UNIDADE 4 – CONSTRUINDO UM *SOFTWARE* WEB
MÓDULO 1 – API JDBC (JAVA DATABASE CONNECTIVITY)

01

1- A API JDBC (JAVA DATABASE CONNECTIVITY)

As aplicações, em geral, necessitam gravar os seus dados e informações de modo permanente. Atualmente, uma das ferramentas mais usadas para armazenamento são os chamados Sistemas Gerenciadores de Banco de Dados (SGBD). Para conectar-se a um banco de dados poderíamos abrir *sockets* diretamente com o servidor que o hospeda, por exemplo, um Oracle ou MySQL e nos comunicarmos com ele através de seu protocolo proprietário. Mas conhecer um protocolo proprietário não é uma tarefa simples. A complexidade envolvida neste trabalho é árdua e muitos desistem pelo caminho.

Uma segunda ideia seria utilizar uma API (*Application Programming Interface*) específica para cada banco de dados. Antigamente, nas muitas linguagens como C, por exemplo, tinha uma única maneira de acessar o SGBD Oracle, que era por meio do uso de funções como oracle_connect, oracle_result, e assim por diante. Para acesso ao MySQL, uma outra API (similar) tinha suas funções análogas, como mysql_connect, mysql_result, dentre outras. Essa abordagem facilitou em muito o trabalho de conexão, uma vez que não se tornou mais necessário entender o protocolo proprietário de cada banco de dados.

Porém, essa abordagem faz com que seja necessário conhecer várias APIs, uma para cada SGBD, cada qual com inúmeras semelhanças e também algumas diferenças. Mas o trabalho maior não é o de conhecer as diversas APIs. O maior trabalho é o de, caso necessário, substituir o SGBD por outro, pois isso envolve trocar todas as funções de uma API específica, por outra. Ou seja, o código fonte deverá ser reescrito para refletir a nova situação de modo que a aplicação possa funcionar com o novo SGBD.

Para começar a usar a API JDBC, é necessário algum conhecimento acerca de banco de dados relacional, bem como sobre SQL, portanto, caso você não conheça esses assuntos, recomendamos que estude um pouco.

02

No caso do Java, para evitar que cada banco tenha a sua própria API e conjunto de classes e métodos, temos um único conjunto de interfaces muito bem definidas que devem ser implementadas. Esse conjunto de interfaces fica dentro do pacote "*java.sql*" e nos referiremos a ela como **JDBC**.

A **API JDBC** é um conjunto de recursos padronizados que prove um acesso universal aos dados armazenados em um banco de dados qualquer para aqueles aplicativos escritos em linguagem de programação Java.

Esse conjunto de interfaces padronizadas pela **API JDBC** necessita ser implementado em classes concretas cuja finalidade é a de prover uma ponte entre o código cliente que usa a **API JDBC** e o SGBD propriamente dito. São essas classes que sabem se comunicar através do protocolo proprietário do banco de dados. Esse conjunto de classes recebe o nome de ***driver***. Todos os principais bancos de dados do mercado possuem ***drivers* JDBC** para que você possa utilizá-los com Java. O nome driver é análogo ao que usamos para impressoras, por exemplo. Como é impossível que um sistema operacional saiba conversar com todo tipo de impressora existente, precisamos de um driver que faça o papel de "tradutor" dessa conversa.

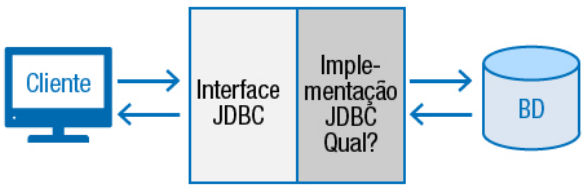
Os programas escritos em Java se comunicam com os bancos de dados para fins de manipulação de seu conteúdo por meio do *driver JDBC*. O *driver JDBC* é um componente de *software* que permite aos aplicativos Java realizarem diversos tipos de *queries* tais como: *select*, *insert*, *delete*, *update*, dentre outras. Na maioria das vezes, o *driver JDBC* é utilizado para permitir a conexão do aplicativo Java aos bancos de dados relacionais.

03

Atualmente, os mais populares *SGBD* (Sistemas Gerenciadores de Banco de Dados) fornecem *drivers JDBC* para permitir a comunicação com aplicativos escritos em Java. Isso significa que para cada *SGBD* diferente, existe um *driver* específico. Além disso, existem diversos *drivers JDBC* independentes disponíveis na web.

Um ponto importante que se deve observar é que a *API JDBC* e o *driver JDBC* são elementos separados e complementares. A *API JDBC* apenas especifica e padroniza as interfaces de comunicação. Já o *driver JDBC* especifica a implementação dessas interfaces. A principal razão para essa separação é permitir que seja possível a troca de um determinado SGBD por outro, caso seja necessário, lembrando que a referida troca não deve, pelo menos em teoria, propagar alterações ao seu aplicativo web. Ou seja, a *API JDBC* tem como objetivo desacoplar o seu aplicativo web do *SGBD* em uso.

A figura abaixo mostra essa separação.



O processo de armazenamento de dados é também chamado de **persistência**.

04

Sob uma dimensão, com o passar do tempo, foi se percebendo que o acoplamento das aplicações escritas em linguagem de programação Java estava um pouco além da questão relativa a, simplesmente, conexão. As queries (selects, inserts, updates, deletes, dentre outras) escritas dentro do código ainda

impediam que os SGBDs pudessem ser trocados de um modo fácil pois a persistência dos dados criava uma dependência. Isso acontece porque os programadores procuram utilizar as extensões específicas (proprietárias) de cada SGBD, que vão além da especificação padrão, pois as mesmas facilitam/agilizam, e muito, o trabalho de desenvolvimento do *software*. É claro que, se o programador utiliza-se apenas o SQL ANSI, esse problema não aconteceria, ou pelo menos, seria reduzido a, praticamente, alguns pouquíssimos problemas pontuais e de soluções instantâneas. Mas, para muitos, usar apenas o padrão, é ter que reinventar a roda e aumentar os prazos de entrega do produto. Portanto, a API JDBC necessitava de auxílio para permitir de fato, um desacoplamento entre a aplicação e o SGBD.

Sob outra dimensão, a linguagem Java foi construída para oferecer suporte, principalmente, ao paradigma de programação orientada a objeto cujo fundamento se baseia na teoria dos grafos da matemática. Como os SGBDs mais usados no mundo inteiro são relacionais, ou seja, tem como fundamento a teoria dos conjuntos da matemática, criou-se uma dificuldade entre persistir objetos Java em bancos de dados relacionais.

Diante dessas duas dimensões, como tentativa de solucionar tais dificuldades, criou-se o conceito de **ORM (*Object Relational Mapping*)** que tem como propósito equalizar as diferenças existentes entre os dois fundamentos de modo que ambos possam trabalhar de modo conjunto para um objetivo comum.

No caso do Java, criou-se uma API denominada de **JPA** *(Java Persistence Application)* que especifica e padroniza interfaces para persistir objetos Java e banco de dados relacionais. A partir disso, tem-se diversos *frameworks* no mercado que implementam a referida API de modo que classes concretas possam realizar o trabalho de fato. Um desses frameworks, muito conhecido pela comunidade Java no mundo todo, é aquele denominado de **Hibernate**.

05

O JDBC suporta quatro **categorias de drivers**:

1) Driver de ponte JDBC-para-ODBC (tipo 1)
2) Driver parcialmente Java para a API Nativa (tipo 2)
3) Drivers-cliente para servidor Pure Java (tipo 3)
4) Drivers Pure Java (tipo 4)

A documentação da API JDBC, para o Java 7, pode ser acessada nos dois endereços abaixo:

http://docs.oracle.com/javase/7/docs/api/java/sql/package-summary.html

http://docs.oracle.com/javase/7/docs/api/javax/sql/package-summary.html

O acesso a documentação é fundamental para a plena compreensão da classe de exemplo denominada de **PersistenciaDados.java** que será mostrada logo adiante.

driver de ponte JDBC-para-ODBC

O **driver de ponte JDBC-para-ODBC** conecta programas Java a origens de dados Microsoft ODBC (Open Database Connectivity). A JDK inclui o driver de ponte nativamente (sun.jdbc.odbc.JdbcOdbcDriver). Esse driver somente deve ser utilizado como última alternativa ao suporte de conexão com qualquer banco de dados, uma vez que o mesmo introduz uma carga de processamento extra que poderá deixar o aplicativo lento.

driver parcialmente Java para a API nativa
O **driver parcialmente Java para a API nativa** permite que os aplicativos JDBC utilizem API's específicas de banco de dados, normalmente escritas em linguagem C ou C++. Essa integração entre o Java e o C é feito por meio da Java Native Interface (JNI). A JNI é uma ponte entre a JVM (Java Virtual Machine) e o código escrito e compilado em linguagem específica e dependente da plataforma. Por dependente de plataforma, entendam o acoplamento do código binário executável ao Sistema Operacional e ao set de instruções da CPU (Unidade Central de Processamento).

drivers-cliente para servidor Pure Java
Os **drivers-cliente para servidor Pure Java** aceitam as solicitações JDBC e as traduzem em um protocolo de rede que não é específico ao banco de dados. Essas solicitações são enviadas para um servidor que converte as solicitações de banco de dados em um protocolo especifico para o SGBD em questão.

drivers Pure Java
Os **drivers Pure Java** implementam protocolos de rede específicos ao banco de dados para que programas Java possam conectar-se diretamente a um banco de dados. Sempre que possível, o uso deste tipo de driver é altamente recomendado pelo fato de não introduzirem overhead à conexão.

06

2 - Instalando o Driver JDBC para o MySQL

A instalação consiste em copiar o driver para a pasta WEB-INF/lib do seu aplicativo web.

Para realizar o *download* do driver, acesse o endereço abaixo:

http://central.maven.org/maven2/mysql/mysql-connector-java/5.1.18/mysql-connector-java-5.1.18.jar

07

2.1 - Conectando com o MySQL

Para se conectar com um banco de dados são necessários dois **passos**:

1- registrar o driver JDBC;

2- utilizar o driver para abrir a conexão.

Para realizar o registro do driver é necessário conhecer o FQDN da classe principal do driver. No caso do SGBD MySQL, o valor é uma constante e dado pela String: “com.mysql.jdbc.Driver”. O registro é feito através da utilização do método *forname* da classe *Class*, conforme instrução abaixo:

```
Class.forname(“com.mysql.jdbc.Driver”).
```

Após o registro, o driver estará disponível para uso. A classe **DriverManager** é a responsável por utilizar o driver de conexão registado. Para isso, invocamos o método estático **getConnection** com uma String que indica a qual banco desejamos nos conectar.

Essa String - chamada de String de conexão JDBC - que utilizaremos para acessar o MySQL tem sempre a seguinte forma genérica:

```
jdbc:mysql://nome_ou_IP/nome_do_banco
```

Devemos substituir a variável *nome_ou_IP* pelo endereço IP ou pelo nome da máquina que está executando o processo do SGBD. Já a variável *nome_do_banco* deve ser substituída pelo nome do banco de dados a ser utilizado.

O supracitado método é sobrecarregado na referida classe e desta forma, na maioria das vezes, por questões relativas à configuração do ambiente computacional (redes, SGBD, dentre outros), é também necessário informar, além da URL, o usuário e a senha de conexão ao SGBD.

Nos exemplos a seguir, a classe “PersistenciaDados.java” foi abstraída para implementar, além do registro e da conexão, também a execução de algumas queries.

08

3 - Exemplos de uso da API JDBC

Como forma de demonstrar o uso da API JDBC descrita neste módulo, seguem abaixo os códigos de exemplo. Observe que os códigos fontes estão devidamente comentados. Desta forma, a leitura do código fonte é importante para continuidade da compreensão do conteúdo. O referido exemplo é uma evolução daquele apresentado no módulo anterior.

Para acessá-los adequadamente, após a devida implantação no servidor de aplicação, basta realizar uma requisição HTTP, por meio do browser web, para a URL adequada que possui a página dinâmica “FormularioLivroVisitas.jsp”

3.1 Script DDL do banco de dados MySQL

```
#comando do Mysql para criar o banco de dados
CREATE DATABASE aiec;

#comando do MySQL para utilizar a base de dados criada
USE aiec;

#comando do MySQL para criar a tabela
CREATE TABLE tbl_pessoa (

      id INT AUTO_INCREMENT PRIMARY KEY,
      nome VARCHAR(100),
      dataNascimento DATE
)
```

09

3.2 text_pt_BR.properties

```
# Esse arquivo deve ser instalado no pacote "br.aiec.i18n"
#
# Nome do arquivo: text_pt_BR.properties
#
site.titulo = Sistema i18n (Internacionalização)
site.titulo.historico = Histórico de Visitantes
site.saudacao = Seja bem Vindo

site.campo.indice = Indice
site.campo.nome = Nome
site.campo.idade = Idade
site.campo.dataNascimento = Data de Nascimento
```

10

3.3 text.properties

```
# Esse arquivo deve ser instalado no pacote "br.aiec.i18n"
#
# Esse é considerado o arquivo DEFAULT de idiomas (en_US)
#
# Nome do arquivo: text.properties
#

site.titulo = i18n System
site.titulo.historico = History of Visits
```

```
site.saudacao = Welcome

site.campo.indice = Index
site.campo.nome = Name
site.campo.idade = Age
site.campo.dataNascimento = Birthday
```


3.4 FormularioLivroVisitas.jsp

```
<%-- Está página é a primeira página da aplicação web de exemplo.
---- É aquela que constroi o formulário HTML de entrada dos dados
--%>
<%@page language="java"
          contentType="text/html; charset=UTF-8"
          pageEncoding="UTF-8"%>

<%--Importando as taglibs core e formating JSTL --%>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>

<%--Definindo a variável idioma por sessão --%>
<c:set var="idioma" value="${not empty param.language ? param.language
: not empty language ? language : pageContext.request.locale}"
scope="session" />

<%--Alterando o locale --%>
<fmt:setLocale value="${idioma}" />

<%--Definindo o pacote.nomeArquivo (fqdn) das mensagens --%>
<fmt:setBundle basename="br.aiec.i18n.text" />

<!DOCTYPE html>
<html lang="${idioma}">
     <head>
          <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
          <title>
               <fmt:message key="site.titulo" />
          </title>
     </head>
     <body>
          <h1 align="center">
```

```
                <fmt:message key="site.saudacao"/>
            </h1>
            <form action="BeanPessoaLivroVisitas.jsp" method="post">
                <fmt:message key="site.campo.nome" />
                <input type="text" name="nome"><br />

                <fmt:message key="site.campo.dataNascimento" />
                <input type="text" name="dataNascimento">
(dd/MM/yyyy)<br />

                <input type="submit">
            </form>
        </body>
</html>
```

12

3.5 BeanPessoaLivroVisitas.jsp

```
<%-- Está página não produz qualquer resultado para a interface do
usuário.
---- O objetivo da mesma ter sido utilizada foi, apenas, para fins
didáticos.
---- Nesta página são demonstrados quatro coisas:
----
---- 1) uso de tags de ação padrão
---- 2) criação do bean pessoa a partir dos parâmetros do formulário
html
---- 3) uso da taglib de formatação para converter a data
---- 4) redirecionamento interno da requisição
--%>

<%@page language="java"
            contentType="text/html; charset=UTF-8"
            pageEncoding="UTF-8"%>

<%@page import="br.aiec.negocio.Pessoa"%>

<%--Importando a taglibs formating JSTL --%>
<%@taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>

<!DOCTYPE html>
<html lang="${idioma}">
        <head>
```

```
            <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
            <title></title>
      </head>
      <body>
            <jsp:useBean id="pessoaBean" class="br.aiec.negocio.Pessoa"
scope="request">

                  <%--Utilizando o elemento de ação para atribuir o
valor do nome que foi recebido como parâmetro --%>
                  <jsp:setProperty name="pessoaBean" property="nome"
                        value="${param.nome}" />

                  <%--Utilizando a taglib parseDate para converter o
parâmetro date recebido em formato String --%>
                  <fmt:parseDate pattern="dd/MM/yyyy"
var="dtNascimento"
                        value="${param.dataNascimento}" />

                  <%--Utilizando o elemento de ação para atribuir o
valor da data convertida --%>
                  <jsp:setProperty name="pessoaBean"
property="dataNascimento"
                        value="${dtNascimento}" />

            </jsp:useBean>

            <%-- Redirecionando para o servlet de historico de visitas
--%>
            <jsp:forward page="/historicoLivroVisitas.do" />

      </body>
</html>
```

13

3.6 HistoricoLivroVisitasDemoServlet.java

```
package br.aiec.controlador;

import java.io.IOException;
```

```
import java.util.List;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import br.aiec.negocio.Pessoa;
import br.aiec.persistencia.PersistenciaDados;

@WebServlet(name="HistoricoLivroVisitas",
urlPatterns="/historicoLivroVisitas.do")
public class HistoricoLivroVisitasDemoServlet extends HttpServlet {

      @Override
      protected void doPost(HttpServletRequest request,
HttpServletResponse response)
                  throws ServletException, IOException {

            //Recuperando o bean criado na JSP
            Pessoa pessoaBean =
(Pessoa)request.getAttribute("pessoaBean");

            //Criando o objeto responsávevl por estabelecer a conexão
com o banco de dados
            PersistenciaDados pd = new PersistenciaDados();

            //Persistindo o objeto pessoa no banco de dados
            pd.gravarPessoa(pessoaBean);

            //Consultando todos as pessoas existentes no banco de dados
            List<Pessoa> historicoVisitantes =
pd.consultarTodasPessoas();

            //Compartilhando a lista de histórico no contexto da
aplicação
            getServletContext().setAttribute("historyVisits",
historicoVisitantes);

            //Redirecionando para uma JSP exibir o histórico de
visitantes
            RequestDispatcher dispatcher =
request.getRequestDispatcher("ExibirHistorico.jsp");
            dispatcher.forward(request, response);
      }
}
```

Observe que nesta classe não existe mais o atributo do tipo **List**. Ou seja, os dados agora passarão a ser persistidos no banco de dados de modo definitivo e não mais na memória RAM de modo temporário, como vinha acontecendo nos exemplos anteriores.

