitos do paradigma de orientação a objetos dentro de um projeto tende a permitir que o desenvolvedor possa dividir sua aplicação em módulos de baixo acoplamento entre si. E graças a essa baixa dependência entre os componentes, é possível a utilização dos mesmos em não somente um, mas em diversos outros projetos.

Por padrão, no Java, o uso desses pacotes é possível ao adicionar o arquivo JAR no classpath da aplicação em desenvolvimento. Para quem não está familiarizado com o termo classpath, este representa o caminho, dentro de sua máquina, que a JVM utiliza para carregar, dinamicamente, as classes e bibliotecas que são utilizadas dentro de um sistema Java.

Portanto, a adição de dependências pode ser feita simplesmente copiando arquivos .jar para os diretórios de classpath de nossa aplicação, sem nenhum segredo. O grande problema, no entanto, vem quando precisamos gerenciar essas dependências, ou seja, alterar versões, corrigir conflitos de pacotes e encontrar as dependências encadeadas dos arquivos JAR que desejamos utilizar.

Exemplificando, vamos imaginar que, em um projeto, se faça uso da biblioteca *A.jar*. Esse pacote, por sua vez, faz referência à biblioteca *B.jar* e *C.jar*. Portanto, para conseguirmos utilizar **A**, precisamos, obrigatoriamente, adicionar **B** e **C** em nosso classpath,

uma tarefa um tanto trabalhosa. Pior ainda, caso **B** ou **C** façam referência a uma biblioteca *D.jar*, também precisaríamos adicionar essa ao nosso projeto. Através desse exemplo é fácil observar como uma situação de gerenciamento de dependências pode fugir do controle de nosso time de desenvolvimento, causando um trabalho enorme no gerenciamento de um projeto.

### Ferramentas para gerenciar suas dependências

Com o intuito de resolver esse problema, existem os chamados gerenciadores de dependências. Esse tipo de aplicação serve para que seja possível, ao desenvolvedor, facilmente definir as bibliotecas que devem ser incluídas, bem como a versão de cada uma.

Além disso, uma das grandes vantagens desse tipo de ferramenta é a capacidade de realizar o download das bibliotecas automaticamente, através do uso de certos sites que fornecem o chamado repositório de dependências. Esses repositórios servem como uma espécie de arquivo público com diversos pacotes, que por sua vez podem ser baixados através dos gerenciadores de dependências.

Em nosso artigo, iremos abordar três ferramentas para gerenciar dependências: o Apache Maven, o Apache Ivy e, por fim, o Gradle. Pretendemos introduzir as características de cada uma, bem como suas vantagens e desvantagens, expondo exemplos de utilização de cada uma.

### Configurando nossa IDE

Antes de começarmos a desenvolver nossos exemplos, no entanto, é necessário adequarmos nossa IDE a trabalhar com esses três tipos de ferramentas. Para isso, existem diversos plug-ins prontos que, uma vez baixados e instalados, facilitam a utilização e o desenvolvimento com essas ferramentas.

Nesse artigo, a IDE de escolha para trabalharmos será o Eclipse. Caso você não tenha essa aplicação instalada, o download da mesma pode ser feito através do endereço que apresentamos no fim do artigo, na seção **Links**.

Uma vez que o Eclipse esteja instalado, precisamos baixar nossos plug-ins. Isso é feito clicando no menu *Help > Eclipse Marketplace*, localizado no menu superior da IDE, e escolhendo, dentro da lista de plug-ins disponíveis, os que iremos utilizar. Para auxiliar no processo de encontrar os plug-ins, na **Figura 1** colocamos todos os que iremos utilizar e que precisam ser instalados pelo Eclipse Marketplace.

Esses plug-ins servem para, respectivamente, controlar as ferramentas de gerenciamento de dependências do Gradle, Apache Ivy e Maven. Será necessário baixar um de cada vez e, ao final da instalação, reiniciar o Eclipse para que as alterações tomem efeito.

### Apache Maven

Agora que configuramos nossa IDE, vamos introduzir nossa primeira (e talvez a mais famosa) ferramenta de gerenciamento de dependências. O Apache Maven foi criado com o princípio de ser uma aplicação capaz de automatizar o processo de compilação, muito semelhante ao Apache Ant e, também, trazer outras diversas funcionalidades interessantes ao processo de build de um projeto.

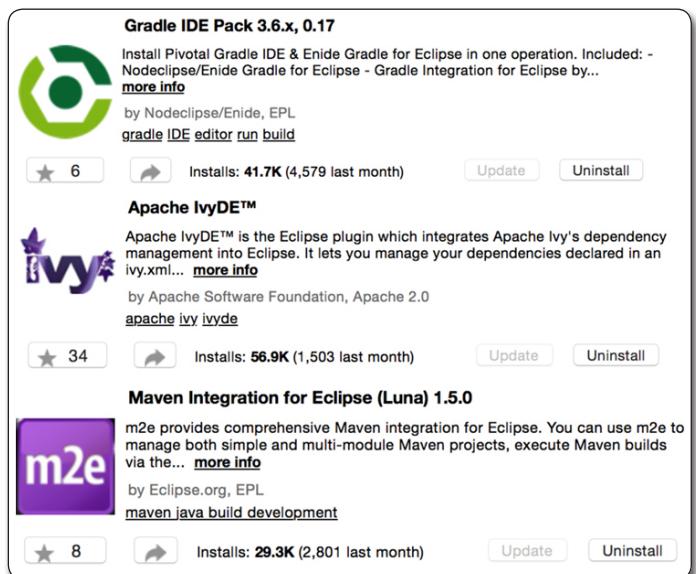


Figura 1. Plugins a serem baixados pelo Eclipse Marketplace

#### Nota

Caso você queira saber um pouco mais sobre o Apache Ant, adicionamos um link ao fim do artigo com algumas informações sobre esse projeto da Apache.

Entre essas funcionalidades, uma das mais úteis e que será o foco de nosso artigo é a capacidade de automatizar o gerenciamento de dependências de um projeto. Seu funcionamento é bastante simplificado: através de um repositório online, o Maven baixa todas as bibliotecas especificadas dentro do projeto em que está sendo utilizado.

Essas bibliotecas, por sua vez, são definidas dentro de um arquivo XML denominado *pom.xml*, onde, além das dependências, também podemos especificar alguns passos no processo de build de um projeto. Na **Listagem 1** apresentamos um exemplo de *pom.xml*, que usaremos mais adiante, em nosso projeto exemplo.

Após essa análise, podemos constatar que o *pom.xml* possui um formato bastante intuitivo. Em primeiro momento, definimos uma tag “pai”, que representa nosso projeto, denominada **<project>**. Dentro dela, por sua vez, incluímos algumas informações de nosso projeto, tal como versão, nome do projeto, tipo de empacotamento, entre outras.

Logo em seguida, definimos outra tag, chamada de **<build>**. Esta será responsável por especificar os itens relacionados exclusivamente ao processo de construção de nosso projeto como, por exemplo, o tipo do compilador, a versão do Java que desejamos utilizar e se queremos usar algum plugin ou não.

Por fim, o trecho que mais nos interessa está entre as tags **<dependencies>**. É neste espaço que podemos especificar as bibliotecas que iremos utilizar como dependência de nosso projeto, através de tags **<dependency>**. É importante notar que, para cada dependência do Maven, precisamos definir três itens: **groupId**, **artifactId** e **version**.

# Gerenciamento de dependências no Java

O groupId (especificado através da tag <groupId>) representa, na maioria dos casos, a empresa ou projeto do qual a biblioteca em questão faz parte. No caso de nosso projeto, como estamos usando uma biblioteca do projeto **Apache Commons**, nosso groupId ficou definido como **org.apache.commons**.

**Listagem 1.** Exemplo de pom.xml para definir as dependências do Maven.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>br.com.javamagazine</groupId>
<artifactId>mavenProject</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>
<build>
    <sourceDirectory>src</sourceDirectory>
    <plugins>
        <plugin>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.1</version>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
            </configuration>
        </plugin>
    </plugins>
</build>
<dependencies>
    <dependency>
        <groupId>org.apache.commons</groupId>
        <artifactId>commons-lang3</artifactId>
        <version>3.3.2</version>
    </dependency>
</dependencies>
</project>
```

Logo após, na tag <artifactId>, informamos o nome da biblioteca em si e, na tag <version>, a versão da mesma. Em nosso caso, determinamos os seguintes valores: **commons-lang3** e **3.3.2**, respectivamente.

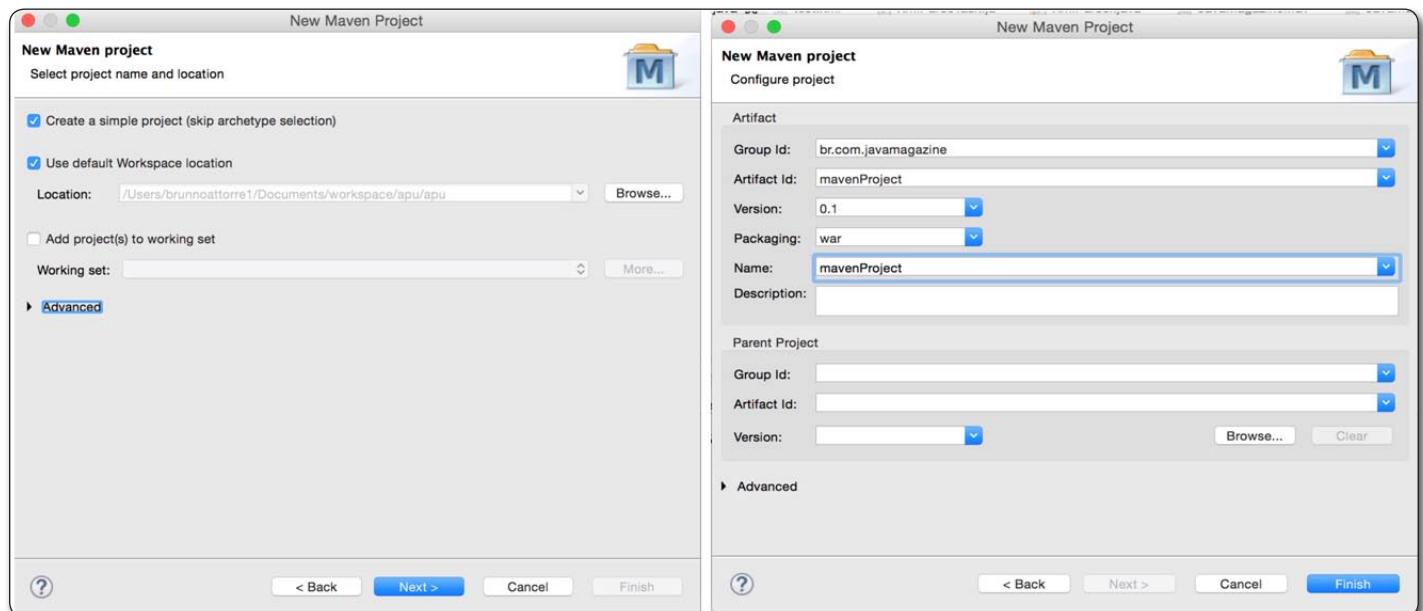
## Nota

Para auxiliar os desenvolvedores a encontrar as bibliotecas que necessitam, os repositórios remotos do Maven normalmente disponibilizam uma interface web que permite a busca de bibliotecas e devolve, como retorno, as configurações do pom.xml para adicionar a mesma em seu projeto. Na seção **Links** adicionamos o caminho para o site Maven repository, que possui uma interface desse tipo.

Agora que entendemos as partes do arquivo *pom.xml*, vamos criar nosso projeto que utiliza o Maven como gerenciador de dependências. Para isso, com o plugin do Maven instalado no Eclipse, basta criarmos um projeto do tipo *Maven Project*. Na **Figura 2** apresentamos as telas de criação para nosso projeto exemplo e os passos que devemos seguir.

Conforme podemos verificar, a criação de um projeto Maven é dividida em um wizard composto por duas telas principais. A primeira, localizada à esquerda, apresenta algumas configurações básicas como a localização de nosso projeto (*workspace*) e, também, se desejamos utilizar algum tipo de *Working set* do Eclipse. É importante notar ainda que selecionamos o primeiro checkbox dessa tela, que indica que não queremos usar nenhum *archetype* do Maven para criar nosso projeto.

Os chamados archetypes são espécies de templates que nos ajudam a criar um esqueleto de nosso projeto, com as source folders e configurações iniciais a partir de diversos modelos pré-definidos (aplicações web, EARs, aplicações desktop, etc.). Como em nosso exemplo iremos criar o projeto sozinhos, optamos por não os utilizar.



**Figura 2.** Telas para configuração do projeto Maven

Na tela seguinte temos a opção de definir o nome do nosso projeto. É interessante observar que, assim como as dependências que adicionamos, os projetos no Maven são nomeados por três atributos: *Group id*, *Artifact id* e *Version*. Para o nosso projeto, definimos os nomes como **br.com.javamagazine**, para o *groupId*, e **mavenProject** para o *artifact id*. Em relação à versão, escolhemos atribuir ao nosso exemplo o versionamento 0.1, ficando a critério do leitor atribuir uma versão diferente ao projeto que está criando.

Uma vez criado o projeto, podemos ver que foi adicionado um arquivo *pom.xml* à raiz do mesmo, local onde também adicionaremos as suas dependências. Para isso, vamos copiar o trecho entre as tags **<dependencies>** da **Listagem 1** e adicioná-lo no *pom.xml* que acabou de ser criado. Feito isso, para permitir que o plugin do Maven faça o download automático das bibliotecas, basta clicarmos com o botão direito em cima do projeto e escolher a opção *Run As > Maven Install*.

Logo após, você poderá observar pelo console de saída do Eclipse, que o Maven automaticamente irá baixar as bibliotecas indicadas no *pom.xml* e também empacotar seu projeto de acordo com as instruções do mesmo arquivo. Caso você queira testar a biblioteca recém-baixada do Apache Commons, apresentamos na **Listagem 2** o código da classe **TesteDependencias**, que utiliza essa dependência para imprimir um texto no terminal da IDE.

#### **Listagem 2.** Classe de exemplo que utiliza as dependências do Apache Commons.

```
public class TesteDependencias {  
  
    public static void main(String[] args) {  
        System.out.println(StringUtils.capitalize("usando a dependência do Apache  
        Commons!"));  
    }  
}
```

#### **Listagem 3.** *pom.xml* com o uso do módulo **moduloMaven**.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"  
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0  
                            http://maven.apache.org/xsd/maven-4.0.0.xsd">  
    <modelVersion>4.0.0</modelVersion>  
    <groupId>JavaMagazineMaven</groupId>  
    <artifactId>JavaMagazineMaven</artifactId>  
    <version>0.0.1-SNAPSHOT</version>  
    <packaging>pom</packaging>  
    <build>  
        <sourceDirectory>src</sourceDirectory>  
        <plugins>  
            <plugin>  
                <artifactId>maven-compiler-plugin</artifactId>  
                <version>3.1</version>  
                <configuration>  
                    <source>1.8</source>  
                    <target>1.8</target>  
                </configuration>  
            </plugin>  
        </plugins>  
    </build>  
    <dependencies>  
        <dependency>  
            <groupId>org.apache.commons</groupId>  
            <artifactId>commons-lang3</artifactId>  
            <version>3.3.2</version>  
        </dependency>  
    </dependencies>  
    <modules>  
        <module>../moduloMaven</module>  
    </modules>  
</project>
```

A segunda mudança, indicada pelas tags **<modules>** e **<module>**, serve para, explicitamente, indicar onde está localizado nosso módulo. Como ainda não criamos esse módulo, para que essa configuração funcione, podemos seguir os mesmos procedimentos para criação de um projeto Maven e criar um novo, com o nome **moduloMaven**.