14

3.7 Pessoa.java

```
package br.aiec.negocio;

import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;

public class Pessoa {
	private String nome;
	private Date dataNascimento;
	public String getNome() {
		return nome;
	}

	public void setNome(String nome) {
		this.nome = nome;
	}

	public Date getDataNascimento() {
		return dataNascimento;
	}

	public void setDataNascimento(Date dataNascimento) {
		this.dataNascimento = dataNascimento;
	}

	public int getIdade(){
		Calendar hoje = GregorianCalendar.getInstance();
		hoje.setTime(new Date());

		Calendar nascimento = GregorianCalendar.getInstance();
		nascimento.setTime(dataNascimento);

		int quantidadeAnos = hoje.get(Calendar.YEAR) -
nascimento.get(Calendar.YEAR);
		nascimento.add(Calendar.YEAR, quantidadeAnos);
		if(nascimento.after(hoje)){
			quantidadeAnos--;
		}
				return quantidadeAnos;
	}

}
```

15

3.8 ExibirHistorico.jsp

```
<%@page language="java"
```

```
        contentType="text/html; charset=UTF-8"
        pageEncoding="UTF-8"%>

<%@page import="br.aiec.negocio.Pessoa"%>

<%--Importando as taglibs core e formating JSTL --%>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>

<%--Alterando o locale --%>
<fmt:setLocale value="${idioma}" />

<%--Definindo o pacote.nomeArquivo (fqdn) das mensagens --%>
<fmt:setBundle basename="br.aiec.i18n.text"/>

<!DOCTYPE html>
<html lang="${idioma}">
      <head>
            <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
            <title>
                  <fmt:message key="site.titulo"/>
            </title>
      </head>
      <body>
            <h1>
                  <fmt:message key="site.titulo.historico"/>
            </h1>

            <table border="1">
                  <thead>
                        <tr>
                              <td><fmt:message
key="site.campo.indice"/></td>
                              <td><fmt:message
key="site.campo.nome"/></td>
                              <td><fmt:message
key="site.campo.dataNascimento"/></td>
                              <td><fmt:message
key="site.campo.idade"/></td>
                        </tr>
                  </thead>
                  <tbody>
                        <%--Esse laço percorre a lista de visitantes,
exibindo cada um deles --%>
                        <c:forEach var="pessoa" items="${historyVisits}"
varStatus="i">
                              <tr>
                                    <td>${i.count}</td>
                                    <td>${pessoa.nome}</td>
                                    <td><fmt:formatDate
```

```
pattern="dd/MM/yyyy" value="${pessoa.dataNascimento}" /></td>
                                    <td>${pessoa.idade}</td>
                                </tr>
                            </c:forEach>
                        </tbody>
                    </table>
            </body>
</html>
```

16

3.9 PersistenciaDados.java

```
package br.aiec.persistencia;

import java.sql.Connection;
import java.sql.Date;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.LinkedList;
import java.util.List;

import br.aiec.negocio.Pessoa;

public class PersistenciaDados {

      //FQDN da classe principal do driver JDBC para o Mysql
      private String DRIVER = "com.mysql.jdbc.Driver";

      //URL de conexão com o banco de dados aiec
      private String URL = "jdbc:mysql://localhost:3306/aiec";

      //Usuário de conexão do banco de dados
      private String  USER = "root";

      //Senha do usuário de conexão do banco de dados
      private String PASSWORD = "123456";

      //Objeto de conexão com o banco de dados
      private Connection conexao;

      /**
       * O método construtor tem por finalidade registrar o driver
JDBC e inicializar a conexão com o SGBD
       */
      public PersistenciaDados() {
            try {
                  //Registrando o Driver para o MySQL
```

```
                Class.forName(DRIVER);

                //Estabelecendo a conexão com SGBD
                conexao = DriverManager.getConnection(URL, USER,
PASSWORD);

           } catch (SQLException e) {
                throw new RuntimeException("Falha na conexão com o
SGBD.", e);

           } catch (ClassNotFoundException e) {
                throw new RuntimeException("Driver JDBC não
encontrado.", e);
           }
      }

      /**
       * O método tem por finalidade gravar um objeto na tabela de
pessoa do banco de dados
       *
       * @param pessoa
       */
      public void gravarPessoa(Pessoa pessoa){
           //Query sql para inserir os dados da pessoa no banco
           String sql = "INSERT INTO tbl_pessoa (nome, dataNascimento)
VALUES (?, ?)";

           try {
                //Interface universal da API JDBC
                PreparedStatement pstmt;

                //criando o objeto do tipo PreparedStatement a partir
da conexão
                pstmt = conexao.prepareStatement(sql);

                //Alterando o primeiro simbolo de ? da query pelo
nome do parâmetro do tipo Pessoa
                pstmt.setString(1, pessoa.getNome());

                //Alterando o segundo simbolo de ? da query pela data
de nascimento do parâmetro do tipo Pessoa
                pstmt.setDate(2, new
Date(pessoa.getDataNascimento().getTime()));

                //Executando a query no banco
                pstmt.execute();

                //Finalizando a query
                pstmt.close();

           } catch (SQLException e) {
```

```
                throw new RuntimeException("Falha ao inserir a pessoa
na tbl_pessoa", e);
            }
      }

      /**
       * O método tem por finalidade consultar todas as pessoas
(registros) presentes no banco de dados
       * @return
       */
      public List<Pessoa> consultarTodasPessoas() {

            //Query sql para consultar todas as pessoas do banco
            String sql = "SELECT * FROM tbl_pessoa";

            //Lista de retorno que conterá as pessoas registradas no
banco
            List<Pessoa> listPessoas = new LinkedList<Pessoa>();

            try {

                  //Interface universal da API JDBC
                  PreparedStatement pstmt;

                  //criando o objeto do tipo PreparedStatement a partir
da conexão
                  pstmt = conexao.prepareStatement(sql);

                  //Executando a query no banco
                  ResultSet rs = pstmt.executeQuery();

                  //Percorrendo o resultado da query
                  while (rs.next()) {

                        //Criando um objeto do tipo Pessoa
                        Pessoa pessoa = new Pessoa();

                        //Lendo o registro nome da posição atual e
gravando no objeto do tipo Pessoa
                        pessoa.setNome(rs.getString("nome"));

                        //Lendo o registro dataNascimento da posição
atual e gravando no objeto do tipo Pessoa

      pessoa.setDataNascimento(rs.getDate("dataNascimento"));

                        //Adicionando o objeto do tipo pessoa a lista de
resultados
                        listPessoas.add(pessoa);
                  }
```

```
            } catch (SQLException e) {
                  throw new RuntimeException("Falha ao consultar todas
as pessoas na tbl_pessoa", e);
            }

            //retornando a lista com todos os registros presentes no
banco.
            return listPessoas;
      }
}
```

Observe que a classe “PersistenciaDados.java” é responsável por persistir os dados no banco de dados utilizando a API JDBC e o driver JDBC para o SGBD MySQL.

17

RESUMO

Neste módulo vimos o modo como um aplicativo Java se comunica com um SGBD por meio da API JDBC e do Driver JDBC. A API JDBC é um conjunto de recursos padronizados que provê um acesso universal aos dados armazenados em um banco de dados qualquer para aqueles aplicativos escritos em linguagem de programação Java. O JDBC suporta quatro **categorias de drivers**: Driver de ponte JDBC-para-ODBC (tipo 1), Driver parcialmente Java para a API Nativa (tipo 2), Drivers-cliente para servidor Pure Java (tipo 3), Drivers Pure Java (tipo 4). Aqui também foi demonstrado como se pode utilizar a linguagem SQL integrada à linguagem Java por meio do JDBC.

UNIDADE 4 – CONSTRUINDO UM *SOFTWARE* WEB
MÓDULO 2 – PADRÕES DE PROJETO

01

1- PADRÕES DE PROJETO DE *SOFTWARE* ORIENTADO A OBJETO

Deste módulo até o final desta unidade, para uma melhor compreensão do conteúdo, é fundamental que você já conheça a técnica de desenvolvimento orientado a objeto bem como uma linguagem de programação que suporte as estruturas da referida técnica, como por exemplo, o Java, além de uma linguagem de modelagem gráfica como a UML (Unified Modeling Language).

Os padrões de projeto apresentados aqui descrevem soluções simples para problemas específicos no projeto que envolve *softwares* orientados a objetos. Em geral, os padrões refletem modelagens e recodificações cujo resultado se deve ao esforço dos desenvolvedores por maior reutilização, flexibilidade, modularidade, escalabilidade, portabilidade, extensibilidade, interoperabilidade, segurança, coerência, coesão e semântica em seus sistemas.

Os padrões de projeto não exigem nenhum recurso incomum da linguagem de programação e nem qualquer tipo de truque de programação. No geral, os padrões são independentes das linguagens, podendo ser implementados em diversas linguagens distintas, sendo que, em algumas delas, a materialização pode ser mais simples do que em outras.

Todas as arquiteturas orientadas a objeto bem estruturadas fazem o uso de padrões de projeto. Isso significa que uma das maneiras de se medir a qualidade de um *software* orientado a objeto é avaliar se os desenvolvedores tomaram o devido cuidado com as colaborações comuns entre seus objetos.

02

A importância de padrões na criação de sistemas complexos foi, há muito tempo, reconhecida em vários ramos da ciência. Christopher Alexander - um reconhecido arquiteto, matemático e urbanista austríaco – é considerado por muitos, como um dos pioneiros a propor a ideia de se utilizar uma linguagem de padrões em projetos de edificações e cidades.

Segundo Christopher:

"Cada padrão descreve um problema no nosso ambiente e o cerne da sua solução, de tal forma que você possa usar essa solução mais de um milhão de vezes, sem nunca fazê-lo da mesma maneira."

Muito embora Christopher estivesse falando de padrões em construção civil, a definição de padrões de projeto é plenamente válida e aderente para a construção de *software*s orientados a objeto.

As soluções que serão construídas nesta disciplina devem criar classes, atributos, métodos, objetos, interfaces, dentre outros ao invés de paredes, portas, maçanetas, janelas, plantas hidráulicas e elétricas.

Christopher Alexander
Christopher Alexander foi um dos críticos da arquitetura moderna apontando a desagregação social causada por ela. Seus estudos contribuíram para a utilização de padrões geométricos e matemáticos no urbanismo e arquitetura. Uma de suas obras, "A Pattern Language" em português: *Uma Linguagem de Padrões*, virou *best seller* entre os arquitetos. A Linguagem de Padrões constitui-se uma compilação de parâmetros de projetos estabelecidos pelo arquiteto e sua equipe, com o intuito principal de auxiliar a interlocução entre profissionais e usuários de edificações e empreendimentos urbanísticos, em processos participativos.

03

Em geral, um padrão possui quatro **elementos essenciais**:

Um padrão de projeto nomeia, abstrai e identifica os aspectos chave de uma estrutura de projeto de modo a torná-la útil, eficaz e eficiente. O padrão de projeto identifica as classes e instâncias participantes, seus papéis, colaborações e a distribuição de responsabilidades.

Resumindo, um **padrão de projeto orientado a objeto** pode ser definido como sendo descrições de classes e objetos relacionados construídos para resolver um problema geral em um contexto específico.

Três dimensões
O problema deve abranger três dimensões essenciais:

a) **claro e preciso**, ou seja, todos os conceitos e termos diretamente ligados ao problema devem ser conhecidos e não podem gerar dúvidas.

b) **empírico**, ou seja, deve ser observável (uso de métodos e técnicas apropriados) na realidade.
c) **delimitado**, ou seja, ter seu escopo bem como suas fronteiras bem delimitadas.

Por exemplo, o problema pode descrever como representar algoritmos em classes/objetos, ou pode descrever estruturas de classe/objetos sintomáticos de um projeto inflexível.

04

2- Padrões versus Estratégias (Implementações)

Os padrões são modelos abstratos de alto nível cujo objetivo é documentar o contexto, seu respectivo problema, bem como sua solução.

Os padrões de projeto são independentes de linguagem de programação e podem ser aplicados a inúmeras delas. Apesar de alguns catálogos serem específicos de determinadas linguagens, a maioria dos padrões poderá ser implementado em uma linguagem diversa da utilizada pelo respectivo catálogo.

Um **padrão de projeto**...	A **estratégia** (também chamada de implementação)...
...descreve apenas ***o que*** fazer e não *como* fazer.	...descreve o **como** fazer de um padrão.

Inclusive, o mesmo padrão pode ter várias estratégias/implementações diferentes. Isso significa que um padrão define seus limites, suas fronteiras e sua estrutura geral. Enquanto isso, a sua materialização em alguma linguagem de programação, é dada pela estratégia/implementação.

Os padrões orientados a objetos são descritos em vários catálogos disponíveis eletronicamente e em material impresso. A seguir, apresentamos dois exemplos de catálogos, bem como seus padrões. O objetivo não é de estudar de modo aprofundado todos eles. Entretanto, alguns padrões serão detalhados nos módulos seguintes a esse.

05

2.1 - Catálogo GOF (Gang of Four)

O movimento ao redor de padrões de projeto ganhou popularidade com o livro Design Patterns: Elements of Reusable Object-Oriented *Software*, publicado em 1995. Os autores desse livro, Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, são conhecidos como a "Gangue dos Quatro" (Gang of Four) ou simplesmente "GoF".

O catálogo GOF possui 23 padrões de projeto catalogados que são classificados sob duas dimensões complementares:

a) dimensão **escopo** e

b) dimensão **finalidade/propósito**.

Esse catálogo é considerado o pioneiro na área de padrões de projetos voltados para construção de *software* orientado a objeto.

Na **dimensão escopo** existem dois critérios:

Padrões de classe	Padrões de objeto
•São considerados padrões estáticos, pois utilizam o relacionamento de herança como forma de colaboração entre as estruturas essenciais.	•São considerados dinâmicos, pois utilizam outros relacionamentos (composição, agregação, realização, dentre outros) para estabelecer a colaboração entre as estruturas essenciais.

06

Na dimensão **finalidade/propósito** existem três critérios:

Padrão criacional	Padrão estrutural	Padrão comportamental
• Se preocupa com o procedimento de criação de objetos/instâncias em memória de trabalho (RAM).	•Lida com os relacionamentos entre classes ou objetos/instâncias.	• Caracterizam as maneiras pelas quais classes ou objetos /instâncias interagem e distribuem responsabilidades.

A tabela abaixo demonstra a referida classificação.

		Propósito		
		De criação	**Estrutural**	**Comportamental**
Escopo	**Classe**	Factory Method	Adapter (class)	Interpreter Template Method
	Objeto	Abstract Method Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsability Command Iterator Mediator Memento Observer State Strategy Visitor

A dimensão finalidade/propósito é a mais comumente utilizada para se referir ao catálogo GOF. Deste modo, as tabelas apresentadas a seguir se utilizam desta dimensão para expor uma breve sinopse sobre cada um deles.

07

O critério **criacional** reúne 5 padrões de projeto, conforme mostrados abaixo.

Nome do Padrão	Sinopse
Factory Method	Define uma interface para criar um objeto, porém deixa que as subclasses decidam qual classe deve ser instanciada. Esse padrão permite a uma classe postergar a instanciação para as subclasses.
Abstract Factory	Fornece uma interface para criação de uma família de objetos relacionados ou dependentes sem especificar suas subclasses concretas.
Builder	Separa a construção de um objeto complexo da sua representação, de modo que o mesmo procedimento de construção possa criar diferentes representações.
Prototype	Especifica os tipos de objetos a serem criados usando uma instância prototípica, ou seja, criar novas instâncias baseadas no protótipo (modelo).
Singleton	Garante que uma classe tenha somente uma instância e fornece um ponto global de acesso para ela.

08

O critério **estrutural** reúne 7 padrões de projeto:

Nome do Padrão	Sinopse
Adapter	Converte a interface de uma classe em outra interface esperada pelos clientes. Esse padrão permite que classes com interfaces incompatíveis consigam colaborar e se relacionar.
Bridge	Separa uma abstração de sua implementação de modo que as duas possam variar independentemente.
Composite	Compõem objetos em estrutura de árvore para representar hierarquias do tipo partes-todo. Esse padrão permite que os clientes tratem objetos individuais e composições de objetos de maneira uniforme.
Decorator	Atribui responsabilidades adicionais a um objeto dinamicamente. Esse padrão fornece uma alternativa flexível ao uso de subclasses para extensão de funcionalidades.
Façade	Fornece uma interface unificada para um conjunto de interfaces em um subsistema. Esse padrão define uma interface de nível mais alto que torna o subsistema mais simples de utilizar.
Flyweight	Utiliza-se de compartilhamento para suportar grandes quantidades de objetos, de granularidade fina, de maneira eficiente.
Proxy	Fornece um objeto representante, ou um marcador de outro objeto, para controlar o acesso ao mesmo. Esse padrão é bem versátil, pois permite que os acessos sejam remotos, virtuais, protegidos e smarts (ponteiros simples).

09

O critério **comportamental** reúne 11 padrões de projeto.

Nome do Padrão	Sinopse
Interpreter	Dada uma linguagem, define uma representação para sua gramática juntamente com um interpretador que usa a representação para interpretar sentenças nessa linguagem.
Template Method	Define o esqueleto de um algoritmo em uma operação, postergando a definição de alguns passos para as subclasses. Esse padrão permite que as subclasses redefinam certos passos de um algoritmo sem

	mudar sua estrutura.
Chain of Responsability	Evita o acoplamento do remetente de uma solicitação ao seu destinatário, dando a mais de um objeto a chance de tratar a solicitação. Encadeia os objetos receptores e passa a solicitação ao longo da cadeia até que um objeto a trate.
Command	Encapsula uma solicitação como um objeto permitindo que seja possível: parametrizar os clientes com diferentes solicitações, enfileirar solicitações, registrar (logs) solicitações ou desfazer solicitações.
Iterator	Fornece uma maneira de acessar sequencialmente os elementos de uma agregação de objeto sem expor sua representação subjacente.
Mediator	Define um objeto que encapsula a forma como um conjunto de objetos interage. Esse padrão promove o acoplamento fraco ao evitar que os objetos se refiram explicitamente uns aos outros, permitindo que seja possível variar suas interações de modo independente.
Memento	Sem violar o encapsulamento, captura e externaliza um estado interno de um objeto, de modo que o mesmo possa, posteriormente, ser restaurado para aquele estado anterior.
Observer	Define uma dependência uma-para-muitos entre objetos, de modo que, quando um objeto muda de estado, todos os seus dependentes (observadores) são automaticamente notificados e atualizados por meio do serviço de *publish/subscribe*.
State	Permite que um objeto altere seu comportamento quando seu estado interno muda, fazendo-se perceber tal modificação e acreditando parecer que o referido objeto mudou de classe.
Strategy	Define uma família de algoritmos, encapsula cada um deles e os tornam intercambiáveis. Esse padrão permite que o algoritmo varie independentemente dos clientes que o utilizam.
Visitor	Representa uma operação a ser executada sobre os elementos da estrutura de um objeto. Esse padrão permite que se defina uma nova operação sem mudar as classes dos elementos sobre os quais opera.

10

2.2 - Catálogo J2EE

O catálogo J2EE possui 21 padrões de projeto catalogados e divididos em três camadas (apresentação, negócio e integração). Esses padrões foram percebidos e idealizados para uso específico com a linguagem de programação Java. Porém, os respectivos padrões podem ser escritos em qualquer linguagem de programação com mais ou menos facilidade, a depender da linguagem escolhida. Observe que os padrões criados neste catálogo possuem uma semelhança muito grande com diversos dos padrões descritos no catálogo GOF.

O catálogo J2EE divide os padrões em três camadas:

- apresentação,
- negócio e
- integração.

A camada de apresentação reúne os oito padrões descritos na tabela abaixo:

Nome do Padrão	Sinopse
Intercepting Filter	Facilita os processamentos prévio e posterior de uma solicitação.
Front Controller	Fornece um controlador centralizado para gerenciar o tratamento de uma solicitação.
Context Object	Encapsula o estado em um modo independente de protocolo para ser compartilhado em toda a sua aplicação.
Application Controller	Centraliza e modula o gerenciamento de ação e visualização.
View Helper	Encapsula a lógica que não esteja relacionada a formatação da apresentação em componentes auxiliares (Helper).
Composite View	Cria uma visualização (View) agregada a partir de subcomponentes muito pequenos.
Service to Worker	Combina um componente distribuidor (Dispatcher) com os padrões Front Controller e View Helper.
Dispacther View	Combina um componente distribuidor (Dispatcher) com os padrões Front Controller e View Helper de modo a transferir muitas atividades para o processamento de visualização.

apresentação
Os padrões da camada de apresentação contêm os padrões relacionados às tecnologias de Servlet e JSP

(*Java Server Page*).

negócio
Os padrões da camada de negócio contêm os padrões relacionados à tecnologia EJB (*Enterprise Java Bean*).

integração
Os padrões da camada de integração contêm os padrões relacionados ao JMS (*Java Message Service*) e ao JDBC (*Java Database Connectivity*).

11

A **camada de negócio** reúne os nove padrões descritos na tabela abaixo:

Nome do Padrão	Sinopse
Businness Delegate	Encapsula o acesso a um serviço de negócio.
Service Locator	Encapsula a pesquisa de serviços e componentes.
Session Façade	Encapsula os componentes da camada de negócio e expõe um serviço de granularidade grossa para os clientes remotos.
Application Service	Centraliza e agrega o comportamento para fornecer uma camada de serviços uniforme.
Business Object	Separa os dados e a lógica de negócios usando um modelo de objeto.
Composite Entity	Implementa um Business Object persistente usando beans de entidades e POJOs locais.
Transfer Object	Transfere dados através de uma camada.
Transfer Object Assembler	Monta um objeto de transferência composto a partir e diversas fontes de dados.
Value List Handler	Manipula a pesquisa, armazena os resultados em cache e seleciona itens no resultado.

12

A **camada de integração** reúne os quatro padrões descritos na tabela abaixo:

Nome do Padrão	Sinopse
Data Access Object (DAO)	Abstrai e encapsula o acesso ao armazenamento persistente
Service Activator	Recebe mensagens e chama o processamento de modo assíncrono
Domain Store	Fornece um mecanismo de persistência transparente para os objetos de negócio.
Web Service Broker	Expõe um ou mais serviços utilizando protocolos web e XML.

De acordo com o catálogo J2EE, um padrão de projeto é um modelo a ser seguido. Alguns especialistas definem um padrão como sendo uma solução permanente para um problema recorrente em um determinado contexto.

Para o referido catálogo, os padrões devem seguir um modelo genérico conforme mostrado a seguir.

13

2.2.1 - Modelo Genérico de Padrão

- **Problema:** descreve as questões de *design* enfrentadas pelo desenvolvedor.
- **Forças**: lista as razões e motivações que afetam o problema e a solução. A lista de forças destaca as razões pelas quais alguém poderia optar por utilizar o padrão e fornece uma justificativa para o mesmo.
- **Solução**: descreve o núcleo da solução, resumidamente, bem como os elementos detalhados da mesma. A solução possui duas subdivisões:
    - **Estrutura**;
    - **Estratégia**.

- **Consequências**: descreve as vantagens e as desvantagens do padrão propriamente dito ou de sua(s) implementação(ões). Geralmente, esse item nos resultados obtidos registrando os prós e os contras na aplicação do referido modelo.

estrutura
Estrutura: utilizam-se os diagramas de classes e de sequência da UML para mostrar a estrutura básica de uma solução.

Estratégia
Estratégia: descreve as diferentes maneiras pela qual um padrão pode ser implementado.

14

RESUMO

Vimos, neste módulo, o conceito e as características relativas a padrões de projeto de *software* orientado a objeto. De forma sucinta, um padrão de projeto orientado a objeto pode ser definido como sendo descrições de classes e objetos relacionados construídos para resolver um problema geral em um contexto específico. Em geral, um padrão possui quatro elementos essenciais: o nome, o problema, a solução e as consequências. Os padrões de projeto são independentes de linguagem de programação e podem ser aplicados a inúmeras delas. Um padrão de projeto descreve apenas *o que* fazer e não *como* fazer, enquanto a estratégia (também chamada de implementação) descreve o *como* fazer de um padrão, sendo que o mesmo padrão pode ter várias estratégias/implementações diferentes. Os padrões orientados a objetos são descritos em vários catálogos disponíveis eletronicamente e em material impresso, dentre eles, vimos o catálogo GOF (Gang of Four) e o catálogo J2EE. Estudamos, também, alguns padrões que são parte integrante dos catálogos GOF e J2EE, reconhecidos mundialmente e de uso contínuo pelos desenvolvedores de *software* orientado a objeto.

UNIDADE 4 – CONSTRUINDO UM *SOFTWARE* WEB
MÓDULO 3 – DATA ACCESS OBJECT (DAO)

01

1- PADRÃO DE PROJETO DAO - DATA ACCESS OBJECT

A maioria das aplicações empresariais normalmente usa SGBD (Sistemas Gerenciadores de Banco de Dados) relacional como armazenamento persistente. Entretanto os dados podem residir em outros tipos de repositórios, como *mainframes* ou sistemas legados, repositórios LDAP (Lightweight Directory Access Protocol), banco de dados orientados a objeto (OODB), arquivos de texto, xml, dentre outros.

Os dados também podem ser armazenados em outros sistemas externos, como por exemplo, serviços de B2B (*business-to-bussiness*), serviços de cartão de crédito, serviços de *cloud computing*, dentre outros.

Diante desta variedade de mecanismos de armazenamento, as diversas APIs disponíveis para cada um desses mecanismos, não só *podem* como, naturalmente, *são* diferentes. Além disso, cada API pode fornecer extensões padronizadas ou proprietárias.

As **extensões padronizadas** não são um problema porque criam um baixo acoplamento entre o aplicativo web e o sistema de armazenamento. Porém, as **extensões proprietárias** são um problema à medida que criam um alto acoplamento entre ambas as partes envolvidas.

02

O padrão de projeto **DAO - Data Access Object** - procura resolver, exatamente, o cenário caótico descrito anteriormente. Deste modo, as forças que regem o referido padrão são:

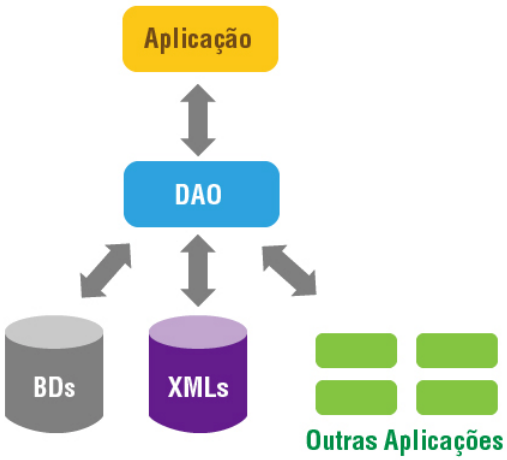
- Implementar os mecanismos de acesso aos dados para acessar e manipular dados em um armazenamento persistente;
- Desacoplar a implementação do armazenamento persistente do restante da aplicação;
- Fornecer uma interface de acesso uniforme aos dados para um mecanismo persistente para variados tipos de fonte de dados, como SGBD relacionais, LDAP, OODB, XML, arquivos texto, dentre outros.
- Organizar os recursos de lógica de acesso a dados e encapsular os recursos proprietários de modo a facilitar a manutenção e a portabilidade.

Deste modo, o DAO implementa o mecanismo de acesso necessário para se trabalhar com alguma fonte de armazenamento de dados. Independentemente do tipo de fonte de armazenamento, o DAO sempre fornece uma interface uniforme para o restante da aplicação.

O componente de negócio que necessita acessar os dados armazenados utiliza a interface simples disponibilizada pelo DAO, pois é função do referido padrão ocultar a complexidade e os detalhes de implementação da respectiva fonte de dados subjacente. Essencialmente, o DAO atua como um **adaptador** entre o componente de negócio e a fonte de dados.

03

O DAO é implementado como um objeto sem informação de estado, por padrão. Ele não armazena em cache os resultados de nenhuma execução de consulta. Isso torna os DAOs objetos leves e evita os problemas potenciais advindos do processamento concorrente ou paralelo. Por exemplo, quando uma aplicação utiliza o JDBC, o DAO encapsula toda a utilização do JDBC dentro da camada de acesso aos dados e não expõe nenhuma exceção, estrutura de dados, objetos ou interfaces que pertençam aos pacotes java.sql.* ou javax.sql.*.



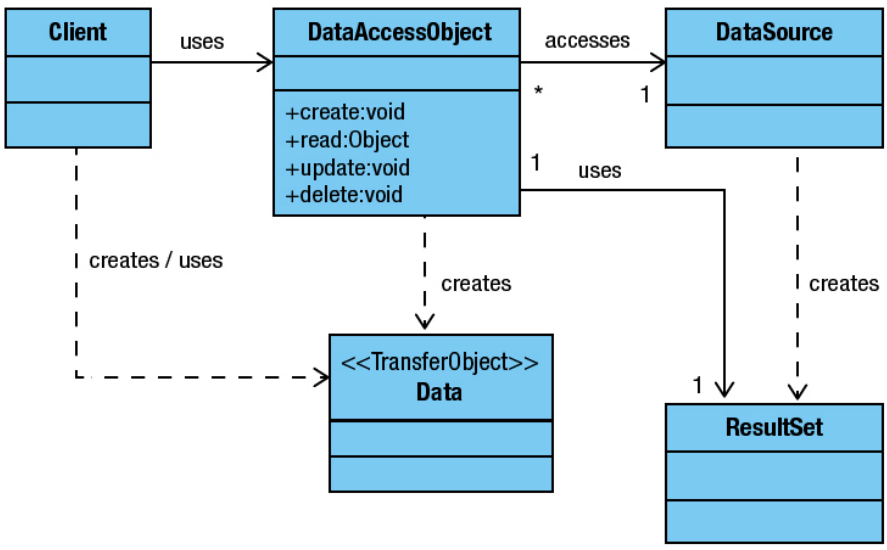
A falta de estado pode também ser um problema para a aplicação, uma vez que consultas subsequentes sobre o mesmo domínio de dados podem se tornar cada vez mais lentas em função do aumento na quantidade de solicitações consecutivas combinadas ao intervalo de tempo reduzido, feitas ao servidor de aplicação. Deste modo, visando melhorar a escalabilidade da aplicação, o padrão DAO permite o uso de ***Cached RowSet.***

Cached RowSet significa que os dados de uma consulta são armazenados em *cached* (memória RAM) para que as consultas subsequentes sejam executadas de modo mais rápido.

04

1.1 - Estrutura

Abaixo apresentamos um diagrama de classes (estruturas em UML) simplificado do padrão DAO.



Client

O *Client* é um objeto que requer o acesso à fonte de dados. Ele pode ser um Business Object, um Session Façade, um Application Serices, um Value List Handler, um Transfer Object Assembler, ou qualquer outro objeto auxiliar (Helper Object) que precise acessar os dados persistentes.

DataAccessObject

O *DataAccessObject* é o objeto de função principal desse padrão. Ele abstrai a implementação de acesso a dados subjacente para o cliente a fim de permitir um acesso transparente a fonte de dados. O referido objeto implementa os métodos de criação (insert), consulta (select), atualização (update) e exclusão (delete). Esse conjunto de métodos que permitem a realizam de tais operações são conhecidos pela sigla CRUD (create, retrieve, update e delete).

DataSource

O *DataSource* represente uma implementação da fonte de dados que pode ser, por exemplo, um SGBD relacional ou quaisquer outros mecanismos/sistema/serviço de armazenamento persistente.

ResultSet

O *ResultSet* representa o resultado de uma execução de um comando na fonte de dados. Por exemplo, para a API JDBC, essa função é desempenhada pela interface *java.sql.ResultSet.*

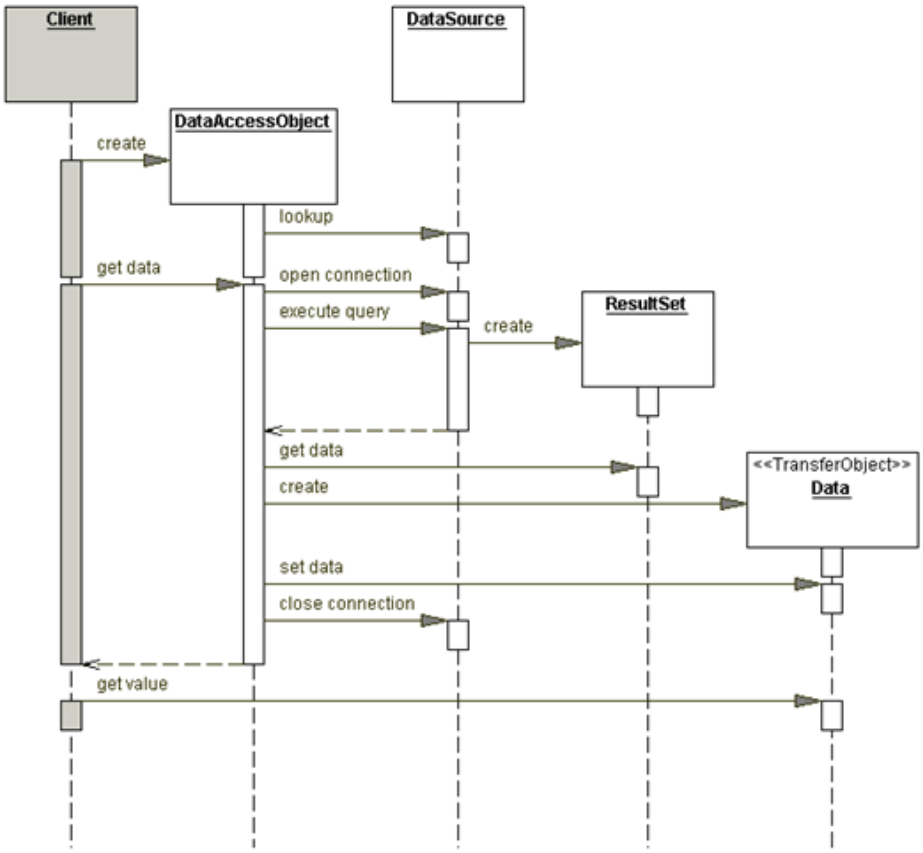
Data

O Data representa um objeto de transferência usado como um carregador de dados. Esse objeto não somente pode ser utilizado, como deve ser utilizado para permitir a comunicação bidirecional entre a camada de persistência e as demais camadas do aplicativo. Uma das funções essenciais desse objeto é o de criar um baixo acoplamento tornando a interface mais flexível entre a camada de persistência e os *clients*. Esse objeto pode ser representado pelos JavaBeans ou pelos POJO's (Plain Old Java Object).