Uma vez criado esse projeto, precisamos apenas modificar seu *pom.xml* para referenciarmos o projeto pai do qual esse subprojeto depende – em nosso exemplo, o projeto **JavaMagazineMaven**. Na **Listagem 4** apresentamos o *pom.xml* do projeto recém-criado indicando esse relacionamento.

Nesse código podemos ver que o relacionamento de um projeto com seu “pai” é indicado pela tag **<parent>**. Dentro dessa tag, relacionamos todos os dados do projeto pai, inclusive seu caminho relativo.

Assim que os dois projetos estiverem configurados, ao rodar o comando *mvn install* no projeto pai, veremos que todos os projetos declarados na seção de módulos serão compilados e seus pacotes finais adicionados à pasta *target* de cada um.

#### **Vantagens e desvantagens do Maven**

Agora que completamos nosso primeiro exemplo utilizando o Maven, vamos analisar os pontos positivos e negativos de usar essa tecnologia.

Conforme dissemos no início do artigo, a tarefa de gerenciamento de dependências é extremamente complexa.

#### **Múltiplos módulos**

Além dos archetypes e do gerenciamento de dependências, o Maven também permite a criação de módulos dentro de seus projetos. Um módulo funciona como uma dependência, mas, dentro do workspace, é representado por um projeto Java tradicional.

Para trabalhar com módulos, basta adicionar essa informação em nosso *pom.xml*. Visando demonstrar esse recurso, vamos alterar esse arquivo em nosso projeto Maven adicionando algumas linhas. Na **Listagem 3** apresentamos o novo *pom.xml*, já com as informações de nosso módulo.

Foram realizadas duas alterações para a inclusão do novo módulo. A primeira, relacionada à tag **<packaging>**, é a modificação do tipo de empacotamento de **jar** para **pom**. Essa mudança é necessária para projetos Maven que utilizam módulos em sua construção e irá, automaticamente, disparar o build e empacotamento dos subprojetos declarados no *pom.xml*.

É importante observar que, ao escolher o tipo de empacotamento **pom**, nosso projeto pai deixa de ser compilado e passa a servir somente de referência para que os projetos declarados na seção de módulos sejam empacotados.

# Gerenciamento de dependências no Java

Listagem 4. Pom.xml indicando o relacionamento entre o módulo e o projeto pai.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
<parent>
    <groupId>JavaMagazineMaven</groupId>
    <artifactId>JavaMagazineMaven</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <relativePath>..</relativePath>
</parent>
<modelVersion>4.0.0</modelVersion>
<groupId>br.com.javamag</groupId>
<artifactId>moduloMaven</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>moduloMaven</name>
</project>
```

Com a capacidade de baixar automaticamente as dependências de repositórios online, o Maven traz como grande vantagem a praticidade e simplicidade na configuração dessas bibliotecas e, como ponto extra, também permite a criação de esqueletos de aplicações, através dos archetypes. Essa automação, se comparada com a configuração manual dos projetos, traz um ganho enorme tanto em manutenção, ao evitar a verificação e atualização das bibliotecas manualmente, como em velocidade de desenvolvimento, diminuindo bastante a complexidade do gerenciamento das dependências.

Além disso, ao especificar seu projeto com os mesmos atributos que definem uma dependência (criando o **artifactId**, **groupId** e **version**), o Maven permite que, facilmente, possa se compartilhar o projeto como dependência de outros projetos. Essa facilidade de uso é um dos pontos mais vantajosos do Maven e, devido a isso, podemos dizer que essa ferramenta é uma das mais utilizadas por desenvolvedores em todo o mundo.

No entanto, também existem alguns pontos negativos com a utilização dessa ferramenta. O primeiro deles, relacionado ao gerenciamento de dependências, é a dificuldade de se eliminar conflitos de dependências, ou seja, quando temos duas dependências iguais (mas de versões diferentes) no mesmo projeto. Apesar de ser possível excluir bibliotecas, quando temos uma árvore muito grande de dependências, a tarefa de encontrar e remover as bibliotecas em conflito se torna bastante complicada.

Ainda como ponto negativo, pelo fato do Maven se basear em XMLs para sua configuração, a configuração do processo de build se torna limitada a instruções pré-definidas, não dando muito espaço para customização. Para conseguir realizar manipulações mais complexas dentro do processo de build, como usar alguma lógica de programação, é necessária a adoção de outra ferramenta, como o Apache Ant, que já abre mais espaço para builds customizados.

Por fim, os próprios archetypes, implementados com o objetivo de servir como forma de criar esqueletos de projetos, também não são passíveis de customização, deixando o desenvolvedor engessado nos padrões adotados pelo Maven.

## Apache Ivy

Com o intuito de possibilitar uma maior liberdade para o desenvolvedor construir seus projetos, a Apache desenvolveu outro projeto, implementado em paralelo ao Apache Maven, chamado Apache Ivy. Essa ferramenta tem como objetivo propiciar as vantagens do gerenciamento de dependências do Maven integradas com um processo de build mais dinâmico.

Para isso, esse processo utiliza outra solução da Apache, o chamado Apache Ant. Para quem não está familiarizado, o Ant permite que se especifique instruções para a construção de um projeto, tais como cópia de arquivos e execução de comandos shell, através de um arquivo XML denominado *build.xml*. Você pode encontrar a URL oficial do projeto na seção [Links](#).

As dependências do Ivy, por sua vez, são gerenciadas pelo próprio Ivy, em um arquivo denominado *ivy.xml*. Sua configuração é bastante semelhante ao *pom.xml* do Maven e um exemplo desse arquivo pode ser visto na [Listagem 5](#).

Listagem 5. Exemplo de arquivo ivy.xml.

```
<ivy-module version="2.0">
    <info organisation="java-mag" module="ivy"/>
    <dependencies>
        <dependency org="commons-lang" name="commons-lang" rev="2.0"/>
        <dependency org="commons-cli" name="commons-cli" rev="1.0"/>
    </dependencies>
</ivy-module>
```

Como podemos verificar, a sintaxe do Ivy é muito semelhante à do Maven, com apenas algumas pequenas diferenças. As bibliotecas declaradas como dependências são, da mesma forma que no Maven, especificadas dentro da tag **<dependencies>**, porém, em vez de possuírem os parâmetros **groupId**, **artifactId** e **version**, característicos do Maven, possuem os atributos **org**, **name** e **rev** que, em conjunto, identificam a dependência que queremos utilizar.

Ainda de modo semelhante ao Maven, onde utilizamos a tag **project**, declaramos na tag **info** do Ivy as informações de nosso projeto. E dentro dessa tag também declaramos o atributo **organisation** e **module**. É importante notar que, ao contrário do Maven, o Ivy não possui instruções de build dentro do *ivy.xml*. Estas instruções, conforme dissemos anteriormente, ficam em um arquivo à parte, chamado *build.xml*, e são rodadas pelo Ant.

Para conseguirmos executar as tarefas de baixar as dependências e construir nosso projeto localizadas nesse arquivo, temos que instalar mais alguns plug-ins no Eclipse. Isso pode ser feito escolhendo a opção *Help > Install New Software* na interface de sua IDE e, na próxima janela, buscar pelo plugin *Apache Ivy Ant Tasks* dentro dos softwares disponíveis para download. Na [Figura 3](#) mostramos qual a biblioteca que deve ser baixada para conseguirmos prosseguir com o nosso exemplo.

Uma vez instalado, basta adicionarmos o JAR desse plugin ao nosso processo de build do Ant. Isso pode ser feito nas preferências do Eclipse, escolhendo, nas opções da tela de preferências,

localizadas no menu lateral à esquerda, a opção *Ant > Runtime*. Uma vez dentro desse menu de configuração de *Runtime* do Ant, basta clicar no botão *Add External Jar* e escolher, dentro do diretório de instalação do Eclipse, na pasta *plugins*, o JAR do Ivy, que deve ter a nomenclatura semelhante a *org.apache.ivy\_2.X.X.XXXXXXXXXXX.jar*.

Feito isso, basta construirmos um arquivo de build, o que é alcançado definindo o arquivo *build.xml*, como pode ser visto na **Listagem 6**. Este servirá como exemplo para adicionarmos o Ant em nosso projeto Ivy e automatizarmos o processo de build de nosso projeto exemplo, que criaremos mais adiante.

Neste código podemos ver alguns comandos característicos do Ant. O primeiro deles é a divisão do processo de build por **targets**, ou seja, sub-tarefas, responsáveis por executar um conjunto de instruções. Dentro de cada uma dessas sub-tarefas, temos as instruções do Ant, como criação de diretórios, através da tag **<mkdir>**, e compilação e empacotamento de projetos, através das tags **<javac>** e **<jar>**.

Para incluirmos o Ivy no nosso processo de build do Ant, adicionamos uma tag especial denominada **<ivy:retrieve>**, que serve para indicar, dentro do Ant, que desejamos que sejam baixadas todas as dependências do arquivo *ivy.xml*. Dessa forma, conseguimos unir o processo de build do Ant com o gerenciamento de dependências do Ivy.

Por fim, para testarmos todas as nossas configurações, basta criarmos um projeto Java simples em nossa IDE e adicionar, no diretório raiz do projeto, os arquivos *ivy.xml* e *build.xml* apresentados nas **Listagens 5 e 6**. Uma vez adicionados, podemos clicar com o botão direito em cima do arquivo *build.xml* e escolher a opção *Run As > Ant Build* para baixar as dependências e construir nosso projeto.

### Vantagens e desvantagens do Ivy

Como informado no início da discussão sobre o Ivy, essa ferramenta foi criada com o propósito de eliminar a dificuldade de customizar o gerenciamento de dependências e processos de build, como acontece no Maven, permitindo a criação de tarefas complexas de construção de projetos de maneira mais simples.

Esse propósito é parcialmente atendido com o uso do Apache Ant, através da customização disponível por essa aplicação, porém o Ivy ainda traz algumas limitações. A primeira delas, que podemos verificar claramente, é a necessidade de termos duas ferramentas distintas para conseguir alcançar esse objetivo, uma vez que o Ivy sozinho só realiza o gerenciamento de dependências e não a construção de projetos. A complexidade de configuração de um ambiente e a necessidade de manutenção de ambas as

aplicações e seus arquivos pode, muitas vezes, ser um problema maior que a solução trazida pela ferramenta.

Em segundo lugar, apesar do dinamismo e flexibilidade no processo de build do Apache Ant, esta ferramenta ainda nos deixa presos a instruções de build definidas na sua especificação, o que pode ser visto como uma grande limitação à automatização dos builds.

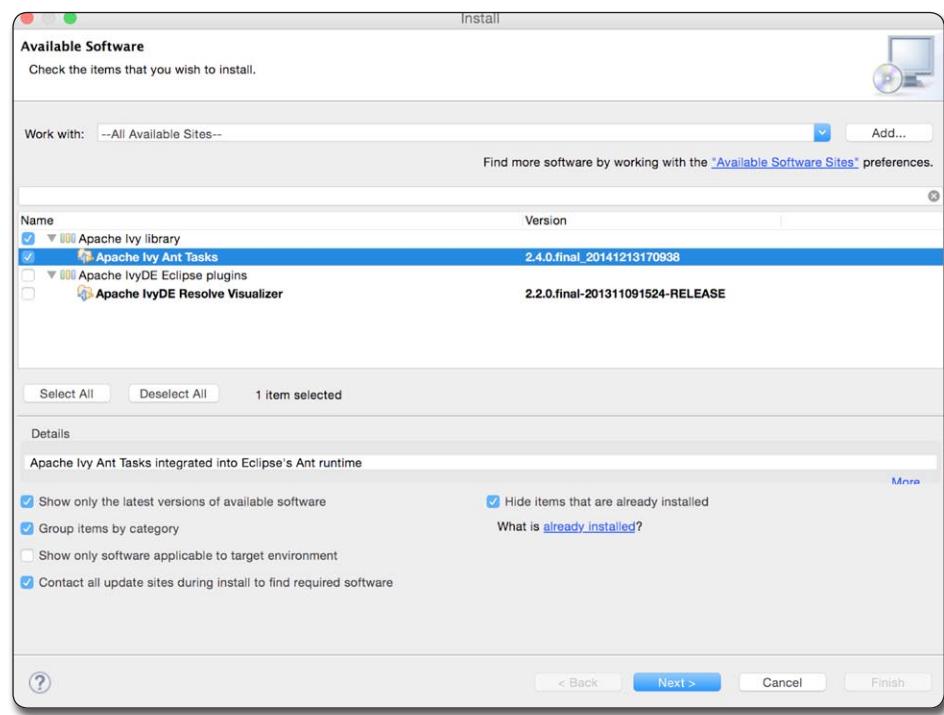
**Listagem 6.** Arquivo build.xml usando a integração do Ant com o Ivy.