05

1.2 - Diagrama de Sequência

O diagrama de sequência abaixo diz respeito à demonstração de um uso básico a fim de obter/ler/selecionar dados de uma fonte de dados qualquer.



06

1.3 - JPA (Java Persistence API)

A especificação JPA é um conjunto de classes/interfaces de uso específico (persistência) criado para ser utilizado juntamente com a linguagem de programação Java, independentemente de qualquer outro *framework* como, por exemplo, Servlet, JSP, EJB, dentre outros.

Apesar da JPA ter surgido internamente em função das necessidades da especificação EJB, a especificação JPA se tornou uma funcionalidade independente. Porém, é importante frisar que o uso de JPA em conjunto com EJB permite a criação de serviços de dados remotos altamente eficazes.

A primeira especificação JPA (JPA 1.0) foi originada pela Java Comunity Process (JCP) *Java Specification Request (JSR)* de número 220 em maio de 2006, ou simplesmente JCP/JSR 220. A JPA 2.0 foi o resultado da JCP/JSR 317 em dezembro de 2009 e a versão atual da JPA 2.1 foi o resultado da JCP/JSR 338 de abril de 2013.

O uso do JPA requer o conhecimento em SQL e em banco de dados relacional. Isso significa que a JPA encapsula o uso do padrão DAO de forma específica para banco de dados relacionais. A JPA facilita o processo de persistência de dados, uma vez que automatiza a persistência além de torná-la uniforme aos vários dialetos utilizados pelos diversos SGBD.

Apesar da linguagem SQL ser padronizada pelo ISO/IEC 9075, cada fabricante de SGBD's tem a liberdade de estender a padronização e criar suas próprias funcionalidades/facilidades/dialetos para uso do respectivo produto. Deste modo, utilizar o padrão DAO envolve conhecer todos os detalhes que permeiam os referidos dialetos. E, portanto, em função da quantidade de detalhes que circundam os diversos dialetos, a especificação JPA foi concebida.

A JPA é uma especificação, ou seja, possui um conjunto de classes/interfaces bem definidas para permitir desacoplar a aplicação dos inúmeros SGBD's existentes atualmente no mercado de TIC. Para que essa especificação possa ser executada, se faz necessário que a referida especificação seja implementada. Deste modo, existem diversos *providers*/*frameworks* de mercado que implementam a referida especificação tais como:

providers/frameworks

Dentre os *providers*/*frameworks* de mercado que implementam a especificação JPA, podemos citar

- *Hibernate*,
- *EclipseLink (TopLink),*
- *ObjectDB,*
- *Kundera*, dentre outros.

07

2- Exemplo do padrão DAO

Como forma de demonstrar o uso do padrão DAO descrito nesse módulo, a seguir apresentaremos os códigos de exemplo. Observe que os códigos fontes estão devidamente comentados. Desta forma, a leitura do código fonte é importante para continuidade da compreensão do conteúdo. O referido exemplo é uma evolução daquele apresentado no módulo anterior.

Neste exemplo, o padrão DAO permite à aplicação gravar os dados em um SGBD MySQL ou em um arquivo de texto (ASCII puro).

Para acessá-los adequadamente, após a devida implantação no servidor de aplicação, basta realizar uma requisição HTTP, por meio do browser web, para a URL adequada que possui a página dinâmica "FormularioLivroVisitas.jsp"

08

2.1- text_pt_BR.properties

```
# Esse arquivo deve ser instalado no pacote "br.aiec.i18n"
#
# Nome do arquivo: text_pt_BR.properties
#
```

```
site.titulo = Sistema i18n (Internacionalização)
site.titulo.historico = Histórico de Visitantes
site.saudacao = Seja bem Vindo

site.campo.indice = Indice
site.campo.nome = Nome
site.campo.idade = Idade
site.campo.dataNascimento = Data de Nascimento
site.campo.tipoBaseDados = Tipo de Base de Dados
site.campo.tipoBaseDados.mysql = Base de Dados MySQL
site.campo.tipoBaseDados.text = Arquivo Texto
```

09

2.2 - text.properties

```
# Esse arquivo deve ser instalado no pacote "br.aiec.i18n"
#
# Esse é considerado o arquivo DEFAULT de idiomas (en_US)
#
# Nome do arquivo: text.properties
#

site.titulo = i18n System
site.titulo.historico = History of Visits
site.saudacao = Welcome

site.campo.indice = Index
site.campo.nome = Name
site.campo.idade = Age
site.campo.dataNascimento = Birthday
site.campo.tipoBaseDados = Database Type
site.campo.tipoBaseDados.mysql = MySQL Data Base
site.campo.tipoBaseDados.text = Text File
```

10

2.3 - FormularioLivroVisitas.jsp

```
<%-- Esta página é a primeira página da aplicação web de exemplo.
---- É aquela que constroi o formulário HTML de entrada dos dados
--%>
<%@page language="java"
          contentType="text/html; charset=UTF-8"
          pageEncoding="UTF-8"%>

<%--Importando as taglibs core e formating JSTL --%>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>
```

```
<%--Definindo a variável idioma por sessão --%>
<c:set var="idioma" value="${not empty param.language ? param.language
: not empty language ? language : pageContext.request.locale}"
scope="session" />

<%--Alterando o locale --%>
<fmt:setLocale value="${idioma}" />

<%--Definindo o pacote.nomeArquivo (fqdn) das mensagens --%>
<fmt:setBundle basename="br.aiec.i18n.text" />

<!DOCTYPE html>
<html lang="${idioma}">
      <head>
            <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
            <title>
                  <fmt:message key="site.titulo" />
            </title>
      </head>
      <body>
            <h1 align="center">
                  <fmt:message key="site.saudacao"/>
            </h1>
            <form action="BeanPessoaLivroVisitas.jsp" method="post">
                  <fmt:message key="site.campo.nome" />
                  <input type="text" name="nome"><br />

                  <fmt:message key="site.campo.dataNascimento" />
                  <input type="text" name="dataNascimento">
(dd/MM/yyyy)<br /><br />

                  <fmt:message key="site.campo.tipoBaseDados" /><br />
                  <input type="radio" name="tipoBD"
value="MYSQL"><fmt:message key="site.campo.tipoBaseDados.mysql" /><br
/>
                  <input type="radio" name="tipoBD"
value="TEXT"><fmt:message key="site.campo.tipoBaseDados.text" /><br />


                  <input type="submit">
            </form>
      </body>
</html>
```

Observe que o formulário solicita também em qual base de dados o conteúdo informado deverá ser gravado. Essa informação será importante para que o Servlet possa estabelecer o canal de comunicação correto com a persistência.

2.4 - BeanPessoaLivroVisitas.jsp

```
<%-- Está página não produz qualquer resultado para a interface do
usuário.
---- O objetivo da mesma ter sido utilizada foi, apenas, para fins
didáticos.
---- Nesta página são demonstrados quatro coisas:
----
---- 1) uso de tags de ação padrão
---- 2) criação do bean pessoa a partir dos parâmetros do formulário
html
---- 3) uso da taglib de formatação para converter a data
---- 4) redirecionamento interno da requisição
--%>

<%@page language="java"
            contentType="text/html; charset=UTF-8"
            pageEncoding="UTF-8"%>

<%@page import="br.aiec.negocio.Pessoa"%>

<%--Importando a taglibs formating JSTL --%>
<%@taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>

<!DOCTYPE html>
<html lang="${idioma}">
      <head>
            <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
            <title></title>
      </head>
      <body>
            <jsp:useBean id="pessoaBean" class="br.aiec.negocio.Pessoa"
scope="request">

                  <%--Utilizando o elemento de ação para atribuir o
valor do nome que foi recebido como parâmetro --%>
                  <jsp:setProperty name="pessoaBean" property="nome"
                        value="${param.nome}" />

                  <%--Utilizando a taglib parseDate para converter o
parâmetro date recebido em formato String --%>
                  <fmt:parseDate pattern="dd/MM/yyyy"
var="dtNascimento"
                        value="${param.dataNascimento}" />

                  <%--Utilizando o elemento de ação para atribuir o
valor da data convertida --%>
                  <jsp:setProperty name="pessoaBean"
property="dataNascimento"
```

```
                        value="${dtNascimento}" />

            </jsp:useBean>

            <%-- Redirecionando para o servlet de historico de visitas
--%>
            <jsp:forward page="/historicoLivroVisitas.do" />

      </body>
</html>
```

12

2.5 - ExibirHistorico.jsp

```
<%@page language="java"
        contentType="text/html; charset=UTF-8"
        pageEncoding="UTF-8"%>

<%@page import="br.aiec.negocio.Pessoa"%>

<%--Importando as taglibs core e formating JSTL --%>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>

<%--Alterando o locale --%>
<fmt:setLocale value="${idioma}" />

<%--Definindo o pacote.nomeArquivo (fqdn) das mensagens --%>
<fmt:setBundle basename="br.aiec.i18n.text"/>

<!DOCTYPE html>
<html lang="${idioma}">
      <head>
            <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
            <title>
                  <fmt:message key="site.titulo"/>
            </title>
      </head>
      <body>
            <h1>
                  <fmt:message key="site.titulo.historico"/>
            </h1>

            <table border="1">
                  <thead>
                        <tr>
                              <td><fmt:message
key="site.campo.indice"/></td>
                              <td><fmt:message
```

```
key="site.campo.nome"/></td>
                              <td><fmt:message
key="site.campo.dataNascimento"/></td>
                              <td><fmt:message
key="site.campo.idade"/></td>
                        </tr>
                  </thead>
                  <tbody>
                        <%--Esse laço percorre a lista de visitantes,
exibindo cada um deles --%>
                        <c:forEach var="pessoa" items="${historyVisits}"
varStatus="i">
                              <tr>
                                    <td>${i.count}</td>
                                    <td>${pessoa.nome}</td>
                                    <td><fmt:formatDate
pattern="dd/MM/yyyy" value="${pessoa.dataNascimento}" /></td>
                                    <td>${pessoa.idade}</td>
                              </tr>
                        </c:forEach>
                  </tbody>
            </table>
      </body>
</html>
```

13

2.6 - HistoricoLivroVisitasDemoServlet.java

```
package br.aiec.controlador;

import java.io.IOException;
import java.util.List;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import br.aiec.negocio.Pessoa;
import br.aiec.persistencia.FabricaDao;
import br.aiec.persistencia.IDAOPessoa;
import br.aiec.persistencia.TipoBD;

@WebServlet(name="HistoricoLivroVisitas",
urlPatterns="/historicoLivroVisitas.do")
public class HistoricoLivroVisitasDemoServlet extends HttpServlet {

      @Override
```

```
     protected void doPost(HttpServletRequest request,
HttpServletResponse response)
                throws ServletException, IOException {

          //Recuperando o bean criado na JSP
          Pessoa pessoaBean =
(Pessoa)request.getAttribute("pessoaBean");

          //Criando o objeto responsávevl por estabelecer a conexão
com a persistência
          TipoBD tipoBD =
TipoBD.valueOf(request.getParameter("tipoBD"));
          IDAOPessoa daoPessoa = FabricaDao.getDaoPessoa(tipoBD);

          //Persistindo o objeto pessoa no banco de dados
          daoPessoa.inserir(pessoaBean);

          //Consultando todos as pessoas existentes no banco de dados
          List<Pessoa> historicoVisitantes =
daoPessoa.consultarTodasPessoas();

          //Compartilhando a lista de histórico no contexto da
aplicação
          getServletContext().setAttribute("historyVisits",
historicoVisitantes);

          //Redirecionando para uma JSP exibir o histórico de
visitantes
          RequestDispatcher dispatcher =
request.getRequestDispatcher("ExibirHistorico.jsp");
          dispatcher.forward(request, response);
     }
}
```

Observe que o servlet recebe como parâmetro o tipo de base de dados (tipoBD) em que os dados devem ser gravados. A partir daí, o objeto DAO apropriado será criado e já se torna possível perceber o funcionamento do padrão de projeto em questão.

14

2.7 - Pessoa.java

```
package br.aiec.negocio;

import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;

public class Pessoa {
```

```
	private Integer id;
	private String nome;
	private Date dataNascimento;

	public Integer getId() {
		return id;
	}

	public void setId(Integer id) {
		this.id = id;
	}

	public String getNome() {
		return nome;
	}

	public void setNome(String nome) {
		this.nome = nome;
	}

	public Date getDataNascimento() {
		return dataNascimento;
	}

	public void setDataNascimento(Date dataNascimento) {
		this.dataNascimento = dataNascimento;
	}

	public int getIdade() {
		Calendar hoje = GregorianCalendar.getInstance();
		hoje.setTime(new Date());

		Calendar nascimento = GregorianCalendar.getInstance();
		nascimento.setTime(dataNascimento);

		int quantidadeAnos = hoje.get(Calendar.YEAR)
				- nascimento.get(Calendar.YEAR);

		nascimento.add(Calendar.YEAR, quantidadeAnos);

		if (nascimento.after(hoje)) {
			quantidadeAnos--;
		}

		return quantidadeAnos;
	}

}
```

15

2.8 - TipoBD.java

```
package br.aiec.persistencia;

public enum TipoBD {
	MYSQL,
	TEXT;
}
```

2.9 - IDAOPessoa.java

```
package br.aiec.persistencia;

import java.util.List;

import br.aiec.negocio.Pessoa;

/**
 * A referida interface define as operações básicas de um CRUD
(create, retrieve, update e delete)
 *
 * @author Guilherme Veloso
 *
 */
public interface IDAOPessoa {

	public void inserir(Pessoa pessoa);

	public void atualizar(Pessoa pessoa);

	public void excluir(Pessoa pessoa);

	public List<Pessoa> consultarTodasPessoas();
}
```

16

2.10 - FabricaDao.java

```
package br.aiec.persistencia
;

import br.aiec.persistencia.arquivotexto.DAOPessoaTexto;
import br.aiec.persistencia.mysql.DAOPessoaMySQL;

/**
 * Fabrica de DAOs para o objetos de persistência
 *
```

```
 * @author Guilherme Veloso
 *
 */
public class FabricaDao {

	/**
	 * Método responsável por criar objetos DAOPessoa.
	 *
	 * O DAOPessoaMySQL será criado, por default, em caso de
parâmetro igual a "null".
	 *
	 * @param tipoBD
	 * @return
	 */
	public static IDAOPessoa getDaoPessoa(TipoBD tipoBD) {

		if(tipoBD.equals(TipoBD.TEXT)){
			return new DAOPessoaTexto();
		}

		return new DAOPessoaMySQL();
	}

}
```

17

2.11 - DAOPessoaMySQL.java

```
package br.aiec.persistencia.mysql;

import java.sql.Connection;
import java.sql.Date;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.LinkedList;
import java.util.List;

import br.aiec.negocio.Pessoa;
import br.aiec.persistencia.IDAOPessoa;

/**
 * A referida classe implementa o padrão Data Access Object para um
objeto do tipo Pessoa para o banco de dados MySQL.
 *
 * @author Guilherme Veloso
 *
 */
public class DAOPessoaMySQL implements IDAOPessoa {
```

```
	private Connection getConexao() {
		return new DataSourceMySQL().getConexao();
	}

	/**
	 * O método tem por finalidade inserir um objeto na tabela de
pessoa do
	 * banco de dados
	 *
	 * @param Pessoa
	 */
	@Override
	public void inserir(Pessoa pessoa) {
		// Obtendo a conexão com SGBD
		Connection conexao = getConexao();

		// Query sql para inserir os dados da pessoa no banco
		String sql = "INSERT INTO tbl_pessoa (nome, dataNascimento)
VALUES (?, ?)";

		try {
			// Interface universal da API JDBC
			PreparedStatement pstmt;

			// criando o objeto do tipo PreparedStatement a
partir da conexão
			pstmt = conexao.prepareStatement(sql);

			// Alterando o primeiro simbolo de ? da query pelo
nome do parâmetro
			// do tipo Pessoa
			pstmt.setString(1, pessoa.getNome());

			// Alterando o segundo simbolo de ? da query pela
data de nascimento
			// do parâmetro do tipo Pessoa
			pstmt.setDate(2, new
Date(pessoa.getDataNascimento().getTime()));

			// Executando a query no banco
			pstmt.executeUpdate();

			// Finalizando a query
			pstmt.close();

		} catch (SQLException e) {
			throw new RuntimeException(
					"Falha ao inserir a pessoa na tbl_pessoa",
e);
		}
	}
```

```
	/**
	 * O método tem por finalidade atualizar uma pessoa (registro)
presentes no
	 * banco de dados
	 *
	 * @param Pessoa
	 */
	@Override
	public void atualizar(Pessoa pessoa) {

		// Obtendo a conexão com SGBD
		Connection conexao = getConexao();

		// Query sql para consultar todas as pessoas do banco
		String sql = "UPDATE tbl_pessoa SET nome = ?,
dataNascimento = ? WHERE id = ?";

		try {
			// Interface universal da API JDBC
			PreparedStatement pstmt;

			// criando o objeto do tipo PreparedStatement a
partir da conexão
			pstmt = conexao.prepareStatement(sql);

			// Alterando o primeiro simbolo de ? da query pelo
nome do parâmetro
			// do tipo Pessoa
			pstmt.setString(1, pessoa.getNome());

			// Alterando o segundo simbolo de ? da query pela
data de nascimento
			// do parâmetro do tipo Pessoa
			pstmt.setDate(2, new
Date(pessoa.getDataNascimento().getTime()));

			// Alterando o terceiro simbolo de ? da query pelo id
do parâmetro
			// do tipo Pessoa
			pstmt.setInt(3, pessoa.getId());

			// Executando a query no banco
			pstmt.executeUpdate();

			// Finalizando a query
			pstmt.close();

		} catch (SQLException e) {
			throw new RuntimeException("Falha ao atualizar o
registro: "
```

```
                                + pessoa.getId(), e);
            }

      }

      /**
       * O método tem por finalidade excluir uma pessoa (registro)
presentes no
       * banco de dados
       *
       * @param Pessoa
       */
      @Override
      public void excluir(Pessoa pessoa) {

            // Obtendo a conexão com SGBD
            Connection conexao = getConexao();

            // Query sql para consultar todas as pessoas do banco
            String sql = "DELETE FROM tbl_pessoa WHERE id = ?";

            try {
                  // Interface universal da API JDBC
                  PreparedStatement pstmt;

                  // criando o objeto do tipo PreparedStatement a
partir da conexão
                  pstmt = conexao.prepareStatement(sql);

                  // Alterando o primeiro simbolo de ? da query pelo id
do parâmetro
                  // do tipo Pessoa
                  pstmt.setInt(1, pessoa.getId());

                  // Executando a query no banco
                  pstmt.executeUpdate();

                  // Finalizando a query
                  pstmt.close();

            } catch (SQLException e) {
                  throw new RuntimeException("Falha ao exluir o
registro: "
                                + pessoa.getId(), e);
            }

      }

      /**
       * O método tem por finalidade consultar todas as pessoas
(registros)
```

```
	 * presentes no banco de dados
	 *
	 * @return List<Pessoa>
	 */
	@Override
	public List<Pessoa> consultarTodasPessoas() {

		// Obtendo a conexão com SGBD
		Connection conexao = getConexao();

		// Query sql para consultar todas as pessoas do banco
		String sql = "SELECT * FROM tbl_pessoa";

		// Lista de retorno que conterá as pessoas registradas no
banco
		List<Pessoa> listPessoas = new LinkedList<Pessoa>();

		try {

			// Interface universal da API JDBC
			PreparedStatement pstmt;

			// criando o objeto do tipo PreparedStatement a
partir da conexão
			pstmt = conexao.prepareStatement(sql);

			// Executando a query no banco
			ResultSet rs = pstmt.executeQuery();

			// Percorrendo o resultado da query
			while (rs.next()) {

				// Criando um objeto do tipo Pessoa
				Pessoa pessoa = new Pessoa();


				// Lendo o id(chavve primária) da posição atual
e gravando no objeto
				// do tipo Pessoa
				pessoa.setId(rs.getInt("id"));

				// Lendo o registro nome da posição atual e
gravando no objeto
				// do tipo Pessoa
				pessoa.setNome(rs.getString("nome"));

				// Lendo o registro dataNascimento da posição
atual e gravando
				// no objeto do tipo Pessoa

	pessoa.setDataNascimento(rs.getDate("dataNascimento"));
```

```
                        // Adicionando o objeto do tipo pessoa a lista
de resultados
                        listPessoas.add(pessoa);
                  }

                  // Finalizando a query
                  pstmt.close();

            } catch (SQLException e) {
                  throw new RuntimeException(
                              "Falha ao consultar todas as pessoas na
tbl_pessoa", e);
            }

            // retornando a lista com todos os registros presentes no
banco.
            return listPessoas;
      }

}
```

18

2.12 - DataSourceMySQL.java

```
package br.aiec.persistencia.mysql;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DataSourceMySQL {

      // FQDN da classe principal do driver JDBC para o Mysql
      private String DRIVER = "com.mysql.jdbc.Driver";

      // URL de conexão com o banco de dados aiec
      private String URL = "jdbc:mysql://localhost:3306/aiec";

      // Usuário de conexão do banco de dados
      private String USER = "root";

      // Senha do usuário de conexão do banco de dados
      private String PASSWORD = "123456";

      // Objeto de conexão com o banco de dados
      private Connection conexao;

      /**
       * O método construtor tem por finalidade registrar o driver
```

```
JDBC e
	 * inicializar a conexão com o SGBD
	 */
	public DataSourceMySQL() {
		try {
			// Registrando o Driver para o MySQL
			Class.forName(DRIVER);

			// Estabelecendo a conexão com SGBD
			conexao = DriverManager.getConnection(URL, USER,
PASSWORD);

		} catch (SQLException e) {
			throw new RuntimeException("Falha na conexão com o
SGBD.", e);

		} catch (ClassNotFoundException e) {
			throw new RuntimeException("Driver JDBC não
encontrado.", e);
		}
	}

	/**
	 * Esse método tem por finalidade retornar o objeto de conexão
que foi criado.
	 *
	 * @return Connection
	 */
	public Connection getConexao(){
		return this.conexao;
	}
}
```

19

2.13 - DAOPessoaTexto.java

```
package br.aiec.persistencia.arquivotexto;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.Reader;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.LinkedList;
import java.util.List;
```

```
import java.util.StringTokenizer;

import br.aiec.negocio.Pessoa;
import br.aiec.persistencia.IDAOPessoa;

/**
 *
 *     A referida classe é apenas para uso didático. Deste modo, toda
a lógica presente foi reduzida ao mínimo necessário,
 *
 *   uma vez que o objetivo é demonstrar o padrão DAO (Data Access
Object). Portanto,a lógica apresentada
 *
 *   não contempla elementos como: segurança, escalabilidade,
integridade, dentre outros.
 *
 * @author Guilherme Veloso
 *
 */
public class DAOPessoaTexto implements IDAOPessoa {

      //Método utilizado para definir onde o arquivo texto será
gravado.
      private File getFile() {
            return new DataSourceTexto().getArquivoTexto();
      }

      /**
       * O método tem por finalidade inserir um objeto no arquivo
texto
       *
       * @param Pessoa
       */
      @Override
      public void inserir(Pessoa pessoa) {
            //Obtendo o arquivo de texto
            File file = getFile();

            //Obtendo o separador de linha do sistema. Para windows
(CR+LF ou \r\n). Para linux (LF ou \n)
            String newLine = System.getProperty("line.separator");

            try {
                  //Criando um objeto que permite escrever/adicionar
conteúdo no arquivo texto
```

```
                FileWriter writer = new FileWriter(file, true);

                //Formatador de data
                SimpleDateFormat sf = new
SimpleDateFormat("dd/MM/yyyy");

                //Adicionado, ao final do arquivo, o conteúdo do
parâmetro.
                writer.append(pessoa.getNome() + Delimitador.SIMBOLO
                          + sf.format(pessoa.getDataNascimento()) +
newLine);

                //Fechando o arquivo
                writer.close();

           } catch (IOException e) {
                throw new RuntimeException("Falha ao criar/abrir o
arquivo texto: "
                          + file.getAbsolutePath(), e);
           }
     }

     /**
      * O método tem por finalidade atualizar um registro no arquivo
texto
      *
      * @param Pessoa
      */
     @Override
     public void atualizar(Pessoa pessoa) {
          //obtendo a lista de todos os registros
          List<Pessoa> listPessoas = consultarTodasPessoas();

          //removendo o parâmetro da lista
          listPessoas.remove(pessoa.getId().intValue());

          //Adicionando o novo parâmetro na lista
          listPessoas.add(pessoa.getId().intValue(), pessoa);

          //Obtendo o arquivo
          File file = getFile();

          //Apagando arquivo
          file.delete();
```

```
            //Inserindo cada elemento da lista em um novo arquivo
            for (int i = 0; i < listPessoas.size(); i++) {
                  inserir(listPessoas.get(i));
            }
      }

      /**
       * O método tem por finalidade excluir um registro no arquivo
texto
       *
       * @param Pessoa
       */
      @Override
      public void excluir(Pessoa pessoa) {
            //obtendo a lista de todos os registros
            List<Pessoa> listPessoas = consultarTodasPessoas();

            //removendo o parâmetro da lista
            listPessoas.remove(pessoa.getId().intValue());

            //Obtendo o arquivo
            File file = getFile();

            //Apagando arquivo
            file.delete();

            //Inserindo cada elemento da lista em um novo arquivo
            for (int i = 0; i < listPessoas.size(); i++) {
                  inserir(listPessoas.get(i));
            }
      }

      /**
       *
       * O método tem por finalidade consultar todas as pessoas
(registros)
       *
       * presentes no arquivo de texto
       *
       * @return List<Pessoa>
       */
      @Override
      public List<Pessoa> consultarTodasPessoas() {

            //Obtendo o arquivo de texto
```

```
            File file = getFile();

            // Lista de retorno que conterá as pessoas registradas no
banco
            List<Pessoa> listPessoas = new LinkedList<Pessoa>();

            try {
                  //Criando um Reader para ler o arquivo de texto
                  Reader in = new FileReader(file);

                  //Criando um buffer reader para facilitar a leitura
por linhas do reader
                  BufferedReader reader = new BufferedReader(in);

                  //Variável para contar o número de linhas lidas (Esse
número é considerado a chave primária do registro)
                  Integer numberLine = 0;

                  //Variável para guardar o conteúdo da linha
propriamente dita
                  String line;

                  //Laço que irá percorrer todas as linhas do arquivo
                  while ((line = reader.readLine()) != null) {

                        //Cada linha possui um DELIMITADOR
                        StringTokenizer st = new StringTokenizer(line,
Delimitador.SIMBOLO.toString());

                        //Criando um objeto do tipo pessoa
                        Pessoa pessoa = new Pessoa();

                        //atribuindo o numero da linha(chave primária do
registro)
                        pessoa.setId(numberLine++);

                        //atribuindo o primeiro token (nome)
                        pessoa.setNome(st.nextToken());

                        //Formatador de datas
                        SimpleDateFormat sf = new
SimpleDateFormat("dd/MM/yyyy");

                        //convertendo o segundo token (dataNascimento de
String para Date) e atribuindo o resultado ao objeto pessoa
```

```
      pessoa.setDataNascimento(sf.parse(st.nextToken()));

                        //adicionando a pessoa (um registro lido) a
lista de resultados
                        listPessoas.add(pessoa);
                  }

                  //fechando o reader(arquivo texto)
                  reader.close();

            } catch (IOException e) {
                  throw new RuntimeException("Falha ao ler o arquivo
texto: "
                              + file.getAbsolutePath(), e);
            } catch (ParseException e) {
                  throw new RuntimeException("Falha ao converter a
data: "
                              + file.getAbsolutePath(), e);
            }

            // retornando a lista com todos os registros presentes no
arquivo texto.
            return listPessoas;
      }

}
```

20

2.14 - DataSourceTexto.java

```
package br.aiec.persistencia.arquivotexto;

import java.io.File;

public class DataSourceTexto {

      //Atributo para armazenar o arquivo texto
      private File file;

      /**
       * Esse método tem por finalidade criar o arquivo texto no
diretório corrente de execução da aplicação
       */
      public DataSourceTexto() {
```

```
            //Diretório de trabalho corrente
            String diretorioUsuario = System.getProperty("user.dir");

            //Separador de arquivo. Para linux "/" e para windows "\"
            String separadorArquivo =
System.getProperty("file.separator");

            //Nome do arquivo que irá armazenar o conteúdo da aplicação
            String nomearquivo = "arquivoRegistroPessoas.txt";

            //Local onde o arquivo será gravado.
            String pathname = diretorioUsuario + separadorArquivo +
nomearquivo;

            //Criando o objeto File para o arquivo
            file = new File(pathname);

      }

      /**
       *  Esse método tem por finalidade retornar o arquivo texto
       *
       * @return File
       */
      public File getArquivoTexto() {
            return file;
      }

}
```

21

2.15 - Delimitador.java

```
package br.aiec.persistencia.arquivotexto;

/**
 *
 * O referido DELIMITADOR é utilizado para separar o nome da data de
nascimento de uma pessoa em cada registro
 * do arquivo de texto (nome#dataNascimento).
 *
 * @author Guilherme Veloso
 *
 */
public enum Delimitador {
      SIMBOLO("#");

      private String valor;

      private Delimitador(String valor) {
```

```
            this.valor = valor;
      }

      @Override
      public String toString() {
            return this.valor;
      }
}
```

22

RESUMO

Apresentamos, neste módulo, o padrão de projeto DAO do catálogo JEE. Isso significa que os componentes integrantes do padrão como DataSource, ResultSet, Data Transfer Object (DTO) são construídos de maneira clara e objetiva. O padrão DAO tem como objetivos:

- Implementar os mecanismos de acesso aos dados para acessar e manipular dados em um armazenamento persistente;
- Desacoplar a implementação do armazenamento persistente do restante da aplicação;
- Fornecer uma interface de acesso uniforme aos dados para um mecanismo persistente para variados tipos de fonte de dados, como SGBD relacionais, LDAP, OODB, XML, arquivos texto, dentre outros.
- Organizar os recursos de lógica de acesso a dados e encapsular os recursos proprietários de modo a facilitar a manutenção e a portabilidade.

DAO atua como um adaptador entre o componente de negócio e a fonte de dados. O referido padrão é demonstrado, permitindo que o aplicativo web armazene os dados da aplicação em um arquivo texto (ASCII puro) ou em um SGBD relacional (MySQL).

UNIDADE 4 – CONSTRUINDO UM *SOFTWARE* WEB
MÓDULO 4 – MODEL–VIEW–CONTROLER (MVC)

01

1 - O PADRÃO MODEL-VIEW-CONTROLLER (MVC)

Os padrões de projeto, frequentemente, são utilizados de modo combinado dentro de uma mesma solução de *design* de *software*.

Model-View-Controller (MVC) é um padrão de arquitetura de *software* para implementação de interfaces com o usuário, que se utiliza desta combinação de outros padrões mais simples.

O MVC foi criado por Trygve Reenskaug no final da década de 1970, dentro da linguagem de programação orientada a objeto denominada de SmallTalk-76, concebida pela Xerox PARC. Nesta época, em 1979, os desenvolvedores de *software* passavam por grandes dificuldades para construir qualquer tipo de interface com o usuário que fosse flexível. Isso significa que para cada novo aplicativo que era construído, os programadores tinham que se preocupar, sempre, com a apresentação (localização, tamanho, forma, distância, cor, dentre outras) do conteúdo para os usuários dos computadores.

Deste modo, os programadores gastavam muito mais tempo e energia com a construção dessas apresentações do que com a lógica do negócio propriamente dita. Nesta mesma época, a computação começou a se popularizar devido aos esforços de padronização criados pelo hardware durante a década de 70. Portanto, vários novos dispositivos, principalmente àqueles diretamente ligados a permitir a criação de um canal de comunicação com o usuário como, por exemplo, o monitor, o mouse, o teclado, a impressora, dentre vários outros, começaram a se tornar disponíveis e de fácil acesso a uma quantidade cada vez mais crescente de usuários.

Diante disso, a programação de novos aplicativos que atendessem a todas essas manifestações de interfaces ficou algo extremamente complexo e de difícil construção, uma vez que os programadores passaram a ter que se preocupar com os inúmeros detalhes advindos do uso desses novos dispositivos.

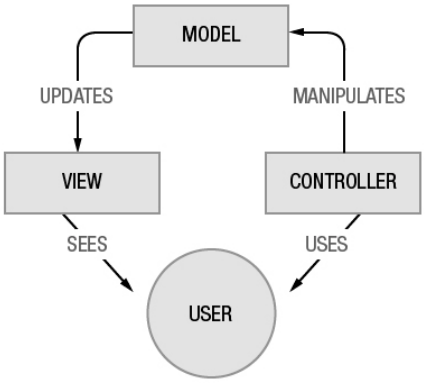
Diante disso, o MVC foi criado para desacoplar a camada de Visão (**View**) da camada de Modelo (**Model**), principalmente para as interfaces que utilizavam o monitor como dispositivo de comunicação com o usuário. Diante disso, o Controlador (**Controller**) nada mais é do que uma camada de indireção criada para permitir uma relação de baixo acoplamento entre a View e o Model.