```
<project xmlns:ivy="antlib:org.apache.ivy.ant" name="JavaMagazineIvy"
default="dist" basedir="">
<property name="src" location="src" />
<property name="build" location="build" />
<property name="dist" location="dist" />
<taskdef resource="org/apache/ivy/ant/antlib.xml" uri="antlib:org.apache.ivy.ant"/>
<target name="init">
    <tstamp />
    <mkdir dir="${build}" />
</target>

<target name="compile" depends="init" description="compile the source">
    <ivy:retrieve/>
    <javac srcdir="${src}" destdir="${build}" />
</target>

<target name="dist" depends="compile" description="generate the distribution">
    <mkdir dir="${dist}/lib" />
    <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar" basedir="${build}" />
</target>

<target name="clean" description="clean up">
    <delete dir="${build}" />
    <delete dir="${dist}" />
</target>
</project>
```



**Figura 3.** Plugin necessário para rodar as tarefas do Ivy dentro do Ant

Apesar do leque de instruções ser bem maior que a oferecida pelo Maven, tarefas fora do padrão podem se tornar extremamente complexas de implementar devido à limitação mencionada, ainda mais quando precisamos de algo em nossos processos que exige lógicas extensas ou tarefas fora do padrão do Ant como, por exemplo, o uso de lógica de programação para configurar nossos pacotes.

## Gradle

Com a solução desses problemas em mente, foi desenvolvida a ferramenta para gerenciamento de dependências Gradle. Eliminando o uso de XMLs e aplicando, dentro do processo de compilação e empacotamento, o uso de uma linguagem de programação, o Gradle disponibiliza uma ferramenta completa de gerenciamento e automatização de builds.

Seu princípio fundamental é o uso da linguagem Groovy nos scripts de build permitindo, assim, que qualquer tarefa seja programável, utilizando recursos de linguagens de programação como a criação de funções, loops e ifs. Ademais, o Gradle disponibiliza uma sintaxe simples para declarar as dependências, semelhante ao modo de declaração do Maven e, ainda, possibilita a invocação de scripts Ant.

Portanto, podemos dizer que o Gradle é uma ferramenta extremamente poderosa, principalmente quando precisamos criar scripts de build um pouco mais complexos. Inclusive, diversos projetos importantes, como o SDK do Android e o Hibernate, já utilizam o Gradle em seus desenvolvimentos, mostrando a confiabilidade e qualidade desta solução.

## Criando um projeto com o Gradle

Iniciando nosso exemplo sobre o Gradle, o processo de construção de um projeto que utiliza essa ferramenta se baseia nas instruções declaradas no arquivo *build.gradle*. Assim como no Ivy ou no Maven, esse arquivo serve para explicitarmos nosso processo de build e, também, para especificarmos nossas dependências.

Como diferença, no Gradle esse arquivo não se baseia mais em um documento XML e, sim, na linguagem de programação Groovy. Além desse arquivo, o Gradle também utiliza o arquivo *settings.gradle*, responsável por definir algumas propriedades de build do projeto e, principalmente, gerenciar a criação dos chamados **multi-projects**.

Esse tipo de projeto (multi-projects) é bastante útil para desenvolvedores que pensam em melhorar a organização de seu código, quebrando um sistema grande em subprojetos menores, cada um com suas respectivas dependências. Uma abordagem como essa permite maior desacoplamento entre as partes do projeto e, no caso de alguma das partes precisar ser reutilizada, fica muito mais simples o uso somente do que é necessário.

Outro grande atrativo do Gradle é a facilidade de criação e gerenciamento desse tipo de abordagem (multiprojetos). Assim, pretendemos, nesse tópico, apresentar como podemos aliar essa funcionalidade ao gerenciamento de dependências.

Um exemplo de arquivo *settings.gradle* pode ser visto a seguir, onde expomos a configuração de nosso multi-project exemplo:

```
include "java_mag1";java_mag2"
```

Neste caso, especificamos que nosso multiprojeto será composto por dois subprojetos: um chamado *java\_mag1* e outro chamado *java\_mag2*.

Para testarmos o funcionamento do Gradle, vamos mover esse arquivo para uma pasta dentro do nosso diretório definido como Workspace. No nosso caso, criamos uma pasta denominada *java\_mag* e adicionamos, dentro dela, o arquivo *settings.gradle*. É válido observar que ainda não criamos nosso projeto na IDE.

Feito isso, vamos ao próximo passo: criar o *build.gradle*. Nesse momento, vamos criar apenas um arquivo de texto em branco (com o nome *build.gradle*) no mesmo diretório de nosso *settings.gradle*. Além disso, também precisamos criar os diretórios *java\_mag1* e *java\_mag2*, na mesma pasta em que adicionamos os arquivos *build.gradle* e *settings.gradle*, permitindo que o Eclipse possa criar esses subprojetos em seu workspace assim que importamos o projeto principal.

Com a estrutura de diretórios pronta, já é possível para o plugin do Eclipse reconhecer a existência de um projeto Gradle e seus projetos filhos, através do arquivo *settings.gradle* (onde declaramos o nome dos subprojetos), e criar os mesmos dentro de nossa IDE. Para isso, basta escolher a opção *Import > Gradle Project* na IDE e, na tela que surgir, selecionar o diretório em que criamos os arquivos e pastas e clicar no botão *Build Model*. Na **Figura 4** apresentamos essa tela e como ela deve ficar após seguir os passos supracitados.

Como podemos verificar, no Eclipse o nosso projeto Gradle foi quebrado em três: o projeto principal, denominado *java\_mag* e, logo em seguida, os dois subprojetos (*java\_mag1* e *java\_mag2*), conforme declarado no *settings.gradle*. Para importá-los, basta selecionar o checkbox à esquerda do nome do projeto e clicar em *Finish*.

Outro ponto importante nessa configuração multi-project está relacionado à herança das dependências. No Gradle todos os subprojetos irão herdar todas as dependências do projeto pai automaticamente e podem, através de configurações no *build.gradle*, herdar dependências de outros subprojetos.

É importante ressaltar que, para cada projeto, o arquivo *build.gradle* localizado na sua raiz irá identificar quais são as suas dependências. Em casos de subprojetos, se o arquivo *build.gradle* não existir (como no nosso exemplo), esses projetos irão assumir somente as dependências do projeto principal.

Portanto, se quisermos adicionar dependências no projeto *java\_mag*, basta alterar o arquivo de configuração em sua raiz (*build.gradle*) e caso seja necessário incluir diferentes dependências em *java\_mag1* ou *java\_mag2*, basta adicionar as configurações dentro de um arquivo *build.gradle* em cada um deles.

Deste modo, temos flexibilidade para configurar nossas dependências exatamente onde quisermos.

Como exemplo, demonstraremos a seguir como adicionar algumas dependências no nosso projeto pai, o chamado **java\_mag**. Para isso, basta abrir o arquivo *build.gradle* desse projeto e adicionar as linhas presentes na **Listagem 7**.

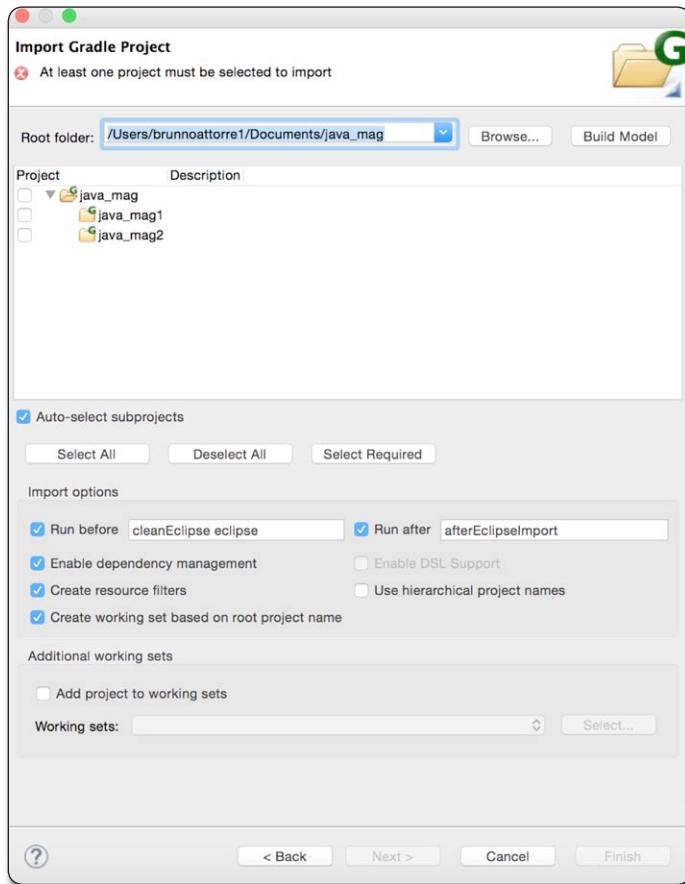
Nessa listagem podemos ter uma ideia da sintaxe do arquivo de dependências do Gradle. Definimos, na primeira linha, que estamos trabalhando com um projeto Java, através do plugin **java** e, logo em seguida, explicitamos qual será nosso repositório; neste caso, o Maven Central.

**Listagem 7.** Arquivo *build.gradle* – Definindo as dependências de nossos projetos.

```
apply plugin:'java'

repositories {
    mavenCentral()
    flatDir {
        dirs'libs'
    }
}

dependencies {
    compile'org.apache.commons:commons-lang3:3.3.2'
}
```



**Figura 4.** Tela de configuração de nosso projeto Gradle

Por fim, no trecho denominado **dependencies**, declaramos a dependência que desejamos utilizar. É interessante notar que, de modo semelhante ao Maven, cada dependência é composta por três partes, separadas pelo caractere ":" neste caso. Essas partes são, respectivamente, o **group**, representando o projeto ou empresa do qual a biblioteca faz parte, o **name**, que indica o nome da dependência em si e, por último, a versão da biblioteca que queremos adotar.

Uma vez adicionadas as dependências ao nosso arquivo, basta clicar com o botão direito em cima do projeto *java\_mag* na IDE Eclipse e escolher a opção *Gradle > Refresh Dependencies*. Assim, as dependências declaradas no arquivo serão baixadas e uma pasta denominada *Gradle Dependencies* será adicionada ao projeto.

#### Plugins do Gradle e a construção de pacotes

No último exemplo, o leitor mais atento deve ter verificado que utilizamos o plugin Java do Gradle. Neste tópico, explicaremos o funcionamento desse plugin e, também, faremos uma breve introdução sobre alguns dos principais plug-ins que podemos adotar em nossos processos de build do Gradle e suas respectivas formas de utilização.

O primeiro deles, já introduzido no exemplo anterior, é o plugin responsável pela compilação das classes do nosso projeto. Para executá-lo adicionamos a seguinte linha em nosso arquivo de configuração: **apply plugin: 'java'**. E além de compilar o código, ele é responsável por empacotar o mesmo em um arquivo JAR.

Tarefas como a compilação e o empacotamento de classes em um JAR podem ser disparadas em nosso processo de build através da criação das chamadas **tasks**. Estas são definidas dentro do arquivo *build.gradle*, onde podemos criar diversas tarefas, que funcionam como funções.

Por padrão, os plugins do Gradle já trazem algumas tarefas implementadas, consideradas essenciais para seu funcionamento. O plugin do Java, por exemplo, já vem com uma tarefa denominada **jar** que permite o empacotamento do projeto em um arquivo *.jar*.

Para testar essa tarefa, basta entrar no Eclipse, clicar com o botão direito em nosso projeto e escolher a opção *Task Quick Launcher*. Logo em seguida, na caixa de texto que será exibida na IDE, digite a task que deseja executar. Ao pressionar o botão *Enter*, esta task será executada. Portanto, para empacotarmos nosso projeto como um JAR, basta digitar a palavra *jar* e deixarmos o build do Gradle rodar automaticamente.

Uma vez empacotado, o arquivo gerado será colocado na pasta *build/libs* da raiz do projeto. Caso seja necessário executar uma instrução mais customizada, fora dos padrões, o Gradle permite ainda que algumas definições de suas tasks sejam sobreescritas, como é o caso do diretório em que são gerados os pacotes. Na **Listagem 8** apresentamos o arquivo *build.gradle* sobrepondo algumas configurações default da task de geração do arquivo JAR.

Com o exemplo podemos ver como é fácil, dentro do Gradle, sobrepor uma task. Neste caso, em vez de sobrepor a implementação de toda a função, apenas sobreponemos a propriedade **destinationDir**.

# Gerenciamento de dependências no Java

## Listagem 8. Sobrecrevendo a tarefa de geração do JAR no arquivo build.gradle.

```
apply plugin:'java'

repositories {
    mavenCentral()
    flatDir {
        dirs 'libs'
    }
}

jar {
    destinationDir= file("pacote")
}
dependencies {
    compile 'org.apache.commons:commons-lang3:3.3.2'
}
```

## Listagem 9. Utilizando e sobrecrevendo a tarefa de geração do war no arquivo build.gradle.

```
apply plugin:'java'
apply plugin:'war'

repositories {
    mavenCentral()
    flatDir {
        dirs 'libs'
    }
}

war {
    webXml = file('web.xml')
}
jar {
    destinationDir= file("pacote")
}
dependencies {
    compile 'org.apache.commons:commons-lang3:3.3.2'
}
```

Deste modo, ao rodarmos a task **jar**, o arquivo será gerado na pasta **pacote**, em vez de na pasta default.

Outro plugin para geração de pacotes é o plugin para criação de arquivos do tipo WAR. Assim como a task de geração de arquivos JAR, essa task gera o pacote automaticamente em um diretório padrão, mas também abre espaço para customização de algumas propriedades.

Na **Listagem 9** apresentamos um exemplo de uso do plugin war do Gradle para gerar um pacote desse tipo, também adotando algumas customizações.

A sobreescrita dessa task, como podemos verificar, é muito semelhante à que fizemos com a task jar. A diferença é que, neste caso, sobrecrevemos a localização do arquivo *web.xml*, indicando onde desejamos que o Gradle o busque e o insira no pacote.

Feito isso, para testar essa tarefa, precisamos criar um arquivo com o nome *web.xml* e adicioná-lo na raiz de nosso projeto. Para auxiliar o leitor, apresentamos na **Listagem 10** um *web.xml* vazio, que pode ser utilizado neste exemplo.

## Listagem 10. Arquivo web.xml exemplo para nosso projeto.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    id="WebApp_ID" version="2.5">

</web-app>
```

## Conhecimento faz diferença!

The magazine covers include:

- Issue 24: Gerência de Configuração (Definição + Ferramentas)
- Issue 28: Evolução do Software (Definições, preocupações e custo)
- Issue 29: Automação de Testes (Cuidados a serem tomados na implantação)

Faça já sua assinatura digital! | [www.devmedia.com.br/es](http://www.devmedia.com.br/es)

## Faça um upgrade em sua carreira

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



DEV MEDIA

Característica/Ferramenta	Maven	Ivy	Gradle
Utilização	Gerenciamento de dependências e templates de projetos (Archetypes).	Gerenciamento de dependências e integração com o Apache Ant para construção de projetos.	Gerenciamento de dependências e automatização no processo de build, com alta possibilidade de customização.
Linguagem principal	XML	XML	Groovy
Popularidade	Talvez o mais popular, sendo muito fácil encontrar um desenvolvedor que saiba utilizar	Um pouco menos popular, mas como tem integração com o Apache Ant (bem conhecido no mercado), se torna mais simples de ser adotado por desenvolvedores já familiarizados com essa ferramenta.	Opção relativamente nova e, apesar de ser considerada uma tendência para a automatização de builds e gerenciamento de dependências, ainda é pouco utilizada.
Capacidade de customização	Um dos principais problemas do Maven está no fato de suas builds serem bastante engessadas. A partir disso, qualquer customização se torna bastante complexa.	Como possui integração com o Apache Ant, a capacidade de customização é mais extensa, mas se limita às ferramentas disponibilizadas pelo Ant.	Por utilizar uma linguagem de programação em vez de simples configurações por XML, o Gradle traz uma facilidade bem maior para customizar e desenvolver processos de build mais complexos.
Usabilidade e plugins	O Maven é bem fácil de ser utilizado e disponibiliza diversos plugins para as mais diversas IDEs. Por ser a ferramenta mais adotada, é fácil encontrar informações e tutoriais a respeito.	Mais complicado, pois o próprio plugin muitas vezes não atende a todas as necessidades do desenvolvedor, além de ser mais difícil de ser configurado.	Bastante simples, principalmente ao utilizar os plugins disponíveis no Eclipse.
Capacidades extras	Os Maven Archetypes possibilitam a criação e utilização de templates, facilitando a criação de projetos novos.	Sua integração com o Ant é algo de grande utilidade para projetos que já se integram com essa ferramenta.	O gerenciamento de multiprojetos e a capacidade de definir processos e dependências diferentes para cada um é extremamente útil para projetos de grande porte.