02

O padrão MVC divide a aplicação em três partes interconectadas de modo que a representação interna da informação seja separada em camadas lógicas cujo significado seja construído por componentes que procuram representar, de modo pleno e em todas as suas dimensões, o domínio do problema.

Considere que o uso do MVC, apesar de popularmente conhecido e definido, ainda necessita, no momento de sua materialização/implementação/codificação, de uma carga de subjetividade inerente ao conhecimento das várias equipes de desenvolvimento de sistemas computacionais. Portanto, não se esqueça daquilo que foi descrito sobre os padrões de projeto: os **padrões de projeto são independentes de sua implementação**, ou seja, para cada padrão podem existir diversas implementações com pequenas variações e diferenças.

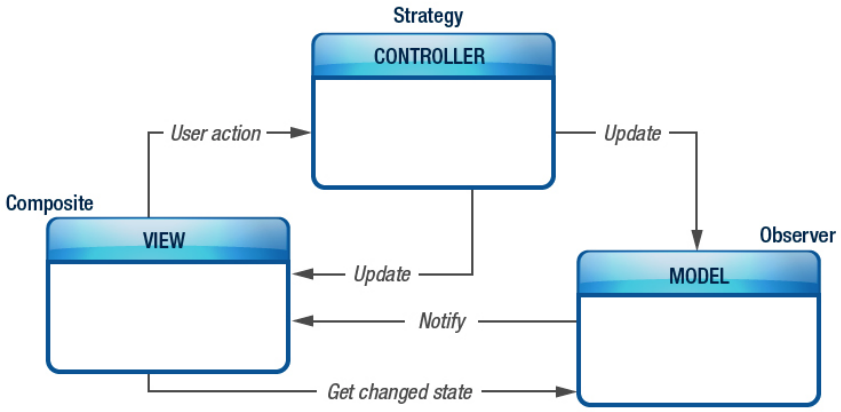
A figura abaixo demonstra o fluxo de dados mais comum do referido padrão.



03

O MVC, além de ser considerado um padrão arquitetural, é também considerado um padrão de projeto híbrido ou complexo ou composto, uma vez que o mesmo se utiliza, basicamente, de três padrões GOF (**Observer**, **Composite** e **Strategy**) harmonicamente interligados. Apesar de outros padrões como o Factory Method, Decorator, Mediator, Command, dentre outros também poderem acrescentar capacidades ao MVC, a essência do MVC está ligada ao relacionamento criado pelos padrões Observer, Composite e Strategy.

Uma imagem mais genérica do padrão MVC está logo abaixo:



A visualização e o controlador estabelecem uma relação por meio do padrão **Strategy** clássico, ou seja, a visualização é um objeto configurado como uma estratégia. O controlador é quem fornece essa estratégia. Portanto, a visualização somente precisa se preocupar com os aspectos visuais do aplicativo, porque todas as decisões sobre o comportamento da interface são de responsabilidade do controlador.

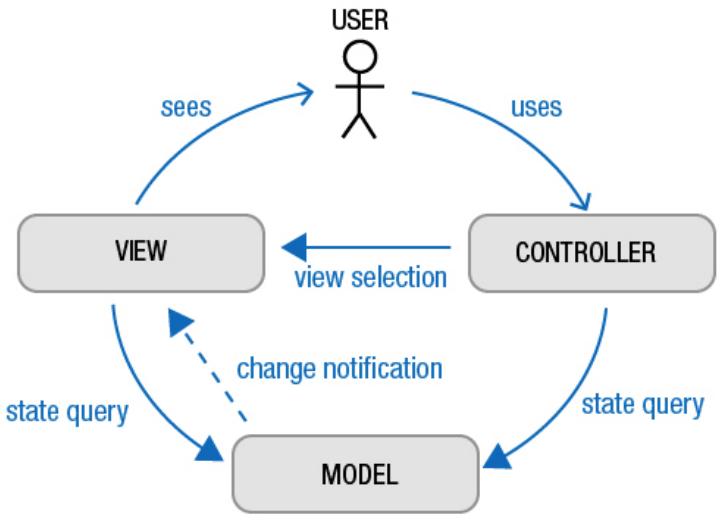
O uso do padrão Strategy também mantém a visualização distante do modelo, ou seja, a visualização apenas se registra ao modelo desconhecendo-o quase que totalmente. Isso significa que a responsabilidade pela interação com o modelo, para executar as solicitações do usuário, é delegada apenas ao controlador. Deste modo a visualização não tem a menor ideia de como isso é feito.

04

A visualização, por si só, também implementa o padrão **Composite**. A tela consiste em um conjunto aninhado de janelas, painéis, botões, dentre diversos outros, compostos para permitir a construção de tal camada. Isso significa que quando o controlador determina que a visualização atualize a tela, essa ordem é enviada apenas ao componente de mais alto nível. A partir daí, o Composite se encarrega de atualizar todos os demais componentes da tela.

O Modelo implementa o padrão de projeto **Observer** para manter os objetos interessados constantemente informado sobre suas mudanças de estado. O uso do padrão Observer mantém o Modelo fracamente acoplado à Visualização e ao Controlador, permitindo deste modo o uso de várias visões diferentes ligadas ao mesmo Modelo ou até mesmo o uso de múltiplas visões simultâneas.

Para finalizar, uma última figura, menos formal, que ilustra a compreensão que se deve ter do padrão MVC.



05

2- MVC 2 – Model 2

Em 1998, a atual Sun Microsystems publicou um release da especificação JSP, versão 0.92. Nesta especificação a Sun dizia haver dois modelos para uso de JSPs: Model 1 e Model 2.

O **Model 1** era um modelo simplista que utilizava páginas JSP sozinhas e isoladas (*standalone*) desvinculadas de qualquer outro componente.

Inclusive, o Model 1 defendia que toda a lógica de negócios deveria estar escrita na própria JSP.

O **Model 2** utilizava Servlet e JSP de modo integrado e combinado com divisão de responsabilidades clara e precisa.

Em dezembro de 1999, Govind Seshadri publicou um artigo no JavaWorld intitulado *Understanding JavaServer Pages Model 2 architecture.* Em uma de suas passagens, Govind dizia que era necessário retirar a lógica de negócios da JSP e escrevê-la em alguma estrutura de dados à parte. Neste caso, Govind utilizou um vetor para demonstrar tal estrutura. Portanto, fica claro que Govind estava demonstrando que as JSP representariam a camada de Visão do modelo MVC enquanto que a estrutura de dados vetor seria o Modelo.

Neste contexto, o Servlet representaria o Controlador, cuja finalidade era a de permitir uma comunicação entre a Visão e o Modelo. A partir de então, várias iniciativas surgiram para fomentar o Model 2 como, por exemplo, o framework Apache Struts, que foi criado no ano 2000.

06

Desde então, para a **arquitetura JEE**, basicamente:

- as JSPs representam a Visão,
- os Servlets representam o Controlador e
- os EJBs representam o Modelo.

Essa divisão lógica é questionável, pois toda página JSP é um Servlet, porém é uma prática muito comum e bem aceita pela maioria dos desenvolvedores e centro de pesquisas espalhados pelo planeta Terra.

Nesta disciplina em específico, adotaremos essa divisão. Porém, a camada de Modelo, será representada, praticamente, pelos JavaBeans, POJO e Business Object (BO) adaptado uma vez que os EJBs estão além do escopo desta cadeira acadêmica. Inclusive, uma simples observação, em todos os códigos fontes de exemplos presente nos módulos anteriores, já demonstra o uso disseminado de várias ideias relacionadas ao padrão de projeto MVC.

As camadas que ainda restam serem evoluídas e que serão objeto de estudo neste módulo são: a camada de **Controle** e a camada **Modelo**. Ambas ainda se encontram em fase inicial, apenas com a presença de algumas classes.

07

2.1- MVC2 = MVC + Internet

Com o advento da internet, não demorou muito para que os desenvolvedores começassem a adaptar o MVC para atuar na arquitetura cliente/servidor. A adaptação mais comum é conhecida como Model 2 ou **MVC 2**. Esse nome está diretamente ligado à ideia de Govind Seshadri.

A adaptação se deve, basicamente, ao fato de na web o protocolo HTTP ser um protocolo do tipo request/response, ou seja, o servidor não responde de maneira proativa, mas sim de maneira reativa. Isso significa que o servidor de aplicação web consegue responder, se e somente se houver uma requisição de algum cliente.

Essa característica implica em adaptações que devem ser feitas nas formas de notificação que a camada de Modelo realiza proativamente na camada de Visão quando da alteração de estado da primeira camada. Apesar do padrão de projeto Observer permitir que os objetos observadores (View's) verifiquem, de modo ativo, por meio de solicitações, os estados dos objetos observados (Model's), que passivamente respondem, este comportamento não é o comportamento *default*.

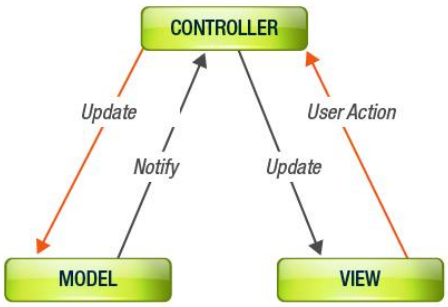
O padrão de projeto Observer possui como comportamento essencial (*default*) aquele que se utiliza do mecanismo de *publish-subscribe* onde o funcionamento é exatamente o contrário do anterior, ou seja, os observadores (View's) são notificados, de modo passivo, pelos observados (Model's), que ativamente realizam as devidas notificações quando da mudança de seus próprios estados. Isso acontece porque a Visão deve refletir o exato momento que o estado do Modelo se altera. Deste modo, apenas o Modelo sabe, de forma precisa e instantânea, quando seu estado é alterado.

A grande questão é que a **View está no lado cliente** (ativo) enquanto que o **Model está no lado servidor** (passivo) e, utilizando-se o protocolo HTTP, é impossível que o servidor (Model's) notifique o cliente (View's). Em função do funcionamento do protocolo HTTP impor limitações ao funcionamento do MVC tradicional, o padrão precisou ser adaptado.

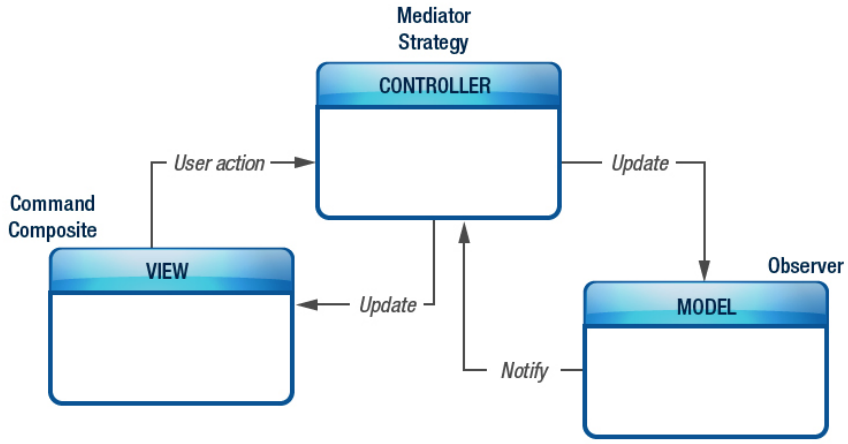
08

Apesar do MVC2 adaptar o MVC tradicional, isso não representa uma grande mudança nos objetivos do padrão, uma vez que os objetivos continuam sendo desacoplar a camada de Visão da camada de Modelo pelos motivos já expostos anteriormente. Portanto, o que de fato aconteceu, é que o Model agora não notifica mais a View no MVC 2.

A figura abaixo demonstra essa diferença fundamental.



Essa outra figura também representa o modelo MVC2 sob uma outra dimensão que exibe os padrões GOF essenciais e o relacionamento entre as camadas do *pattern* arquitetural. Observe a alteração na comunicação entre o Model e a View.



09

3- Exemplo do padrão MVC2

É importante salientar que durante todo o curso, o padrão MVC2 já foi sendo construído, mesmo que este ainda não tenha sido definido. Portanto, este exemplo final evoluiu as camadas que ainda se encontravam embrionárias nos exemplos anteriores.

Como forma de demonstrar o pleno uso do padrão MVC2 já iniciado no começo do curso e finalizado neste módulo, a seguir apresentaremos os códigos de exemplo. Observe que os códigos fontes estão devidamente comentados. Desta forma, a leitura do código fonte é importante para continuidade da compreensão do conteúdo. O referido exemplo é uma evolução daquele apresentado no módulo anterior.

Neste exemplo, o padrão MVC2 tem sua camada de Modelo implantada definindo uma única regra de negócio, a saber: **apenas visitantes maiores de 18 anos poderão realizar o registro no livro de visitas**. Além disso, as demais funcionalidades do CRUD que estavam faltando, foram desenvolvidas em suas respectivas camadas.

Observe que além do MVC2, temos também o padrão DAO como integrante deste exemplo em uma quarta camada e interligada à camada de Modelo do MVC2. Ou seja, o referido exemplo demonstra uma arquitetura classificada como “n-camadas”.

Para acessá-los adequadamente, após a devida implantação no servidor de aplicação, basta realizar uma requisição HTTP, por meio do browser web, para a URL adequada que possui a página dinâmica “FormularioLivroVisitasIncluir.jsp”

10

3.1 text_pt_BR.properties

```
# Esse arquivo deve ser instalado no pacote "br.aiec.i18n"
#
# Nome do arquivo: text_pt_BR.properties
#
site.titulo = Sistema i18n (Internacionalização)
site.titulo.historico = Histórico de Visitantes
site.saudacao = Seja bem Vindo

site.campo.indice = Indice
site.campo.nome = Nome
site.campo.idade = Idade
site.campo.dataNascimento = Data de Nascimento
site.campo.editar = Editar
site.campo.excluir = Excluir
site.campo.tipoBaseDados = Tipo de Base de Dados
site.campo.tipoBaseDados.mysql = Base de Dados MySQL
site.campo.tipoBaseDados.text = Arquivo Texto
```

11

3.2 text.properties

```
# Esse arquivo deve ser instalado no pacote "br.aiec.i18n"
#
# Esse é considerado o arquivo DEFAULT de idiomas (en_US)
#
# Nome do arquivo: text.properties
#

site.titulo = i18n System
site.titulo.historico = History of Visits
site.saudacao = Welcome

site.campo.indice = Index
site.campo.nome = Name
site.campo.idade = Age
site.campo.dataNascimento = Birthday
site.campo.editar = Edit
site.campo.excluir = Delete
site.campo.tipoBaseDados = Database Type
site.campo.tipoBaseDados.mysql = MySQL Data Base
site.campo.tipoBaseDados.text = Text File
```

12

3.3 FormularioLivroVisitasIncluir.jsp

```
<%-- Esta página é a primeira página da aplicação web de exemplo.
---- É aquela que constroi o formulário HTML de entrada dos dados
```

```
--%>
<%@page language="java"
            contentType="text/html; charset=UTF-8"
            pageEncoding="UTF-8"%>

<%--Importando as taglibs core e formating JSTL --%>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>

<%--Definindo a variável idioma por sessão --%>
<c:set var="idioma" value="${not empty param.language ? param.language
: not empty language ? language : pageContext.request.locale}"
scope="session" />

<%--Alterando o locale --%>
<fmt:setLocale value="${idioma}" />

<%--Definindo o pacote.nomeArquivo (fqdn) das mensagens --%>
<fmt:setBundle basename="br.aiec.i18n.text" />

<!DOCTYPE html>
<html lang="${idioma}">
      <head>
            <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
            <title>
                  <fmt:message key="site.titulo" />
            </title>
      </head>
      <body>
            <h1 align="center">
                  <fmt:message key="site.saudacao"/>
            </h1>
            <form action="ControladorBeanPessoaLivroVisitas.jsp"
method="post">
                  <fmt:message key="site.campo.nome" />
                  <input type="text" name="nome"><br />

                  <fmt:message key="site.campo.dataNascimento" />
                  <input type="text" name="dataNascimento">
(dd/MM/yyyy)<br /><br />

                  <fmt:message key="site.campo.tipoBaseDados" /><br />
                  <input type="radio" name="tipoBD"
value="MYSQL"><fmt:message key="site.campo.tipoBaseDados.mysql" /><br
/>
                  <input type="radio" name="tipoBD"
value="TEXT"><fmt:message key="site.campo.tipoBaseDados.text" /><br />

                  <input type="submit">
            </form>
```

```
      </body>
</html>
```