**Tabela 1.** Comparação entre as ferramentas apresentadas

## Comparando cada uma das tecnologias

Concluída nossa introdução sobre cada uma das tecnologias disponíveis para gerenciamento de dependências, faremos agora uma comparação entre elas de forma a auxiliar na escolha do leitor em seus projetos.

Para isso, criamos a **Tabela 1**, onde apresentamos as principais características destas ferramentas e que funciona como uma espécie de resumo. Estas características foram divididas em tópicos, que avaliamos como relevantes para o leitor e adicionamos, para cada um, as vantagens e desvantagens de cada ferramenta.

É interessante observar que, apesar de todas as ferramentas servirem ao propósito do gerenciamento de dependências, cada uma tem uma característica ou funcionalidade distinta. Seja pela simplicidade de uso ou pelo leque de funcionalidades que disponibiliza, os três gerenciadores que analisamos são soluções valiosas para o desenvolvedor.

Essa definição, dentro de um cenário real, é importantíssima uma vez que tanto o problema a ser solucionado no projeto quanto o conhecimento da equipe podem ser fatores fundamentais na escolha do gerenciador de dependências. Muitas vezes, a ferramenta com menos opções pode ser suficiente para projetos mais simples, não exigindo muito da equipe e permitindo uma maior facilidade de adaptação de um time mais leigo a uma ferramenta não tão robusta.

Deste modo, vale ressaltar que não existe a solução bala de prata, que atende a todas as situações para o gerenciamento de dependências e, sim, uma escolha mais adequada para cada tipo de projeto e equipe com a qual se está trabalhando. Ainda assim, é válido ressaltar que em projetos Java, o Maven é a ferramenta mais adotada.

### Autor



**Bruno F. M. Attorre**

[brattorre@gmail.com](mailto:brattorre@gmail.com)

Trabalha com Java há quatro anos. Apaixonado por temas como Inteligência Artificial, ferramentas open source, BI, Data Analysis e Big Data, está sempre à procura de novidades tecnológicas na área. Possui as certificações OCJP, OCWCD e OCEEJBD.



### Links:

**Página para download da IDE Eclipse.**

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

**Página do Maven Repository para encontrar dependências do Maven.**

<http://mvnrepository.com>

**Página com uma introdução sobre os Maven archetypes.**

<http://maven.apache.org/guides/introduction/introduction-to-archetypes.html>

**Repositório do Maven utilizado no Gradle**

<http://search.maven.org>

### Você gostou deste artigo?

Dê seu voto em [www.devmedia.com.br/javamagazine/feedback](http://www.devmedia.com.br/javamagazine/feedback)

Ajude-nos a manter a qualidade da revista!



# Big Data: MapReduce na prática

## Como adotar o MapReduce em seu dia a dia

**C**onjuntos de dados têm como solução ideal o uso de um modelo de processamento paralelo e distribuído que se adapta a qualquer volume e grau de complexidade. Esse é o caso do MapReduce, uma técnica que abstrai os detalhes de paralelização e distribuição do processamento de dados, que pode ser utilizada em aplicações que necessitem dessas características, como ocorre com algumas abordagens não triviais, a exemplo do tratamento do “big data” e do processamento de algoritmos de alta complexidade e escalabilidade.

Tais algoritmos podem ser empregados em diversas situações que envolvam resolver problemas de busca e otimização, normalmente presentes em sistemas de tomada de decisão e descoberta de conhecimento. Por exemplo, escolher qual a melhor empresa para investir o capital (quantia) na bolsa de valores; solucionar problemas de agendamento e planejamento de recursos; auxiliar na organização e alocação de turmas a professores, presente na definição de grades horárias de trabalho; e qualquer situação que necessite uma boa solução (entre tantas), considerando as regras específicas para o domínio do problema a fim de alcançar o melhor resultado possível. Como podemos notar, são problemas complexos, muitas vezes de difícil solução e que envolvem significativas reduções de custos, melhorias dos tempos de processos e/ou melhor alocação dos recursos em atividade.

Como o MapReduce pode ajudar na solução desses problemas? Em essência, a resposta está na capacidade do poder de processamento paralelo e distribuído da técnica, fatores que permitem alta escalabilidade à solução.

O MapReduce é baseado no paradigma de programação funcional, adotando duas funções que dão nome ao modelo: a função *map* e a função *reduce*. Esse modelo estabelece uma abstração que permite construir aplicações com operações simples, escondendo os detalhes da paralelização. Em resumo, tais funções transformam um grande volume de dados de entrada em um conjunto

### Fique por dentro

Este artigo aborda o uso do MapReduce na solução de um problema complexo, que é melhor resolvido com a técnica de algoritmos genéticos (AG). Apesar da natureza iterativa dos AGs, com algumas adaptações é possível construir aplicações que necessitem de grande poder de processamento e que podem ser executadas de forma paralela e distribuída no modelo MapReduce. Entre os softwares que podem ser beneficiados, há algoritmos de inteligência computacional, como AGs e redes neurais, e qualquer problema de otimização que necessite encontrar boas respostas em situações que normalmente são difíceis de serem resolvidas com aplicações tradicionais em uma única máquina. Por exemplo, sistemas para identificar padrões em biometria, formulação de regras em jogos de estratégia, na busca de melhores rotas (caminhos) a serem percorridas em um conjunto de cidades, como é o caso do exemplo tratado neste artigo, entre outros.

resumido e agregado na saída, sendo cada função executada em uma etapa distinta. Na primeira etapa, *Map*, uma função de mapeamento distribui os dados em diversos nós de processamento e armazenamento. Na segunda etapa, *Reduce*, uma função agrupa e sumariza os resultados obtidos no mapeamento, para gerar um resultado final.

A técnica MapReduce pode ser aplicada em vários campos, como o agrupamento de dados, aprendizado de máquina e visão computacional. Outro exemplo de campo que pode adotar essa técnica é da Inteligência Computacional, em especial o dos Algoritmos Genéticos, que devem tratar uma grande base de dados (a chamada *população de indivíduos*) para localizar valores para uma tomada de decisão. Tais algoritmos exigem um alto custo de processamento para ser executado em uma única máquina, o que torna a técnica MapReduce ideal para ser adotada.

Com base nisso, este artigo demonstra o uso combinado da abordagem AG com MapReduce para resolver um tipo especial de problema complexo, chamado de “caixeiro viajante” (ou PCV). Este problema busca identificar os melhores caminhos para se percorrer um conjunto de cidades, visitando pelo menos uma vez cada cidade em um determinado percurso. O PCV envolve um

número de combinações de caminhos de crescimento exponencial, em função do número de cidades, fato que o torna complexo para ser resolvido com algoritmos tradicionais. Para validar a técnica proposta (AGs adaptados ao modelo MapReduce), um cenário de teste foi aplicado para um conjunto de vinte cidades, o que demonstrou um bom desempenho na geração de boas respostas.

## Pré-requisitos

Este tutorial foi projetado para ser executado em um computador com sistema operacional Linux, seja nativo ou rodando em uma máquina virtual (VMware ou VirtualBox, por exemplo), uma vez que o framework Hadoop (que implementa o MapReduce) utiliza tal ambiente. Em ambos os casos (nativo ou virtualizado), recomenda-se que a memória principal tenha no mínimo 1 Gigabyte e espaço em disco suficiente para comportar a instalação do pacote Hadoop. Por se tratar de uma aplicação que simula a maioria dos dados em memória, o espaço em disco exigido é no mínimo de 2 Gigabytes.

Além do espaço da aplicação, ao final foram utilizados aproximadamente dez gigabytes de espaço em disco para a distribuição Linux (Ubuntu, versão 12), para a instalação do Apache Hadoop (versão 1.2) e para a IDE Eclipse (versão Kepler 4.2).

## Algoritmos Genéticos

A otimização é o processo de encontrar a melhor solução (ou solução ótima) de um conjunto de soluções para um problema. Normalmente, tais problemas envolvem um procedimento para escolha otimizada de recursos, que pode ser de natureza temporal/cronológica, financeira/econômica, de espaço físico, de priorização de tarefas, etc. Sendo assim, um processo de otimização procura, geralmente, maximizar lucros, minimizar perdas, realizar projetos econômicos e seguros, maximizar a capacidade de transmissão de uma rede satisfazendo suas limitações, escolher o melhor investimento, definir quando comprar e quando vender ações na bolsa de valores, traçar o roteiro de viagem e muitos outros exemplos. Em síntese, quando se fala em otimização, está se pensando em maximizar ou minimizar uma função, chamada de *função objetivo*, sujeita a certas restrições.

As técnicas de otimização devem ser utilizadas quando não existe uma solução simples e diretamente calculável para o problema. Isso geralmente ocorre quando a estrutura do problema é complexa ou existem muitas formas possíveis de resolver. Nesses casos, é possível que não exista uma equação direta ou um procedimento matemático ou algorítmico para resolver o problema, de forma que uma técnica de otimização seja a mais indicada para encontrar (ou se aproximar) da melhor resposta possível para o problema.

Para aplicar uma técnica de otimização, dois conceitos são relevantes: o **espaço de busca**, “local” onde todas as possíveis soluções do problema se encontram; e a **função objetivo**, utilizada para avaliar as soluções produzidas, associando a cada uma delas um valor que denota uma nota ou peso a ser considerado na avaliação.

O Algoritmo Genético é uma técnica de otimização inspirada no conceito da evolução natural das espécies. Essa técnica parte da ideia da sobrevivência do indivíduo mais apto em uma população. Traduzindo para o contexto computacional, um indivíduo pode ser a representação de uma informação em um conjunto de dados para um domínio particular, e o mais apto é o indivíduo que está próximo da melhor informação que se tenta localizar no espaço de busca (ou base de dados da solução). Dessa forma, um AG é um procedimento de otimização para encontrar a melhor (ou melhores) resposta(s), sem a necessidade de explorar todas as soluções possíveis nesse espaço de busca.

Para criar uma população de possíveis respostas para um problema, o AG usa um processo evolutivo. Em seguida, combina as melhores soluções para criar uma nova geração de soluções que deve ser melhor do que a geração anterior. Portanto, é um processo iterativo realizado em várias etapas (ou gerações) constituído das seguintes atividades:

- 1. Inicialização:** É a criação da população inicial para o primeiro ciclo do algoritmo. A criação envolve produzir um conjunto de dados pertinentes com o contexto real do problema, podendo ser gerada aleatoriamente por meio de uma rotina automática;
- 2. Avaliação:** Avalia-se a aptidão das soluções analisando a resposta de cada uma ao problema proposto;
- 3. Seleção:** Os indivíduos são selecionados para combinação das características (para a reprodução). A seleção é baseada na aptidão dos indivíduos;
- 4. Cruzamento:** Características das soluções escolhidas são re-combinadas, gerando novos indivíduos;
- 5. Mutação:** Características dos indivíduos resultantes do processo de reprodução são alteradas;
- 6. Atualização:** Os indivíduos criados na iteração da etapa corrente são inseridos na população que será tratada na próxima iteração;
- 7. Finalização:** Verifica se as condições de encerramento da evolução foram atingidas.

Em síntese, o processo inicia com a definição para o conceito de “indivíduo”, que deve ser codificado em uma representação que possa estruturar os dados para a solução do problema. A população inicial de indivíduos é então preparada, geralmente de forma aleatória, seguindo critérios estabelecidos a partir de uma *função objetivo*, que define a aptidão de cada indivíduo ao contexto do problema. Essa aptidão, um valor obtido a partir do cálculo de uma função, é usada para encontrar os indivíduos para cruzamento com outros indivíduos, assim recombinados para criar novos indivíduos que são copiados para a nova geração. Além do cruzamento, alguns indivíduos podem ser escolhidos ao acaso para sofrer mutação, com o objetivo de melhorar a diversidade da população. Após esse processo, uma nova geração é formada e o processo é repetido até que algum critério de parada tenha sido atingido. Nesse ponto, o indivíduo que está mais próximo do ideal é identificado e o processo é concluído.

## Problema do Caixeiro Viajante

O Problema do Caixeiro Viajante (PCV) consiste em encontrar o menor caminho para um caixeiro viajante que, dado um conjunto de cidades, deve visitar todas elas precisamente uma vez até retornar à cidade de origem.

Apesar da simplicidade em sua formulação, este é um problema de difícil solução, tornando-se um dos problemas de otimização mais estudados. A sua complexidade decorre do crescimento exacerbado do espaço de busca que surge a partir dos diversos caminhos a percorrer, uma vez que o tamanho aumenta exponencialmente com o aumento do número de cidades.

Há muitas aplicações reais para o PCV nos campos da logística, do transporte de pessoas e cargas, e na pesquisa operacional de um modo geral. Por esse motivo, muitos algoritmos foram desenvolvidos tentando alcançar os melhores resultados esperados, entre eles a abordagem dos Algoritmos Genéticos.

## Representação do problema

No PCV, o conceito de indivíduo é aplicado à representação do caminho a ser percorrido entre as cidades, e a função objetivo para calcular a aptidão do indivíduo é determinada pela distância obtida para esse caminho. Assim, uma forma de estruturar a codificação do indivíduo é representar o caminho de cidades usando um grafo não direcionado com pesos nas arestas, como será explicado a seguir. Por se inspirar na matemática que lida com a geometria dos planos, essa representação é chamada PCV Simétrico Euclidiano (PCVS).

Considere, por exemplo, um conjunto de dez cidades posicionadas em um plano bidimensional, com dois eixos ( $x$  e  $y$ ) iniciados em zero, como visto na **Figura 1**. No PCVS para este caso (dez cidades), os caminhos são identificados por uma sequência de caracteres, onde cada caractere representa uma cidade utilizando uma letra do alfabeto (A, B, ..., Z). No plano, cada letra é uma coordenada  $x,y$ . Para simplificar a demonstração da solução, optou-se por empregar um conjunto de dez cidades, que segue a sequência de "A" até "J".

Inicialmente, para resolver o PCVS é necessário conhecer as distâncias entre todas as cidades. Considerando que uma cidade ( $C$ ) está localizada em um ponto  $(x, y)$  do plano (mapa), como visto na **Figura 1**, o processo para calcular a distância total ( $D$ ) entre uma cidade origem ( $C_1$ ) e a  $n$ -ésima ( $C_n$ ) cidade (dentro de um caminho válido para  $n$  cidades) é determinado pela soma das distâncias ( $d$ ) das coordenadas  $(x, y)$  das cidades que fazem parte desse caminho, aplicando a fórmula vista na **Figura 2**. Em outras palavras, o que estamos fazendo é calcular o perímetro do polígono formado pela sequência de cidades pertencentes ao caminho válido.

A título de exemplo, adotando-se as coordenadas relativas ( $x, y$ ) para cálculo das distâncias entre as cidades (como visto na **Figura 1**), encontramos os valores disponíveis na **Tabela 1**. Para esse problema, como mencionado anteriormente, as cidades são identificadas por letras, e um caminho é representado pela sequência das letras destas cidades, como no caso "ABCDEFGHIJ".

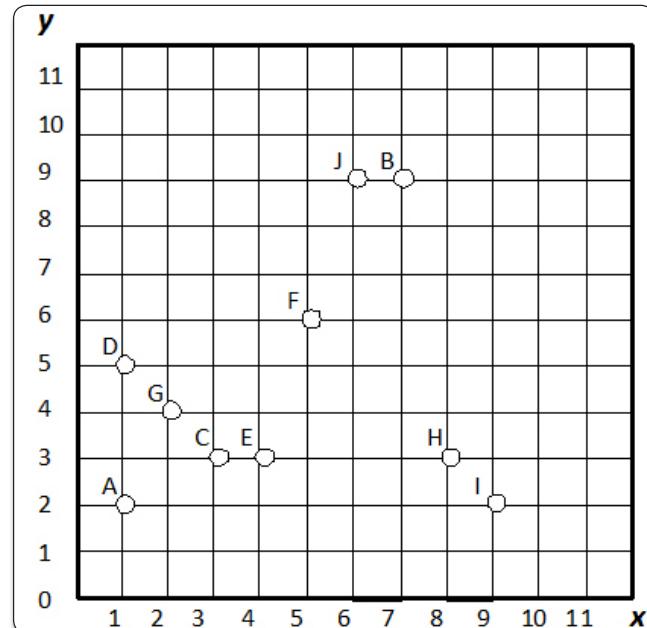


Figura 1. Exemplo de mapa para dez cidades do PCVS

$$D = d(C_n, C_1) + \sum_{i=1}^{n-1} d(C_i, C_{i+1})$$

$$\text{onde: } d(C_i, C_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Figura 2. Equação para calcular a distância entre cidades

Cidade	X	Y
A	1	2
B	7	9
C	3	3
D	1	5
E	4	3
F	5	6
G	2	4
H	8	3
I	9	2
J	6	9

Tabela 1. Coordenadas relativas das cidades no plano euclidiano

Dessa forma, aplicando a equação da **Figura 2** calcula-se a distância percorrida em um caminho. Por exemplo, no cálculo do caminho "ABCDEFGHIJ" chega-se ao valor aproximado de 53.3, conforme demonstra a **Figura 3**.

## Modelagem no MapReduce

A implementação de um Algoritmo Genético no paradigma do MapReduce apresenta alguns desafios. O maior deles é adaptar

a natureza iterativa do AG, onde cada iteração produz uma população ou geração de indivíduos, ao modelo em duas etapas do MapReduce. Embora encontremos algumas propostas na literatura especializada da área, neste trabalho optou-se pela chamada **Dupla Geração**, pois esta se adequa bem e executa eficientemente o AG do PCVS no MapReduce. Em síntese, a Dupla Geração realiza o AG em dois passos de gerações, denominadas de *local* e *global*.

As *gerações locais* são criadas na etapa que executa a função de mapeamento. Com uma população inicial de indivíduos (os dados de entrada), o AG cria novas gerações no processo de mapeamento, resultando em uma população bem melhor (em termos de indivíduos) do que a população inicial. Isso ocorre devido aos operadores genéticos aplicados (seleção, cruzamento e mutação), acrescido do mecanismo chamado de *elitismo*, que acrescenta o indivíduo que possui a melhor aptidão entre todos os outros indivíduos da população inicial. Tal resultado é então enviado para a fase intermediária do framework MapReduce, conhecida como *Shuffle e Sort*.

As *gerações globais* são criadas na etapa da função de redução, a partir da população gerada no mapeamento. Portanto, o AG atua novamente agora na população que vem do mapeamento, formando uma nova população, uma evolução em relação à inicial. O resultado da redução estabelece uma pequena população com os melhores indivíduos, sendo que a melhor resposta é aquele indivíduo que possui a melhor aptidão (menor distância).

A Figura 4 apresenta um esquema resumido do funcionamento do AG no MapReduce. A parte superior da figura mostra as principais atividades envolvidas no processamento: a entrada, o mapeamento, a fase intermediária (*shuffle & sort*), a redução e a saída. Na entrada, a população inicial é gerada randomicamente fora da tarefa MapReduce. Ao entrar no processamento de mapeamento, essa população inicial é dividida e distribuída entre as máquinas que executam a função *map*. Cada divisão forma uma subpopulação local, que é processada pela função *map* através do AG do PCVS. O resultado das fases de mapeamento e intermediária produz uma coleção de dados, constituída por linhas de chave/valor, onde a chave é a *aptidão* e o valor é um *indivíduo*, representado por uma estrutura combinada pelo caminho de cidades e sua aptidão (a distância do caminho). A fase de redução

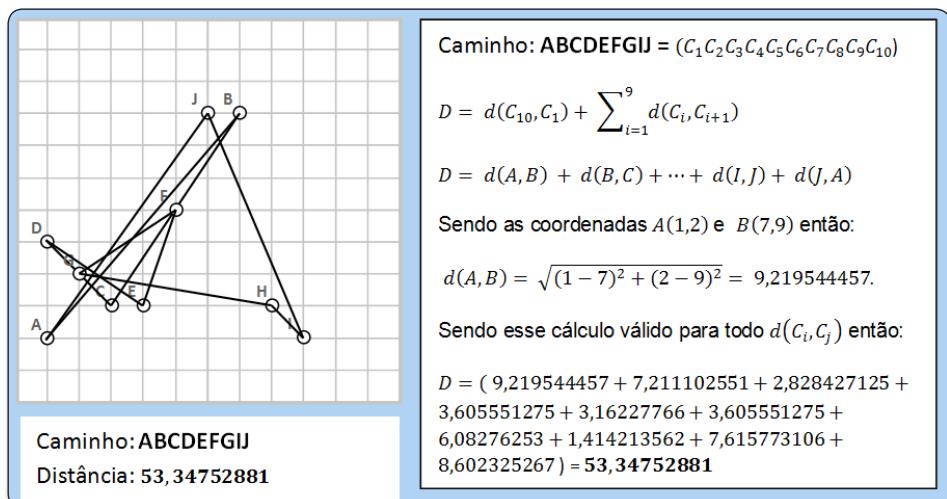


Figura 3. Exemplo do cálculo da distância entre as cidades “ABCDEFGHIJ”

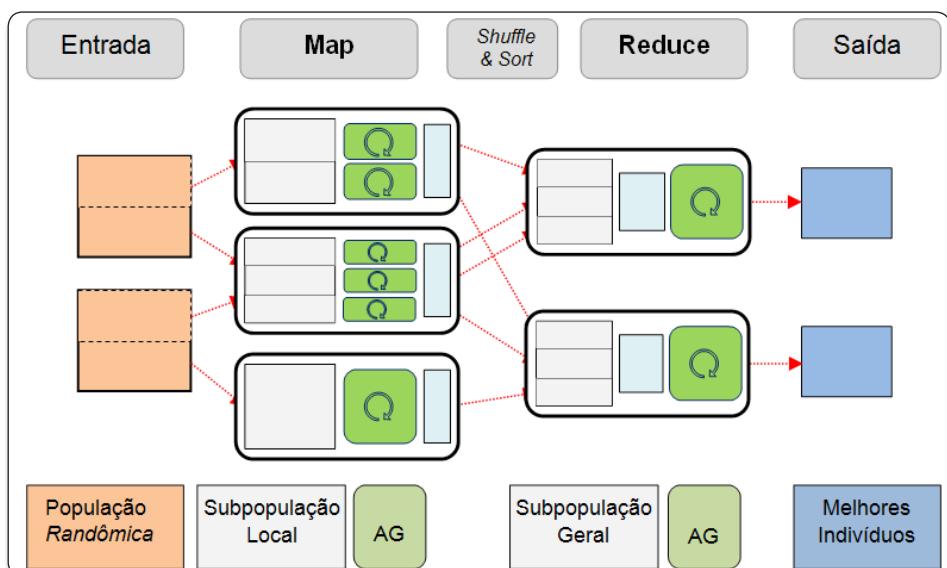


Figura 4. Funcionamento do processamento MapReduce para o Algoritmo Genético

recebe a população formada por indivíduos com boas aptidões (menores distâncias) em relação à população inicial, e depura ainda mais essa população identificando aquele indivíduo com a melhor aptidão.

O poder do processamento paralelo permite dividir a população em várias tarefas *map*, que correspondem à execução paralela de vários processos de AG sobre uma subpopulação. Isso é possível graças a capacidade de executar em paralelo várias tarefas em máquinas distintas, aumentando o desempenho de processamento dos algoritmos genéticos.

Do ponto de vista do fluxo de dados, o funcionamento do AG na técnica MapReduce pode ser descrito em quatro momentos, explicados da seguinte forma:

- 1) A entrada dos dados recebe um arquivo texto, em que cada linha é uma representação codificada de um indivíduo.

Para simplificar o exemplo, cada letra presente na cadeia de strings do indivíduo possui uma correspondência com uma cidade no mapa, e as coordenadas (x, y) de cada cidade está registrada na aplicação. A seguir temos um exemplo de quatro linhas de indivíduos:

```
ABCDEIGHFJ
GIJABEFCDH
BEFGHIJACD
EBGFHCDIJA
```

2) Logo após, para atender as regras do MapReduce, os dados de entrada são estruturados em dois campos: chave e valor. A chave representa o número da linha e o valor é o conteúdo original da linha, que contém o indivíduo, representando o caminho com a sequência das cidades (em forma de letras). A função map avalia e armazena uma quantidade de linhas (indivíduos) em uma lista, formando a população inicial para o processo AG gerar novos indivíduos. Em seguida, esses indivíduos são tratados e codificados na forma do par “aptidão/indivíduo” (chave/valor), sendo que a chave é o arredondamento da aptidão para um número inteiro, e o valor é um par <cromossomo/aptidão>. O resultado produz um conjunto de dados em forma de linhas, como no exemplo a seguir:

```
(53, <ABCDEFGHIJ, 53.3475288096413>)
(51, <GHIJABEFC, 50.6532924781218>)
(53, <EBFHGIJACD, 52.8074130652778>)
(51, <EBHFCDIJGA, 51.4288322764095>)
```

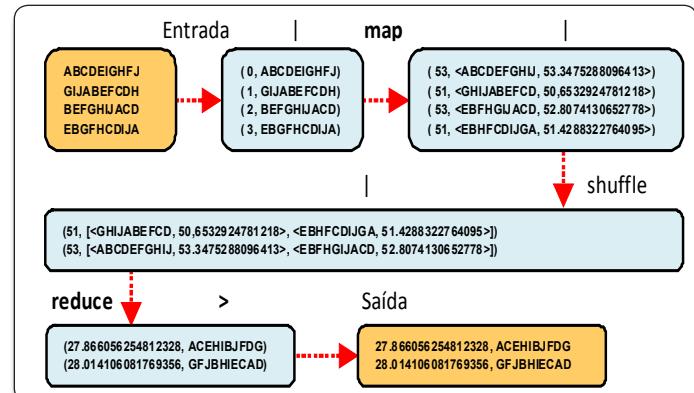
3) Depois, a saída da função map é processada pela fase intermediária (*Shuffle e Sort*), agrupando os pares aptidão/indivíduo (chave/valor) pela aptidão, onde a chave é um número inteiro calculado pelo arredondamento da aptidão (distância aproximada) e o valor é uma lista de indivíduos que pertencem àquela chave pela aptidão aproximada ao número inteiro da chave. Após a fase intermediária, os dados são enviados para a fase reduce da seguinte forma:

```
(51, [<GHIJABEFC, 50.6532924781218>, <EBHFCDIJGA, 51.4288322764095>])
(53, [<ABCDEFGHIJ, 53.3475288096413>, <EBFHGIJACD, 52.8074130652778>])
```

4) Note que para cada chave há uma lista de indivíduos que é usada pela função reduce para gerar uma nova população através do processo AG. Seguindo a característica paralela/distribuída do MapReduce, cada tarefa reduce aplica sobre o “valor” (com a lista de indivíduos) o AG para identificar os melhores indivíduos, produzindo registros de pares “aptidão/indivíduo”, como:

```
(27.866056254812328, ACEHIBJFDG)
(28.014106081769356, GFJBHIECAD)
```

Todo esse processo de fluxo de dados pode ser visto na **Figura 4**.



**Figura 5.** Fluxo de dados do processo MapReduce para o AG

Na essência do processo, as funções map e reduce mantêm o controle dos melhores indivíduos, evidenciados no fluxo de dados da **Figura 5**. No término do processo, é realizada a gravação das saídas da fase reduce em um arquivo no sistema de arquivos do Hadoop (o HDFS).

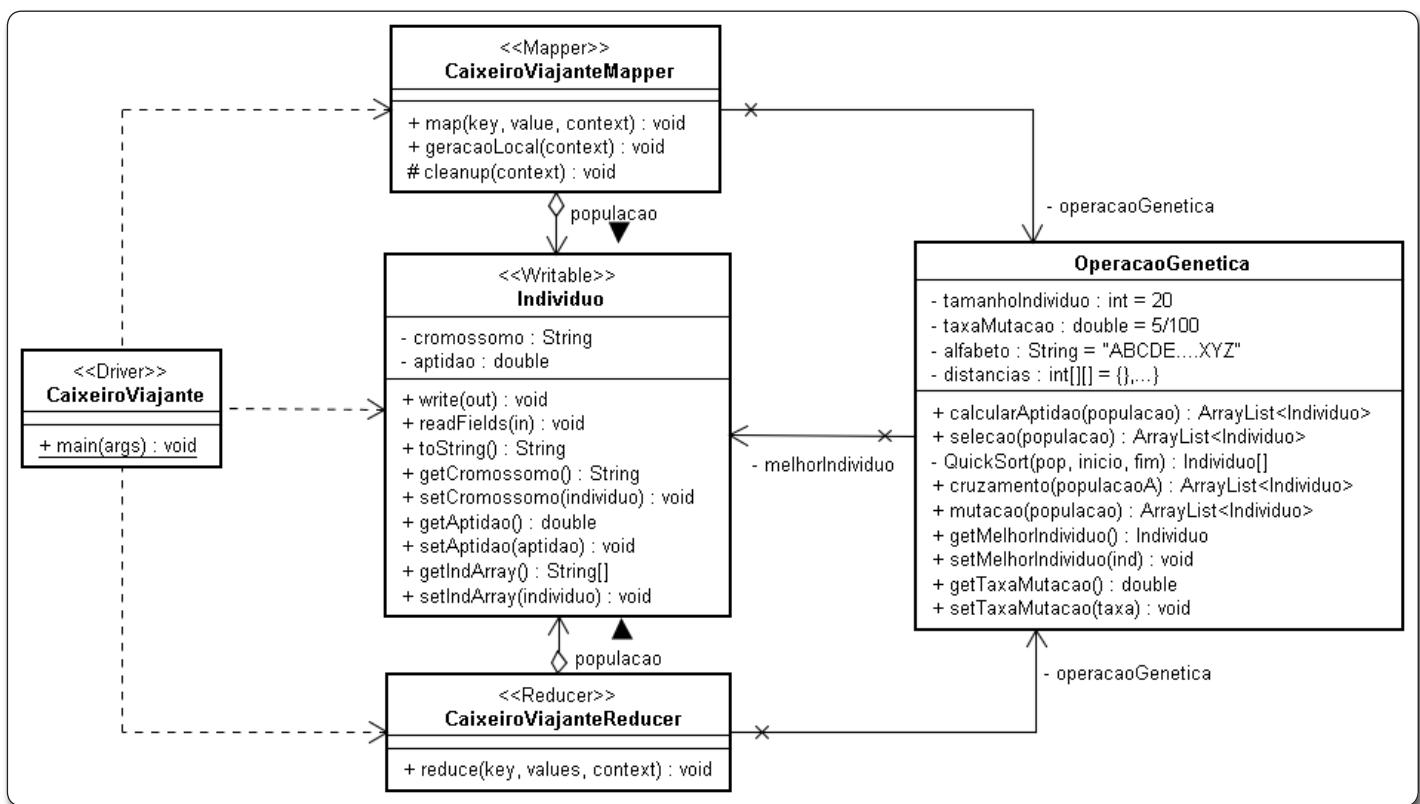
O HDFS (*Hadoop Distributed File System*) é um sistema de arquivos distribuído projetado para armazenar dados em uma rede de computadores de baixo custo. Ele faz parte da arquitetura Apache Hadoop e foi otimizado para aplicações nas quais é necessário ler uma quantidade muito grande de dados.