13

3.4 ControladorBeanPessoaLivroVisitas.jsp

```
<%-- Está página não produz qualquer resultado para a interface do
usuário.
---- O objetivo da mesma ter sido utilizada foi, apenas, para fins
didáticos.
---- Nesta página são demonstrados quatro coisas:
----
---- 1) uso de tags de ação padrão
---- 2) criação do bean pessoa a partir dos parâmetros do formulário
html
---- 3) uso da taglib de formatação para converter a data
---- 4) redirecionamento interno da requisição
--%>

<%@page language="java"
           contentType="text/html; charset=UTF-8"
           pageEncoding="UTF-8"%>

<%@page import="br.aiec.negocio.Pessoa"%>

<%--Importando a taglibs formating JSTL e core --%>
<%@taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<!DOCTYPE html>
<html lang="${idioma}">
     <head>
          <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
          <title></title>
     </head>
     <body>
          <jsp:useBean id="pessoaBean" class="br.aiec.negocio.Pessoa"
scope="request">

               <%--Utilizando o elemento de ação para atribuir o
valor do id que foi recebido como parâmetro --%>
               <jsp:setProperty name="pessoaBean" property="id"
                    value="${param.id}" />

               <%--Utilizando o elemento de ação para atribuir o
valor do nome que foi recebido como parâmetro --%>
               <jsp:setProperty name="pessoaBean" property="nome"
```

```
                        value="${param.nome}" />

                <%--Utilizando a taglib parseDate para converter o
parâmetro date recebido em formato String --%>
                <fmt:parseDate pattern="dd/MM/yyyy"
var="dtNascimento"
                        value="${param.dataNascimento}" />

                <%--Utilizando o elemento de ação para atribuir o
valor da data convertida --%>
                <jsp:setProperty name="pessoaBean"
property="dataNascimento"
                        value="${dtNascimento}" />

           </jsp:useBean>

           <c:choose>
                <c:when test="${empty param.id}">
                     <%-- Redirecionando para o servlet de incluir
visitas --%>
                     <jsp:forward page="/livroVisitasIncluir.do" />
                </c:when>
                <c:otherwise>
                     <%-- Redirecionando para o servlet de alterar
visitas --%>
                     <jsp:forward page="/livroVisitasAlterar.do" />
                </c:otherwise>
           </c:choose>
      </body>
</html>
```

14

3.5 FormularioLivroVisitasAlterar.jsp

```
<%-- Esta página é a primeira página da aplicação web de exemplo.
---- É aquela que constroi o formulário HTML de entrada dos dados
--%>
<%@page language="java"
           contentType="text/html; charset=UTF-8"
           pageEncoding="UTF-8"%>

<%--Importando as taglibs core e formating JSTL --%>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>

<%--Definindo a variável idioma por sessão --%>
<c:set var="idioma" value="${not empty param.language ? param.language
: not empty language ? language : pageContext.request.locale}"
scope="session" />
```

```
<%--Alterando o locale --%>
<fmt:setLocale value="${idioma}" />

<%--Definindo o pacote.nomeArquivo (fqdn) das mensagens --%>
<fmt:setBundle basename="br.aiec.i18n.text" />

<!DOCTYPE html>
<html lang="${idioma}">
      <head>
            <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
            <title>
                  <fmt:message key="site.titulo" />
            </title>
      </head>
      <body>
            <h1 align="center">
                  <fmt:message key="site.saudacao"/>
            </h1>
            <form action="ControladorBeanPessoaLivroVisitas.jsp"
method="post">
                  <fmt:message key="site.campo.nome" />
                  <input type="text" name="nome"
value="${pessoa.nome}"><br />

                  <fmt:message key="site.campo.dataNascimento" />
                  <input type="text" name="dataNascimento"
value="<fmt:formatDate pattern="dd/MM/yyyy"
value="${pessoa.dataNascimento}" />"> (dd/MM/yyyy)<br /><br />

                  <%--Criando campos ocultos para permitir a
identificação por parte do Servlet que irá efetuar o procedimento de
alteração dos dados --%>
                  <input type="hidden" name="tipoBD"
value="${param.tipoBD}">
                  <input type="hidden" name="id" value="${pessoa.id}">

                  <input type="submit">
            </form>
      </body>
</html>
```

15

3.6 Excecoes.jsp

```
<%@page language="java"
            contentType="text/html; charset=UTF-8"
      pageEncoding="ISO-8859-1"%>
```

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Página de Mensagens de Exceções</title>
</head>
<body>
      <h1 align="center">${excecao.message}</h1>
</body>
</html>
```

16

3.7 ExibirHistorico.jsp

```
<%@page language="java"
            contentType="text/html; charset=UTF-8"
      pageEncoding="UTF-8"%>

<%@page import="br.aiec.negocio.Pessoa"%>

<%--Importando as taglibs core e formating JSTL --%>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>

<%--Alterando o locale --%>
<fmt:setLocale value="${idioma}" />

<%--Definindo o pacote.nomeArquivo (fqdn) das mensagens --%>
<fmt:setBundle basename="br.aiec.i18n.text"/>

<!DOCTYPE html>
<html lang="${idioma}">
      <head>
            <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
            <title>
                  <fmt:message key="site.titulo"/>
            </title>
      </head>
      <body>
            <h1>
                  <fmt:message key="site.titulo.historico"/>
            </h1>

            <table border="1">
                  <thead>
                        <tr>
                              <td><fmt:message
```

```
key="site.campo.indice"/></td>
                              <td><fmt:message
key="site.campo.nome"/></td>
                              <td><fmt:message
key="site.campo.dataNascimento"/></td>
                              <td><fmt:message
key="site.campo.idade"/></td>
                              <td><fmt:message
key="site.campo.editar"/></td>
                              <td><fmt:message
key="site.campo.excluir"/></td>

                        </tr>
                  </thead>
                  <tbody>
                        <%--Esse laço percorre a lista de visitantes,
exibindo cada um deles --%>
                        <c:forEach var="pessoa" items="${historyVisits}"
varStatus="i">
                              <tr>
                                    <td>${i.count}</td>
                                    <td>${pessoa.nome}</td>
                                    <td><fmt:formatDate
pattern="dd/MM/yyyy" value="${pessoa.dataNascimento}" /></td>
                                    <td>${pessoa.idade}</td>
                                    <td><a
href="livroVisitasAlterar.do?id=${pessoa.id}&tipoBD=${param.tipoBD}">e
ditar</a></td>
                                    <td><a
href="livroVisitasExcluir.do?id=${pessoa.id}&tipoBD=${param.tipoBD}">e
xcluir</a></td>
                              </tr>
                        </c:forEach>
                  </tbody>
            </table>
      </body>
</html>
```

17

3.8 LivroVisitasIncluirDemoServlet.java

```
package br.aiec.controlador;

import java.io.IOException;
import java.util.List;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
```

```
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import br.aiec.negocio.ModeloPessoa;
import br.aiec.negocio.Pessoa;
import br.aiec.persistencia.TipoBD;

@WebServlet(name = "LivroVisitasIncluir", urlPatterns =
"/livroVisitasIncluir.do")
public class LivroVisitasIncluirDemoServlet extends HttpServlet {

	@Override
	protected void doPost(HttpServletRequest request,
			HttpServletResponse response) throws
ServletException, IOException {

		try {
			// Recuperando o bean criado na JSP
			Pessoa pessoaBean = (Pessoa)
request.getAttribute("pessoaBean");

			// Criando o objeto responsávevl por estabelecer a
conexão com o
			// Modelo (Regras de Negócio)
			TipoBD tipoBD =
TipoBD.valueOf(request.getParameter("tipoBD"));
			ModeloPessoa modelo = new ModeloPessoa(tipoBD);

			// Acessando o modelo de dados para os objetos do
tipo Pessoa
			modelo.inserir(pessoaBean);

			// Consultando todos as pessoas existentes no modelo
			List<Pessoa> historicoVisitantes =
modelo.consultarTodasPessoas();

			// Compartilhando a lista de histórico no contexto da
aplicação
			getServletContext().setAttribute("historyVisits",
					historicoVisitantes);

			// Redirecionando para uma JSP exibir o histórico de
visitantes
			RequestDispatcher dispatcher = request

	.getRequestDispatcher("ExibirHistorico.jsp");
			dispatcher.forward(request, response);

		} catch (Exception e) {
```

```
                // Redirecionando para uma JSP de exceção
                RequestDispatcher dispatcher = request
                          .getRequestDispatcher("Excecoes.jsp");

                //Atribuindo o objeto de exceção como um atributo da
requisição
                request.setAttribute("excecao", e);

                //redirecionamento interno
                dispatcher.forward(request, response);
          }
     }
}
```

18

3.9 LivroVisitasAlterarDemoServlet.java

```
package br.aiec.controlador;

import java.io.IOException;
import java.util.List;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import br.aiec.negocio.ModeloPessoa;
import br.aiec.negocio.Pessoa;
import br.aiec.persistencia.TipoBD;

@WebServlet(name = "LivroVisitasAlterar", urlPatterns =
"/livroVisitasAlterar.do")
public class LivroVisitasAlterarDemoServlet extends HttpServlet {

     @Override
     protected void doGet(HttpServletRequest request,
                HttpServletResponse response) throws
ServletException, IOException {

          Pessoa parametro = new Pessoa();

     parametro.setId(Integer.parseInt(request.getParameter("id")));

          // Criando o objeto responsávevl por estabelecer a conexão
com o
          // Modelo (Regras de Negócio)
```

```
            TipoBD tipoBD =
TipoBD.valueOf(request.getParameter("tipoBD"));
            ModeloPessoa modelo = new ModeloPessoa(tipoBD);

            // Acessando o modelo de dados para os objetos do tipo
Pessoa
            Pessoa pessoa = modelo.consultarPessoa(parametro);

            request.setAttribute("pessoa", pessoa);

            // Redirecionando para uma JSP exibir o histórico de
visitantes
            RequestDispatcher dispatcher = request

      .getRequestDispatcher("FormularioLivroVisitasAlterar.jsp");
            dispatcher.forward(request, response);
      }

      @Override
      protected void doPost(HttpServletRequest request,
HttpServletResponse response)
                  throws ServletException, IOException {

            // Recuperando o bean criado na JSP
            Pessoa pessoaBean = (Pessoa)
request.getAttribute("pessoaBean");

            // Criando o objeto responsávevl por estabelecer a conexão
com o
            // Modelo (Regras de Negócio)
            TipoBD tipoBD =
TipoBD.valueOf(request.getParameter("tipoBD"));
            ModeloPessoa modelo = new ModeloPessoa(tipoBD);

            // Acessando o modelo de dados para os objetos do tipo
Pessoa
            modelo.atualizar(pessoaBean);

            // Consultando todos as pessoas existentes no modelo
            List<Pessoa> historicoVisitantes =
modelo.consultarTodasPessoas();

            // Compartilhando a lista de histórico no contexto da
aplicação
            getServletContext().setAttribute("historyVisits",
historicoVisitantes);

            // Redirecionando para uma JSP exibir o histórico de
visitantes
            RequestDispatcher dispatcher = request
                        .getRequestDispatcher("ExibirHistorico.jsp");
```

```
            dispatcher.forward(request, response);
      }
}
```

19

3.10 LivroVisitasExcluirDemoServlet.java

```
package br.aiec.controlador;

import java.io.IOException;
import java.util.List;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import br.aiec.negocio.ModeloPessoa;
import br.aiec.negocio.Pessoa;
import br.aiec.persistencia.TipoBD;

@WebServlet(name = "LivroVisitasExcluir", urlPatterns =
"/livroVisitasExcluir.do")
public class LivroVisitasExcluirDemoServlet extends HttpServlet {

      @Override
      protected void doGet(HttpServletRequest request,
                  HttpServletResponse response) throws
ServletException, IOException {

            Pessoa pessoaBean = new Pessoa();

      pessoaBean.setId(Integer.parseInt(request.getParameter("id")));

            // Criando o objeto responsávevl por estabelecer a conexão
com o
            // Modelo (Regras de Negócio)
            TipoBD tipoBD =
TipoBD.valueOf(request.getParameter("tipoBD"));
            ModeloPessoa modelo = new ModeloPessoa(tipoBD);

            // Acessando o modelo de dados para os objetos do tipo
Pessoa
            modelo.excluir(pessoaBean);

            // Consultando todos as pessoas existentes no modelo
            List<Pessoa> historicoVisitantes =
```

```
modelo.consultarTodasPessoas();

            // Compartilhando a lista de histórico no contexto da
aplicação
            getServletContext().setAttribute("historyVisits",
historicoVisitantes);

            // Redirecionando para uma JSP exibir o histórico de
visitantes
            RequestDispatcher dispatcher = request
                        .getRequestDispatcher("ExibirHistorico.jsp");
            dispatcher.forward(request, response);
      }
}
```

20

3.11 ModeloPessoa.java

```
package br.aiec.negocio;

import java.util.List;

import br.aiec.persistencia.FabricaDao;
import br.aiec.persistencia.IDAOPessoa;
import br.aiec.persistencia.TipoBD;

/**
 * Classe que possui as Regras de Negócio para os objetos do tipo
Pessoa
 *
 * @author Guilherme Veloso
 *
 */
public class ModeloPessoa {

      private IDAOPessoa daoPessoa;

      public ModeloPessoa(TipoBD tipoBD) {
            daoPessoa = FabricaDao.getDaoPessoa(tipoBD);
      }

      /**
       * É necessário que as pessoas tenham 18 ou mais anos para que o
registro seja realizado.
       *
       * @param Pessoa
       */
      public void inserir(Pessoa pessoaBean) {
```

```
            //Regra de negócio que permite gravar apenas usuários com
18 ou mais anos de idade!
            if(pessoaBean.getIdade() < 18){
                  throw new RuntimeException("É preciso ter 18 ou mais
anos para efetuar os registro no livro de visitas!");
            }
            daoPessoa.inserir(pessoaBean);
      }

      /**
       * Consultar todas as pessoas existentes no sistema
       *
       * @return List<Pessoa>
       */
      public List<Pessoa> consultarTodasPessoas() {
            return daoPessoa.consultarTodasPessoas();
      }

      /**
       *
       * Excluir uma pessoa do sistema
       *
       * @param Pessoa
       */
      public void excluir(Pessoa pessoaBean) {
            daoPessoa.excluir(pessoaBean);
      }

      /**
       *
       * Atualizar uma pessoa do sistema
       *
       * @param Pessoa
       */
      public void atualizar(Pessoa pessoaBean) {
            daoPessoa.atualizar(pessoaBean);
      }

      /**
       *
       * Método para consultar a pessoa pelo identificador (Id)
       *
       * @param Pessoa
       */
      public Pessoa consultarPessoa(Pessoa pessoaBean) {
            return daoPessoa.consultarPessoa(pessoaBean);
      }
      }
```

3.12 Pessoa.java

```
package br.aiec.negocio;

import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;

public class Pessoa {

	private Integer id;
	private String nome;
	private Date dataNascimento;

	public Integer getId() {
		return id;
	}

	public void setId(Integer id) {
		this.id = id;
	}

	public String getNome() {
		return nome;
	}

	public void setNome(String nome) {
		this.nome = nome;
	}

	public Date getDataNascimento() {
		return dataNascimento;
	}

	public void setDataNascimento(Date dataNascimento) {
		this.dataNascimento = dataNascimento;
	}

	public int getIdade() {
		Calendar hoje = GregorianCalendar.getInstance();
		hoje.setTime(new Date());

		Calendar nascimento = GregorianCalendar.getInstance();
		nascimento.setTime(dataNascimento);

		int quantidadeAnos = hoje.get(Calendar.YEAR)
				- nascimento.get(Calendar.YEAR);

		nascimento.add(Calendar.YEAR, quantidadeAnos);

		if (nascimento.after(hoje)) {
```

```
                quantidadeAnos--;
            }

            return quantidadeAnos;
      }

}
```

22

3.13 IDAOPessoa.java

```
package br.aiec.persistencia;

import java.util.List;

import br.aiec.negocio.Pessoa;

/**
 * A referida interface define as operações básicas de um CRUD
(create, retrieve, update e delete)
 *
 * @author Guilherme Veloso
 *
 */
public interface IDAOPessoa {

      public void inserir(Pessoa pessoa);

      public void atualizar(Pessoa pessoa);

      public void excluir(Pessoa pessoa);

      public List<Pessoa> consultarTodasPessoas();

      public Pessoa consultarPessoa(Pessoa parametro);
}
```

3.14 TipoBD.java

```
package br.aiec.persistencia;
public enum TipoBD {
      MYSQL,
      TEXT;
}
```

23

3.15 FabricaDAO.java

```
package br.aiec.persistencia;

import br.aiec.persistencia.arquivotexto.DAOPessoaTexto;
import br.aiec.persistencia.mysql.DAOPessoaMySQL;

/**
 * Fabrica de DAOs para o objetos de persistência
 *
 * @author Guilherme Veloso
 *
 */
public class FabricaDao {

	/**
	 * Método responsável por criar objetos DAOPessoa.
	 *
	 * O DAOPessoaMySQL será criado, por default, em caso de
parâmetro igual a "null".
	 *
	 * @param tipoBD
	 * @return
	 */
	public static IDAOPessoa getDaoPessoa(TipoBD tipoBD) {

		if(tipoBD.equals(TipoBD.TEXT)){
			return new DAOPessoaTexto();
		}

		return new DAOPessoaMySQL();
	}

}
```

24

3.16 DataSourceMySQL.java

```
package br.aiec.persistencia.mysql;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DataSourceMySQL {

	// FQDN da classe principal do driver JDBC para o Mysql
	private String DRIVER = "com.mysql.jdbc.Driver";

	// URL de conexão com o banco de dados aiec
	private String URL = "jdbc:mysql://localhost:3306/aiec";
```

```
	// Usuário de conexão do banco de dados
	private String USER = "root";

	// Senha do usuário de conexão do banco de dados
	private String PASSWORD = "123456";

	// Objeto de conexão com o banco de dados
	private Connection conexao;