## Implementação

A aplicação MapReduce para resolver o problema do PCV é construída com base no diagrama de classes da **Figura 6**. Os estereótipos <<Mapper>> e <<Reducer>> denotam os papéis das funções de mapeamento e redução, respectivamente. O estereótipo <<Driver>> é a aplicação principal que orquestra a execução das duas funções. A lógica do AG está toda concentrada em uma classe de apoio, chamada **OperacaoGenetica**, enquanto a classe **Individuo** (estereotipada com <<Writable>>) é utilizada para representar o conteúdo a ser processado pelo AG. O código-fonte destas classes foi implementado em Java e será explicado a seguir.

Considerando que o ambiente de desenvolvimento está montado com o sistema operacional Linux, o IDE Eclipse e o framework Apache Hadoop (versão 1.2), vamos iniciar o passo a passo para a construção do projeto. Neste cenário, o Eclipse ainda deve estar configurado com um plugin para uso do framework Hadoop, que está disponível para download em um arquivo JAR, denominado *hadoop-eclipse-plugin-1.2.jar*, preparado para ser instalado na versão Kepler desta IDE (veja o endereço na seção **Links**). Para isso, adicione o plugin copiando o arquivo JAR para a pasta *plug-ins* do Eclipse. Para configurar o plugin, execute a IDE, acesse o menu *Window* e escolha *Preferences*. No item *Hadoop Map/Reduce*, informe o local onde o Hadoop foi instalado e clique em *Ok*.

Com o plugin instalado e configurado, podemos criar o projeto MapReduce para a aplicação deste artigo. Sendo assim, com o Eclipse aberto, escolha no menu *File* a opção *New Project* e em seguida o item *Map/Reduce Project*, como mostra a **Figura 7**.

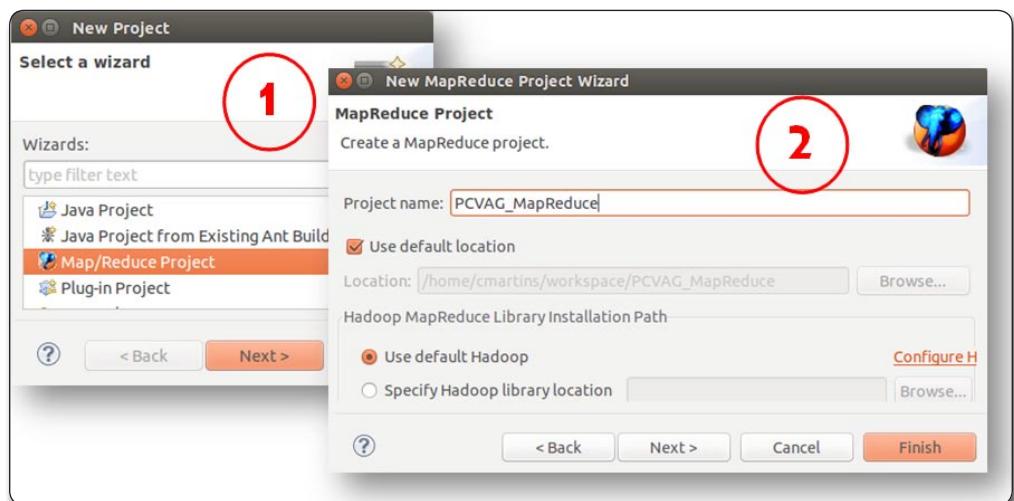


**Figura 6.** Diagrama de classe do PCV-AG para o MapReduce

No passo seguinte, informe o nome do projeto, confirme e aguarde a criação da estrutura do projeto contendo a pasta de código (*src*) e todas as bibliotecas do Hadoop importadas. Agora estamos prontos para implementar as classes da aplicação.

## Codificando o Indivíduo

A Listagem 1 apresenta o código da classe **Individuo**, que instanciará os objetos (dados) trabalhados nas funções *map* e *reduce*. Como esses objetos não seguem os tipos padrões do MapReduce (que são *Integer*, *Float*, *Text*, etc.), foi necessário implementar a interface **Writable**, que permite personalizar a estrutura de dados com dois atributos: o cromossomo do indivíduo, representado por uma cadeia de caracteres (*String*), e sua aptidão, que é um número real (*double*). O termo “cromossomo” foi usado porque no linguajar dos algoritmos genéticos cromossomo é a própria representação do indivíduo; neste caso, a cadeia de caracteres que especifica o caminho das cidades percorridas. Já a aptidão é o resultado do cálculo da distância total desse caminho. Em resumo, os objetos instanciados dessa classe são usados em todo o processamento do MapReduce.



**Figura 7.** Criação de um projeto MapReduce

**Nota**

O Hadoop disponibiliza a interface **Writable**, solução que permite criar objetos para o processamento distribuído do MapReduce. Para isso, basta criar uma classe que a implemente. Tal interface especifica dois métodos que devem ser construídos: o *write()*, para gravar os objetos na saída (*DataOutput*), que normalmente é um arquivo texto; e *readFields()*, para a leitura dos objetos obtidos no fluxo de entrada (*DataInput*), que normalmente são obtidos de um arquivo texto. No projeto deste artigo, a classe **Individuo** implementa **Writable**, mas na maioria das vezes, customizar a interface **Writable** é desnecessário, pois o Hadoop disponibiliza estruturas de dados prontas para objetos em forma de arrays, mapas e todos os tipos primitivos da linguagem Java (com exceção ao char).

## Listagem 1.

Código da classe Individuo.

```
// imports omitidos...

public class Individuo implements Writable {

    private String cromossomo;
    private double aptidao;

    @Override
    public void write(DataOutput out) throws IOException {
        out.writeUTF(cromossomo);
        out.writeDouble(aptidao);
    }

    @Override
    public void readFields(DataInput in) throws IOException {
        this.cromossomo = in.readUTF();
        this.aptidao = in.readDouble();
    }

    // gets e sets
}
```

## A classe de mapeamento CaixeiroViajanteMapper

O papel de mapeamento é realizado pela classe **CaixeiroViajanteMapper**, cujo código é apresentado na **Listagem 2**. Por ser uma extensão da classe genérica **Mapper**, é necessário declarar os tipos de dados que serão manipulados como chave e valor de entrada (neste caso, **Object** e **Text**) e os tipos (chave e valor) de saída (**IntWritable** e **Individuo**, respectivamente).

Para cumprir a função de mapeamento, **CaixeiroViajanteMapper** possui os atributos **populacao** e **operacaoGenetica** que são, respectivamente, uma lista do tipo **Individuo** e o objeto que encapsula todas as operações do algoritmo genético (cruzamento, mutação, etc.). Portanto, tais campos (atributos) são usados nas tarefas de criação dos grupos de indivíduos, no armazenamento do melhor indivíduo e para fornecer as operações ao processo AG.

### Nota

No Hadoop, a função map é representada por uma classe derivada da classe abstrata **Mapper**, um tipo genérico que permite tratar os dados definidos no par chave/valor. Mapper trabalha com quatro parâmetros que especificam a chave de entrada, o valor de entrada, a chave de saída e o valor de saída. Além disso, essa classe declara quatro métodos (**setup()**, **map()**, **cleanup()** e **run()**) que podem ser usados pela aplicação do usuário. Dentre eles, **map()** é o único método que é obrigatório implementar. Na ordem de chamadas, o primeiro método a ser executado é o **setup()**, que realiza algum procedimento de inicialização da tarefa (job) de mapeamento. O método **run()** permite que o programador altere alguns dos parâmetros de configuração do job, que normalmente são especificados na classe principal da aplicação. O método **map()**, por sua vez, realiza a lógica de processamento do mapeamento. Por fim, o método **cleanup()** pode ser implementado para executar algum código no final da tarefa de mapeamento, por exemplo, fechamento de conexões com o banco de dados e reinitialização de variáveis.

A classe **CaixeiroViajanteMapper** ainda possui dois métodos herdados da classe **Mapper**: **map()** e **cleanup()**, e um método auxiliar, denominado **geracaoLocal()**. O **map()** é o método principal da classe e trabalha com três parâmetros: **Object key**, **Text value**,

**Context context**. Os dois primeiros são correspondentes ao par chave/valor de entrada. Já o último parâmetro, o objeto **Context**, representa o objeto que permite que a aplicação se comunique e envie dados para o framework Hadoop a serem processados na fase reduce.

## Listagem 2.

Código da função map – classe CaixeiroViajanteMapper.

```
// imports omitidos...

public class CaixeiroViajanteMapper extends
    Mapper<Object, Text, IntWritable, Individuo> {

    private ArrayList<Individuo> populacao = new ArrayList<Individuo>();
    private OperacaoGenetica operacaoGenetica = new OperacaoGenetica();

    @Override
    public void map(Object key, Text value, Context context)
        throws IOException, InterruptedException {
        ArrayList<Individuo> popula = new ArrayList<Individuo>();
        Individuo individuo = new Individuo();
        individuo.setCromossomo(value.toString());

        popula.add(individuo);
        popula = operacaoGenetica.calcularAptidao(popula);
        populacao.add(popula.get(0));
        operacaoGenetica.setTaxaMutacao(0.05);
        if (populacao.size() == 100) {
            geracaoLocal(context);
        }
    }

    public void geracaoLocal(Context context) throws IOException,
        InterruptedException {
        int CondicaoDeParada = 10;
        int geracao = 0;
        while (CondicaoDeParada != geracao) {
            populacao = operacaoGenetica.selecao(populacao);
            populacao = operacaoGenetica.cruzamento(populacao);
            populacao = operacaoGenetica.mutacao(populacao);
            populacao = operacaoGenetica.calcularAptidao(populacao);
            geracao++;
        }
        populacao.add(operacaoGenetica.getMelhorIndividuo());
        int Aptidao;
        for (Individuo individuo : populacao) {
            Aptidao = (int) Math.round(individuo.getAptidao());
            context.write(new IntWritable(Aptidao), individuo);
        }
        populacao.clear();
    }

    @Override
    protected void cleanup(Context context) throws IOException,
        InterruptedException {
        geracaoLocal(context);
    }
}
```

O método **geracaoLocal()** foi criado para modularizar o processo AG, na função de mapeamento. Este método é chamado para atuar em um conjunto de 100 indivíduos, onde são aplicadas as operações genéticas de seleção, cruzamento e mutação sobre 5% da população. Essas operações são executadas 10 vezes (gerações), em um laço **while** cuja condição de parada ocorre quando a variável **CondicaoDeParada** for igual a 10.

Após o laço, é acrescentada à população um indivíduo considerado o melhor, na chamada operação de *elitismo*. No fim, o método **geracaoLocal()** utiliza o objeto passado no argumento **Context** para produzir a população de indivíduos para as fases seguintes (intermediária e redução), via instrução **context.write(...)**, selecionados pelo critério de aptidão (menor caminho).

Concluindo, o método **cleanup()** foi implementado para ser chamado na finalização da tarefa de mapeamento. Neste caso, irá executar mais uma etapa do AG, por meio da chamada ao método **geracaoLocal()**, apenas para garantir que o AG atuará nos indivíduos que não entraram no último conjunto de cem indivíduos. Lembre-se que cada conjunto é selecionado para ser tratado pelos operadores AG no método **map()**, na condição representada pelo comando **if (populacao.size() == 100)**.

### A classe de redução CaixeiroViajanteReducer

A classe **CaixeiroViajanteReducer**, apresentada na **Listagem 3**, representa a função de redução. Para isso, ela deve estender **Reducer**, uma classe genérica que exige a definição dos tipos (classes) para os dados de entrada e saída do processo de redução. Neste caso, os tipos de entrada para o par chave/valor são, respectivamente, as classes **IntWritable** e **Individuo**, que formam a estrutura para os dados gerados no processo de mapeamento, representando, respectivamente, a distância (em inteiro) e o indivíduo (que representa o caminho e aptidão). Para os tipos da saída, as classes **DoubleWritable** e **Text** definem, respectivamente, para o par chave/valor, que a chave terá um formato decimal (**Double**) para representar a distância e o valor terá uma cadeia de caracteres (**Text**) para identificar o caminho (de cidades) para aquela distância. Outros detalhes da classe, como os atributos **populacao** e **operacaoGenetica**, seguem as mesmas explicações dadas na classe **CaixeiroViajanteMapper**.

#### Nota

A função **reduce** é representada por uma classe derivada da classe abstrata **Reducer**, um tipo genérico semelhante ao **Mapper**. O **Reducer** possui quatro parâmetros que são usados para especificar os tipos (classe) de objetos manipulados na função de redução. Em essência, eles definem dois pares de objetos chave/valor, sendo um par para entrada e outro para saída. Da mesma forma que **Mapper**, a classe **Reducer** fornece quatro métodos (**setup()**, **reduce()**, **cleanup()** e **run()**) que podem ser implementados na classe de redução. Na prática, apenas a implementação do método **reduce()** é obrigatória, pois é ele quem realiza a função de redução. Para os outros métodos (**setup()**, **cleanup()** e **run()**), o framework os trata da mesma maneira como na classe **Mapper**.

O método **reduce()** trabalha com três parâmetros. Os dois primeiros (**key** e **values**) são referentes ao par chave/valor de entrada, que correspondem aos tipos de saída da fase de mapeamento (um inteiro representando a aptidão e uma lista de indivíduos que pertencem àquela aptidão). O último parâmetro (o objeto **context**) é responsável pelo resultado final da tarefa de redução, gerando os dados de saída.

No trecho **operacaoGenetica.setTaxaMutacao(0.9)**, a taxa de mutação é definida em 90%, mais alta que no mapeamento (de apenas 5%),

pois cada tarefa reduce recebe indivíduos de aptidão muito próximos, aumentando assim a possibilidade de indivíduos semelhantes. Essa situação (de indivíduos com aptidões próximas) pode elevar o risco de convergência prematura, um fenômeno presente na técnica dos algoritmos genéticos que produz uma solução precoce, distante da solução ótima para o problema. Esse problema pode ser resolvido aumentando a diversidade da população, aplicando-se o operador de mutação com uma taxa elevada.

#### Listagem 3. Código da função de redução – classe CaixeiroViajanteReducer.

```
// imports omitidos...

public class CaixeiroViajanteReducer extends
    Reducer<IntWritable, Individuo, DoubleWritable, Text> {

    private ArrayList<Individuo> populacao = new ArrayList<Individuo>();
    private OperacaoGenetica operacaoGenetica;

    @Override
    public void reduce(IntWritable key, Iterable<Individuo> values,
        Context context) throws IOException, InterruptedException {
        operacaoGenetica = new OperacaoGenetica();
        operacaoGenetica.setTaxaMutacao(0.9);
        for (Individuo val : values) {populacao.add(val); }
        if (populacao.size() == 1) {
            operacaoGenetica.setMelhorIndividuo(populacao.get(0));
        }

        int CondicaoDeParada = 10;
        int geracao = 0;
        while (geracao < CondicaoDeParada) {
            populacao = operacaoGenetica.selecao(populacao);
            populacao = operacaoGenetica.cruzamento(populacao);
            populacao = operacaoGenetica.mutacao(populacao);
            populacao = operacaoGenetica.calcularAptidao(populacao);
            geracao++;
        }

        Individuo ind = operacaoGenetica.getMelhorIndividuo();
        context.write(new DoubleWritable(ind.getAptidao()),
            new Text(ind.getCromossomo()));
        populacao.clear();
    }
}
```

Em resumo, a finalidade do método **reduce()** é recuperar a lista de indivíduos de entrada e processar o AG nessa população. O AG é executado pelo atributo **operacaoGenetica**, que realiza dez vezes (10 gerações) as operações genéticas de seleção, cruzamento e mutação, além de calcular a aptidão para cada indivíduo da população. Ao final, acrescenta-se à população já processada o melhor indivíduo, pela chamada ao método **operacaoGenetica.getMelhorIndividuo()**, tratamento esse chamado de elitismo. Finalizando, o método **reduce()** gera na saída a população contendo os melhores indivíduos, gravando os dados na pasta **/Output** (como veremos a seguir) pela chamada ao método **write()** do objeto **context**.

### A classe da lógica AG: OperacaoGenetica

A classe **OperacaoGenetica** encontra-se toda a inteligência do algoritmo genético. Esta disponibiliza todas as operações previstas

na técnica dos AGs, que são: seleção, cruzamento, mutação, cálculo de aptidão e elitismo. Tais operações são traduzidas em métodos da classe e são explicados a seguir.

Na **Listagem 4** vê-se a definição dos atributos da classe **OperacaoGenetica**: **melhorIndividuo**, **tamanhoIndividuo**, **taxaMutacao**, **alfabeto** e **distancias**. O atributo **melhorIndividuo** tem o objetivo de guardar o melhor indivíduo encontrado durante o processo AG. O atributo **tamanhoIndividuo**, por sua vez, define o tamanho da cadeia de cidades (neste caso, 20). Já o atributo **alfabeto** representa as vinte cidades, onde cada cidade é representada por uma letra do alfabeto, enquanto **distancias** representa uma matriz contendo as coordenadas de cada cidade (letra) no plano cartesiano. Por fim, o atributo **taxaMutacao** define a taxa (5%) que será aplicada na operação de mutação da população.

**Listagem 4.** Código resumido da classe auxiliar OperacaoGenetica.

```
// imports omitidos...

public class OperacaoGenetica {
    private Individuo melhorIndividuo = new Individuo();
    private int tamanhoIndividuo = 20;
    private double taxaMutacao = 0.05;
    private String alfabeto = "ABCDEFGHIJKLMNPQRSTUVWXYZ";
    private int[][] distancias = {
        {1,8},{1,10},{2,13},{3,15},{5,16},
        {8,17},{10,17},{13,16},{15,15},{16,13},
        {17,10},{17,8},{16,5},{15,3},{13,2},
        {10,1},{8,1},{5,2},{3,3},{2,5}};
}

public ArrayList<Individuo> calcularAptidao(ArrayList<Individuo> populacao) {...};
public ArrayList<Individuo> selecao(ArrayList<Individuo> populacao) {...};
private Individuo[] quickSort(Individuo[] pop, int inicio, int fim) {...};
public ArrayList<Individuo> cruzamento(ArrayList<Individuo> populacaoA) {...};

public ArrayList<Individuo> mutacao(ArrayList<Individuo> populacao) {...};
public Individuo getMelhorIndividuo() {...}
public void setMelhorIndividuo(Individuo ind) {...}
public double getTaxaMutacao() {...}
public void setTaxaMutacao(double taxa) {...}
}
```

Como o próprio nome indica, o método **calcularAptidao()**, visto na **Listagem 5**, serve para calcular a aptidão da função objetivo do problema em questão, que se resume em determinar a distância a ser percorrida na roda de navegação entre as cidades especificadas em cada indivíduo. Para isso, o método localiza as coordenadas das cidades (no atributo **alfabeto**) para determinar o valor do atributo **distancias**. Em seguida, aplica o cálculo da distância euclidiana e acumula o valor na variável **aptidao**, registrando o resultado no atributo correspondente daquele indivíduo (usando o método **setAptidao()**). Depois é chamado o método **setMelhorIndividuo()** para identificar o melhor indivíduo (com a menor distância), para ser utilizado na operação de elitismo. No final, o método **calcularAptidao()** retorna a lista (população) de indivíduos e suas aptidões calculadas.

Já o método **selecao()**, cujo código é apresentado na **Listagem 6**, realiza a operação de seleção dos indivíduos mais aptos. Para escolher os melhores, uma chamada ao método de classificação

**quickSort()** garante uma lista de indivíduos ordenada de forma descendente, do que possui a maior aptidão para o de menor aptidão. Com o objetivo de reduzir a quantidade de indivíduos menos aptos que podem estar presentes na população, uma quantidade dos melhores indivíduos é novamente adicionada à população. Isso é feito limpando a lista, via **populacao.clear()**, e adicionando a ela os 70% dos melhores indivíduos do ranking de classificação e depois outros 30% também dos melhores indivíduos. Mesmo duplicados, tais indivíduos sofrerão um processo de mutação na etapa seguinte. Dessa forma garantimos o retorno de uma nova população com os indivíduos de maior aptidão.

Na operação genética de seleção é necessário utilizar algum método de ranqueamento para a escolha dos indivíduos mais aptos. Assim, foi implementado o método **quickSort()**, exposto na **Listagem 7** e que realiza a ordenação de uma lista de indivíduos (população), para ser usado no método **selecao()**.

**Listagem 5.** Código do método calcularAptidao().

```
public ArrayList<Individuo> calcularAptidao(ArrayList<Individuo> populacao)
    throws IOException {
    ArrayList<Individuo> popula = new ArrayList<Individuo>();
    double aptidao = 0;
    int x1, x2, y1, y2, pos;
    String[] indString;
    for (Individuo indi : populacao) {
        indString = new String[tamanhoIndividuo];
        indString = indi.getIndArray();
        for (int j = 0; j < tamanhoIndividuo; j++) {
            pos = alfabeto.indexOf(indString[j]);
            x1 = distancias[pos][0];
            y1 = distancias[pos][1];
            pos = alfabeto.indexOf(indString[(j + 1)%tamanhoIndividuo]);
            x2 = distancias[pos][0];
            y2 = distancias[pos][1];
            aptidao += Math.sqrt(Math.pow(x1 - x2, 2)
                + Math.pow(y1 - y2, 2));
        }
        indi.setAptidao(aptidao);
        setMelhorIndividuo(indi);
        popula.add(indi);
        aptidao = 0;
    }
    return popula;
}
```

**Listagem 6.** Código do método selecao().

```
public ArrayList<Individuo> selecao(ArrayList<Individuo> populacao) throws
IOException {
    int tamanhoPop = populacao.size();
    Individuo[] popula = new Individuo[tamanhoPop];
    popula = populacao.toArray(popula);
    popula = quickSort(popula, 0, tamanhoPop - 1);
    populacao.clear();

    for(int i=0; i < (popula.length*(0.7)); i++){
        populacao.add(popula[i]);
    }
    for(int i=0; i < (popula.length*0.3); i++){
        populacao.add(popula[i]);
    }
    return populacao;
}
```

Para efetuar essa ordenação, o método divide a lista em segmentos, seleciona um indivíduo de referência (variável **pivo**) e depois o compara a todos os demais. Os mais aptos são deslocados para o segmento inicial da lista, os menos aptos para o fim e entre ambos (no meio da lista) fica o segmento do **pivo**. De forma recursiva, cada segmento realiza o mesmo procedimento chamando o método **quickSort()** até que não haja mais segmento para classificar.

#### Listagem 7. Código do método quickSort().

```
private Individuo[] quickSort(Individuo[] pop, int inicio, int fim) {
    int meio;
    if (inicio < fim) {
        int topo, indice;
        Individuo pivo;
        pivo = pop[inicio];
        topo = inicio;
        for (indice = inicio + 1; indice <= fim; indice++) {
            if (pop[indice].getAptidao() < pivo.getAptidao()) {
                pop[topo] = pop[indice];
                pop[indice] = pop[topo + 1];
                topo++;
            }
        }
        pop[topo] = pivo;
        meio = topo;
        quickSort(pop, inicio, meio);
        quickSort(pop, meio + 1, fim);
    }
    return pop;
}
```

Ao fim do processamento, é retornada uma lista com os indivíduos ordenados do mais apto ao menos apto.

A **Listagem 8** mostra o método da operação genética de cruzamento. Este visa produzir novos indivíduos misturando características de dois indivíduos “pais”. Para efetuar o cruzamento são utilizados dois pontos de corte que atuam em posições fixas da representação do indivíduo (variáveis **pp** e **sp**). Essas posições são determinadas pelo valor do atributo **tamanhoIndividuo**, que estabelece o tamanho da seção de corte. Em seguida, um par de indivíduos “pai” é selecionado a partir de dois indivíduos consecutivos na população (nas variáveis **indPai1** e **indPai2**) e suas seções de corte são utilizadas para compor os novos indivíduos (variáveis **indFilho1** e **indFilho2**). Feito isso, cada filho criado preserva a seção central de seu indivíduo pai, mas todos os outros dados são trocados entre os progenitores, gerando filhos com características distintas. Essa explicação é apresentada de forma gráfica na **Figura 8**.

Dessa forma, os novos indivíduos gerados no processo de cruzamento formarão uma nova população, construída e evoluída a partir da população anterior.

O código exposto na **Listagem 9** trata do método que realiza a operação genética de mutação, necessária para a introdução e manutenção da diversidade genética da população. Antes de realizar essa operação é verificado se a população é de um único indivíduo, fato que estabelece a taxa de mutação em 100%. Caso contrário, essa taxa é previamente configurada no atributo **taxa-Mutacao** para cada função com seus valores *default*, que são de 5% no mapeamento e 90% na redução.

#### Listagem 8. Código do método cruzamento().

```
public ArrayList<Individuo> cruzamento(ArrayList<Individuo> populacaoA) throws
IOException {
    int tamanhoPop = populacaoA.size();
    Individuo[] populacao = new Individuo[tamanhoPop];
    populacao = populacaoA.toArray(populacao);
    populacaoA.clear();
    String[] indPai1, indPai2, indFilho1, indFilho2;
    int pp = 3, sp = 7, k, c1, c2;
    int tamanhoSecao = (int) tamanhoIndividuo / 2;
    Random r = new Random();
    boolean presente;
    for (int i = 0; i < tamanhoPop; i += 2) {
        indFilho1 = new String[tamanhoIndividuo];
        indFilho2 = new String[tamanhoIndividuo];
        pp = r.nextInt(tamanhoSecao - 1); // ponto de corte
        sp = pp + tamanhoSecao; // ponto de corte
        if (i + 1 < tamanhoPop) {
            indPai1 = populacao[i].getIndArray();
            indPai2 = populacao[i + 1].getIndArray();
            for (int c = pp; c <= sp; c++) {
                indFilho1[c] = indPai1[c];
                indFilho2[c] = indPai2[c];
            }
            c1 = sp + 1;
            c2 = sp + 1;
            for (int j = sp + 1; j != pp; j = (j + 1) % tamanhoIndividuo) {
                do {
                    presente = false;
                    for (k = pp; k <= sp; k++) {

```

```
                        presente |= indPai2[c2].equals(indFilho1[k]);
                    }
                    if (!presente) {
                        indFilho1[j] = indPai2[c2];
                    }
                    c2 = (c2 + 1) % tamanhoIndividuo;
                } while (presente);
                do {
                    presente = false;
                    for (k = pp; k <= sp; k++) {
                        presente |= indPai1[c1].equals(indFilho2[k]);
                    }
                    if (!presente) {
                        indFilho2[j] = indPai1[c1];
                    }
                    c1 = (c1 + 1) % tamanhoIndividuo;
                } while (presente);
            }
            Individuo indF1 = new Individuo();
            Individuo indF2 = new Individuo();
            indF1.setIndArray(indFilho1);
            indF2.setIndArray(indFilho2);
            populacaoA.add(indF1);
            populacaoA.add(indF2);
        } else {
            populacaoA.add(populacao[i]);
        }
    }
    return populacaoA;
}
```

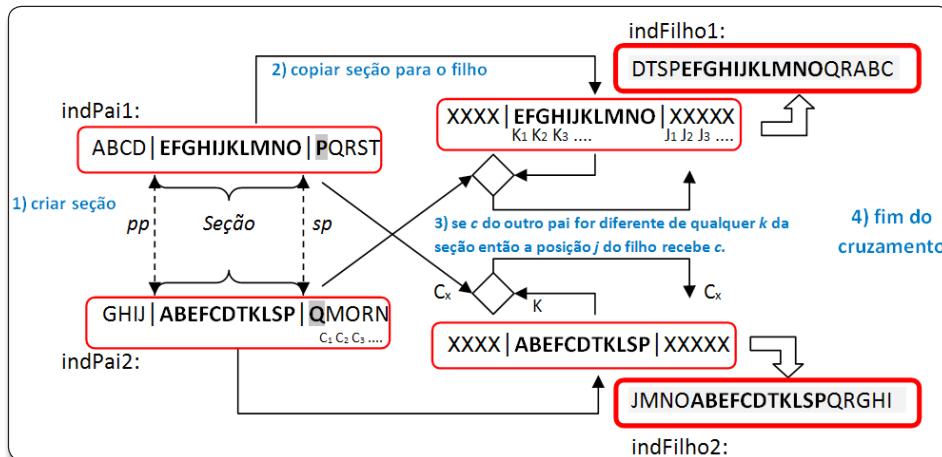


Figura 8. Funcionamento do operador de cruzamento

Listagem 9. Código do método mutacao().

```
public ArrayList<Individuo> mutacao(ArrayList<Individuo> populacao) throws
IOException {
    int tamanhoPop = populacao.size();
    String cidade;
    String[] indString;
    Individuo indi;
    Random indRandom = new Random();
    int ri, rc1, rc2;
    if(tamanhoPop == 1){
        taxaMutacao = 1;
    }

    for (int i = 0; i < tamanhoPop * taxaMutacao; i++) {

        ri = indRandom.nextInt(tamanhoPop);
        indi = populacao.get(ri);
        indString = indi.getIndArray();
        rc1 = indRandom.nextInt(tamanhoIndividuo);
        rc2 = indRandom.nextInt(tamanhoIndividuo);
        cidade = indString[rc1];
        indString[rc1] = indString[rc2];
        indString[rc2] = cidade;
        indi.setIndArray(indString);
        populacao.set(ri, indi);
    }
    return populacao;
}
```

Em seguida, com um subconjunto da população (gerado a partir da **taxaMutacao**), um indivíduo é escolhido aleatoriamente (variável **indi**, na posição **ri** da lista). Desse indivíduo são selecionadas duas cidades, cujas posições na cadeia que formam o indivíduo também são escolhidas de forma aleatória (e atribuídas nas variáveis **rc1** e **rc2**). Em seguida, estas cidades são permutadas entre si. O resultado desse processo é a substituição do indivíduo original da população pelo “mutante” (**indi**) via instrução **populacao.set(ri, indi)**. Ao final de sua execução, o método **mutacao()** retorna uma população com alguns indivíduos modificados.