	/**
	 * O método construtor tem por finalidade registrar o driver
JDBC e
	 * inicializar a conexão com o SGBD
	 */
	public DataSourceMySQL() {
		try {
			// Registrando o Driver para o MySQL
			Class.forName(DRIVER);

			// Estabelecendo a conexão com SGBD
			conexao = DriverManager.getConnection(URL, USER,
PASSWORD);

		} catch (SQLException e) {
			throw new RuntimeException("Falha na conexão com o
SGBD.", e);

		} catch (ClassNotFoundException e) {
			throw new RuntimeException("Driver JDBC não
encontrado.", e);
		}
	}

	/**
	 * Esse método tem por finalidade retornar o objeto de conexão
que foi criado.
	 *
	 * @return Connection
	 */
	public Connection getConexao(){
		return this.conexao;
	}
}
```

25

3.17 DAOPessoaMySQL.java

```
package br.aiec.persistencia.mysql;
```

```
import java.sql.Connection;
import java.sql.Date;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.LinkedList;
import java.util.List;

import br.aiec.negocio.Pessoa;
import br.aiec.persistencia.IDAOPessoa;

/**
 * A referida classe implementa o padrão Data Access Object para um
objeto do
 * tipo Pessoa para o banco de dados MySQL.
 *
 * @author Guilherme Veloso
 *
 */
public class DAOPessoaMySQL implements IDAOPessoa {

	private Connection getConexao() {
		return new DataSourceMySQL().getConexao();
	}

	/**
	 * O método tem por finalidade inserir um objeto na tabela de
pessoa do
	 * banco de dados
	 *
	 * @param Pessoa
	 */
	@Override
	public void inserir(Pessoa pessoa) {
		// Obtendo a conexão com SGBD
		Connection conexao = getConexao();

		// Query sql para inserir os dados da pessoa no banco
		String sql = "INSERT INTO tbl_pessoa (nome, dataNascimento)
VALUES (?, ?)";

		try {
			// Interface universal da API JDBC
			PreparedStatement pstmt;

			// criando o objeto do tipo PreparedStatement a
partir da conexão
			pstmt = conexao.prepareStatement(sql);

			// Alterando o primeiro simbolo de ? da query pelo
```

```
nome do parâmetro
                // do tipo Pessoa
                pstmt.setString(1, pessoa.getNome());

                // Alterando o segundo simbolo de ? da query pela
data de nascimento
                // do parâmetro do tipo Pessoa
                pstmt.setDate(2, new
Date(pessoa.getDataNascimento().getTime()));

                // Executando a query no banco
                pstmt.executeUpdate();

                // Finalizando a query
                pstmt.close();

           } catch (SQLException e) {
                throw new RuntimeException(
                          "Falha ao inserir a pessoa na tbl_pessoa",
e);
           }
     }

     /**
      * O método tem por finalidade atualizar uma pessoa (registro)
presentes no
      * banco de dados
      *
      * @param Pessoa
      */
     @Override
     public void atualizar(Pessoa pessoa) {

           // Obtendo a conexão com SGBD
           Connection conexao = getConexao();

           // Query sql para consultar todas as pessoas do banco
           String sql = "UPDATE tbl_pessoa SET nome = ?,
dataNascimento = ? WHERE id = ?";

           try {
                // Interface universal da API JDBC
                PreparedStatement pstmt;

                // criando o objeto do tipo PreparedStatement a
partir da conexão
                pstmt = conexao.prepareStatement(sql);

                // Alterando o primeiro simbolo de ? da query pelo
nome do parâmetro
                // do tipo Pessoa
```

```
                pstmt.setString(1, pessoa.getNome());

                // Alterando o segundo simbolo de ? da query pela
data de nascimento
                // do parâmetro do tipo Pessoa
                pstmt.setDate(2, new
Date(pessoa.getDataNascimento().getTime()));

                // Alterando o terceiro simbolo de ? da query pelo id
do parâmetro
                // do tipo Pessoa
                pstmt.setInt(3, pessoa.getId());

                // Executando a query no banco
                pstmt.executeUpdate();

                // Finalizando a query
                pstmt.close();

           } catch (SQLException e) {
                throw new RuntimeException("Falha ao atualizar o
registro: "
                           + pessoa.getId(), e);
           }

     }

     /**
      * O método tem por finalidade excluir uma pessoa (registro)
presentes no
      * banco de dados
      *
      * @param Pessoa
      */
     @Override
     public void excluir(Pessoa pessoa) {

           // Obtendo a conexão com SGBD
           Connection conexao = getConexao();

           // Query sql para consultar todas as pessoas do banco
           String sql = "DELETE FROM tbl_pessoa WHERE id = ?";

           try {
                // Interface universal da API JDBC
                PreparedStatement pstmt;

                // criando o objeto do tipo PreparedStatement a
partir da conexão
                pstmt = conexao.prepareStatement(sql);
```

```
                // Alterando o primeiro simbolo de ? da query pelo id
do parâmetro
                // do tipo Pessoa
                pstmt.setInt(1, pessoa.getId());

                // Executando a query no banco
                pstmt.executeUpdate();

                // Finalizando a query
                pstmt.close();

          } catch (SQLException e) {
                throw new RuntimeException("Falha ao exluir o
registro: "
                            + pessoa.getId(), e);
          }

     }

     /**
      * O método tem por finalidade consultar todas as pessoas
(registros)
      * presentes no banco de dados
      *
      * @return List<Pessoa>
      */
     @Override
     public List<Pessoa> consultarTodasPessoas() {

          // Obtendo a conexão com SGBD
          Connection conexao = getConexao();

          // Query sql para consultar todas as pessoas do banco
          String sql = "SELECT * FROM tbl_pessoa";

          // Lista de retorno que conterá as pessoas registradas no
banco
          List<Pessoa> listPessoas = new LinkedList<Pessoa>();

          try {

                // Interface universal da API JDBC
                PreparedStatement pstmt;

                // criando o objeto do tipo PreparedStatement a
partir da conexão
                pstmt = conexao.prepareStatement(sql);

                // Executando a query no banco
                ResultSet rs = pstmt.executeQuery();
```

```
                // Percorrendo o resultado da query
                while (rs.next()) {

                      // Criando um objeto do tipo Pessoa
                      Pessoa pessoa = new Pessoa();

                      // Lendo o id(chavve primária) da posição atual
e gravando no
                      // objeto
                      // do tipo Pessoa
                      pessoa.setId(rs.getInt("id"));

                      // Lendo o registro nome da posição atual e
gravando no objeto
                      // do tipo Pessoa
                      pessoa.setNome(rs.getString("nome"));

                      // Lendo o registro dataNascimento da posição
atual e gravando
                      // no objeto do tipo Pessoa

      pessoa.setDataNascimento(rs.getDate("dataNascimento"));

                      // Adicionando o objeto do tipo pessoa a lista
de resultados
                      listPessoas.add(pessoa);
                }

                // Finalizando a query
                pstmt.close();

            } catch (SQLException e) {
                  throw new RuntimeException(
                              "Falha ao consultar todas as pessoas na
tbl_pessoa", e);
            }

            // retornando a lista com todos os registros presentes no
banco.
            return listPessoas;
      }

      @Override
      public Pessoa consultarPessoa(Pessoa parametro) {
            // Obtendo a conexão com SGBD
            Connection conexao = getConexao();

            // Query sql para consultar todas as pessoas do banco
            String sql = "SELECT * FROM tbl_pessoa WHERE id = ?";

            // Lista de retorno que conterá as pessoas registradas no
```

```
banco
            Pessoa resultado = new Pessoa();

            try {

                  // Interface universal da API JDBC
                  PreparedStatement pstmt;

                  // criando o objeto do tipo PreparedStatement a
partir da conexão
                  pstmt = conexao.prepareStatement(sql);

                  // Alterando o primeiro simbolo de ? da query pelo id
do parâmetro
                  // do tipo Pessoa
                  pstmt.setInt(1, parametro.getId());

                  // Executando a query no banco
                  ResultSet rs = pstmt.executeQuery();

                  // Percorrendo o resultado da query
                  while (rs.next()) {

                        // Lendo o id(chavve primária) da posição atual
e gravando no
                        // objeto
                        // do tipo Pessoa
                        resultado.setId(rs.getInt("id"));

                        // Lendo o registro nome da posição atual e
gravando no objeto
                        // do tipo Pessoa
                        resultado.setNome(rs.getString("nome"));

                        // Lendo o registro dataNascimento da posição
atual e gravando
                        // no objeto do tipo Pessoa

      resultado.setDataNascimento(rs.getDate("dataNascimento"));

                  }

                  // Finalizando a query
                  pstmt.close();

            } catch (SQLException e) {
                  throw new RuntimeException(
                              "Falha ao consultar todas as pessoas na
tbl_pessoa", e);
            }
```

```
            // retornando a pessoa pesquisada
            return resultado;
      }

}
```

26

3.18 DelimitadorTexto.java

```
package br.aiec.persistencia.arquivotexto;

/**
 *
 * O referido DELIMITADOR é utilizado para separar o nome da data de
nascimento de uma pessoa em cada registro
 * do arquivo de texto (nome#dataNascimento).
 *
 * @author Guilherme Veloso
 *
 */
public enum DelimitadorTexto {
      SIMBOLO("#");

      private String valor;

      private DelimitadorTexto(String valor) {
            this.valor = valor;
      }

      @Override
      public String toString() {
            return this.valor;
      }
}
```

27

3.19 DataSourceTexto.java

```
package br.aiec.persistencia.arquivotexto;

import java.io.File;

public class DataSourceTexto {

      //Atributo para armazenar o arquivo texto
      private File file;
```

```
	/**
	 * Esse método tem por finalidade criar o arquivo texto no
diretório corrente de execução da aplicação
	 */
	public DataSourceTexto() {

		//Diretório de trabalho corrente
		String diretorioUsuario = System.getProperty("user.dir");

		//Separador de arquivo. Para linux "/" e para windows "\"
		String separadorArquivo =
System.getProperty("file.separator");

		//Nome do arquivo que irá armazenar o conteúdo da aplicação
		String nomearquivo = "arquivoRegistroPessoas.txt";

		//Local onde o arquivo será gravado.
		String pathname = diretorioUsuario + separadorArquivo +
nomearquivo;

		//Criando o objeto File para o arquivo
		file = new File(pathname);

	}

	/**
	 *  Esse método tem por finalidade retornar o arquivo texto
	 *
	 * @return File
	 */
	public File getArquivoTexto() {
		return file;
	}

}
```

28

3.20 DAOPessoaTexto.java

```
package br.aiec.persistencia.arquivotexto;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.Reader;
import java.text.ParseException;
import java.text.SimpleDateFormat;
```

```
import java.util.LinkedList;
import java.util.List;
import java.util.StringTokenizer;

import br.aiec.negocio.Pessoa;
import br.aiec.persistencia.IDAOPessoa;

/**
 *
 * A referida classe é apenas para uso didático. Deste modo, toda a
lógica
 * presente foi reduzida ao mínimo necessário,
 *
 * uma vez que o objetivo é demonstrar o padrão DAO (Data Access
Object).
 * Portanto,a lógica apresentada
 *
 * não contempla elementos como: segurança, escalabilidade,
integridade, dentre
 * outros.
 *
 * @author Guilherme Veloso
 *
 */
public class DAOPessoaTexto implements IDAOPessoa {

	// Método utilizado para definir onde o arquivo texto será
gravado.
	private File getFile() {
		return new DataSourceTexto().getArquivoTexto();
	}

	/**
	 * O método tem por finalidade inserir um objeto no arquivo
texto
	 *
	 * @param Pessoa
	 */
	@Override
	public void inserir(Pessoa pessoa) {
		// Obtendo o arquivo de texto
		File file = getFile();

		// Obtendo o separador de linha do sistema. Para windows
(CR+LF ou
		// \r\n). Para linux (LF ou \n)
		String newLine = System.getProperty("line.separator");

		try {
			// Criando um objeto que permite escrever/adicionar
conteúdo no
```

```
                // arquivo texto
                FileWriter writer = new FileWriter(file, true);

                // Formatador de data
                SimpleDateFormat sf = new
SimpleDateFormat("dd/MM/yyyy");

                // Adicionado, ao final do arquivo, o conteúdo do
parâmetro.
                writer.append(pessoa.getNome() +
DelimitadorTexto.SIMBOLO
                            + sf.format(pessoa.getDataNascimento()) +
newLine);

                // Fechando o arquivo
                writer.close();

           } catch (IOException e) {
                throw new RuntimeException("Falha ao criar/abrir o
arquivo texto: "
                            + file.getAbsolutePath(), e);
           }
     }

     /**
      * O método tem por finalidade atualizar um registro no arquivo
texto
      *
      * @param Pessoa
      */
     @Override
     public void atualizar(Pessoa pessoa) {
          // obtendo a lista de todos os registros
          List<Pessoa> listPessoas = consultarTodasPessoas();

          // removendo o parâmetro da lista
          listPessoas.remove(pessoa.getId().intValue());

          // Adicionando o novo parâmetro na lista
          listPessoas.add(pessoa.getId().intValue(), pessoa);

          // Obtendo o arquivo
          File file = getFile();

          // Apagando arquivo
          file.delete();

          // Inserindo cada elemento da lista em um novo arquivo
          for (int i = 0; i < listPessoas.size(); i++) {
                inserir(listPessoas.get(i));
          }
```

```
	}

	/**
	 * O método tem por finalidade excluir um registro no arquivo
texto
	 *
	 * @param Pessoa
	 */
	@Override
	public void excluir(Pessoa pessoa) {
		// obtendo a lista de todos os registros
		List<Pessoa> listPessoas = consultarTodasPessoas();

		// removendo o parâmetro da lista
		listPessoas.remove(pessoa.getId().intValue());

		// Obtendo o arquivo
		File file = getFile();

		// Apagando arquivo
		file.delete();

		// Inserindo cada elemento da lista em um novo arquivo
		for (int i = 0; i < listPessoas.size(); i++) {
			inserir(listPessoas.get(i));
		}
	}

	/**
	 *
	 * O método tem por finalidade consultar todas as pessoas
(registros)
	 *
	 * presentes no arquivo de texto
	 *
	 * @return List<Pessoa>
	 */
	@Override
	public List<Pessoa> consultarTodasPessoas() {

		// Obtendo o arquivo de texto
		File file = getFile();

		// Lista de retorno que conterá as pessoas registradas no
banco
		List<Pessoa> listPessoas = new LinkedList<Pessoa>();

		try {
			// Criando um Reader para ler o arquivo de texto
			Reader in = new FileReader(file);
```

```
                // Criando um buffer reader para facilitar a leitura
por linhas do
                // reader
                BufferedReader reader = new BufferedReader(in);

                // Variável para contar o número de linhas lidas
(Esse número é
                // considerado a chave primária do registro)
                Integer numberLine = 0;

                // Variável para guardar o conteúdo da linha
propriamente dita
                String line;

                // Laço que irá percorrer todas as linhas do arquivo
                while ((line = reader.readLine()) != null) {

                      // Cada linha possui um DELIMITADOR
                      StringTokenizer st = new StringTokenizer(line,
                                  DelimitadorTexto.SIMBOLO.toString());

                      // Criando um objeto do tipo pessoa
                      Pessoa pessoa = new Pessoa();

                      // atribuindo o numero da linha(chave primária
do registro)
                      pessoa.setId(numberLine++);

                      // atribuindo o primeiro token (nome)
                      pessoa.setNome(st.nextToken());

                      // Formatador de datas
                      SimpleDateFormat sf = new
SimpleDateFormat("dd/MM/yyyy");

                      // convertendo o segundo token (dataNascimento
de String para
                      // Date) e atribuindo o resultado ao objeto
pessoa

      pessoa.setDataNascimento(sf.parse(st.nextToken()));

                      // adicionando a pessoa (um registro lido) a
lista de resultados
                      listPessoas.add(pessoa);
                }

                // fechando o reader(arquivo texto)
                reader.close();

           } catch (IOException e) {
```

```
                throw new RuntimeException("Falha ao ler o arquivo
texto: "
                        + file.getAbsolutePath(), e);
            } catch (ParseException e) {
                throw new RuntimeException("Falha ao converter a
data: "
                        + file.getAbsolutePath(), e);
            }

            // retornando a lista com todos os registros presentes no
arquivo texto.
            return listPessoas;
      }

      @Override
      public Pessoa consultarPessoa(Pessoa parametro) {
            // obtendo a lista de todos os registros
            List<Pessoa> listPessoas = consultarTodasPessoas();

            // retornando o parâmetro da lista
            return listPessoas.get(parametro.getId().intValue());
      }

}
```

29

RESUMO

Nesse módulo definimos o padrão de projeto MVC (Model-View-Controler) tradicional. O padrão MVC é um padrão de arquitetura de *software* para implementação de interfaces com o usuário, que se utiliza desta combinação de outros padrões mais simples. Vimos, também, a adaptação do referido padrão para ambientes web, denominado de MVC2. O primeiro modelo (Model 1) era um modelo simplista, que utilizava páginas JSP sozinhas e isoladas (*standalone*) desvinculadas de qualquer outro componente, já o segundo modelo (Model 2) utilizava Servlet e JSP de modo integrado e combinado com divisão de responsabilidades clara e precisa.