O método **setMelhorIndividuo()** cumpre o papel da operação genética de *elitismo*. Seu código (visto na Listagem 10) realiza

a verificação do indivíduo mais apto, lembrando que a melhor aptidão significa um indivíduo com um valor de distância menor para o caminho entre as cidades. Portanto, para cada indivíduo submetido para o método (pelo parâmetro **ind**), é verificado se ele tem aptidão menor que a aptidão do **melhorIndividuo**, atributo que é definido no método **calcularAptidao()**. Caso verdadeiro, o melhor indivíduo (presente na variável **melhorInd**) passa a ser o indivíduo presente no parâmetro **ind**.

Por fim, os métodos **getTaxaMutacao()** e **setTaxaMutacao()**, mostrados na Listagem 11, são, por padrão, os métodos de acesso ao atributo **taxaMutacao**.

Listagem 10. Código dos métodos **getMelhorIndividuo()** e **setMelhorIndividuo()**.

```
public Individuo getMelhorIndividuo() {
    return melhorIndividuo;
}

public void setMelhorIndividuo(Individuo ind) {
    if (getMelhorIndividuo().getAptidao() == 0
        || getMelhorIndividuo().getAptidao() > ind.getAptidao()) {
        Individuo melhorInd = new Individuo();
        melhorInd.setCromossomo(ind.getCromossomo());
        melhorInd.setAptidao(ind.getAptidao());
        this.melhorIndividuo = melhorInd;
    }
}
```

Listagem 11. Código dos métodos **getTaxaMutacao()** e **setTaxaMutacao()**.

```
public double getTaxaMutacao(){
    return this.taxaMutacao;
}

public void setTaxaMutacao(double taxa){
    this.taxaMutacao = taxa;
}
```

## Classe principal da aplicação: CaixeiroViajante

A classe que é considerada a principal da aplicação chama-se **CaixeiroViajante** e é apresentada na Listagem 12. Ela representa a aplicação propriamente dita, isto é, é a classe que tem o papel de executar o MapReduce. Para isso, a partir do método estático **main()**, toda a configuração para o controle da execução das tarefas é declarada. Por exemplo, as definições das pastas dos dados de entrada e saída são informadas às classes **FileInputFormat** e **FileOutputFormat**. Também são informadas quais classes executarão as funções map e reduce, o que é feito via o objeto **job** e as chamadas aos métodos **setMapperClass()** e **setReducerClass()**. Além disso, são declarados os tipos de dados dos pares chave/valor de entrada e saída, obrigatórios nas classes que representam as funções map e reduce. No final do método **main()**, há

ainda o monitoramento do progresso de execução e finalização da aplicação MapReduce, o que é realizado pelo método `job.waitForCompletion()`.

**Listagem 12.** Código da classe CaixeiroViajante.

```
// imports omitidos...

public class CaixeiroViajante {

    public static void main(String[] args) throws Exception {
        if (args.length != 2) {
            args = new String[2];
            args[0] = System.getProperty("user.dir") + "/Input";
            args[1] = System.getProperty("user.dir") + "/Output";
        }
        Job job = new Job();
        job.setJarByClass(CaixeiroViajante.class);
        job.setJobName("Problema do Caixeiro Viajante");
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        job.setMapperClass(CaixeiroViajanteMapper.class);
        job.setReducerClass(CaixeiroViajanteReducer.class);
        job.setMapOutputKeyClass(IntWritable.class);
        job.setMapOutputValueClass(Individuo.class);
        job.setOutputKeyClass(DoubleWritable.class);
        job.setOutputValueClass(Text.class);
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

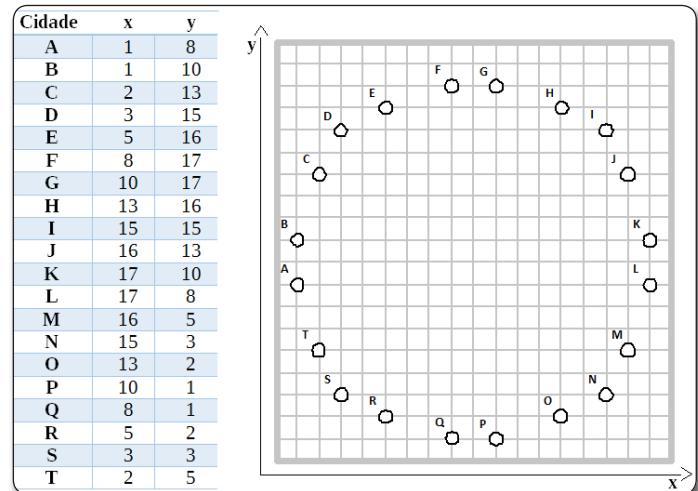
## Teste da aplicação MapReduce no Eclipse

Para testar a aplicação foi utilizada uma configuração com vinte cidades, que combinadas geram um espaço de busca maior que 60 quatrilhões de possibilidades de caminhos. Tal cenário permite validar a aplicação em uma situação de maior complexidade, exigindo um número elevado de indivíduos na população inicial.

A aplicação foi executada em um só computador, no modo local (*standalone mode*) do Hadoop, que é o mais indicado para o desenvolvimento de aplicações MapReduce, porém não fornece todo o poder de processamento paralelo presente no modo distribuído em uma rede de computadores. Portanto, a avaliação da aplicação está focada apenas na capacidade de resolver o problema usando a modelagem MapReduce, não considerando os aspectos de desempenho em tempo de resposta ou no poder de processamento paralelo, pois o ambiente em modo local do Hadoop simula o paralelismo em uma única máquina.

As coordenadas cartesianas das vinte cidades formam graficamente uma imagem circular, como é apresentado na **Figura 9**. Esse arranjo circular para o caminho entre as cidades foi escondido pelo fato de ser possível reconhecer visualmente o melhor caminho, permitindo assim uma análise mais simplificada sobre os indivíduos (caminhos e aptidões) encontrados pela aplicação. As aptidões (distâncias) geradas pela aplicação podem então ser comparadas com o resultado conhecido pelo caminho circular indicado na figura. Tal resultado é representado pela ordem da

seguinte sequência de pontos: "ABCDEFGHIJKLMNPQRST", que gera uma distância (melhor aptidão) próxima de 51,187 (caminho ótimo, menor distância). Apesar desse caminho iniciar em "A", o resultado da melhor aptidão vale também para qualquer outro ponto que respeite a sequência alfabética, por exemplo: "BCDEFGHIJKLMNOPQRSTA".



**Figura 9.** Coordenadas relativas das cidades aplicadas no teste e o mapa no plano euclidiano

Para realizar os testes, foram usadas cinco bases de dados de entrada, representando cinco escalas de população: a primeira contendo mil linhas de indivíduos e as outras com 10, 20, 30 e 40 mil indivíduos. A expectativa é demonstrar o efeito do número menor ou maior de indivíduos na população inicial (base de entrada) nos resultados gerados pelo AG. Após executar a aplicação para cada conjunto de dados de entrada, algumas dezenas de indivíduos foram gerados na saída, da qual apenas o melhor indivíduo, para cada cenário do conjunto de entrada, corresponde à melhor resposta processada pelo AG, aproximando-se do valor da solução ótima.

O resultado obtido com as cinco populações iniciais é apresentado no gráfico da **Figura 10**. Percebe-se no gráfico que uma população inicial com uma quantidade maior de indivíduos gera as melhores soluções, que no exemplo representa o indivíduo com menor tamanho de caminho (cuja aptidão é a distância próxima de 51). Esta afirmação pode ser observada pela linha da média gerada e também pela tendência das linhas do melhor e pior indivíduo gerados. Se aumentássemos ainda mais o número de indivíduos da população de entrada (inicial), a tendência seria a convergência para valores próximos do melhor caminho, uma característica inerente dos algoritmos genéticos.

Outra forma de observar o resultado do melhor e o pior indivíduo encontrados pode ser vista na **Figura 11**. O melhor indivíduo é o caminho representado pela sequência "ONMKLJHIGFEDC-BATSRQP", com aptidão aproximada de 59.2897, e o pior indivíduo é o caminho "JIHSCFEDBARTQPONKLMG", com aptidão aproximada de 110.0509.

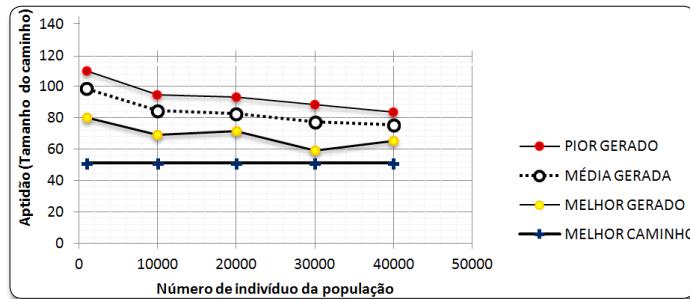


Figura 10. Resultados obtidos e comparação com o menor caminho (melhor aptidão)

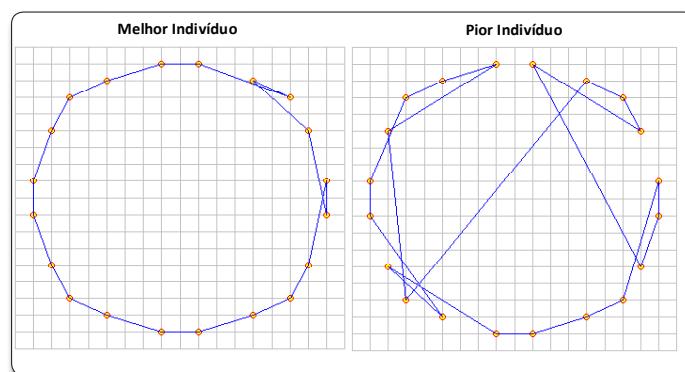


Figura 11. Melhor e pior indivíduo encontrados

Para que você possa reproduzir esses testes, as cinco bases de dados estão disponíveis para download (veja a seção Links). Cada base é um arquivo contendo um conjunto de linhas de texto, onde cada linha representa um indivíduo, ou seja, uma sequência não repetida de cidades (letras).

Há muitos exemplos de casos reais do uso de AGs que podem ser adaptados ao MapReduce. Entre eles, podemos citar os sistemas de classificação de imagens de satélites, utilizados para identificar áreas florestais, queimadas, etc. Normalmente, aplicações dessa natureza trabalham com bancos de imagens que manipulam bilhões de dados em forma de pixels.

Além desse estudo, que mostrou a viabilidade do uso de MapReduce no processamento de um algoritmo genético, muitas outras aplicações podem se beneficiar dessa técnica. Em destaque, aplicações que envolvam carga e tratamento de um grande volume de dados, que manipulem fontes com formatos diversificados e não estruturados, ou necessitem de eficiência de processamento em larga escala, em situações que a solução tradicional não resolve.

## Autor



Cláudio Martins  
claudiomartins2000@gmail.com



Mestre em Computação pela Universidade Federal do Rio Grande do Sul (UFRGS), professor do Instituto Federal do Pará (IFPA) e analista de sistemas da Companhia de Informática de Belém (Cinbesa). Trabalha há dez anos com a plataforma Java.

## Autor



Wesley Louzeiro Mota  
wesleylouzeiro@hotmail.com



É graduado em Tecnologia em Análise e Desenvolvimento de Sistemas pelo IFPA, em 2015. Possui experiência em programação Java desde 2012. Seu trabalho de conclusão de curso abordou o uso combinado de MapReduce e algoritmo genético na solução do Problema do Caixeiro Viajante.

## Links:

Página oficial do projeto Apache Hadoop.

<http://hadoop.apache.org/>

Apostila “Algoritmos Genéticos: uma Introdução”, de Diogo C. Lucas e Luiz O. Alvares, Inf/UFRGS, 2002.

<http://www.inf.ufrgs.br/~alvares/INF01048IA/ApostilaAlgoritmosGeneticos.pdf>

Link reduzido para download do plugin para uso do Hadoop no Eclipse.

<http://bit.ly/1dTezxt>

Link reduzido para download do projeto da aplicação (Eclipse) e os arquivos de entrada com as populações iniciais.

<http://bit.ly/1Hi0J4e>

## Livros

Hadoop: The Definitive Guide - 3rd Edition. Tom White. O'Reilly, 2012.

Livro que aborda o Hadoop de forma didática e atualizada em sua atual versão (2.x), apresentando estudos de caso usados para resolver problemas no modelo mapreduce.

Sistemas inteligentes: fundamentos e aplicações. Solange Oliveira Rezende. Manole Ltda, 2003.

Traz os avanços de metodologias e técnicas utilizadas no desenvolvimento de Sistemas Inteligentes, mostrando os princípios e aplicações práticas das técnicas.

## Você gostou deste artigo?

Dê seu voto em [www.devmedia.com.br/javamagazine/feedback](http://www.devmedia.com.br/javamagazine/feedback)

Ajude-nos a manter a qualidade da revista!



# 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.

## Supporte 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](http://porta80.com.br) | [comercial@porta80.com.br](mailto:comercial@porta80.com.br) | [twitter.com/porta80](http://twitter.com/porta80)

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486

**“Estudando Sistemas de Informação na Metodista eu me sinto desafiado a desenvolver minha capacidade dentro da sala de aula e no mercado de trabalho.”**

Com os cursos da área de Exatas e Tecnologia da Metodista, o Sérgio tem contato com laboratórios e ferramentas de última geração. Além disso, desenvolve seu lado profissional por meio de Projetos de Ação Profissional, simulando na prática a resolução de problemas reais.

Sérgio  
Aluno de Sistemas de Informação



#### CONHEÇA OS CURSOS DA ÁREA DE EXATAS E TECNOLOGIA DA METODISTA:

- Análise e Desenvolvimento de Sistemas (Presencial e EAD)
- Automação Industrial (Presencial)
- Engenharia Ambiental e Sanitária (Presencial)
- Engenharia de Produção (Presencial)
- Gestão da Tecnologia da Informação (Presencial)
- Sistemas de Informação (Presencial)



EDUCAÇÃO A DISTÂNCIA

*Transforme sua vida. Transforme a realidade.*

**[processoseletivo.metodista.br](http://processoseletivo.metodista.br)**

QUALIDADE | INOVAÇÃO | DESENVOLVIMENTO REGIONAL | INTERNACIONALIZAÇÃO



INOVAÇÃO DESDE 1938